COLLEGES BRETAGNE LOIRE | SCIENCES POUR L'INGENIEUR ET LE NUMERIQUE

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

DOCTORAL THESIS

Hardware Acceleration of Interval Arithmetic in the RISC-V and Application to Mobile Robotics

AUTHOR: Pierre Filiol

SUPERVISORS: Luc Jaulin
Jean-Christophe Le Lann
Théotime Bollengier

LABORATORY: Lab-STICC

DOCTORAL SCHOOL №648
TOPIC : Computer Science, Digital Electronics

Thesis defended:
Brest, April 28 2026

REVIEWERS:

Medhi Lhommeau: Professor,Laris Laboratory
Christophe Jego: Professor,IMS Laboratory

EXAMINATION COMMITTEE:

Members:	**Natalie Revol:**	Professor, LIP Laboratory
	Florent de Dinechin:	Professor, INSA Lyon
	Kevin Martin:	Professor, Laris Laboratory
	Luc Jaulin:	Professor, Ensta
	Jean-Christophe Le Lann:	Professor, Ensta
	Théotime Bollengier:	PhD, Ensta

COLLEGES BRETAGNE LOIRE | SCIENCES POUR L'INGENIEUR ET LE NUMERIQUE

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Title: Hardware Acceleration of Interval Arithmetic in the RISC-V and Application to Mobile Robotics

Keywords: Robotics, Interval Arithmetic, Hardware Acceleration, RISC-V, ISA Extension, FPGA

Abstract:

Many engineering problems can be formulated as systems of nonlinear equations or nonconvex optimization problems, for which computing reliable solution sets is challenging. Interval analysis provides a rigorous framework by representing variables as intervals and propagating constraints to compute guaranteed enclosures in the presence of uncertainties and numerical errors. In embedded robotics, interval methods are well suited to applications such as localization and robust control, but their practical deployment is limited by the performance overhead of software-based interval arithmetic, notably due to frequent rounding-mode changes and the lack of correctly rounded elementary functions on conventional processors.

This thesis proposes a hardware-oriented approach to interval arithmetic acceleration based on the RISC-V open instruction set architecture. A custom extension, named *xinterval*, is introduced to efficiently support interval operations tailored to embedded robotics. By encoding rounding modes at the instruction level and implementing dedicated hardware operators, the proposed solution overcomes key limitations of traditional floating-point units.In addition, this work refines interval contractor theory to correctly handle complementary contractors involving partially defined functions. The concepts of total contractors and an iota flag allow domain violations to be explicitly tracked, addressing known inaccuracies in existing interval libraries. The *xinterval* extension is implemented in VHDL and synthesized on FPGA. Experimental results on representative robotics benchmarks demonstrate speedups between 2.7 $\times$ and 7.8 $\times$ compared to a standard RISC-V platform, highlighting the effectiveness of hardware-accelerated interval arithmetic.

Titre: Accélération matérielle de l'arithmétique des intervalles et application à la robotique mobile.

Mots-Clés: Robotique, Arithmétique des Intervalles, Acceleration Matérielle, RISC-V, Jeu d'instruction, FPGA

Résumé:

Beaucoup de problèmes d'ingénierie sont modélisables par des systèmes d'équations non linéaires ou des problèmes de minimisation, dont la résolution peut s'avérer difficile. L'analyse par intervalles est une méthode numérique qui permet, par propagation de contraintes, de générer une estimation garantie d'un ensemble de solutions, même en présence d'incertitudes. En robotique mobile, les méthodes par intervalles sont très adaptées à des problèmes récurrents comme la localisation ou le contrôle robuste, mais leur utilisation embarquée est rendue difficile par les bibliothèques logicielles disponibles. La raison majeure réside dans la gestion que font les processeurs généralistes des arrondis et des fonctions élémentaires sur les nombres à virgule flottante.

Cette thèse propose une approche matérielle de l'arithmétique des intervalles, basée sur le développement d'une extension du jeu d'instructions de l'architecture RISC-V, nommée *xinterval*. Cette approche explore la possibilité de créer une architecture matérielle dédiée aux intervalles, adaptée à la robotique embarquée et qui ne souffre pas des mêmes limitations que les unités de calcul en virgule flottante des architectures traditionnelles. Par la même occasion, ces travaux illustrent une méthode pour générer efficacement des complémentaires de contracteurs dans le cas où la contrainte considérée repose sur des fonctions partiellement définies. L'extension *xinterval* est implémentée intégralement en VHDL et synthétisée sur FPGA. Les résultats expérimentaux montrent un facteur d'accélération allant de 2,7$\times$ à 7,8$\times$ sur des benchmarks d'applications robotiques représentatives, témoignant ainsi de l'intérêt de la méthode.

DOCTORAL THESIS

Hardware Acceleration of Interval Arithmetic in the RISC-V and Application to Mobile Robotics

AUTHOR: Pierre Filiol

SUPERVISORS: Luc Jaulin
Jean-Christophe Le Lann
Théotime Bollengier

LABORATORY: Lab-STICC

This page intentionally left blank.

Theory and practice sometimes clash.
And when that happens, theory loses.
Every single time.

Linus Torvalds

Acknowledgements

I would like to express my gratitude to my supervisors Luc Jaulin, Jean-Christophe Le Lann and Théotime Bollengier for their continuous guidance and support throughout the course of this PhD. Our discussions greatly influenced this work and benefited me on a technical level.

I would also like to thank the reviewers Medhi Lhommeau and Christophe Jego, for the time they have devoted to evaluating this manuscript, and to the members of the jury for agreeing to examine my work. I am equally grateful to the Lab-Sticc and Ensta for providing a supportive research environment as well as the SPIN doctoral school, in particular Patricia Cabel, for their administrative assistance.

Finally, my thoughts and gratitude are addressed to my parents, friends, and colleagues for their encouragement and support throughout all these years, which helped me maintain balance and motivation.

Abstract

Many engineering problems can be formulated as systems of nonlinear equations or non-convex optimization problems, for which computing reliable solution sets remains challenging. Interval analysis provides a rigorous numerical framework to address these issues by representing variables as intervals and propagating constraints to compute guaranteed enclosures of the solution set, even in the presence of uncertainties and numerical errors. In embedded robotics, interval methods are particularly relevant for applications such as localization, estimation, and robust control, where reliability and safety are critical. However, their practical deployment is limited by the performance overhead of software-based interval arithmetic, notably due to frequent floating-point rounding-mode changes and the lack of correctly rounded elementary functions on conventional processors.

This thesis explores a hardware-centered approach to interval arithmetic acceleration, leveraging the flexibility of the RISC-V open instruction set architecture. A custom extension, named *xinterval*, is proposed to efficiently support interval arithmetic operations tailored to embedded robotics. By encoding rounding modes at the instruction level and introducing dedicated hardware operators, the proposed solution overcomes the limitations of traditional floating-point units and enables efficient, concurrent computation of interval bounds.

Beyond architectural aspects, this work introduces a refinement of interval contractor theory to correctly handle complementary contractors involving partially defined functions. The proposed notion of total contractors, combined with an iota flag, allows domain violations to be explicitly tracked, addressing known inaccuracies in existing interval libraries for operators such as square root and logarithm.

The *xinterval* extension is specified, implemented in VHDL using floating-point primitives generated by the FloPoCo library, and synthesized on FPGA. A hardware-in-the-loop evaluation platform is developed to validate functionality and assess performance on representative robotics benchmarks. Experimental results show speedups ranging from 2.7× to 7.8× compared to a standard RISC-V platform, demonstrating the effectiveness of hardware-accelerated interval arithmetic for embedded robotics applications.

Contents

List of Figures ix

List of Tables 1

Acronyms 1

Introduction **3**

Context 3

Problem 3

Objectives of the Thesis 4

Contributions 5

Novelties 5

Comprehensive Listing of Publications 6

Organization of the Manuscript 8

I Theoretical Background **9**

1 Interval Methods **11**

1.1 Motivations 11

1.2 Interval Analysis Fundamentals 13

1.2.1 Intervals 13

1.2.2 Interval Computation 14

1.2.3 Interval Vectors 18

1.2.4 Interval Contractors 20
1.2.5 Subpavings 31
1.3 A Complete Localization Example 35
1.3.1 Modelling the Problem 35
1.3.2 Building The Contractors 36
1.3.3 Computing The Subpavings 39
1.4 Practical Implementation 41
1.4.1 An Overview of the IEEE-1788 Standard 41
1.4.2 The Software Ecosystem 44
1.4.3 The Limits of Software 46
1.5 Conclusion 48

2 Hardware Acceleration Strategies 51

2.1 Context 51
2.2 Graph-Level Acceleration 53
2.2.1 GPU Strategy 53
2.3 Contractor-Level Acceleration 64
2.3.1 Using SIMD Instructions 64
2.3.2 Using Custom Hardware 64
2.4 Conclusion 71

3 RISC-V Architecture 73

3.1 Motivations for the RISC-V Architecture 73
3.1.1 Early Processor Architectures 73
3.1.2 Transition from CISC to RISC 74
3.1.3 Limitations of Legacy ISAs 76
3.1.4 Origin and Objectives of the RISC-V 78
3.2 Fundamental Aspects of RISC-V Architecture 79
3.2.1 Core Architectural Design 79

3.2.2 The Modular Extension System 82
3.3 Conclusion 85

II Contributions 87

4 Accurate Generation of the Complementary of Contractor 89

4.1 Problem 90
4.2 Approach 91
4.3 Total extension 94
4.3.1 Definitions 94
4.3.2 Illustration 95
4.3.3 Properties 96
4.3.4 Total real arithmetic 97
4.3.5 Link with the complex number i 98
4.4 Total intervals 99
4.4.1 Intervals in unions of lattices 99
4.4.2 Total intervals 100
4.4.3 Total interval arithmetic 100
4.4.4 Total interval vector 101
4.5 Total contractors 101
4.5.1 Total directed contractor for a binary constraint 101
4.5.2 Total directed contractor for a ternary constraint 104
4.5.3 Total forward-backward contractor 107
4.6 Conclusion 110

5 A High-Level View of the Xinterval Extension 113

5.1 The Xinterval Custom Extension 113
5.2 Encoding Intervals in RISC-V's Registers 114
5.3 Basic Interval Computations 117

5.3.1 Creating Intervals From Single-Precision Bounds 117
5.3.2 Loading And Storing Intervals 119
5.3.3 Extracting Information From An Interval 119
5.3.4 Set Operations 120
5.4 Half-Contractors 123
5.4.1 Arithmetic Operators 123
5.4.2 Other Real Functions 126
5.5 Conclusion 129

6 The Micro-Architecture of the Xinterval Extension 131

6.1 The Software/Hardware Interface 132
6.1.1 From ISA Instructions to Hardware Circuits 132
6.1.2 The Minimal Xinterval Core 133
6.2 Review Of Hardware Primitives For Intervals 135
6.2.1 A Word About Floating-Point Arithmetic 135
6.2.2 Conversions Between Intervals and Floating-Points 136
6.2.3 Itv_set 141
6.2.4 Itv_add 143
6.2.5 Itv_mul 146
6.2.6 Itv_div 150
6.2.7 Itv_sqrt 154
6.2.8 Itv_exp 156
6.2.9 Itv_log 157
6.2.10 Itv_trig and Itv_trig_inv 159
6.3 Synthesis Results 168

7 Design, Validation and Benchmarking methodology for the Xinterval Extension 169

7.1 A Testbench Platform for the *Xinterval* Extension 170

7.1.1 Designing the Extension 170
7.1.2 Architecture of the Testbench Platform 172
7.2 In-depth Analysis of the *SoC* Simulator 173
7.2.1 Developping Applications for the SoC 174
7.2.2 The Hardware-In-The-Loop RISC-V ISA Simulation 177
7.2.3 Available *Xinterval* Wrappers for the RISC-V Core Simulator 182
7.3 Evaluating the Performances of the *Xinterval* Extension 189
7.3.1 Presentation of the Benchmarks 189
7.3.2 A Methodology for Fair Comparisons 190
7.3.3 Results Obtained 191

Conclusion **193**
Synthesis of the Contributions 193
Achievements 193
Limitations 194
Future Works 195

Bibliography 197

Appendices

A Top Level view ot the NVIDIA Hooper Architecture **215**

B Implementing Addition Contractors in *Libctc_no_acc* **217**

C Implementing Addition Contractors in *Libctc_acc* **221**

List of Figures

1.1 Overview of various set approximation techniques 12

1.2 Interval 14

1.3 Elementary functions applied to intervals 18

1.4 Interval vectors 19

1.5 Illustration of some contraction cases 22

1.6 Trivial HC4R 24

1.7 Interval functions square and square-root 26

1.8 Complex Constraint AST 30

1.9 Complex HC4R Forward-Backward 30

1.10 Subpaving obtained via SIVIA for $\mathcal{H}$ (1.36) 33

1.11 Sivia in depth 34

1.12 A range-only localization problem 36

1.13 HC4R for the C_1 38

1.14 SIVIA subpaving for one landmark 39

1.15 SIVIA subpaving for $\overleftrightarrow{C_{\mathcal{H}'}}$ 40

1.16 SIVIA subpaving for $\overleftrightarrow{C_{\mathcal{H}}}$ 40

1.17 IEEE-1788 Levels 42

1.18 IEEE-1788 Decorators For Set-Based Flavor 44

1.19 Interval Libraries Ecosystem By Release Date 45

1.20 Impact Of The Rounding-Mode Switching For x86 And Arm64 47

1.21 Software Libraries Chain of dependencies 49
1.22 Performance impact MPFR 49

2.1 Sequential SIVIA represented as a graph 52
2.2 Improvement rate of processors since 2000's 54
2.3 Codenames for recent NVIDIA GPU architectures 56
2.4 A comparison of CPU and GPU architectures 56
2.5 Content of a typical streaming multiprocessor 57
2.6 Computation primitives on NVIDIA A100 and H100 58
2.7 CUDA grid/block 60
2.8 An overview of Collange et al.'s implementation 61
2.9 Speed-up achieved with Collange et al.'s implementation 62
2.10 Parallel B&B algorithm in Bagóczki et al.'s work 62
2.11 Results obtained by Bagoczki et al.'s 63
2.12 Parallel B&B algorithm used by Zhang et al.'s 64
2.13 Anatomy of a FPGA Chip 65
2.14 The Traditional FPGA Design Workflow 67
2.15 IEEE-754 encoding For Single-Precision 68
2.16 IEEE-754 Rounding with Guard, Round and Sticky Bits 69
2.17 FloPoCo, a Parametrized Floating-Point Code Generator 71

3.1 Timeline of Notable Computer Designs 75
3.2 RISC-V General-Purpose Registers and Associated Calling Convention 79
3.3 Overview of the RV32I and Zicsr Extensions 80
3.4 Valid RISC-V Instruction Formats 81
3.5 RISC-V Opcode Space and Format Association 82
3.6 Nomenclature to Identify a RISC-V Processor 83
3.7 Main RISC-V Standard Extensions 84
3.8 RISC-V Floating-Point Registers and Associated Calling Convention 84

3.9 Instructions Supported by a RV32IMAF Core 85

4.1 Example of an Inaccurate Set Estimation in Codac 91

4.2 Contractor tree for $\mathbb{X}_1 \cup \mathbf{f}^{-1}(\mathbb{X}_2 \cap \mathbb{X}_3)$ 92

4.3 Contractor tree for the complementary of $\mathbb{X}_1 \cup \mathbf{f}^{-1}(\mathbb{X}_2 \cap \mathbb{X}_3)$ 94

4.4 A partial function f 95

4.5 Introduction of Not a Number ι 96

4.6 Total intervals are intervals of $\mathring{\mathbb{R}} = \mathbb{R} \cup \{\iota\}$ 100

4.7 AST for the constraint $y + \sqrt{x+y} \in [1,2]$ (left) and its complementary (right) 107

4.8 Paving obtained using the contractor for $\mathbb{S}$ and its complementary 108

4.9 AST for the constraint $\mathbf{f}(\mathbf{x}) \in \mathbb{Y}$ 109

4.10 Paving representing the solution set for Test-case 2 110

5.1 R-type instruction format in the RISC-V 114

5.2 NaN boxing of narrower floating-points in registers 115

5.3 Interval data-structure in *xinterval* 115

5.4 Comparison of the Range of Representable Numbers for Float31 and Float32 116

5.5 Instructions to create an interval (with or without iota) 117

5.6 Encoding of the Empty Interval in *xinterval* 118

5.7 Compilation of a simple *xinterval* code 119

5.8 Instructions to extract information from an interval 119

5.9 Instructions to Perform Set operations on Intervals 120

5.10 Behavior of Union and Intersection in **itv.set.u** and **itv.set.i** 122

5.11 Instructions for Half-Contractors Based on Arithmetic Operators 124

5.12 Instructions for Half-Contractors Based on Real Functions 126

5.13 Illustration of the Square Backward-Propagation 127

5.14 Illustration of the Cosine Backward-Propagation 129

5.15 Listing of *xinterval* instructions 130

6.1 R-type instruction format in the RISC-V 132
6.2 Encoding of two *xinterval* instructions 133
6.3 Common Hardware Interface For R-Type Instructions 133
6.4 The Minimal *xinterval* Core 135
6.5 Example Of a FloPoCo Operator Block 136
6.6 Top-level Entity For Itv_make 137
6.7 Datapath of Itv_make 138
6.8 Classification Of Float32 Bounds 139
6.9 Range Checks And Conversion Between Float32 and Float32 139
6.10 Top-level Entity For Itv_extract 140
6.11 Datapath of Itv_extract 141
6.12 Interval Fields Splitting 141
6.13 Top-level Entity For Itv_set 142
6.14 Datapath of Itv_set 143
6.15 Top-level Entity For Itv_add 144
6.16 Datapath of Itv_add 145
6.17 Top-level Entity For Itv_mul 146
6.18 Classification Process for Interval Input in Itv_mul 147
6.19 Datapath of Itv_mul 148
6.20 Regular Cases for Itv_mul 149
6.21 Regular Cases for Itv_sqr 149
6.22 Top-level Entity For Itv_div 150
6.23 Datapath of Itv_div 151
6.24 Classification Process for Interval Input itv_1 in Itv_div 152
6.25 Classification Process for Interval Input itv_2 in Itv_div 152
6.26 Regular Cases for Itv_div 153
6.27 Top-level Entity For Itv_sqrt 154
6.28 Datapath of Itv_sqrt 155

6.29 Top-level Entity For Itv_exp 156

6.30 Datapath of Itv_exp 157

6.31 Top-level Entity For Itv_log 157

6.32 Datapath of Itv_log 158

6.33 Top-level Entity For Itv_trig 159

6.34 Datapath of Itv_trig 160

6.35 Datapath of argument reduction for itv$_trig$ 161

6.36 Datapath of CORDIC evaluation for itv$_trig$ 162

6.37 Interval sine/cosine resolution when $q_{lb} \neq q_{ub}$ 163

6.38 Interval sine/cosine resolution when $q_{lb} = q_{ub}$ 164

6.39 Top-level Entity For Itv_trig_inv 164

6.40 Datapath of Itv_trig_inv 165

6.41 Graph of Arcsine/Arccosine Functions 166

6.42 Datapath of angle commputation using asin/acos 167

6.43 Synthesis of the Xinterval Core on Xilinx XC7Z020 FPGA @100MHz 168

7.1 Development Cycles for *Xinterval* 170

7.2 Top-Level View of the Architecture of the Testbench Platform 172

7.3 A Detailed View of the RISC-V-based *SoC* Simulation 173

7.4 Overview of the Two Flavors of the Implemented Contractor Library 174

7.5 Code Generation With GCC 176

7.6 A Schematic View of the ISA Simulator Deployed in the *SoC* 178

7.7 Architecture of the *xinterval core* 181

7.8 Communication Between the ISA Simulator and the *xinterval core* 181

7.9 Workflow Used in the Evolution of Wrapper Methodology for *Xinterval Core* . . 182

7.10 Interfacing the ISA Simulator with the *Xinterval Core* through GHDL cosimulation 183

7.11 Summary of the Exchanges Between the VHDL and C Contexts 184

7.12 Pros and Cons for the GHDL Wrapper 185

7.13 Interfacing the ISA Simulator with the *Xinterval Core* through the UART protocol186

7.14 Pros and Cons for the UART Wrapper 186

7.15 Interfacing the ISA Simulator with the *Xinterval Core* through the AXI protocol 187

7.16 Vivado Block Design for the Zynq-PL 188

7.17 Pros and Cons for the Zynq Wrapper 189

7.18 Pavings Obtained for the Benchmarks 192

A.1 Architecture of the NVIDIA H100 216

List of Tables

1.1 $\mathbb{IR}$ formats 13

1.2 Elementary contractors (arith op.) 25

1.3 Elementary contractors (real func.) 25

1.4 Expressions for the forward-backward contractors based on arithmetic operators 26

1.5 Expression of the elementary square contractor 28

2.1 Pre-Defined Binary Formats in IEEE-754 68

2.2 Encoding of Special-Values in IEEE-754 68

2.3 Some of the Rounding-Modes Defined by IEEE-754 69

5.1 Use of set operators in backward contractors 120

6.1 Relationship Between Hardware Circuits and *xinterval* ISA Instructions 132

6.2 Synthesis Results For The Main FloPoCo Operators 136

6.3 Synthesis Results For The Interval Operators 168

7.1 Proposition of a Latency Model for the ISA Simulator 180

7.2 Speedup Estimate Relative to a RISC-V Processor Without *Xinterval* 191

Introduction

Context

Many engineering problems essentially amount to solving systems of equations or finding the optimal parameter for a given cost function. These tasks become challenging as soon as nonlinear equations or nonconvex cost functions are involved, which is often the case in real-world applications. Interval analysis is a robust numerical method especially well suited to compute a guaranteed approximation of the solution set of such problems using subpavings. Its main feature is to rely on floating-point intervals instead of scalars to perform the computations, which allows to take into account physical uncertainties or compensate for the limited ability of computers to represent real numbers exactly.

Embedded robotics greatly benefits from this technique as some of its most recurring applications, such as localization or robust control, can be modeled as constraint satisfaction problems and solved efficiently with intervals. Interval contractors play a major role in the implementation of solving algorithms and are used to reduce the initial search space down to the feasible solutions through constraint propagation.

Starting in 2015, a normalization effort known as the IEEE-1788 standard specifies the expected behavior of operators involving intervals and provides guidelines for implementations. Nowadays, programmers have access to a large ecosystem of software libraries, compliant with the standard to varying degrees, to help with the development of interval applications.

In the aforementionned context, this thesis raises the question of finding an optimal way to execute interval-based robotics algorithms in an embedded environment and with moderate energy consumption.

Problem

Software libraries for interval arithmetic are notoriously difficult to optimize due to constraints imposed by the use of the underlying floating-point hardware. To ensure guar-

anteed calculations, mathematically correct operators need to evaluate lower and upper bounds of intervals using different IEEE-754 rounding modes. On mainstream architectures such as x86 or Arm, the rounding mode is set on a per-core fashion which causes the whole instruction pipeline to be flushed when a change needs to be performed. In a typical interval program, this happens very frequently and the impact on final performances is not to be overlooked. The change of rounding mode may also not be fully portable for older compilers as the C language provides no guarantee about it until the release of C99.

Using libm as a basis for interval-related computations is also discouraged, especially when searching for guarantees and high-precisions. This is due to the fact that IEEE-754 specifies nothing about the rounding of elementary functions, contrary to arithmetic operators and square-root, which leads most standard implementations to provide primitives which are not accurate down to the last ulp. This forces many existing interval libraries to rely on other math libraries such as Crlibm or softfloat libraries like MPFR which are designed specially for correct rounding.

However, these alternatives are designed with a focus on double-precision (or even higher for MPFR-based solutions) at the expense of performances, especially when using softfloats. This tradeoff does not feels like the most sensible choice for embedded robotics. Although the computational guarantees provided by the aforementionned libraries are very desirable, most applications would trade a bit of excess precision for greater speed as time is often a critical factor in robotics. It also feels wrong to be sabotaged by the hardware architecture and lose most of the performance though costly context switches of the rounding mode.

Objectives of the Thesis

The thesis advocates in favor of a novel approach which handles interval arithmetic directly in hardware, instead of relying on a software library built on top of the floating-point stack (whether hardfloat or softfloat). The main goal is to circumvent the floating-point hardware limitations mentionned previously and propose a solution tailored for the needs of embedded robotics. Ideally, the changes should be made transparent for the developpers and should not imply any new programming paradigm (e.g Cuda). The main reason is to facilitate the porting of existing applications and encourage the adoption by end users.

The emergence of the RISC-V standard, an open-source specification for processor architecture, is an unprecedented opportunity to achieve this goal. Its design philosophy makes it a good candidate for the hardware acceleration of interval arithmetic.

- Instructions are bundled in purpose-oriented packages called extensions. With

the notable exception of extension I, which contains the minimal set instructions required to make a viable processor, all other extensions are optional and can be instantiated only when the target application requires it.
- The standard requires the floating-point extensions F and D, respectively meant for computations on single and double precisions values, to define the operating rounding mode as part of the encoding of the instruction. The consequence is that changing the rounding mode has no impact on performances contrary to x86 and arm architectures.
- If standard extensions are not sufficient to achieve the desired performances on a specific application, the RISC-V standard offers a way to create a custom extension with dedicated primitives and extend the instruction set. This process is also greatly simplified from the software side with a dedicated GCC toolchain designed for extensibility.
- The simplicity of the microarchitecture allows processors to be shipped as soft-cores and synthesized on hardware devices such FPGA or ASICS. This makes the technology accessible for research purposes and enable design modifications.

The thesis takes advantage of the strengths of the RISC-V standard to design a custom extension called *xinterval* whose goal is to accelerate interval arithmetic to meet the specific requirements of embedded robotics. The proposed process to achieve this goal is as follows:

- Definition of a set of instructions consistent with the needs of robotics in terms of intervals, to the extent of the capabilities offered by the RISC-V interface.
- Implementation of the backend for *xinterval* instructions by designing an optimized hardware architecture for interval in VHDL. This approach allows to break free from the limitations imposed by traditional floating-point unit. Potential benefits include, among other things, the possibility to compute both bounds concurrently using different rounding modes, for a wide variety of interval operators.
- Development of a performance evaluation protocol to assess the speedup achieved by *xinterval* on a set of typical robotics applications.

Contributions

Novelties

First of all, the thesis takes advantage of the opportunity provided by the re-implementation of interval arithmetic in hardware to add novelties to the mathematical background. A new method is proposed to correctly generate the complementary of a contractor involving partially defined functions or operators. This solves a well known issue in popular interval contractor librairies where inaccurate contractors are produced for square-root

or logarithm constraints leading to erroneous estimation of the solution set. The proposed workaround presents the concept of total contractor and introduces the iota flag as a mean to track partial definition problems in forward-backward propagations.

The main contribution of the thesis lies in the design of the *xinterval* RISC-V custom extension. This attempt is documented through three publications with complementary objectives.

A first paper presents the chosen strategy, which focuses on accelerating elementary interval contractors to support the construction of more complex contractors using the forward-backward methodology. A list of instructions consistent with this approach is proposed as well as a discussion regarding the best way to integrate them into the interfaces offered by the RISC-V standard.

A second paper focuses on the actual implementation of the *xinterval* microarchitecture in VHDL, followed by a synthesis on FPGA. The proposed hardware operators rely on the floating-point primitives generated by the FloPoCo library. Particular attention is paid to the precise description of the datapath associated with each operator, followed by an analysis of the performance obtained in synthesis on FPGA.

The last paper presents the platform used to help with the development, validation and performance evaluation of the designed hardware. The strategy uses an hardware-in-the-loop approach in which the RISC-V standard ISA is simulated entierely in software and rely on the real hardware only when executing *xinterval* instructions. This allows the execution of high-level robotics applications at any stage of the design process while maintaining a high-degree of hardware representativity. To evaluate the gain in performances brought by *xinterval* compared to a standard RISC-V platform without accelerated intervals, a set of reference benchmarks composed of contractor applications of increasing complexities are executed for both configurations. In these conditions, *xinterval* achieves speedups comprised between 2.7 and 7.8 times depending of the application, with room for improvement.

Comprehensive Listing of Publications

Research Journal Papers

► Pierre Filiol, Theotime Bollengier, Luc Jaulin and Jean-Christophe Le Lann. "A New Interval Arithmetic To Generate the Complementary of Contractors". Acta Cybernetica 26 (2024) 817-838. DOI: `https://doi.org/10.14232/actacyb.300840`

► Pierre Filiol, Theotime Bollengier, Luc Jaulin and Jean-Christophe Le Lann. "RISC-V Based Hardware Acceleration of Interval Contractor Primitives in the Con-

text of Mobile Robotics". Acta Cybernetica 26 (2024) 889-912. DOI: https://doi.org/10.14232/actacyb.306636

International conferences with selection committee and proceedings

► Pierre Filiol, Theotime Bollengier, Luc Jaulin and Jean-Christophe Le Lann, "Efficient Hardware Primitives for Interval Contractors in Robotics and Integration to a Custom RISC-V ISA Extension," 2025 23rd IEEE Interregional NEWCAS Conference (NEWCAS), Paris, France, 2025, pp. 548-552, DOI: https://doi.org/10.1109/NewCAS64648.2025.11107029.

► Pierre Filiol, Luc Jaulin, Jean-Christophe Le Lann, Theotime Bollengier. "An Iterative Design and Validation Methodology for RISC-V Custom Extension", 2025, 32nd IEEE International Conference on Electronics, Circuits and Systems (ICECS'2025) Marrakech, Morocco. https://doi.org/10.1109/ICECS66544.2025.11270698

► Pierre Filiol, Luc Jaulin, Theotime Bollengier and Jean-Christophe LeLann. "Hardware accelerated interval arithmetic for mobile robotics using RISC-V ISA extension". Proceeding publishing pending (see https://uol.de/en/scan2025/proceedings) Slides on https://uol.de/f/2/dept/informatik/ag/dcis/SCAN2025/presentations/SCAN_2025_paper_35.pdf?v=1762273157

International conferences with selection committee without proceedings

► Pierre Filiol, Théotime Bollengier, Luc Jaulin, Jean-Christophe Le Lann. "A new type of intervals for solving problems involving partially defined functions". 13th Summer Workshop on Interval Methods (SWIM) 2022, Hannover, Germany. https://doi.org/10.15488/13203. Abstract available on https://repo.uni-hannover.de/items/1bbe295c-ea10-4d63-98d9-ce1dceb0e945

► Pierre Filiol. "Acceleration of contractor algebra on RISCV in the context of mobile robotic". 14th Summer Workshop on Interval Methods (SWIM) 2023, Angers, France. https://swim2023.sciencesconf.org/program?lang=en

► Pierre Filiol. "Hardware Acceleration of Interval Calculus on FPGAs". International Online Seminar on Interval Methods in Control Engineering. 10 Juin 2024. Lien youtube : https://www.youtube.com/watch?v=636gLI31iu4

Organization of the Manuscript

The document assumes no initial knowledge about the topics covered, such as robotics, interval arithmetic, hardware design on FPGAs, or RISC-V microarchitecture. All the concepts necessary for the understanding of the contributions are introduced in Part I of the document, in order to make the work accessible to a wide audience.

Chapter 1 presents set-based interval methods as found in embedded robotics, through its most relevant mathematical concepts such as contractor algebra, forward-backward propagation and estimation of the solution set. These tools are then applied to solve a classic localization problem involving the navigation in a field of landmarks with known position using only range measurements. The last part focuses on software implementations of interval arithmetic and addresses topics such as the IEEE-1788 standard, popular libraries and inherent challenges to low-level optimizations.

Chapter 2 focuses on relevant hardware acceleration strategies for interval arithmetic. It starts by a short literature survey about the gpu-based approach, by far the most popular, and shows how the branch-and-bound strategy used in Chapter 1 can be efficiently parallelized. It then introduces FPGA devices as a support for an alternative approach based on operator-level acceleration. The workflow and challenges associated with hardware design on this platform are described, particularly in terms of floating-point arithmetic which plays a crucial role in interval operators. In this context, the FloPoCo library, a VHDL code generator for floating-point operator primitives, is presented in anticipation for the upcoming contributions.

Chapter 3 is an introduction to the RISC-V standard which is the support used contributions to integrate the interval hardware operators. It starts with a review of the main design choices made by the microarchitecture and covers topics such as standard extensions, custom instructions and the software stack.

After the necessary reminders, Part II is entirely devoted to the contributions made in the context of the thesis.

Chapter 4 introduces a refinement in interval arithmetic to accurately generate the complementary of a contractor involving partially defined operators or real functions.

Chapter 5 describes the instructions provided by the *xinterval* custom extension and their integration in the RISC-V.

Chapter 6 discusses the implementation of the hardware operators responsible for the backend of *xinterval* instructions and records the performances achieved when synthesizing the design on FPGA.

Chapter 7 details the hardware-in-the-loop methodology used to design, validate and estimate the performances of the *xinterval* extension on a set of benchmark applications.

Part I

Theoretical Background

1

Interval Methods Applied To Robotics

Contents

1.1 Motivations **11**
1.2 Interval Analysis Fundamentals **13**
1.2.1 Intervals 13
1.2.2 Interval Computation 14
1.2.3 Interval Vectors 18
1.2.4 Interval Contractors 20
1.2.5 Subpavings 31
1.3 A Complete Localization Example **35**
1.3.1 Modelling the Problem 35
1.3.2 Building The Contractors 36
1.3.3 Computing The Subpavings 39
1.4 Practical Implementation **41**
1.4.1 An Overview of the IEEE-1788 Standard 41
1.4.2 The Software Ecosystem 44
1.4.3 The Limits of Software 46
1.5 Conclusion **48**

1.1 Motivations

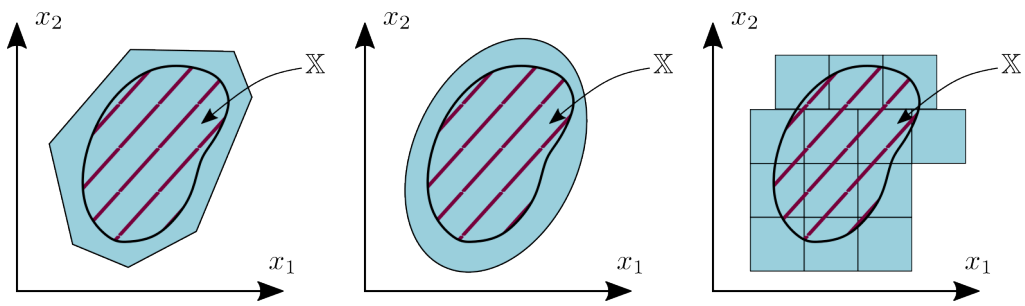
The ability to truthfully compute the solution of a set of equations or to find an optimal parameter to a particular cost function is of the utmost importance in many practical engineering problems. While being recurring, such tasks are generally deemed hard as

soon as non-linear equations or non-convex cost functions are involved. Unfortunately, the vast majority of real-world applications falls in this category and robotics, which is the primary target of the hereby PhD thesis, is no exception.

A common way to handle these difficult cases is to rewrite the problem as a *Constraint Network* (CN) or *Constraint Satisfaction Problem* (CSP), which is a triple (X, D, C) defined by:

- $X = \{x_1, x_2, ..., x_n\}$: a set of variables
- $D = \{\mathbb{X}_1, \mathbb{X}_2, ..., \mathbb{X}_n\}$: a set of domains (the allowed values for each x_i)
- $C = \{C_1, C_2, ..., C_m\}$: a set of constraints

The associated solution is the set made of the subsets of the $\mathbb{X}_i$ which are consistent with all the constraints. In order to be efficiently manipulated by a computer, the solution set is approximated using various techniques depicted in the literature, such as polyhedral enclosures [1], ellipsoids [2] or subpavings [3](Figure 6.43).



(**a**) *polyhedrals* (**b**) *ellipsoids* (**c**) *subpavings*

Figure 1.1: *Overview of various set approximation techniques*

Interval analysis is a numerical method used for computing guaranteed approximations of a solution set using subpavings. The technique was pioneered by Moore [4] in 1966 as an attempt to quantify the error on numerical computations by using intervals as operands instead of scalar numbers. This approach allows to take into account physical uncertainties such as measurement errors or compensate for the limited ability of computers to represent real numbers exactly. In recent history, some famous disasters were attributed to floating-point computation errors such as the Patriot missile failure from 1991 [5] or the explosion of the Ariane 5 rocket in 1996 [6]. As a consequence of the growing concern of engineering for guaranteed results, interval analysis has gained a lot of traction over the last 50 years with fundamental research works from Sunaga [7], Hansen [8], Alefeld [9] and Kulisch [10].

Nowadays, interval analysis is an efficient tool for the following classes of problems:

- Solving a set of possibly non-linear equations or inequalities [11].

- Solving ordinary and partial differential equations [12].
- Constrained optimization [13].
- Generic proofs for set-related properties [14].

In robotics, some recurring applications can be expressed as set-membership problems and handled using interval analysis:

- Localization algorithms [15].
- Path planning [16].
- Robust control [17].
- Collision detection [18].

This chapter presents the most fundamental interval analysis concepts and demonstrates their usage in the context of typical robotics applications.

1.2 Interval Analysis Fundamentals

1.2.1 Intervals

An interval $[x]$ is a connected subset of $\mathbb{R}$ defined by a lower bound $\underline{x}$ and an upper bound $\bar{x}$ (sometimes found as x_{lb} and x_{ub} in the literature) such as:

$$[x] = [\underline{x}, \bar{x}] \tag{1.1}$$

$$\underline{x} = sup\{a \in \mathbb{R} \cup \{-\infty, \infty\} \mid \forall x \in [x], a \leq x\} \tag{1.2}$$

$$\bar{x} = inf\{b \in \mathbb{R} \cup \{-\infty, \infty\} \mid \forall x \in [x], x \leq b\} \tag{1.3}$$

As a consequence, $\underline{x}$ is the largest number at the left of $[x]$ and $\bar{x}$ is the smallest number on its right. An interval is said to be *degenerate* and denoted as $\{x\}$ when its bounds are equal i.e. $\underline{x} = \bar{x}$, thus allowing to represent scalar values as intervals. The empty set $\varnothing$ is also introduced to represent the absence of solution to a problem.

In the context of this thesis, the term *interval* will be used interchangeably to refer to *closed intervals* of $\mathbb{R}$. Denote by $\mathbb{IR}$ the set of intervals falling in this category. Trivially, all the members of $\mathbb{IR}$ have one of the forms presented in Table 1.1.

[a, b]	[-∞, a]	[a, +∞]	[-∞, +∞]	$\varnothing$

Table 1.1: *Possible formats for members of* $\mathbb{IR}$ *with* $(a, b) \in \mathbb{R}^2$ *and* $a \leq b$

For convenience, the *width* of an interval $[x]$ and its *midpoint* are defined as follows:

$$w([x]) = \bar{x} - \underline{x} \tag{1.4}$$

$$mid([x]) = \frac{\underline{x} + \bar{x}}{2} \tag{1.5}$$

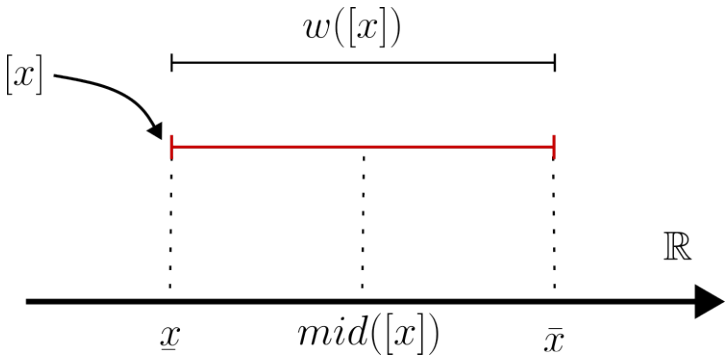


Figure 1.2: *A closed interval $[x]$ from IR*

Example

$[x_1] = [-0.5, 0.7]$	$w([x_1]) = 1.2$	$mid([x_1]) = 0.1$
$[x_2] = [1.6, +\infty]$	$w([x_2]) = +\infty$	$mid([x_2]) = +\infty$
$[x_3] = [-\infty, 4.5]$	$w([x_3]) = +\infty$	$mid([x_3]) = -\infty$
$[x_4] = [-\infty, \infty] = \mathbb{R}$	$w([x_4]) = +\infty$	$mid([x_4]) = 0$
$[x_5] = \{5.6\}$	$w([x_5]) = 0$	$mid([x_5]) = 5.6$
$[x_6] = \varnothing$	$w([x_6]) = -\infty$ **(convention)**	
$[x_7] = [1.1, 0.3]$	$\rightarrowtail$ **illegal: unordered bounds**	

1.2.2 Interval Computation

1.2.2.1 Set Operators

Traditional set operations, such as intersection and union, are defined for intervals and have the following behavior:

Intersection:

$$[x] \cap [y] = [max\{\underline{x}, \underline{y}\}, min\{\bar{x}, \bar{y}\}] \tag{1.6}$$

$$[x] \cap \varnothing = \varnothing \qquad \textit{(absorbing element)} \tag{1.7}$$

$$[x] \cap \mathbb{R} = [x] \qquad \textit{(neutral element)} \tag{1.8}$$

Example

$$[-5, 2] \cap [-1, 4] = [-1, 2]$$
$$[-5, 2] \cap [3, 6] = \varnothing$$

Union:

$$[x] \cup [y] = [min\{\underline{x}, \underline{y}\}, max\{\bar{x}, \bar{y}\}] \tag{1.9}$$

$$[x] \cup \varnothing = [x] \qquad \textit{(neutral element)} \tag{1.10}$$

$$[x] \cup \mathbb{R} = \mathbb{R} \qquad \textit{(absorbing element)} \tag{1.11}$$

Example

$$[4, 9] \cup [5, 11] = [4, 11] \qquad \textit{(connected operands)}$$
$$[-3, -1] \cup [1, 6] = [-3, 6] \qquad \textit{(disconnected operands)}$$

1.2.2.2 Binary Operators

Set operations and real arithmetic can also be applied to intervals. The result of any binary operator $\diamond$ applied between two intervals $[x]$ and $[y]$ from $\mathbb{IR}$ is defined according to (1.12) as illustrated by Moore [4].

$$[x] \diamond [y] = [\{x \diamond y \in \mathbb{R} \mid x \in [x], y \in [y]\}] \tag{1.12}$$

where $[.]$ is the *convex hull* which represents the smallest interval encompassing all the feasible values of $x \diamond y$ for $x \in [x]$ and $y \in [y]$.

This behavior can be illustrated on some recurring operators. Let $[x]$ and $[y]$ be two closed intervals from $\mathbb{IR}$:

Addition:

$$[x] + [y] = [\underline{x} + \underline{y}, \bar{x} + \bar{y}] \tag{1.13}$$

$$[x] + \varnothing = \varnothing \qquad \textit{(absorbing element)} \tag{1.14}$$

$$[x] + \mathbb{R} = \mathbb{R} \qquad \textit{(absorbing element)} \tag{1.15}$$

Example

$$[4, 9] + [-1, 6] = [3, 15]$$

Subtraction:

$$[x] - [y] = [\underline{x} - \bar{y}, \bar{x} - \underline{y}] \tag{1.16}$$

$$[x] - \varnothing = \varnothing \qquad \textit{(absorbing element)} \tag{1.17}$$

$$[x] - \mathbb{R} = \mathbb{R} \qquad \textit{(absorbing element)} \tag{1.18}$$

Example

$$[4, 9] - [-1, 6] = [4, 9] + [-6, 1] = [-2, 10]$$

Multiplication:

$$[x] \times [y] = [min\{\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y}\}, max\{\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y}\}] \tag{1.19}$$

$$[x] \times \varnothing = \varnothing \qquad \textit{(absorbing element)} \tag{1.20}$$

Example

$$[4, 9] \times [-1, 6] = [min(-4, 24, -9, 54), max(-4, 24, -9, 54)] = [-4, 54]$$

Division:

$$\frac{[x]}{[y]} = [x] \times \frac{1}{[y]} = \begin{cases} \varnothing & \textbf{if } [y] = \{0\}, \\ [x] \times [\frac{1}{\bar{y}}, \frac{1}{\underline{y}}] & \textbf{if } 0 \notin [y] \\ [x] \times [\frac{1}{\bar{y}}, +\infty] & \textbf{if } \underline{y} = 0 \text{ and } \bar{y} \geq 0 \\ [x] \times [-\infty, \frac{1}{\underline{y}}] & \textbf{if } \underline{y} < 0 \text{ and } \bar{y} = 0 \\ [-\infty, \infty] & \textbf{if } \underline{y} < 0 \text{ and } \bar{y} > 0 \end{cases} \tag{1.21}$$

Example

$$\frac{[4,9]}{[1,6]} = [4,9] \times [\frac{1}{6}, 1] = [min(\frac{2}{3}, 4, \frac{3}{2}, 9), max(\frac{2}{3}, 4, \frac{3}{2}, 9)] = [\frac{2}{3}, 9]$$

It must be noted that some algebraic laws are weakened when moving from reals to intervals:

- $[x]$ - $[x]$ is not always equal to {0}, ie $[x]$ - $[x] \ni 0$.
  $Example$: [0, 4] - [0, 4] = [0, 4] + [-4, 0] = [-4, 4] $\neq 0$
- $[z]([x]$ + $[y]) \subseteq$ [z][x]+[z][y] (**subdistributivity**)

As a consequence, the factorized form of an algebraic expression can be generally evaluated with less pessimism than its developped counterpart making it preferable in almost every cases.

1.2.2.3 Elementary functions

Elementary functions like exponential, logarithm or square-root are also applicable to intervals from $\mathbb{IR}$ by following a similar rule to arithmetic operators:

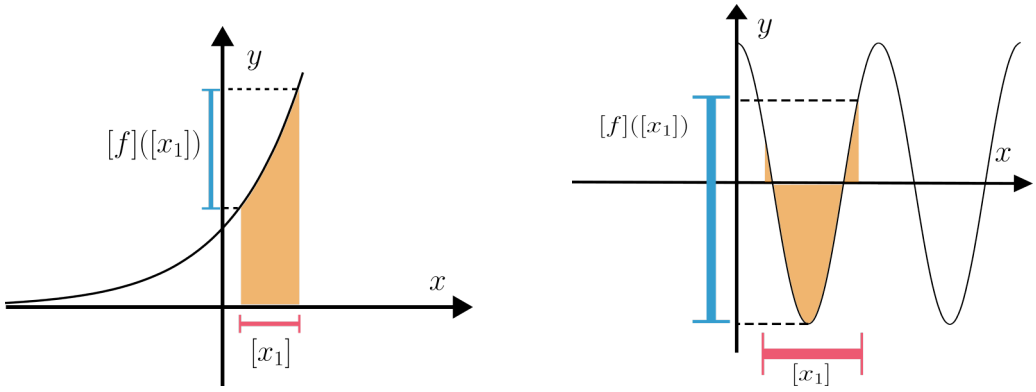
$$[f]([x]) = [\{f(x) \mid x \in [x]\}] \text{ where } f \text{ is a function from } \mathbb{R} \text{ to } \mathbb{R} \tag{1.22}$$

When *f* is continuous and strictly monotonous over its domain of definition (Figure 1.3a), the equation (1.22) can be rewritten as 1.23:

$$[f]([x]) = [min(f(\underline{x}), f(\bar{x})), max(f(\underline{x}), f(\bar{x}))] \tag{1.23}$$

For non-monotonous and periodic functions such as trigonometry operators (Figure 1.3b),

dedicated algorithms must be built to take into account the specificity of each function (Algorithm 1).



(a) *exponential* (b) *cosine*

Figure 1.3: *Elementary functions applied to intervals*

Example

$[\ln]([4,9]) = [\ln(4), ln(9)] \approx [1.3862, 2.1972]$	the logarithm is strictly monotonous
$[-6,9]^2 = [0,81]$	the square function is not strictly monotonous

Algorithm 1: An algorithm to compute $[\cos]([x])$

Data: $[x]$
Result: $[r] = [\cos]([x])$

```
r ← −1;
r̄ ← 1;
if width(x) ≥2π then
    return [r, r̄]
end
if ∄k ∈ ℤ | (2k − 1)π ∈ [x] then
    r ← min(cos(x), cos(x̄));
end
if ∄k ∈ ℤ | 2kπ ∈ [x] then
    r̄ ← max(cos(x), cos(x̄));
end
return [r, r̄]
```

1.2.3 Interval Vectors

An *interval vector* $[\mathbf{x}]$, or simply *interval box* is a Cartesian product of n intervals from $\mathbb{IR}$. The *i-th* component $[x_i]$ is the projection of $[\mathbf{x}]$ onto the *i-th* axis:

$$[\mathbf{x}] = [x_1] \times [x_2] \times ... \times [x_n] \tag{1.24}$$

It can be represented graphically using an axis-aligned box in n dimensions as depicted in Figure 1.4.

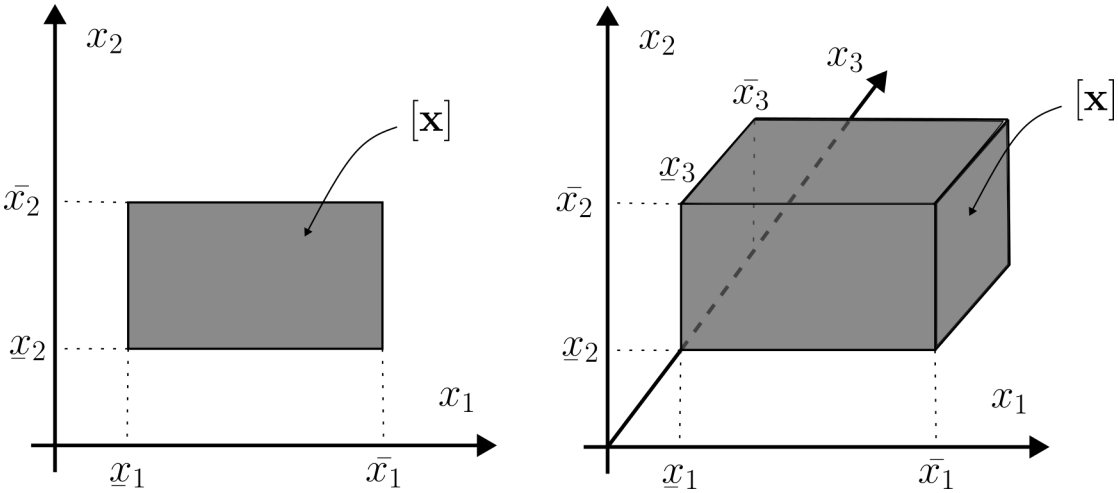


Figure 1.4: *Interval vectors of* $\mathbb{IR}^2$ *(left) and* $\mathbb{IR}^3$ *(right)*

The concept of width introduced previously for intervals is still applicable for boxes and is equal to the greatest width among all its interval components (1.25).

$$w([\mathbf{x}]) = max(w([x_1]), w([x_2]), ..., w([x_n])) \tag{1.25}$$

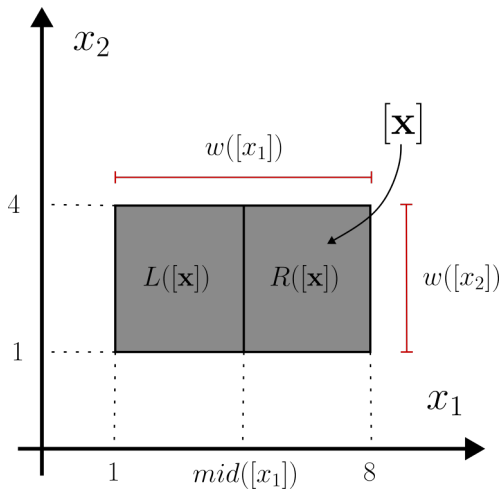
Another useful operation on boxes is the *bissection* denoted $bis([\mathbf{x}])$ which splits an input box into a left part ($L([\mathbf{x}])$) and a right part ($R([\mathbf{x}])$) with respect to its wider dimension. It is defined as follows:

$$\begin{aligned} &\textbf{if } j = min\{i \mid w([x_i]) = w([\mathbf{x}])\} \textbf{ then} \\ &L([\mathbf{x}]) = [x_1] \times ... \times [\underline{x}_j, mid([x_j])] \times ... \times [x_n] \\ &R([\mathbf{x}]) = [x_1] \times ... \times [mid([x_j]), \bar{x}_j] \times ... \times [x_n] \end{aligned} \tag{1.26}$$

Example

Let $[\mathbf{x}] = [x_1] \times [x_2] = [1, 8] \times [1, 4]$,

$w([\mathbf{x}]) = max(w[x_1], w[x_2]) = max(7, 3) = 7$
$mid([x_1]) = \frac{\underline{x_1}+\bar{x_1}}{2} = \frac{1+8}{2} = 4.5$
$bis([\mathbf{x}]) = (L([\mathbf{x}]), R([\mathbf{x}]))$
$L([\mathbf{x}]) = [\underline{x_1}, mid([x_1])] \times [x_2] = [1, 4.5] \times [1, 4]$
$R([\mathbf{x}]) = [mid([x_1]), x_1] \times [x_2] = [4.5, 8] \times [1, 4]$



1.2.4 Interval Contractors

1.2.4.1 Purpose

Consider the CSP $\mathcal{H}$ (refer to Section 1.1) defined below and its associated solution set $\mathbb{S}$.

$$\mathcal{H} : \begin{cases} X : (x_1, x_2, x_3) \\ D : (\mathbb{R}, \mathbb{R}, [0, +\infty]) \\ C : \mathbf{f}(X) = \mathbf{0} \end{cases} \quad \text{where} \quad \mathbf{f}(X) = \begin{cases} x_1 + \sin(x_2) + x_3 \\ x_1 - x_2 - 3 \end{cases} \tag{1.27}$$

The literature [19–21] shows how to compute a guaranteed enclosure of $\mathbb{S}$ through interval analysis using algorithms with polynomial complexity. Most of these methods rely on consistency techniques such as interval contractors [22]. Contracting $\mathcal{H}$ means replacing the domains D by a smaller domain D' without removing any valid solution for the constraints, ie $\mathbb{S} \subset D' \subset D$. The optimal solution is reached when D' becomes the smallest axis-aligned box which contains $\mathbb{S}$.

1.2.4.2 Definitions and Properties

The contractor for a set $\mathbb{X}$ of $\mathbb{R}^n$ is denoted $\mathcal{C}_{\mathbb{X}}$ and defined with respect to (1.28).

$$\mathcal{C}_{\mathbb{X}} : \begin{cases} \mathbb{IR}^n \to \mathbb{IR}^n \\ [\mathbf{x}] \to \mathcal{C}_{\mathbb{X}}([\mathbf{x}]) \end{cases} \tag{1.28}$$

It must satisfy the rules from (1.29):

$$\forall [\mathbf{x}] \in \mathbb{R}^n, \begin{cases} \mathcal{C}_{\mathbb{X}}([\mathbf{x}]) \subset [\mathbf{x}] & \textit{(Contractance)} \\ \mathcal{C}_{\mathbb{X}}([\mathbf{x}]) \cap \mathbb{X} = [\mathbf{x}] \cap \mathbb{X} & \textit{(Consistency)} \\ [\mathbf{x}] \subset [\mathbf{y}] \Rightarrow \mathcal{C}_{\mathbb{X}}([\mathbf{x}]) \subset \mathcal{C}_{\mathbb{X}}([\mathbf{y}]) & \textit{(Monotonicity)} \end{cases} \tag{1.29}$$

Remarks

From (1.29) , if $[\mathbf{x}']$ = $\mathcal{C}_{\mathbb{X}}([\mathbf{x}])$, then

- $[\mathbf{x}']$ is a subset of $[\mathbf{x}]$.
- Some non-consistent elements of $[\mathbf{x}]$ (sometimes all, sometimes none) with $\mathbb{X}$ has been discarded to form $[\mathbf{x}']$.
- No value consistent with $\mathbb{X}$ has been discarded.

The composition of contractors is defined by ${\mathcal{C}_{\mathbb{X}}}^{(n)}([\mathbf{x}])$ = $C_{\mathbb{X}} \circ C_{\mathbb{X}} \circ ... \circ C_{\mathbb{X}}([\mathbf{x}])$. When applying a Scott-continuous contractor several times on an input box $[\mathbf{x}]$, a point of stability $[\mathbf{x_s}]$ is generally reached for which no further contraction is possible (1.30). The number of iterations to reach this state depends of the nature of the contractor.

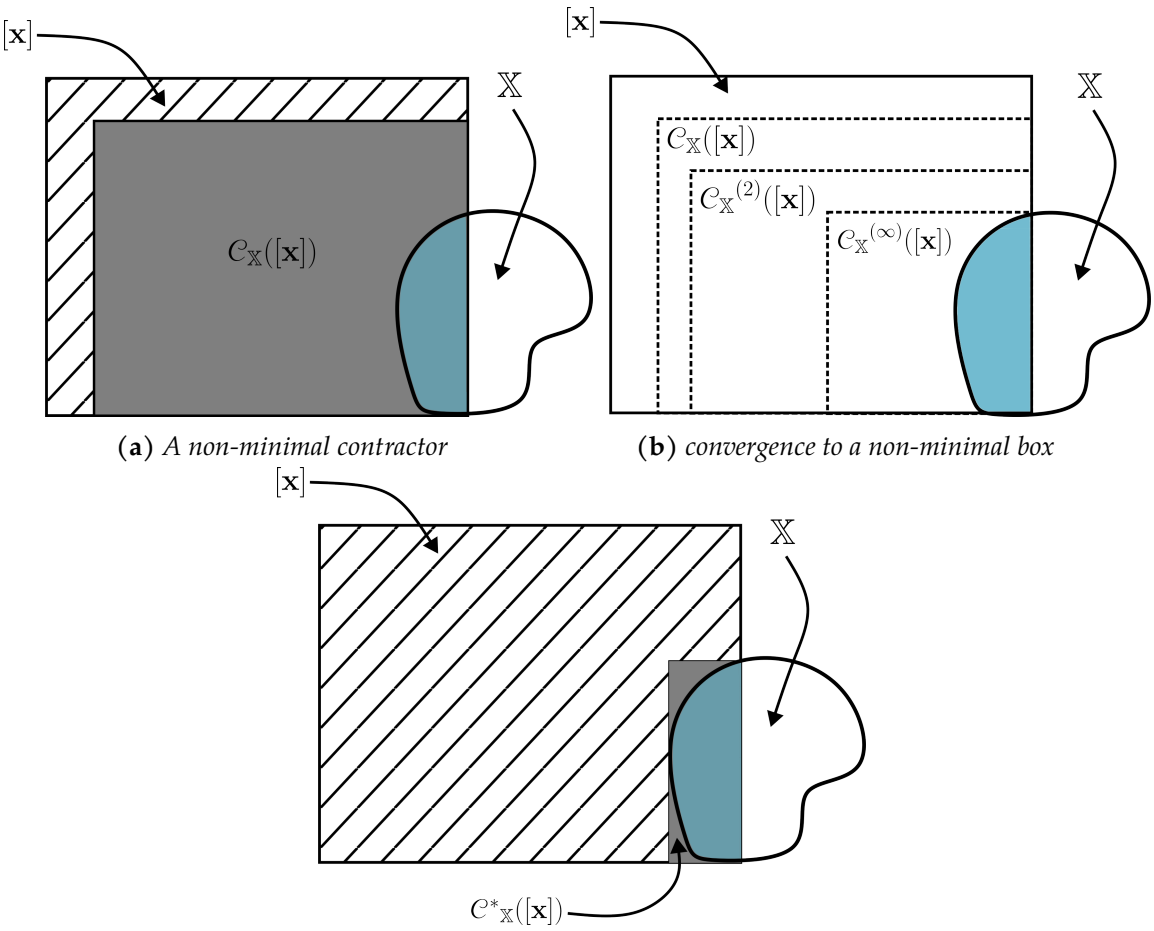
$$\begin{aligned} {\mathcal{C}_{\mathbb{X}}}^{(n)}([\mathbf{x}]) = [\mathbf{x_s}] \text{ when } n \to +\infty \\ \mathcal{C}_{\mathbb{X}}([\mathbf{x_s}]) = [\mathbf{x_s}] \end{aligned} \tag{1.30}$$

A contractor is said to be *minimal* when the resulting box is the smallest axis-aligned box containing $\mathbb{X}$. This is expressed by (1.31), where $[.]$ is still the convex hull. In this case, the contractor is denoted as ${\mathcal{C}^*}_{\mathbb{X}}$.

$$\forall [\mathbf{x}] \in \mathbb{IR}^n, {\mathcal{C}^*}_{\mathbb{X}}([\mathbf{x}]) = [[x] \cap \mathbb{X}] \tag{1.31}$$

The concepts presented beforehand are illustrated on Figure 1.5. On Figure 1.5a, the box $[\mathbf{x}]$ is contracted once using a non-minimal contractor and the hatched area of the input box is discarded. Figure 1.5b shows the convergence to a non-minimal box which occurs when the contractor is called repeatedly. Figure 1.5c presents a case of minimal contraction where the output box is the smallest axis-aligned box which encloses the set.

Finally, set operations such as intersection and union are also available for contractors [23]. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two contractors:



(**a**) *A non-minimal contractor* (**b**) *convergence to a non-minimal box*

(**c**) *The minimal contractor*

Figure 1.5: *Illustration of some contraction cases*

$$(\mathcal{C}_1 \cap \mathcal{C}_2)([\mathbf{x}]) = \mathcal{C}_1([\mathbf{x}]) \cap \mathcal{C}_2([\mathbf{x}]) \qquad \textit{(Intersection)} \tag{1.32}$$

$$(\mathcal{C}_1 \cup \mathcal{C}_2)([\mathbf{x}]) = \mathcal{C}_1([\mathbf{x}]) \cup \mathcal{C}_2([\mathbf{x}]) \qquad \textit{(Union)} \tag{1.33}$$

$$(\mathcal{C}_1 \circ \mathcal{C}_2)([\mathbf{x}]) = \mathcal{C}_1(\mathcal{C}_2([\mathbf{x}])) \qquad \textit{(Composition)} \tag{1.34}$$

This result is especially useful as it allows to build complex contractors from more simple ones. For example, solving $\mathcal{H}$ from (1.27) can be achieved by finding a contractor for each equation of $\mathbf{f}(X)$ and then using an intersection to build the contractor for the the whole problem.

1.2.4.3 Forward-Backward Contractors

Practically, the forward-backward methodology [24, 25] is often the preferred way to generate contractors for the domain of a CSP $\mathcal{H} : (\mathbf{f}(\mathbf{x}) \in [\mathbf{y}])$. This technique operates in two major steps and is based on constraint propagation techniques [26, 27]. The first

step (*forward step*) consists of a simple evaluation of $\mathbf{f}(\mathbf{x})$ using classic interval arithmetic with the rules presented in Section 1.2.2. In the second step (*backward step*), the result of this computation is intersected with $[\mathbf{y}]$ and retro-propagated toward the initial variables to reduce the input domain.

The operations used in the first and second steps are denoted as the *forward contractor* and *backward contractor* and respectively denoted as $\overrightarrow{\mathcal{C}}$ and $\overleftarrow{\mathcal{C}}$. The whole forward-backward contractor $\overleftrightarrow{\mathcal{C}}$ is then composed of this pair of contractors (1.35).

$$\overleftrightarrow{\mathcal{C}} = (\overrightarrow{\mathcal{C}}, \overleftarrow{\mathcal{C}}) \tag{1.35}$$

The example details this methodology for a simple addition, which is further illustrated on Figure 1.6 using the *Abstract Syntax Tree* (AST) of the associated constraint.

Example

Let $\mathcal{H}$ be the following CSP:

$$\mathcal{H} : \begin{cases} X : (x_1, x_2, x_3) \\ D : ([-\infty, 5], [-\infty, 4], [6, +\infty]) \\ C : \mathbf{f}(X) = \mathbf{0} \end{cases} \quad \text{where} \quad \mathbf{f}(X) = \{x_1 + x_2 - x_3\}$$

The constraint can be rewritten as a simple addition: $(x_1 + x_2 = x_3)$ and the associated forward-backward contractor is generated as follows:

<u>*Forward contractor*</u>:
The function is evaluated using interval arithmetics ...

$$[r_1] = [x_1] + [x_2] = [-\infty, 5] + [-\infty, 4] = [-\infty, 9]$$

... and intersected with $[x_3]$ to form a contracted $[x_3']$:

$$[x_3'] = [r_1] \cap [x_3] = [-\infty, 9] \cap [6, +\infty] = [6, 9]$$

$[x_3']$ now replaces $[x_3]$ in the domain of feasible values:

$$D \leftarrow ([x_1], [x_2], [x_3']) = ([-\infty, 5], [-\infty, 4], [6, 9])$$

Example (cont'd)

<u>*Backward contractor*</u>:
The new knowledge about $[x'_3]$ is retro-propapagated to contract $[x_1]$ and $[x_2]$ and form $[x'_1]$ and $[x'_2]$.

$$x_1 + x_2 = x'_3 \Leftrightarrow \begin{cases} x'_1 = (x'_3 - x_2) \cap x_1 = ([6,9] - [-\infty,4]) \cap [-\infty,5] = [2,5] \\ x'_2 = (x'_3 - x_1) \cap x_1 = ([6,9] - [-\infty,5]) \cap [-\infty,4] = [1,4] \end{cases}$$

In the end, the initial domain D has been contracted into D' which is much smaller:

$$D : \begin{cases} [x_1] : [-\infty,4] \\ [x_2] : [-\infty,5] \\ [x_3] : [6,+\infty] \end{cases} \quad \Longrightarrow \quad D' : \begin{cases} [x'_1] : [2,5] \\ [x'_2] : [1,4] \\ [x'_3] : [6,9] \end{cases}$$

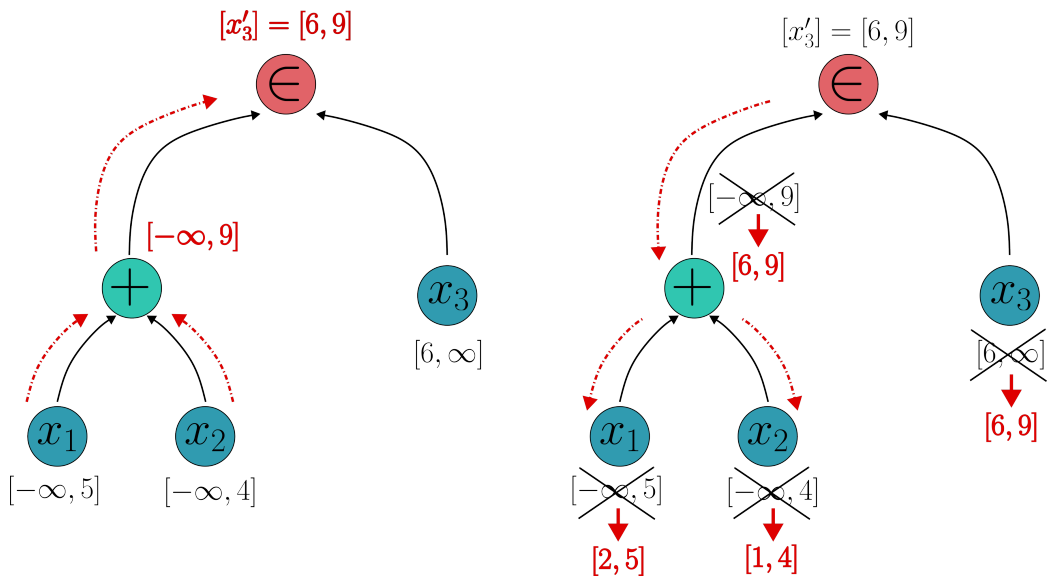


Figure 1.6: *Illustration of the forward (left) and backward (right) iterations. Red arrows indicate the flow of propagation.*

1.2.4.4 Minimal Contractors For Elementary Constraints

The forward-backward methodology illustrated in the previous example can be applied to generate *minimal* contractors for various elementary constraints. An elementary constraint is typically associated to an arithmetic operator or an algebraic function and has an expression which cannot be decomposed further.

> ***Example***
>
> The constraint $[x_3] = [x_1] + [x_2]$ is elementary.
>
> The constraint $[x_3] = [x_1] + \sqrt{[x_2]}$ is not elementary.
> It can be decomposed in two constraints: $[a_1] = \sqrt{[x_2]}$ and $[x_3] = [x_1] + [a_1]$

The forward and backward contractors for an operator $*$ associated to an elementary constraint are respectively denoted as $\overrightarrow{\mathcal{C}}_*$ and $\overleftarrow{\mathcal{C}}_*$. The elementary contractors listed in Tables 1.2 and 1.3 will be used frequently throughout the thesis and used as building blocks for complex contractors in combination with the technique presented in Section 1.2.4.5. Their expressions are well defined in the literature [22] but some of them will still be presented below as an example.

Arithmetic Operators		
$\overleftrightarrow{\mathcal{C}}_+$	[x] + [y] = [z]	Addition
$\overleftrightarrow{\mathcal{C}}_-$	[x] - [y] = [z]	Subtraction
$\overleftrightarrow{\mathcal{C}}_\times$	[x] * [y] = [z]	Multiplication
$\overleftrightarrow{\mathcal{C}}_/$	[x] / [y] = [z]	Division

Table 1.2: *Elementary contractors (arith op.)*

Real Functions		
$\overleftrightarrow{\mathcal{C}}_{sqr}$	$[x]^2 = [y]$	Square Function
$\overleftrightarrow{\mathcal{C}}_\sqrt{}$	$\sqrt{[x]}$ = [y]	Square Root
$\overleftrightarrow{\mathcal{C}}_{exp}$	$\exp([x])$ = [y]	Exponential
$\overleftrightarrow{\mathcal{C}}_{log}$	$\log([x])$ = [y]	Logarithm
$\overleftrightarrow{\mathcal{C}}_{cos}$	$\cos([x])$ = [y]	Cosine
$\overleftrightarrow{\mathcal{C}}_{sin}$	$\sin([x])$ = [y]	Sine

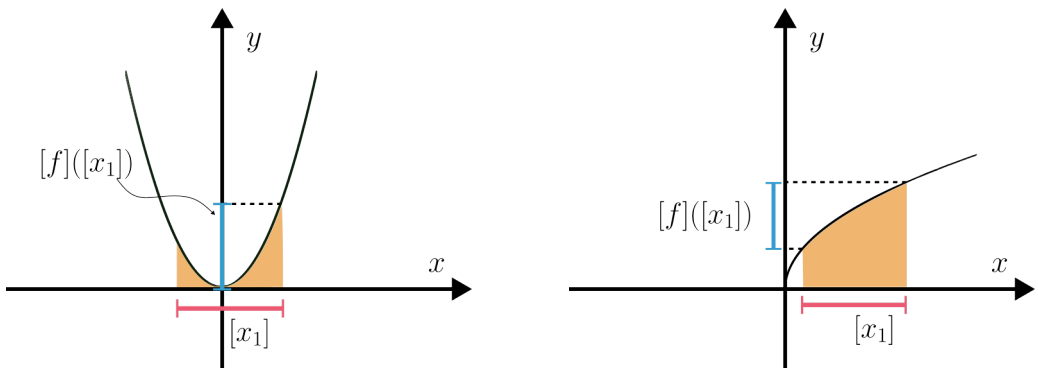
Table 1.3: *Elementary contractors (real func.)*

Arithmetic operators have trivial forward-backward contractors that are built in the same fashion as the previous example. Their expressions are listed in Table 1.4.

	Addition	Subtraction	Multiplication	Division
Constraint	$x_3 = x_1 + x_2$	$x_3 = x_1 - x_2$	$x_3 = x_1 \times x_2$	$x_3 = x_1 / x_2$
$\overrightarrow{\mathcal{C}_*}$	$[x_3] = [x_1] + [x_2]$	$[x_3] = [x_1] - [x_2]$	$[x_3] = [x_1] \times [x_2]$	$[x_3] = [x_1] / [x_2]$
$\overleftarrow{\mathcal{C}_*}$	$[x_1] = [x_3] - [x_2]$ $[x_2] = [x_3] - [x_1]$	$[x_1] = [x_3] + [x_2]$ $[x_2] = [x_1] - [x_3]$	$[x_1] = [x_3] / [x_2]$ $[x_2] = [x_3] / [x_1]$	$[x_1] = [x_3] \times [x_2]$ $[x_2] = [x_1] / [x_3]$

Table 1.4: *Expressions for the forward-backward contractors based on arithmetic operators*

As an example of a minimal contractor for real functions, consider the case of the square constraint $[x]^2 = [y]$. Figure 1.7 illustrates the interval function for both square and square-root. As it was previously discussed in Section 1.2.2.3, the algorithm to compute interval square-root is straightforward due to the strict monotonicity of the associated real function (Algorithm 3). However, this is not the case of square with has a change of monotonicity around 0 and thus requires a (slightly) more complicated procedure as shown in Algorithm 2.



(a) *square* (b) *square-root*

Figure 1.7: *Interval functions square and square-root*

Algorithm 2: Interval square

Data: $[x]$
Result: $[r] = [x]^2$
$\underline{r} \leftarrow \min(\underline{x}^2, \bar{x}^2)$
$\bar{r} \leftarrow \max(\underline{x}^2, \bar{x}^2)$
if $0 \in [x]$ **then**
  return $[0, \bar{r}]$
end
return $[\underline{r}, \bar{r}]$

Algorithm 3: Interval square-root

Data: $[x]$
Result: $[r] = \sqrt{[x]}$
if $\bar{x} \leq 0$ **then**
  return $\varnothing$
end
$\underline{r} \leftarrow \sqrt{\min(0, \underline{x})}$;
$\bar{r} \leftarrow \sqrt{\bar{x}}$;
return $[\underline{r}, \bar{r}]$

Example

Let $\mathcal{H}$ be the following CSP:

$$\mathcal{H} : \begin{cases} X : (x_1, x_2) \\ D : ([-1, 8], [13, 72]) \\ C : \mathbf{f}(X) = \mathbf{0} \end{cases} \qquad \text{where} \qquad \mathbf{f}(X) = \left\{ {x_1}^2 - x_2 \right.$$

The constraint can be rewritten as a simple addition: $({x_1}^2 = x_1)$ and the associated forward-backward contractor is generated as follows:

<u>*Forward contractor*</u>:
The function is evaluated using interval arithmetics ...

$$[r_1] = [x_1]^2 = [-1, 8]^2 = [0, 64]$$

... and intersected with $[x_2]$ to form a contracted $[x_2']$:

$$[x_2'] = [r_1] \cap [x_2] = [0, 64] \cap [13, 72] = [13, 64]$$

$[x_2']$ now replaces $[x_2]$ in the domain of feasible values:

$$D \leftarrow ([x_1], [x_2']) = ([-1, 8], [13, 64])$$

<u>*Backward contractor*</u>:
The new knowledge about $[x_2']$ is retro-propapagated to contract $[x_1]$
and form $[x_1']$.

$${x_1}^2 = x_2' \Leftrightarrow \left\{ x_1' = \sqrt{x_2'} \cap x_1 = (\sqrt{x_2'}) \cap [-1, 8] = [3.6055.., 8] \cap [-1, 8] = [3.6055.., 8] \right.$$

In the end, the initial domain D has been contracted into D' which is much smaller:

$$D : \begin{cases} [x_1] : [-1, 8] \\ [x_2] : [13, 72] \end{cases} \qquad \Longrightarrow \qquad D' : \begin{cases} [x_1'] : [3.6055.., 8] \\ [x_2'] : [13, 64] \end{cases}$$

The previous example shows how to build the elementary contractor corresponding to a square constraint. The associated expression is reported in Table 1.5.

	Square
Constraint	${x_1}^2 = x_2$
$\overrightarrow{\mathcal{C}_{sqr}}$	$[x_2] = [x_1]^2$
$\overleftarrow{\mathcal{C}_{sqr}}$	$[x_1] = \sqrt{[x_2]} \cap [x_1]$

Table 1.5: *Expression of the elementary square contractor*

1.2.4.5 Extension To Complex Contractors

The *HC4-Revise* (HC4R) algorithm [24, 25] is a generalization of the aforementionned forward-backward methodology and used to procedurally generate contractors for a constraint. It operates by traversing the AST of the mathematical expression associated to the constraint twice: the first traversal goes from leaves (variables) to root (top-intersection) and performs an evaluation of the result using interval aritmetic, the second one goes the opposite way and retro-propagates the new knowledge to the original variables. This approach has already been used in a much simpler context to construct the elementary contractor for addition (refer to Figure 1.6).

The main benefit of the algorithm is to work similarly with non-elementary (or complex) constraints, *i.e.*, that can be broken down into several elementary constraints (refer to Section 1.2.4.4). The only condition is that the associated mathematical expression must be exclusively made of continuous operators. An even more interesting result [24] is that the generated contractor is assured to be minimal if the expression satisfies the following additional requirements:

- Req$_1$: A minimal contractor is available for each elementary operator involved.
- Req$_2$: Each variable is present at most one time for a given degree.

Remarks

The need for the second requirement can be sensed easily.
If $[x] = [-2, 6]$, the value of $[x]^2$ can be computed using two different methods:

Through multiplication: $[x]^2 = [x] \times [x] = [-2, 6] \times [-2, 6] = [-12, 36]$
The multiplication formula from Equation (1.19) is applied.

Through square function: $[x]^2 = \text{sqr}([x]) = [-2, 6]^2 = [0, 36]$
The square function from Algorithm 2 is applied.

Remarks (cont'd)

The first method which involves the x variable two times at degree 1 generates more pessimism than the direct square function: $[x]^2 \subset [x] \times [x]$. This can be explained by the fact that Algorithm 2 uses an additional information compared to plain multiplication which is the equality of both operands.

The following example illustrates these requirements on several constraints:

Example

Consider the following constraints:

$C_1 : x_1 + \sin(x_2) + 2{x_1}^2$
$C_2 : e^{(x_1+x_2)} + \sin(x_2) + {x_1}^2$
$C_3 : \frac{\sin({x_1}^2+{x_2}^2)}{e^{x_1}+{x_2}^2}$

All the operators featured in these constraints have an associated minimal contractor presented in Tables 1.2 or 1.3: $\overleftrightarrow{\mathcal{C}}_+$, $\overleftrightarrow{\mathcal{C}}_\times$, $\overleftrightarrow{\mathcal{C}}_/$, $\overleftrightarrow{\mathcal{C}}_{sqr}$, $\overleftrightarrow{\mathcal{C}}_{\exp}$, $\overleftrightarrow{\mathcal{C}}_{\sin}$.

Based on the rules outlined above, only C_1 leads to a minimal contractor with HC4R: C_2 uses the variable x_2 twice at degree 1 and C_3 uses variable x_2 twice at degree 2.

As an illustration, HC4R will be applied on the constraint C_3 from the previous example to derive the associated contractor.

Let $\mathcal{H}$ be the following CSP:

$$\mathcal{H} : \begin{cases} X : (x_1, x_2) \\ D : ([-3.5, 2], [-3, 3]) \\ C : \mathbf{f}(X) \in [0.325, +\infty] \end{cases} \quad \text{where} \quad \mathbf{f}(X) = \left\{ \frac{\sin({x_1}^2+{x_2}^2)}{e^{x_1}+{x_2}^2} \right. \tag{1.36}$$

Figure 1.8 breaks down the mathematical expression of the constraint into an AST and creates some intermediate variables (boxed values) linked together by elementary constraints.

For the forward iteration, the strategy is close from the one used in the left part of Figure 1.6. The whole expression is evaluated using interval arithmetic by exploiting both the relationships between the intermediate variables and the forward elementary contractors presented in Section 1.2.4.4. The propagation flow is schematized on the left part of the Figure 1.9 and the associated algorithm is shown on Algorithm 4.

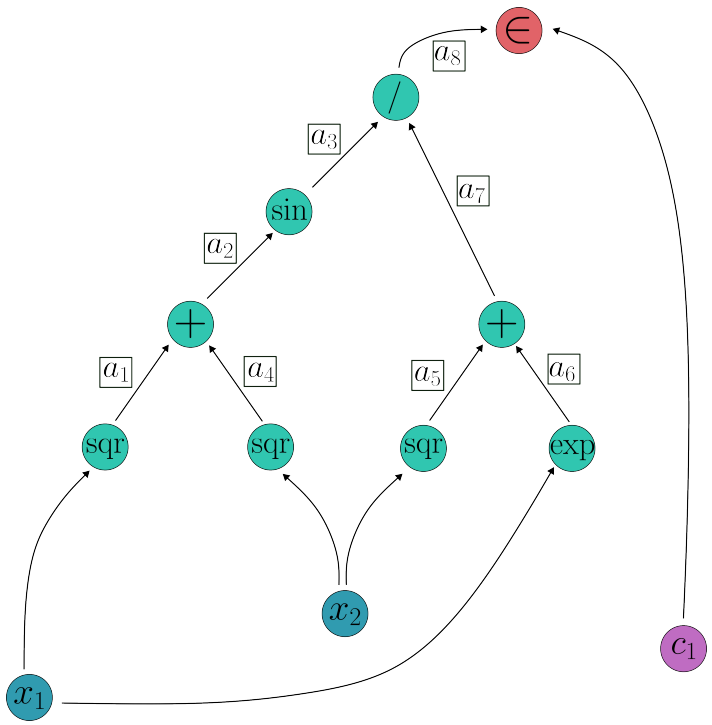


Figure 1.8: *Decomposition of a complex constraint into elementary parts using an AST*

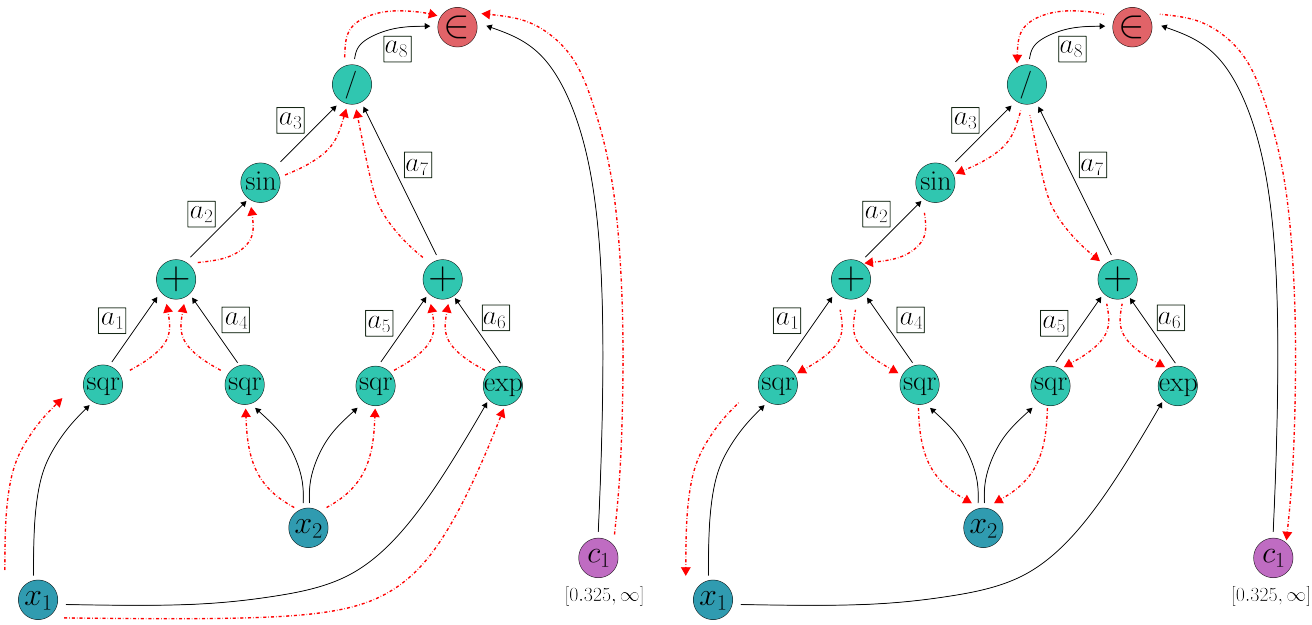


Figure 1.9: *Forward (left) and backward (right) propagation on the AST using HC4R*

For the backward iteration, the strategy is again similar to the one used in the right part of Figure 1.6. The knowledge obtained in the forward iteration is retro-propagated to the original variables by exploiting both the relationships between the intermediate variables and the backward elementary contractors presented in Section 1.2.4.4. The propagation flow is schematized on the right part of the Figure 1.9 and the associated algorithm is shown on Algorithm 5.

Algorithm 4: Contractor $\overrightarrow{\mathcal{C}_{\mathcal{H}}}$

Data: $[x_1]$,$[x_2]$

$[a_1] = \overrightarrow{\mathcal{C}_{sqr}}([x_1])$;
$[a_4] = \overrightarrow{\mathcal{C}_{sqr}}([x_2])$;
$[a_5] = \overrightarrow{\mathcal{C}_{sqr}}([x_2])$;
$[a_6] = \overrightarrow{\mathcal{C}_{exp}}([x_1])$;
$[a_2] = \overrightarrow{\mathcal{C}_{+}}([a_1, a_4])$;
$[a_3] = \overrightarrow{\mathcal{C}_{sin}}([a_2])$;
$[a_7] = \overrightarrow{\mathcal{C}_{+}}([a_5, a_6])$;
$[a_8] = \overrightarrow{\mathcal{C}_{/}}([a_3, a_7])$;
$[a_8] = [a_8] \cap [c_1]$;
return $[x_1], [x_2], [a_{1-8}]$

Algorithm 5: Contractor $\overleftarrow{\mathcal{C}_{\mathcal{H}}}$

Data: $[x_1], [x_2], [a_{1-8}]$

$[a_3], [a_7] = \overleftarrow{\mathcal{C}_{/}}([a_3], [a_7], [a_8])$;
$[a_2] = \overleftarrow{\mathcal{C}_{sin}}([a_2], [a_3])$;
$[a_1], [a_4] = \overleftarrow{\mathcal{C}_{+}}([a_1], [a_4], [a_2])$;
$[x_1] = \overleftarrow{\mathcal{C}_{sqr}}([x_1], [a_1])$;
$[x_2] = \overleftarrow{\mathcal{C}_{sqr}}([x_2], [a_4])$;
$[a_5], [a_6] = \overleftarrow{\mathcal{C}_{+}}([a_5], [a_6], [a_7])$;
$[x_1] = \overleftarrow{\mathcal{C}_{exp}}([x_1], [a_6])$;
$[x_2] = \overleftarrow{\mathcal{C}_{sqr}}([x_2], [a_5])$;
return $[x_1]$,$[x_2]$

1.2.5 Subpavings

Let $\mathbf{f}$ be a function from $\mathbb{R}^n$ to $\mathbb{R}^m$ (possibily non-linear) and $\mathbb{Y}$ a subset of $\mathbb{R}^m$. A set inversion problem consists in the characterization of

$$\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{f}(\mathbf{x}) \in \mathbb{Y}\} = \mathbf{f}^{-1}(\mathbb{Y}) \tag{1.37}$$

The *Set Inverter Via Interval Analysis* (SIVIA) algorithm [28] can be used to build two regular subpavings $\underline{\mathbb{X}}$ and $\bar{\mathbb{X}}$ respectively called inner and outer approximations such as $\underline{\mathbb{X}} \subset \mathbb{X} \subset \bar{\mathbb{X}}$. This procedure (Algorithm 6) relies on the forward-backward contractor $\overleftrightarrow{C}$ associated with the constraint $\mathbf{f}(\mathbf{x}) \in \mathbb{Y}$, whose generation method was thoroughly described in section 1.2.4. The main steps of SIVIA are described below and in Figure 1.11:

Input values:

- $[\mathbf{x_0}]$: a box from $\mathbb{R}^n$ corresponding to the initial domain of the underlying CSP.
- $\overleftrightarrow{C}$: the forward-backward contractor associated with the constraint $\mathbf{f}(\mathbf{x}) \in \mathbb{Y}$.
- ϵ: the minimum width for any box in resulting inner/outer approximations (precision control).

Output values:

- $\mathcal{L}_{in}$: the list of all the boxes from the inner approximation of $\mathbb{X}$.

Algorithm 6: The SIVIA algorithm

```
Data: [x_0], ↔C, ϵ
L ← list([x_0]);
L_in ← L_boundary = list();
while (L not empty) do
    Pop [x] from L;
    [x'] ← ↔C([x]);
    if [x'] = ∅ then
        Go to next iteration;
    else if →C([x]) ⊂ 𝕐 then
        Append [x] to L_in;
        Go to next iteration;
    else if w([x]) > ϵ then
        L([x]), R([x]) ←bis([x])
        Append L([x]), R([x]) to L;
        Go to next iteration;
    else
        Append [x] to L_boundary;
        Go to next iteration;
    end
end
return L_in, L_boundary
```

- $\mathcal{L}_{boundary}$: the list of all the boxes from the inner approximation of $\mathbb{X}$.

Initialization:
Three lists are initialized at the beginning of the algorithm. $\mathcal{L}_{in}$ and $\mathcal{L}_{boundary}$ are originally empty and grow during the execution. $\mathcal{L}$ contains the pool of boxes which are awaiting for an evaluation through the forward-backward contractor $\overleftrightarrow{C}$. This list starts with only $[\mathbf{x_0}]$ but is populated during the execution with the results of the multiple bisections.

Main loop:
The main loop runs until $\mathcal{L}$ contains no more boxes. Each iteration pops a box $[\mathbf{x}]$ from $\mathcal{L}$ which is contracted using $\overleftrightarrow{C}$ to obtain $[\mathbf{x}']$. Several things may then happen:

- $[\mathbf{x}']$ is equal to $\varnothing$: this means that $[\mathbf{x}] \cap \mathbb{X} = \varnothing$ and $[\mathbf{x}] \not\subset \bar{\mathbb{X}}$.
  $\Rightarrow$ Discard $[\mathbf{x}]$ and go to the next iteration.
- $\overrightarrow{C}([\mathbf{x}]) \subset \mathbb{Y}$: the result of the forward contractor $\overrightarrow{C}$ is fully included in $\mathbb{Y}$ and thus $[\mathbf{x}] \subset \underline{\mathbb{X}}$.
  $\Rightarrow$ Append $[\mathbf{x}]$ to $\mathcal{L}_{in}$ and go to the next iteration.
- $\overrightarrow{C}([\mathbf{x}]) \not\subset \mathbb{Y}$ and $w([\mathbf{x}]) > \epsilon$: the result of the forward contractor $\overrightarrow{C}$ is partly included in $\mathbb{Y}$ which means that a subset of $[\mathbf{x}]$ belongs to $\mathbb{X}$. As the output box is deemed wide enough (ϵ criteria), $[\mathbf{x}]$ can be bisected according to (1.26) for each halves to be evaluated independantly by the contractor.

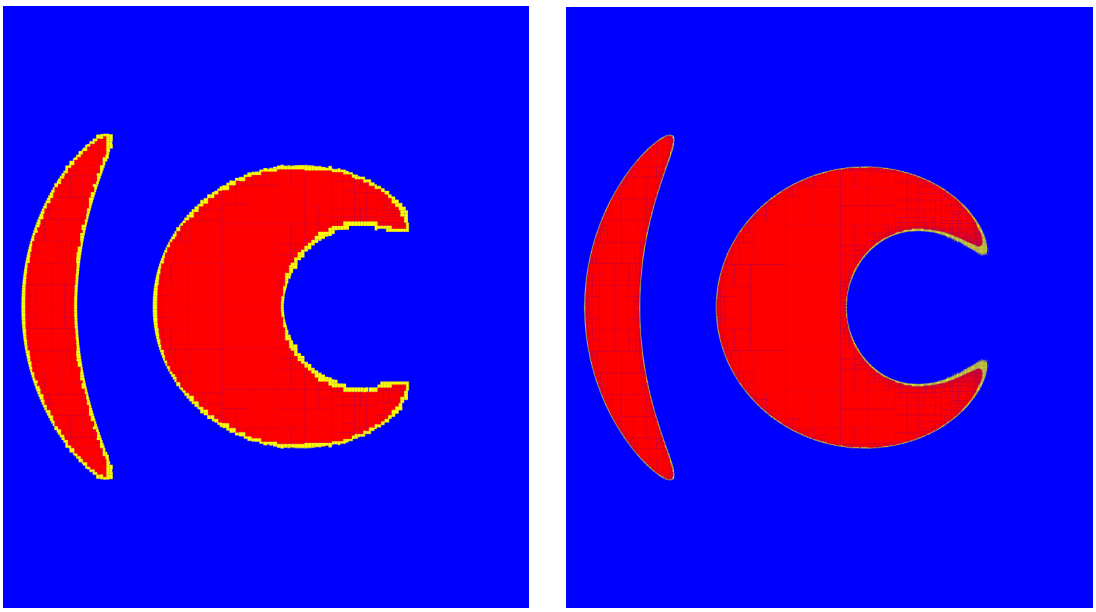
$\Rightarrow$ Push L([**x**])/R([**x**]) in $\mathcal{L}$ and go to the next iteration.

- $\overrightarrow{C}([\mathbf{x}]) \not\subset \mathbb{Y}$ and $w([\mathbf{x}]) \leq \epsilon$: Similarly to the previous case, a subset of $[\mathbf{x}]$ belongs to $\mathbb{X}$. However, the output box is too narrow for further bisection and is directly added to $\bar{\mathbb{X}}$.

  $\Rightarrow$ Append $[\mathbf{x}]$ to $\mathcal{L}_{boundary}$ and go to the next iteration.

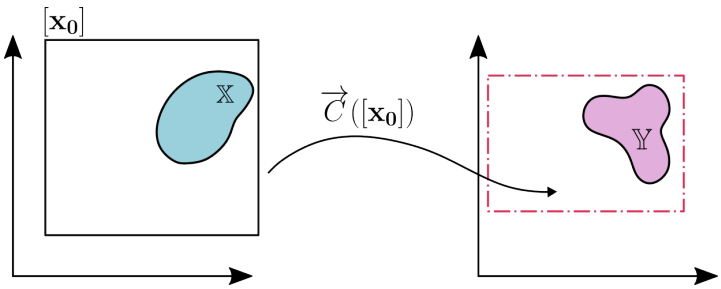
Termination:
At the end of the algorithm, $\mathcal{L}$ is empty and $\mathcal{L}_{in}$ and $\mathcal{L}_{boundary}$ contain respectively all the boxes from $\underline{\mathbb{X}}$ (inner approximation) and $\bar{\mathbb{X}}$ (outer approximation).

As a matter of example, the SIVIA algorithm can be used to generate the paving associated to the CSP $\mathcal{H}$ defined in (1.36) whose associated contractor has been generated using HC4R (Algorithms 4 and 5). By adjusting the value of ϵ, the granularity of the output paving can be controlled to match the requirements of the underlying application. For example, a smaller ϵ generates a more precised paving composed of many smaller boxes which is beneficial to accuracy but increases the computation resources required to manipulate the data.

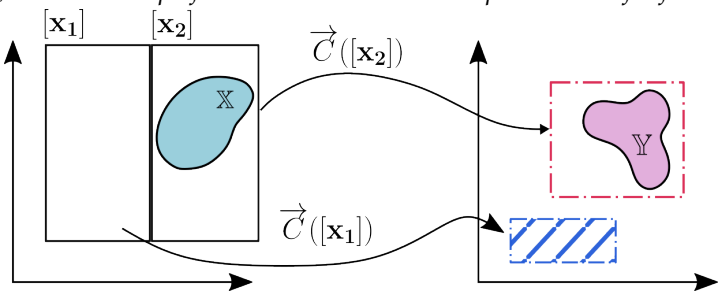


(**a**) *Paving with* $\epsilon = 0.1$ (**b**) *Paving with* $\epsilon = 0.01$

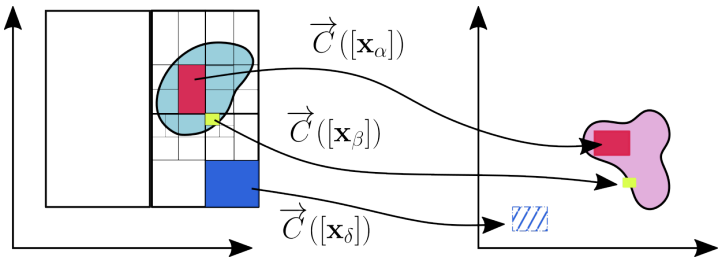
Figure 1.10: *Subpaving obtained via SIVIA for* $\mathcal{H}$ *(1.36)*



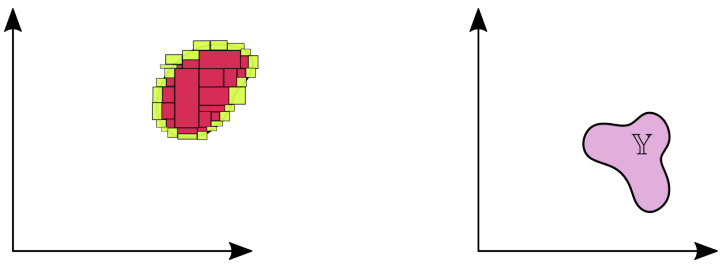
(a) $\overrightarrow{C}([\mathbf{x_0}])$ *contains the whole set* $\mathbb{Y}$ *which implies that a subset of* $[\mathbf{x_0}]$ *belongs to* $\mathbb{X}$*. As the contracted box is wide enough, a bisection is performed and both halves are pushed to* $\mathcal{L}$ *for further inspection.*



(b) $\overrightarrow{C}([\mathbf{x_1}])$ *(blue box) has not intersection with* $\mathbb{Y}$ *which means that* $[\mathbf{x_1}] \cap \mathbb{X} = \varnothing$*. As a consequence it is useless to explore* $[\mathbf{x_1}]$ *further and the box is discarded. However,* $\overrightarrow{C}([\mathbf{x_2}])$ *(red box) still fully encompasses* $\mathbb{Y}$ *so another cycle of bisection is scheduled.*



(c) *The algorithm goes on with bisections to explore all the promising paths. For a given box* $[\mathbf{x}]$*, if* $\overrightarrow{C}([\mathbf{x}]) \subset \mathbb{Y}$ *(red box), then* $[\mathbf{x}] \subset \mathbb{X}$ *and the box is not bisected further and directly added to* $\mathcal{L}_{in}$*. If* $\overrightarrow{C}([\mathbf{x}])$ *is only partially included in* $\mathbb{Y}$ *and* $w([\mathbf{x}]) < \epsilon$ *(yellow box) then* $[\mathbf{x}]$ *is considered to be a part of the outer approximation and thus added to* $\mathcal{L}_{boundary}$*.*



(d) *At the end of the execution,* $\mathbb{X}$ *is fully characterized by a list of red boxes (the inner approximation) and yellow boxes (outer approximation).*

Figure 1.11: *A graphic illustration of the SIVIA algorithm*

1.3 A Complete Localization Example

The goal of this section is to demonstrate the use of interval methods in the context of real-world applications. As stated in the introduction of this chapter, a lot of recurring tasks encountered in robotics can be identified as set-membership problems and solved efficiently with intervals using the tools presented in Section 1.2. A perfect example is the autonomous localization in a partially known environment, a classic data-fusion [29] problem where a robot attempts to estimate its position using a set of possibly heterogenous sensors. This application is challenging due to the fact that the knowledge of the environment is often inaccurate or outdated and that the sensor measurements come with uncertainties and are generally error-prone. Some well-established techniques such as the Kalman filtering [30, 31] are successfully used in these situations but they have the drawback of requiring a precise estimation of the initial conditions (orientation, position, ...) which can be a lot harder than the actual localization [32]. Interval methods make it possible to solve the problem even when the initial conditions are not precisely known and absorb most of the uncertainties imposed by sensor measurements.

1.3.1 Modelling the Problem

Figure 1.12 illustrates a situation where a robot (green triangle) attempts to localize itself using three landmarks (red circles). The robot starts at an unknown position (x_r, y_r) and uses its internal sensors (camera, telemetry, sonar, ...) to compute the distances d_i toward landmark i with $i \in \{1, 2, 3\}$. These distances are represented as intervals to account for the measurement errors. The coordinates of each landmark are exactly known prior to the execution and are denoted by a_i with $i \in \{1, 2, 3\}$.

Each distance d_i to landmark a_i produced by the sensors indicates that the robot is somewhere in the ring centered around a_i with an inner radius of d_i, an outer radius of d_i and a thickness equal to $w([d_i])$. This means that a couple of coordinates (x_r, y_r) from $\mathbb{R}^2$ is consistent with the measurements if the constraint (1.38) is satisfied.

$$(x_r - x_{a_i})^2 + (y_r - y_{a_i})^2 \in {d_i}^2 \text{ (disk equation)} \tag{1.38}$$

As a consequence, the whole problem is equivalent to solving the following CSP where each landmark is given a constraint similar to (1.38):

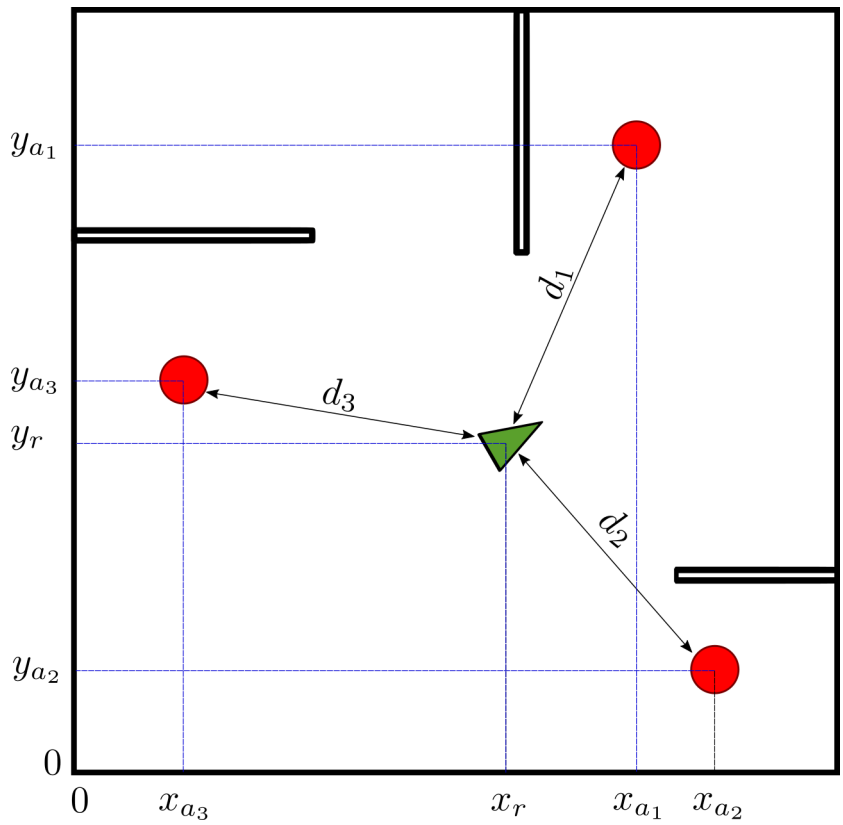


Figure 1.12: *Localization of a robot using range measurements and known landmarks.*

$$\mathcal{H} : \begin{cases} X : (x, y) \\ D : ([0, \infty], [0, \infty]) \\ \end{cases} \quad \text{where} \quad \mathbf{f}(X) = \begin{cases} (x - x_{a_1})^2 + (y - y_{a_1})^2 \\ (x - x_{a_2})^2 + (y - y_{a_2})^2 \\ (x - x_{a_3})^2 + (y - y_{a_3})^2 \end{cases} \tag{1.39}$$

1.3.2 Building The Contractors

At first, the constraints from $\mathcal{H}$ are handled independently:

$$\begin{aligned} C_1 &: (x - x_{a_1})^2 + (y - y_{a_1})^2 \in [d_1]^2 && \textit{(Landmark 1)} \\ C_2 &: (x - x_{a_2})^2 + (y - y_{a_2})^2 \in [d_2]^2 && \textit{(Landmark 2)} \\ C_3 &: (x - x_{a_3})^2 + (y - y_{a_3})^2 \in [d_3]^2 && \textit{(Landmark 3)} \end{aligned}$$

The HC4R algorithm can be applied on each constraint to generate the associated forward-backward contractor. Since the conditions presented in Section 1.2.4.5 are satisfied, the resulting contractor is assured to be minimal.

As a matter of example, Figure 1.13 illustrates the constraint propagation of HC4R on the AST of C_1, which leads to $\overrightarrow{C_1}$ (Algorithm 7) and $\overleftarrow{C_1}$ (Algorithm 8). The process is naturally similar for C_2 and C_3.

Algorithm 7: Contractor $\overrightarrow{C_1}$

Data: $[x]$, $[y]$

1 $[a_1] = \overrightarrow{C_-}([x], [x_{a_1}])$;
2 $[a_2] = \overrightarrow{C_-}([y], [y_{a_1}])$;
3 $[a_3] = \overrightarrow{C_{sqr}}([d_1])$;
4 $[a_4] = \overrightarrow{C_{sqr}}([a_1])$;
5 $[a_5] = \overrightarrow{C_{sqr}}([a_2])$;
6 $[a_6] = \overrightarrow{C_+}([a_6], [a_3])$;
7 $inc = 1$ **if** $([a_6] \in [a_3])$ **else** 0;
8 $[a_6] = [a_6] \cap [a_3]$;
9 **return** $[x], [y], [a_{1-6}], inc$

Algorithm 8: Contractor $\overleftarrow{C_1}$

Data: $[x], [y], [a_{1-6}]$

1 $[a_4], [a_5] = \overleftarrow{C_+}([a_4], [a_5], [a_6])$;
2 $[a_1] = \overleftarrow{C_{sqr}}([a_1], [a_4])$;
3 $[a_2] = \overleftarrow{C_{sqr}}([a_2], [a_5])$;
4 $[x] = \overleftarrow{C_-}([x], [x_{a_1}], [a_1])$;
5 $[y] = \overleftarrow{C_-}([y], [y_{a_1}], [a_2])$;
6 **return** $[x]$,$[y]$

Remarks

In Algorithm 7, the variable inc represents an inclusion test performed juste before intersecting a_6 and a_3. This value will be used later to compute intersection and union of contracted boxes.
In Algorithm 8, x_{a_1}, y_{a_1} and d_1 are left out of the retro-propagation since they are given as input values to the problem. The position of the landmark is supposed to be perfectly known and there would be little physical meaning in contracting the sensor measurements.

The contractor for the whole CSP $\mathcal{H}$ can be built by taking advantage of the intersection property highlighted in (1.32) *i.e.* $\overleftrightarrow{C_{\mathcal{H}}} = \overleftrightarrow{C_1} \cap \overleftrightarrow{C_2} \cap \overleftrightarrow{C_3}$ (Algorithm 9). Similarly, an hypothetical CSP $\mathcal{H}'$ where a pair of coordinates (x, y) is required to be compatible with at least one measurement is solved using contractor union i.e. $\overleftrightarrow{C_{\mathcal{H}'}} = \overleftrightarrow{C_1} \cup \overleftrightarrow{C_2} \cup \overleftrightarrow{C_3}$ (Algorithm 10). Both algorithms rely on the $\overleftrightarrow{C_n}$ contractors to compute the contracted box $[\mathbf{c_{n_{out}}}]$ and the inclusion test variable inc_{c_n} before performing the appropriate set operation.

Algorithm 9: Contractor $\overleftrightarrow{C_{\mathcal{H}}}$

Data: $[x], [y]$

1 $[\mathbf{c_{1_{out}}}], inc_{c_1} = \overleftrightarrow{C_1}([x], [y])$;
2 $[\mathbf{c_{2_{out}}}], inc_{c_2} = \overleftrightarrow{C_2}([x], [y])$;
3 $[\mathbf{c_{3_{out}}}], inc_{c_3} = \overleftrightarrow{C_3}([x], [y])$;
4 $inc_{\mathcal{H}} = inc_{c_1}$ **and** inc_{c_2} **and** inc_{c_3};
5 $[\mathbf{c}_{\mathcal{H}_{\mathbf{out}}}] = [\mathbf{c_{1_{out}}}] \cap [\mathbf{c_{2_{out}}}] \cap [\mathbf{c_{3_{out}}}]$;
6 **return** $[\mathbf{c}_{\mathcal{H}_{\mathbf{out}}}], inc_{\mathcal{H}}$

Algorithm 10: Contractor $\overleftrightarrow{C_{\mathcal{H}'}}$

Data: $[x], [y]$

1 $[\mathbf{c_{1_{out}}}], inc_{c_1} = \overleftrightarrow{C_1}([x], [y])$;
2 $[\mathbf{c_{2_{out}}}], inc_{c_2} = \overleftrightarrow{C_2}([x], [y])$;
3 $[\mathbf{c_{3_{out}}}], inc_{c_3} = \overleftrightarrow{C_3}([x], [y])$;
4 $inc_{\mathcal{H}'} = inc_{c_1}$ **or** inc_{c_2} **or** inc_{c_3};
5 $[\mathbf{c}_{\mathcal{H}'_{\mathbf{out}}}] = [\mathbf{c_{1_{out}}}] \cup [\mathbf{c_{2_{out}}}] \cup [\mathbf{c_{3_{out}}}]$;
6 **return** $[\mathbf{c}_{\mathcal{H}'_{\mathbf{out}}}], inc_{\mathcal{H}'}$

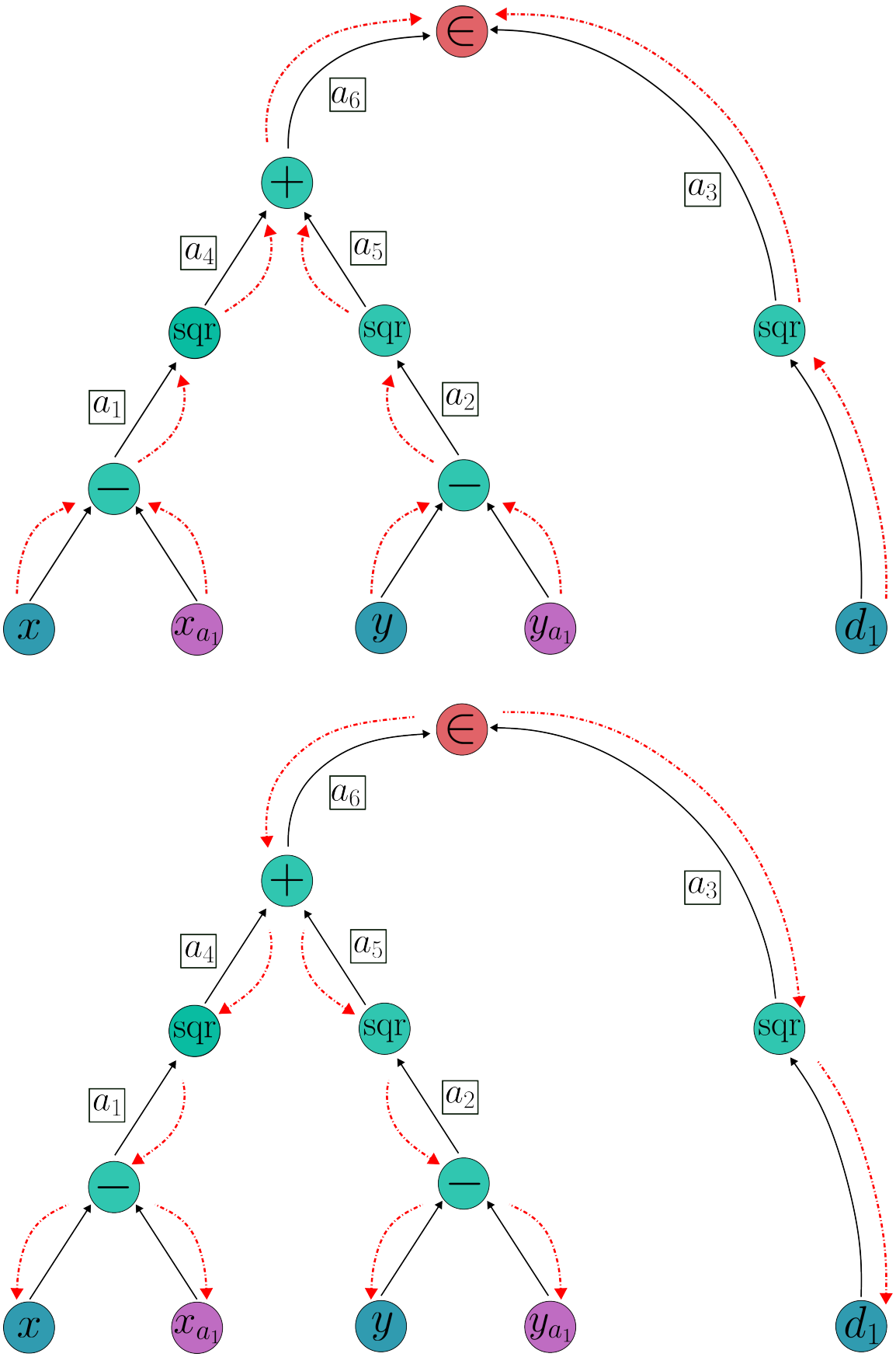


Figure 1.13: *Forward (up) and backward (down) propagation on the AST for C_1*

1.3.3 Computing The Subpavings

The SIVIA algorithm is used to generate the subpavings for the contractors $\overleftrightarrow{C_1}$, $\overleftrightarrow{C_{\mathcal{H}}}$, $\overleftrightarrow{C_{\mathcal{H}'}}$.

Example

For the sake of reproductability, the constants declared in Equation (1.39) are given the following values throughout this section:

	x_{a_n}	y_{a_n}	d_n
$n = 1$	22	24.5	[11.5, 13.5]
$n = 2$	25	4	[11.5, 13.5]
$n = 3$	4.5	15	[11.5, 13.5]

For $\overleftrightarrow{C_1}$, the set of feasible values takes the form of a ring centered around the upper-left landmark, which matches the inital intuition (Figure 1.14).

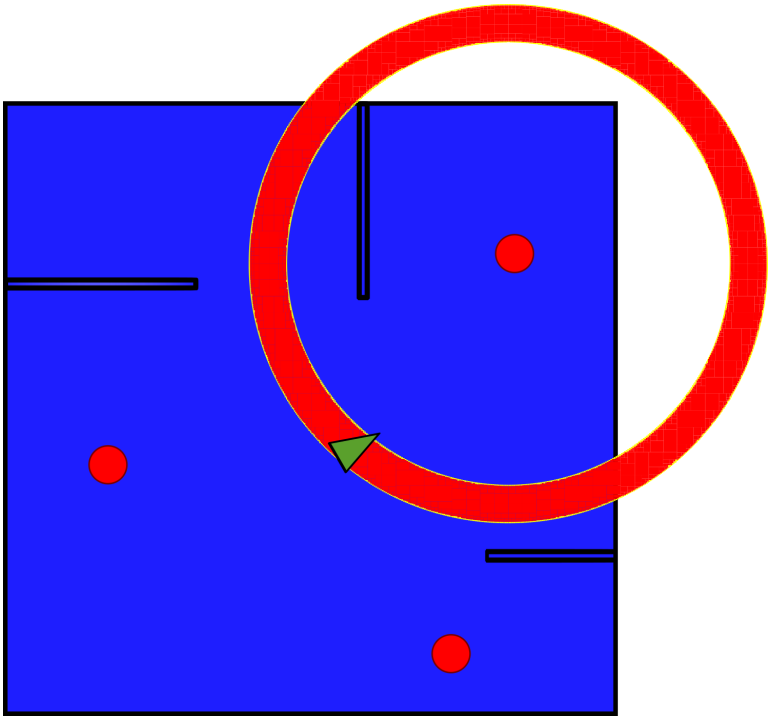


Figure 1.14: *SIVIA subpaving for one landmark (using* $\overleftrightarrow{C_1}$*)*

Using $\overleftrightarrow{C_{\mathcal{H}'}}$, SIVIA generates the subpavings illustrated on Figure 1.15. It can be intepreted as the set of values which are consistent with at least one of the measurements, which is the same as belonging to one of the three rings.

Finally, the contractor $\overleftrightarrow{C_{\mathcal{H}}}$ allows to generate the subpaving corresponding to the solution of the CSP $\mathcal{H}$ (Figure 1.16). It corresponds to the set of coordinates which are consistent with all the measurements, *i.e.* the intersection of all the rings.

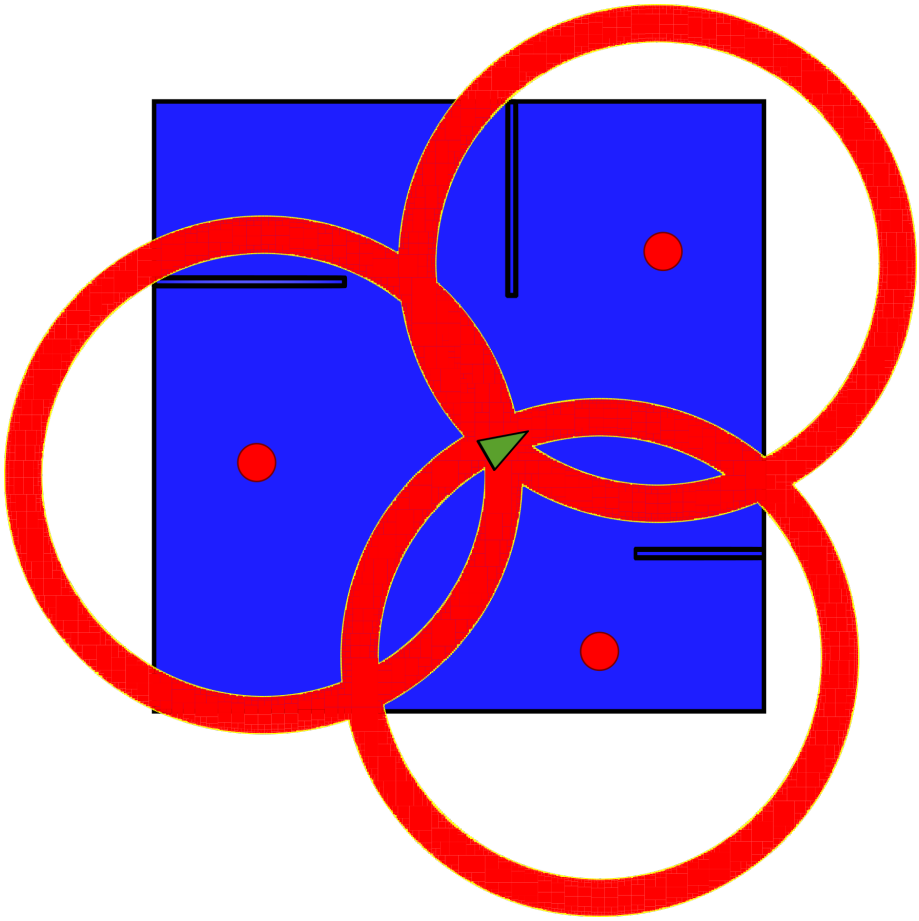
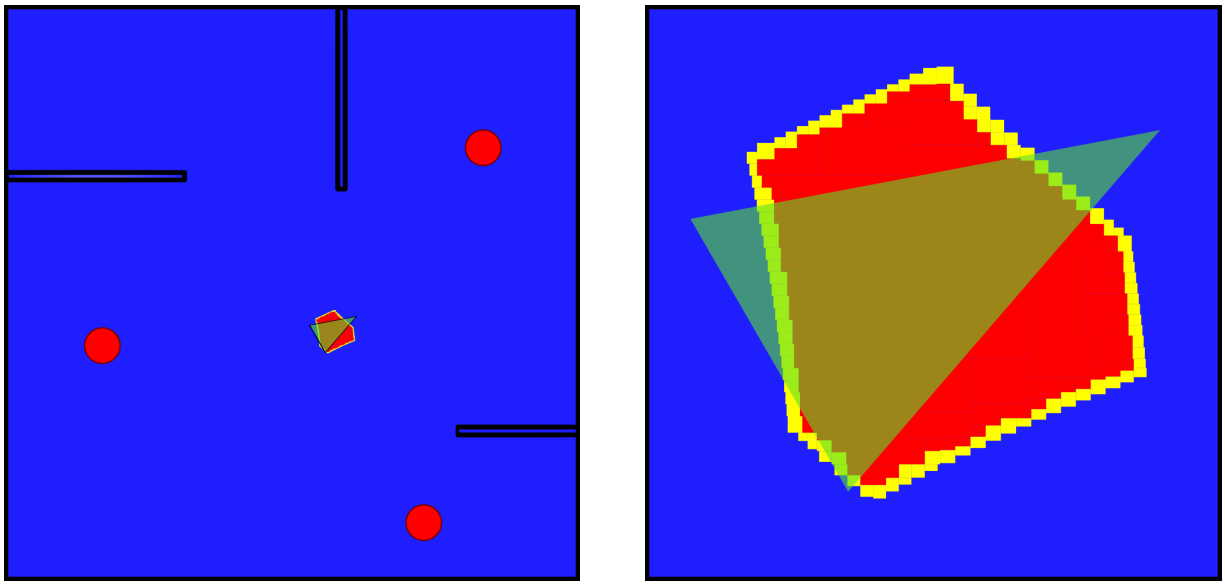


Figure 1.15: *SIVIA subpaving for the consistency with at least one landmark (using* $\overleftrightarrow{C_{\mathcal{H}'}}$*)*



(**a**) *overwiew* (**b**) *zoomed in*

Figure 1.16: *SIVIA subpaving for the consistency with all landmarks (using* $\overleftrightarrow{C_{\mathcal{H}}}$*)*

1.4 Practical Implementation

Now that the theoretical principles governing interval arithmetics have been established, this last section focuses on the efficient implementation of these problems on a computer. While all the behaviors presented in Section 1.2 can be re-implemented from scratch using any popular programming language (as long as it has a decent floating-point support), the most broadly used and time-saving approach is to pick one of the dedicated software libraries available in the rich ecosystem of interval analysis tools. Sometimes, the results obtained for the same computation can vary slightly depending of the library used, which proceeds from subtle differences in the way intervals are interpreted with respect to some specific domains of application. For example, a valid interval $[a, b]$ in the sense of Moore (1967) is necessarily *ordered* i.e $a \leq b$, *closed*, *bounded* and *non-empty*. This prevents the use of the *empty* interval $\varnothing$ and makes some operations illegal such as $[2, 4]/[-1, 1]$ (the result being unbounded). The interpretation presented in Section 1.2, by far the most commonly used in robotics application, is based on set theory and allows *non-bounded* and *empty* intervals. Some other exotic systems exists [33, 34] and have niche usages. This is the case of *Kaucher* intervals [35, 36] for which bounds can be *improper* (i.e. $a > b$) and which have found use in fields such as computer graphics or chemistry [37].

As a consequence, an IEEE standardization effort regarding intervals was initiated in 2008 (IEEE-1788 [38]) to bring order to all the available models and encourage the adoption of the technique by the industry. Since then, the standard has known two revisions in 2015 [39] and 2017 [40]. This process is especially useful to both mathematicians and programmers as it allows to create unified tools with reproducible behaviors.

The current section iterates over the main ideas developped in the IEEE-1788 standard and then conducts a commented survey of the major interval arithmetics software libraries available.

1.4.1 An Overview of the IEEE-1788 Standard

The standard is meant to aggregate multiple interval behaviors in a coherent way while remaining fairly easy to extend. This goal is achieved throught three key ideas:

- The concept of *flavors* allows fundamentaly divergent approaches to interval analysis to coexist.
- A mathematical theory organized in multiple layers of abstraction called *levels*.
- A *decoration* system to track relevant events during the course of a calculation.

1.4.1.1 Interval Flavors

As explained previously, intervals can be interpreted in many ways depending of the considered application, the most commonly used being the *set-based*, the *Kaucher* and the *cset* (containment set). The standard allows the coexistence of these fundamentaly different approaches to interval analysis through the concept of flavor. A first choice in the standard is to separate the properties shared by all the models from the specific ones. The consensus reached in the community is that all the valid intervals from Moore's $\mathbb{IR}$, the associated arithmetic operators $(+, -, \times, /)$ and some recurring functions such as $width$, $union$ or $intersection$ should be supported by all the flavors. The extensive list of common features is debated from Clauses 1 to 9 in the standard.

For each ratified flavor, the standard defines a set of operators and functions as well as their expected behaviors. As of 2025, the only fully specified flavor is the *set-based*, which was the main topic of Section 1.2, and is discussed in Clauses 10 to 14 in the standard.

1.4.1.2 Structuration Into Levels

Figure 1.17 illustrates the structuration into levels adopted by the IEEE-1788 standard which is borrowed to some extent from the IEEE-754 [41] standard on floating-point computations. This approach provides progressive steps to transition from abstract mathematical concepts to an efficient low-level implementation.

Relationships between specification levels for interval arithmetic for a given flavor $\mathfrak{F}$ and a given finite-precision bare interval type $\mathbb{T}$ of $\mathfrak{F}$.		
Level 1	Number system used by flavor $\mathfrak{F}$. Set of mathematical $\mathfrak{F}$-intervals. Principles of how $+$, $-$, $\times$, $\div$ and other arithmetic operations are extended to $\mathfrak{F}$-intervals.	Mathematical interval model
	$\downarrow \mathbb{T}$-*interval hull* total, many-to-one b) *identity map* $\uparrow$ total, one-to-one a)	
Level 2	A finite subset $\mathbb{T}$ of the $\mathfrak{F}$-intervals—the $\mathbb{T}$-interval datums—and operations on them.	Discretization
	represents $\uparrow$ partial, many-to-one, onto c)	
Level 3	Representations of $\mathbb{T}$-interval datums, e.g., by two floating-point datums.	Representation
	encodes $\uparrow$ partial, many-to-one, onto d)	
Level 4	Bit strings `0111000...`	Encoding

Figure 1.17: *The structuration into levels used in IEEE-1788. Credit to IEEE-1788* [39]

<u>Level 1: Mathematical model</u>

This level contains the mathematical theory behind a specific flavor. The entities found at this level are for example the valid intervals, the supported *decorators* (refer to Section 1.4.1.3) as well as the definitions for the available operators and functions.

Level 2: Discretization
This level provides tools to approximate the mathematical theory from level 1 by an implementation-defined finite set of entities and operations. The main entity of this level are the interval *datum* and *type*. The former is used to represent the intervals from level 1 with a finite precision, the latter represents finite set of interval *datum* made to abstract a particular way of representing intervals.

Level 3: Representation
This level represents interval *datum* in a more realistic way by introducing floating-points.

Level 4: Encoding
This level maps an interval from Level 3 to its final representation in memory.

1.4.1.3 The Decoration System

A decoration is an additional information joined to an interval to track relevant events which may have occured during the computation. It is typically meant to retain some properties for the function or the operator used to produce the interval, not directly for the interval itself. As stated in the IEEE-1788 standard, interval computations have two main purposes. The first is to obtain the correct range enclosure when evaluating a real function over an interval. The second one is to ensure the existence and the uniqueness of the result. Classic interval analysis targets the first goal whereas decorators provide efficient tools for the second one.

As an illustration of the dilemna solved by decorator, consider the evaluation of the interval $[x] = [-4, 4]$ by the function $\mathbf{f} : x \rightarrowtail \sqrt{x} - 0.5x$ using the *set-based* flavor.

$$\mathbf{f}([x]) = \sqrt{[-4, 4]} - [-2, 2] = [0, 2] + [-2, 2] = [-2, 4]$$

The problem raised by this computation is that $\mathbf{f}$ is not continuous over *x* and not even defined everywhere. This issue could be solved by returning a *NaN*-based value as soon as the input is not valid for the function, but it would happen so often over the course of a real application that interval analysis would be made inefficient. The solution adopted by the standard is to return the $[-2, 4]$ interval, add a decoration to indicate that the event "*out of bound*" occurred and let the user decide what to do with the result. Such an interval is then called a *decorated interval*.

Practically, the set of events worth being tracked by decorators depends mostly of the

considered flavor. Figure 1.18 presents the decorators defined by the standard for the *set-based* flavor.

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 $\mathrm{Dom}(f) = \emptyset$.

Figure 1.18: *The decorators for the set-based flavor of IEEE-1788. Credit to IEEE-1788* [39]

These decorators are linked together by an inclusion relationship, such as

$$\textbf{com} \subset \textbf{dac} \subset \textbf{def} \subset \textbf{trv} \supset \textbf{ill} \tag{1.40}$$

$$\textbf{com}, \textbf{dac}, \textbf{def} \text{ are disjoint from } \textbf{ill} \tag{1.41}$$

1.4.2 The Software Ecosystem

This section presents the most popular interval libraries and performs a short analysis of their design philosophy, internal architecture and performance trade-offs. For a more detailed review of the ecosystem, the reader may want to refer to sources such as [42, 43]. Figure 1.19 is a snapshot of the situation as of 2025 which indicates the timespan between the initial and last release for each project as well as the compliance to the IEEE-1788 standard.

The *Extension for Scientific Computation* (XSC-1) [44] is a pioneer implementation of interval arithmetic which started in 1967 at the university of Karlsruhe. It was repeatedly ported during the two following decades to support the contemporary architectures (Zuse Z23, Zilog Z80, System 370, ...) and languages (Pascal, Fortran77, ...). The first C version appeared at the beginning of the 90s and was maintained until 2001. Its successor, XSC-2 [45], was a C++ port of XSC-C released in 2004 which provided many significant improvements.

An other historical library is *PROFIL/BIAS* [46] (for *Programmer's Runtime Optimized Fast*

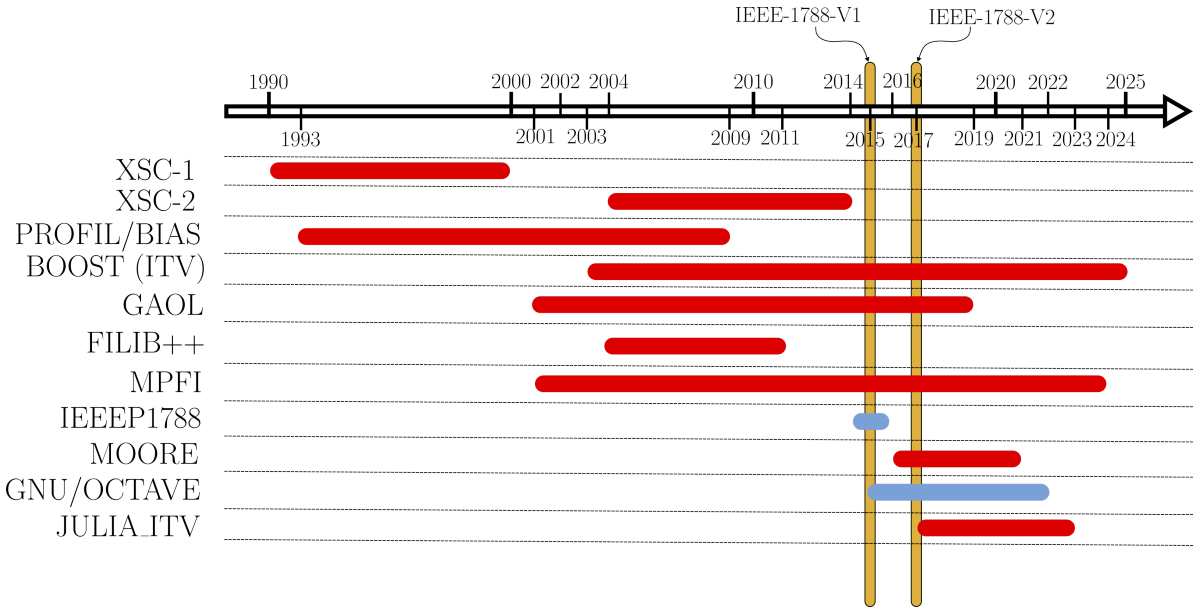


Figure 1.19: *Timespan between the first and last release for the major interval software libraries. Blue bars indicate a full compliance to the IEEE-1788 standard.*

Interval Library, Basic Interval Arithmetic) which is written in C++ by the Hamburg University of Technology. The efficient use of the underlying floating-point hardware is a major concern of the implementation which attempts to keep rounding-mode context changes to a minimum . This kind of code is essentially non portable and was thus declined to support a large number of architectures (now mostly outdated).

An interval analysis package [47] was added to the C++ *Boost* library in version 1.3 (2003) and frequently updated until 2024. This is a template-based implementation which supports a wide variety of bounds, such as hardware floating-points, multi-precision soft-floats or rationals. The novelty of this approach is to introduce the notion of policies (rounding, checks and comparaisons) to override the generic implementation with custom behaviors.

The *Gaol* library [48] is a popular interval library used as a dependency for a lot of robotics-oriented high-level libraries such as Ibex [49] or CODAC [50]. This is a C++ implementation relying on an external floating-point math library which can be either INRIA's *CRlibm* or IBM's *APmathlib*. Several architectures are supported such as x86/amd64, arm64 or Sparc.

Filib [51] is another template-based C++ library maintained between 2004 and 2011. Its objectives are similar to Gaol but it relies on the C++ standard library for the floating-point computations instead of a third-party math library. Filib can be an optional backend to Ibex in the place of Gaol.

MPFI [52] (for *Multiple Precision Floating-Point Interval*) is a portable C library which provides *guaranteed* and *precise* interval computation. It mainly focuses on multi-precision floating-point bounds computation, which is achieved through a dependency to the MPFR

[53] library to get correct roundings. This leads to high-precision results at the expense of speed due to the exclusive use of softfloats.

The *libieeep1788* [54] is a fully compliant implementation of the IEEE-1788 standard which supports the set-based flavor and the associated decorators. It was written in 2015 at the release of the standard and has not been updated since then. This is a template-based C++ library which relies on the correct roundings performed by the *MPFR* library.

The *Moore* library [55] is another interval library which claims partial compliance to the IEEE-1788, especially regarding the behavior of *set-based* intervals. However, its authors strongly advocate against the relevance of decorators and have chosen to put them aside of the implementation. It provides native hardware supports for float32, float64 and float128 interval bounds and relies on *MPFR* for bigger precisions.

GNU/Octave [56] is an other fully compliant impementation of the IEEE-1788 standard written in the Octave language, which is closely related to Matlab. Once again, it relies on MPFR for correct bounds computations.

The *ValidatedNumerics* package [57] of the Julia programming language is an other popular option which has gained a lot of tractions in the community over recent years. It relies on an internal binding over CRlibm and *MPFR* for the computation of the bounds.

1.4.3 The Limits of Software

The previous review highlighted the two major bottlenecks encountered during the software implementation of interval arithmetic.

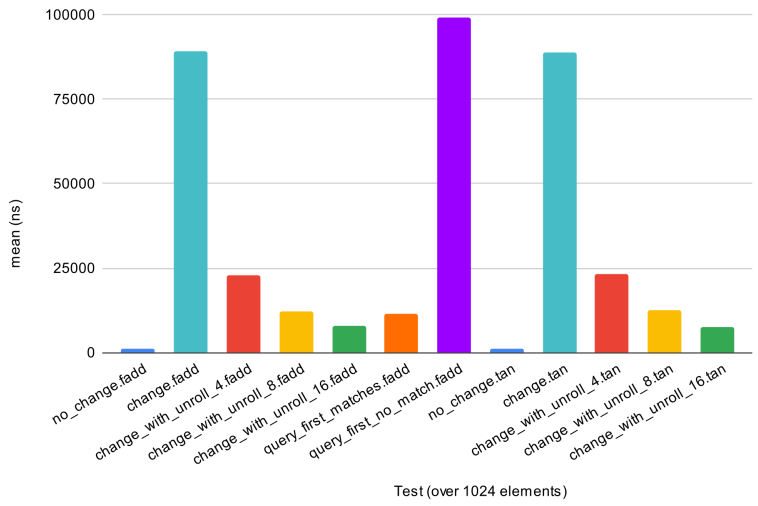
1.4.3.1 The Impact Of Rounding-Mode Context Change

Each computation is performed on interval operands, which are assumed to contain the true value for the associated physical measurement. The result is also an interval which, according to the *Fundamental Theorem of Interval Analysis* (FTIA), is guaranteed to contain the exact value of the ouput. A requirement of the FTIA is to ensure that the bounds of the result have been obtained through correct IEEE-754 floating-point rounding (refer to Section 2.3.2.3 for more details), i.e. round-downward (RD or round to $-\infty$) for the lower bound and round-upward (RU or round to ∞) for the upper bound.

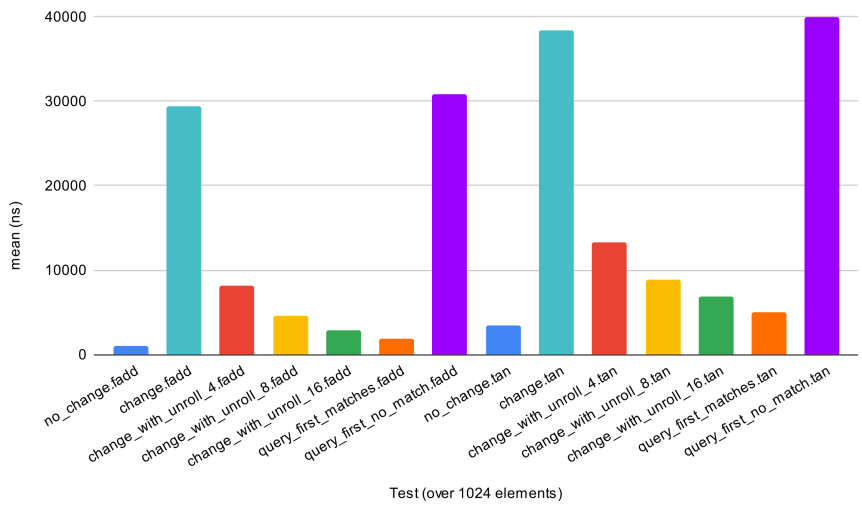
For example, the addition from Equation 1.13 can be rewritten for correct rounding as Equation 1.42 where $\lfloor . \rfloor$ and $\lceil . \rceil$ are RD and RU roundings.

$$[x] + [y] = [\lfloor \underline{x} + \underline{y} \rfloor, \lceil \bar{x} + \bar{y} \rceil] \tag{1.42}$$

As a consequence, a typical interval analysis application must perform frequent context changes between three modes: RD and RU for bounds and Round-To-Nearest (RNE) for everything else. The ISO-C standard provides the *fenv.h* header and the functions **fesetround** and **fegetround** to respectively modify and read the current rounding-mode. Their actual implementations are specific to the target architecture and may have a major impact on performances. For example, x86 and arm64 maintain the rounding-mode for the whole floating-point pipeline through dedicated CPU registers (respectively *mxcsr* and *fpcr*) which means that any change triggers an immediate stall and flush of the pending instructions. An estimation of the time penalty caused by the context switch is performed in [58] for x86 and arm64. The results are presented on Figure 1.20 and suggest an average $30\times$ slowdown on arm64 and $70\times$ on x86.



(**a**) *x86*



(**b**) *arm64*

Figure 1.20: *Impact on the performances caused by a change of floating-point rounding mode for x86 and arm64. Credit to Neil Henning [58]*

The workaround used by some implementations such as *Gaol* or *Boost* is to use the symmetry properties of IEEE-754 roundings to compute the lower bound while using the RU mode, as depicted in Equation 1.43 for the addition. However, this often requires extra side-computations and does not supress the need to switch to RNE from time to time.

$$\lfloor \underline{x} + \underline{y} \rfloor = -(\lceil (-\underline{x}) + (-\underline{y}) \rceil) \tag{1.43}$$

Finally, some hardware architectures, such as RISC-V (refer to Chapter 3), implement floating-point roundings on a per-instruction basis, which means that a context switching comes at a much lower price and does not trigger the flush of the pipeline.

1.4.3.2 The Efficient Use of Floating-Point Acceleration Primitives

Most of the software libraries presented in Section 1.4.2 rely on a third-party softfloat library like *CRLibm* or *MPFR* (Figure 1.21), which means that all the IEEE-754 floating-point computations are performed using large integers instead of using the dedicated hardware *Floating Point Unit* (FPU). This approach has several benefits; not only does it allow to run practically everywhere even when no FPU is available, but it also supports non-standard floating-point formats (outside of float32 and float64). Custom precisions are a truly interesting asset of softfloat libraries, especially applications whose needs are not covered by float32 or float64, despite being the only formats commonly supported by FPUs. However, such flexibility comes at the cost of performances, which can be several orders of magnitude slower than their hardware-accelerated counterparts. Figure 1.22 presents the relative slowdown achieved by *MPFR* (when using a mantissa of 52 bits *i.e.* equivalent to float64) compared to hardware floating-points (float64) on a set of operations (addition, subtraction, multiplication, division, square-root and cosine). The results shows a slowdown which can be as high as seventy times.

1.5 Conclusion

This section presented the fundamentals of interval analysis and applied the theory in the context of a typical robotics localization problem. An overview of the ecosystem of software libraries dedicated to intervals was also performed as well as a discussion regarding their implementation choices and conformity to the IEEE-1788 standard. The wide range of options available reflects the diversity of application areas for interval analysis, each which different expectations regarding speed and precision. However, this analysis reveals a strong emphasis on high-precision calculations, as indicated by the number of libraries using softfloats, at the expense of execution speed. This ap-

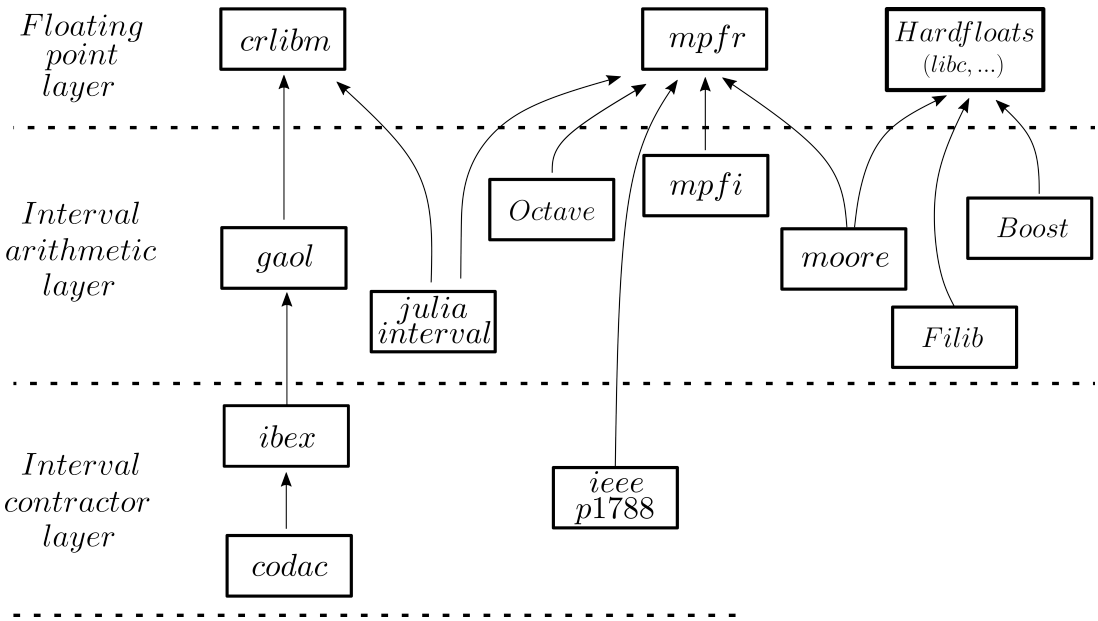


Figure 1.21: *Chain of dependencies for the interval software libraries.*

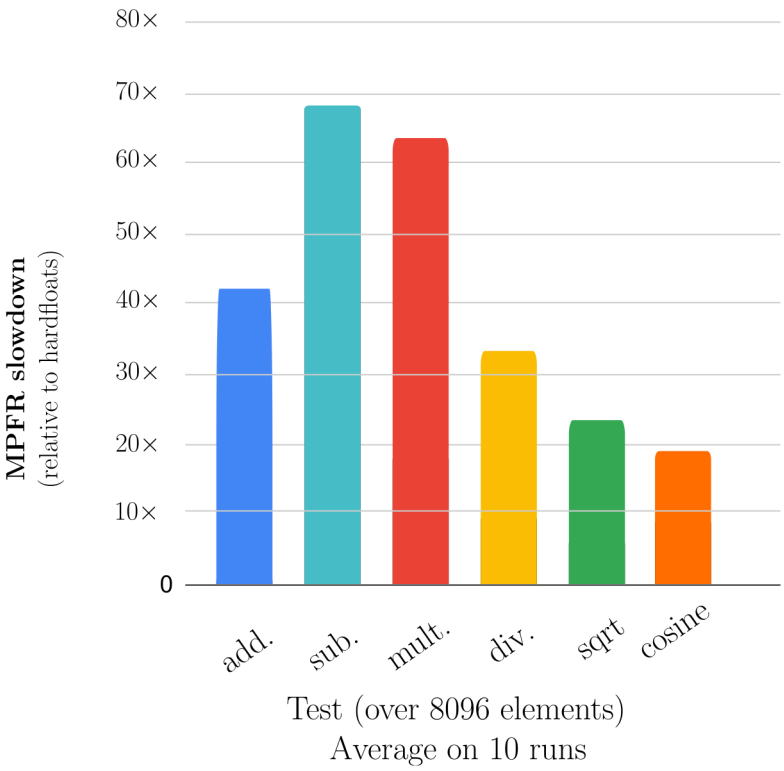


Figure 1.22: *Relative slowdown of MPFR compared to hardfloats.*

proach is generally not desirable in robotics where the core of a lot of problems is to feed real-time sensor measurements to an interval contractor (Section 1.3). These measurements rapidly become obsolete and are replaced by new ones, which means that the evaluation must be performed within a small time frame. It is thus preferable to get a

pessimistic yet guaranteed result (a bigger interval which still encloses the real value) in a short amount of time and move to the next iteration, rather than waiting longer for extra bits of precisions. The hereby thesis advocates in favor of this strategy and proposes a hardware approach to robotics-friendly interval analysis in the hope to mitigate the usual bottlenecks presented in Section 1.4.3.

2

Hardware Acceleration Strategies for Interval Arithmetic

Contents

2.1 Context **51**
2.2 Graph-Level Acceleration **53**
2.2.1 GPU Strategy 53
2.3 Contractor-Level Acceleration **64**
2.3.1 Using SIMD Instructions 64
2.3.2 Using Custom Hardware 64
2.4 Conclusion **71**

2.1 Context

The previous chapter presented the fundamentals of interval analysis and applied them to traditional robotics problems such as localization. The proposed method relies on three main steps:

- 1) Characterize the associated *Constraint Satisfaction Problem* (CSP).
- 2) Build the relevant contractors with the HC4R algorithm.
- 3) Use a *Branch and Bound* (B&B) strategy such as SIVIA to solve the problem in the selected search space.

B&B algorithms, although very precise, are known to be resources-demanding and can end up being a cause of bottleneck, especially in time-critical applications. For example, the amount of computation required by SIVIA grows exponentially with the size of the search space and the complexity of the contractor used. The typical execution flow of

SIVIA can be illustrated by the mean of a graph (Figure 2.1) where each node accounts for one evaluation of a box through a contractor. Following the algorithm presented in Section 1.2.5, a box can be either dropped (not a solution), stored (full-included in the solution space) or bisected (partially included in the solution space). The last situation produces a child node for each bisected halves and schedules new iterations over this search space.

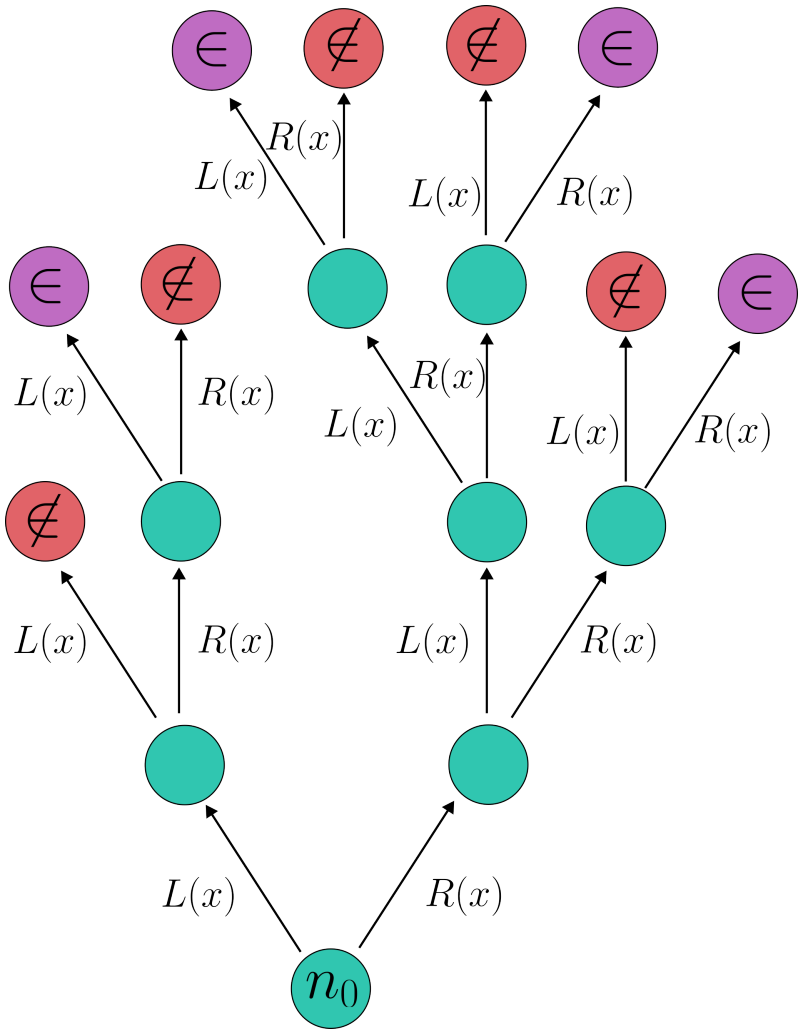


Figure 2.1: *Execution flow of a sequential SIVIA represented as a graph. Each color of node represents a particular output following the evaluation of a box by the associated contractor. Green stands for a partial inclusion in the solution space leading to a bisection. Red and Purple are leaf nodes accounting respectively for disjoint and full-inclusion in the solution space*

Intuitively, B&B algorithms feature a high degree of parallelism which has been thoroughly explored in the literature [59–61]. These studies come as a response to the end of Moore's law [62], an empirical observation made by Gordon Moore in 1965 predicting that the number of transistors on a chip would double every two years. This principle was verified until 2010's and harware vendors would achieve regular and tremendous single-core performances boosts as the size of transistors decreased. During this period, changing the processor was a guarantee of a near 30% speed-up, which indubitably brought laziness in software development as naive approaches to programming were

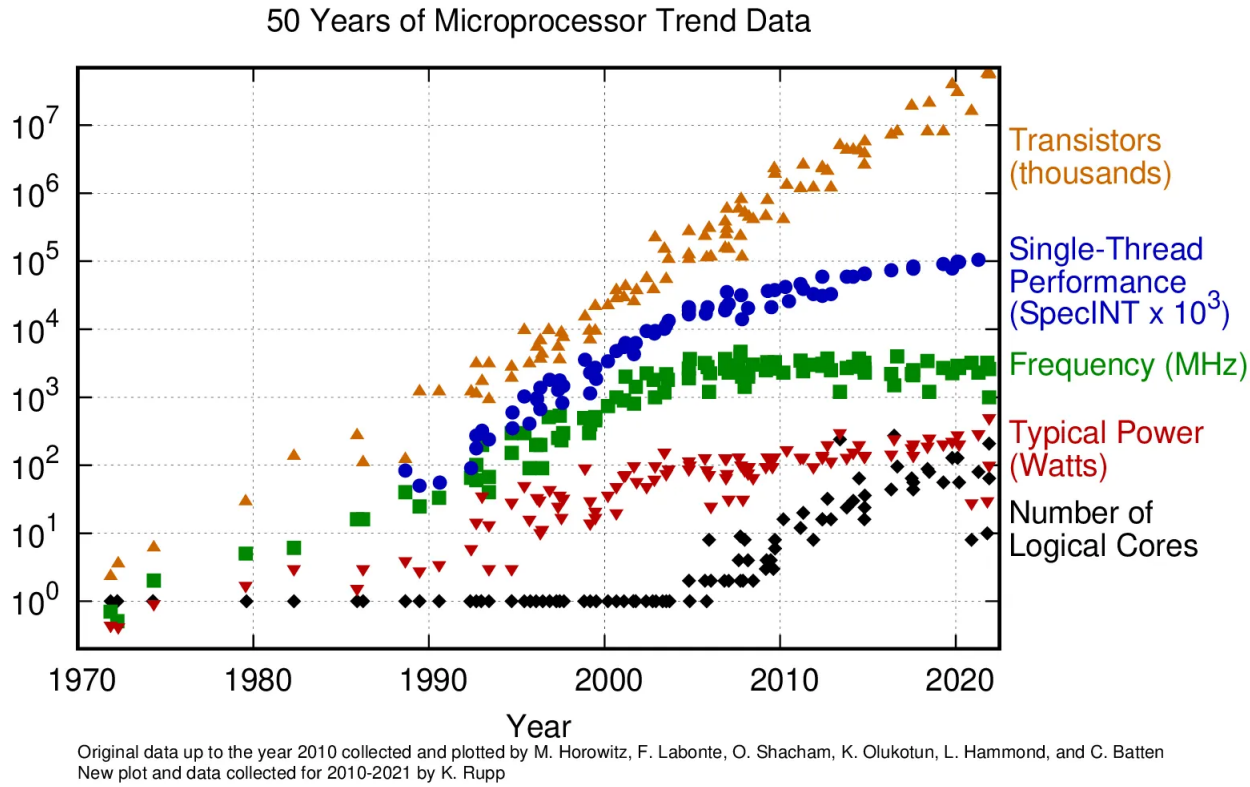
still rewarded. Funnily, this led to the Wirth's law [63] (1995), an adage explaining that the software gets slower more rapidly that the hardware gets faster, or put differently "What Intel Giveth, Microsoft Taketh Away". However, the golden age has been over for more than a decade as the transistor manufacturing processes are reaching the limits imposed by laws of physics (2nm in 2025). The whole situation is illustrated in Figures 2.2a and 2.2b. As a result, hardware vendors began to consider other ways to increase chip performances which did not involve the improvement of single-thread performances. A first solution was to generalize the use of many-cores processors in the consumer segment to support task-parallelism directly at the hardware level. Nowadays, eight cores processors have become the standard in desktop computers and having more than hundred is not uncommon in servers or compute nodes. The second one was to turn away from generic-purpose architectures and design application-specific accelerators such as *Graphics Processor Units* (GPUs) for graphic-intensive tasks, *Digital Signal Processors* (DSPs) or more recently *Neural Processor Units* (NPUs) for AI acceleration. This approach is very efficient but complicates the life of the developpers who need to master the optimal programming paradigms for the platform (e.g. the CUDA/OpenCl toolkits for GPUs, ...).

Two major axis of hardware acceleration are usually considered in interval analysis applications. The first one targets the inherent data-parallelism of B&B algorithms in which the same contractor is called repeatedly on varying input data. This strategy is best handled on many-cores platforms and associates a node of the graph from Figure 2.1 to each available execution units for an evaluation in parallel. Throughout this chapter, such approach will be refered as a *graph-level* acceleration. The second method operates at a lower-level and attempts to optimize the execution of the contractor itself by providing a set of dedicated hardware primitives to accelerate the most recurring interval operations (see Section 1.2.2). This approach is especially useful in embedded robotics applications where no massively-parallel hardware (GPUs, remote computing on a cluster, ..) is available. An example is the localization problem described in Section 1.3, which would achieve poor performances, should the SIVIA from Figure 1.15 be executed for each new measurement. The workaround is to ommit the B&B step and replace it by a series of call to the contractor (centered around the previous known location) until a convergence is reached (refer to Figure 1.5b). In this context, the faster the contractor evaluation, the better. Such strategy is referred as a *contractor-level* acceleration.

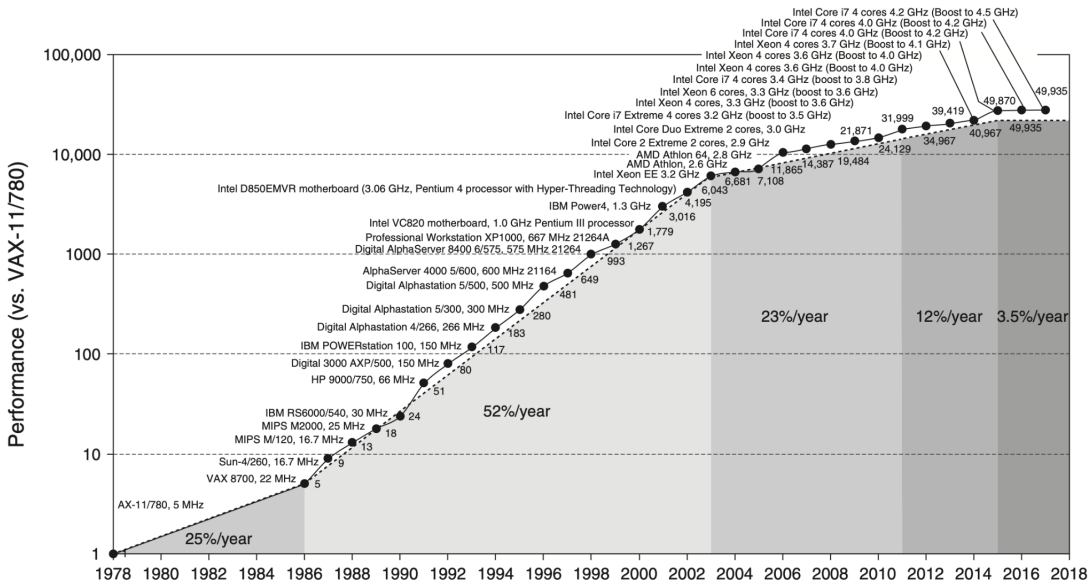
2.2 Graph-Level Acceleration

2.2.1 GPU Strategy

The term *General-Purpose Processing on Graphics Processor Unit* (GPGPU) has been around since NVIDIA released its first graphics-dedicated chip (GeForce 256) in year 1999-



(**a**) *Evolution of some relevant parameters of CPU micro-architectures over the years. The firs signs of the end of Moore's appear in late 2000's with a slower increase of single thread performances, stalling frequencies and a new focus on many-cores architectures. Credit to* [64]



(**b**) *Paving with* $\epsilon = 0.01$

Figure 2.2: *Relative increase in performances for the flagship processors of each generation since 1978. The steady increase started to slow down at the beginning of the 2000's. Credit to* [65]

2000. Many researchers from fields such as medical imaging, electromagnetics or fluid mechanics immediately tried to take advantage of this massively parallel hardware to solve non graphics related tasks. This process was not facilitated by the contemporary GPU programming frameworks which were designed solely for graphics rendering (e.g. OpenGL [66]). NVIDIA, recognizing the potential of bringing those performances to the scientific community released the CUDA [67] framework in 2007 which exposed multiple levels of abstraction to the programmers to perform both graphics-rendering and general-purpose computing efficiently. An open-source standard named OpenCl [68] was then released in 2009 by the Khronos group and common allowed GPGPU primitives to be implemented by every GPUs vendors.

Since then, the performances of GPU chips have known regular and significant increases. Video games have always been a huge factor of innovation on the graphics-side due to an ever growing demand on better framerates, high-fidelity lightning and higher resolutions. However, the biggest surge came from the compute-side with the spreading of *Artificial Intelligence* (AI) at the beginning of the 2010's and its massive boom in 2022 following the introduction of *Large Language Models* (LLM) such as Chat-GPT. GPUs instantly became the preferred platform to train and execute AI models, leading to the addition of new hardware primitives called Tensor-Cores specifically designed to accelerate these applications.

This section performs a brief presentation of the compute capabilities of NVIDIA GPUs by reviewing both the relevant hardware architecture points and the essentials of the CUDA programming model. In a second time, the main strategies used to implement interval analysis efficiently on the GPU are reviewed.

2.2.1.1 Architecture

The architecture of NVIDIA GPUs has evolved since the introduction of CUDA in 2009. Major releases (denoted by specific codenames as shown in Figure 2.3) happen roughly every two years and come with several improvements or new features. This event is generally accompanied by the release of the associated architecture whitepaper which outlines the key architecture details for the programmer. As an example, Figure A.1 from Appendix A originates from the Hooper whitepaper [69] and presents a detailed view of the H100 chip, the flagship of this architecture.

For educational purposes, the simplified depiction from Figure 2.4 will be used in this section to compare the differences between the organization of a CPU and a GPU chip. A high-performance multi-cores CPU is built around several elements: a few *Arithmetic and Logic Units* (ALU) which are the main execution units, a control mechanism capable of performing complex branch predictions and out-of-order execution and a multi-level cache hierarchy to provide efficient access to memory. GPUs do things the opposite way by embarking a large number of *Streaming Multiprocessors* (SMs), which are roughly the

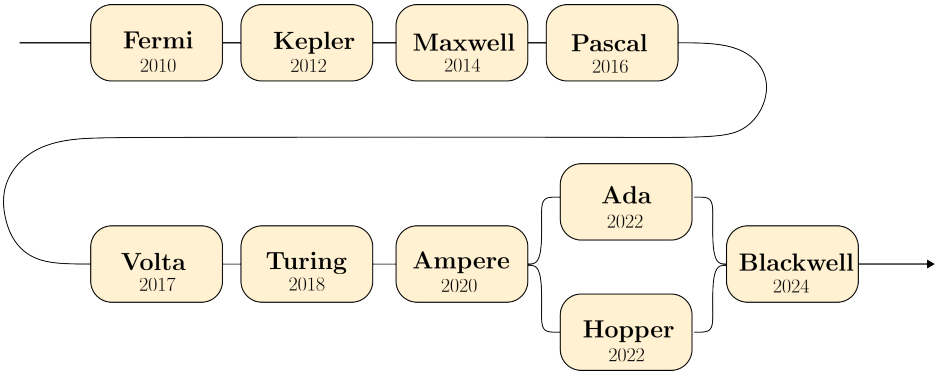


Figure 2.3: *Codenames for recent NVIDIA GPU architectures*

equivalent of CPU's ALU (green squares on Figure 2.4), which share rudimentary control units and caches.

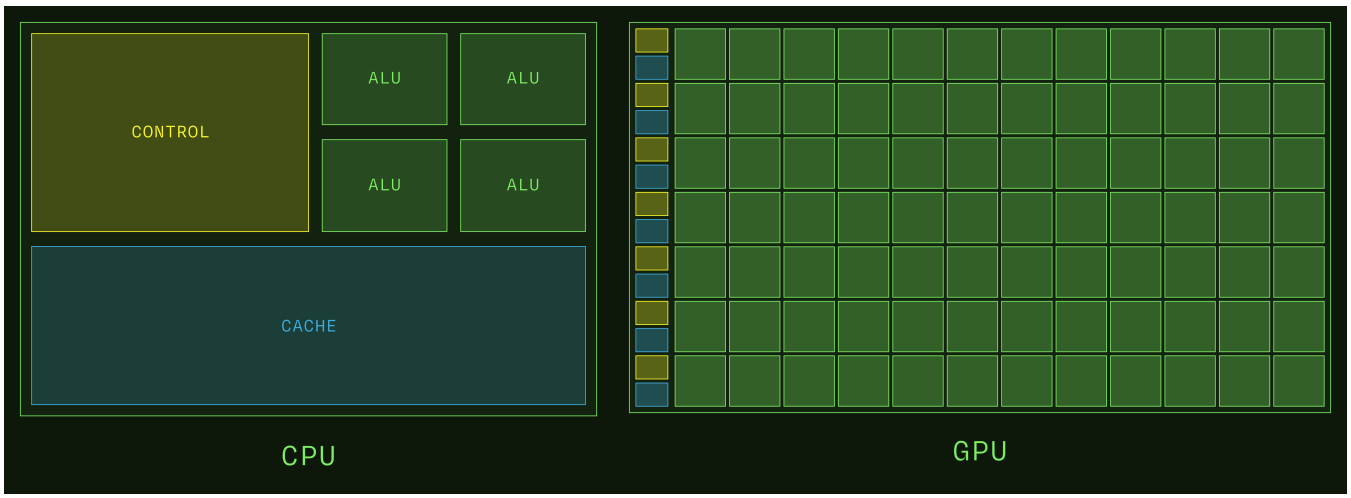


Figure 2.4: *A comparison of CPU and GPU architectures - Credit to Modal's GPU glossary*[70]

The organization of a SM is illustrated in Figure 2.5, although the exact content may vary slightly depending on the considered architecture. Its main benefit is the ability to execute a large number of lightweight and concurrent threads, far beyond the capabilities of the most performant CPUs. However, these threads cannot be issued instructions independently and are instead controlled in packs of 32. NVIDIA terminology refers to such a group as a *warp*.

The main components of a SM are described below:

A common instruction cache: The instructions intended for a particular SM are stored in a dedicated cache (L0) and waited to be issued to a warp.

Warp schedulers and dispatch units: In a SM, a warp can be either busy or available for work. Busy warps cannot be issued new instructions until the pending one finishes, which may take a few clock cycles depending of the complexity of the operation. At each

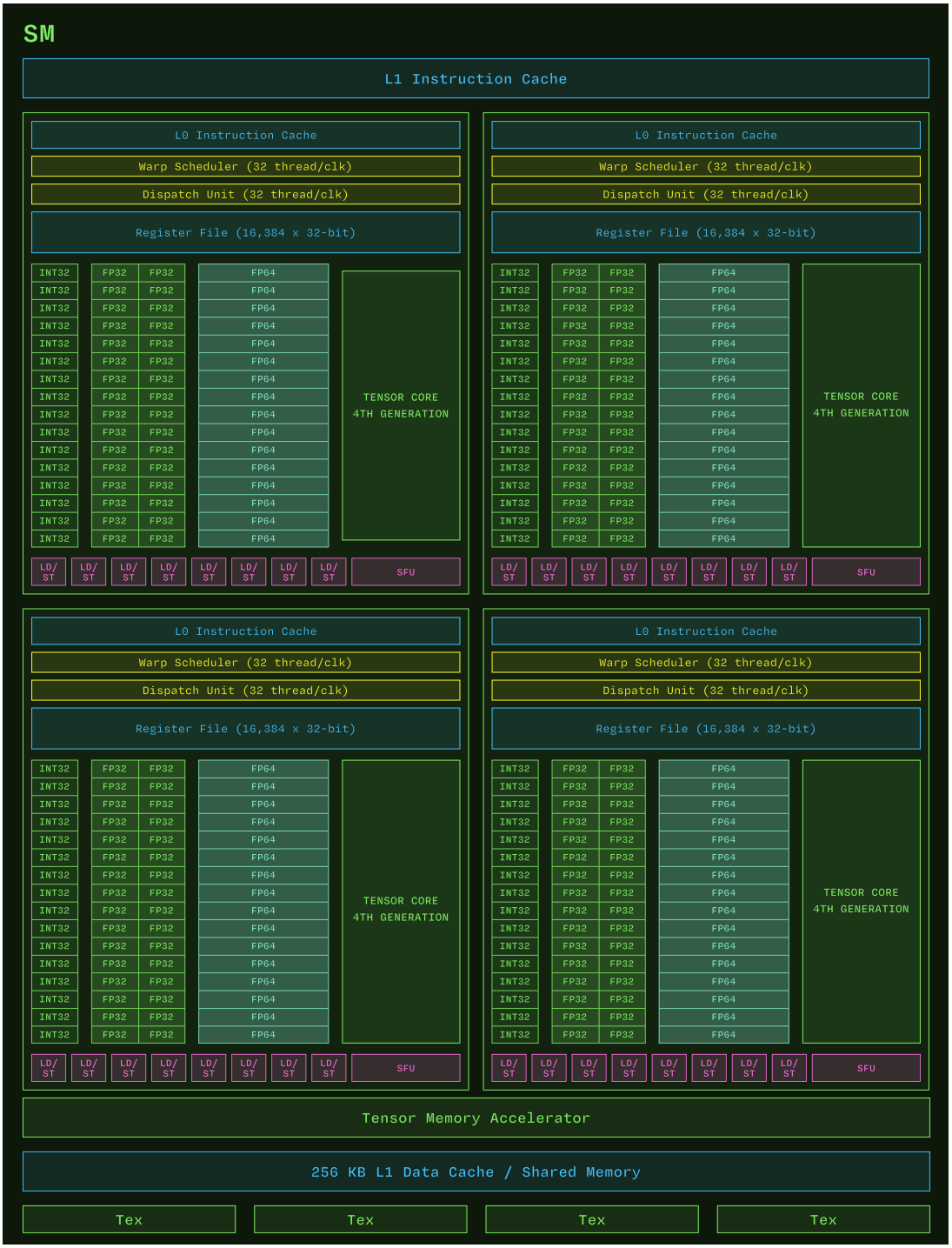


Figure 2.5: *Content of a typical streaming multiprocessor - Credit to Modal's GPU glossary*[70]

clock cycle, warp schedulers each elect an available warp and issue one instruction from the cache. This low-penalty context-switch allows to reach a maximum utilization of the hardware resources and effectively hides the latencies of the pending instructions.

A dedicated set of registers: The instructions may be issued for a warp but each thread has access to dedicated registers with its own input data, thus enforcing the *Single Instruction Multiple Data* (SIMD) paradigm.

Computation primitives: Each SM owns a predefined number of hardware blocks to accelerate some specific computations. The actual repartition depends on the chip considered as shown on Figure 2.6 for the A100 and H100 GPUs.
The main primitives are :

- Arithmetic operators: For both integers (int8, int32) and floats (fp16, fp32, fp64)
- Algebraic and transcendental functions: The *Special Function Unit* (SFU) contains implementations for the square-root, logarithm, exponential, sine, cosine, ...
- Tensor-cores: Efficient matrix multiplications, Accumulators, ...

GPU Features	NVIDIA A100	NVIDIA H100 SXM5	NVIDIA H100 PCIe
GPU Architecture	NVIDIA Ampere	NVIDIA Hopper	NVIDIA Hopper
GPU Board Form Factor	SXM4	SXM5	PCIe Gen 5
SMs	108	132	114
TPCs	54	66	57
FP32 Cores / SM	64	128	128
FP32 Cores / GPU	6912	16896	14592
FP64 Cores / SM (excl. Tensor)	32	64	64
FP64 Cores / GPU (excl. Tensor)	3456	8448	7296
INT32 Cores / SM	64	64	64
INT32 Cores / GPU	6912	8448	7296
Tensor Cores / SM	4	4	4
Tensor Cores / GPU	432	528	456

Figure 2.6: *The computation primitives available on NVIDIA A100/H100 and their respective quantites - Credit to NVIDIA Hooper Whitepaper* [69]

2.2.1.2 Programming with CUDA

The CUDA programming model follows a heterogenous computing strategy in which a CPU (the *host*) delegates data-intensive tasks to a guest GPU (the *device*). The language itself is essentially an extended version of C with handy constructs to implement the typical four-step CUDA flow:

- Description of the work on the host-side.
- Copy of the program instructions and input data to the device memory.
- Computation on the device, results are stored locally.
- Copy of the result from host to device memory.

The simplest CUDA program is shown on Listing 1 and is composed of two main code segments. The function *kernel_hello* is prefixed by the keyword *__global__* which tells the compiler to run this portion of code on the device side. Such segments are called

kernel in the CUDA terminology. The fonction *main* is the standard entry point of C executables and is run on the host when the program starts. In this simplified example, it declares a block and a grid which will be used as launch parameters for the kernel. This is an abstraction mechanism provided by CUDA to organize the pool of worker threads responsible for the computation. In this case, *kernel_hello* is executed using a 2×3 grid composed of 3×5 blocks, for a total of 90 threads. CUDA runtime automatically populates the *blockIdx* and *threadIdx* variables in each thread (lines 7 and 8) to provide it with a bit of context in the global execution.

Listing 1: *Hello world in CUDA*

```
1  #include <cuda_runtime.h>
2  #include <stdio.h>
3
4  __global__ void kernel_hello() {
5      printf("#########################");
6      printf("Hello from a GPU thread !\n");
7      printf("BlockID: [%u %u %u]\n", blockIdx.x, blockIdx.y, blockIdx.z);
8      printf("ThreadID: [%u %u %u]\n", threadIdx.x, threadIdx.y, threadIdx.z);
9      printf("#########################");
10     return;
11 }
12
13 int main(int argc, char** argv) {
14
15     /* Specifying the number and organisation of worker threads */
16     dim3 grid = (2, 3, 1); /* A grid made of 2 blocks by 3 */
17     dim3 block = (3, 5, 1); /* Each block has 3 by 5 */
18     /* Launching kernel */
19     kernel_hello<<<grid, block>>>();
20     return 0;
21 }
```

When the execution starts, the kernel is passed to the GPU and executed by all the threads specified in the grid/block configuration (Figure 2.7). This abstraction is only relevant to the programmer as the actual behavior of the low-level side remains faithful to Section 2.2.1.1.

2.2.1.3 Intervals on the GPU

Interval analysis and its efficient implementation on a GPU has been the topic of many research works over the years due to the suitability of this hardware for the kind of B&B problems depicted in Figure 2.1. This section discusses some of the major attempts.

An early work from Collange et al. [71, 72] in 2010 proposed a CUDA-based library called *"cuda interval rounded arith"* dedicated to interval arithmetic. This implementation takes

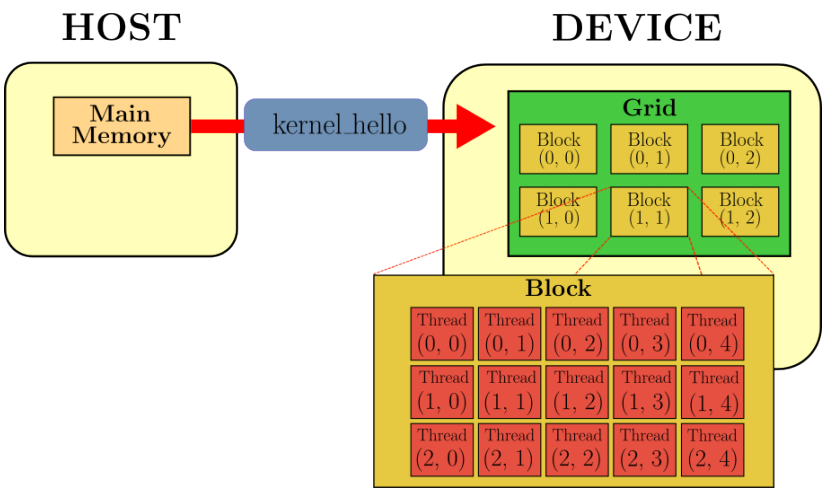


Figure 2.7: *Example of CUDA grid/block mapping on the GPU*

advantage of the fact that NVIDIA GPUs are capable of performing IEEE-754 arithmetic computations in any rounding mode without having to pay the price for context-switches like on x86 CPUs, thus removing a major performance bottleneck in traditional interval libraries (see Section **??**). The CUDA programming language provides a set of compiler intrinsics [73] functions to perform correctly-rounded operations (Listing 2) which are then translated to NVIDIA's PTX assembly [74] instructions during the compilation process.

Listing 2: *CUDA compiler intrinsics for correctly-rounded float32 addition*

```

/* Add two floating-point values in round-down mode */
 __device__ float __fadd_rd(float x, float y);

/* Add two floating-point values in round-to-nearest-even mode */
__device__ float __fadd_rn(float x, float y);

/* Add two floating-point values in round-up mode */
__device__ float __fadd_ru(float x, float y);

/* Add two floating-point values in round-towards-zero mode */
__device__ float __fadd_rz(float x, float y);


```

By using these elementary constructs (Figure 2.8a), the authors managed to build an interval template class (Figure 2.8b) which support the common operations in both single and double-precision.

The performances of the library are then compared with those obtained with the CPU interval library *Boost interval* (refer to Section 1.4.2). The considered benchmark searches the root of the set of functions $\mathbf{f}(x) \rightarrow x^2 + (\alpha - 2)x + 1$ (α varies among worker threads)

```
template<class T> inline __device__
interval<T> operator+(interval<T> const & x,
    interval<T> const & y)
{
    rounded_arith<T> rnd;
    return interval<T>(rnd.add_down(x.lower(), y.lower()),
                       rnd.add_up(x.upper(), y.upper()));
}
```

```
template<>
struct rounded_arith<float>
{
    __device__ float add_down (const float& x, const float& y) {
        return __fadd_rd(x, y);
    }
    ...
}
```

(**a**) *Binding over CUDA's intrinsics*

```
template<class T>
struct rounded_arith
{
    __device__ T add_down (const T& x, const T& y);
    __device__ T add_up   (const T& x, const T& y);
    __device__ T sub_down (const T& x, const T& y);
    __device__ T sub_up   (const T& x, const T& y);
    __device__ T mul_down (const T& x, const T& y);
    __device__ T mul_up   (const T& x, const T& y);
    __device__ T div_down (const T& x, const T& y);
    __device__ T div_up   (const T& x, const T& y);
    __device__ T median   (const T& x, const T& y);
    __device__ T sqrt_down(const T& x);
    __device__ T sqrt_up  (const T& x);
    __device__ T int_down (const T& x);
    __device__ T int_up   (const T& x);
};
```

(**b**) *Main template class*

Figure 2.8: *An overview of Collange et al.'s implementation - Credit to* [71]

and exhibits multiple branch divergences and irregular memory accesses to measure the impact on throughput. The results are presented in Figure 2.9 and show a speed-up by an order of magnitude of 1000 for the GTX 480 GPU and by 100 for the Tesla C1060 GPU compared to the CPU implementation. This difference is explained by the Tesla card belonging to an older GPU familly which is not as good for cached-memory accesses, which add credits to NVIDIA's effors tregarding micro-architecture improvements.

Collange et al.'s library was re-used in Bagóczki et al.'s [75] paper from 2017. The goal of this work was to propose a parallel B&B strategy to solve the problem of circle covering

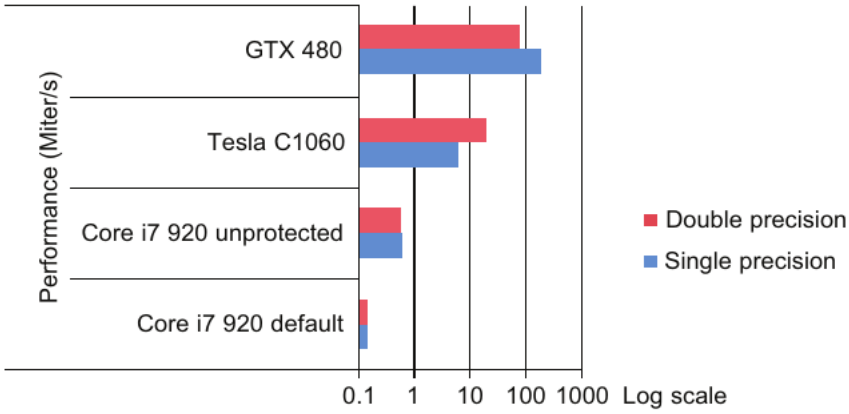


Figure 2.9: *Speed-up achieved with Collange et al.'s implementation - Credit to* [71]

using interval analysis, i.e. finding the densest covering of a shape using n congruent circles. The pseudo-code of the algorithm used in the paper is shown on Figure 2.10

Algorithm 1 A parallel B&B algorithm.

```
Funct PIBB(param, ϵ)
Set the working buffer B_W := {param} and the result set L := {}
Calculate current thread ID: tid
while B_W is not empty do
  Pop v from B_W and split along all sides, and take the subproblem which
  belongs to our current thread: v_tid
  if maxWidth(v_tid) < ϵ then                          Termination rule
    L^tid = false
  else if isMetQuality(v_tid) is false then
    Push v_tid into B_W
    break
  end if
  Synchronize threads
  if tid = 0 then
    for i = 1, ..., sizeOf(L) do
      if L^i = false then
        print Cannot decide whether the quality is not met or the result is
        smaller than ϵ
        return false
      end if
    end for
  end if
end while
print The quality is met
return true
```

Figure 2.10: *Parallel B&B algorithm in Bagóczki et al.'s work - Credit to* [75]

The performances obtained for the circle and sphere covering applications were compared with a CPU implementation and the results are shown on Figure 2.11. The speed-up obtained ranges from 10 to 30 times in the circle case and as high as 1500 times in the sphere case, depending of the problem size n.

The research effort regarding parallel B&B for intervals analysis is still going on with a recent paper by Zhang et al. [76] (2025). Its approach is overall similar to Bagoczki et al.

n	Circle			Sphere	
	CPU (s)		GPU (s)	CPU (s)	GPU (s)
	1 core	2 cores	256 cores	2 cores	256 cores
1	0.000	0.000	0.000034	0.000	0.000035
2	0.016	0.016	0.000036	0.015	0.00004
3	0.015	0.015	0.001071	0.188	0.0018
4	0.016	0.016	0.000049	0.032	0.0001
5	0.031	0.046	0.004135	2.703	0.0134
6	0.078	0.063	0.005870	7.516	0.0242
7	0.140	0.078	0.010647	18.750	0.05675
8	0.016	0.016	0.000098	0.390	0.000627
9	0.312	0.204	0.019143	72.609	0.1624
10	0.406	0.235	0.028382	124.656	0.2489
11	0.593	0.359	0.033319	219.141	0.3049
12	0.765	0.469	0.054516	332.390	0.5652
13	1.000	0.594	0.053796	557.468	0.5919
14	1.281	0.766	0.069647	824.157	0.8735
15	1.625	0.969	0.081841	1069.800	1.0883
16	0.110	0.063	0.000297	23.094	0.00503
17	2.422	1.375	0.097214	2127.890	1.6199
18	2.782	1.562	0.109281	2915.480	1.8684
19	3.422	1.906	0.132101	3848.860	2.3156
20	3.719	2.094	0.136277	4481.250	2.7

Figure 2.11: *Speed-up achieved with Bagoczki et al.'s implementation - Credit to* [75]

(Figure 2.12) but offers several improvements, particularly in the way the search space is distributed among threads with a heuristic called "adaptive partionning strategy".

Finally, another active community in the area of GPU-accelerated intervals comes from the Julia Programming language with the *JuliaIntervals* library [77]. This is essentially a wrapper around the CUDA C library in a more modern language which provides efficient higher-level primitives for intervals. The package has been used in several works such as [78, 79].

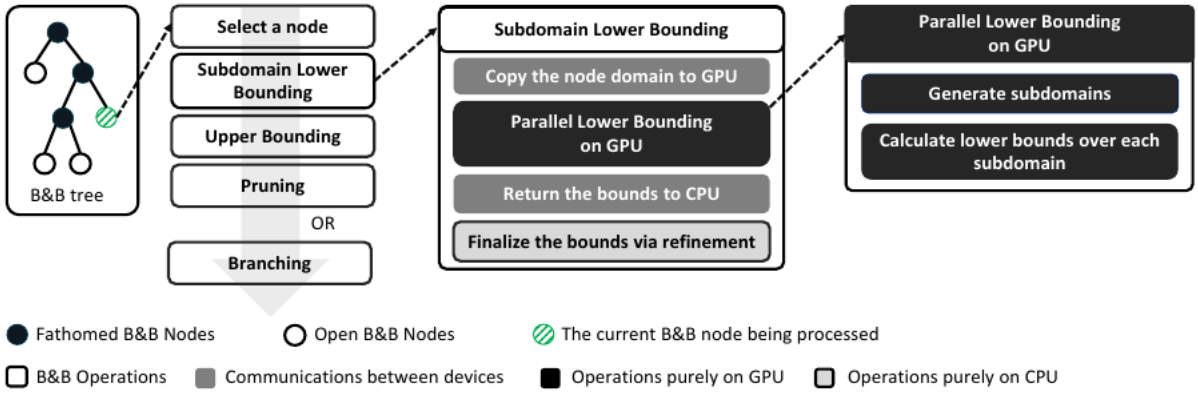


Figure 2.12: *Parallel B&B algorithm used by Zhang et al.'s - Credit to* [76]

2.3 Contractor-Level Acceleration

2.3.1 Using SIMD Instructions

Several studies have attempted to implement efficient interval algorithms using SIMD parallelism thanks to frameworks such as OpenMP [80, 81]. In particular, [82] shows how it can be used to accelerate interval matrix multiplications.

The Gaol library [48] proposes an implementation of elementary arithmetic interval operators based on the use of the SSE instruction set from x86 as a workaround for the inability of the architecture to switch efficiently between floating-point rounding modes. The idea is to take advantage of the symmetry between the round-down and round-up modes (refer to Equation 1.43) to evaluate both bounds with the same rounding mode and in parallel.

This technique can be applied to generate the interval version of every floating-point operator supported by the SSE extension. This is sadly not the case for recurrent real functions such as logarithm, exponential or sine/cosine.

2.3.2 Using Custom Hardware

2.3.2.1 Motivations

The use general-purpose hardware severely hinders the performance of interval operators. Some workarounds exist, such as the one used by Gaol and presented in Section 2.3.1 but they are limited to the set of arithmetic operators that are supported by hardware SIMD extensions.

The benefit of designing custom hardware is to address the structural bottlenecks of general-purpose architectures directly and propose a solution tailored for the needs of interval arithmetic. The defining features of such hardware should be:

- Parallel evaluation of interval bounds.
- Appropriate rounding for each bound at no extra cost.
- Precision of floating-point bounds suited for the target application.

The next Section presents the FPGA technology which is a popular support for designing custom hardware.

2.3.2.2 The FPGA Platform

A *Field-Programmable Gate Array (FPGA)* is a reconfigurable integrated circuit that can be programmed after manufacturing to implement custom hardware architectures. It mainly consists of an array of logic elements laid on programmable interconnects, as illustrated in Figure 2.13.

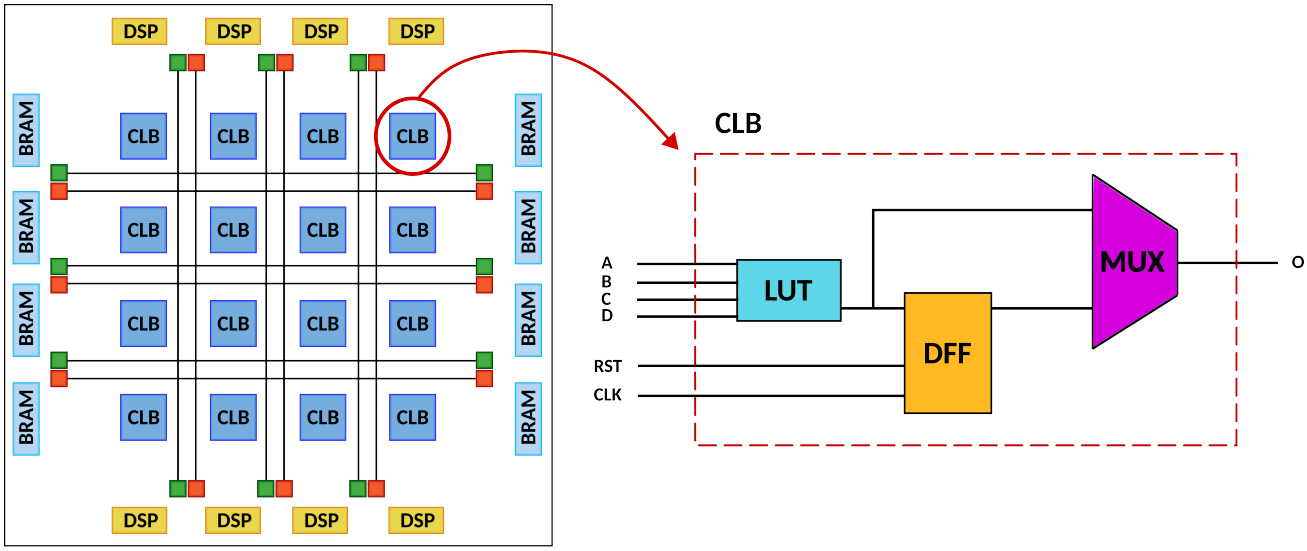


Figure 2.13: *Anatomy of a FPGA Chip*

Modern FPGA platforms typically include the following elements:

- **Configurable Logic Blocks (CLB):** elementary building blocks for combinatorial (LUTs and Muxes) and sequential logic (D flip-flops).
- **I/O Blocks:** configurable interfaces for other peripheral such as RAM, PCIe or GPIOs. Represented in red and green in Figure 2.13.
- **Block RAM (BRAM)** on-chip memory.
- **Digital Signal Processor (DSP):** dedicated arithmetic primitives (multipliers, ...).
- **Programmable Network:** configurable routing between logic elements.

A configuration bitstream is used to reconfigure the FPGA and implement the custom logic. More precisely, its role is to program the logic functions implemented by the LUTs, define how elements should be interconnected or initializes memory and DSP blocks.

To generate a bitstream, the design workflow described in Figure 2.14 is generally applied. It starts by a description of the desired circuit using *Hardware Description Languages (HDL)* such as Verilog or VHDL. The later defines abstractions to represent circuits and explain how they interact and exchange data together. The synthesis stage then parses this source code to infer the actual logic and map it into one of the hardware primives available on the FPGA (LUTS, DFF, ...) to create a gate level netlist. Thr implementation stage pushes this process further and attempts to assign specific physical locations for all the elements of the netlist to produce the desired bitstream. It deals with problematics such as managing the finite hardware ressources of the target FPGA or coming with a feasible routing between the designated elements. During the synthesis or the implementation, the developer is strongly advised to perform circuit simulation to debug the logic and ensure it works as expected.

The FPGA technology has several noticeable advantages which makes it a good candidate to implement dedicated interval primitives. A major factor is economical due to the reconfigurability which allows for cost-efficient and flexible prototyping compared to an ASIC chips, even if it comes at the price of a bit of performance. A major benefit of FPGAs compared to sequential instruction-based processors is to allow spatial parallelism. Complex hardware blocks, such as floating-point primitives, can be instantiated multiple times and truly run concurrently, as long as their implementation does not require more logic blocks than available. Another defining feature of this technology is that circuits implemented on a FPGA are explicitely defined, with predictable latencies and cycle-accurate. As a consequence, FPGAs are particularly well suited for the most critical applications such as real-time systems, signal processing and cryptography.

However, despite the advantages, FPGA development presents significant technical challenges. Hardware description and simulation are complicated processes which require profiency in a lot of vendor-specific tools and to master the underlying hardware. For example, The pool of logic blocks available on the chip must be carefully managed for an efficient utilization and poor architectural choices in the HDL stage may lead to logic congestion, resources overuse or routing failures. Another point of attention is related to timing closures. To generate the explicitely defined circuits mentionned above, each clock domain must satisfy appropriate specific time constraints and operate above a predefined maximum clock frequency. These limitations practically prevent from designing long-combinatorial paths which must therefore be broken into several steps separated by a clock tick.

For all the reasons described previously, the FPGA technology will be used to implement the custom interval acceleration primitives. This particularly due to its ability to perform

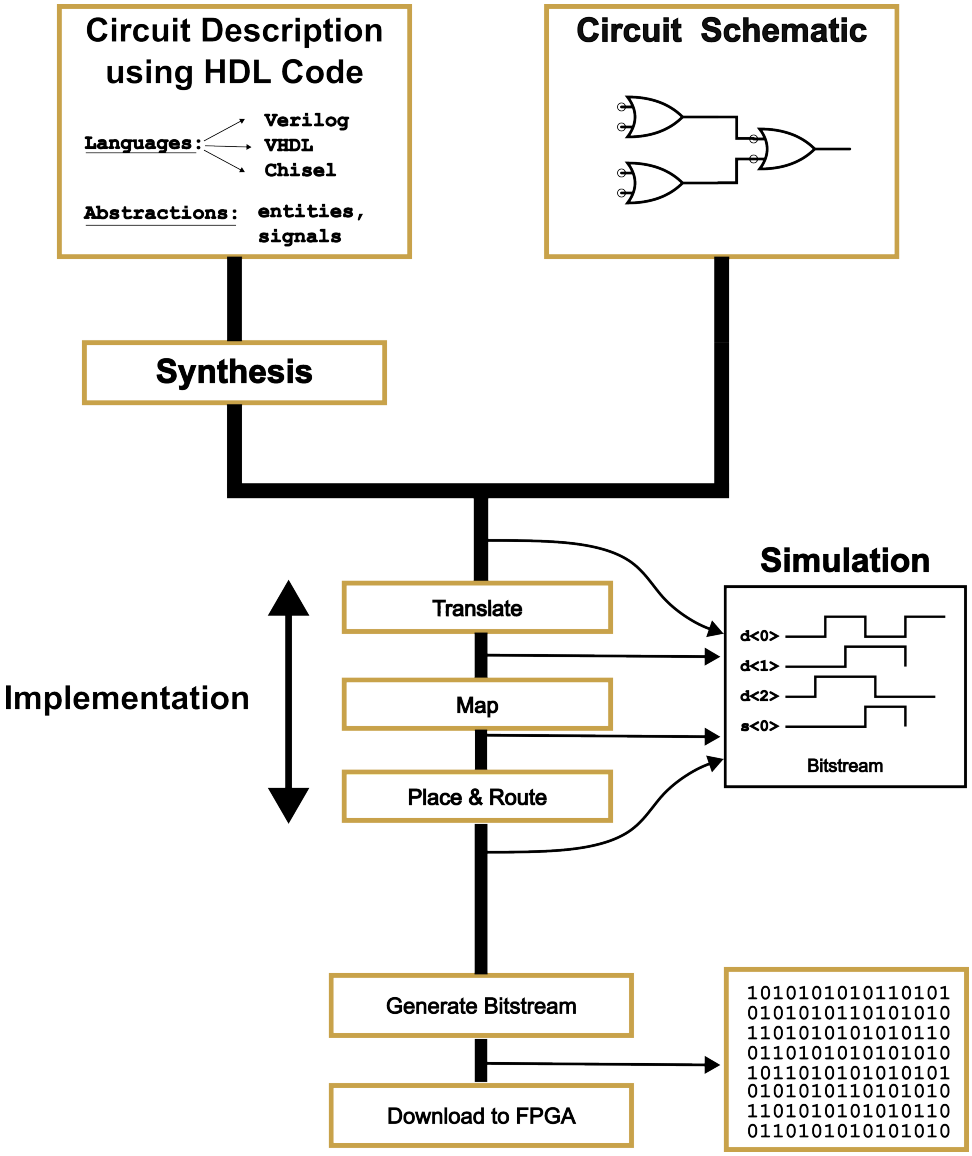


Figure 2.14: *The Traditional FPGA Design Workflow*

concurrent and correctly-rounded evaluations of floating-point operations on the lower and upper bounds of an interval.

2.3.2.3 Hardware Implementation of IEEE-754 Floating-Point Arithmetic

In the 1970s and early 1980s, floating-point arithmetic generaly behaved differently across computer systems. Hardware manufacturers were free to use their own representations and implemented precision, rounding and exception handling in their own way. This led numerical programs to be mostly non-portable and hard to verify [83].

In this context, the introduction of the IEEE-754 [84] in 1985 has been a major improvement. It not only defined a standard binary format for floating-point numbers but explicitely specified the behavior of notable arithmetic operations, rounding rules and ex-

ception handling, in an attempt to increase the interoperability of existing systems. The standard has known two major revisions in 2008 and 2019 but the core principles remain unchanged.

IEEE-754 represents real numbers using a scientific notation in base two (Equation 2.1) and the encoding of floating-point values i based on three fields:

$$(-1)^s \times m \times 2^e \tag{2.1}$$

where

- **s** is the *sign bit* which indicates a positive of negative number.
- **m** is the *mantissa* which encodes the precision of the number.
- **e** is the *exponent* which encodes the scale factor of the number. The width of this field has a direct impact on the range of representable values.

The standard introduces several pre-defined binary formats which offer various tradeoffs between width, precision and range:

	Single-Precision	Double-Precision	Quad-Precision
width	32	64	128
sign-bit	1	1	1
exponent	8	11	15
mantissa	23	52	112

Table 2.1: *Pre-Defined Binary Formats in IEEE-754*

As an example, Figure 2.15 illustrates the binary encoding associated to single-precision floating points.

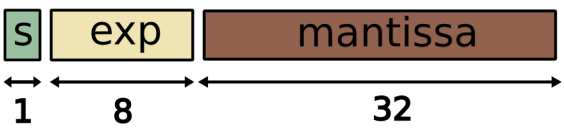


Figure 2.15: *IEEE-754 encoding For Single-Precision*

This representation allows to standardize the encoding of special values, as shown in Table 2.2, which is useful to represent overflow/underflow results or undefined computations.

	Sign	Exponent	Mantissa
zero	0 or 1	all 0	all 0
subnormal	0 or 1	all 0	non 0
infinity	0 or 1	all 1	all 0
NaN	0 or 1	all 1	non 0

Table 2.2: *Encoding of Special-Values in IEEE-754*

IEEE-754 specifies the behavior of the following arithmetic operations:

- Addition/Subtraction
- Multiplication/Division
- Square-root
- Fused multiply-add (FMA)
- ...

Each of these operators must produce a correctly-rounded result of the associated, which is hard to achieve in practice. Floating-point formats can represent only a finite subset of real numbers and most arithmetic operations produce results that require more precision bit than available in the chosen binary representation, for example the result of $0.1+0.2$ cannot be represented exactly in single-precision. As a consequence, the result must be rounded to one of the nearest representable numbers.

Several rounding modes are explicitely defined by the standard (Table 2.3) and corresponds to as many strategies for chosing the appropriate representable number in the vicinity of the real result. They are especially important in numerical algorithms such as interval arithmetic where error bounds and monotonicity properties must be controlled

Rounding Mode	Effect
round-to-nearest-even	return the nearest representable number
round to zero	apply truncation to the real result
round to toward $+\infty$	return the next representable number (toward $+\infty$)
round to toward $-\infty$	return the next representable number (toward $-\infty$)

Table 2.3: *Some of the Rounding-Modes Defined by IEEE-754*

To produce an accurate rounding of floating-point operators, FPUs compute intermediate results with a bigger mantissa than required by the chosen format and use the extra-precision to generate three special bits called *guard*, *round* and *sticky* which are used to generate the rounded mantissa. This process is illustrated by Figure 2.16.

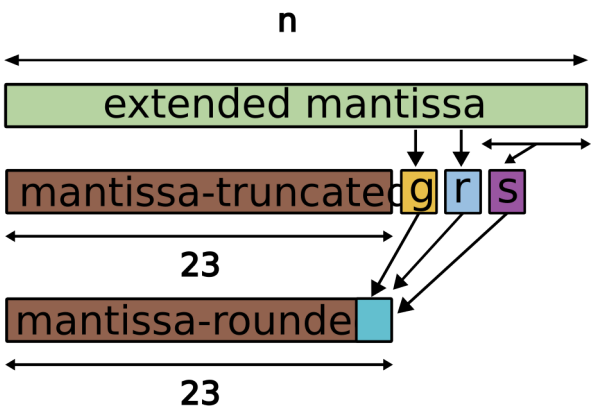


Figure 2.16: *IEEE-754 Rounding with Guard, Round and Sticky Bits*

Implementing floating-point arithmetic in hardware, particularly on a FPGA, is notori-

ously more challenging [85, 86] than using fixed-point for various reasons.

First of all, floating-point operators require multiple processing stages (see below) resulting in large and often irregular datapaths. This problem is addressed through aggressive pipelining which is which a very error-prone process in hardware design.

- Exponent comparison
- Mantissa alignment
- Arithmetic operation
- Normalization
- Rounding
- Exception handling

Normalization and rounding stages are especially complex. The former requires the instantiation of large shifters and leading-zero detectors that are very expensive in terms of hardware resources on a FPGA while the later manages large bit vectors (refer to Figure 2.16) which negatively impact the maximum operating frequency.

Floating-point operators play a huge role in interval arithmetic and it thus is absolutely necessary to have a complete control over this layer when attempting to design a dedicated accelerator for intervals. However, all the aforementionned arguments regarding the complexity of hardware floating-point advocate in favor of re-using existing implementations to limit the risk of error.

In the context of this thesis, the FloPoCo project [87, 88], a very popular solution in the literature [89, 90], is the main source of floating-point primitives. Rather than providing a fixed library of IP blocks, it acts as a parametrized hardware generator which produces VHDL code, with the following features:

- A large catalog of primitives, from arithmetic operators to transcendental functions
- Generation of synthesizable VHDL entities, pipelined to user's requirements
- Respect of IEEE-754 behaviors
- Optimized primitives for FPGA-specific structures (DSPs, LUTs, BRAM)
- Support for arbitrary precision formats
- Correctly-rounded operators

Practically, the tool requires several input constraints to produce pipelined operators tailored to the user's requirements, as shown in Figure 2.17.

Due to its ability to generate correctly-rounded floating-point primitives, possibly in arbitrary precision formats, FloPoCo is a perfect candidate for the upcoming hardware implementation of a dedicated interval architecture.

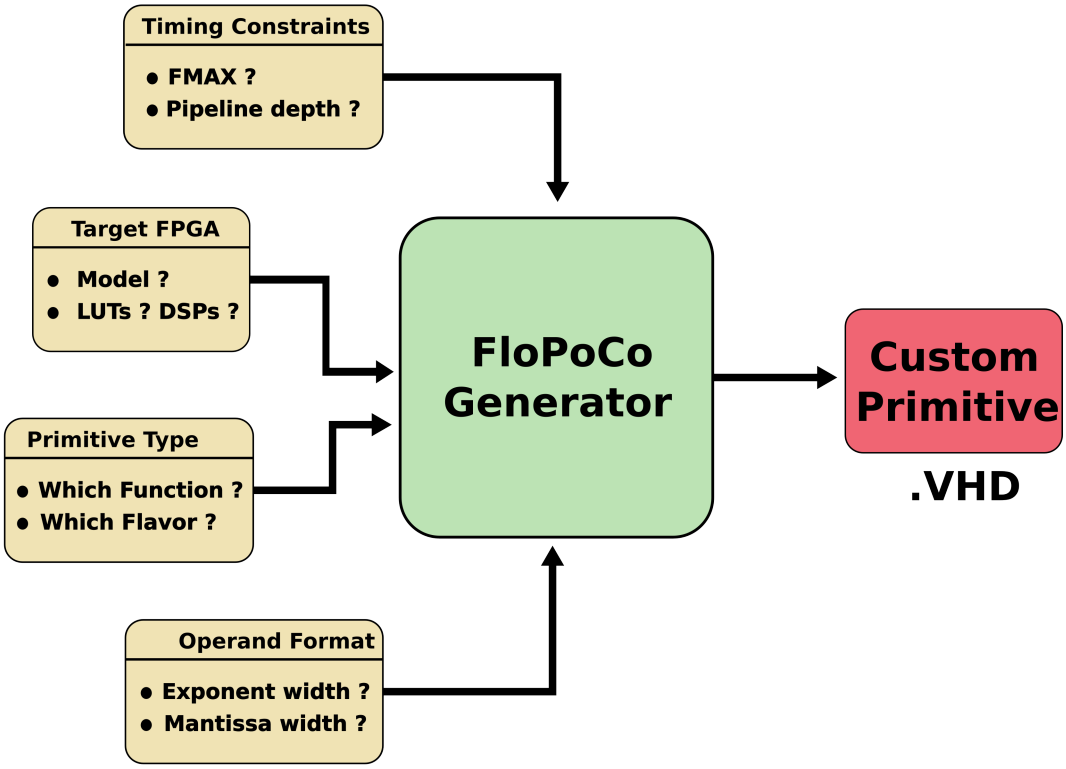


Figure 2.17: *FloPoCo, a Parametrized Floating-Point Code Generator*

2.4 Conclusion

In this Chapter, we decided in favor of a custom hardware strategy where interval operators are implemented direcly on a FPGA. The next challenge is to find the most appropriate way to expose the hardware to the software developper for him to create interval applications. The traditional processor model seems like a sensible choice as it is well understood by most programmers and offers a lot of flexibility. However, popular architectures are mostly proprietary and do not allow the addition of custom extensions, with the notable exception of RISC-V, which is the topic of the next Chapter.

3

The RISC-V Architecture

Contents

3.1 Motivations for the RISC-V Architecture 73

3.1.1 Early Processor Architectures 73

3.1.2 Transition from CISC to RISC 74

3.1.3 Limitations of Legacy ISAs 76

3.1.4 Origin and Objectives of the RISC-V 78

3.2 Fundamental Aspects of RISC-V Architecture 79

3.2.1 Core Architectural Design 79

3.2.2 The Modular Extension System 82

3.3 Conclusion 85

3.1 Motivations for the RISC-V Architecture

This section summarizes the historical context and the technical problems which encouraged the development of the popular *Reduced Instruction Set Computer (RISC)* architectures. An analysis of the limitations met by early RISC *Instuction Set Architectures (ISAs)* is then conducted followed by a discussion about the solutions brought by the RISC-V standard.

3.1.1 Early Processor Architectures

In the 1970s, most commercial processors adopted what is now called as the *Complex Instruction Set Computer (CISC)* philosophy with prominent examples such as the VAX-11/780 (1977) [91], the Intel 8086 (1978) [92] or the the Motorola 68k (1979). One

of the fundamental characteristics of these architectures lies in the size of the underlying instruction sets. These typically consisted of hundreds of instructions, often replicated for different addressing modes, which implemented complex functionalities such as memory-to-memory operations. Decoding the compiled instructions was also difficult, mostly because of variable-length instructions, between one and fifteen bytes for x86, which required complex fetching logic and practically limited pipelining due to unpredictable instruction boundaries. In addition, many instructions were implemented as sequences of micro-operations, called microcode, which were stored in an embedded ROM. This technique allowed to implement the most complex instructions without exponentially increasing the hardware complexity.

In summary, most of the complexity of processors at that time was handled by the hardware. Although this approach may seem outdated from today's perspective, it was in fact perfectly suited to the technical constraints of this period. A primary argument in favor of CISC is that memory was much more expensive compared to today, whether in terms of RAM or storage media (floppy disks, ...). Reducing program sizes was a major concern that was solved by using highly specialized instructions with the most compact encoding possible. A second argument lies in the relative immaturity of compilers which could only perform limited optimizations and were unable to restructure code for performance. To escape the complexity of software tools, ISA designers chose to integrate high-level constructs directly into hardware, using a microcode-based approach. This technique brought a lot of flexibility in the implementation of complex ISAs and even allowed post-fabrication bug patching, with the downside of increasing the average cycles per instruction.

3.1.2 Transition from CISC to RISC

3.1.2.1 Emergence of the RISC Architecture

Starting from early 1980's, several factors have challenged the hegemony of the CISC model [93]. Due to rapid improvements in manufacturing processes, the density of transistors available on a chip kept rising, effectively shifting the major bottleneck from memory size to execution speed. This trend allowed RTL designers to use more registers, build deeper pipelines and venture into parallelism at a reduced cost. During the same period, several projects proposed innovative architectures that layed the foundations for the RISC philosophy.

The *IBM 801* [94](1975) project aimed at creating a minimalist ISA by eliminating everything superfluous in previous CISC model. The decision to keep or discard a particular instruction was made based on statistical analysis of compilation output. The architecture was optimized for efficient pipelining with most instructions requiring only one clock cycle and used a large register file to reduce memory traffic. A novelty of this approach

was to give a greater importance to the compiler to compensate for the fewer and overall simpler instructions. This study demonstrated that a simple load/store architecture with multiple registers and paired with an aggressive pipeling strategy generally outperformed complex CISC designs.

The *RISC-I* [95] (1982) introduced the term RISC for the first time. It was a research project carried out at the UC Berkeley which extended IBM 801 ideas using VLSI technolgy and attempted to prove that a reduced instruction set could lead to higher clock frequencies and a more efficient silicon usage through simpler control logic.

The *MIPS* (1985) [96] was another research work from the Standford University and attempted to design an efficient processor based on a simple and fast pipeline. One of the core design principle pushed by this implementation was to rely on the compiler to detect hazards in the pipeline and insert NOPs when necessary to mitigate them.

Figure 3.1 presents a timeline of the most notable computer designs since the 1950's. It shows a trend where RISC-based designs have risen in popularity since the 1980's and are likely there to stay.

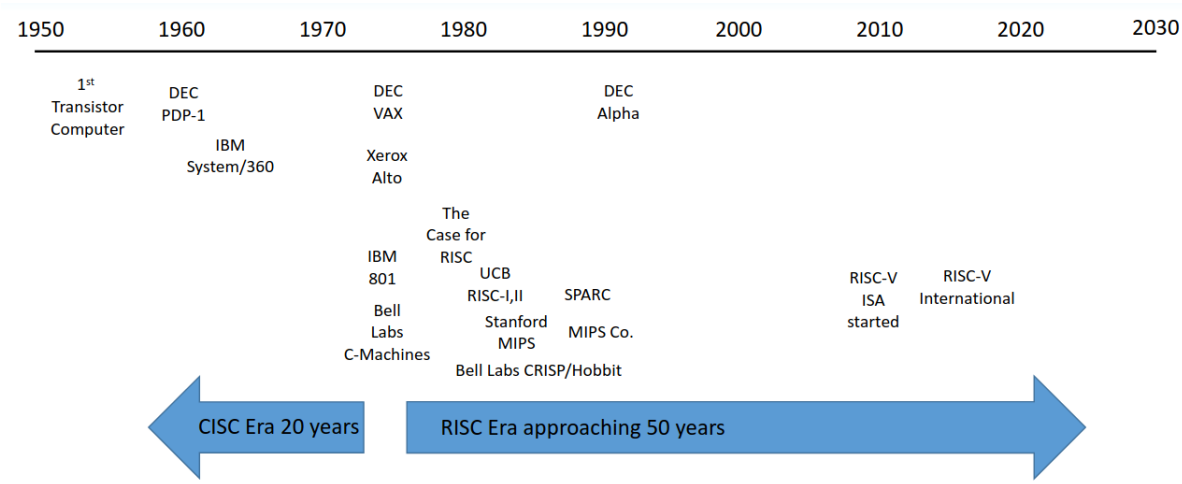


Figure 3.1: *Timeline of Notable Computer Designs - Credit to* [97]

3.1.2.2 Core Design Principles of RISC

The preceding examples have highlighted some of the fundamental characteristics of the RISC philosophy [98]. The main ones are detailed below:

- **simple instruction set:** RISC architectures favor ISAs made of a small number of simple instructions, with minimal support for exotic addressing modes. The principle is born from the assumption that complex instructions can be decomposed into sequences of simpler operations.
- **load/store architecture:** RISC processors separate computations from memory accesses, which are only served though load and store instructions. This design

choice keeps the datapath simple, reduces memory bottlenecks and greatly facilitates pipelining.

- **easy decoding:** Most RISC ISAs use fixed-width instructions, which has the advantage of simplifying the fetching and decoding processes. The predictability of instruction boundaries also helps with the pipelining.
- **single-cycle execution model:** RISC architectures emphasize deep pipelining as a mean to reach high clock frequencies. Most instructions require only one clock cycle to complete.
- **workload deported to the compiler:** A core principle is that the complexity should be shifted from the hardware to the compiler. The latter becomes responsible for instruction scheduling, register allocations and code optimization.

3.1.3 Limitations of Legacy ISAs

Over the years, most commercial ISAs, whether built on a CISC or RISC basis, have encountered technical limitations, which are detailed in this Section.

3.1.3.1 The Burden of Backward Compatibility

Most long-lived ISAs have a policy of binary backward compatibility which can take various shapes. This strategy is economically advantageous for both hardware designers and software developers, as the former can stop producing older processors while the latter do not need to update their code. ISAs usually keep growing over the years as new needs appear, practically leading to long-term architectural accumulation where new features are appended rather than redesigned.

The x86 is the textbook example of this phenomenon [99]. Thus, a modern processor still supports all the obsolete execution modes released over the last 40 years, such as the real mode (Intel 8086), the protected mode (Intel 80286) and the compat mode (32-bit programs on 64-bit systems). At the ISA level, multiple historical additions have been made:

- **x87:** floating-point stack.
- **streaming SIMD extension (SSE):** 128-bit SIMD instructions, implemented through several revisions: SSE $\rightarrow$ SSE2 $\rightarrow$ SSE3 $\rightarrow$ SSE4.
- **advanced vector extension(AVX):** Vector instructions, once again implemented through several revisions: AVX $\rightarrow$ AVX2 $\rightarrow$ AV-512.
- **virtulization technology (VT-X)** Support for virtualization.
- **and much more:** cryptography, security, ...

None of the old SIMD formats have been removed from modern processors, even though they are mostly redundant with newer extensions. The same thing happens for rarely

used legacy features such as BCD arithmetic or memory segmentation. This architectural accumulation directly explains the complexity of the x86 platform in terms of decoding logic, microcode engine, silicon area and validation efforts.

The ARM architecture, despite being RISC-based, also suffers from the problem of accumulation but handles it differently. It defines several architectural versions (e.g. ARMv4, ARMv5, ...) which each support one to several ISA extensions among the available (ARM-32, Thumb, Thumb-2, AArch32, ...). Each new version may choose to drop support for obsolete extensions, like ARMv8 [100] which introduced AArch64, a redesigned 64-bit ISA, and marked AArch32 as optional. In this context, hardware designers were free not to implement 32-bit (AArch32) at all, which can be seen as a deliberate architectural reset to simplifly the accumulated bagage of ARMv7. Many SoCs vendors practically kept implementing 32-bit at the release of ARMv8, mostly due to commercial constrainsts (Android used mostly 32-bit applications), but as more and more software stopped depending on it over the years, it progressively disappeared. The architecture cleans itself following market demands, which makes a huge difference with x86 where features are here forever by design.

3.1.3.2 Licensing and Governance Constraints

A major structural limitation of legacy ISAs, which is not technical but rather institutional, is due to proprietary control.

Following historical cross-licensing agreements, Intel and AMD exclusively own the patents covering the x86 ISA [101]. It prevents any third company from implementing its own design unless given explicit permission by right-holders. Similarly, Arm Holdings is the proprietary of the ARM architecture and the only entity allowed to modify its ISA. It offers paid-licenses to companies who want to implement ARM processors, sometimes allowing them to customize the underlying microarchitecture (e.g. Apple Silicon [102]), but enforces the strict respect of the ISA.

It can be argued that governance centralization is a problem in itself, since the evolution of commercial ISAs is driven by corporate strategies, rather than an open consensus. Research proposals may not align with vendor priorities and some interesting ideas may remain unpublished due to the lack of legal framework.

This situation creates a high entry cost for startups due to licensing and royalties fees which practically discourages competition. Moreover, licensed companies are dependent of a central authority, whether it be Intel, AMD, Arm Holdings or others, and are vulnerable to the slightest change in exploitation policy. Finally, ISAs being locked by licensers creates a barrier to innovation, especially coming from the academic world where budgets are typically much lower.

3.1.3.3 Restricted Extensibility

Legacy ISAs were not designed for long-term extensibility but to solve engineering problems from the 1980's and design decisions taken at that time still impact the potential for growth.

Some of the reasons are technical and are mostly linked to the phenomenon of architectural accumulation described in Section 3.1.3.1. The extensibility of old platforms is often more reactive (e.g. SIMD instructions from x86) than structural and several issues have not been properly anticipated. For example, x86 has no mechanism to separate the base ISA from optional features and the opcode space is now densely packed with little room left for new instructions. Early ARM releases did not have any concept of namespace extensions and were not designed to cleanly incorporate 64-bit. To this day, there is still no way to automatically discover optional features.

Other barriers to extensibility are of legal nature, as illustrated in Section 3.1.3.2. For x86, only Intel and AMD are allowed to insert new instructions in the ISA. For ARM, vendor-extensions can be implemented by companies with a costly architectural license as long at it does not break the compatibility with official extensions.

3.1.4 Origin and Objectives of the RISC-V

The RISC-V standard [103] is an ISA specification which started in 2010 at the UC Berkeley. This initiative was motivated by the fact that most existing ISAs, such as x86, ARM, MIPS and others, were proprietary and made it difficult to teach computer design to students, experiment with innovative architectures and openly publish research papers. As the evolution of legacy ISAs had introduced several structural limitations over the years (refer to Section 3.1.3), the RISC-V project chose to adopt a clean-slate ISA strategy and rethink entirely some architectural foundations.

It is built around the following design principles:

open-standard: the ISA specification is open and delivered to anyone willing to implement it without licensing or royalties fees, making it far more accessible than any existing ISA for academic or commercial uses. RISC-V is governed by a non-profit consortium (RISC-V International) where architecture evolutions are publicly debated and ratified through various working groups. This prevents hardware-vendor from influencing ISA directions too much which addresses long-term governance risks.

simplicity: the architecture is built following the classic RISC philosophy and deliberately avoids complex constructs to make the hardware simpler.

modularity: the standard is build with modularity in mind. It revolves around a small frozen ISA which can be complemented at will with optional extensions.

extensivity: To circumvent the potential limitations of standard extensions, the RISC-V specification provides a mecanism to insert vendor extensions.

3.2 Fundamental Aspects of RISC-V Architecture

The RISC-V International publishes and maintains the official RISC-V specification [104, 105], which serves as the normative reference for the architecture. The goal of this document is to define the ISA and precisely establish the boundaries between hardware and software to ensure interoperability across implementations. To achieve this, a strong focus is made on the explicitation of the behavior of instructions, registers, memory access or the interactions between the available privilege levels. The main benefit of this clear separation between the ISA and the microarchitectural implementation is to allow designers to innovate internally as long as they remain compliant at the ISA level. This Section briefly reviews the key design points of the RISC-V architecture in order to justify the use of this platform in the work carried out in this thesis.

3.2.1 Core Architectural Design

3.2.1.1 Base ISA

As per standard specification, RISC-V processors must implement thirty-two general-purpose registers with a width of XLEN-bit, that can be set either to 32, 64 or 128-bit. This flexibility allows to adapt to a wide range of applications, from cryptographic tasks which may take full advantage of 128-bit registers to embedded applications which are fine with only 32-bit registers. Figure 3.2 presents the standard registers as well as the default calling convention used by C programs.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Figure 3.2: *RISC-V General-Purpose Registers and Associated Calling Convention*

The minimal set of instructions that any implementation must support then forms the base ISA, denoted as RV32I, RV64I or RV128I depending of the XLEN chosen. These instructions can be broadly separated in the following categories:

- Logic/Arithmetic
- Load/store
- Branching
- System call
- Memory synchronization
- Hardware Breakpoints

Figure 3.3 lists all the instructions required in the base ISA when XLEN=32 and shows how minimalist a RISC-V implementation can be.

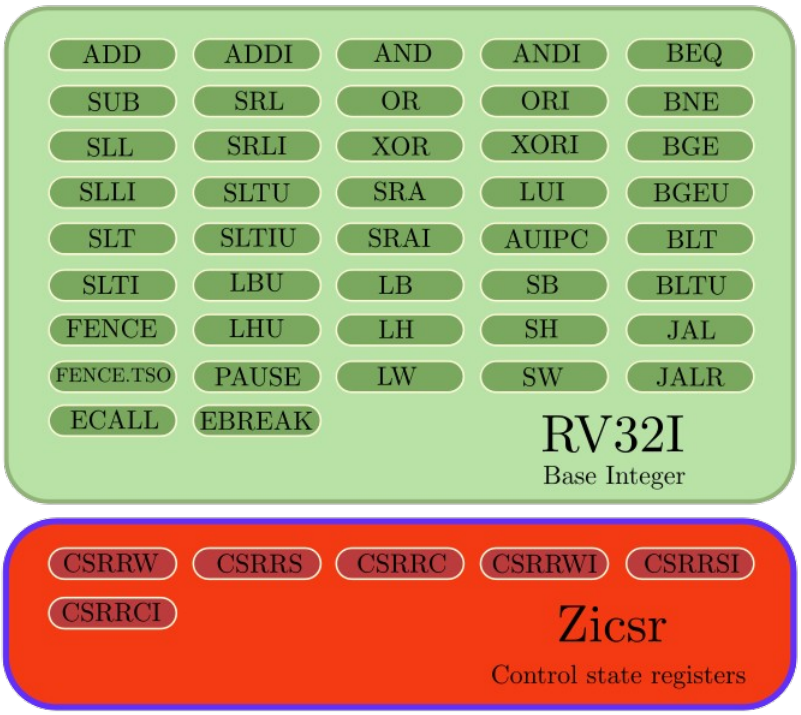


Figure 3.3: *Overview of the RV32I and Zicsr Extensions*

Practically, the base ISA is most of the time complemented with the satellite extension *Zicsr* which adds dedicated instructions (CSRRW, CSRRS, CSRRC, ...) to interact with the *Control and Status Registers (CSRs)* in an atomically.These additional registers, totalling 4096 entries, are designed to configure, control, and observe the processor's behavior beyond the basic instruction execution model. They are used for various purposes:

- Implementation of privilege levels.
- Exception and interrupt management. When a trap occursm the processor automatically updates specific CSRs with information related to the event, such as the cause and the associated program counter.
- Memory management and protection. This includes features required by modern

operating systems, such as memory paging or physical memory protection.
- Discovery of supported hardware extensions. Refer to Section 3.2.2
- Performance monitoring through various hardware counters. It is used to measure execution time, cache events and other microarchitectural statistics.
- Events linked to the floating-point unit, if supported.
- ...

3.2.1.2 Instruction Encoding

One of the priorities of the RISC-V standard, in line with the RISC philosophy, is to allow the decoding of compiled instructions in the simplest way possible. This is achieved through a clever encoding system in which instructions can only adopt one of the formats displayed in Figure 3.4.

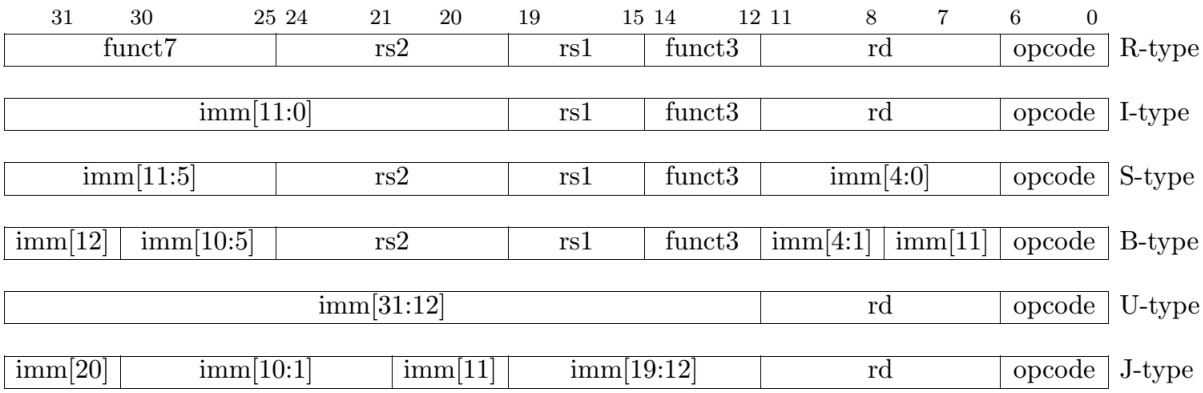


Figure 3.4: *Valid RISC-V Instruction Formats*

The valid instruction formats are denoted by a unique letter which suggests the most appropriate usage for each of them:

- **R-type:** register-register operations.
- **R4-type:** special case of R-type, the funct7 field is partially replaced by a third source register rs3. This is mostly used for FMA instructions.
- **I-type:** short-immediate and load operations.
- **S-type:** store operations.
- **B-type:** conditional branch operations.
- **U-type:** long-immediate operations.
- **J-type:** unconditional branch operations.

A major benefit of this approach is that instructions are all 32-bit long, which greatly facilitates fetching and eliminates the need for complex pre-decoding logic as found on x86. For the same reason, the valid formats rely on a set of internal fields, listed below, which are layed out in a consistent way across formats. It is possible to retrieve the meaningful parameters of each instruction using the same piece of circuit.

- **opcode:** unique 7-bit identifier which points to a family of instructions.
- **funct3/funct7:** additional parameters (respectively 3-bit and 7-bit long) to uniquely identify an instruction. They are meant to be used in combination with the *opcode* field.
- **rd:** id of the destination register in which the result of the current instruction will be placed.
- **rs1/rs2:** id of the sources registers which hold the input parameters of a specific instruction.
- **imm:** an immediate value (constant) passed as parameter of a specific instruction.

As an additional optimization, the encoding ensures that for a give opcode value, all the associated instructions will share the same format as showed in Figure 3.5, which allows for efficient and early routing to the right portion of decoder.

opcode[6:5] \ opcode[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	OP-V	CUSTOM 2	48b
11	BRANCH	JARL	reserved	JAL	SYSTEM	OP-VE	CUSTOM 3	$\geq$ 80b

I-Type U-Type R4-Type B-Type Vector-specific
S-Type R-Type J-Type Custom Reserved

Figure 3.5: *RISC-V Opcode Space and Format Association*

3.2.2 The Modular Extension System

Beyond the ISA base, which is mandatory in any implementation, the RISC-V standard is built around the concept of extension. Each extension is identified by a single letter, "I" in the case of the base ISA (hence the name RV32I), and consists of a list of thematically related instructions used to accelerate specific operations or support a particular programming paradigm. Their integration into a RISC-V processor design is entirely optional and is left to the discretion of the hardware vendor, depending on the target application or physical constraints such as power consumption, silicon area, etc. This flexibility allows the creation of tailor-made processors perfectly suited to specific needs, which helps to limit production costs and time to market.

In this context, the term "RISC-V processor" becomes vague, as it can refer to a minimalist processor designed for embedded systems with only the basic ISA, or, conversely, to a processor sized for HPC with support for vector and double-precision floating-point operations. In response, the standard has established a nomenclature, given in Figure 3.6, that allows the extensions supported by a given design to be precisely identified. As a consequence, a RV64IMA RISC-V processor uses 64-bit registers and implements the base ISA I as well as standard extensions M and A.

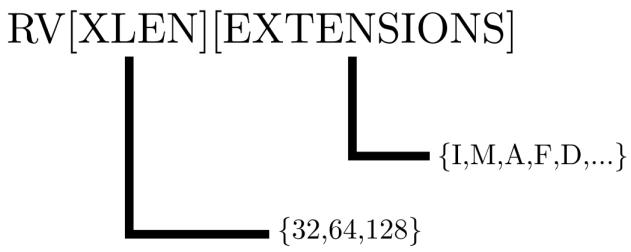


Figure 3.6: *Nomenclature to Identify a RISC-V Processor*

3.2.2.1 Standard Extensions

Standard extensions are those that have been officially ratified in the RISC-V International standardization document. As such, the behavior of the underlying instructions is explicitly detailed, and each modification is tracked using a versioning mechanism. As explained previously, they are entirely optional and are intended to provide additions to the ISA base in specific areas. Figure 3.7 lists the main standard extensions.

The purposes of the meaningful standard extensions in the context of this thesis are detailed below:

- **M:** additional instructions for hardware integer multiplication, division and remainder operations.
- **A:** additional instructions for atomic memory operations and synchronization primitives based on *load-reserved* and *store-conditional*.
- **F,D,Q:** additional instructions for IEEE-754 floating-point operations, respectively in single (32-bit), double (64-bit) or quad (128-bit) precision. When either of these extensions are instantiated, the standard requires to define thirty registers dedicated to floating-point computations (Figure 3.8), with a width corresponding to the chosen format. It is also possible to support multiple floating-point extensions (e.g. F and D). The associated registers are then instantiated with the biggest width (here 64-bit) and the smallest type occupies the lower part of the registers.

As an example of the RISC-V modularity, Figure 3.9 presents an exhaustive list of the instructions supported by a RV32IMAF processor.

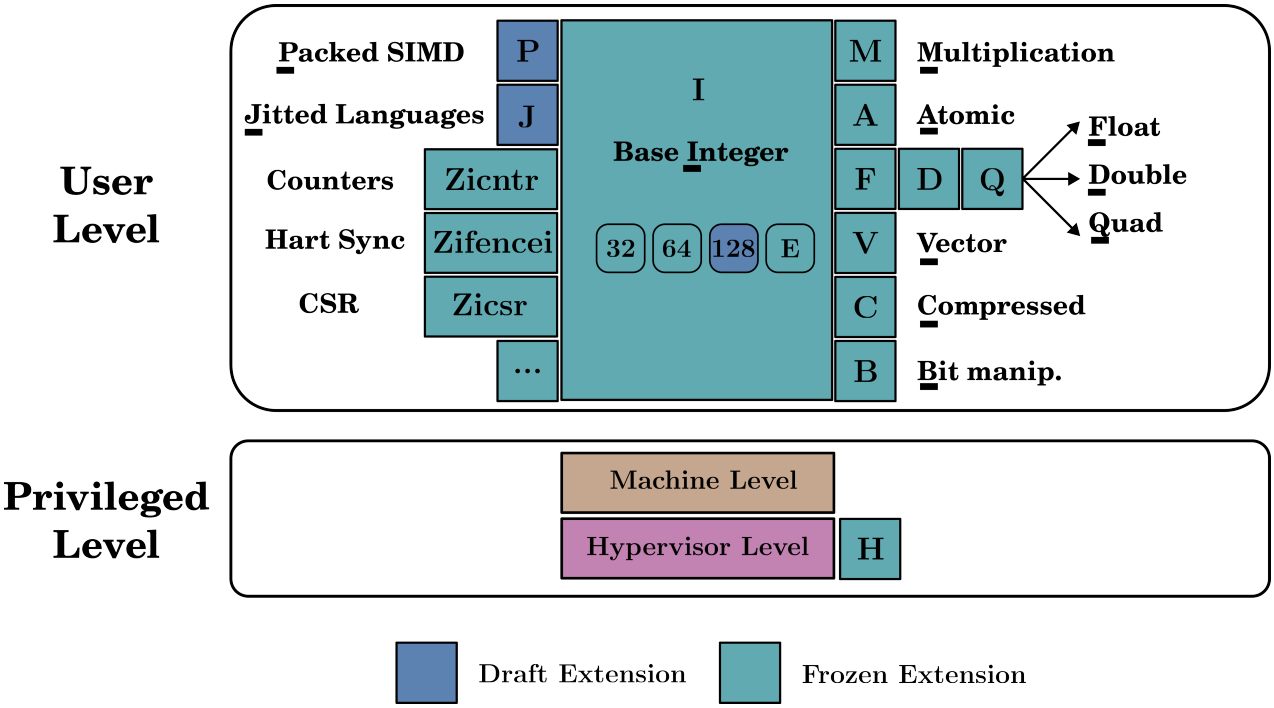


Figure 3.7: *Main RISC-V Standard Extensions*

f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figure 3.8: *RISC-V Floating-Point Registers and Associated Calling Convention*

3.2.2.2 Custom Extensions

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 standard with the concept of custom extensions.

Figure 3.5 shows that four opcodes, named *custom-0* to *custom-3* (in red), have been left unutilized and are the suggested insertion point for specialized instructions. This is meant to be a clean interface which allows the cohabitation between standard and custom extensions while maintaining a high inter-operability and without the need to fork the ISA. In accordance with the RISC-V philosophy, a range of CSR registers is also dedicated to custom extensions as a mean to allow extra configuration.

This mechanism is often used by hardware-vendors as it typically involves less complexity and thus time-to-market to develop efficient application-specific processors. For example, *Andes Technology* offers a toolchain that let designers define and integrate cus-

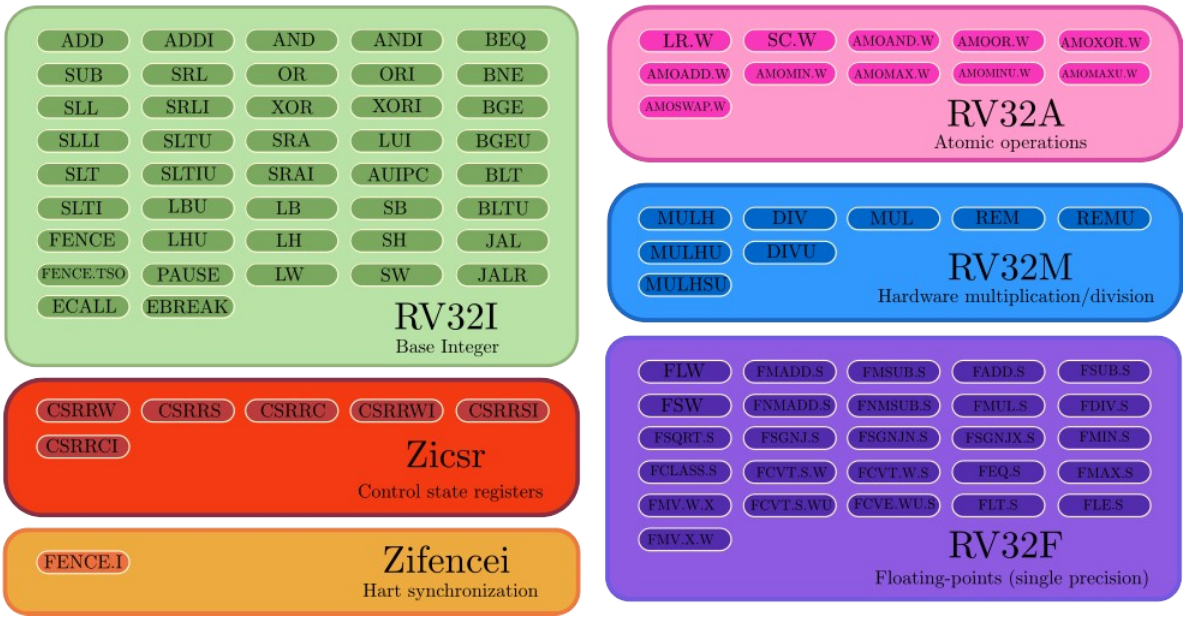


Figure 3.9: *Instructions Supported by a RV32IMAF Core*

tom instructions to optimize performance for specific application domains [106]. Similarly, Sifive has develop a custom extension to accelerate int8-based matrix multiplication for AI [107]. In the academic world , custom extensions have been used successfully to create hardware accelerators [108], particularly in the field of AI [109], and to develop additional security mechanisms [110].

3.3 Conclusion

The RISC-V standard is a modern look on RISC design principles and attempt to provide solutions to well-documented issues of legacy ISAs. The problem of architectural accumulation is addressed in a more structured way thanks to a fully modular approach to ISA design and extensibility is anticipated from the start using a dedicated encoding space and mecanism for inserting new instructions. Governance and licensing issues are solved thanks to RISC-V being an open and royalty-free standard governed by a non-profit organization.

The RISC-V standard has made ISA design more accessible than ever. This is not only due to its openness but also to the freely available software ecosystem made of compilers, emulators or softcores which give great opportunities for experimentation, especially in the field of research. In the context of this thesis, the natural extensibility of the RISC-V will be used to incorporate a custom extension dedicated to interval operators to execute robotics-oriented applications.

Part II

Contributions

4

Accurate Generation of the Complementary of a Contractor

The work described in this chapter has been published in the following paper:

Contributions

► Pierre Filiol, Theotime Bollengier, Luc Jaulin and Jean-Christophe Le Lann. "A New Interval Arithmetic To Generate the Complementary of Contractors". Acta Cybernetica 26 (2024) 817-838. DOI: `https://doi.org/10.14232/actacyb.300840`

Contents

4.1 Problem **90**

4.2 Approach **91**

4.3 Total extension **94**

4.3.1 Definitions 94

4.3.2 Illustration 95

4.3.3 Properties 96

4.3.4 Total real arithmetic 97

4.3.5 Link with the complex number i 98

4.4 Total intervals **99**

4.4.1 Intervals in unions of lattices 99

4.4.2 Total intervals 100

4.4.3 Total interval arithmetic 100

4.4.4 Total interval vector 101

4.5 Total contractors **101**

4.5.1 Total directed contractor for a binary constraint 101

4.5.2 Total directed contractor for a ternary constraint 104
4.5.3 Total forward-backward contractor 107
4.6 Conclusion 110

4.1 Problem

Chapter 1 showed how an interval contractor associated to a complex mathematical expression could be built proceduraly using the forward-backward methodology. The latter is based primarily on constraint propagation and accurate evaluation of interval contractors on elementary constraints. However, there are known limitations in existing solvers as soon as the domains of functions involved in the problem are restricted.

As an illustration, an inner and outter approximation of the set $\mathbb{X}$ defined in Equation 4.1 is computed using the Codac library [111] (refer to Figure 1.21) and the code presented in Listing 3.

$$\mathbb{X} = \{(x_1, x_2) \,|\, x_2 + \sqrt{x_1 + x_2} \in [1, 2]\}. \tag{4.1}$$

Listing 3: *Attempting to Characterize the set $\mathbb{X}$ using Codac and Forward-Backward Contractors*

```
from codac import *
from vibes import *

# Search space for (x1, x2)
X0 = IntervalVector([[-10,10],[-10,10]])

# Create the contractor
f = Function(x1,x2,x2+sqrt(x1+x2))
S = SepFwdBwd(f,sqr(Interval(1,2)))

# Run Sivia
vibes.beginDrawing()
SIVIA(X0,S,0.01)
```

The execution of the SIVIA algorithm leads to the paving displayed in Figure 4.1 where the blue boxes are supposed to be outside of $\mathbb{X}$ and the magenta boxes are supposed to be inside. However, when taking a closer look, some boxes are wrongly classified as *inside* when they are outside in reality.

This phenomenon occurs for all existing solvers which are able to provide an inner approximation. The reasons for this is that contractor-based methods obtain an inner ap-

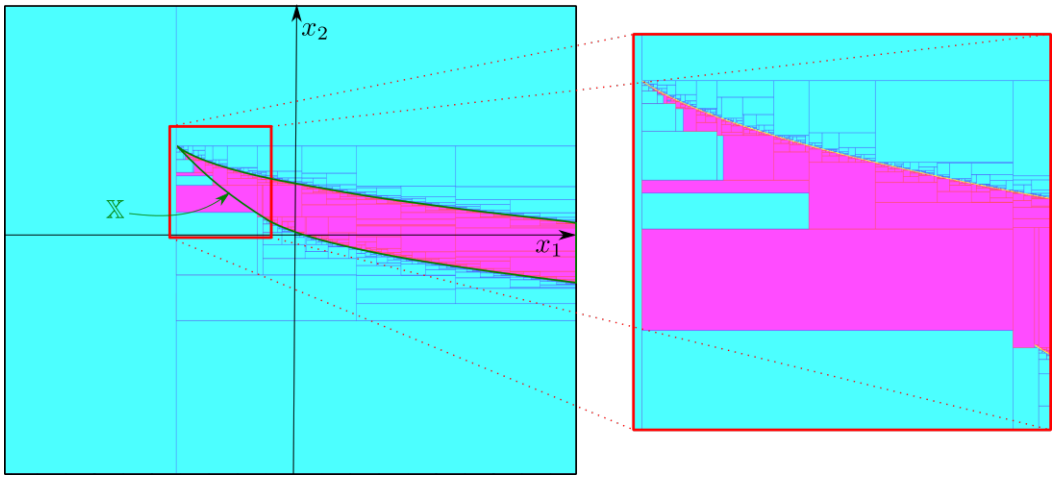


Figure 4.1: *Example of an Inaccurate Set Estimation in Codac. Left: Paving generated by Codac when the real set $\mathbb{X}$ is delimited by the green line. Right: zoom on the red box.*

proximation by considering a contractor for the complementary of $\mathbb{X}$ as

$$\{(x_1, x_2) \,|\, x_2 + \sqrt{x_1 + x_2} \notin [1, 2]\} \tag{4.2}$$

whereas it should be

$$\overline{\mathbb{X}} = \{(x_1, x_2) \,|\, x_2 + \sqrt{x_1 + x_2} \notin [1, 2] \text{ or } x_1 + x_2 < 0\}. \tag{4.3}$$

In Figure 4.1, some magenta zones are wrongly classified as *inside* because in these zones, $\sqrt{x_1 + x_2}$ is not defined.

This chapter presents a rigorous way to build contractors associated with the complementary of a set in the case where functions involved in the constraint are not defined everywhere. It is organized as follows. Section 4.2 explains the approach which motivates a new arithmetic. Section 4.3.4 presents an extension of the arithmetic on real numbers, named *total real arithmetic*, and shows the role of a flag named ι in the case where partial functions are involved. Section 4.4 introduces the notion of the total interval arithmetic. Section 4.4 provides the notion of total contractors and extends the classical forward-backward contractor to total intervals. Section 4.6 concludes the discussion.

4.2 Approach

The Contractor algebra presented throughout Chapter 1 does not allow any non-monotonic operation. It means that if a contractor $\mathcal{C}$ is defined by an expression $\mathcal{E}$ of other contractors $\mathcal{C}_i$ then we always have

$$\forall i, \mathcal{C}_i \subset \mathcal{C}_i^{'} \Rightarrow \mathcal{E}\left(\mathcal{C}_1, \mathcal{C}_2, \dots\right) \subset \mathcal{E}\left(\mathcal{C}_1^{'}, \mathcal{C}_2^{'}, \dots\right). \tag{4.4}$$

As a consequence, the complementary $\overline{\mathcal{C}}$ of a contractor $\mathcal{C}$ or the restriction $\mathcal{C}_1 \setminus \mathcal{C}_2$ of two contractors $\mathcal{C}_1, \mathcal{C}_2$ (which both correspond to non-monotonic operations) is not defined.

To be more precise, contractor algebra allows us to construct a contractor for expressions of sets defined by union, intersection and inversion of other sets. Take for instance the set

$$\mathbb{X} = \mathbb{X}_1 \cup \mathbf{f}^{-1}(\mathbb{X}_2 \cap \mathbb{X}_3). \tag{4.5}$$

The expression of the set $\mathbb{X}$ from Equation 4.5 can be represented by the tree of Figure 4.2 or equivalently by the following expressions

$$\begin{array}{rcl} \mathbb{X} & = & \mathbb{X}_1 \cup \mathbb{B} \\ \mathbb{B} & = & \mathbf{f}^{-1}(\mathbb{A}) \\ \mathbb{A} & = & \mathbb{X}_2 \cap \mathbb{X}_3 \end{array} \tag{4.6}$$

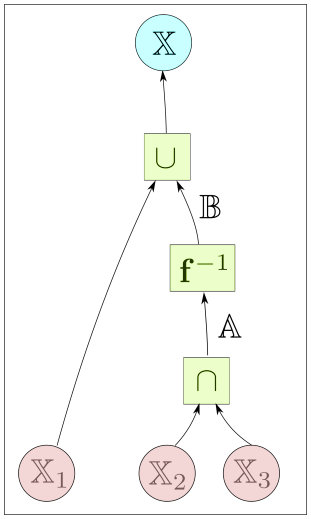


Figure 4.2: *Contractor tree for* $\mathbb{X}_1 \cup \mathbf{f}^{-1}(\mathbb{X}_2 \cap \mathbb{X}_3)$

The intermediate sets $\mathbb{A}$ and $\mathbb{B}$ correspond to nodes of the tree. In practice, the leaves $\mathbb{X}_i$ of the tree are *set reverse* (or equivalently *inequality constraints*) of the form

$$\mathbb{X}_i = \varphi_i^{-1}([\mathbf{y}_i]) = \{\mathbf{x}_i \mid \varphi_i(\mathbf{x}_i) \in [\mathbf{y}_i]\} \tag{4.7}$$

where φ_i is a function defined by an algorithm and $[\mathbf{y}_i]$ is a box of $\mathbb{R}^n$. A contractor for $\mathbb{X}_i$ is usually built by a forward-backward procedure such as the HC4R algorithm presented in Section 1.2.4.5. The contractor associated with the constraint $\varphi(\mathbf{x}) \in [\mathbf{y}]$ is denoted

by $\mathcal{C}^{\updownarrow}_{\varphi^{-1}([\mathbf{y}])}$ and follows the Algorithm 11.

Algorithm 11: Contractor for $\mathbb{X} = \mathbb{X}_1 \cup \mathbf{f}^{-1}(\mathbb{X}_2 \cap \mathbb{X}_3)$ - Obtained through HC4R

Data: $[\mathbf{x}]$
1 $[\mathbf{x}_1] = \mathcal{C}_{\mathbb{X}_1}([\mathbf{x}])$;
2 $[\mathbf{b}] = [\mathbf{x}]$;
3 $[\mathbf{a}] = \mathbf{f}([\mathbf{b}])$;
4 $[\mathbf{x}_2] = \mathcal{C}_{\mathbb{X}_2}([\mathbf{a}])$;
5 $[\mathbf{x}_3] = \mathcal{C}_{\mathbb{X}_3}([\mathbf{a}])$;
6 $[\mathbf{a}] = [\mathbf{x}_2] \cap [\mathbf{x}_3]$;
7 $[\mathbf{b}] = \mathcal{C}^{\updownarrow}_{\mathbf{f}^{-1}([\mathbf{a}])}([\mathbf{b}])$;
8 $[\mathbf{x}] = [\mathbf{x}_1] \sqcup [\mathbf{b}]$;
9 **return** $[\mathbf{x}]$

The *De Morgan's laws* are used to express the complement $\overline{\mathbb{X}}$. It states that:

- the complement of the union of two sets is the intersection of their complement.
- the complement of the intersection of two sets is the union of their complement.

We get

$$\overline{\mathbb{X}} = \overline{\mathbb{X}}_1 \cap \left(\mathbf{f}^{-1}(\overline{\mathbb{X}}_2 \cup \overline{\mathbb{X}}_3) \cup \overline{\mathrm{dom}\mathbf{f}}\right). \tag{4.8}$$

Note that we had to introduce the domain of $\mathbf{f}$, denoted by dom$\mathbf{f}$, to take into account the fact that $\mathbf{f}$ may be a partial function (*i.e.*, not defined everywhere).

If we define the set-valued function $\mathring{\mathbf{f}}^{-1} : \mathscr{P}(\mathbb{R}^m) \mapsto \mathscr{P}(\mathbb{R}^n)$ as

$$\mathring{\mathbf{f}}^{-1}(\mathbb{Y}) = \mathbf{f}^{-1}(\mathbb{Y}) \cup \overline{\mathrm{dom}\mathbf{f}}, \tag{4.9}$$

then we have

$$\overline{\mathbb{X}} = \overline{\mathbb{X}}_1 \cap \left(\mathring{\mathbf{f}}^{-1}(\overline{\mathbb{X}}_2 \cup \overline{\mathbb{X}}_3)\right). \tag{4.10}$$

The decomposition for $\overline{\mathbb{X}}$ is defined by

$$\begin{array}{rcl} \overline{\mathbb{X}} & = & \overline{\mathbb{X}}_1 \cap \overline{\mathbb{B}} \\ \overline{\mathbb{B}} & = & \mathring{\mathbf{f}}^{-1}(\overline{\mathbb{A}}) \\ \overline{\mathbb{A}} & = & \overline{\mathbb{X}}_2 \cup \overline{\mathbb{X}}_3 \end{array} \tag{4.11}$$

which corresponds to the tree of Figure 4.3.

Since the sets $\mathbb{X}_i$ were defined by $\varphi_i(\mathbf{x}_i) \in [\mathbf{y}_i]$, the complement is by

$$\overline{\mathbb{X}}_i = \mathring{\varphi}_i^{-1}(\overline{[\mathbf{y}_i]}). \tag{4.12}$$

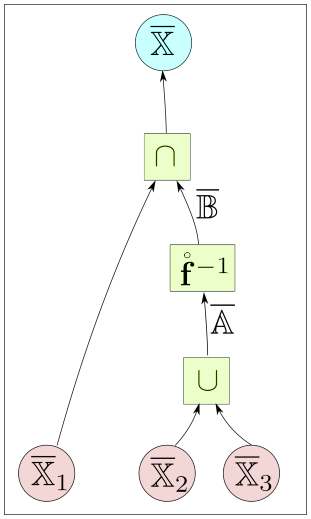


Figure 4.3: *Contractor tree for the complementary of* $\mathbb{X}_1 \cup \mathbf{f}^{-1}(\mathbb{X}_2 \cap \mathbb{X}_3)$

To implement the complementary of a contractor using the *De Morgan's law,* the only brick we need is the forward-backward contractor for the set

$$\mathring{\mathbf{f}}^{-1}([\mathbf{y}]) = \mathbf{f}^{-1}([\mathbf{y}]) \cup \overline{\text{dom}\mathbf{f}} \tag{4.13}$$

Now, the set $\mathring{\mathbf{f}}^{-1}([\mathbf{y}])$ is not a set reverse as defined by (4.7) and thus we cannot apply a forward-backward contractor without the extension proposed in Sections 4.3.4 to 4.4.

4.3 Total extension

4.3.1 Definitions

In mathematics, a function $f : X \mapsto Y$ which is defined for all $x \in X$ is said to be *total*. Equivalently, a function $f : X \mapsto Y$ is total if

$$\forall x \in X, \exists y \in Y \text{ such that } f(x) = y. \tag{4.14}$$

A function f which is not defined for all x is said to be *partial*. Given a partial function f, the *total extension* is obtained by adding an element to Y, say ι which collects all $x \notin \text{dom} f$.

Definition 1

The *total extension* of the partial function $f : X \mapsto Y$ is $\mathring{f} = X \cup \{\iota\} \mapsto Y \cup \{\iota\}$ with

$$\mathring{f}(x) = \begin{cases} f(x) & \text{if } x \in \text{dom} f \\ \iota & \text{otherwise} \end{cases} \tag{4.15}$$

Note that since $\iota \notin \text{dom} f$, we have $\mathring{f}(\iota) = \iota$.

4.3.2 Illustration

Consider the partial function f as given in Figure 4.4. We have

$$\begin{array}{rcl} f^{-1}(\mathbb{Y}) & = & \{\beta, \gamma, \varepsilon\} \\ f^{-1}(\overline{\mathbb{Y}}) & = & \{\alpha\} \\ \text{dom} f & = & \{\alpha, \beta, \gamma, \varepsilon\} \end{array} \tag{4.16}$$

Now, since $f(\{\gamma, \delta\}) \subset \mathbb{Y}$, some would classify δ inside $f^{-1}(\mathbb{Y})$ which is wrong. This would only be true if f was total.

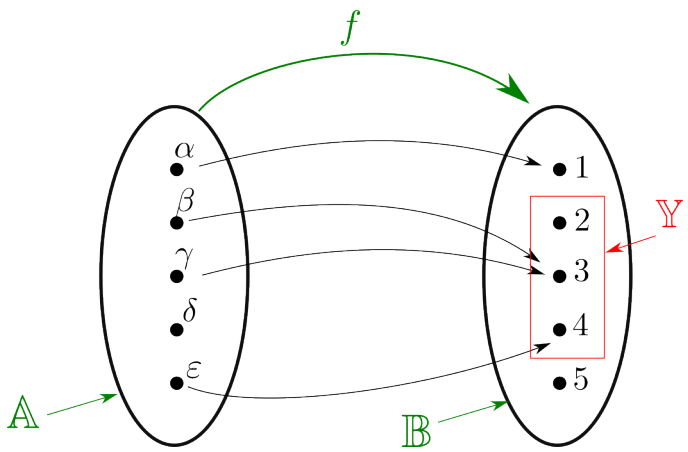


Figure 4.4: *A partial function f*

Introducing the indeterminate *NaN* (Not a number) in the sets, denoted by ι, allows us to get rid of the problem involved by the partiality of f.

Given a set $\mathbb{A}$, we define the *extended total set* as $\mathring{\mathbb{A}} = \mathbb{A} \cup \{\iota\}$. Thus, $\mathring{f} : \mathring{\mathbb{A}} \mapsto \mathring{\mathbb{B}}$ is the total extension of $f : \mathbb{A} \mapsto \mathbb{B}$ as illustrated by Figure 4.5. Following Definition 1, extended functions can be used to set as follows:

$$\mathring{f}(\mathbb{X}) = \begin{cases} f(\mathbb{X}) & \text{if } \mathbb{X} \subset \text{dom} f \\ f(\mathbb{X}) \cup \{\iota\} & \text{otherwise} \end{cases} \tag{4.17}$$

where $\mathbb{X} \subset \mathring{\mathbb{A}}$. Note that, in the figure, whereas $f(\{\gamma, \delta\}) = \{3\} \subset \mathbb{Y}$, we have $\mathring{f}(\{\gamma, \delta\}) = \{3, \iota\} \not\subset \mathbb{Y}$.

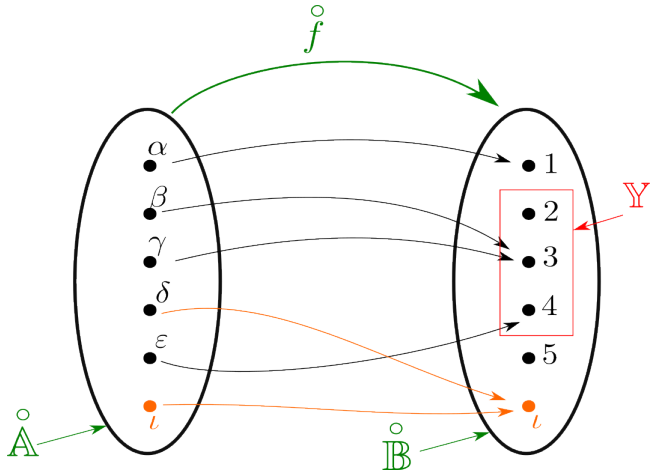


Figure 4.5: *Introduction of Not a Number ι*

4.3.3 Properties

For total functions, we have some properties that will be useful in our algorithms.

> ***Proposition 1***
>
> If $\mathring{f}$ is a total extension of f, we have
>
> $$\begin{array}{cc} \mathring{f}^{-1}(\overline{\mathbb{Y}}) = \overline{\mathring{f}^{-1}(\mathbb{Y})} & (i) \\ \mathring{f}(\mathbb{X}) \subset \mathbb{Y} \Rightarrow \mathbb{X} \subset \mathring{f}^{-1}(\mathbb{Y}) & (ii) \\ \mathring{f} \circ \mathring{f}^{-1}(\mathbb{Y}) = \mathbb{Y} & (iii) \end{array} \tag{4.18}$$

Proof:
Let us prove (ii) only. We have:

$$\begin{array}{rcl} \mathring{f}(\mathbb{X}) \subset \mathbb{Y} & \Leftrightarrow & \mathring{f}(\mathbb{X}) \cap \overline{\mathbb{Y}} = \emptyset \\ & \Leftrightarrow & \underbrace{\mathring{f}^{-1} \circ \mathring{f}(\mathbb{X})}_{\supset \mathbb{X}} \cap \underbrace{\mathring{f}^{-1}(\overline{\mathbb{Y}})}_{=\overline{\mathring{f}^{-1}(\mathbb{Y})}} = \emptyset \\ & \Rightarrow & \mathbb{X} \cap \overline{\mathring{f}^{-1}(\mathbb{Y})} = \emptyset \\ & \Leftrightarrow & \mathbb{X} \subset \mathring{f}^{-1}(\mathbb{Y}) \end{array} \tag{4.19}$$

> ***Proposition 2***
>
> If $\mathring{f}$, $\mathring{g}$ are total extensions of f and g then the total extension of $f \circ g$ is $\mathring{f} \circ \mathring{g}$.

Proof:
If $h = f \circ g$, we have

$$\mathring{f} \circ \mathring{g}(x) = \begin{cases} f \circ g(x) & \text{if } x \in \text{dom}g \text{ and } g(x) \in \text{dom}f \\ \iota & \text{otherwise} \end{cases} \tag{4.20}$$

Now, since

$$\text{dom}h = \text{dom}f \circ g = \text{dom}g \cap g^{-1}(\text{dom}f) \tag{4.21}$$

We get

$$\mathring{f} \circ \mathring{g}(x) = \begin{cases} h(x) & \text{if } x \in \text{dom}h \\ \iota & \text{if } x \notin \text{dom}h \end{cases} \tag{4.22}$$

which corresponds to $\mathring{h}(x)$.

Example. To illustrate the proposition, take

$$\begin{aligned} f(x) &= \sqrt{1-x} \\ g(x) &= \sqrt{x-1} \end{aligned} \tag{4.23}$$

Note that

$$\begin{aligned} \text{dom}f &= (-\infty, 1] \\ \text{dom}g &= [1, \infty) \end{aligned} \tag{4.24}$$

We have

$$h(x) = f \circ g(x) = \sqrt{1 - \sqrt{x-1}} \tag{4.25}$$

Since

$$\begin{aligned} \text{dom}h &= \text{dom}g \cap g^{-1}(\text{dom}f) \\ &= [1, \infty) \cap g^{-1}((-\infty, 1]) \\ &= [1, \infty) \cap [0, 2] = [1, 2] \end{aligned} \tag{4.26}$$

we get

$$\mathring{h}(x) = \begin{cases} \sqrt{1 - \sqrt{x-1}} & \text{if } x \in [1, 2] \\ \iota & \text{otherwise} \end{cases} \tag{4.27}$$

4.3.4 Total real arithmetic

We define the total extension of the classical arithmetic on real numbers. Consider the extended total set of reals:

$$\mathring{\mathbb{R}} = \mathbb{R} \cup \{\iota\}. \tag{4.28}$$

Adding such a special value for real numbers is now classical since it has been introduced by the IEEE-754 floating-point standard. Operations on real numbers can be extended to $\mathring{\mathbb{R}}$ as follows:

$$\begin{aligned} f(x) &= \iota \qquad \text{if } x \notin \text{dom}(f) \\ f(\iota) &= \iota \\ \iota \diamond x &= \iota \end{aligned} \tag{4.29}$$

where f is any partial function and $x \in \mathbb{R}$ and any binary operator $\diamond$.

Note that we do not define comparisons, which means that if we have the relation $a \leq b$ then both a and b belong to $\mathbb{R}$ (or equivalently neither a nor b can be equal to ι).

> ***Proposition 3***
>
> Consider a partial function $\mathbf{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$ given by an expression $\mathbf{f}(x_1, \ldots, x_n)$ including elementary functions (sin, $\sqrt{}$, log,...) and elementary operators $(+, -, *, /, \ldots)$. An expression for $\mathring{\mathbf{f}}$ can be obtained by the total real arithmetic.

Proof:
The proof is a direct consequence of the fact that the total extension is preserved by composition.

An element of the Cartesian product $\mathring{\mathbb{R}}^n = \mathring{\mathbb{R}} \times \cdots \times \mathring{\mathbb{R}}$ is called a *total vector*.

4.3.5 Link with the complex number i

The set of complex numbers $\mathbb{C}$ extends the set of real numbers by adding a number i such that $i^2 = -1$. The extension preserves some properties such that the fact $\mathbb{C}$ is a group with respect to the addition. Due to this, i has an opposite: $-i$. Indeed, $i + (-i) = 0$.

Take now the set $\mathring{\mathbb{R}}$ and let us check if ι has an opposite. We solve $\iota + x = 0$ and we get no solution for x. This means that $\mathring{\mathbb{R}}$ is not anymore a group and many properties we had for $\mathbb{R}$ are lost. As a consequence, symbolic resolution and group-based simplifications are not allowed in $\mathring{\mathbb{R}}$.

Moreover, adding i to build $\mathbb{C}$ involves the addition of many numbers of the form $a + ib$. In $\mathring{\mathbb{R}}$, we just add a single number: ι.

There exists a tiny link between $\mathring{\mathbb{R}}$ and $\mathbb{C}$ in the construction since we add one number. But the link stops here. Whereas complex numbers can be used to build a huge numbers of theorems and theories, the total numbers will be used as a tool to build the complementary of contractors.

4.4 Total intervals

In this section, we introduce the notion of intervals for $\mathring{\mathbb{R}}$, called *total intervals*.

4.4.1 Intervals in unions of lattices

On a lattice $(\mathbb{A}, \leq_{\mathbb{A}})$, we can define the notion of intervals, interval hull and contractors. This has been used for several types of lattices such as real numbers, integers, trajectories, graphs, etc. To be able to use interval methods, the lattice structure is required. We show here that it is not strictly necessary by considering union of lattices.

Definition 2

Consider two lattices $(\mathbb{A}, \leq_{\mathbb{A}})$ and $(\mathbb{B}, \leq_{\mathbb{B}})$ that are disjoint. Denote by $\mathbb{IA}$ and $\mathbb{IB}$, the set of all intervals of $\mathbb{A}$ and $\mathbb{B}$. We can define intervals of $\mathbb{C} = \mathbb{A} \cup \mathbb{B}$ as subsets $\mathbb{C}$ which have the form

$$[c] = [a] \cup [b], \tag{4.30}$$

where $[a] \in \mathbb{IA}$ and $[b] \in \mathbb{IB}$.

Indeed, the set $(\mathbb{C}, \leq_{\mathbb{C}})$ can be equipped with an order relation:

$$x \leq_{\mathbb{C}} y \Leftrightarrow \begin{cases} & x \in \mathbb{A}, y \in \mathbb{A}, x \leq_{\mathbb{A}} y \\ \text{or} & x \in \mathbb{B}, y \in \mathbb{B}, x \leq_{\mathbb{B}} y \end{cases} \tag{4.31}$$

Now, $\mathbb{C}$ is not a lattice, *i.e.*, if $x \in \mathbb{A}$, $y \in \mathbb{B}$ we cannot define $x \wedge y$ and $x \vee y$. This is due to the fact that we cannot provide a common lower or upper bounds for x, y.

Example. Consider the case where $\mathbb{A} = \mathbb{R}$ the set of real numbers and $\mathbb{B} = \{a, b, c, \ldots, z\}$ the set of letters. Both can be equipped with an order relation and both are lattices. Examples of intervals for the set $\mathbb{C} = \mathbb{A} \cup \mathbb{B}$ are

$$\begin{aligned} [c_1] &= [2, 5] \\ [c_2] &= \{e, f, g, h\} \\ [c_3] &= [2, 5] \cup \{e, f, g, h\} \\ [c_4] &= [4, 9] \cup \{g, h, i\} \\ [c_5] &= \emptyset \\ [c_6] &= \mathbb{A} \cup \mathbb{B} \end{aligned} \tag{4.32}$$

It is easy to check that the intervals of $\mathbb{C}$ is closed under intersection. It is thus a Moore family [112],[113] which means that contractor methods can be used.

4.4.2 Total intervals

Consider the singleton $\{\iota\}$ which is equipped with the trivial order relation : $\iota \leq \iota$. The set of all intervals of $\{\iota\}$ is $\{\emptyset, \{\iota\}\}$. The set $\mathring{\mathbb{R}}$ can be equipped with a partial order relation $\leq_{\mathring{\mathbb{R}}}$ derived from $\mathbb{R}$:

$$\begin{array}{c} \iota \leq_{\mathring{\mathbb{R}}} \iota \\ a \in \mathbb{R}, b \in \mathbb{R} \quad \text{then} \quad a \leq_{\mathring{\mathbb{R}}} b \text{ iff } a \leq_{\mathbb{R}} b \end{array} \tag{4.33}$$

Total intervals are denoted by $[\mathring{x}]$.

Examples of intervals of $\mathring{\mathbb{R}}$ are:

$$\begin{array}{c} [\mathring{a}] = [1, \infty) \\ [\mathring{b}] = [-1, 0] \cup \{\iota\} \\ [\mathring{c}] = \{\iota\} \\ [\mathring{d}] = \emptyset, \end{array} \tag{4.34}$$

as illustrated by Figure 4.6.

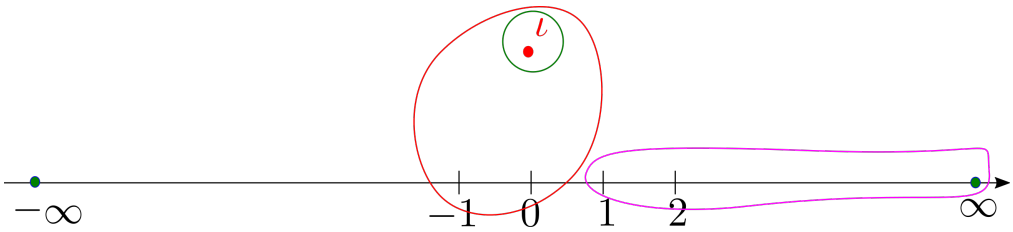


Figure 4.6: *Total intervals are intervals of* $\mathring{\mathbb{R}} = \mathbb{R} \cup \{\iota\}$

The set of total intervals is denoted by $\mathbb{I}\mathring{\mathbb{R}}$. We define the *hull* of a subset of $\mathring{\mathbb{X}}$ of $\mathring{\mathbb{R}}$ as the smallest total interval $[\mathring{x}]$ which encloses $\mathring{\mathbb{X}}$. We will write $[\mathring{x}] = \mathring{\mathbb{X}}$. For instance

$$\begin{array}{ccc} \{1, 2, 3\} & = & [1, 3] \\ \{1, 2, 3, \iota\} & = & [1, 3] \cup \{\iota\} \\ \{\iota\} & = & \{\iota\} \end{array} \tag{4.35}$$

4.4.3 Total interval arithmetic

Consider a partial function $f : \mathbb{R} \mapsto \mathbb{R}$. We define its *total interval extension* as follows

$$[\mathring{f}] = \{\mathring{f}(\mathring{x}), \mathring{x} \in [\mathring{x}]\}. \tag{4.36}$$

For instance $\sqrt{[-1,4]} = [0,2] \cup \{\iota\}$.

In the same manner, if $\diamond \in \{+,-,\cdot,/\}$, we define

$$[\mathring{a}] \diamond [\mathring{b}] = \{\mathring{a} \diamond \mathring{b}, \mathring{a} \in [\mathring{a}], \mathring{b} \in [\mathring{b}]\} \tag{4.37}$$

4.4.4 Total interval vector

The set of interval vectors $\mathring{\mathbb{R}}^n$ is a lattice [114]. We can thus define intervals of $\mathring{\mathbb{R}}^n$. The set of interval vectors has the form $\mathbb{I}\mathring{\mathbb{R}}^n = \mathbb{I}\mathring{\mathbb{R}} \times \cdots \times \mathbb{I}\mathring{\mathbb{R}}$.

We define the *hull* of a subset of $\mathring{\mathbb{X}}$ of $\mathring{\mathbb{R}}^n$ as the smallest $[\mathring{\mathbf{x}}]$ which encloses $\mathring{\mathbb{X}}$. We will write $[\mathring{\mathbf{x}}] = \mathring{\mathbb{X}}$. For instance,

$$([1,2] \times \{\iota\}) \cup ([3,4] \times [5,6]) = [1,4] \times ([5,6] \cup \{\iota\}). \tag{4.38}$$

4.5 Total contractors

This section extends the notion of contractor to total intervals. We first consider the case of elementary contractors built from elementary functions. Then, we consider the case of contractors defined from elementary operators.

4.5.1 Total directed contractor for a binary constraint

Consider a constraint of the form $y = f(x)$, where $f : \mathbb{R} \mapsto \mathbb{R}$: is a partial function with domain dom$f$. We can extend the constraint to $\mathring{\mathbb{R}}$ by the following decomposition

$$\left\{\begin{array}{c} \mathring{y} = f(\mathring{x}) \\ \mathring{x} \in \mathring{\mathbb{R}} \\ \mathring{y} \in \mathring{\mathbb{R}} \end{array}\right. \Leftrightarrow \left\{\begin{array}{ll} & \mathring{y} = f(\mathring{x}),\ \mathring{x} \in \text{dom} f,\ \mathring{y} \in \mathbb{R} \\ \text{or} & \mathring{x} \in \mathbb{R} \setminus \text{dom} f\ ,\ \mathring{y} = \iota \\ \text{or} & \mathring{x} = \iota,\ \mathring{y} = \iota \end{array}\right. \tag{4.39}$$

This means that $\iota = f(x)$ is considered as true only if and only if $x = \iota$ or if $x \notin$ dom$f$.

We define the *forward directional contractor* as

$$\overrightarrow{\mathcal{C}_f}([\mathring{x}]) = \{\mathring{y} \,|\, \exists \mathring{x} \in [\mathring{x}], \mathring{y} = f(\mathring{x})\} \tag{4.40}$$

and the *backward directional contractor*

$$\overleftarrow{\mathcal{C}_f}([\mathring{x}],[\mathring{y}]) = \{\mathring{x} \in [\mathring{x}] \,|\, \exists \mathring{y} \in [\mathring{y}],\, \mathring{y} = f(\mathring{x})\}. \tag{4.41}$$

Proposition 4

The forward directional contractor associated with f is

$$\overrightarrow{\mathcal{C}_f}([\mathring{x}]) = f([\mathring{x}] \cap \mathbb{R}) \cup ([\mathring{x}] \cap \{\iota\}) \cup \iota([\mathring{x}] \cap (\mathbb{R} \setminus \mathsf{dom} f)), \tag{4.42}$$

where ι is the constant function ι, i.e,

$$\iota(\mathbb{A}) = \begin{cases} \iota & \text{if } \mathbb{A} \neq \emptyset \\ \emptyset & \text{if } \mathbb{A} = \emptyset \end{cases} \tag{4.43}$$

Proof:

Since

$$\begin{array}{rcl} \mathring{x} \in \mathsf{dom} f & \Rightarrow & \mathring{y} = f(\mathring{x}) \\ \mathring{x} = \iota & \Rightarrow & \mathring{y} = \iota \\ \mathring{x} \in \mathbb{R} \setminus \mathsf{dom} f & \Rightarrow & \mathring{y} = \iota \end{array} \tag{4.44}$$

we have

$$f([\mathring{x}]) = \underbrace{f([\mathring{x}] \cap \mathsf{dom}\, f)}_{f([\mathring{x}]\cap\mathbb{R})} \cup \underbrace{\iota([\mathring{x}] \cap \{\iota\})}_{=[\mathring{x}]\cap\{\iota\}} \cup \iota([\mathring{x}] \cap (\mathbb{R} \setminus \mathsf{dom} f)). \tag{4.45}$$

Thus

$$\begin{array}{rcl} \overrightarrow{\mathcal{C}_f}([\mathring{x}]) & = & \{\mathring{y} \,|\, \exists \mathring{x} \in [\mathring{x}], \mathring{y} = f(\mathring{x})\} \\ & = & f([\mathring{x}] \cap \mathbb{R}) \cup ([\mathring{x}] \cap \{\iota\}) \\ & & \cup \iota([\mathring{x}] \cap (\mathbb{R} \setminus \mathsf{dom}\, f)) \end{array} \tag{4.46}$$

As a consequence, $\overrightarrow{\mathcal{C}_f}([\mathring{x}])$ can be generated using Algorithm 12.

Algorithm 12: Forward directional contractor $\overrightarrow{\mathcal{C}_f}$

```
Data: f, [x̊]
[ẏ] = f([x̊] ∩ ℝ);
[ẏ] = [ẏ] ∪ ([x̊] ∩ {ι}) if [x̊] ⊄ dom f then
    [ẏ] = [ẏ] ∪ {ι}
end
return [ẏ]
```

Proposition 5

The backward directional contractor associated with f is

$$\overleftarrow{\mathcal{C}_f}([\mathring{x}],[\mathring{y}]) = [\mathring{x}] \cap \left(f^{-1}([\mathring{y}] \cap \mathbb{R}) \cup \mathbb{I}([\mathring{y}])\right) \tag{4.47}$$

where

$$\mathbb{I}([\mathring{y}]) = \begin{cases} \{\iota\} \cup (\mathbb{R} \setminus \text{dom } f) & \text{if } \iota \in [\mathring{y}] \\ \emptyset & \text{otherwise} \end{cases} \tag{4.48}$$

Proof:

We have

$$\begin{aligned} \mathring{y} \in \mathbb{R} \quad &\Leftrightarrow \quad \mathring{x} \in f^{-1}(\{\mathring{y}\}) \\ \mathring{y} = \iota \quad &\Leftrightarrow \quad (\mathring{x} = \iota) \vee (\mathring{x} \in \mathbb{R} \setminus \text{dom } f) \\ &\Leftrightarrow \quad \mathring{x} \in \{\iota\} \cup (\mathbb{R} \setminus \text{dom } f) \end{aligned} \tag{4.49}$$

As a consequence, $\overleftarrow{\mathcal{C}_f}([\mathring{x}],[\mathring{y}])$ can be generated using Algorithm 13.

Algorithm 13: Backward directional contractor $\overleftarrow{\mathcal{C}_f}$

```
Data: f^{-1}, [x̊], [ẙ]
[ṙ] = ∅;
if [ẙ] = ∅ then
    return [ṙ];
end
[ṙ] = f^{-1}([ẙ] ∩ ℝ);
if ι ∈ [ẙ] then
    [ṙ] = [ṙ] ∪ (ℝ \ dom f) ∪ {ι};
end
return [ṙ] ∩ [x̊];
```

Example. *Total contractor for the square root.* Consider the constraint

$$y = \sqrt{x} \tag{4.50}$$

where all variables belong to $\mathring{\mathbb{R}}$. The values $(9,3),(-4,\iota),(\iota,\iota)$ for (x,y) are consistent with the constraint (4.50) whereas $(9,2),(-4,2),(9,\iota),(\iota,2)$ are inconsistent.

For instance, assume that we have $x \in [\mathring{x}] = [-2, 9], y \in [\mathring{y}] = [-1, 2] \cup \{\iota\}$. We obtain

$$\begin{array}{lcl} \overrightarrow{\mathcal{C}_{\sqrt{\cdot}}}([\mathring{x}]) & = & \sqrt{[-2,9]} = [0,3] \cup \{\iota\} \\ \overrightarrow{\mathcal{C}_{\sqrt{\cdot}}}([\mathring{x}]) \cap [\mathring{y}] & = & ([0,3] \cup \{\iota\}) \cap ([-1,2] \cup \{\iota\}) \\ & = & [0,2] \cup \{\iota\} \\ \overleftarrow{\mathcal{C}_{\sqrt{\cdot}}}([\mathring{x}], [\mathring{y}]) & = & [-2,4] \end{array} \tag{4.51}$$

It means that $x \in [-2, 4]$ and $y \in [0, 2] \cup \{\iota\}$.

Assume now that $x \in [\mathring{x}] = [4, 9], y \in [\mathring{y}] = [3, 15] \cup \{\iota\}$. We obtain

$$\begin{array}{lcl} \overrightarrow{\mathcal{C}_{\sqrt{\cdot}}}([\mathring{x}]) & = & \sqrt{[4,9]} = [2,3] \\ \overrightarrow{\mathcal{C}_{\sqrt{\cdot}}}([\mathring{x}]) \cap [\mathring{y}] & = & [2,3] \cap ([3,15] \cup \{\iota\}) = \{3\} \\ \overleftarrow{\mathcal{C}_{\sqrt{\cdot}}}([\mathring{x}], [\mathring{y}]) & = & \{9\} \end{array} \tag{4.52}$$

It means that $x = 9$ and $y = 3$.

4.5.2 Total directed contractor for a ternary constraint

Consider the ternary constraint of $z = x + y$. The case of constraints involving $-, \cdot, /$ can be defined from $+$ and binary constraints already treated in the previous section. The following reasoning can also be done for these operators.

We can extend the constraint $z = x + y$ to $\mathring{\mathbb{R}}$ by the following decomposition

$$\left\{ \begin{array}{l} \mathring{z} = \mathring{x} + \mathring{y} \\ \mathring{x} \in \mathring{\mathbb{R}} \\ \mathring{y} \in \mathring{\mathbb{R}} \\ \mathring{z} \in \mathring{\mathbb{R}} \end{array} \right. \Leftrightarrow \left\{ \begin{array}{ll} & \mathring{z} = \mathring{x} + \mathring{y},\ \mathring{x} \in \mathbb{R}, \mathring{y} \in \mathbb{R},\ \mathring{z} \in \mathbb{R} \\ \text{or} & (\mathring{x} = \iota \vee \mathring{y} = \iota) \wedge \mathring{z} = \iota \end{array} \right. \tag{4.53}$$

Note that in $\mathring{\mathbb{R}}$, we do not have

$$\mathring{z} = \mathring{x} + \mathring{y} \Leftrightarrow \mathring{x} = \mathring{z} - \mathring{y}. \tag{4.54}$$

Indeed, take $\mathring{x} = 1, \mathring{y} = \iota, \mathring{z} = \iota$. We have $\mathring{z} = \mathring{x} + \mathring{y}$ whereas $\mathring{x} \neq \mathring{z} - \mathring{y}$. As a consequence, the values $(2, 3, 5), (2, \iota, \iota), (\iota, \iota, \iota)$ for (x, y, z) are consistent with the constraint whereas $(2, 3, 6), (2, \iota, 4), (2, 3, \iota)$ are inconsistent.

We define the forward directed contractor

$$\overrightarrow{\mathcal{C}_+}([\mathring{x}],[\mathring{y}]) = \{\mathring{z} \mid \exists \mathring{x} \in [\mathring{x}], \exists \mathring{y} \in [\mathring{y}], \mathring{z} = \mathring{x} + \mathring{y}\} \tag{4.55}$$

and the backward directed contractor

$$\overleftarrow{\mathcal{C}_+}([\mathring{x}],[\mathring{y}],[\mathring{z}]) = \{(\mathring{x},\mathring{y}) \in [\mathring{x}] \times [\mathring{y}] \mid \exists \mathring{z} \in [\mathring{z}],\ \mathring{z} = \mathring{x} + \mathring{y}\} \tag{4.56}$$

We get Algorithms 14 and 15 for $\overrightarrow{\mathcal{C}_+}$ and $\overleftarrow{\mathcal{C}_+}$.

Algorithm 14: Forward directed contractor $\overrightarrow{\mathcal{C}_+}$

Data: $[\mathring{x}], [\mathring{y}]$
1 $[\mathring{z}] = ([\mathring{x}] \cap \mathbb{R}) + ([\mathring{y}] \cap \mathbb{R})$;
2 $[\mathring{z}] = [\mathring{z}] \cup ([\mathring{x}] \cap \{\iota\}) \cup ([\mathring{y}] \cap \{\iota\})$;
3 **return** $[\mathring{z}]$;

Step 1 computes to the interval containing of all feasible $z \in \mathbb{R}$.
Step 2 adds ι when $\iota \in [\mathring{x}]$ or when $\iota \in [\mathring{y}]$.

Algorithm 15: Backward directed contractor $\overleftarrow{\mathcal{C}_+}$

Data: $[\mathring{x}], [\mathring{y}], [\mathring{z}]$
1 **if** $\iota \notin [\mathring{z}]$ **then**
2 $\quad [\mathring{x}] = [\mathring{x}] \cap ([\mathring{z}] - [\mathring{y}])$;
3 $\quad [\mathring{y}] = [\mathring{y}] \cap ([\mathring{z}] - [\mathring{x}])$;
4 **end**
5 **return** $[\mathring{x}], [\mathring{y}]$

The implementation for $\overleftarrow{\mathcal{C}_+}$ is simplified by the fact that it is called after $\overrightarrow{\mathcal{C}_+}$.

Remarks

The contractor $\overleftarrow{\mathcal{C}_+}$ is often minimal, but not always. Indeed, there exist some rare counterexamples. Consider for instance the case

$$\begin{aligned} x &\in [1,2] \\ y &\in [3,4] \cup \{\iota\} \\ z &\in [6,9] \cup \{\iota\} \end{aligned} \tag{4.57}$$

If we call $\overrightarrow{\mathcal{C}_+}$, we get

Remarks (cont'd)

$$\begin{aligned} x &\in [1,2] \\ y &\in [3,4] \cup \{\iota\} \\ z &\in [6,6] \cup \{\iota\} \end{aligned} \tag{4.58}$$

The backward contractor $\overleftarrow{\mathcal{C}}_+$ (see Algorithm 15) yields no contraction for x and y whereas it should conclude the following contraction for y :

$$y \in [4,4] \cup \{\iota\}.$$

An optimal backward contractor could be obtained by Algorithm 16

Algorithm 16: An optimal backward contractor

Data: $[\mathring{x}], [\mathring{y}], [\mathring{z}]$
$[\mathring{r}_x] = \emptyset$,$[\mathring{r}_y] = \emptyset$;
$[x] = [\mathring{x}] \cap \mathbb{R}$ $[y] = [\mathring{y}] \cap \mathbb{R}$ $[z] = [\mathring{z}] \cap \mathbb{R}$;
if $[x] \neq \emptyset, [y] \neq \emptyset, [z] \neq \emptyset$ **then**
  $[\mathring{r}_x] = [x] \cap ([z] - [y])$ $[\mathring{r}_y] = [y] \cap ([z] - [x])$;
end
$[y] = [\mathring{y}] \cap \{\iota\}; [z] = [\mathring{z}] \cap \{\iota\}$;
if $[x] \neq \emptyset, [y] \neq \emptyset, [z] \neq \emptyset$ **then**
  $[\mathring{r}_y] = [\mathring{r}_y] \cup \{\iota\}$;
end
$[y] = [\mathring{y}] \cap \mathbb{R}$ $[x] = [\mathring{x}] \cap \{\iota\}$ $[z] = [\mathring{z}] \cap \{\iota\}$;
if $[x] \neq \emptyset, [y] \neq \emptyset, [z] \neq \emptyset$ **then**
  $[\mathring{r}_x] = [\mathring{r}_x] \cup \{\iota\}$
end
$[x] = [\mathring{x}] \cap \{\iota\}$ $[y] = [\mathring{y}] \cap \{\iota\}$ $[z] = [\mathring{z}] \cap \{\iota\}$;
if $[x] \neq \emptyset, [y] \neq \emptyset, [z] \neq \emptyset$ **then**
  $[\mathring{r}_x] = [\mathring{r}_x] \cup \{\iota\}$ $[\mathring{r}_y] = [\mathring{r}_y] \cup \{\iota\}$
end
return $[\mathring{r}_x], [\mathring{r}_y]$

Now, this algorithm improves the efficiency of a propagation only for rare situations. This is why we will preferred the use of the backward contractor of Algorithm 15, even if not fully minimal.

4.5.3 Total forward-backward contractor

We show how the forward-backward contractor works on two test-cases.

Test-case 1:

Consider the set

$$\mathbb{S} = \{(x, y) \mid y + \sqrt{x + y} \in [1, 2]\}. \tag{4.59}$$

We built the AST (Abstract Syntax Tree) associated with $\mathbb{S}$ as shown in Figure 4.7(a). We also build the AST for $\overline{\mathbb{S}}$ as in Figure 4.7(b). Note that the two trees are identical except the images that are complementary in $\mathring{\mathbb{R}}$, i.e.,

$$[1, 2] \cup ((-\infty, 1] \cup [2, \infty) \cup \{\iota\}) = \mathring{\mathbb{R}}. \tag{4.60}$$

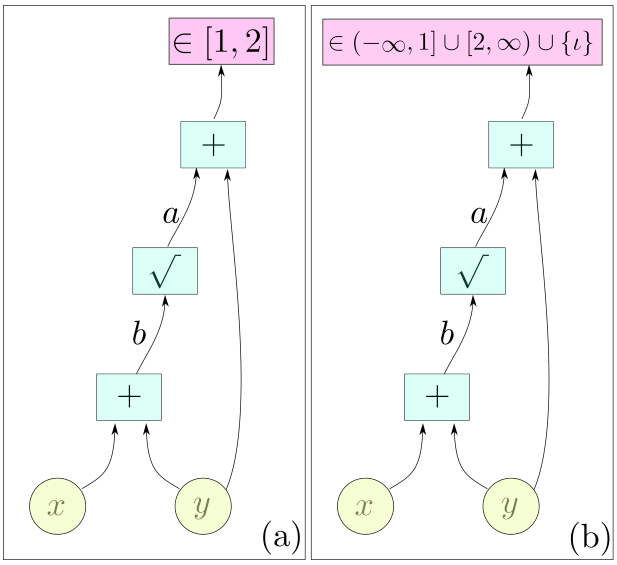


Figure 4.7: *AST for the constraint* $y + \sqrt{x + y} \in [1, 2]$ *(left) and its complementary (right)*

A forward-backward contractor yields Algorithm 17. Note that the set $\mathbb{Z}$ used below is not the set of relative integers (as often used in math books), but an interval of $\mathbb{R}$.

To have a contractor for $\mathbb{S}$ we call Algorithm 17 with $\mathbb{Z} = [1, 2]$. To get a contractor for $\overline{\mathbb{S}}$, we call the same algorithm with $\mathbb{Z} = (-\infty, 1] \cup [2, \infty) \cup \{\iota\}$. Using a paver with these two contractors, we are able to generate the approximation illustrated by Figure 4.8. The frame box is $[-10, 10] \times [-10, 10]$. An implementation is given in [115].

Algorithm 17: Contractor for the constraint $y + \sqrt{x+y} \in \mathbb{Z}$

Data: $[\mathring{x}], [\mathring{y}], \mathbb{Z}$

$[\mathring{b}] = \overrightarrow{\mathcal{C}_{+}}([\mathring{x}], [\mathring{y}])$;

$[\mathring{a}] = \overrightarrow{\mathcal{C}_{\sqrt{}}}([\mathring{b}])$;

$[\mathring{z}] = \overrightarrow{\mathcal{C}_{+}}([\mathring{a}], [\mathring{y}])$;

$[\mathring{z}] = [\mathring{z}] \cap \mathbb{Z}$;

$[\mathring{a}], [\mathring{y}] = \overleftarrow{\mathcal{C}_{+}}([\mathring{z}], [\mathring{a}], [\mathring{y}])$;

$[\mathring{b}] = \overleftarrow{\mathcal{C}_{\sqrt{}}}([\mathring{b}], [\mathring{a}])$;

$[\mathring{x}], [\mathring{y}] = \overleftarrow{\mathcal{C}_{+}}([\mathring{b}], [\mathring{x}], [\mathring{y}])$;

return $[\mathring{x}], [\mathring{y}]$;

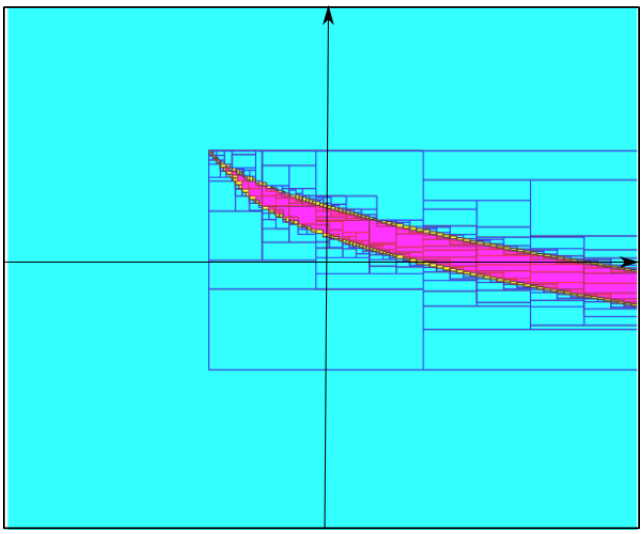


Figure 4.8: *Paving obtained using the contractor for $\mathbb{S}$ and its complementary*

Test-case 2:

Consider the discrete-time state space system, inspired from Henon map, of the form $\mathbf{x}(k+1) = \mathbf{f}(\mathbf{x}(k))$ with

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} bx_1 \\ 1 + ax_1^2 + \sqrt{x_2^2 + c} \end{pmatrix} \tag{4.61}$$

where $a = -1.4$, $b = 0.3$ and $c = 0.075$. The behavior of this system may not lead to a well defined state $\mathbf{x}(k)$ if the initial state vector $\mathbf{x}(0)$ is not chosen properly. We want to compute the set $\mathbb{S}$ of all initial vectors $\mathbf{x}(0)$ which lead to a state vector of $\mathbb{R}^2$ when $k = 5$. We define

$$\begin{array}{rcl} \mathbf{f}^0(\mathbf{x}) & = & \mathbf{x} \\ \mathbf{f}^{k+1}(\mathbf{x}) & = & \mathbf{f}^k \circ \mathbf{f}(\mathbf{x}). \end{array} \tag{4.62}$$

We have

$$\mathbb{S} = \{\mathbf{x} \in \mathbb{R}^2 \,|\, \mathbf{f}^5(\mathbf{x}) \in \mathbb{R}^2\}. \tag{4.63}$$

The forward backward contractor (see Algorithm 18) associated to the constraint $\mathbf{f}(\mathbf{x}) \in \mathbb{Y}$ is based on the AST of Figure 4.9.

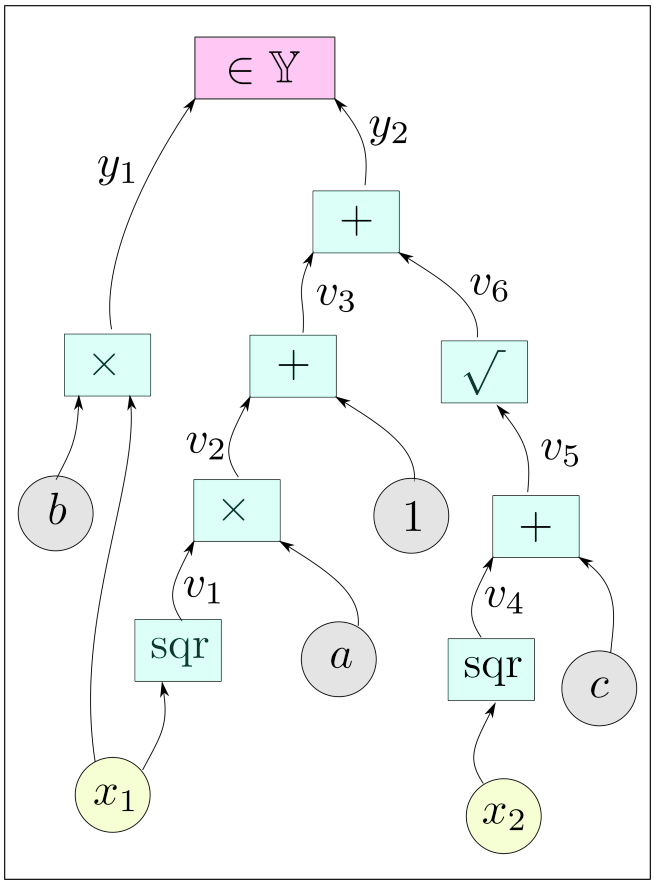


Figure 4.9: *AST for the constraint* $\mathbf{f}(\mathbf{x}) \in \mathbb{Y}$

To have a contractor for $\mathbb{S}$ we call Algorithm 18 with $\mathbb{Y} = \mathbb{R}^2$. To get a contractor for $\overline{\mathbb{S}}$, the same algorithm is called with $\mathbb{Y} = (\mathbb{R} \times \{\iota\}) \cup (\{\iota\} \times \mathbb{R}) \cup (\{\iota\} \times \{\iota\})$ which is the complementary of $\mathbb{R}^2$ in $(\mathbb{R} \cup \{\iota\})^2$ Using a paver with these two complementary contractors, we are able to generate the approximation illustrated by Figure 4.10. The frame box is $[-5, 5] \times [-5, 5]$.

Algorithm 18: Contractor for a constraint $\mathbf{f}(\mathbf{x}) \in \mathbb{Y}$ for Test-case 2

Data: $[\mathring{x}_1], [\mathring{x}_2], \mathbb{Y}$

1 $[\mathring{a}] = -1.4$; $[\mathring{b}] = 0.3$; $[\mathring{c}] = -0.075$;
2 $[\mathring{v}_1] = \overrightarrow{\mathcal{C}_{sqr}}([\mathring{x}_1])$;
3 $[\mathring{v}_2] = \overrightarrow{\mathcal{C}_{\times}}([\mathring{a}], [\mathring{v}_1])$;
4 $[\mathring{v}_3] = 1 + [\mathring{v}_2]$;
5 $[\mathring{v}_4] = \overrightarrow{\mathcal{C}_{sqr}}([\mathring{x}_2])$;
6 $[\mathring{v}_5] = \overrightarrow{\mathcal{C}_{+}}([\mathring{v}_4], [\mathring{c}])$;
7 $[\mathring{v}_6] = \overrightarrow{\mathcal{C}_{\sqrt{}}}([\mathring{v}_5])$;
8 $[\mathring{y}_1] = \overrightarrow{\mathcal{C}_{\times}}([\mathring{x}_1], [\mathring{b}])$;
9 $[\mathring{y}_2] = \overrightarrow{\mathcal{C}_{+}}([\mathring{v}_3], [\mathring{v}_6])$;
10 $[\mathring{y}_1] \times [\mathring{y}_2] = ([\mathring{y}_1] \times [\mathring{y}_2]) \cap \mathbb{Y}$;
11 $[\mathring{v}_3], [\mathring{v}_6] = \overleftarrow{\mathcal{C}_{+}}([\mathring{y}_2], [\mathring{v}_3], [\mathring{v}_6])$ (see Step 9);
12 $[\mathring{x}_1], [\mathring{b}] = \overleftarrow{\mathcal{C}_{\times}}([\mathring{y}_1], [\mathring{x}_1], [\mathring{b}])$ (see Step 8);
13 $[\mathring{v}_5] = \overleftarrow{\mathcal{C}_{\sqrt{}}}([\mathring{v}_5], [\mathring{v}_6])$ (see Step 7);
14 $[\mathring{v}_4], [\mathring{c}] = \overleftarrow{\mathcal{C}_{+}}([\mathring{v}_5], [\mathring{v}_4], [\mathring{c}])$ (see Step 6);
15 $[\mathring{x}_2] = \overleftarrow{\mathcal{C}_{sqr}}([\mathring{v}_4], [\mathring{x}_2])$ (see Step 5);
16 $[\mathring{v}_2] = ([\mathring{v}_3] - 1) \cap [\mathring{v}_2]$ (see Step 4);
17 $[\mathring{a}], [\mathring{v}_1] = \overleftarrow{\mathcal{C}_{\times}}([\mathring{v}_2], [\mathring{a}], [\mathring{v}_1])$(see Step 3);
18 $[\mathring{x}_1] = \overleftarrow{\mathcal{C}_{sqr}}([\mathring{x}_1], [\mathring{v}_1])$ (see Step 2);
19 **return** $[\mathring{x}_1], [\mathring{x}_2]$;

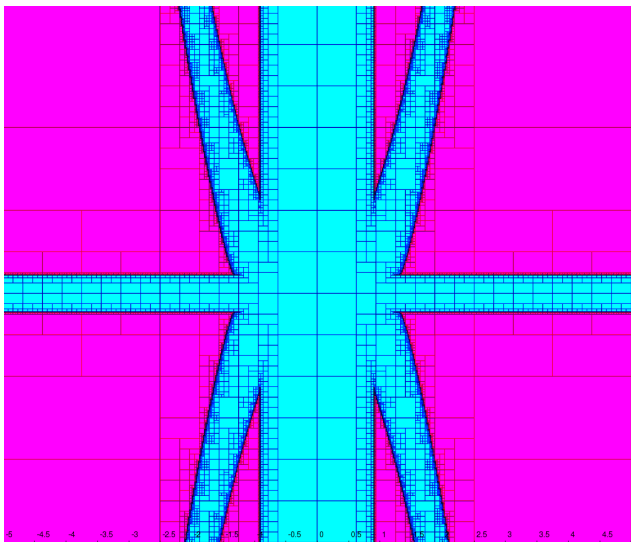


Figure 4.10: *Paving representing the solution set for Test-case 2*

4.6 Conclusion

This section proposed an extension to the interval arithmetic developed by Moore [116] in order to facilitate the implementation of complementary of contractors. For this purpose, we proposed to add a flag ι to each interval to form *total intervals*. This imply an

evolution of the associated arithmetic. For example, we now have $\sqrt{[-1,1]} = [0,1] \cup \{\iota\}$ instead of $\sqrt{[-1,1]} = [0,1]$. This is due to the fact that $\sqrt{\cdot}$, which is a partial function, has been made total. The ι number is not seen anymore as an exception, but as a possible value.

The flag ι has similarities with some decorations already used in the context of interval computation [117], [118]. The main advantage of our extension is to allow the interval propagation when some partial functions are involved in the definition of the constraints. We have presented a generalization of the forward-backward propagation to total intervals with an application to two test-cases.

5

A High-Level View of the Xinterval Extension

The work described in this chapter has been published in the following paper:

> ***Contributions***
>
> ► *Filiol Pierre et al. "RISC-V Based Hardware Acceleration of Interval Contractor Primitives in the Context of Mobile Robotics." Acta Cybern. 26 (2024): 889-912.*

Contents

5.1 The Xinterval Custom Extension 113
5.2 Encoding Intervals in RISC-V's Registers 114
5.3 Basic Interval Computations 117
5.3.1 Creating Intervals From Single-Precision Bounds 117
5.3.2 Loading And Storing Intervals 119
5.3.3 Extracting Information From An Interval 119
5.3.4 Set Operations 120
5.4 Half-Contractors 123
5.4.1 Arithmetic Operators 123
5.4.2 Other Real Functions 126
5.5 Conclusion 129

5.1 The Xinterval Custom Extension

The main ambition of this PhD dissertation is to complement the RISC-V ISA with a custom extension dedicated to interval arithmetic with a focus on embedded robotics. For

compliancy with the RISC-V specification regarding naming conventions [119], the proposed extension will be referred as *xinterval*.

The philosophy of *xinterval* is to optimize the workflow presented in Section 1.2.4.5 which is central to a lot of robotics problems. As a reminder, complex interval contractors can be generated procedurally through the use of the HC4R algorithm which breaks down the mathematical expression of the associated contraint into multiple elementary sub-constraints and then perform a forward-backward propagation. In this context, the evaluation of forward-backward contractors over elementary interval constraints, whether arithmetic (addition, ...) , algebraic (square-root, ...), or transcendental (cosine, ...) play a critical role in the overall performances and are the main focus of the *xinterval* implementation.

In addition to contractors, the micro-architecture of *xinterval* adds native hardware support for the interval data-type and defines multiple instructions to manipulate it from a high-level language such as C.

5.2 Encoding Intervals in RISC-V's Registers

All the instructions from *xinterval* use the opcode *custom-0* (value 0xB) which is dedicated to custom extensions [120]. The decision was made to refrain from using the vacant instruction slots from standard opcodes in order to limit potential collisions with future standard extensions. However, due to RISC-V architectural constrainsts, instructions sharing an opcode must adopt the same instructions format (see Section 3.2.1.2).

The R-type instruction format (Figure 5.1), seemed like the most sensible choice for *xinterval* as it is already used in the standard for most arithmetic operators on integers and floating-points. The content of two source registers rs_1 and rs_2 is read as input of the computation and the result is then stored in a destination register r_d. This principle is extended to intervals in the context of *xinterval*.

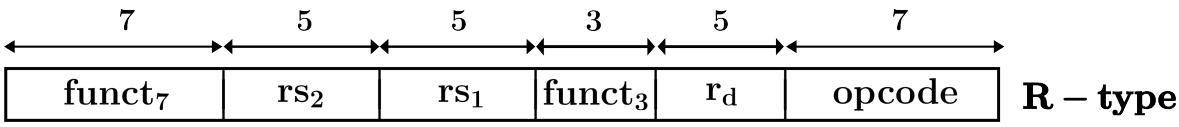


Figure 5.1: *R-type instruction format in the RISC-V*

The RISC-V architecture provides 32 general-purpose integer registers of width *XLEN*, which can be either 32-bit, 64-bit or 128-bit depending of the implementation. If the optional standard extensions F, D, or Q are supported, 32 additional floating-point registers are added with a width equal to the biggest floating-point size supported (32-bit for F, 64-bit for D and 128-bit for Q). When several floating-point extensions are supported in the same design , for example single-precision F and double-precision D, the narrower type is stored in the lower part of the register (Figure 5.2).

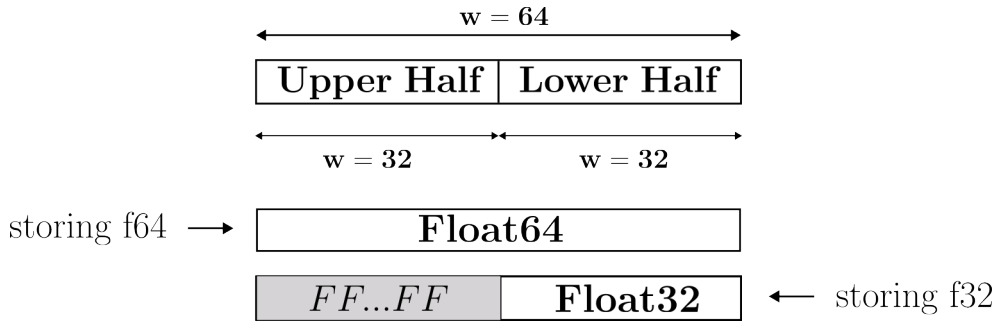


Figure 5.2: *Floating-point register for a processor supporting both F and D extensions. Double-precision numbers fill the whole space while single-precision numbers occupy only the lowest half. In the last case, the upper half is padded with 1's to return a NaN when attempting to read a float64 (NaN boxing).*

In the *xinterval* extension, intervals are represented using the 64-bit data-structure shown on top of Figure 5.3. The goal is to allow intervals to fit in the 64-bit double-precision registers offered by the RISC-V standard extension D to keep architectural impact to a minimum and ease the deployment of the custom extension. The concept of iota, introduced in Chapter 4, is integrated in the chosen representation through a dedicated 1-bit flag as well as another 1-bit flag to indicate the empty interval. This leaves 31 bits of data to represent each bound, which is inferior to the 32 bits used in the common single-precision format (1 sign bit, 8 exponent bits and 23 fraction bits). The choice was made to sacrifice one bit of exponent to keep the same precision as the standard floating-point format. More precisely, the fields of the data-structure are organized as follow:

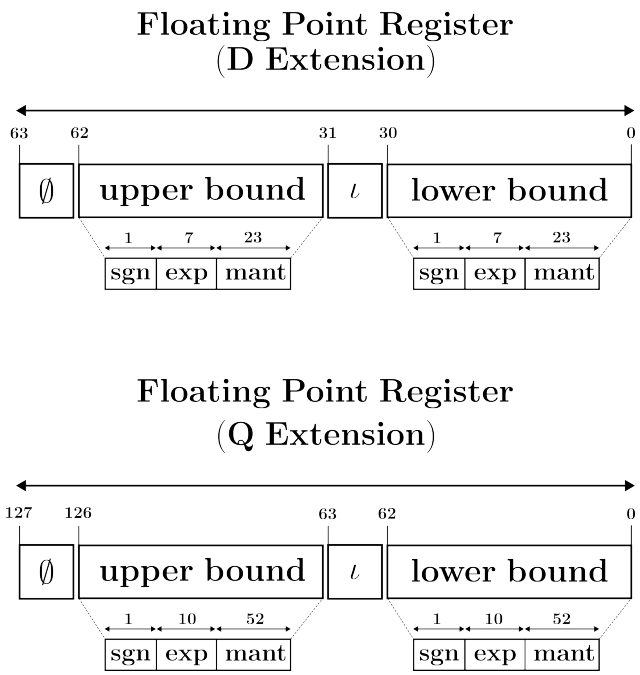


Figure 5.3: *Interval data-structure in xinterval. On top, the 64-bit interval format with Float31 bounds as implemented in xinterval. At the bottom, an example of what a port for double-precision bounds would look like (assuming the support of Q extension).*

Empty Flag (bit 63): this bit is used to mark an interval as empty. Some software implementations of interval arithmetic do not save this state and instead re-evaluate it when needed by comparing both floating-point bounds to ensure proper ordering. Saving this information is very handy in the context of a hardware implementation because 1) checking a bit in a register is almost free and 2) it prevents from instantiating floating-point comparators in each circuit which saves space on the FPGA/ASIC and reduces the depth of the pipeline.

Iota Flag (bit 31): this bit is used to mark an interval which is subject to the iota phenomenon described in Chapter 4. This acts as a hardware trap to prevent a forward/backward propagation to yield incorrect results due real functions such as square-root or logarithm being evaluated out of their domain of definition.

Upper/Lower Bounds (bit [62-32] and [30-0]): to compensate for the two bits used by the empty/iota flags and still fit in the 64-bit D registers, the bounds are encoded using a custom floating-point format denoted as Float31. This format keeps the same mantissa (and thus precision) as the classic single-precision/Float32 numbers and sacrifices one exponent bit (going from 8 to 7). The consequence is a decrease in the range of representable values (Figure 5.4) with little to no impact on the typical robotics applications. Similarly, *xinterval* prevents the use of subnormal bounds in intervals and instead flush them either to zero (round-down) or to the smallest normal number (round-up). The main reason is that the only goal of the proposed implementation is to ensure that the output of the computations will lead to an interval which encloses the real result, but not necessary with the minimal enclosure. In robotics, these guarantees matter important than excessive precision, especially since the support for subnormals (i.e $< 10^{-19}$ for Float31) vastly increases the complexity of hardware operators and rarely makes sense in robotics. The FloPoCo library (refer to Section 2.3.2.3), which is the basis for the floating-point operators used in this study, has also made the choice not to support subnormals, and handle them in a similar way.

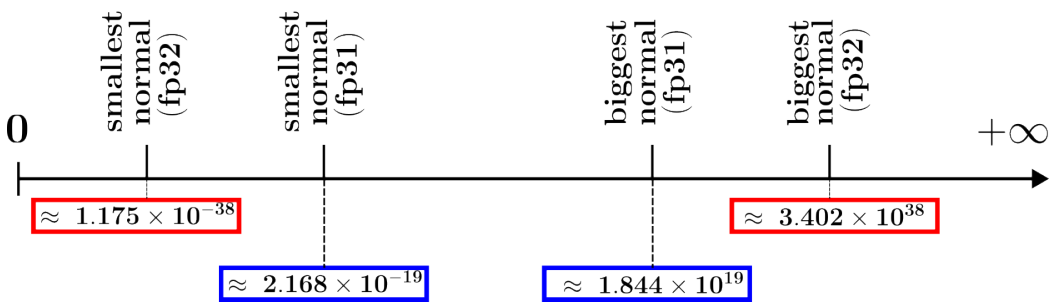


Figure 5.4: *Comparison of the Range of Representable Numbers for Float31 and Float32*

The same kind of encoding could be used to support intervals with double-precision bounds, as depicted at the bottom of Figure 5.3.

5.3 Basic Interval Computations

A significant number of instructions from *xinterval* are dedicated to interfacing the hardware-representation of an interval (Figure 5.3) from a high-level programming language. This section presents the most relevant design choices regarding this topic.

5.3.1 Creating Intervals From Single-Precision Bounds

register content interpretation	
reg interval	reg float32

asm	r_d	rs_1	rs_2	effect	description
itv.mk	✓	✓	✓	$\{r_d\} = \text{itv}(\{rs_1\}, \{rs_2\})$	Create interval from float bounds
itv.mki	✓	✓	✓	$\{r_d\} = \text{itv}(\{rs_1\}, \{rs_2\}) \cup \iota$	Create interval from float bounds + iota

Figure 5.5: *Instructions to create an interval (with or without iota)*

Xinterval handles the creation of new intervals using the **itv.mk** and **itv.mki** instructions (Figure 5.5). The programmer passes two Float32 bounds in source registers rs_1/rs_2 which are used to construct the associated interval in the destination register r_d. The only difference between these instructions lies in the fact that the iota flag is set to 1 for **itv.mki**. Listing 4 gives a preview of how these instructions can be used directly from the C language. Section 7.2.1 goes into more details on how to set up this chain.

More precisely, the creation of an interval involves the following operations:

Bound conversion (Float32 → Float31): the bounds are fetched from double-precision registers and interpreted as Float32 bounds through the mecanism illustrated in Figure 5.2. To generate the encoding of the resulting interval, the circuit must convert both bounds from Float32 to Float31. This operation technically saves the precision as both formats share the same mantissa but additional events may occur:

- Any Float32 subnormal input is flushed to zero, for the reasons given in Section 5.2.
- In some cases, a Float32 input may be converted as a subnormal Float31, which is also flushed to zero.
- A Float32 input may fall out of the range of valid Float31 (Figure 5.4). In these cases, the associated Float31 bound is either set to 0 (input too small) or ∞ (input too big).

Checks for illegal values: after the conversion step, the circuit ensures that bounds are properly ordered, as required by the *set-based* flavor of IEEE-754 (Section 1.4.1.1), and that they do not form an illegal interval ($\{-\infty\}$, $\{+\infty\}$ or anything involving a NaN). If

Listing 4: *Creating intervals using C language and wrappers around* ***itv.mk*** *and* ***itv.mki***

```c
/* typedef double as interval */
typedef double interval_t;

/* values of iota flag */
enum iota_t {
    NO_IOTA = 0,
    IOTA = 1
};

/* inline function which makes use of xinterval instruction itv.mk */
/* inputs loaded from double registers */
/* output stored in double register */
inline interval_t __attribute__((always_inline)) itv_mk(float lb, float ub) {
    interval_t result;
    asm("itv.mk %0, %1, %2" : "=f"(result) : "f"(lb), "f"(ub));
    return result;
}

/* inline function which makes use of xinterval instruction itv.mki */
/* inputs loaded from double registers */
/* output stored in double register */
inline interval_t __attribute__((always_inline)) itv_mki(float lb, float ub) {
    interval_t result;
    asm("itv.mki %0, %1, %2" : "=f"(result) : "f"(lb), "f"(ub));
    return result;
}

/* Create an interval with the dedicated primitive */
interval_t new_itv(float lb, float ub, enum iota_t iota_flag) {
    if(IOTA == iota_flag) {
        return itv_mki(lb, ub);
    }
    else {
        return itv_mk(lb, ub);
    }
}
```

these conditions are not met, the empty interval, whose encoding is presented in Figure 5.6, is stored in the destination register.

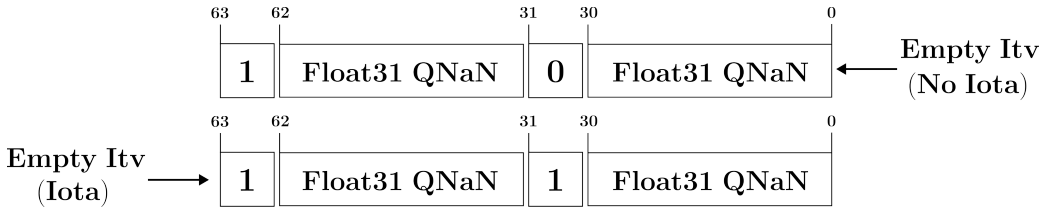


Figure 5.6: *Encoding of the empty interval in xinterval, where the iota flag can be optionnaly set.*

5.3.2 Loading And Storing Intervals

There is no special instructions in *xinterval* to transfer intervals between memory and registers as it would require respectively the use of the S-type (store) and I-type (load) instruction formats which would be against the design choices adopted in Section 5.2.

Practically, this is not an issue because the intervals live in the double-precision floating-point registers of the processor, thus allowing the re-use of the standard RISC-V instructions **fld** and **fsd**. In C, this can be achieved very simply by typedef'ing the *double* type as *interval_t* (Listing 4) which tricks the compiler into using the existing load/store instructions. Figure 5.7 illustrates this fact with the compilation of a simple *xinterval*-based code.

```
1  /* typedef double as interval */
2  typedef double interval_t;
3
4  int main(int argc, char** argv) {
5
6      float lb = -1.0f;
7      float ub = 1.0f;
8      interval_t itv = new_itv(lb, ub,
       ↪  NO_IOTA);
9      return EXIT_SUCCESS;
10 }
```

(a) *xinterval "hello world"*

```
000000000001021c <new_itv>:
   1021c:       fc010113                addi    sp,sp,-64
   10220:       02113c23                sd      ra,56(sp)
   10224:       02813823                sd      s0,48(sp)

   1025c:       fec42787                flw     fa5,-20(s0)
   10260:       fe842707                flw     fa4,-24(s0)
   10264:       00e7978b                itv.mki fa5,fa5,fa4
   10268:       fef43027                fsd     fa5,-32(s0)
   1026c:       fe043787                fld     fa5,-32(s0)
   10270:       02c0006f                j       1029c <new_itv+0x80>
   10274:       fcc42787                flw     fa5,-52(s0)
   10278:       fcf42e27                fsw     fa5,-36(s0)
   1027c:       fc842787                flw     fa5,-56(s0)
   10280:       fcf42c27                fsw     fa5,-40(s0)
   10284:       fdc42787                flw     fa5,-36(s0)
   10288:       fd842707                flw     fa4,-40(s0)
   1028c:       00e7878b                itv.mk  fa5,fa5,fa4
   10290:       fcf43827                fsd     fa5,-48(s0)
   10294:       fd043787                fld     fa5,-48(s0)
   10298:       00000013                nop
   1029c:       e2078553                fmv.x.d a0,fa5
   102a0:       03813083                ld      ra,56(sp)
   102a4:       03013403                ld      s0,48(sp)
   102a8:       04010113                addi    sp,sp,64
   102ac:       00008067                ret
```

(b) *An Extract of the Compiled Code (objdump)*

Figure 5.7: *Compilation of a simple xinterval code*

5.3.3 Extracting Information From An Interval

register content interpretation	
reg interval	*reg* float32
reg integer	

asm	r_d	rs_1	rs_2	effect	description
itv.ext.e	✓	✓	✗	$\{r_d\} = 1$ if $\{rs_1\}$ is empty else 0	Check empty state of selected itv
itv.ext.i	✓	✓	✗	$\{r_d\} = 1$ if $\{rs_1\}$ is iota else 0	Check iota state of selected itv
itv.ext.lb	✓	✓	✗	$\{r_d\} = \mathrm{lb}(\{rs_1\})$	Get lower bound of selected itv
itv.ext.ub	✓	✓	✗	$\{r_d\} = \mathrm{ub}(\{rs_1\})$	Get upper bound of selected itv

Figure 5.8: *Instructions to extract information from an interval*

Xinterval allows the programmer to extract individual fields from the interval data-structure (Figure 5.3) to accomodate most applications.

Checking for empty intervals: this is performed through the **itv.ext.e** instruction which extracts the empty flag from the interval stored in source register rs_1 and returns either '1' (when empty) or '0' (when non-empty) in the integer destination register r_d. The behavior is similar to existing standard RISC-V instructions such as **fclass.d** which returns the result of the classification of a double-precision number in an integer register.

Checking for iota intervals: the **itv.ext.i** instruction works similarly to **itv.ext.e** but operates on the iota flag. Once again, the target interval is passed through source register rs_1 and the result ("1" when iota is set, else "0") is returned in the integer destination register r_d.

Extracting bounds: instructions **itv.ext.lb** and **itv.ext.ub** repectively extract the lower (resp. upper) bound of the interval passed in the source register rs_1 and returns it as a Float32 value in the floating-point destination register r_d. Internally, the instruction must convert Float31 bounds to Float32, which is the opposite of the operation performed in **itv.mk**. This conversion is however far less challenging as the range of valid Float31 numbers is included in Float32's one.

Listings 5 and 6 illustrate how these instructions can be wrapped and used in the C language. Again, refer to Section 7.2.1 for more details.

5.3.4 Set Operations

register content interpretation

reg	interval	

asm	r_d	rs_1	rs_2	effect	description
itv.set.u	✓	✓	✓	$\{r_d\} = \{rs_1\} \cup \{rs_2\}$	Union of two itv
itv.set.i	✓	✓	✓	$\{r_d\} = \{rs_1\} \cap \{rs_2\}$	Intersection of two itv

Figure 5.9: *Instructions to Perform Set operations on Intervals*

Basic set operators such as union or intersection are very common in interval applications, whether as standalone computations or as part of a backward contractor (Table 5.1):

	Addition	Subtraction
Constraint	$x_3 = x_1 + x_2$	$x_3 = x_1 - x_2$
$\overrightarrow{\mathcal{C}_*}$	$[x_3] = [x_1] + [x_2]$	$[x_3] = [x_1] - [x_2]$
$\overleftarrow{\mathcal{C}_*}$	$[x_1] = ([x_3] - [x_2]) \cap [x_1]$ $[x_2] = ([x_3] - [x_1]) \cap [x_2]$	$[x_1] = ([x_3] + [x_2]) \cap [x_1]$ $[x_2] = ([x_1] - [x_3]) \cap [x_2]$

Table 5.1: *Use of set operators in backward contractors*

Interval union and intersection are performed respectively with instructions **itv.set.u**

Listing 5: *Wrappers around **itv.ext** instructions*

```
/* typedef double as interval */
typedef double interval_t;

/* inline function which makes use of xinterval instruction itv.ext.e */
/* inputs loaded from double register */
/* output stored in integer register */
inline int __attribute__((always_inline)) itv_ext_e(interval_t itv) {
    int result;
    asm("itv.ext.e %0, %1" : "=r"(result) : "f"(itv));
    return result;
}

/* inline function which makes use of xinterval instruction itv.ext.i */
/* inputs loaded from double register */
/* output stored in integer register */
inline int __attribute__((always_inline)) itv_ext_i(interval_t itv) {
    int result;
    asm("itv.ext.i %0, %1" : "=r"(result) : "f"(itv));
    return result;
}

/* inline function which makes use of xinterval instruction itv.ext.lb */
/* inputs loaded from double register */
/* output stored in float register */
inline float __attribute__((always_inline)) itv_ext_lb(interval_t itv) {
    float result;
    asm("itv.ext.lb %0, %1" : "=f"(result) : "f"(itv));
    return result;
}

/* inline function which makes use of xinterval instruction itv.ext.ub */
/* inputs loaded from double register */
/* output stored in float register */
inline float __attribute__((always_inline)) itv_ext_ub(interval_t itv) {
    float result;
    asm("itv.ext.ub %0, %1" : "=f"(result) : "f"(itv));
    return result;
}

/* Wrappers */
int itv_empty(interval_t interval) {
    return itv_ext_e(interval);
}
enum iota_t itv_iota(interval_t interval) {
    return (enum iota_t)(itv_ext_i(interval));
}
float itv_lb(interval_t interval) {
    return itv_ext_lb(interval);
}
float itv_ub(interval_t interval) {
    return itv_ext_ub(interval);
}
```

Listing 6: *Extracting Fields from Intervals using C language*

```c
/* typedef double as interval */
typedef double interval_t;

int main(int argc, char** argv) {

    /* Creating interval */
    float lb = -1.0f;
    float ub = 1.0f;
    interval_t itv = new_itv(lb, ub, NO_IOTA);

    /* Testing stuffs */
    if(itv_empty(itv)) {
        printf("This interval is empty !");
    }
    if(itv_iota(itv)) {
        printf("This interval is subject to iota !");
    }

    /* Retrieving bounds */
    float bound_lb = itv_lb(itv);
    float bound_ub = itv_ub(itv);

    return EXIT_SUCCESS;
}
```

and **itv.set.i**. Input intervals passed in source registers rs_1, rs_2 and the result of the set operation is returned in destination register r_d. Internally, several floating-point comparisons are performed between all the input bounds to detect one of the patterns illustrated in Figure 5.10.

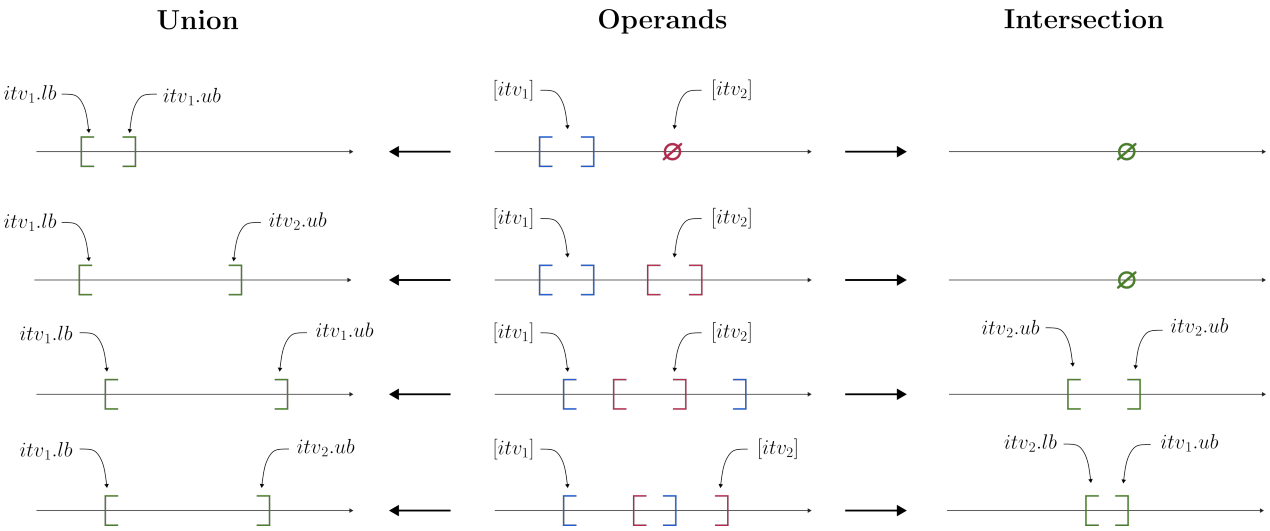


Figure 5.10: *Behavior of Union and Intersection in* ***itv.set.u*** *and* ***itv.set.i***

Remarks

In addition to the regular instruction formats available in the RISC-V format (Figure 3.4), a subtype of the R-type format, called R4-type, allows the use of three source registers rs_1, rs_2 and rs_3:

$\mathrm{rs_3}$	$\mathrm{funct_2}$	$\mathrm{rs_2}$	$\mathrm{rs_1}$	$\mathrm{funct_3}$	$\mathrm{r_d}$	opcode	**R4 – type**

Naively, R4-type format seems perfect to implement backward contractors for two-input operators (addition, subtraction, ...) which require three loads and one store in the registers (Table 5.1). However, this flexibility comes at the price of the $funct_7$ field becoming a $funct_2$, thus reducing the number of instructions available for a given opcode to $2^5 = 32$. This choice may prevent the potential addition of new instructions in *xinterval*, as 25 instructions slots are already in use.
This limitation comforts the decision made to implement every *xinterval* instructions using the R-type format (Section 5.2). As a consequence, the instructions for backward contractors (especially the 2-input ones) actually only perform a half-contraction (without the final re-intersection) and must be followed by a call to the relevant set instruction such as **itv.set.u** or **itv.set.i**.

5.4 Half-Contractors

As stated previously, *xinterval* provides instructions to compute the half-contractors for the most recurring elementary constraints in interval applications. They are organized in two families:

- Half-contractors based on 2-input arithmetic operators.
- Half-contractors based on other real arithmetic/transcendental functions.

5.4.1 Arithmetic Operators

The instructions from *xinterval* responsible for contractors related to arithmetic operators are listed below:

Addition and subtraction:

The forward-backward contractors required in both addition/subtraction constraints are constructed as follow:

register content interpretation	
reg	interval

asm	r_d	rs_1	rs_2	effect	description
itv.add	✓	✓	✓	$\{r_d\} = \{rs_1\} + \{rs_2\}$	add two itv
itv.sub	✓	✓	✓	$\{r_d\} = \{rs_1\} - \{rs_2\}$	subtract two itv
itv.mul	✓	✓	✓	$\{r_d\} = \{rs_1\} \times \{rs_2\}$	multiply two itv
itv.div	✓	✓	✓	$\{r_d\} = \{rs_1\} / \{rs_2\}$	divide two itv

Figure 5.11: *Instructions for Half-Contractors Based on Arithmetic Operators*

Constraint

Let x,y,z three variables linked by the addition constraint $z = x + y$.
As stated in Table 1.4, the z variable can be first evaluated through the forward contractor $\overrightarrow{\mathcal{C}_+}$, which is simply

$$\overrightarrow{\mathcal{C}_+}(x, y) = [x] + [y]$$

After that, x and y can be narrowed down through backward propagation by using respectively $\overleftarrow{\mathcal{C}_{+1}}$ and $\overleftarrow{\mathcal{C}_{+2}}$:

$$\overleftarrow{\mathcal{C}_{+1}}(x, y, z) = ([z] - [y]) \cap [x]$$
$$\overleftarrow{\mathcal{C}_{+2}}(x, y, z) = ([z] - [x]) \cap [y]$$

The logic is analogous for subtraction: if x,y,z are now linked by the constraint $z = x - y$, the associated forward contractor $\overrightarrow{\mathcal{C}_-}$ becomes

$$\overrightarrow{\mathcal{C}_-}(x, y) = [x] - [y]$$

and $\overleftarrow{\mathcal{C}_{-1}}$ and $\overleftarrow{\mathcal{C}_{-2}}$ are

$$\overleftarrow{\mathcal{C}_{-1}}(x, y, z) = ([z] + [y]) \cap [x]$$
$$\overleftarrow{\mathcal{C}_{-2}}(x, y, z) = ([x] - [z]) \cap [y]$$

As a consequence, all the aforementionned contractors can be evaluated by providing dedicated instructions for interval addition (**itv.add**) and subtraction (**itv.sub**) and by relying on **itv.set.i** for the backward intersection. Both **itv.add** and **itv.sub** requires the interval operands to be passed through source registers rs_1/rs_2 and return the result in

the destination register r_d.

Multiplication and division:

The forward-backward contractors required in both multiplication/division constraints are obtained in a similar way as addition and subtraction:

> ***Constraint***
>
> Let x,y,z three variables linked by the multiplication constraint $z = x \times y$.
> As stated in Table 1.4, the z variable can be first evaluated through the forward contractor $\overrightarrow{\mathcal{C}_\times}$, which is simply
>
> $$\overrightarrow{\mathcal{C}_\times}(x, y) = [x] \times [y]$$
>
> After that, x and y can be narrowed down through backward propagation by using respectively $\overleftarrow{\mathcal{C}_{\times 1}}$ and $\overleftarrow{\mathcal{C}_{\times 2}}$:
>
> $$\overleftarrow{\mathcal{C}_{\times 1}}(x, y, z) = \frac{[z]}{[y]} \cap [x]$$
> $$\overleftarrow{\mathcal{C}_{\times 2}}(x, y, z) = \frac{[z]}{[x]} \cap [y]$$
>
> The logic is analogous for division: if x,y,z are now linked by the constraint $z = x/y$, the associated forward contractor $\overrightarrow{\mathcal{C}_/}$ becomes
>
> $$\overrightarrow{\mathcal{C}_/}(x, y) = \frac{[x]}{[y]}$$
>
> and $\overleftarrow{\mathcal{C}_{/_1}}$ and $\overleftarrow{\mathcal{C}_{/_2}}$ are
>
> $$\overleftarrow{\mathcal{C}_{/_1}}(x, y, z) = ([z] \times [y]) \cap [x]$$
> $$\overleftarrow{\mathcal{C}_{/_2}}(x, y, z) = \frac{[x]}{[z]} \cap [y]$$

Once again, it is only necessary to add instructions for interval multiplication (**itv.mul**) and division (**itv.div**). Their calling convention is also similar to **itv.add**/**itv.sub**.

5.4.2 Other Real Functions

Many robotics applications have to compute things like distances, angles or trajectories, which requires the evaluation of real functions such as square, square-root, logarithm or sine/cosine. Xinterval provides primitives to accelerate the related forward-backward contractors. This is an improvement compared to existing solutions such as the Gaol library (refer to Section 1.4.2). The latter relies on x86's SIMD extensions to compute the bounds of intervals in parallel in the case of arithmetic operators, but has no means to do it when dealing with real functions. This is one of the benefit of designing dedicated hardware.

register content interpretation	
reg	interval

asm	r_d	rs_1	rs_2	effect	description
itv.sqr	✓	✓	✗	$\{r_d\} = \text{sqr}(\{rs_1\})$	square of the selected itv
itv.sqr_revh	✓	✓	✗	$\{r_d\} = \text{itv}(-\sqrt{ub(\{rs_1\})}, \sqrt{ub(\{rs_1\})})$	helper for backward square ctc
itv.sqrt	✓	✓	✗	$\{r_d\} = \sqrt{\{rs_1\}}$	square-root of the selected itv
itv.exp	✓	✓	✗	$\{r_d\} = \exp(\{rs_1\})$	exponential of the selected itv
itv.log	✓	✓	✗	$\{r_d\} = \log(\{rs_1\})$	logarithm of the selected itv
itv.cos	✓	✓	✗	$\{r_d\} = \cos(\{rs_1\})$	cosine of the selected itv
itv.sin	✓	✓	✗	$\{r_d\} = \sin(\{rs_1\})$	sine of the selected itv
itv.acos	✓	✓	✗	$\{r_d\} = \arccos(\{rs_1\})$	arccosine of the selected itv
itv.asin	✓	✓	✗	$\{r_d\} = \arcsin(\{rs_1\})$	arcsine of the selected itv

Figure 5.12: *Instructions for Half-Contractors Based on Real Functions*

Square and square-root:

> ***Constraint***
>
> Let x,y be two variables linked by the square constraint $y = x^2$.
> As per Table 1.5, the forward contractor $\overrightarrow{\mathcal{C}_{sqr}}$ is obtained by applying the interval version of the square function:
>
> $$\overrightarrow{\mathcal{C}_{sqr}}(x) = [x]^2$$
>
> The backward contractor $\overleftarrow{\mathcal{C}_{sqr}}$ is slightly more complex because the square-function is not strictly monotonous over its domain of definition (refer to Section 1.2.4.4).

Constraint (cont'd)

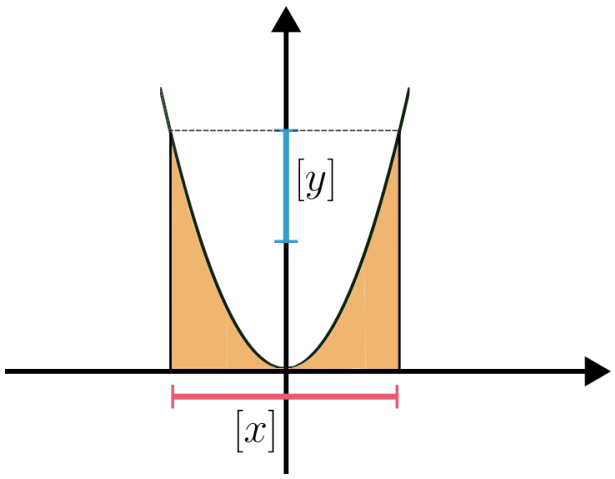


Figure 5.13: *Illustration of the Square Backward-Propagation*

The range of feasible values for $[x]$ after the backward propagation of $[y]$ is illustrated in Figure 5.13, which implies that

$$\overleftarrow{\mathcal{C}_{sqr}}(x, y) = \text{itv}(-\sqrt{\overline{y}}, \sqrt{\overline{y}}) \cap [x]$$

Let now x, y be linked by the constraint $y = \sqrt{x}$. The square-root function is easier to implement, due to its strict monotonicity. The forward contractor $\overrightarrow{\mathcal{C}_{\sqrt{}}}$ is again obtained by simply applying the interval version of the square-root function. However, the subtlety comes from the fact that the square-root function is not defined everywhere, which means that depending of value of x, the iota flag must be raised to guide the future backward propagation.

$$\overrightarrow{\mathcal{C}_{\sqrt{}}}(x) = \sqrt{[x]}$$

The backward contractor $\overleftarrow{\mathcal{C}_{\sqrt{}}}$ is also very simple and relies on the interval square function:

$$\overleftarrow{\mathcal{C}_{\sqrt{}}}(x, y) = [y]^2 \cap [x]$$

Xinterval provides three new instructions to implement all these half-contractors. The squared-interval required by $\overrightarrow{\mathcal{C}_{sqr}}$ and $\overleftarrow{\mathcal{C}_{\sqrt{}}}$ is obtained through the **itv.sqr** instruction. For the reasons exposed in the remarks from Section 1.2.4.5, re-using the **itv.mul** instruction to perform the square would be sub-optimal. Similarly, $\overleftarrow{\mathcal{C}_{sqr}}$ and $\overrightarrow{\mathcal{C}_{\sqrt{}}}$ can be computed respectively using the **itv.sqr.revh** and **itv.sqrt** instructions.

Exponential and logarithm:

Due to their stricty monotonicity, the implementation of the contractors associated to exponential and logarithm is trivial:

> ***Constraint***
>
> Let x,y be two variables linked by the exponential constraint $y = e^x$,
> $\overrightarrow{\mathcal{C}_{exp}}$ and $\overleftarrow{\mathcal{C}_{exp}}$ are simply:
>
> $$\overrightarrow{\mathcal{C}_{sqr}}(x) = e^{[x]}$$
> $$\overleftarrow{\mathcal{C}_{sqr}}(x, y) = \log([y]) \cap [x]$$
>
> If x,y are now linked by $y = \log x$,
> $\overrightarrow{\mathcal{C}_{log}}$ and $\overleftarrow{\mathcal{C}_{log}}$ are also simple:
>
> $$\overrightarrow{\mathcal{C}_{log}}(x) = \log([x])$$
> $$\overleftarrow{\mathcal{C}_{log}}(x, y) = e^{[y]} \cap [x]$$
>
> Similarly to square-root, the only subtlety for logarithm is due to the function being partially defined on $\mathbb{R}$, thus potentially requiring the assertion of a iota flag.

Xinterval adds two additional instructions to implement all these half-contractors, which are **itv.exp** and **itv.log**.

Trigonometry

Two trigonometric functions, namely sine and cosine, are supported in *xinterval*.

> ***Constraint***
>
> Let x,y be two variables linked by the cosine constraint $y = \cos(x)$,
> $\overrightarrow{\mathcal{C}_{cos}}$ is obtained by applying the interval version of the sine function:
>
> $$\overrightarrow{\mathcal{C}_{cos}}(x) = \cos[x]$$
>
> The backward contractor is a lot trickier. It relies both on the ability to compute the arccosine function and select the 2π-wide windows consistent with the initial value of $[x]$. This operation is illustrated in Figure 5.14, where the y interval (in blue) can be contracted to one of the green intervals (numbered from 1 to 8), the third one is finally selected because there is a valid intersection with x (red

Constraint (cont'd)

interval).

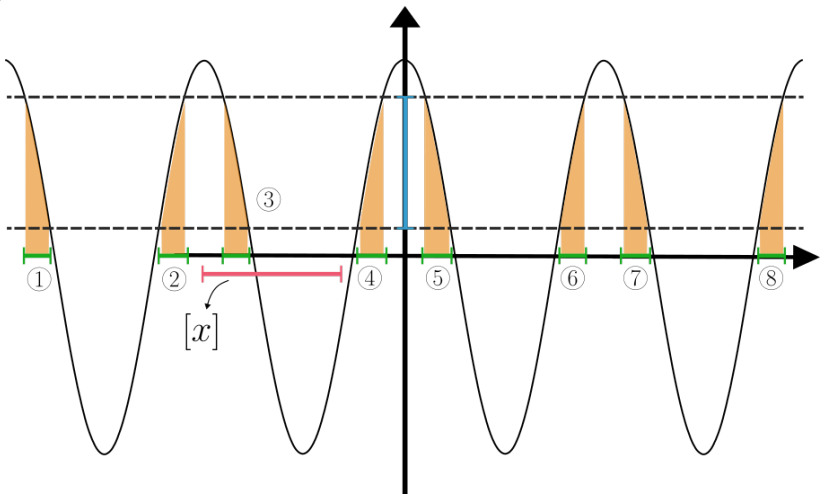


Figure 5.14: *Illustration of the Cosine Backward-Propagation*

The principle is exactly the same for sine and thus not described.

Xinterval provides instructions **itv.cos** and **itv.sin** to compute respectively the forward contractors $\overrightarrow{\mathcal{C}_{cos}}$ and $\overrightarrow{\mathcal{C}_{sin}}$. Regarding the backward contractors, only the interval counterparts of arccosine/arcsine are provided through instructions **itv.acos** and **itv.asin**. The actual backward-contractor is then implemented as software on top of these instructions, since the box-selection algorithm for selecting the right green window in Figure 5.14 implies a lot of branching logic which would perform poorly in hardware.

5.5 Conclusion

This chapter presented the design choices for xinterval at the ISA and software-hardware interface levels, based on the mathematical expressions of the main half-contractors. The next step is now to descend a bit further toward hardware micro-architecture and present the circuits which implement the backend of the aforementionned instructions.

asm	r_d	rs_1	rs_2	effect	description
itv.mk	✓	✓	✓	$\{r_d\} = \text{itv}(\{rs_1\}, \{rs_2\})$	Create interval from float bounds
itv.mki	✓	✓	✓	$\{r_d\} = \text{itv}(\{rs_1\}, \{rs_2\}) \cup \iota$	Create interval from float bounds + iota

asm	r_d	rs_1	rs_2	effect	description
itv.ext.e	✓	✓	✗	$\{r_d\} = 1$ if $\{rs_1\}$ is empty else 0	Check empty state of selected itv
itv.ext.i	✓	✓	✗	$\{r_d\} = 1$ if $\{rs_1\}$ is iota else 0	Check iota state of selected itv
itv.ext.lb	✓	✓	✗	$\{r_d\} = \text{lb}(\{rs_1\})$	Get lower bound of selected itv
itv.ext.ub	✓	✓	✗	$\{r_d\} = \text{ub}(\{rs_1\})$	Get upper bound of selected itv

asm	r_d	rs_1	rs_2	effect	description
itv.set.u	✓	✓	✓	$\{r_d\} = \{rs_1\} \cup \{rs_2\}$	Union of two itv
itv.set.i	✓	✓	✓	$\{r_d\} = \{rs_1\} \cap \{rs_2\}$	Intersection of two itv

asm	r_d	rs_1	rs_2	effect	description
itv.add	✓	✓	✓	$\{r_d\} = \{rs_1\} + \{rs_2\}$	add two itv
itv.sub	✓	✓	✓	$\{r_d\} = \{rs_1\} - \{rs_2\}$	subtract two itv
itv.mul	✓	✓	✓	$\{r_d\} = \{rs_1\} \times \{rs_2\}$	multiply two itv
itv.div	✓	✓	✓	$\{r_d\} = \{rs_1\} / \{rs_2\}$	divide two itv

asm	r_d	rs_1	rs_2	effect	description
itv.sqr	✓	✓	✗	$\{r_d\} = \text{sqr}(\{rs_1\})$	square of the selected itv
itv.sqr_revh	✓	✓	✗	$\{r_d\} = \text{itv}(-\sqrt{ub(\{rs_1\})}, \sqrt{ub(\{rs_1\})})$	helper for backward square ctc
itv.sqrt	✓	✓	✗	$\{r_d\} = \sqrt{\{rs_1\}}$	square-root of the selected itv
itv.exp	✓	✓	✗	$\{r_d\} = \exp(\{rs_1\})$	exponential of the selected itv
itv.log	✓	✓	✗	$\{r_d\} = \log(\{rs_1\})$	logarithm of the selected itv
itv.cos	✓	✓	✗	$\{r_d\} = \cos(\{rs_1\})$	cosine of the selected itv
itv.sin	✓	✓	✗	$\{r_d\} = \sin(\{rs_1\})$	sine of the selected itv
itv.acos	✓	✓	✗	$\{r_d\} = \arccos(\{rs_1\})$	arccosine of the selected itv
itv.asin	✓	✓	✗	$\{r_d\} = \arcsin(\{rs_1\})$	arcsine of the selected itv

Figure 5.15: *Listing of xinterval instructions*

6

The Micro-Architecture of the Xinterval Extension

The work described in this chapter has been published in the following paper:

> ***Contributions***
>
> ► *Filiol Pierre et al. "Efficient Hardware Primitives for Interval Contractors in Robotics and Integration to a Custom RISC-V ISA Extension," 2025 23rd IEEE Interregional NEWCAS Conference (NEWCAS), doi: 10.1109/NewCAS64648.2025.11107029*

Contents

6.1 The Software/Hardware Interface 132

6.1.1 From ISA Instructions to Hardware Circuits 132

6.1.2 The Minimal Xinterval Core 133

6.2 Review Of Hardware Primitives For Intervals 135

6.2.1 A Word About Floating-Point Arithmetic 135

6.2.2 Conversions Between Intervals and Floating-Points 136

6.2.3 Itv_set 141

6.2.4 Itv_add 143

6.2.5 Itv_mul 146

6.2.6 Itv_div 150

6.2.7 Itv_sqrt 154

6.2.8 Itv_exp 156

6.2.9 Itv_log 157

6.2.10 Itv_trig and Itv_trig_inv 159

6.3 Synthesis Results 168

6.1 The Software/Hardware Interface

6.1.1 From ISA Instructions to Hardware Circuits

The classic RISC pipeline involves a decoding process during which the next instruction is analyzed and sent to the appropriate processing unit. The RISC-V architecture greatly reduces the complexity of this operation thanks to clever design choices. For example, instructions have a fixed length of 32-bit (trivial fetching) as well as an $opcode$ field positionned invariantly between bits 0 and 6 to associate them with one of the formats presented in Section 3.2.1.2. After that, a set of additional identification fields ($funct_2$, $funct_3$, $funct_7$, ...) are used to deduce the assembly instruction about to be executed. As mentionned previously, *xinterval* fully relies on the *custom-0* opcode and use a R-type encoding for all its instructions:

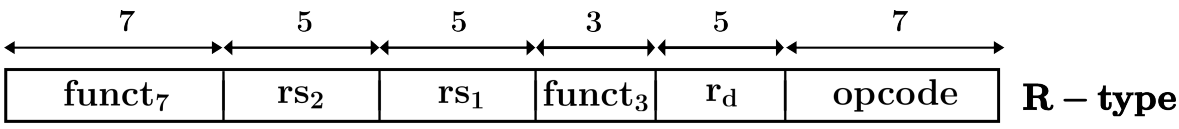


Figure 6.1: *R-type instruction format in the RISC-V*

Due to $funct_3$ and $funct_7$ being respectively 3 and 7-bit long, an opcode set to encode R-type instructions can contain at most $2^{3+7} = 1024$, which is more than enough for intervals. The association between the ISA instructions presented throughout Chapter 5 and the underlying hardware circuits is presented in Table 6.1.

Hardware Entity	*Xinterval* Encoding	Instructions Funct7	Instructions Funct3
itv_make	itv.mk	0x0	0x0
	itv.mki		0x1
itv_extract	itv.ext.e	0x1	0x0
	itv.ext.i		0x1
	itv.ext.lb		0x2
	itv.ext.ub		0x3
itv_set	itv.set.u	0x2	0x0
	itv.set.i		0x1
itv_add	itv.add	0x3	0x0
	itv.sub		0x1
itv_mul	itv.mul	0x4	0x0
	itv.sqr		0x1
itv_div	itv.div	0x5	0x0
itv_sqrt	itv.sqrt	0x6	0x0
	itv.sqr_rev		0x1
itv_exp	itv.exp	0x7	0x0
itv_log	itv.log	0x8	0x0
itv_trig	itv.cos	0x9	0x0
	itv.sin		0x1
itv_trig_inv	itv.arccos	0xA	0x0
	itv.arcsin		0x1

Table 6.1: *Relationship Between Hardware Circuits and xinterval ISA Instructions*

Each hardware entity is associated to one or several closely-related instructions, like **itv_make** which is used for both **itv.mk** and **itv.mki**. To ease the decoding, the *funct7*

field is used to refer to a particular hardware operator while the *funct3* discriminates between all the instructions handled by the circuit. As an example, Figure 6.2 presents the full encoding signature of **itv.add** and **itv.sub**:

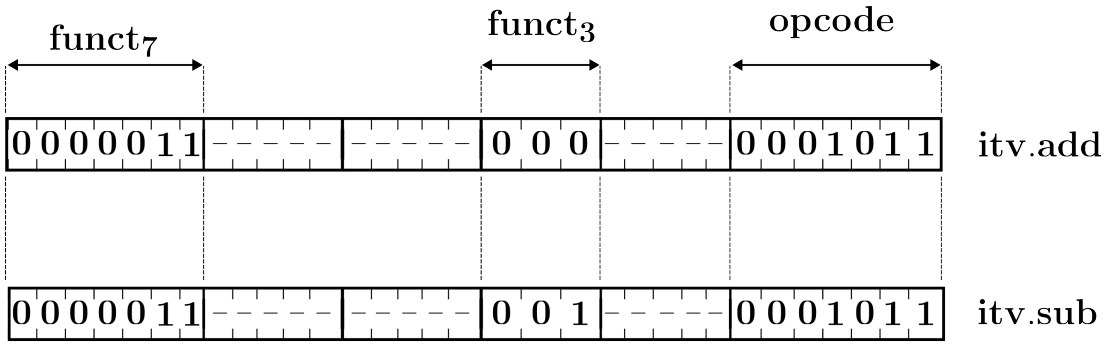


Figure 6.2: *Encoding of two xinterval instructions*

6.1.2 The Minimal Xinterval Core

Since all *xinterval* instructions use the R-type format, the hardware operators described in Table 6.1 can be conveniently wrapped using the interface depicted in Figure 6.3.

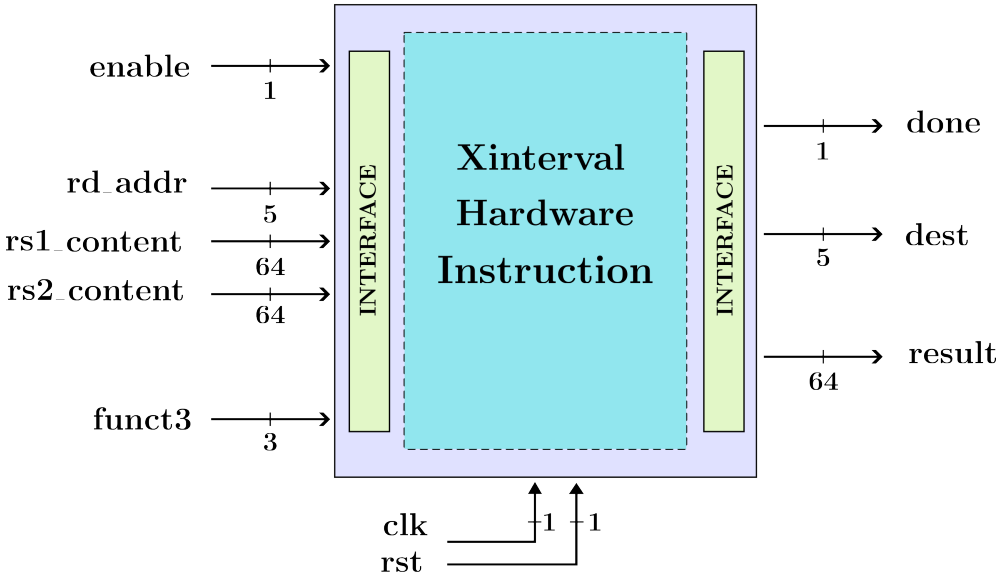


Figure 6.3: *Common Hardware Interface For R-Type Instructions*

The purpose of the ports used in this generic interface are described below:

Input

- **enable:** asserted at the first clock cycle when the instruction execution starts.
- **rd_addr:** id of the destination register (between 0 and 31) where the result of the instruction will be stored.
- **rs1/rs2_content:** content of the source registers rs_1,rs_2 read from register file and required by the instruction for the computation.
- **funct3:** used for instruction identification.

Output

- **done:** asserted when the instruction execution stops.
- **dest:** id of the destination register (between 0 and 31) where the result of the instruction will be stored. Practically propagated from **rd_addr** during the n cycles of the instruction execution.
- **result:** 64-bit value corresponding to the result of the instruction which will be stored in the register identified by $dest$.

To reflect on the viability of interval arithmetic bundled as a RISC-V custom extension, the proposed hardware primitives are evaluated against the following metrics:

- Functional verification (refer to Section 7.2.2.2).
- Speedup on reference benchmarks (refer to Section 7.3).
- Timing estimations: Fmax, latency, ...
- Hardware utilization/Energy consumption.

To achieve this in fairly simple way, a minimalist processor called *xinterval core* and inspired by the traditional RISC pipeline has been developped. The architecture is summarized in Figure 6.4. The core features a decode unit compatible with the RISC-V's R-type instruction format, a local register file and an execution unit which perform multiplexing between all the interval operators presented in Table 6.1.

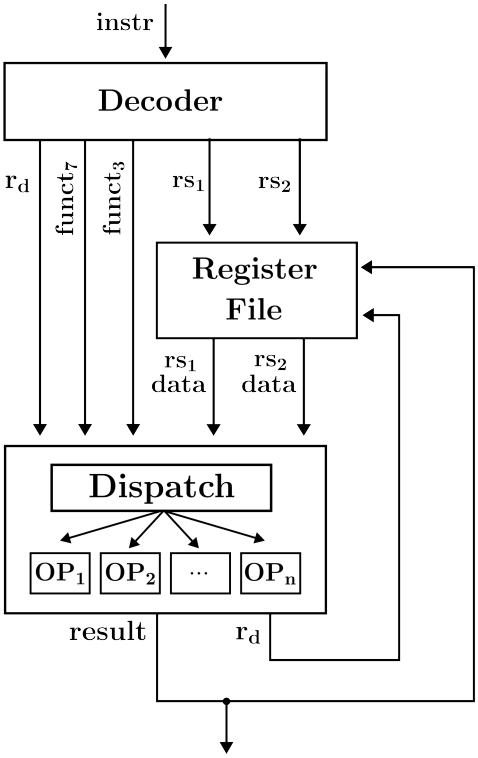


Figure 6.4: *The Minimal xinterval Core*

6.2 Review Of Hardware Primitives For Intervals

6.2.1 A Word About Floating-Point Arithmetic

As presented in Chapter 1, interval computations make an extensive use of IEEE-754 floating-point arithmetic which is notoriously hard to implement on a FPGA, at least in a reliable and performant way. To overcome this, the primitives reviewed in this section rely on the FloPoCo library (see Section) to generate the required operators. As a reminder, the tool has the following advantages:

- Support of custom floating-point formats (i.e Float31).
- Support of every IEEE-754 rounding modes, up and down being required for interval arithmetic.
- Selectable pipeline depth and Fmax given the target FPGA platform.

From now on, FloPoCo operator blocks appearing in interval circuits will be denoted according to Figure 6.5.

For reference, Table 6.2 shows the resource utilization and timings obtained when synthesizing the FloPoCo operators used in *xinterval* primitives on Xilinx XC7Z020 FPGA.

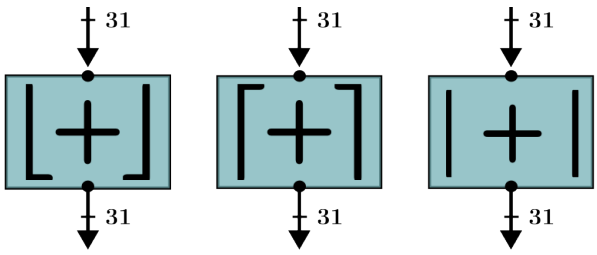


Figure 6.5: *FloPoCo Addition Block For Float31 Operands In Multiple Rounding Modes. Rounded toward $-\infty$ (left), $+\infty$ (center), nearest-even (right).*

FloPoCo Operator	LUTs	FFs	Muxes	DSPs	Cycles	FMax
Comparator	23	0	0	0	1	100 MHz
Addition	455	397	0	0	4	100 MHz
Multiplication	73	96	0	2	2	100 MHz
Division	759	605	13	0	12	100 MHz
Exponential	298	388	5	1	9	100 MHz
Logarithm	355	286	0	0	9	100 MHz
Square-Root	671	779	24	3	9	100 MHz
Fuse-Multiply-Add	822	650	0	2	6	100 MHz
CORDIC (Q1.23)	1332	649	0	0	8	100 MHz

Table 6.2: *Synthesis Results For The Main FloPoCo Operators On Xilinx XC7Z020 FPGA*

6.2.2 Conversions Between Intervals and Floating-Points

The **itv_make** primitive is the hardware backend for instructions **itv.mk** and **itv.mki** whose behaviors have been been presented in Section 5.3.1.

The **itv_extract** primitive is the hardware backend for instructions **itv.ext.e**, **itv.ext.i**, **itv.ext.lb** and **itv.ext.ub** whose behaviors have been been presented in Section 5.3.3. Its main purpose is to operate in the opposite direction as **itv_make** and convert the fields of the interval data-structure to the standard RISC-V types.

6.2.2.1 Itv_make: Operator Interface

The primitive returns the 64-bit interval corresponding to a Float32 lower bound $f32_lb$, a Float32 upper bound $f32_ub$ and a $iota$ flag passed as input. The purpose of this instruction is mostly to ensure that intervals passed as input data (e.g initial search space or other constants) are well-formed in the sense of the IEEE-1788 standard. As a reminder, this implies that for an interval $[x]$, the bounds are such as $\underline{x} \leq \bar{x}$ with $\underline{x} \neq +\infty$ and $\bar{x} \neq -\infty$. Additionaly, it acts as the main bridge between the standard RISC-V and the interval world and ensures that all the created intervals are valid with respect to the

format presented in Figure 5.3 and will be compatible with the other *xinterval* instructions. Implementing this instruction in hardware rather than pure software allows to run the Float32 to Float31 conversions on each bound concurrently which saves a bit of time.

Hardware Entity

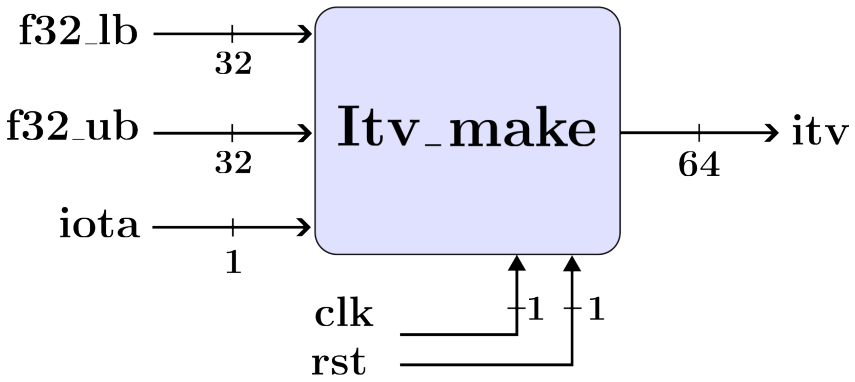


Figure 6.6: *Top-level Entity For Itv_make*

Input Ports

- **f32_lb:** Float32 lower bound for output interval.
- **f32_ub:** Float32 upper bound for output interval.
- **iota:** ota flag for output interval.

Output Ports

- **itv:** output interval.

6.2.2.2 Itv_make: Operator Datapath

The circuit datapath (Figure 6.7) is composed of three main stages:

- Conversion of bounds candidates from Float32 to Float31.
- Ensuring the proper ordering of converted Float31 bounds.
- Crafting a sanitized interval which takes into account potential NaN/unordered bounds after conversion.

Stage 1 - Conversion from Float32 to Float31:

The first step involves a classification of the Float32 input, which can be performed by analyzing the values of the sign,exponent and mantissa fields as specified in the IEEE-754 standard. Subnormals numbers are deemed of little interest in robotics while vastly

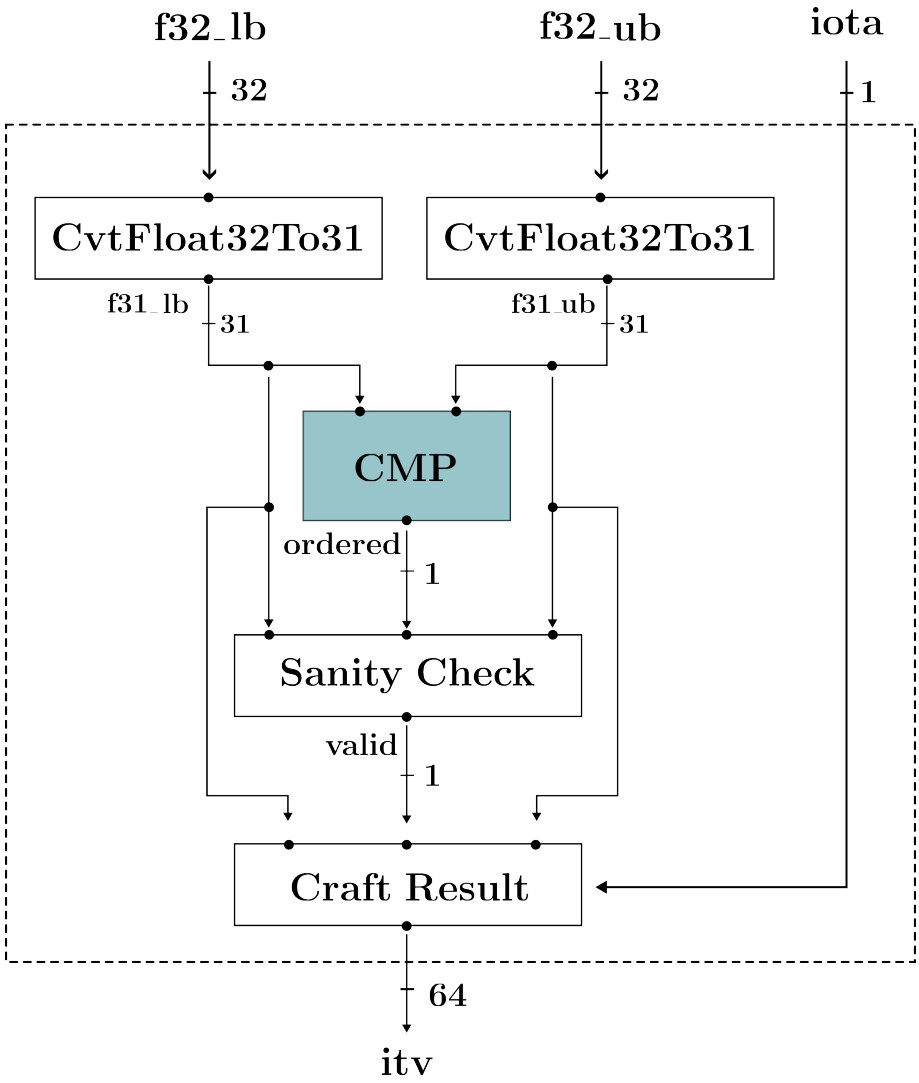


Figure 6.7: *Datapath of Itv_make*

increasing the complexity of floating-point hardware. To simplify things greatly, interval properties allows them to be classified according to the current rounding direction without removing any correct values but at the price of a bit of pessimism. This implies that a subnormal given as a lower bound will be effectively flushed to zero (round-down) and that an upper bound will be set to the smallest normal number (round-up). This leads to the classification process illustrated in Figure 6.8 where the result is a 2-bit encoding: **00** $\rightarrow$ *zero*, **01** $\rightarrow \pm\infty$, **10** $\rightarrow$ *NaN* and **11** $\rightarrow$ *normal*.

Depending of the unbiaised exponent e_u, some Float32 numbers may be either too small (e_u < -64) or too big (e_u > 64) to be represented as valid Float31. Figure 6.9 illustrates this final step of conversion. Special values such as NaN, zero, infinity are replaced by their Float31 counterparts whereas normal values go through the range analysis process which computes the biaised exponent for Float31 e_{b31}.

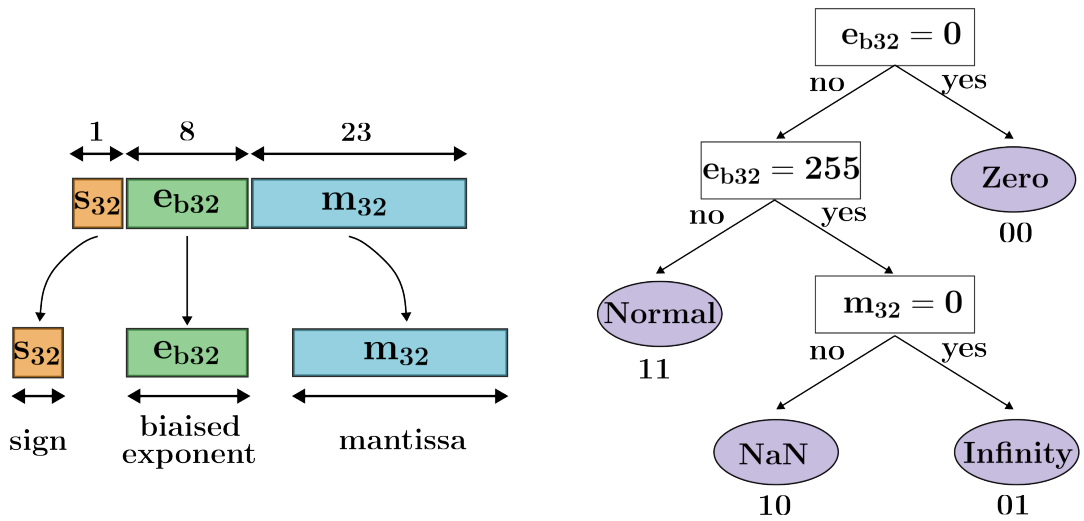


Figure 6.8: *Classification Of Float32 Bounds*

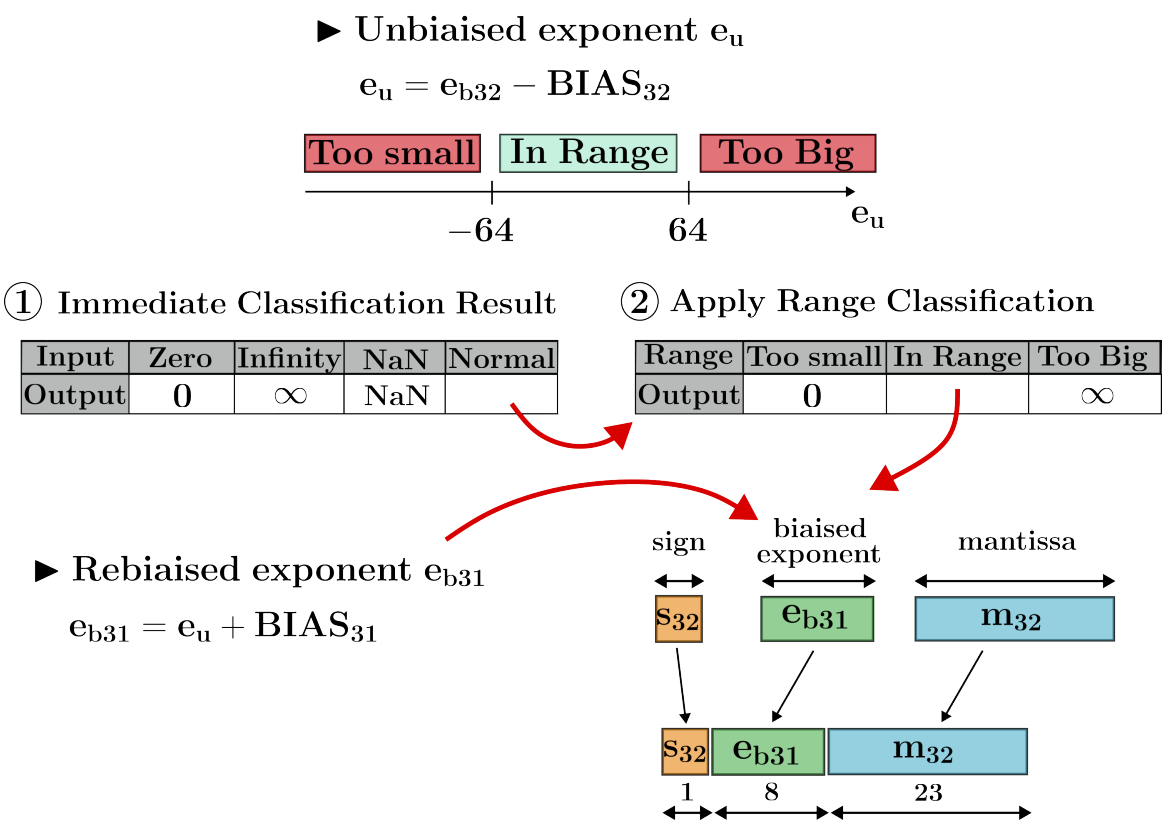


Figure 6.9: *Range Checks And Conversion Between Float32 and Float31*

Stage 2 - Check for converted Float31 bounds: the proper ordering of converted bounds is achieved using a simple FloPoCo Float31 comparison block which account for special values such as NaN and infinity.

Stage 3 - Sanitize interval output: if converted bounds happen to be unordered or if any is a NaN, a properly encoded empty set is returned, composed of an asserted empty flag

and two Float31 QNaNs for the bounds (Figure 5.6). In the other case, the converted bounds are properly packed into a 64-bit interval with the iota flag set as required.

6.2.2.3 Itv_extract: Operator Interface

Hardware Entity

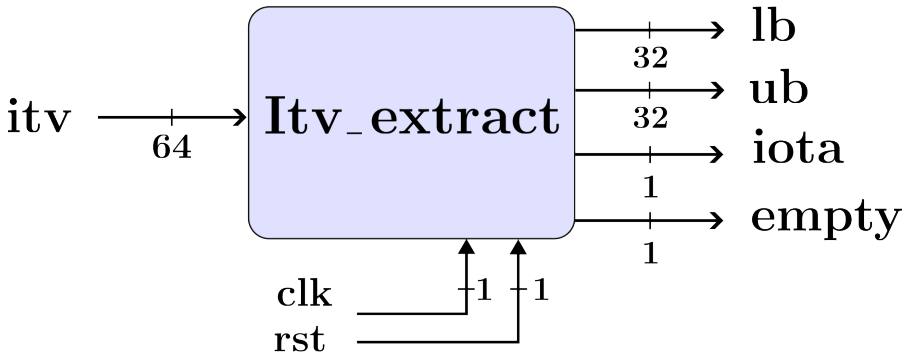


Figure 6.10: *Top-level Entity For Itv_extract*

Input Ports

- **itv:** input interval.

Output Ports

- **lb/ub:** Float32 lower (resp. upper) bound of the input interval.
- **iota/empty:** Int32 iota (resp. empty) flag of the input interval.

6.2.2.4 Itv_extract: Operator Datapath

The circuit datapath (Figure 6.11) is composed of three main stages:

- The fields of the data-structure are separated.
- Empty and iota flags are returned in RISC-V's integer registers.
- Bounds are converted from Float31 to Float32 standard and returned in RISC-V's single-precision registers.

Stage 1/2 - Field extraction + Empty/Iota flags:

The interval data-structure can be trivially split using VHDL formalism, according to Fig-

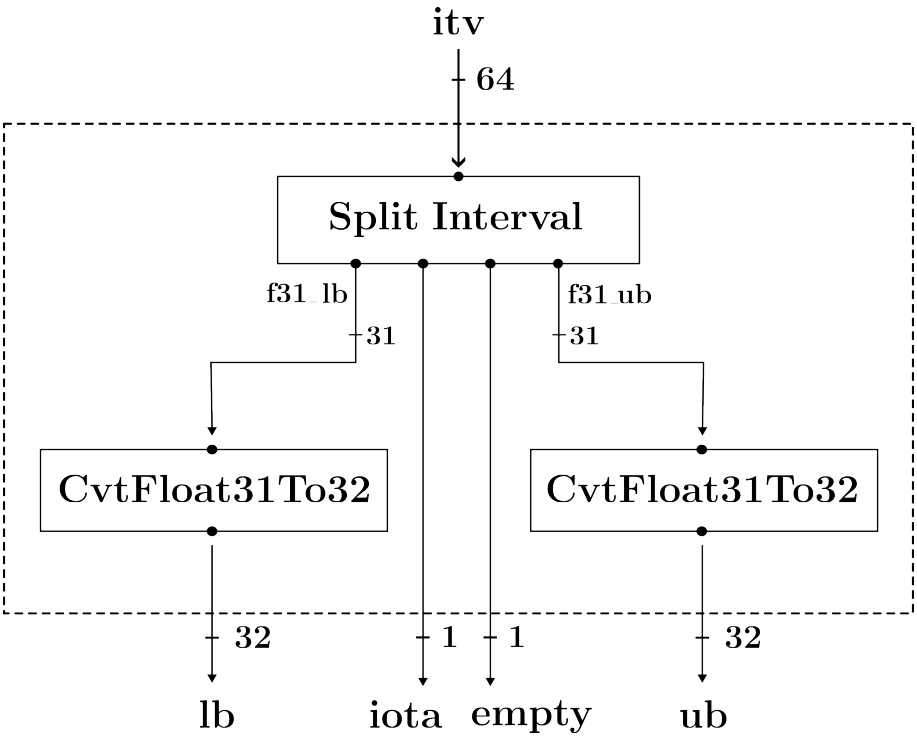


Figure 6.11: *Datapath of Itv_extract*

ure 6.12. Empty and iota flags are then extracted, sign-extended to 32-bit (**0** → *non-empty*, **1** → *empty*) and returned in integer registers.

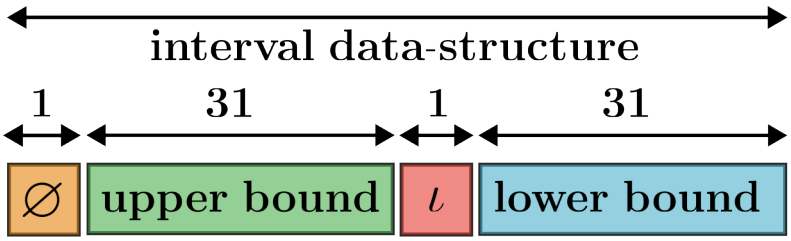


Figure 6.12: *Interval Fields Splitting*

Stage 3 - Bounds conversion back to Float32:

The conversion from Float31 to Float32 is essentially the reverse process as the one depicted in 6.9 and aims at detecting special values and computing the biaised exponent e_{b32}. However, this side is a little bit easier in the sense that all Float31 can technically be represented as Float32 numbers, which makes range checks unnecessary.

6.2.3 Itv_set

The **itv_set** primitive is the hardware backend for instructions **itv.set.u**, **itv.set.i** whose behaviors have been been presented in Section 5.3.4. It implements the interval version

of the intersection and union set operations. The former is notably used to compute the backward contractors associated to the hald-contractors provided by most *xinterval* instructions and the latter is useful for many robotics problems.

6.2.3.1 Operator Interface

Hardware Entity

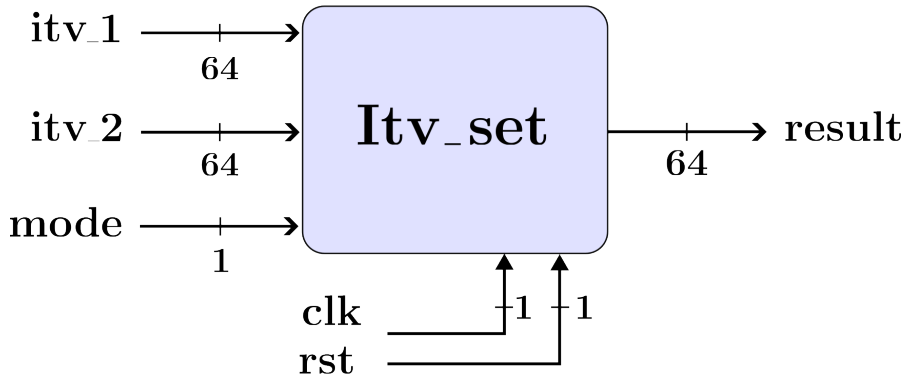


Figure 6.13: *Top-level Entity For Itv_set*

Input Ports

- **itv_1:** interval operand 1.
- **itv_2:** interval operand 2.
- **mode: 0** $\rightarrow$ *union*,
  1 $\rightarrow$ *intersection*.

Output Ports

- **result:** output interval.

6.2.3.2 Operator Datapath

The circuit datapath (Figure 6.14) is composed of two main stages:

- Relevant comparisons between the bounds of both operands are performed.
- The final interval is created using the result of comparators and the values of empty/iota flags of both operands.

Stage 1 - Comparisons:

Several FloPoCo Float31 comparators are instantiated to concurrently evaluate the ordering of lb_1/lb_2, ub_1/lb_2, ub_1/ub_2, lb_1/ub_2. The goal is to detect the current situation from all the possibilities illustrated in Figure 5.10.

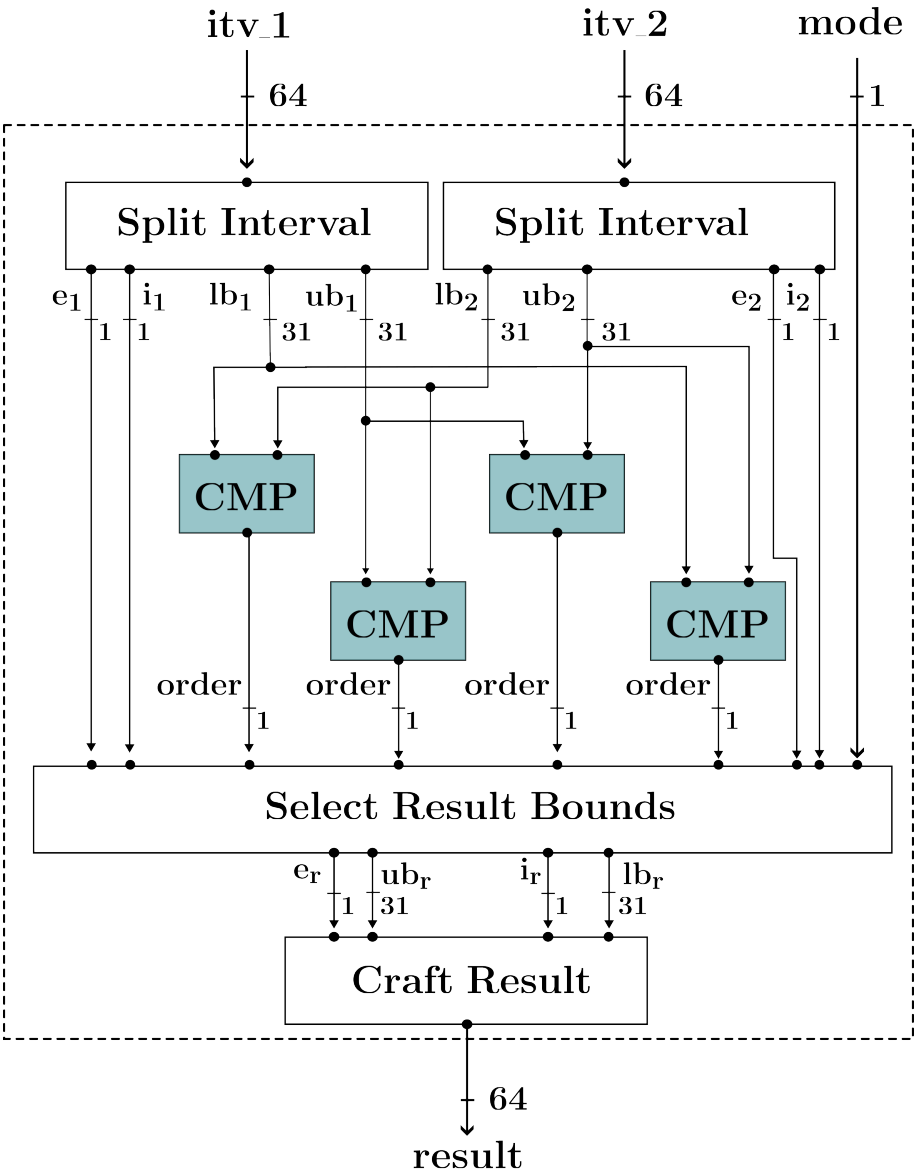


Figure 6.14: *Datapath of Itv_set*

Stage 2 - Final result:

The rules applicable to the selected operation mode are applied with respect to the results obtained in the comparison stage or the potential empty nature of one/both operands. The value of the final iota flag is obtained by OR-ing (union) or AND-ing (intersection) the iota flags of both inputs.

6.2.4 Itv_add

The **itv_add** primitive is the hardware backend for instructions **itv.add** and **itv.sub** which are two half-contractors based on arithmetic operators as presented in Section 5.4.1. The advantage of the hardware implementation is that both bounds of the result can be

evaluated concurrently thanks to a dual instantiation of FloPoCo entities.

6.2.4.1 Operator Interface

Hardware Entity

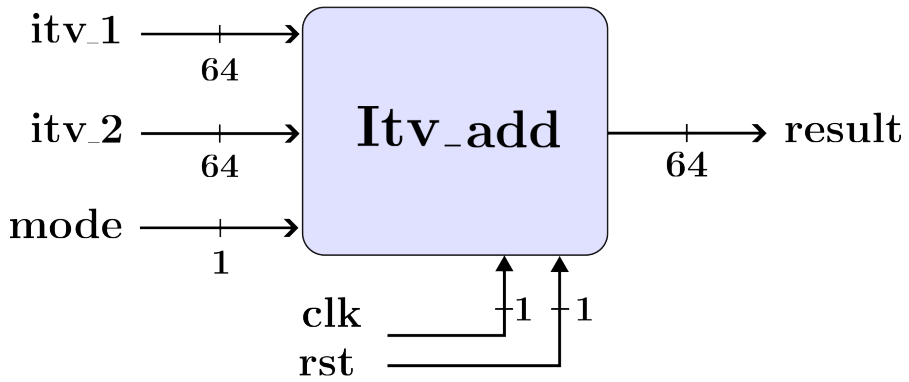


Figure 6.15: *Top-level Entity For Itv_add*

Input Ports

- **itv_1:** interval operand 1.
- **itv_2:** interval operand 2.
- **mode: 0** → *addition*,
  1 → *subtraction*.

Output Ports

- **result:** output interval.

6.2.4.2 Operator Datapath

The circuit datapath (Figure 6.16) is composed of two main stages:

- Concurrent evaluation of both additions (resp. subtractions).
- The final interval is created using the result of additions and the values of empty/iota flags of both operands.

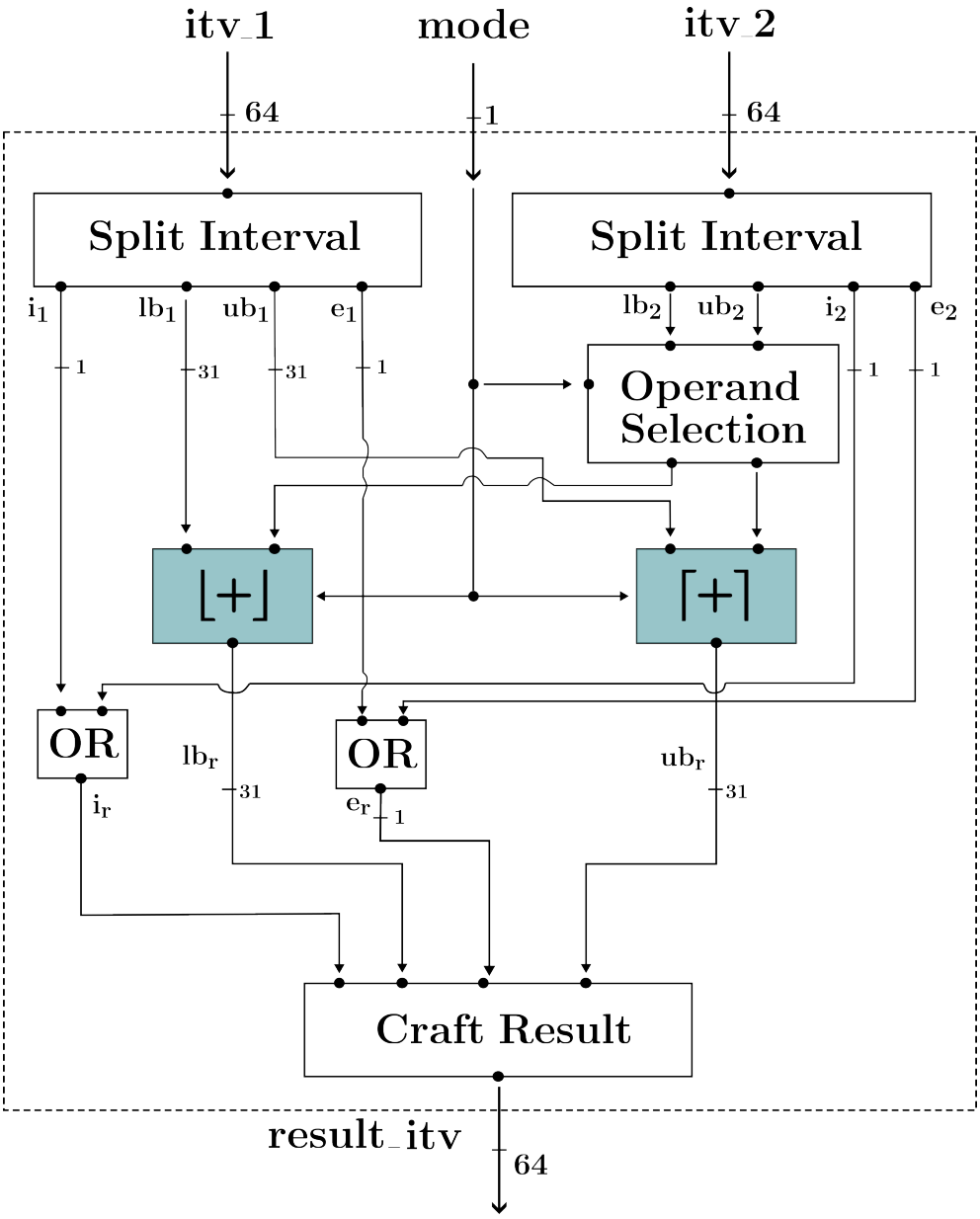


Figure 6.16: *Datapath of Itv_add*

Stage 1 - Adders:

Two FloPoCo adders are instantiated in order to evaluate both bounds concurrently, using respectively the «round toward $-\infty$» mode for the lower bound and the «round toward $+\infty$» for the upper bound. This strategy is made possible by the use of custom hardware and allows to compute a guaranteed estimation of the output interval. An operand selection logic is used to give the proper second operand to each adder depending of the current operation (addition/substraction):

- **Addition:** adder_lb : $lb_1 + lb_2$, adder_ub : $ub_1 + ub_2$.
- **Subtraction:** adder_lb : $lb_1 - ub_2$, adder_ub : $ub_1 - lb_1$.

Stage 2 - Final result:

If at least one of the operands is ∅, the result is set to ∅ are required by the IEEE-1788 standard. In other cases, the result of adders are used for the bounds of the output interval.

6.2.5 Itv_mul

The **itv_mul** primitive is the hardware backend for instructions **itv.mul** and **itv.sqr**. It is practical to handle these operations in the same circuit since they are closely related but with subtle differences, as illustrated in the remarks of Section 1.2.4.5.

6.2.5.1 Operator Interface

Hardware Entity

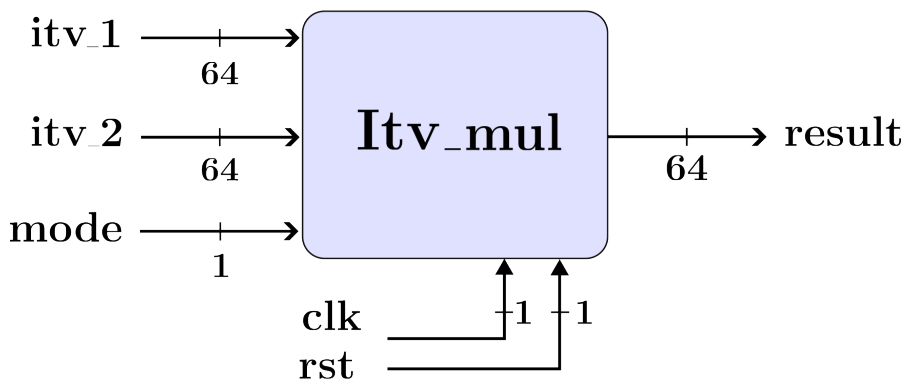


Figure 6.17: *Top-level Entity For Itv_mul*

Input Ports

- **itv_1:** interval operand 1.
- **itv_2:** interval operand 2.
- **mode: 0** → *multiplication*,
  1 → *square*.

Output Ports

- **result:** output interval.

6.2.5.2 Operator Datapath

The circuit datapath (Figure 6.19) is composed of three main stages:

- Classification of interval operands.
- Concurrent evaluation of multiple multiplications + reordering.
- The right bounds are selected from all the candidates and properly re-rounded.

Stage 1 - Interval operands classification:

To prepare for the upcoming multiplications, each interval operand must be classified in one of the format depicted in Figure 6.18, which is achieved by the means of FloPoCo comparators. The result is a 3-bit encoding indicating that the input is:

- **000:** the empty set $\varnothing$.
- **001:** the interval $\{0\}$.
- **010:** the interval $\mathbb{R}$.
- **011:** an interval included in $\mathbb{R}^-$.
- **100:** an interval included in $\mathbb{R}^+$.
- **101:** an interval encompassing 0.

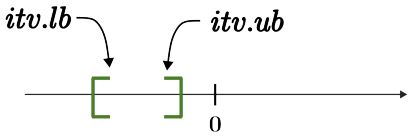
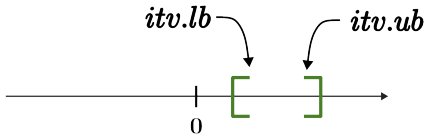
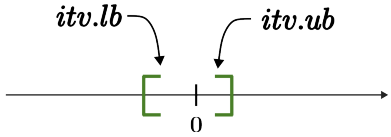
code	format	code	format
000	$\varnothing$	011	
001	$\{0\}$	100	
010	$[-\infty, +\infty]$	101	

Figure 6.18: *Classification Process for Interval Input in Itv_mul*

Stage 2 - Multiplications and re-ordering:

Depending of the class of both interval operands (stage 1), the bounds of the final result may take at most four values which are computed concurrently using the same number of FloPoCo multiplication blocks. At this step, there is no way to predict which pair of products will be finally chosen and thus all the computations are temporarily performed using the «round-to-nearest mode». Two additional comparisons between some of the bound candidates are also made to prepare for the final step.

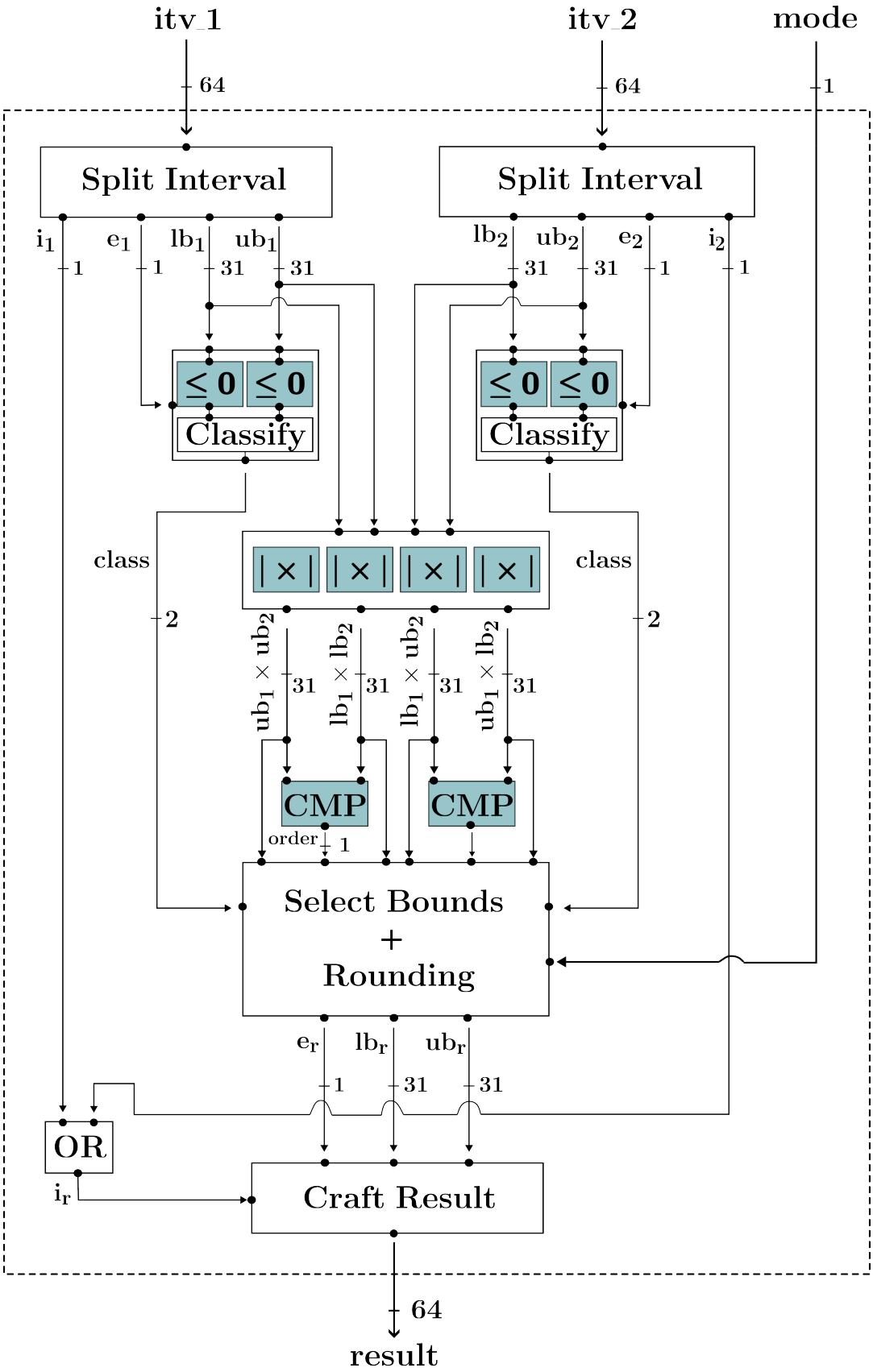


Figure 6.19: *Datapath of Itv_mul*

Stage 3 - Generation of the final result:

The bounds of the final output depend of several factors. The first one is the class of both interval operands (stage 1), the second is the value of all the bound candidates as well as their ordering (stage 2). The last factor is the kind of operation requested:

For the multiplication mode: trivial cases involving $\varnothing$ or $\{0\}$ input are resolved in accordance with the IEEE-1788 standard. The regular cases follow the rule depicted in Figure 6.20.

For the square mode: trivial cases involving $\varnothing$ or $\{0\}$ input are resolved in accordance with the IEEE-1788 standard. The regular cases follow the rule depicted in Figure 6.21.

Once selected among all the candidates, the final bounds are correctly re-rounded toward $-\infty$ (for a lower bound) or $+\infty$ (for an upper bound) to ensure that the result is guaranteed.

itv2 \ itv1	011	100	101
011	$[\mathrm{ub}_1\mathrm{ub}_2, \mathrm{lb}_1\mathrm{lb}_2]$	$[\mathrm{ub}_1\mathrm{lb}_2, \mathrm{lb}_1\mathrm{ub}_2]$	$[\mathrm{ub}_1\mathrm{lb}_2, \mathrm{lb}_1\mathrm{lb}_2]$
100	$[\mathrm{lb}_1\mathrm{ub}_2, \mathrm{ub}_1\mathrm{lb}_2]$	$[\mathrm{lb}_1\mathrm{lb}_2, \mathrm{ub}_1\mathrm{ub}_2]$	$[\mathrm{lb}_1\mathrm{ub}_2, \mathrm{ub}_1\mathrm{ub}_2]$
101	$[\mathrm{lb}_1\mathrm{ub}_2, \mathrm{lb}_1\mathrm{lb}_2]$	$[\mathrm{ub}_1\mathrm{lb}_2, \mathrm{ub}_1\mathrm{ub}_2]$	$[\min(\mathrm{lb}_1\mathrm{ub}_2, \mathrm{ub}_1\mathrm{lb}_2), \max(\mathrm{lb}_1\mathrm{lb}_2, \mathrm{ub}_1\mathrm{ub}_2)]$

Figure 6.20: *Regular Cases for Multiplication*

itv1	011	100	101
result	$[\mathrm{ub}_1\mathrm{ub}_2, \mathrm{lb}_1\mathrm{lb}_2]$	$[\mathrm{lb}_1\mathrm{lb}_2, \mathrm{ub}_1\mathrm{ub}_2]$	$[0, \max(\mathrm{lb}_1\mathrm{lb}_2, \mathrm{ub}_1\mathrm{ub}_2)]$

Figure 6.21: *Regular Cases for Square*

6.2.6 Itv_div

The **itv_div** primitive is the hardware backend for the instruction **itv.div**, which implements the half-contractor associated with the division operator as shown in Section 5.3.4.

6.2.6.1 Operator Interface

Hardware Entity

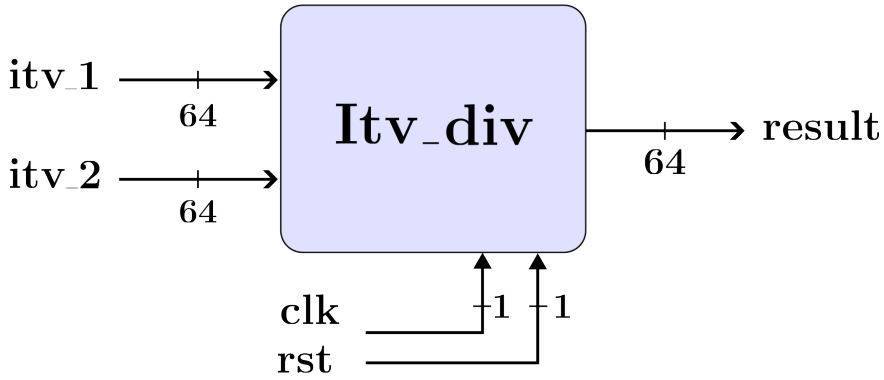


Figure 6.22: *Top-level Entity For Itv_div*

Input Ports

- **itv_1:** interval operand 1.
- **itv_2:** interval operand 2.

Output Ports

- **result:** output interval.

6.2.6.2 Operator Datapath

The circuit datapath (Figure 6.23) is composed of three main stages:

- Classification of interval operands.
- Concurrent evaluation of multiple divisions.
- The right bounds are selected from all the candidates and properly re-rounded.

Stage 1 - Interval operands classification:

The first stage classifies interval operands, which is a slightly more complex process for division compared to multiplication due to the assymetry brought by the numerator and denominator. The 3-bit encoding corresponding to the classification of **itv_1** (the numerator) and **itv_2** (the denominator) are respectively given in Figure 6.24 and 6.25.

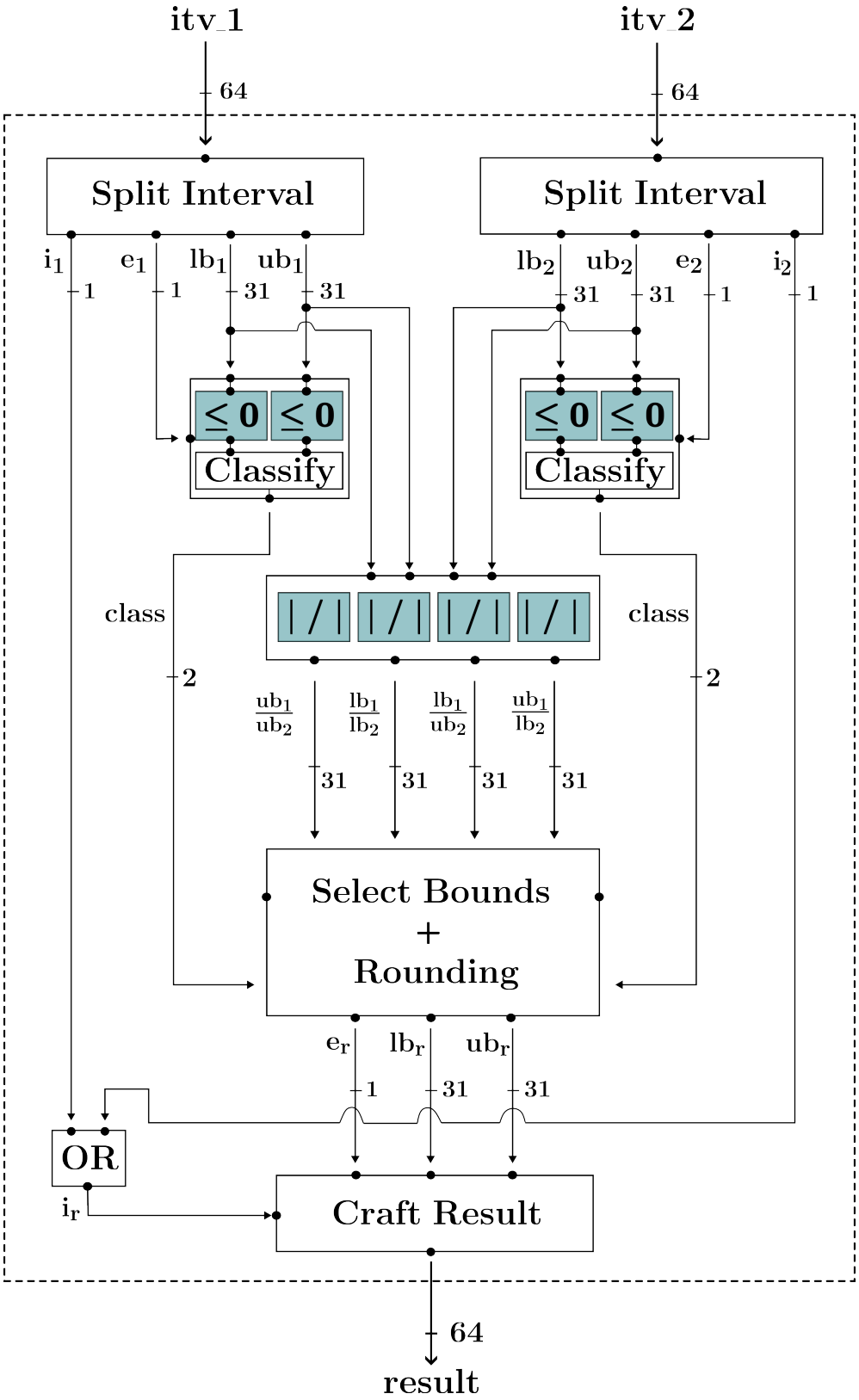


Figure 6.23: *Datapath of Itv_div*

.

The chosen encoding is summarized below:

Numerator classification:

- **000:** the numerator is the empty set $\varnothing$.
- **001:** the numerator is the interval $\{0\}$.
- **010:** the numerator is included in $\mathbb{R}^-$.
- **011:** the numerator is included in $\mathbb{R}^+$.
- **100:** the numerator encompasses 0.

Denominator classification:

- **000:** the denominator is the empty set $\varnothing$.
- **001:** the denominator is the interval $\{0\}$.
- **010:** the denominator has $u_b = 0$.
- **011:** the denominator has $l_b = 0$.
- **100:** the denominator is included in $\mathbb{R}^-$ and $u_b \neq 0$.
- **101:** the denominator encompasses 0.
- **100:** the denominator is included in $\mathbb{R}^+$ and $l_b \neq 0$.

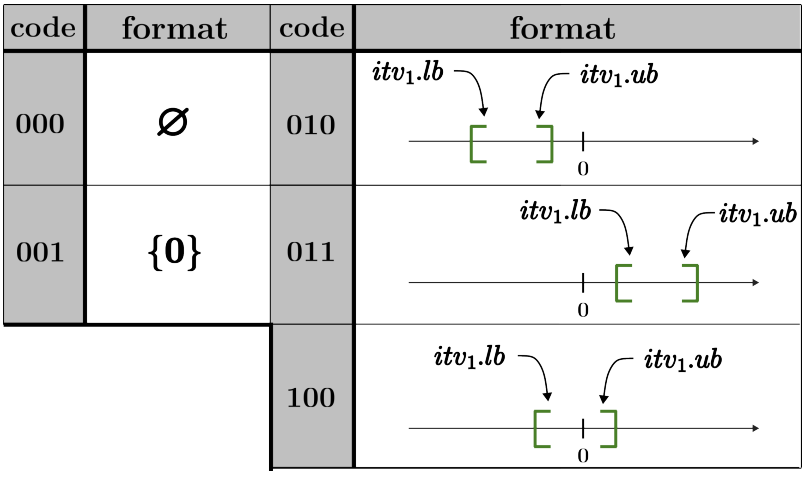


Figure 6.24: *Classification Process for Interval Input itv_1 (numerator) in Itv_div*

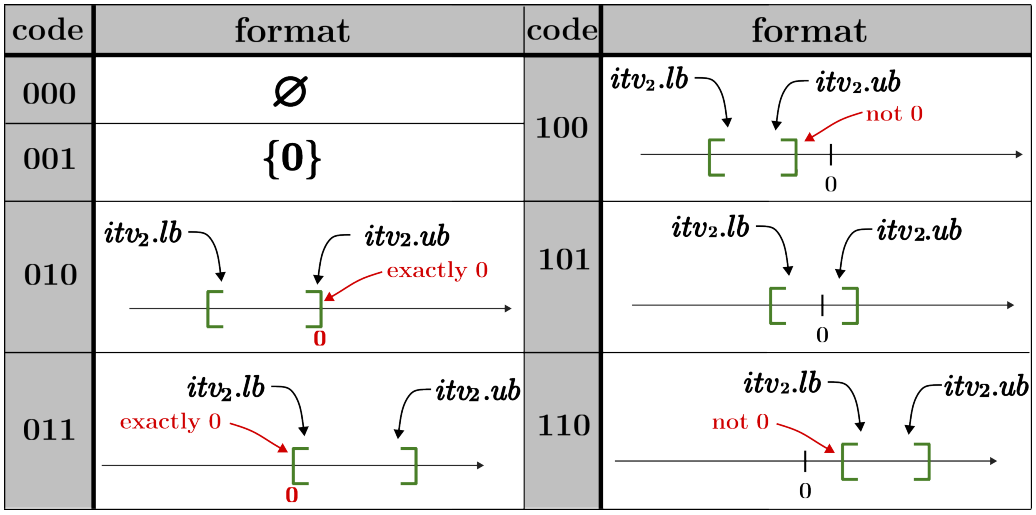


Figure 6.25: *Classification Process for Interval Input itv_2 (denominator) in Itv_div*

Stage 2 - Divisions

Depending of the class of both interval operands (stage 1), the bounds of the final result may take at most four values which are computed concurrently using the same number of FloPoCo division blocks. At this step, there is no way to predict which pair will be finally chosen and thus all the computations are temporarily performed using the «round-to-nearest» mode.

Stage 3 - Generation of the final result:

Using the classification of numerator and denominator (stage 1) and the four possible bounds obtained through divisions (stage 2), the bounds of the result are obtained according to the Figure 6.26. Once selected among all the candidates, the final bounds are correctly re-rounded toward $-\infty$ (for a lower bound) or $+\infty$ (for an upper bound) to ensure that the result is guaranteed.

itv2 \ itv1	000	001	010	011	100
000	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
001	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
010	$\varnothing$	$\{0\}$	$[\frac{ub_1}{lb_2}, +\infty]$	$[-\infty, \frac{lb_1}{lb_2}]$	$[-\infty, +\infty]$
011	$\varnothing$	$\{0\}$	$[-\infty, \frac{ub_1}{ub_2}]$	$[\frac{lb_1}{ub_2}, +\infty]$	$[-\infty, +\infty]$
100	$\varnothing$	$\{0\}$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$
101	$\varnothing$	$\{0\}$	$[\frac{ub_1}{lb_2}, \frac{lb_1}{ub_2}]$	$[\frac{ub_1}{ub_2}, \frac{lb_1}{lb_2}]$	$[\frac{ub_1}{ub_2}, \frac{lb_1}{ub_2}]$
110	$\varnothing$	$\{0\}$	$[\frac{lb_1}{lb_2}, \frac{ub_1}{ub_2}]$	$[\frac{lb_1}{ub_2}, \frac{ub_1}{lb_2}]$	$[\frac{lb_1}{lb_2}, \frac{ub_1}{lb_2}]$

Figure 6.26: *Regular Cases for Division*

6.2.7 Itv_sqrt

The **itv_sqrt** primitive is the hardware backend for the instructions **itv.sqrt** and **itv.sqr_rev**. The first one implements the interval version of the square-root function while the second is the half-contractor associated with the backward propagation of the square function. Their respective behaviors have been described in Section 5.4.2.

6.2.7.1 Operator Interface

Hardware Entity

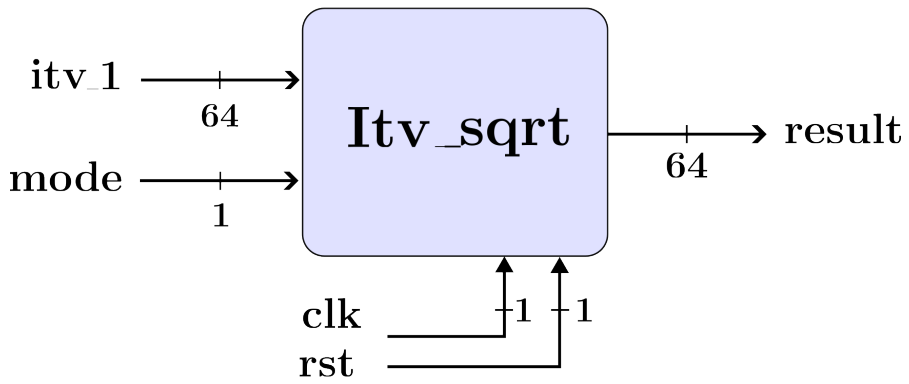


Figure 6.27: *Top-level Entity For Itv_sqrt*

Input Ports

- **itv_1:** interval operand 1.
- **mode: 0** → *square-root*,
  1 → *square_rev*.

Output Ports

- **result:** output interval.

6.2.7.2 Operator Datapath

The circuit datapath (Figure 6.28) is composed of two main stages:

- Concurrent evaluation of two square-root.
- Generation of the output depending of the mode/input.

Stage 1 - Square-roots

Two FloPoCo square-root operators evaluate concurrently the lower and upper bounds of the result using the proper rounding modes, respectively «round to $-\infty$» and «round

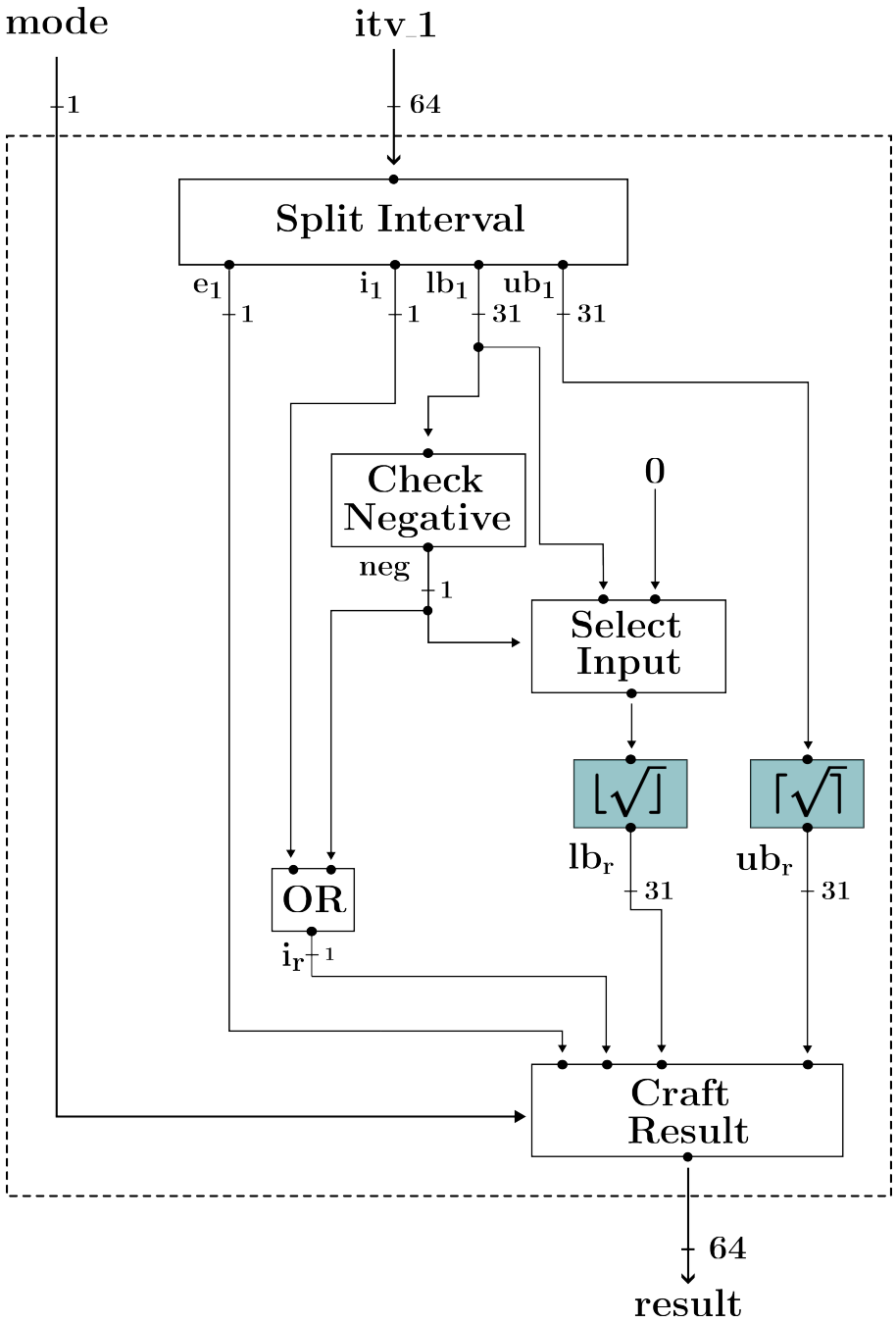


Figure 6.28: *Datapath of Itv_sqrt*

to $+\infty$». The input of the square-root block associated to the lower bound is generally set to the lower bound of the input but may be hardwired to zero if the latter is negative, as required by Algorithm 3.

Stage 2 - Generation of the final result:

Special cases involving $\varnothing$ or $\{0\}$ are handled with respect to the IEEE-1788 standard. Regular cases depend of the output of both square-root oeprators and the operation selected:

For the square-root mode: the bounds are selected accordingly to Algorithm 3.

For the backward-square mode: the output interval is set to $[-\sqrt{\overline{y}}, \sqrt{\overline{y}}]$, where y is the interval input.

6.2.8 Itv_exp

The **itv_exp** primitive is the hardware backend for the instructions **itv.exp** which implements the interval version of the exponential function whose behavior has been described in Section 5.4.2.

6.2.8.1 Operator Interface

Hardware Entity

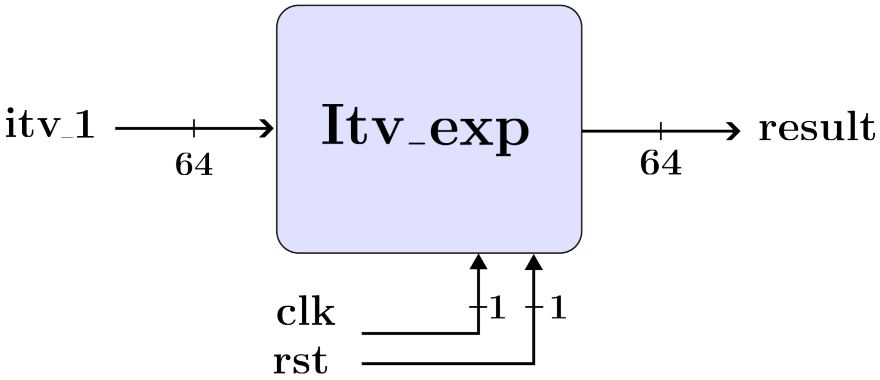


Figure 6.29: *Top-level Entity For Itv_exp*

Input Ports

- **itv_1:** interval operand 1.

Output Ports

- **result:** output interval.

6.2.8.2 Operator Datapath

Since the exponential function is fully defined and strictly monotonous over $\mathbb{R}$ (refer to Section 1.2.2.3), the datapath of the circuit (Figure 6.30) simply instantiates two FloPoCo exponential blocks (properly rounded) and using their output as the bounds of the final results. Special cases involving $\varnothing$ or $\{0\}$ are once again handled with respect to the IEEE-1788 standard.

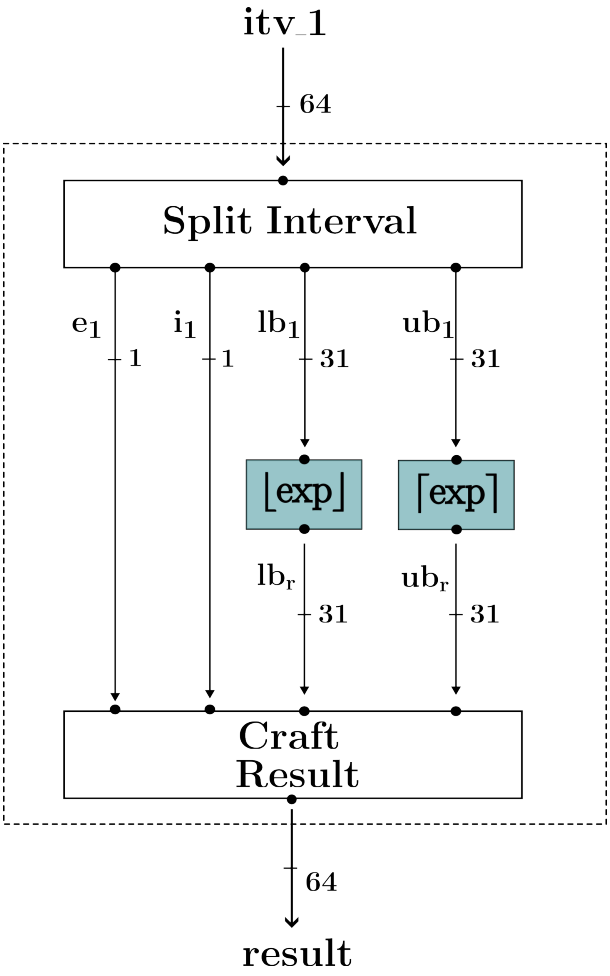


Figure 6.30: *Datapath of Itv_exp*

6.2.9 Itv_log

The **itv_log** primitive is the hardware backend for the instruction **itv.log** which implements the interval version of the logarithm function (refer to Section 5.4.2).

Hardware Entity

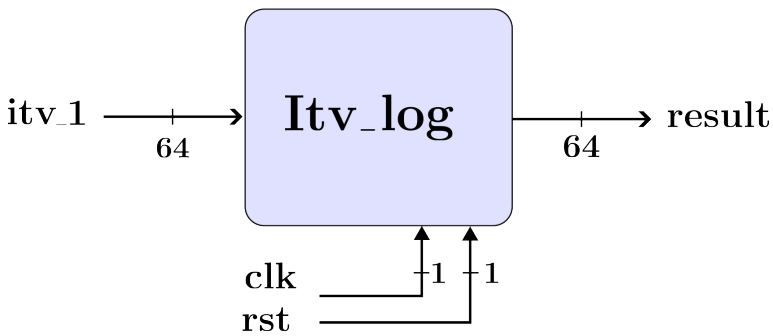


Figure 6.31: *Top-level Entity For Itv_log*

Hardware Entity (cont'd)

Input Ports

- **itv_1:** interval operand 1.

Output Ports

- **result:** output interval.

6.2.9.1 Operator Datapath

The circuit datapath (Figure 6.32) is relatively straightforward and mainly instantiates two FloPoCo logarithm blocks with the proper rounding modes. Similarly to square-root, a preliminary step replaces the lower bound of the interval operand by zero as the entry of the first logarithm block because the function is not defined for $\mathbb{R}^-$.

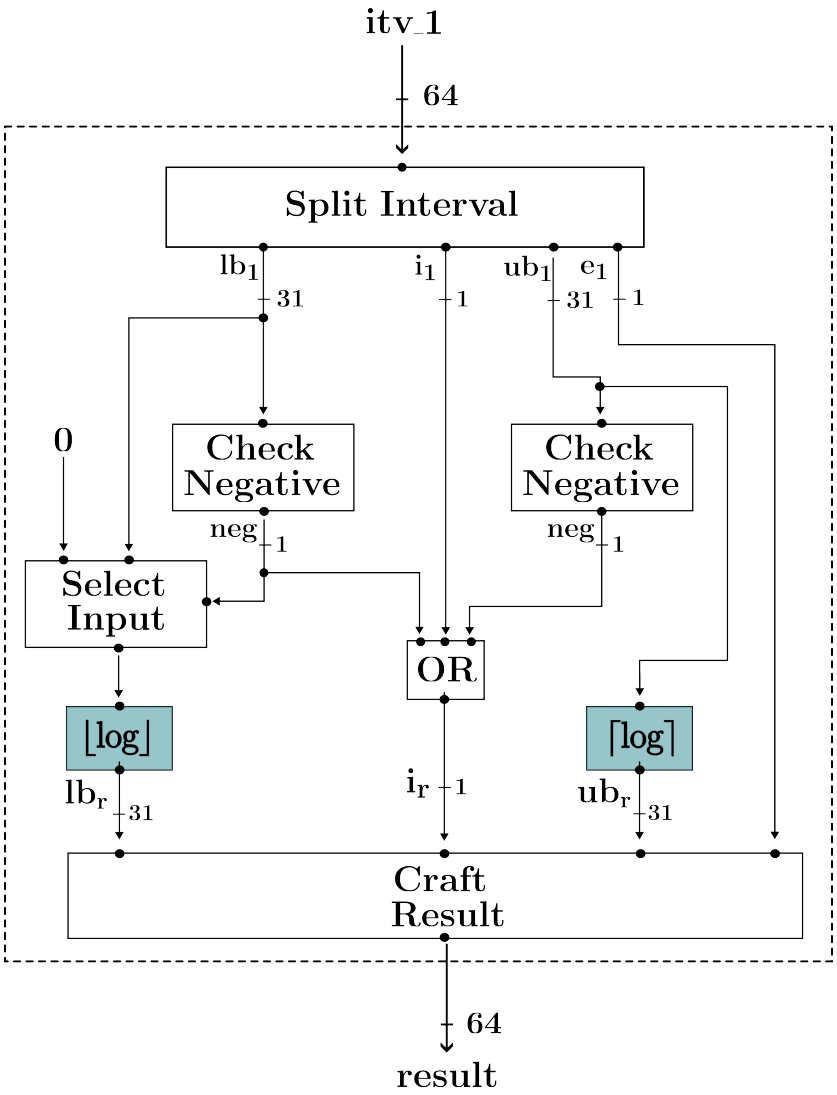


Figure 6.32: *Datapath of Itv_log*

6.2.10 Itv_trig and Itv_trig_inv

Trigonometry on intervals is a recurring part of many robotics applications and is integrated in the micro-archtitecture by the mean of two hardware primitives: **itv_trig** and **itv_trig_inv**. As the names suggest, these circuits respectively implement the interval versions of the sine/cosine functions (for instructions **itv.sin**/**itv.cos**) and the arcsine/arccosine functions (for instructions **itv.arcsin**/**itv.arccos**). Their purpose is to accelerate the most resource-demanding parts in the computation of the forward/backward contractors for constraints of type $y = \sin(x)$ or $y = \cos(x)$ (refer to Section 5.4.2).

6.2.10.1 Operator Interface for Itv_trig

Hardware Entity

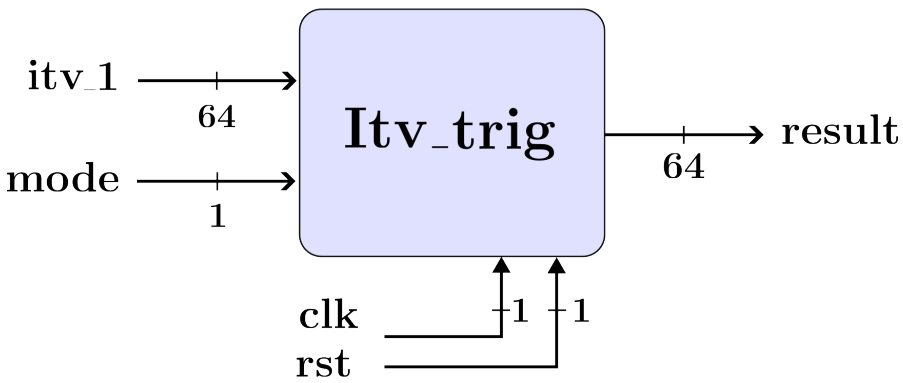


Figure 6.33: *Top-level Entity For Itv_trig*

Input Ports

- **itv_1:** interval operand 1.
- **mode: 0** $\rightarrow$ *cosine*,
  1 $\rightarrow$ *sine*.

Output Ports

- **result:** output interval.

6.2.10.2 Datapath for Itv_trig

The circuit datapath (Figure 6.34) is composed of three main stages:

- Argument reduction for the input bounds to ease the upcoming computations.
- Concurrent evaluation of sine/cosine on both bounds using CORDIC blocks (fixed-point arithmetic).
- Reordering of CORDIC outputs and generation of the final result.

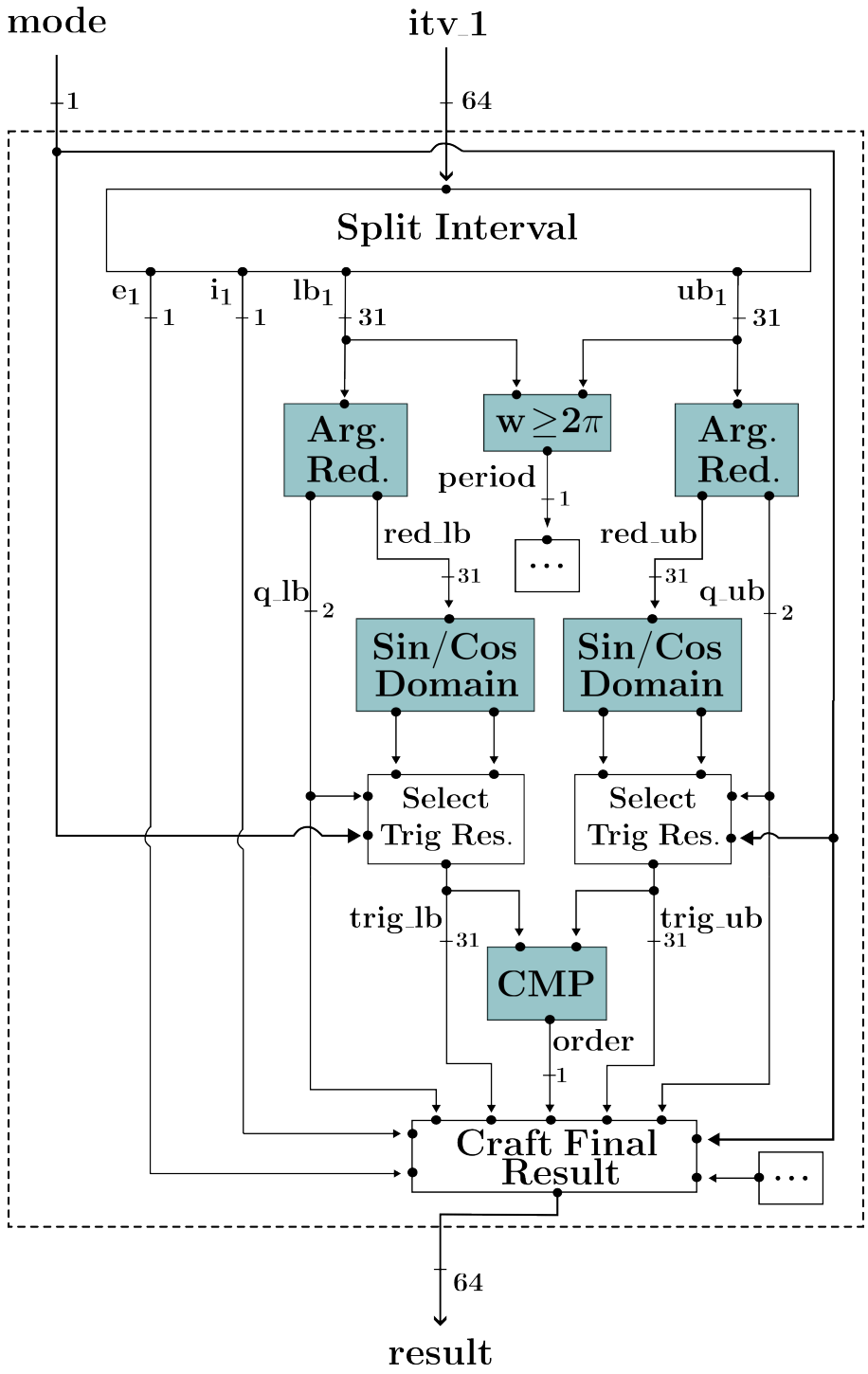


Figure 6.34: *Datapath of Itv_trig*

Stage 1 - Argument reduction

The datapath of the argument reduction block introduced in Figure 6.34 is fully represented in Figure 6.35. A first goal of this circuit is to produce the 2-bit signal $\mathbf{q}$ which maps the input Float32 bounds (denoted $\mathbf{x}$) to the associated quadrant of the trigonometric circle. Practically, this value is obtained by multipling the input by the constant

$\frac{2}{\pi}$, flooring the result and applying a mod-4. The second goal is to compute the signal $\mathbf{x_{red}} = \frac{\mathbf{x_r}}{\pi}$ where $\mathbf{x_r}$ is the mod-2π equivalent to $\mathbf{x}$ in the window $[-\pi, \pi]$. The Cody-Waite algorithm [121], [122] is a traditional solution to perform this kind of range-reduction and especially well suited for moderately large input including Float32/Float31. The goal is to give an estimate of b and k such as $b = a - kC$, where a and C are respectively the initial floating point argument and the constant of the range reduction. The interest of Cody-Waite's range-reduction is to decompose the C constant as $C = C_1 + C_2$ to achieve better precisions. At the hardware level, the computation is performed using two FloPoCo FMA blocks.

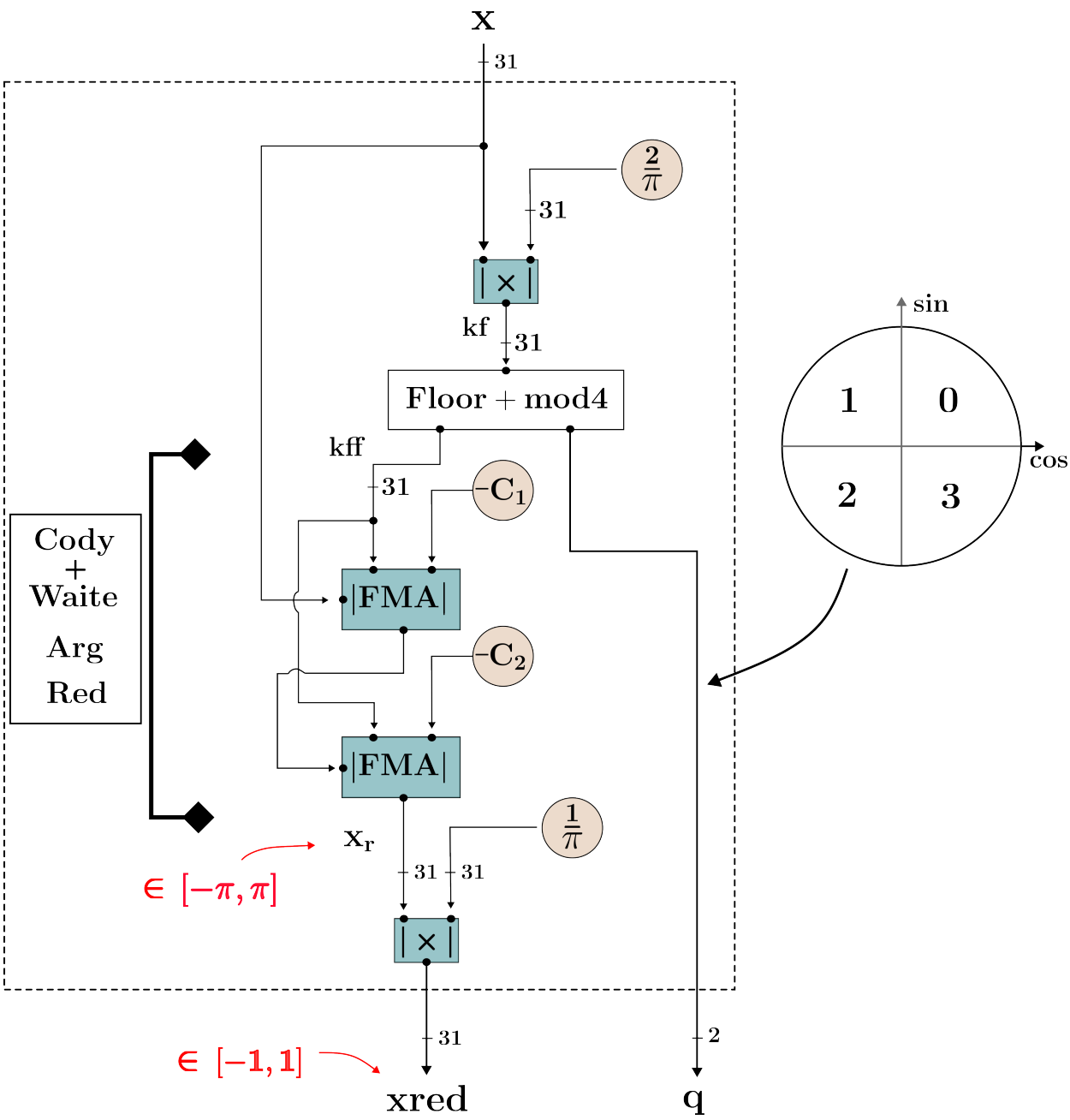


Figure 6.35: *Datapath of argument reduction for itv_trig*

Stage 2 - Concurrent evaluation of sine/cosine for both bounds

The CORDIC implemented in this design uses the traditional range-reduction and iteration technique depicted in [123]. It operates on the reducted argument $\mathbf{x_{red}}$ evaluated during stage 1 (contained in $[-1, 1]$) to computes simultaneously $\sin(\mathbf{x_{red}})$ and $\cos(\mathbf{x_{red}})$. However, the input must be converted into the Q1.23 fixed-point format, which uses 1-bit for the integer part and 23 bits for the fractional part. The inverse conversion is performed after the computation from Q1.23 back to Float31. The datapath for the whole process is presented in Figure 6.36.

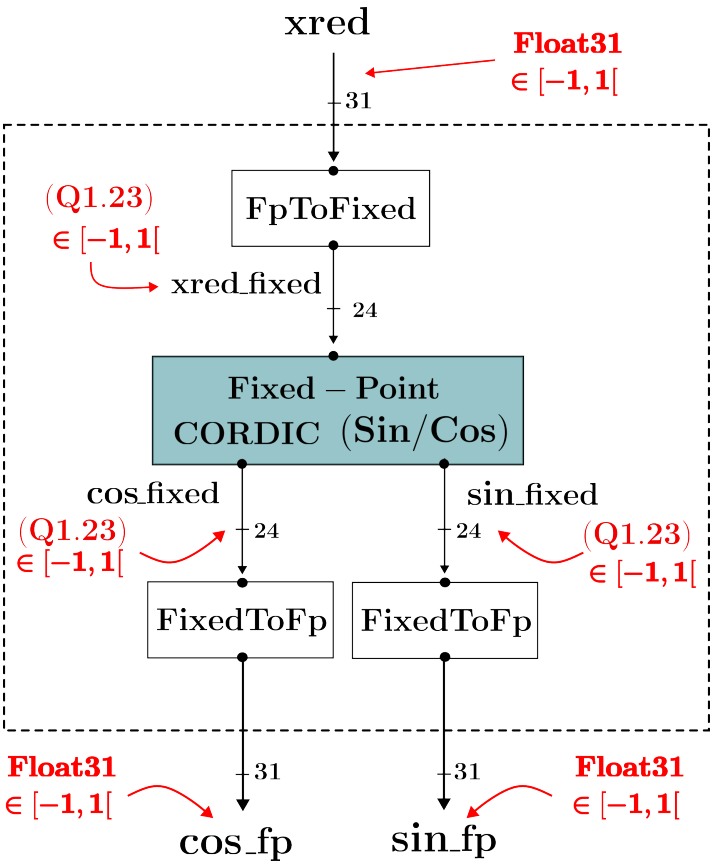


Figure 6.36: *Datapath of CORDIC evaluation for itv_trig*

Stage 3 - Generation of the final result:

With the knowledge of the trigonometric quadrant and the values of sine/cosine associated to input bounds (stage 1 and 2), it is now possible to produce the final output. First, a trivial case happens if the width of the input interval is superior to 2π and generates the $[-1, 1]$ output interval. For regular cases, the result will mostly depend of the combination of $\mathbf{q_{lb}}$ and $\mathbf{q_{ub}}$ which are respectively the trigonometric quadrant of the input lower bound and upper bound. The most simple situation occurs when $\mathbf{q_{lb}} \neq \mathbf{q_{ub}}$ and is resolved as shown in Figure 6.37.

In the event where $\mathbf{q_{lb}} = \mathbf{q_{ub}}$, the results obtained after the CORDIC (for example $\sin(x_{lb})$

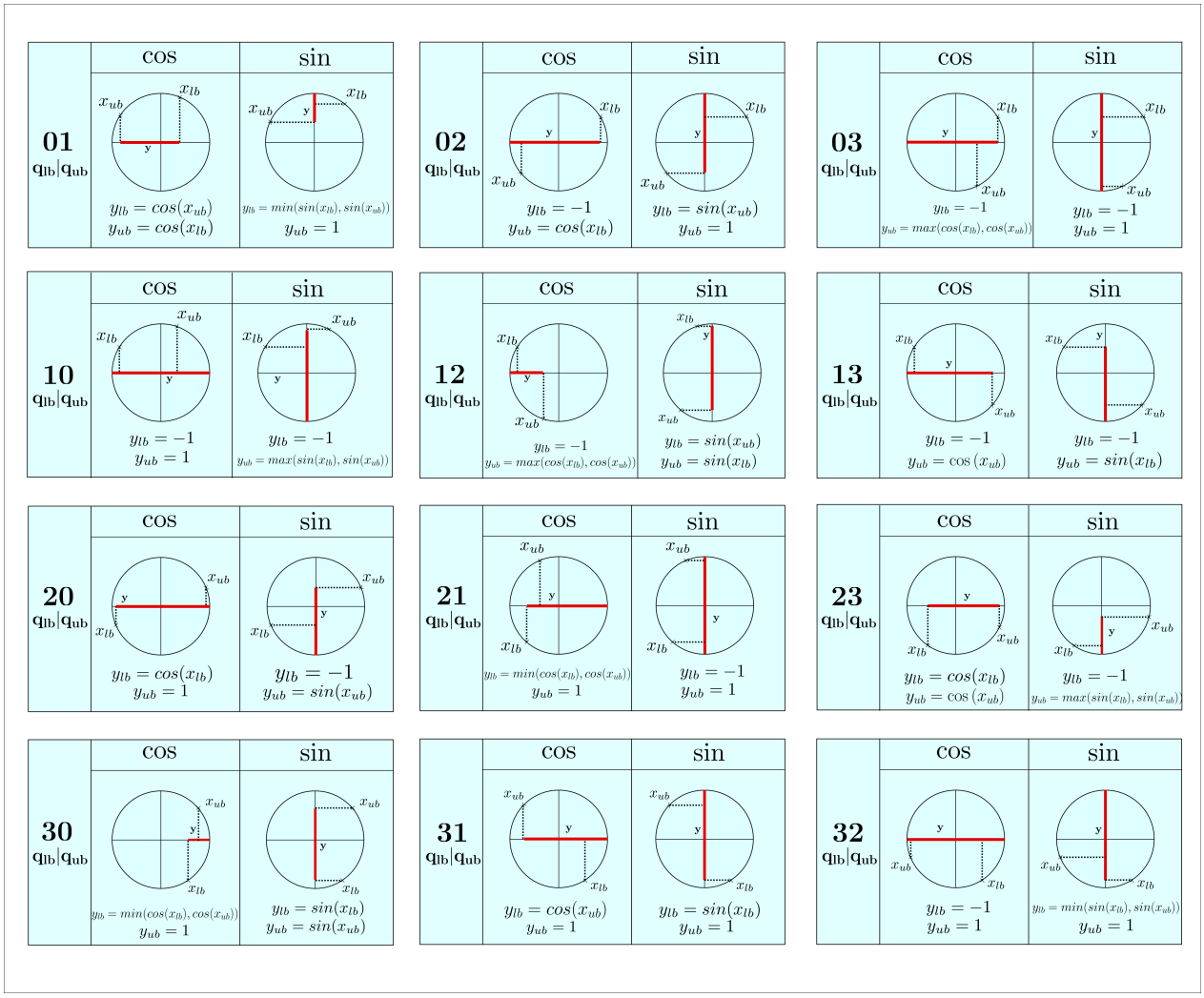


Figure 6.37: *Interval sine/cosine resolution when $q_{lb} \neq q_{ub}$*

and $\sin(x_{ub})$) need to be properly ordered using a dedicated comparator shown in Figure 6.34. The resolution is then performed according to Figure 6.38. The output for particular cases such as $\varnothing$ or $\{0\}$ are handled accordingly to the the IEEE-1788 standard and the iota flag of the output is equal to the one of the input.

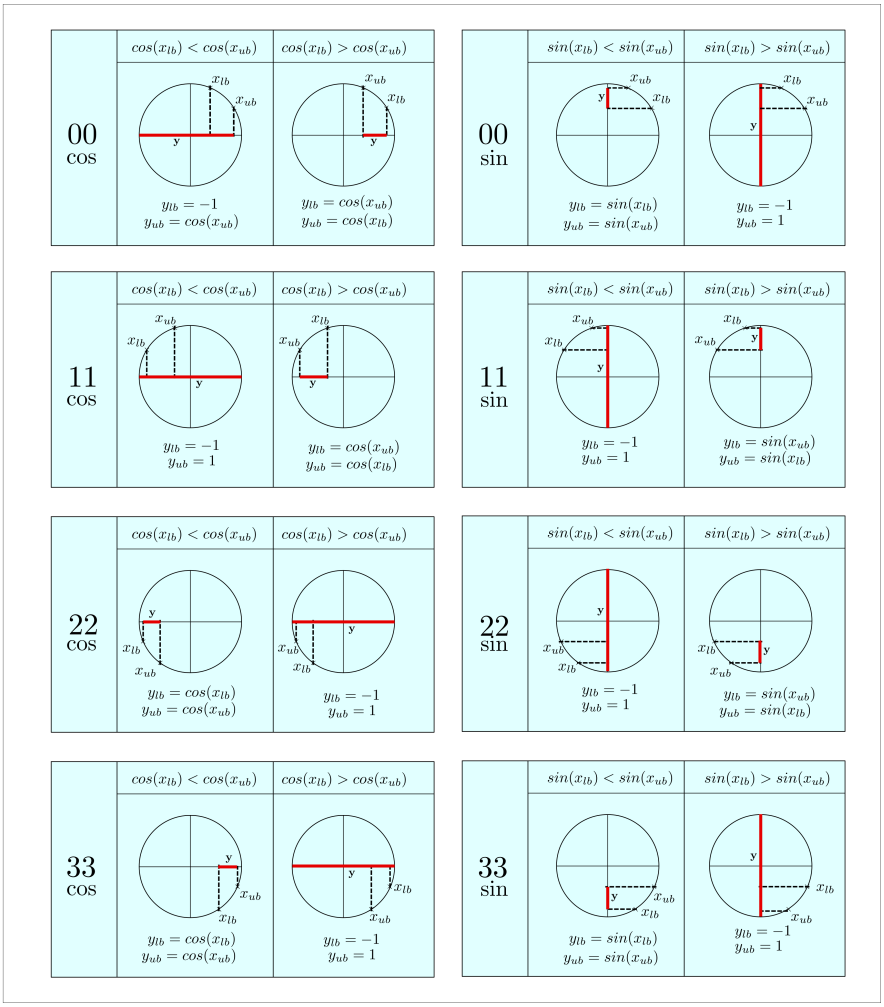


Figure 6.38: *Interval sine/cosine resolution when $q_{lb} = q_{ub}$*

6.2.10.3 Operator Interface for Itv_trig_inv

Hardware Entity

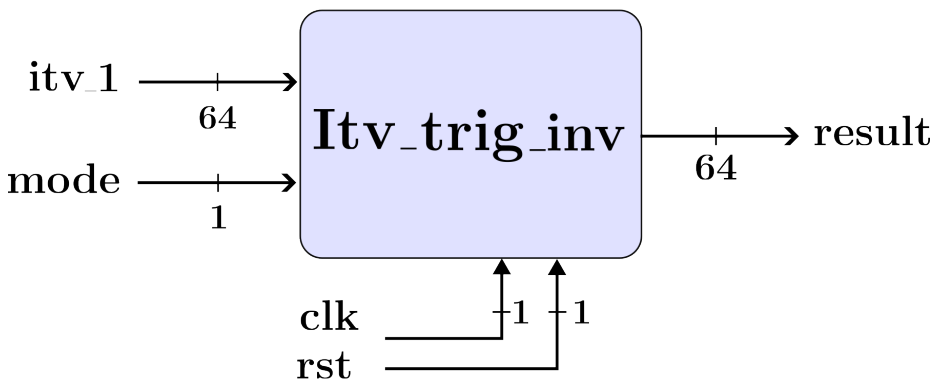


Figure 6.39: *Top-level Entity For Itv_trig_inv*

Hardware Entity (cont'd)

Input Ports

- **itv_1:** interval operand 1.
- **mode: 0** → *arccosine*,
  1 → *asine*.

Output Ports

- **result:** output interval.

6.2.10.4 Datapath for Itv_trig_inv

The circuit datapath (Figure 6.40) is composed of two main stages:

- Affectation of each input bound to one of the Arcsine/Arcosine blocks.
- computation of the angles associated to both bounds using CORDIC blocks (fixed-point arithmetic).

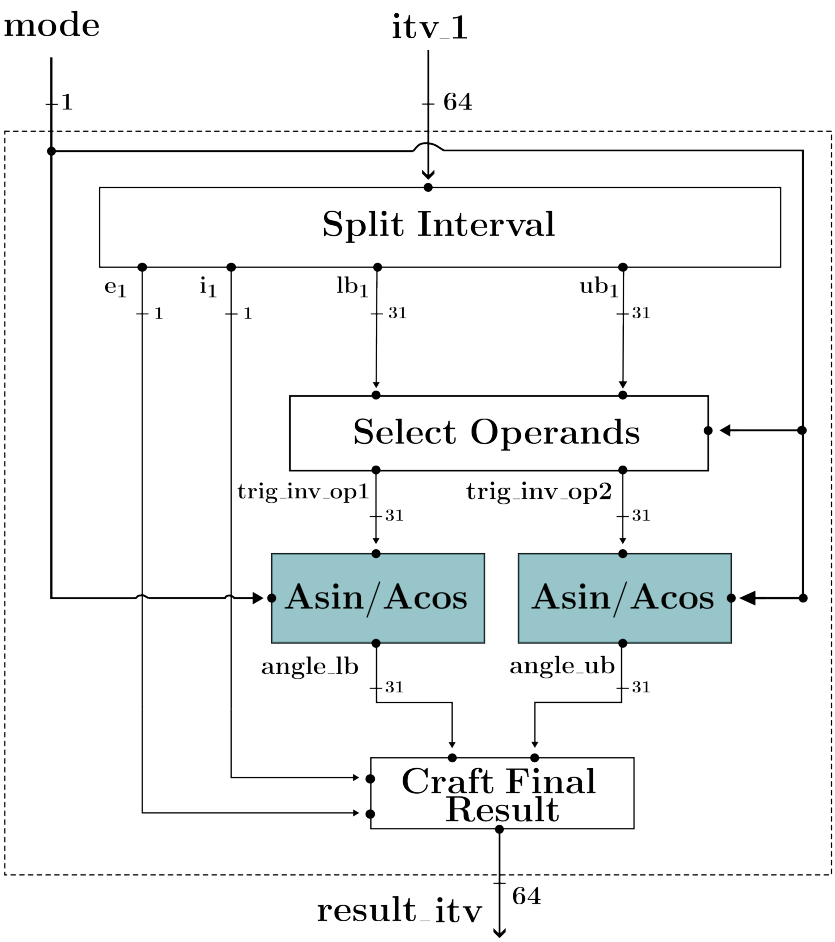


Figure 6.40: *Datapath of Itv_trig_inv*

Stage 1 - Affectation of bounds to the right Arcsine/Arccosine block:

Depending the selected mode (arcsine/arccosine), the order of input bounds may be permuted before being routed to the proper hardware Arcsine/Arccosine block. This is due to the Arcsine and Arcosine graphs being respectively strictly increasing and decreasing over $[-1, 1]$, as show by Figure 6.41.

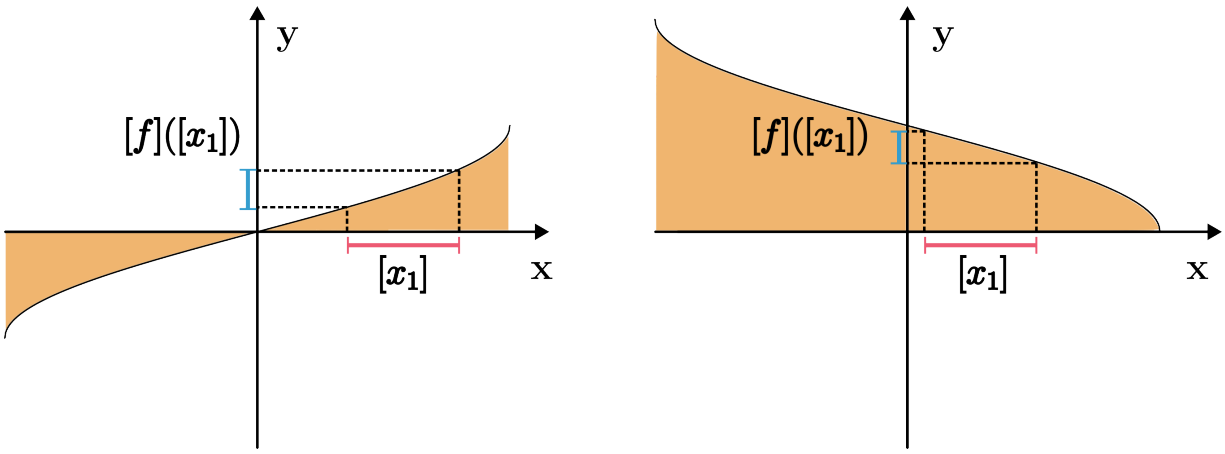


Figure 6.41: *Graph of Arcsine (left) and Arccosine (right) Functions over* $[-1, 1]$

Stage 2 - Angle evaluation

The angles corresponding to the input bounds are computed using CORDIC blocks set to evaluate Arctan2 [124] using the values of sine and cosine (bottom of Figure 6.41 and fully described in Figure 6.42). Depending of the selected mode, only the value for sine is available from the considered bound (resp. cosine) which requires to compute the other one using the the relations $c = \sqrt{1 - s^2}$ and $s = \sqrt{1 - c^2}$. This is done efficiently using FloPoco FMA and Square-root blocks. Similarly to the CORDIC block used for **itv_trig**, the input/output use fixed-point numbers in format Q1.23 which calls for the same conversions.

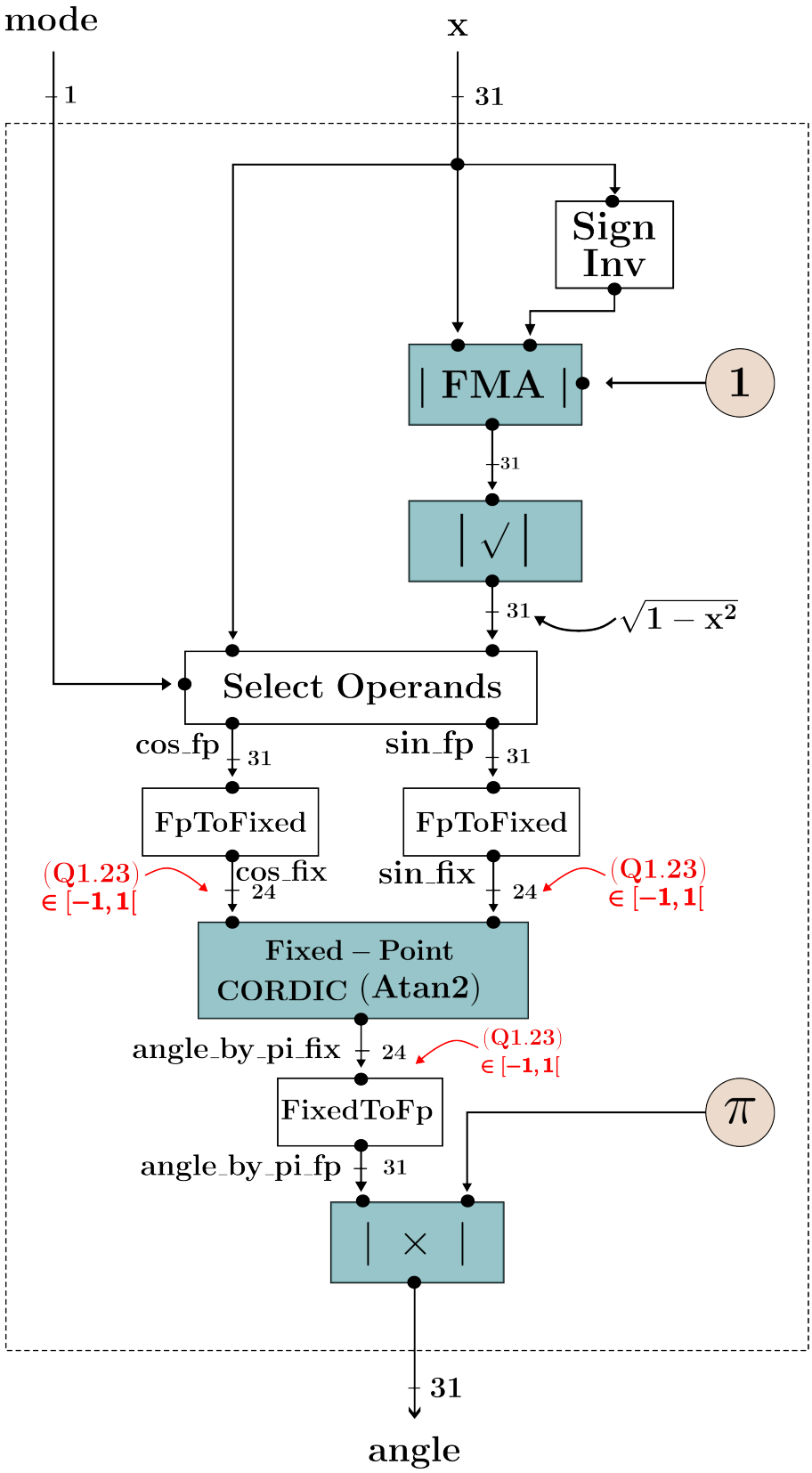


Figure 6.42: *Datapath of angle computation using asin/acos*

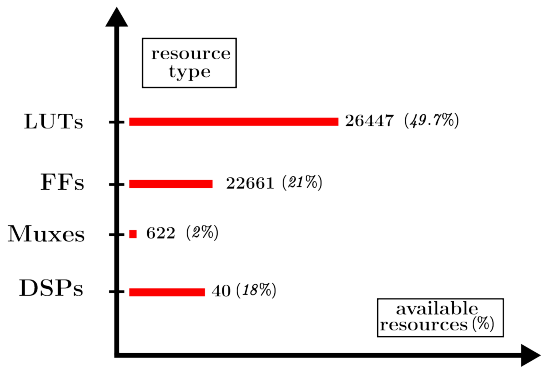
6.3 Synthesis Results

The aforementionned hardware operators have been synthesized on the Xilinx XC7Z020 FPGA in the context of the *xinterval core* described in Figure 6.4. They are all fully pipelined and can produce one result per clock cycle once the initial latency is over. The corresponding per-operator hardware utilization are given in Table 6.3 (percentages relative to the total resources of the FPGA) and must be analyzed while keeping in mind the utilization of the FloPoCo operators from Table 6.2 on the same hardware.

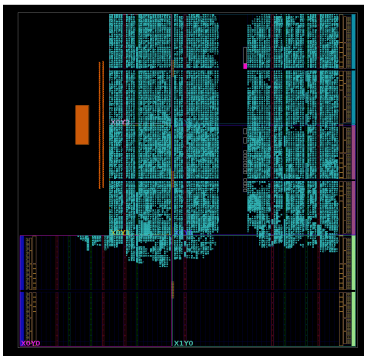
Hardware Entity	LUTs	FFs	Muxes	DSPs	Latency (cycles)	Fmax
itv_make	**214** (0.4%)	**307** (0.3%)	**0** (0%)	**0** (0%)	**5**	**100** MHz
itv_extract	**124** (0.2%)	**182** (0.2%)	**0** (0%)	**0** (0%)	**1**	**100** MHz
itv_set	**458** (0.8%)	**418** (0.4%)	**0** (0%)	**0** (0%)	**4**	**100** MHz
itv_add	**1219** (2.3%)	**1036** (1%)	**0** (0%)	**0** (0%)	**8**	**100** MHz
itv_mul	**951** (1.8%)	**956** (0.9%)	**0** (0%)	**8** (3.6%)	**7**	**100** MHz
itv_div	**3464** (6.5%)	**3161** (3%)	**52** (0.2%)	**0** (0%)	**16**	**100** MHz
itv_sqrt	**794** (1.5%)	**639** (0.6%)	**0** (0%)	**0** (0%)	**12**	**100** MHz
itv_exp	**653** (1.2%)	**894** (0.8%)	**10** (0.03%)	**2** (0.9%)	**13**	**100** MHz
itv_log	**1390** (2.6%)	**1660** (1.6%)	**48** (0.2%)	**6** (2.7%)	**12**	**100** MHz
itv_trig	**9769** (18.4%)	**6289** (5.9%)	**0** (0%)	**16** (7.2%)	**42**	**100** MHz
itv_trig_inv	**5683** (10.7%)	**4228** (4%)	**0** (0%)	**8** (3.6%)	**42**	**100** MHz

Table 6.3: *Synthesis Results For The Interval Operators On Xilinx XC7Z020 FPGA*

Regarding the *xinterval core*, the hardware utilization and an overview of the implemented design in Vivado are given respectively in Figure 6.43a and 6.43b.



(a) *Resources Utilization*



(b) *Implemented Design in Vivado*

Figure 6.43: *Synthesis of the Xinterval Core on Xilinx XC7Z020 FPGA @100MHz*

7

Design, Validation and Benchmarking methodology for the Xinterval Extension

The work described in this chapter has been published in the following paper:

> ***Contributions***
>
> ▶ *Filiol Pierre et al. "An Iterative Design and Validation Methodology for RISC-V Custom Extension" 2025, 32nd IEEE International Conference on Electronics, Circuits and Systems (ICECS). doi: 10.1109/ICECS66544.2025.11270698*

Contents

7.1 A Testbench Platform for the *Xinterval* Extension 170

7.1.1 Designing the Extension 170

7.1.2 Architecture of the Testbench Platform 172

7.2 In-depth Analysis of the *SoC* Simulator 173

7.2.1 Developping Applications for the SoC 174

7.2.2 The Hardware-In-The-Loop RISC-V ISA Simulation 177

7.2.3 Available *Xinterval* Wrappers for the RISC-V Core Simulator . . 182

7.3 Evaluating the Performances of the *Xinterval* Extension 189

7.3.1 Presentation of the Benchmarks 189

7.3.2 A Methodology for Fair Comparisons 190

7.3.3 Results Obtained 191

7.1 A Testbench Platform for the *Xinterval* Extension

7.1.1 Designing the Extension

7.1.1.1 The Development Process

The *xinterval* extension aims at providing the best-suited instructions to accelerate a set of reference benchmarks. The latter are thoroughly presented in Section 7.3 and intend to demonstrate the most recurrent interval primitives in robotics-oriented contractor applications such as localization.

With this goal in mind, the development process follows the iterative strategy depicted in Figure 7.1 and described below:

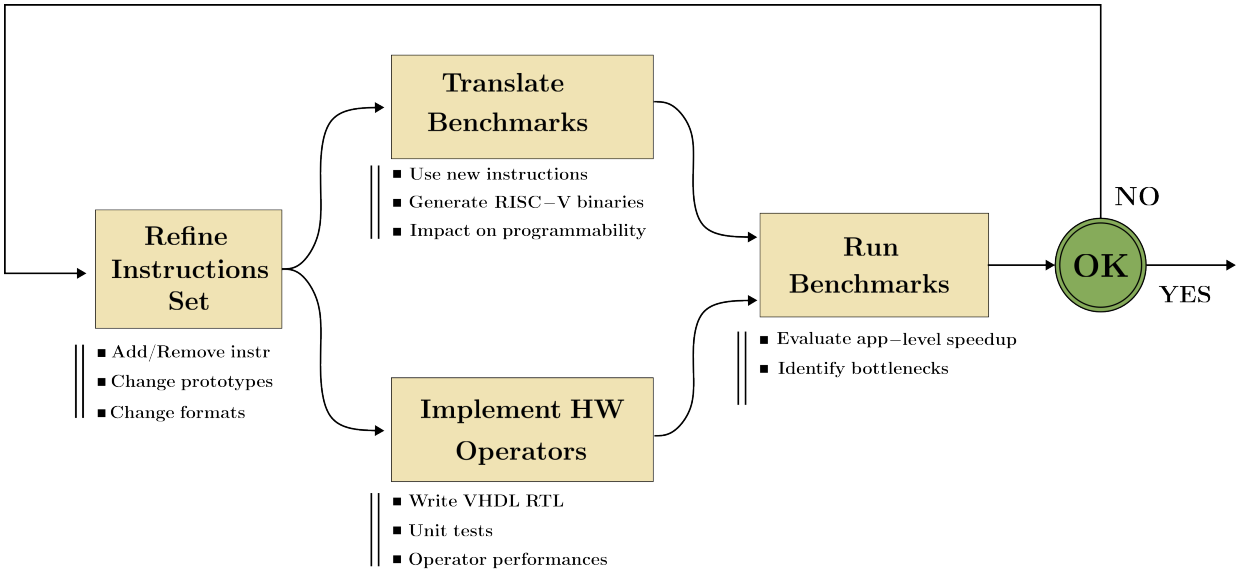


Figure 7.1: *The iterative development process used for the xinterval extension*

Refinement of the instruction set: The instruction set is gradually improved by reflecting on the limitations highlighted by the previous iterations. Several reasons may lead to this reassessment. For example, translating input benchmarks to actual C programs may be hard using the current version of the instruction set. One may also fail to reach the desired performances while implementing a specific hardware operator or realize that the current configuration achieves poor performances on a high-level application. The solution sometimes involves adding new instructions in order to circumvent a particular bottleneck, as it was the case for interval sine/cosine which were first evaluated in software before getting dedicated primitives. Another axis is to change the prototype or split a specific instruction. Section 5.3.4 shows that backward-contractors transitioned from using the R4-type format (to include a third source register for the final re-intersection) to the standard R-type with the introduction of standalone set instructions.

Translation of benchmarks: The C programs associated to the reference benchmarks are re-written to take advantage of the new instructions and are then compiled into native RISC-V binaries in preparation for the upcoming execution. An immediate benefit of this step is to assess the practicality of the instruction set (from the programmer's perspective) to describe real-world applications.

Hardware implementation: The backend of the instruction set is handled by a set of hardware operators such as the one presented in Chapter 6 for the final version of *xinterval*. The main goals are to implement efficient primitives using HDL code (VHDL in this case), ensure their validity through unit testing and evaluate their performances in terms of latency, max frequency or hardware utilization on FPGA.

Benchmark execution: The synthesized hardware is used to execute the benchmark applications in a profiled environment to assess how well the extension perform on real-world problems. Output products of this step are the verification of the correctness of the execution as well as a measure of speedup compared to a non-accelerated version of the benchmark.

7.1.1.2 Practical Challenges and Solutions

Bringing changes to the instruction set, a very central aspect of a processor, naturally causes several side-effects which greatly complicate the design process presented above.

The Toolchain:
The compiler must be gradually enriched with the new interval instructions as the development progresses to generate RISC-V binaries relying on the hardware acceleration brought by *xinterval*. The RISC-V foundation maintains an open-source GCC toolchain which can be extended with a little bit of effort and the knowledge of the GNU assembler (*gas*) syntax to declare instruction prototypes (refer to Section 7.2.1.2).

Running reference benchmarks on real hardware:
The proposed benchmarks are a set of fairly high-level applications which require a fully fledged RISC-V processor to run properly. In the context of a PhD work focused solely on the design of a new extension, it seemed unreasonnable to write a core from scratch to integrate the *xinterval* hardware. Several off-the-shelf RISC-V designs offer predefined ways of interfacing with custom designs. In this regard, Rocket [125] (UC Berkeley) and its ROCC [126] interface are popular choices in the literature [127, 128]. It does technically support accesses to floating-point registers (which is a pre-requisite of *xinterval*) through a recent update called *extended RoCC interface*. However, there are very few documentation an examples available on this topic so this solution was not further explored. On the other side, performing a manual modification of an existing core is somewhat doable but very error-prone without direct support from the authors of the core [129] especially when targeting the floating-point pipeline. The potential outcome

was deemed not worth the risk of introducing sneaky bugs which would only impede the development of the interval operators. The workaround adopted in this study is to rely on a software simulation of a RISC-V core for the standard instructions combined with hardware-in-the-loop capabilities to delegate *xinterval* instructions to real hardware. This approach is detailed in Sections 7.2.2 and 7.2.2.2.

7.1.2 Architecture of the Testbench Platform

The architecture of the testbench platform used to support the design of *xinterval* is presented in Figure 7.2.

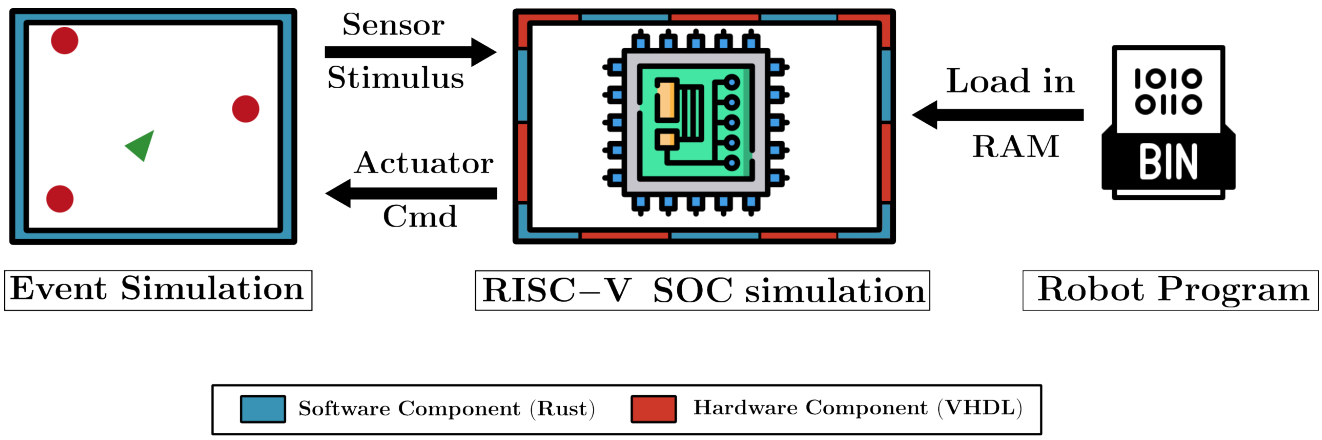


Figure 7.2: *Top-Level View of the Architecture of the Testbench Platform*

The platform allows to execute robotics-oriented applications at any point of the design process, regardless of the progress of the hardware implementation of *xinterval* instructions. The strategy is to progressively replace a software-based golden model for interval instructions by hardware of increasing representativity. An advantage of this approach is to smooth the complexity of RTL design by introducing several easy development milestones which all get validated against the benchmark applications.

The purpose of the system-on-chip (*SoC*) simulation is to perform a low-level modeling of a robot executing a contractor program. This component, whose internal organization is explicited in Figure 7.3, simulates a tiny embedded system composed of a RISC-V core, a main memory and several general-purpose actuators/sensors, all tied to the same interconnect bus. The simulator, while being mostly written using the Rust programming language, provides a plugin system to evaluate *xinterval* instructions using hardware operators written in VHDL rather than the default software model.

Localization and robust control algorithms are primary targets for *xinterval*. An example is the problem of the navigation in a field of landmarks using range-only measurements which has been presented in Section 1.3. In this context, the proposed algorithm estimates the position of the robot using contractor evaluations based on sensor mea-

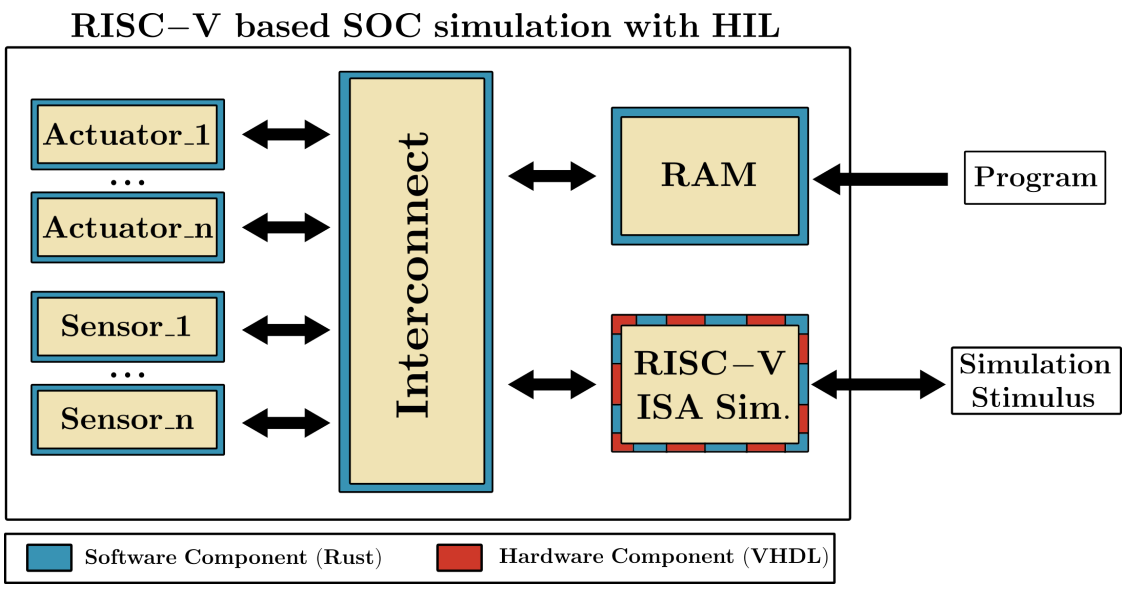


Figure 7.3: *A Detailed View of the RISC-V-based SoC Simulation*

surements and then update the parameters of its actuators to adjust the trajectory. This behavior can be replicated on the testbench platform through a close collaboration between the *SoC* and the event simulator. The *SoC* implements emulated memory-mapped peripherals accessible by the RISC-V core through the interconnect to either read the last measurements obtained (e.g. current distance to each landmark) or write a control update (e.g change the heading) to interact with the event-driven robot simulator. The latter is a really simple program which defines landmarks with known positions and sends periodic robot-to-landmark measurements to the *SoC*, wait for the end of the contractor evaluation and update the pose of the robot using the actuator commands returned by the *SoC*. Such simulation can be achieved by using a simple control models for the robot, such as the cart equations :

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = u_1 \\ \dot{v} = u_2 \end{cases} \tag{7.1}$$

where v is the speed of the cart, θ its orientation and (x, y) the coordinates of its center. The associated state vector is then given by $\mathbf{x} = (x, y, \theta, v)$ and u_1, u_2 are the actuator commands managed by the *SoC*.

7.2 In-depth Analysis of the *SoC* Simulator

This section provides an in-depth analysis of the most important elements of SoC simulation.

7.2.1 Developping Applications for the SoC

7.2.1.1 A Contractor Library With Two Interval Backends

The typical robotic applications targeted by *xinterval*, and particularly the proposed benchmarks, rely heavily on the intensive use of elementary interval contractors. As explained previously, the primitives provided by *xinterval* do not directly implement the usual elementary contractors but provide primitives which facilitate their implementation. As an example, the **itv.add** and **itv.set.i** instructions form the basis of the forward and backward contractors on addition, whose expressions were given in Figure 1.4.

As part of the thesis, a toy library offering an implementation of common contractors was developed using the C language, with a level of abstraction similar to *Ibex*, *Gaol*, or *libieeep1788*. Its purpose is first and foremost to facilitate the implementation of common robotic applications by hiding the low-level interval layer to the end-user, but also to help in the measurement of the real impact of *xinterval* on performances. To achieve this latter objective, it is required to accurately measure, for a given benchmark, the performance gap between a standard RISC-V and another which is allowed to use *xinterval* primitives.

Consequently, the library is available in two flavors respectively denoted as *libctc_noacc* and *libctc_acc*. The list of supported contractors is the same in both cases but the major difference lies in their backend implementation. In the first flavor (non-accelerated), the interval arithmetic and contractor layers are implemented using only standard RISC-V instructions whereas the second one (accelerated) makes use of all the additional instructions provided by *xinterval*. A summary outline in shown in Figure 7.4.

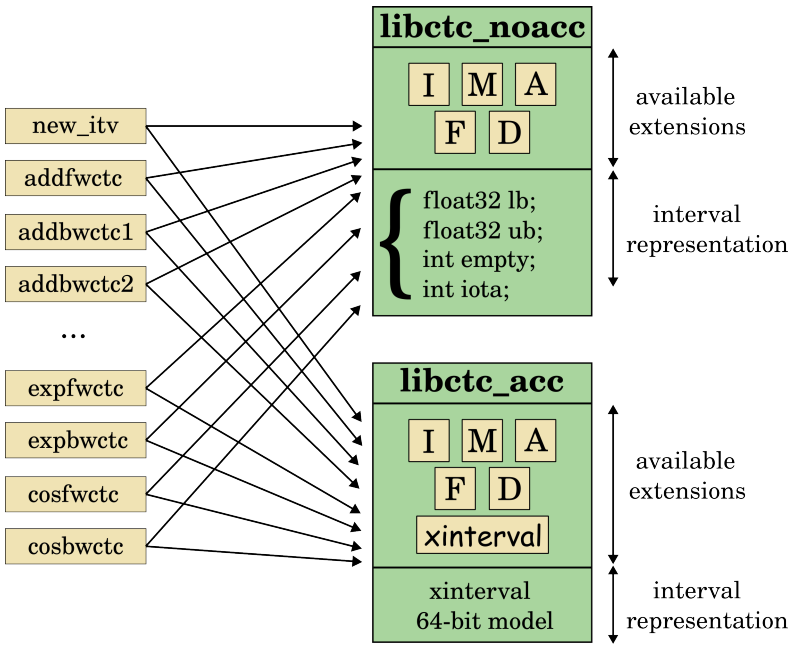


Figure 7.4: *Overview of the Two Flavors of the Implemented Contractor Library*

The datastructure used to represent an interval changes depending of the considered flavor. *Libctc_acc* relies on the *xinterval* 64-bit representation from Figure 5.3 which is type-casted to double in the context of a C language to force the compiler to use the dedicated floating-point registers to process intervals (Listing 8). *Libctc_no_acc* uses the C structure shown on Listing 7 which corresponds to the naive way of representing intervals. In this scenario, bounds are represented by Float32 values which allow computations to take advantage of the hardware floating-point pipeline of the RISC-V.

Listing 7: *Interval Datastructure Used in Libctc_no_acc*

```
1  /* Interval in libctc_noacc */
2  typedef struct {
3      /* Lower bound */
4      float lb;
5      /* Upper bound */
6      float ub;
7      /* Empty flag */
8      uint32_t empty;
9      /* Iota flag */
10     uint32_t iota;
11 } interval_t;
```

Listing 8: *Interval Datastructure Used in Libctc_acc*

```
1  /* Interval in libctc_acc */
2  typedef double interval_t;
```

Appendices B and C illustrates the implementation of the forward and backward contractors for addition respectively in the non-accelerated and accelerated flavor. Both versions starts by defining wrapper functions for the required interval arithmetic operators, in this case addition, subtraction and intersection. *Libctc_noacc* achieved this through Listing 14 using only standard instructions and a similar implementation to existing software interval libraries. On the contrary, *Libctc_acc* directly binds *xinterval* instructions using inline assembly, following Listing 16 and the methodology described in Section 7.2.1.2. Using this abstraction layer, the contractors can be easily implemented for both flavors as shown in Listings 15 and 17.

7.2.1.2 Extending the RISC-V toolchain With *Xinterval*

The RISC-V toolchain is built around the GCC compiler which generates machine code through the process illustrated in Figure 7.5. To compile the source code presented in Listing 16, the compiler must be made aware of the existence and specificities of the *xinterval* instruction. However modifying GCC directly is discouraged due to the complexity of the tool. A better solution, encouraged by the RISC-V standard, is to leave the proces-

sor untouched and intervene at the level of the *binutils* layer to patch the GNU assembler (gas) which is the part responsible for converting text assembly to object code.

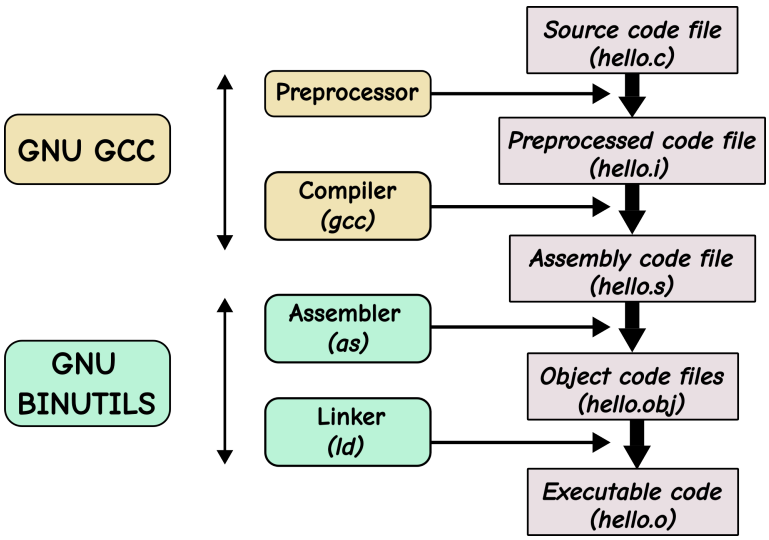


Figure 7.5: *Code Generation With GCC*

New instructions, for example *xinterval* ones, are declared directly in the source code of *binutils* by inserting new instances of the *riscv_opcode* datastructure in a predefined array, as show by Listing 9.

Listing 9: *Declaring Xinterval Instructions in the GNU Assembler*

```
const struct riscv_opcode riscv_opcodes[] = {

    /* Standard instructions */
    /*        (ommited)      */

    /* xinterval instructions */
    {"itv.mk", 0,INSN_XINTERVAL, "D,S,T", MATCH_ITV_MK, MASK_R, match_opcode, 0},
    {"itv.add", 0,INSN_XINTERVAL, "D,S,T", MATCH_ITV_ADD, MASK_R, match_opcode, 0},
    {"itv.sub", 0,INSN_XINTERVAL, "D,S,T", MATCH_ITV_SUB, MASK_R, match_opcode, 0},
    {"itv.mul", 0,INSN_XINTERVAL, "D,S,T", MATCH_ITV_MUL, MASK_R, match_opcode, 0},
    {"itv.sqr", 0,INSN_XINTERVAL, "D,S", MATCH_ITV_SQR, MASK_R, match_opcode, 0},
    {"itv.div", 0,INSN_XINTERVAL, "D,S,T", MATCH_ITV_DIV, MASK_R, match_opcode, 0},
    /*        (ommited)      */
    {"itv.exp", 0,INSN_XINTERVAL, "D,S", MATCH_ITV_EXP, MASK_R, match_opcode, 0},
    {"itv.log", 0,INSN_XINTERVAL, "D,S", MATCH_ITV_LOG, MASK_R, match_opcode, 0},
    {"itv.cos", 0,INSN_XINTERVAL, "D,S", MATCH_ITV_COS, MASK_R, match_opcode, 0},
    {"itv.sin", 0,INSN_XINTERVAL, "D,S", MATCH_ITV_SIN, MASK_R, match_opcode, 0},
    {"itv.acos", 0,INSN_XINTERVAL, "D,S", MATCH_ITV_ACOS, MASK_R, match_opcode, 0},
    {"itv.asin", 0,INSN_XINTERVAL, "D,S", MATCH_ITV_ASIN, MASK_R, match_opcode, 0},

    /* Terminate the list.  */
    {0, 0, INSN_CLASS_NONE, 0, 0, 0, 0, 0}
};
```

Some parameters of this datastructure are worth explaining, for example

- **Argument 1:** Assembly mnemonic (for source code).
- **Argument 3:** Associated RISC-V extension.
- **Argument 4:** A String representing the instruction prototype. For example, letters "d", "s", "t" respectively point to a destination, source_1 and source_2 register, living in integer registers when found in lowercase and in floating-point otherwise.
- **Argument 5:** A match integer which tells the assembler about the values expected in the encoding of the instruction such as opcode, funct3 and funct7. This works in combination with the sixth argument.
- **Argument 6:** A mask integer which tells the assembler about the position of the bits used to decode the instruction. As *xinterval* uses exclusively R-type instructions, this corresponds to the mask made of the opcode, funct3 and funct7 fields.

When the toolchain is recompiled, support for the new instructions is automatically added and robotics applications relying on *xinterval* primitives can be compiled seamlessly. This allows to produce native RISC-V binaries that are given as input to the SoC.

7.2.2 The Hardware-In-The-Loop RISC-V ISA Simulation

The RISC-V ISA simulator is responsible for executing the binaries of the benchmark applications and mimic the behavior of a real processor. It not only supports the standard extensions defined in the RISC-V specification, but also includes a model for all xinterval instructions in order to evaluate the impact of the proposed hardware accelerations. By default, the behavior of *xinterval* instructions is implemented as pure software in the same way as the standard instructions, but the advantage lies in the fact that the same evaluation can be performed using the *xinterval* hardware blocks as their development progresses thanks to a specific plugin system. The architecture of the component is divided into two main parts: the first implements all the basic features of a standard RISC-V as well as an execution profiling mechanism, while the second is used to interface with the *xinterval* hardware modules.

7.2.2.1 The Skeleton of the Simulator

A schematic view of the ISA simulator is shown in Figure 7.6.

According to the RISC-V nomenclature, the simulated processor is a *RV64IMAFD_xinterval* core, which indicates support for *xinterval* and the following standard extensions:

- **I:** Core instructions.
- **M:** Integer multiplication/division/modulos.
- **A:** Atomic operations.

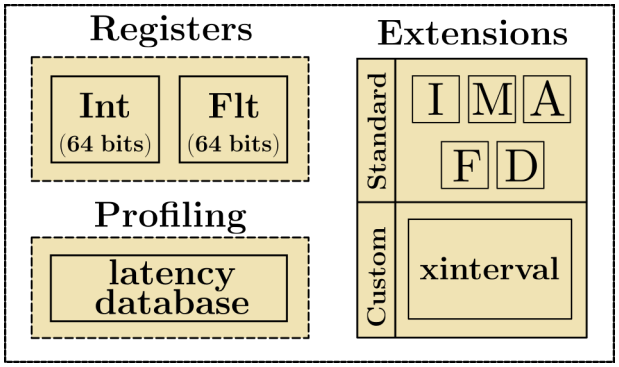


Figure 7.6: *A Schematic View of the ISA Simulator Deployed in the SoC*

- **F:** Single-precision floating-point numbers.
- **D:** Double-precision floating-point numbers.

As required by the RISC-V standard, the processor defines the following registers:

- **General-purpose registers:** 64 registers of width 64-bit.
- **Floating-point registers:** 64 registers of width 64-bit (shared by Float32 and Float64).

For simplicity, the ISA simulator uses a functional approach to binary emulation which treats each instruction as an atomic event. In this scenario, all the potential complications related to the classic RISC pipeline, such as data hazards, are ignored. Each instruction supported by the simulator, whether it comes from the standard or from *xinterval*, is implemented by a dedicated handler function written in Rust which modifies the internal state of the SoC (register, memory, IOs) in the appropriate way. As an example, Listing 10 shows how the decoded instructions are sent to the proper evaluation function and Listing 11 shows of the the **JAL** instruction is implemented.

Functional simulation, although simple and fast, makes it hard to evaluate the performance of the underlying hardware. A class of tools called *computer architecture simulator* precisely tackle this issue by introducing an actual modelling of a given computer architecture in addition to the simple execution of instructions. Features such as CPU pipelines, cache hierarchie and interconnects are taken into account to produce the most accurate prediction of the performances if the considered processor was built for real. The Gem5 simulator performs this task for the RISC-V architecture and allows to precisely configure the latency of each supported instruction to see the impact on the whole execution. At the beginning of the thesis work, The modification of Gem5 sources has been considered to interface it with real hardware, using a tactic similar to the one presented in in Section 7.2.2.2. The goal was to take advantage of its architectural simulation capabilities, which would have been really time-consuming to re-implement otherwhise, and add only the specificities of the *xinterval* model. However, Gem5's internal architecture made this evolution difficult, and the attempt was abandoned.

As an alternative, the proposed ISA simulator records the occurrence of each instruction

Listing 10: *Routing the decoded instruction to the right execution function*

```
1
2  pub fn execute_instruction_i(cpu : &mut RvCore, instr : &RvInstrContent, trace : bool) ->
   ↪  Result<RvCoreException,RvCoreError> {
3      match instr.mnemonic.as_str() {
4          "LUI"       => return lui(cpu, instr, trace),
5          "AUIPC"     => return auipc(cpu, instr, trace),
6          "JAL"       => return jal(cpu, instr, trace),
7          "JALR"      => return jalr(cpu, instr, trace),
8          "BEQ"       => return beq(cpu, instr, trace),
9          "BNE"       => return bne(cpu, instr, trace),
10         "BLT"       => return blt(cpu, instr, trace),
11         "BLTU"      => return bltu(cpu, instr, trace),
12         "BGE"       => return bge(cpu, instr, trace),
13         "BGEU"      => return bgeu(cpu, instr, trace),
14         "LB"        => return lb(cpu, instr, trace),
15         "LBU"       => return lbu(cpu, instr, trace),
16         ....
17         _           => return Err(RvCoreError::ExtensionNotSupportedError)
18     }
19 }
```

Listing 11: *The JAL instruction in the ISA simulator*

```
1  fn jal(cpu : &mut RvCore, instr : &RvInstrContent, trace : bool) ->
   ↪  Result<RvCoreException,RvCoreError> {
2      cpu.write_xregs(instr.rd as usize, cpu.pc().wrapping_add(4))?;
3      let sext_imm = match instr.sext_imm() {
4          Some(v) => v,
5          _ => return Err(RvCoreError::ExecutionInternalError)
6      };
7      let new_pc = cpu.pc().wrapping_add(sext_imm).wrapping_sub(4);
8      cpu.set_pc(new_pc);
9      if trace {
10         println!("jal {:#018x}", new_pc);
11     }
12     return Ok(RvCoreException::NoException);
13 }
```

during the execution of a program and estimates the number of clock cycles and total CPU time required. To this end, the latency of each instruction must be precisely known in order to perform an accurate summation. The latency database mentionned in Figure 7.6 is a way for the user to specify the latency of each instruction prior to the execution and match the performances of a realistic processor. In the absence of real hardware for standard RISC-V instructions, proposing a latency model involves a bias that is kept to a minimum thanks to the strategy presented in Table 7.1, which serves as the basis for the performance measurements in Section 7.3.

Extension	Method for latency selection
I	Analogous to Gem5's default parameters for RISC-V.
M	Analogous to Gem5's default parameters for RISC-V.
A	Analogous to Gem5's default parameters for RISC-V.
F	Latency of the associated FloPoCo operator at the target frequency.
D	Latency of the associated FloPoCo operator at the target frequency.
Xinterval	Latency achieved during synthesis, as announced in Table 6.2.

Table 7.1: *Proposition of a Latency Model for the ISA Simulator*

Remarks

The proposed strategy for performance estimation, based on the addition of the individual latencies of all the executed instructions, is a naive approach which does not take into account some common microarchitecture constructs, such as:

- pipelining
- branch prediction
- instruction reordering
- ...

However, *xinterval* is primarily targeted at embedded robotics applications where energy consumption is a major concern. As a result, the processors used in these areas tend to be simpler and make sparing use of the aforementionned mecanism. The considered approach, although imperfect, offers a beliveable yet pessimistic assessment of the viability of the proposed model, which would benefit from consolidation in future works, particularly by improving the realism of the in the ISA simulator.

7.2.2.2 Interfacing Real Hardware in the Simulator

The purpose of the ISA simulator is to support the development of *xinterval* instructions, by gradually transitioning from the default software model to real hardware. To achieve

this, the *xinterval core* from Section 6.1.2 can be connected to the main loop of the ISA simulator. The *xinterval core*, whose architecture is recalled in Figure 7.7, waits for 32-bit instructions coming from the simulator, perform decoding and execution locally before sending back the content of the destination register.

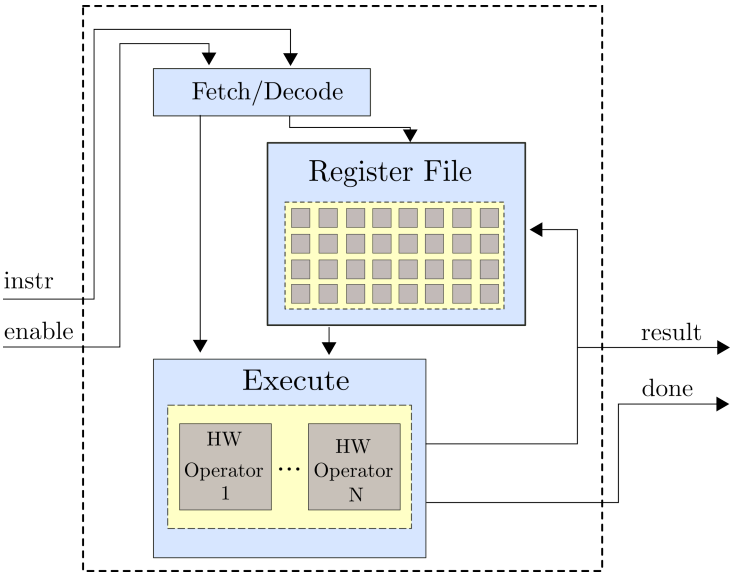


Figure 7.7: *Architecture of the xinterval core*

For the Rust executable and the VHDL model to run cooperatively and execute the application, the *xinterval core* is encapsulated in a wrapper which acts as a bridge between the hardware and the software. Throughout the development process, the method used to implement this wrapper evolves as hardware operators become more reliable to increase representativity. The list of all the available wrappers and their advantages are presented in Section 7.2.3. From the ISA simulator's perspective, the handler functions responsible for *xinterval* instructions revolve around an abstraction mecanism called «session» to communicate with the *xinterval core* with respect to the current wrapper used, as illustrated by Figure 7.8 and Listing 12.

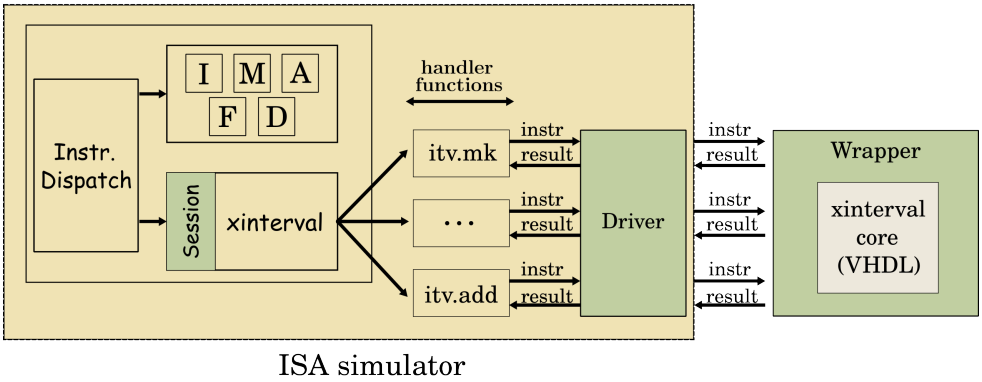


Figure 7.8: *Communication Between the ISA Simulator and the xinterval core*

Listing 12: *The itv.make Instruction in the ISA Simulator*

```
pub fn itvmake(cpu : &mut RvCore, instr : &RvInstrContent, trace : bool) ->
↪   Result<RvCoreException,RvCoreError> {
    /* Get bounds in rs1/rs2 of fpregisters */
    let rs1_content = cpu.read_fpregs_f32(instr.rs1 as usize)?;
    let rs1_content_u64 = rs1_content.to_bits() as u64;
    let rs2_content = cpu.read_fpregs_f32(instr.rs2 as usize)?;
    let rs2_content_u64 = rs2_content.to_bits() as u64;
    /* Check that we are using xinterval core instead of software */
    match cpu.mk1_session_mut() {
        Some(session) => {
            /* Copy bounds in xinterval core registers 1 and 2 */
            session.write_register(1, rs1_content_u64)?;
            session.write_register(2, rs2_content_u64)?;
            /* Run itv.make instruction on core and store result in register 3 */
            let result = session.make_interval_noiota(3, 1, 2)?;
            let result_f64 = f64::from_bits(result);
            /* Copy result back to the simulator */
            cpu.write_fpregs_f64(instr.rd as usize, result_f64)?;
            return Ok(RvCoreException::NoException);
        }
        None => return Err(RvCoreError::Mk1TransactionError)
    }
}
```

7.2.3 Available *Xinterval* Wrappers for the RISC-V Core Simulator

During the development, the *xinterval core* has been exposed to the ISA simulator through several wrappers, following the timeline shown in Figure 7.9.

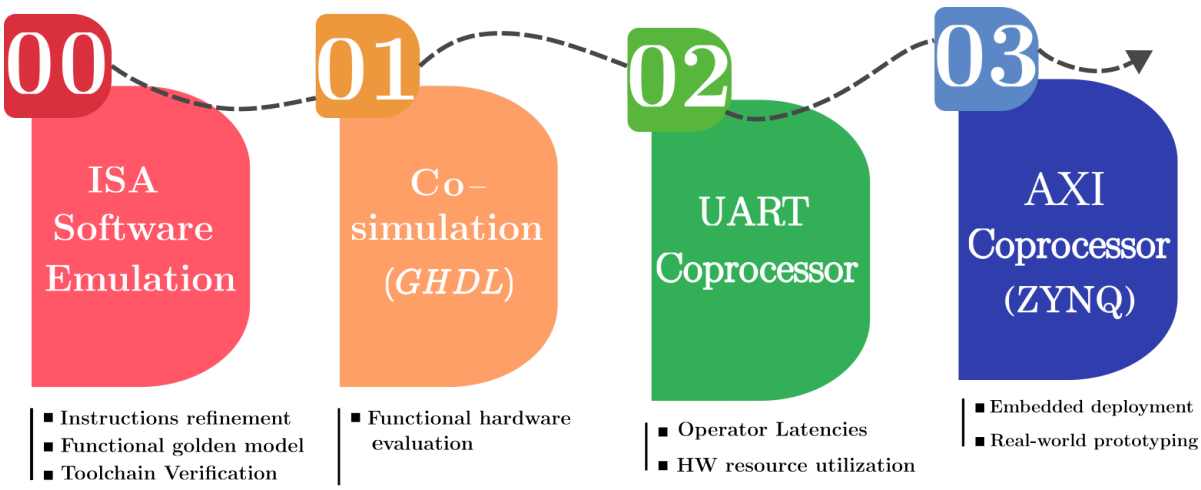


Figure 7.9: *Workflow Used in the Evolution of Wrapper Methodology for Xinterval Core*

Initially, *xinterval* instructions are implemented as pure software (milestone 0 from Figure 7.9), using similar handler functions to the one presented in Listing 11. When the

extension is modified, either by the addition/removal/modification of an instruction, the whole development process reverts to this stage to establish the functional golden model that will be used as a baseline for the subsequent milestones. Sections 7.2.3.1 to 7.2.3.3 presents the philosophy and design choices used to implement the wrappers from milestones 1 to 3.

7.2.3.1 Cosimulation with GHDL

In this setup, the *xinterval core* is evaluated by GHDL [130], an open-source VHDL simulator, and cooperates with the ISA simulator, following the method described in Figure 7.10. This solution heavily relies on the VHPIDIRECT [131] protocol, an higher-level alternative to the VPI [132] and VHPI [133] cosimulation standards, which allows the integration of foreign language functions (e.g written in C) from within the VHDL simulation. In the context of this wrapper, the ISA simulator instantiates a TCP client which operates as the dedicated transfer layer for the session used by the *xinterval* handler functions.

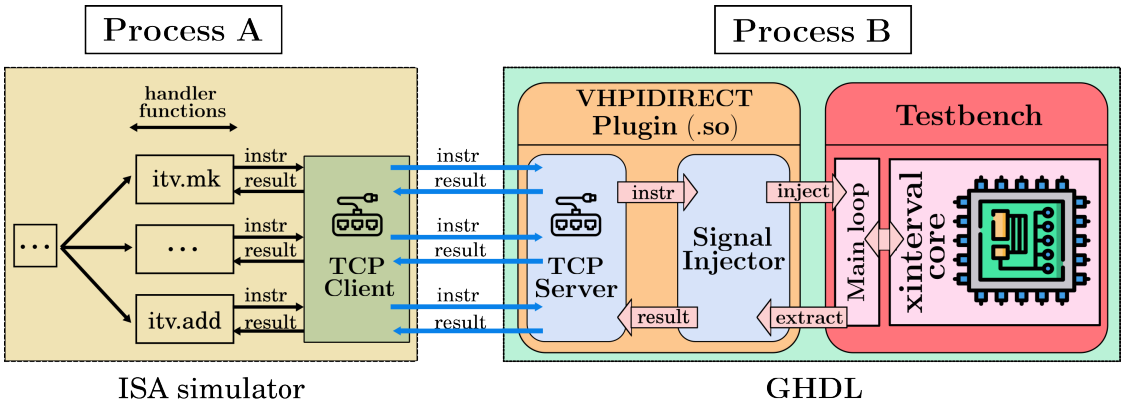


Figure 7.10: *Interfacing the ISA Simulator with the Xinterval Core through GHDL cosimulation*

The GHDL process is composed of two main parts:

The VHPIDIRECT plugin: A standalone shared object library, written in C and loaded alongside GHDL. Its purpose is to run a minimalist TCP server to which the ISA simulator connects and exchanges the instructions/results corresponding to the *xinterval* instructions of the current benchmark. The VHPIDIRECT protocol is used to transfer data at runtime between the C data-structures on the server side and the VHDL signals managed inside the testbench. As a an example, Listing 13 shows how the C function *recv_instruction*, which is responsible for receiving a 32-bit integer instruction from the server, is hooked from the shared object and exposed to the VHDL world.

The main testbench: The VHDL testbench instantiates the *xinterval core* and starts the infinite loop shown in Figure 7.11. The latter blocks until an instruction is received by the server, loads it as an input signal for the core, runs the simulation for a while until the result is ready and returns it to the C context.

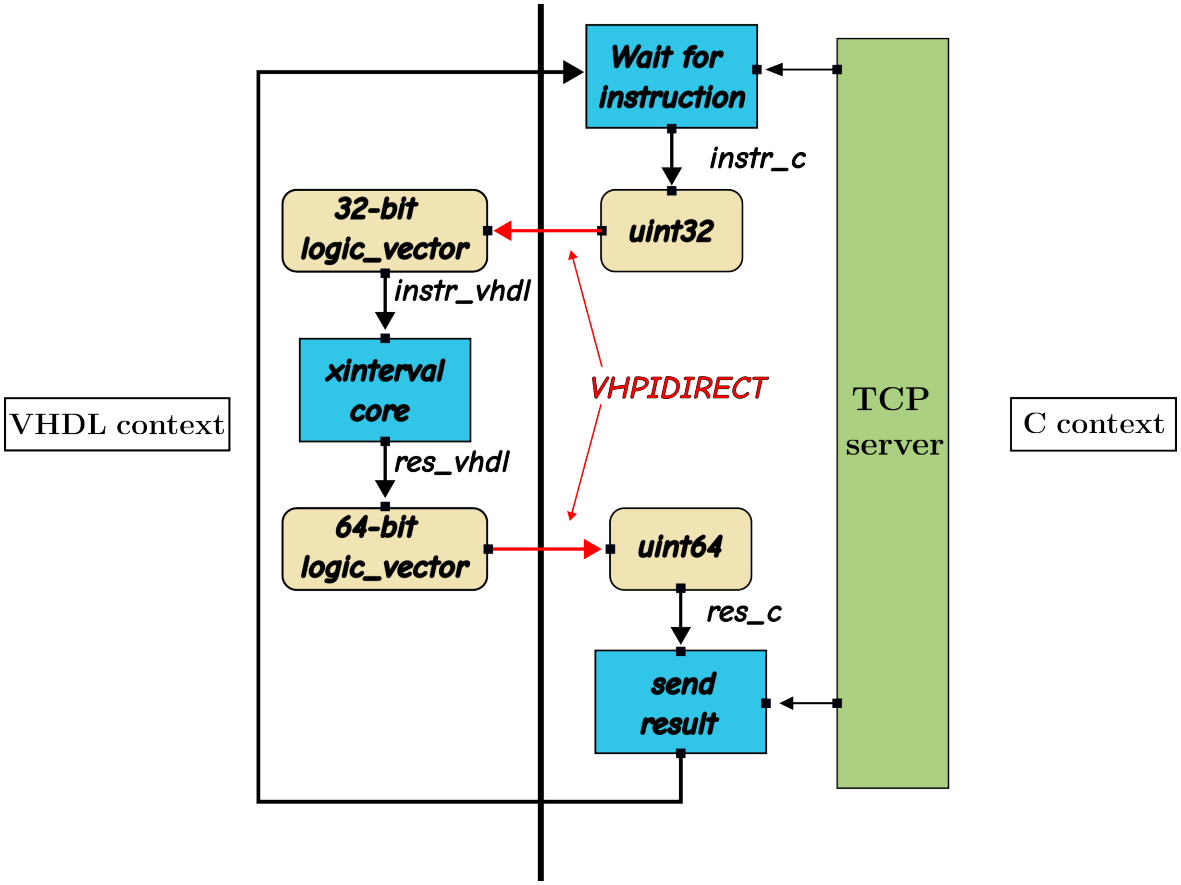


Figure 7.11: *Summary of the Exchanges Between the VHDL and C Contexts*

Listing 13: *Defining the Instruction Type in VHDL and Binding the C Function*

```
package cosim_pkg is

    -- instruction datatype
    type instruction is record
        b0 : integer;
        b1 : integer;
        b2 : integer;
        b3 : integer;
    end record instruction;

    -- pointer to instruction
    type instruction_ptr is access instruction;

    -- receive the next instruction (via libcosim.so)
    function recv_instruction return instruction_ptr;
    attribute foreign of recv_instruction:
        function is "VHPIDIRECT libcosim.so recv_instruction";

end package;
```

The GHDL simulator provides a simple way to prototype hardware operators without the

constraints imposed by the deployment on a real FPGA. At this stage, the main focus is made on functional verification which allows some compromises on the implementation. For example, the operators can be written in portable VHDL (without vendor-specific primitives) and the problems related to synthesis can be temporaly ignored (latency, max frequency, resource utilization). There are still several limitations to this method. The main drawback is that an evaluation performed in the context of a simulator typically fails to capture the complexity of real hardware and provides absolutely no guarantees regarding actual deployment on a chip. It is necessary to go up to the synthesis to accurately estimate the performances achieved by each operator, especially when attempting to evaluate the speedup of *xinterval*. Another drawback is the overall slowness of the simulation process, mostly due to the overhead imposed by the VHPIDIRECT plugin and the TCP communication. While it practically allows to execute real-world binaries with little efforts, the time required to do so can be measured in hours which is not acceptable as the tests grow in complexity. Figure 7.12 summarizes the pros and cons of the GHDL wrapper for the ISA simulator.

Pros	Cons
✓ Easy to setup	✗ Slow simulation
✓ Functional validation of hardware operators	✗ No synthesis
✓ No hardware required	✗ Minimal hardware representativity
	✗ Not suitable for performance evaluation

Figure 7.12: *Pros and Cons for the GHDL Wrapper*

7.2.3.2 UART Coprocessor

The *interval core* is now deployed on a FPGA board and interfaced with the ISA simulator (running on a Linux desktop) through the UART protocol, as shown in Figure 7.13. The main goal of this milestone is to produce interval operators that are meant to be synthesized, which is achieved through refinements of the hardware model used in milestone 1. A big part of the job is to develop a pipelined version of the former single-cycle operators to run at the requested frequency, while making the best use of the limited hardware resources of the target chip. This step introduces more hardware representativity and results in an explicit knowledge of the operator latencies, which is the prerequisite for the proposed performance evaluation strategy.

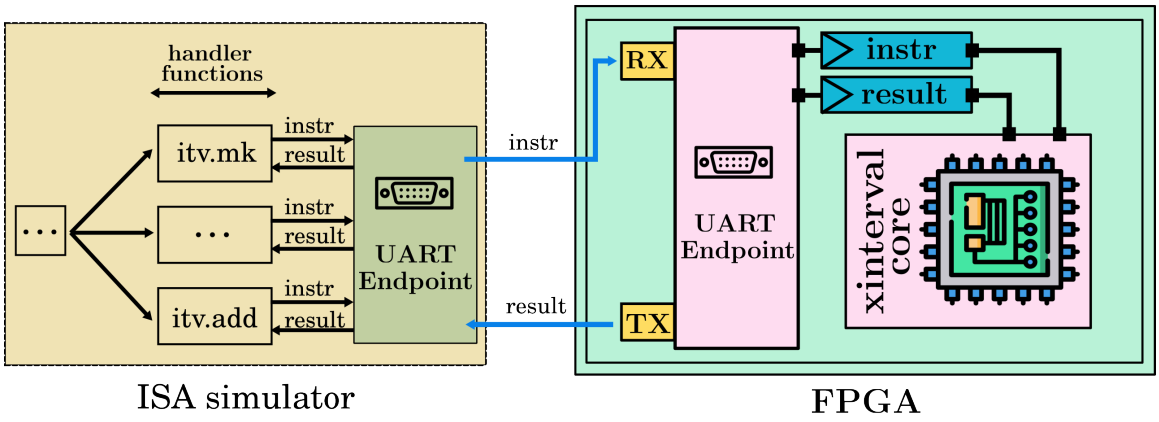


Figure 7.13: *Interfacing the ISA Simulator with the Xinterval Core through the UART protocol*

From the perspective of the ISA Simulator, the *xinterval* handler functions now relies on a dedicated UART session. At the beginning of the simulation, a request is made to the Linux operating system to configure and open a serial port to communicate with the FPGA board. The exchanges of instructions/results are then performed using this channel. From the FPGA's side, the instructions are received through the UART endpoint, stored in a dedicated register and fed to the *interval core*. The result obtained is then sent back to the ISA simulator in a similar way.

This method offers an easy way to execute fairly high-level applications using synthesized hardware, which would be hard to achieve otherwise in a context where no fully fledged RISC-V is available. However, there is still a large performance overhead due to the repeated exchanges of small instruction/result buffers over UART, leading once again to a rather slow simulation. Figure 7.14 summarizes the pros and cons of the UART wrapper for the ISA simulator.

Pros	Cons
✓ Easy to setup	✗ Slow simulation
✓ Synthesized hardware	✗ Not embedded
✓ Accurate latency model	

Figure 7.14: *Pros and Cons for the UART Wrapper*

7.2.3.3 AXI Coprocessor on Zynq

The last wrapper takes advantage of the Xilinx's Zynq [134] platform, an hybrid SoC composed of an ARM processor and a FPGA, which are respectively denoted as the *Process-*

ing System (PS) and the *Programmable Logic (PL)*. The architecture of this solution is displayed in Figure 7.15. Practically, Linux and the ISA simulator are executed on the PS part, the *xinterval core* is synthesized on the PL part and the communication happens through the high-speed AXI bus which ties both parts.

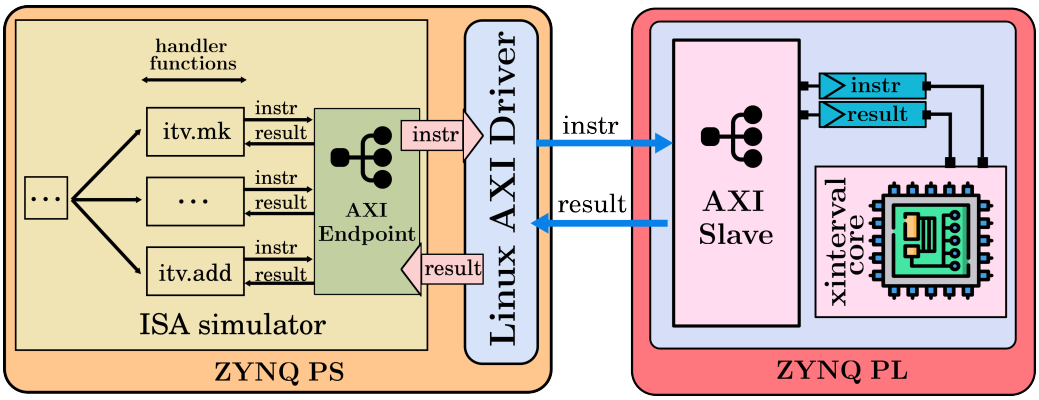


Figure 7.15: *Interfacing the ISA Simulator with the Xinterval Core through the AXI protocol*

The main idea is to directly expose the instruction/result registers of the *xinterval core* on the AXI bus and use the capabilities of the Linux operating system to manipulate their contents from the ISA simulator. This method typically leads to much lower latencies, sometimes by more than an order of magnitude, in the communications between the ISA simulator and the hardware. As a consequence, the speed of the simulation, which was the biggest drawback of the previous wrappers, is greatly improved and running the benchmarks presented in Section 7.3.1 is now a matter of minutes.

The most relevant technical details behind this wrapper are described below:

Configuration of the embedded Linux:

The Linux image used is heavily configured to be compatible with the PS part:

- Generation of a custom *first-stage bootloader (FSBL)* from Vivado.
- Use of the Uboot second-stage bootloader to 1) configure the PL part at startup with the *xinterval core* bitstream produced by Vivado and 2) load the kernel.
- The Xilinx's fork of the Linux kernel is modified to include a device-tree extension which essentially tells the kernel about the presence of the *xinterval core* on the AXI bus.
- A minimal userland is constructed to support the execution of Rust binaries.

The *xinterval* device driver:

- Generation of a custom *first-stage bootloader (FSBL)* from Vivado.
- Use of the Uboot second-stage bootloader to 1) configure the PL part at startup with the *xinterval core* bitstream produced by Vivado and 2) load the kernel.
- The Xilinx's fork of the Linux kernel is modified to include a device-tree extension which essentially tells the kernel about the presence of the *xinterval core* on the

AXI bus.

- A minimal userland is constructed to support the execution of Rust binaries.

A dedicated Linux kernel module has been developped to properly expose the *xinterval core* to applications running on the PS part. It instantiates a simple character device **/dev/xitv** which can be used to send an instruction and wait for the result by the mean of ioctl commands. Internally, the **io_remap** and **io{write,read}32** functions are used to interact with the *xinterval core* physical memory addresses on the AXI bus (which were given to the Linux kernel in the device-tree).

The ISA simulator AXI session:

The handler functions of the ISA simulator now rely on an AXI session which uses the ioctl commands provided by the kernel module to discuss with the hardware.

Exposing the *xinterval core* through an AXI slave:

In the PL part, the *xinterval core* is now packaged as an AXI-LITE slave and several registers are exposed on the AXI bus:

- A 32-bit write-only instruction register.
- Two 32-bit read-only registers called result_high and result_low.

These registers can be accessed by software applications using the aforementionned kernel module. An overview of the vivado block design is available in Figure 7.16.

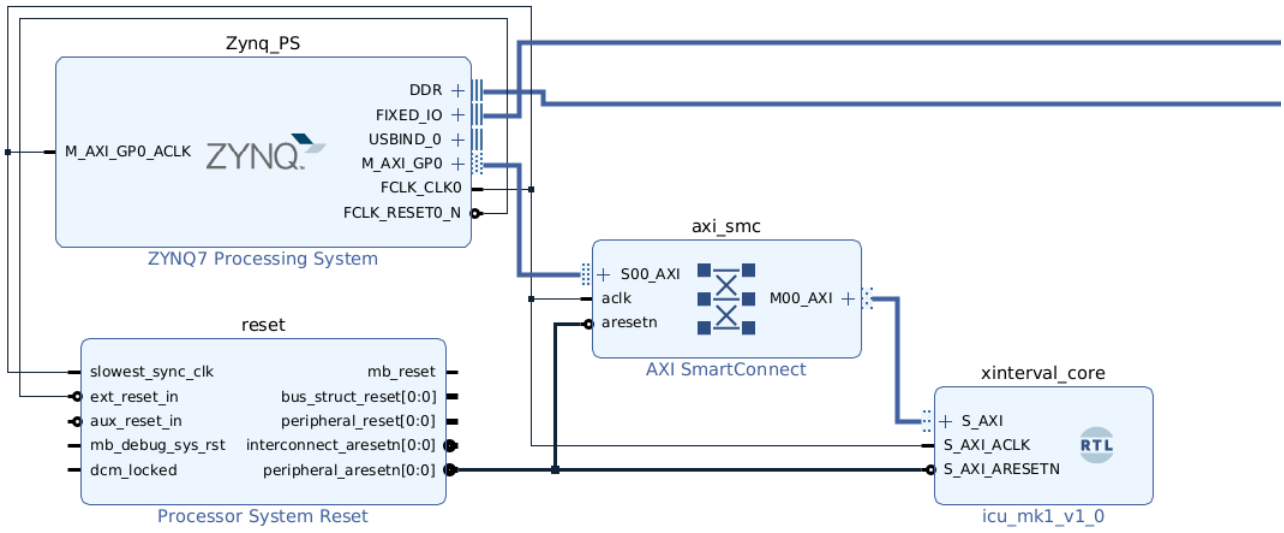


Figure 7.16: *Vivado Block Design for the Zynq-PL*

Finally, the pros and cons of the Zynq-based wrappers are summarized in Figure 7.17.

Pros	Cons
✓ Acceptable simulation speed	✗ Complex to setup
✓ Synthesized hardware	✗ Still not a real RISC-V
✓ Fully embedded	
✓ Accurate latency model	

Figure 7.17: *Pros and Cons for the Zynq Wrapper*

7.3 Evaluating the Performances of the *Xinterval* Extension

This section presents the benchmarks used to evaluate the performance of the *xinterval* extension. An explanation of the experimental methodology is conducted followed by a summary of the result achieved.

7.3.1 Presentation of the Benchmarks

Four case-study of increasing complexities have been chosen to debug and benchmark the performances of the *xinterval* instructions. They all aim at characterizing a solution set using interval contractors and the SIVIA paving algorithm while remaining somewhat representative of robotics-oriented applications.

7.3.1.1 Benchmark 1 - Ring

The first benchmark uses the same contractor as the range-only localization algorithm presented in Section 1.3. The expression of the associated CSP is given in Equation 7.2.

$$\mathcal{H} : \begin{cases} X : (x_1, x_2) \\ D : ([-10, 10], [-10, 10]) \\ C : \mathbf{f}(X) \in [16, 25] \end{cases} \quad \text{where } \mathbf{f}(X) = (x_1 - 1)^2 + (x_2 - 2)^2 \tag{7.2}$$

It uses only relatively simple elementary contractors such as addition, subtraction and square function and it thus a great starter.

7.3.1.2 Benchmark 2 - Waves

The second benchmark introduces more complex elementary contractors such as sine, exponential and division. The expression of the associated CSP is given in Equation 7.3.

$$\mathcal{H} : \begin{cases} X : (x_1, x_2) \\ D : ([-3.5, 2], [-3, 3]) \\ C : \mathbf{f}(X) \in [0.325, +\infty] \end{cases} \quad \text{where } \mathbf{f}(X) = \left\{ \frac{\sin({x_1}^2 + {x_2}^2)}{e^{x_1} + {x_2}^2} \right. \tag{7.3}$$

7.3.1.3 Benchmark 3 - Donut

The third benchmark uses the same elementary contractors are the second benchmark in a slightly different configuration. The expression of the associated CSP is given in Equation 7.4.

$$\mathcal{H} : \begin{cases} X : (x_1, x_2) \\ D : ([-3, 3], [-3, 3]) \\ C : \mathbf{f}(X) \in [-0.2, 0.2] \end{cases} \quad \text{where } \mathbf{f}(X) = e^{x_1 \times x_2} - \sin(x_1 - x_2) \tag{7.4}$$

7.3.1.4 Benchmark 4 - Vortex

The last benchmark features a more complex contractor which introduces partially defined operators such as square-root. The expression of the associated CSP is given in Equation 7.5.

$$\mathcal{H} : \begin{cases} X : (x_1, x_2) \\ D : ([-3.5, 2], [-3, 3]) \\ C : \mathbf{f}(X) \in [0.325, +\infty] \end{cases} \quad \begin{aligned} \text{where } \mathbf{f}(X) = (x_2 - 5)\cos(4\sqrt{(x_1 - 4)^2}) + {x_2}^2 \\ - x_1 \sin(2\sqrt{{x_1}^2 + {x_2}^2}) \end{aligned} \tag{7.5}$$

7.3.2 A Methodology for Fair Comparisons

Most interval-related developments, especially in robotics, are targeted for x86 or ARM architectures where the rounding of floating-point greatly impact performances, as explained in Section 1.4.3. To compensate for this, most of the popular interval software

libraries rely on implementation tricks that have no equivalent in the RISC-V architecture. The later has the advantage of performing floating-point rounding context change at zero cost but suffers from the lack of ASIC-based chips which are serious competitors for x86/ARM alternatives. In this context, it seems hard to come with a setup where processors are equally performant and rely on equally optimized libraries, even when putting *xinterval* aside.

This study hypothesizes that the gradual adoption of consumer RISC-V processors will eventually allow them to compete with x86 chips in terms of raw computing power and that their approach to floating-point rounding will give them a structural advantage for interval arithmetic. As a consequence, the choice has been made to compare the difference of performance obtained between a standard RISC-V and a RISC-V augmented with *xinterval*.

To achieve this, the benchmarks presented in Sections 7.3.1.1 to 7.3.1.4 are compiled against the two flavors of the interval toy-library described in Section 7.2.1.1. For the benchmark X, it leads to two binaries *benchmark_x_no_acc* and *benchmark_x_acc* representing respectively the standalone RISC-V and the accelerated version. For the reasons detailed in 7.1.1.2, the evaluation is based on the Hardware-in-the-loop approach presented throughout the Chapter and uses the instruction occurence-counting strategy from Section 7.2.2.1 despite its obvious limitations. The later are compensated by the fact that the same method is used for both platforms and that the speedup ratio should remain the same.

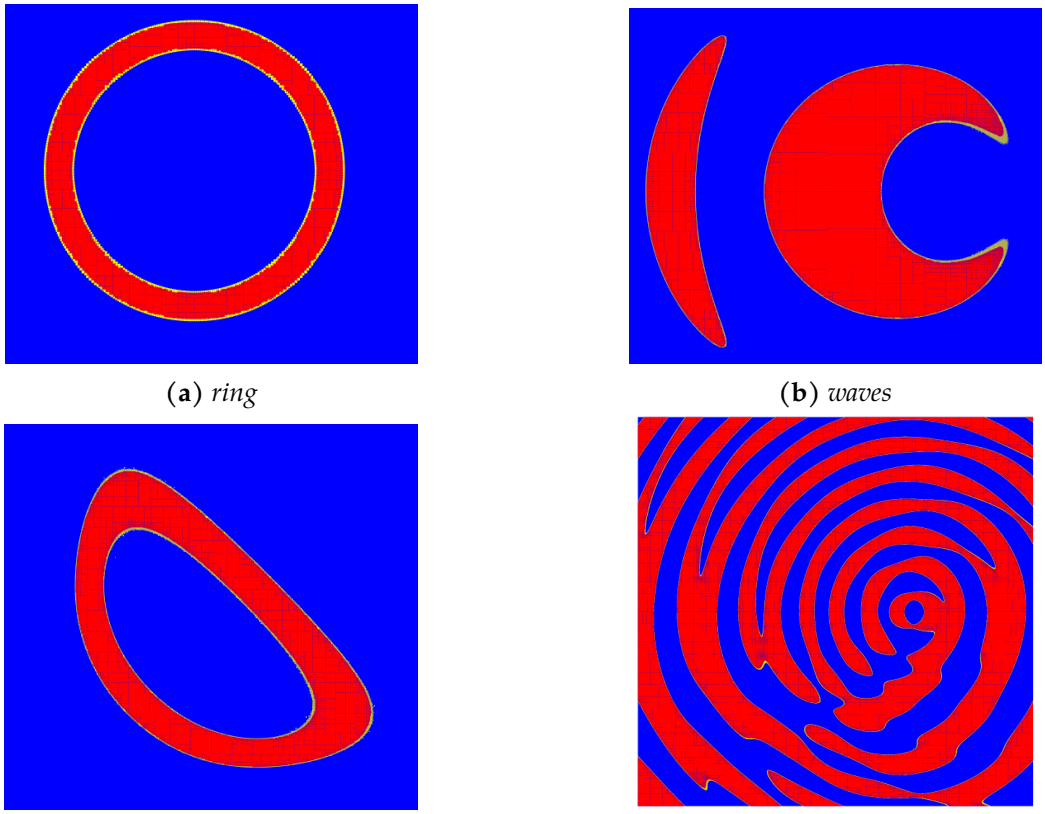
7.3.3 Results Obtained

The pavings shown in Figure 7.18 have been obtained by running the *xinterval* accelerated binaries of the proposed benchmark through the Zynq Wrapper of the ISA simulator. The speedup obtained as well as the simulation time are displayed in Table 7.2.

Benchmark	Speedup	Simulation Time (Zynq)
ring	×3.2	22s
waves	×7.3	156s
donut	×8.2	185s
vortex	×7.8	372s

Table 7.2: *Speedup Estimate Relative to a RISC-V Processor Without Xinterval*

First of all, each benchmarks leads to a speedup above 2 times, which is reassuring considering that the bounds of the intervals are now evaluated in parallel thanks to the dedicated hardware. The first benchmark seems to perform comparatively worse (only 3.2×) than the others, which may be explained by the fact that it mostly involves «easy» elementary contractors such as addition, subtraction or square for which no clever implementation trick is doable beside from calling the associated operator concurrently on



(**a**) *ring* (**b**) *waves*

(**c**) *donut* (**d**) *vortex*

Figure 7.18: *Pavings Obtained for the Benchmarks When Using the Xinterval Flavor of the Interval Toy-Library and the Zynq Wrapper for the ISA Simulator.*

the lower and upper bounds. The gain is essentially made on the other interval operations, such as intersection or the backward-contractor on the square constraint which would require several instructions if *xinterval* was not available. The other benchmarks feature similar levels of gain (between 7.3 and 8.2 $\times$) which is explained by the use of more complex contraints such as multiplication, division or trigonometry whose software equivalents involve at lot of expensive and non-parllelized operations.

These results seems underwhelming compared to what can be typically achieved by hardware acceleration in other areas such as compression. Howver, the proposed operators and their implementation are provided as proof of concept and are far from optimal, since it is indeed possible to increase their performances in several ways. Areas for improvement include, for example, optimizing floating-point primitives, reducing the latency of specific operators or making changes in some algorithms used to evaluate the intervals. In addition, one of the major constraints of the study was to propose a solution that would not disrupt the way of programming with intervals and would maximize the reuse of existing code. In this context, a gain of 10 $\times$ remains very interesting.

Conclusion

Contents

Synthesis of the Contributions **193**
Achievements 193
Limitations 194
Future Works **195**

Synthesis of the Contributions

This thesis was initially motivated by the existence of performance bottlenecks in interval applications used in embedded robotics, such as location or robust control. The proposed work examined how a dedicated hardware acceleration could benefit interval arithmetic, whose current software implementations face issues related to general-purpose architectures. In robotics, interval methods rely mainly on the evaluation of elementary interval contractors, whose optimization is therefore a major focus.

In addition to merely improving the performances of interval applications, the study seeks to propose an acceleration model that allows maximum reuse of existing interval code by easily substituing to popular interval software libraries.

Achievements

The work helped to consolidate the theoretical basis of interval constraint programming, by proposing an extension to interval arithmetic which allows to correctly generate the complementary of a contractor when partially defined functions are involved. This problem was encountered in most of the popular interval libraries and the solution found, i.e. the introduction of the *iota* element, is a promising idea for a potential future revision of the IEEE-1788 standard. This new feature has been successfully implemented in the hardware model described below.

A major achievement is the specification and the implementation of hardware primitives dedicated to the acceleration of elementary interval contractors. The resulting operators have been synthesized on a physical FPGA to profile resources utilization, latency and maximum operating frequency. The potential of customization brought by the FPGA platform has been used to circumvent the limitations of traditional processors and build a microarchitecture suited for intervals. The main optimizations tackle the floating-point layer of interval computations. For a given interval operator, both bounds are now evaluated in a purely concurrent manner and with the appropriate rounding mode, which improves performance significatively compared to a software solution running on x86. Another interesting outcome of tailor-made hardware is that interval bounds can be made to use custom-float formats to closely match the needs of the considered application (e.g. robotics), particularly in term of precision bits.

A discussion related to the integration of interval arithmetic as a RISC-V extension has been conducted. The main difficulty was find the optimal tradeoff between the features that should be handled in hardware and those who belong to software, in accordance with the requirements of embedded robotics, while adapting to the architectural contrainsts of the RISC-V. The resulting instruction set, named *xinterval*, provides a solid basis to implement an interval contractor library, which can then be used as a seamless replacement to existing software libraries.

Finally, the behavior of the *xinterval* extension has been verified using a hardware-in-the-loop approach which interfaces a software RISC-V model with an interval core made of the aforementionned operators and synthesized on a FPGA. This setup allowed to successfully execute a set of reference robotics applications which proves the viability of the ISA extension strategy for interval arithmetic. In addition, the same platform has been used to estimate the performance gain brought by the proposed model compared to a similar implementation running of a non interval-accelerated RISC-V. The experimentation led to speedups ranging from three to ten depending of the considered problem, with room for growth.

Limitations

The hardware operators developed, although functional, still have significant room for improvement. The FloPoCo library has fulfilled its role of providing configurable and synthesizable floating-point operator primitives, without which the work carried out would not have been possible. However, it is still an automatic code generator, and a manual approach optimized for a specific FPGA will yield better results, at the cost of a long and time-consuming development effort that was out of the scope of this study. Some of the choices made in the hardware algorithms can be discussed and improved. For example, trigonometric operators rely on CORDIC primitives which are not known to be fastest method on FPGA. Overall, although the proposed operators are fully pipelined,

little effort has been put into aggressive optimization, which means that the performance announced for *xinterval* can only be improved.

One of the major limitations of the work carried out is that development was not pursued to the point of final integration into a RISC-V hardware model. The main reason for this omission is strategic, as this integration process had the potential to lead to many technical difficulties that would have cut into the time allocated to other elements of the already broad spectrum of work. The hardware-in-the-loop solution used as a replacement proved the viability of the ISA extension for interval arithmetic, but was quite limited in its ability to accurately evaluate performance gains, which can only be done through integration in a fully-fledged processor.

Future Works

This research opens up prospects for future work. As a direct continuation of the thesis, integrating the *xinterval* extension into a synthesized RISC-V processor is a priority, as it will enable more accurate measurement of performance improvements. The Rocket RISC-V, as a reference research implementation, is an ideal candidate for hosting the proposed extension, particularly via its ROCC interface, which simplifies co-processor deployment.

Another option is to port a well-established interval software library in the field of robotics, such as *codac*, so that it uses *xinterval* as the main back-end instead of the general-purpose hardware. This would increase the accessibility of the extension for other research works, enable testing in real-world conditions, and ultimately improve the model.

On a more exploratory note, it may be interesting to explore other acceleration models for interval arithmetic. Still in the context of the traditional processor model, it is possible to study the feasibility of an ISA extension dedicated to SIMD operations with intervals, following the design principles of the V extension of the RISC-V standard. This approach makes sense in robotics for evaluating several branches of the SIVIA algorithm in parallel.

Finally, the «hardware-only» approach to interval arithmetic could also be promising. This strategy completely diverges from the work carried out and sacrifices the flexibility offered by the processor model in order to increase performance. The thesis developped hardware operators for elementary interval contractors which are the low-level building blocks for more complex contractors following the forward-backward methodology. They could be reused in a workflow based on the HC4R algorithm to lay down the entire contractor (forward and backward parts) as a unique combinatorial circuit.

Bibliography

[1] Barton, Paul Harwood, Stuart. "Efficient polyhedral enclosures for the reachable set of nonlinear control systems". In: *Mathematics of Control, Signals, and Systems* 28 (Feb. 2016). ISSN: 1435-568X. DOI: https://doi.org/10.1007/s00498-015-0153-2. URL: https://ieeexplore.ieee.org/document/1057074.

[2] A. Sabharwal et al. "Set estimation via ellipsoidal approximations". In: 2 (1995), 897–900 vol.2. DOI: 10.1109/ICASSP.1995.480319.

[3] Luc Jaulin et al. "Set inversion via interval analysis for nonlinear bounded-error estimation". In: *Automatica* 29.4 (1993), pp. 1053–1064. ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(93)90106-4. URL: https://www.sciencedirect.com/science/article/pii/0005109893901064.

[4] Ramon E. Moore et al. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009. DOI: 10.1137/1.9780898717716. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717716. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898717716.

[5] University of North Carolina. *Patriot Missile Software Problem*. https://www.cs.unc.edu/ smp/COMP205/LECTURES/ERROR/lec23/node4.html. 2006.

[6] Douglas N.Arnold. *The Explosion of the Ariane 5*. https://www-users.cse.umn.edu/ arnold/disasters/ariane.html. 2000.

[7] Teruo Sunaga. "Theory of an Interval Algebra and Its Application to Numerical Analysis". In: *Research Association of Applied Geomet* 2 (1958), pp. 29–46.

[8] E. R. Hansen. "Global optimization using interval analysis: The one-dimensional case". In: *Journal of Optimization Theory and Applications* 29.3 (Nov. 1979), pp. 331–344. ISSN: 1573-2878. DOI: 10.1007/BF00933139. URL: https://doi.org/10.1007/BF00933139.

[9] Jürgen Herzberger Götz Alefeld. *Introduction to Interval Computation*. Elsevier, 1983. ISBN: 978-0-12-049820-8. DOI: https://doi.org/10.1016/C2009-0-21898-8. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717716. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898717716.

[10] Ulrich W. Kulisch. *Complete Interval Arithmetic and Its Implementation on the Computer*. Ed. by Annie Cuyt et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 7–26. ISBN: 978-3-642-01591-5.

[11] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1991. ISBN: 9780521331968.

[12] R. E. Moore. "A survey of interval methods for differential equations". In: *The 23rd IEEE Conference on Decision and Control*. 1984, pp. 1529–1535. DOI: 10.1109/CDC.1984.272338.

[13] R. Baker Kearfott. "Interval analysis: unconstrained and constrained optimizationInterval Analysis: Unconstrained and Constrained Optimization". In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas et al. Boston, MA: Springer US, 2009, pp. 1727–1730. ISBN: 978-0-387-74759-0. DOI: 10.1007/978-0-387-74759-0_302. URL: https://doi.org/10.1007/978-0-387-74759-0_302.

[14] Nicolas Delanoue et al. "Using interval arithmetic to prove that a set is path-connected". In: *Theoretical Computer Science* 351.1 (2006). Real Numbers and Computers, pp. 119–128. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2005.09.055. URL: https://www.sciencedirect.com/science/article/pii/S0304397505006122.

[15] Bastien Vincke et al. "Guaranteed simultaneous localization and mapping algorithm using interval analysis". In: *2014 13th International Conference on Control Automation Robotics Vision (ICARCV)*. 2014, pp. 1409–1414. DOI: 10.1109/ICARCV.2014.7064522.

[16] Abhisha Chauhan et al. "Optimal path planning for a non-holonomic robot using interval analysis". In: *2018 Indian Control Conference (ICC)*. 2018, pp. 184–189. DOI: 10.1109/INDIANCC.2018.8307975.

[17] Josep Vehí et al. "A SURVEY OF APPLICATIONS OF INTERVAL ANALYSIS TO ROBUST CONTROL". In: *IFAC Proceedings Volumes* 35.1 (2002). 15th IFAC World Congress, pp. 389–400. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20020721-6-ES-1901.00399. URL: https://www.sciencedirect.com/science/article/pii/S1474667015388200.

[18] Michael Wagner et al. " Interval-Arithmetic-Based Trajectory Scaling and Collision Detection for Robots with Uncertain Dynamics ". In: *2018 Second IEEE International Conference on Robotic Computing (IRC)*. Los Alamitos, CA, USA: IEEE Computer Society, Feb. 2018, pp. 41–48. DOI: 10.1109/IRC.2018.00015. URL: https://doi.ieeecomputersociety.org/10.1109/IRC.2018.00015.

[19] Eero Hyvönen. "Constraint reasoning based on interval arithmetic: the tolerance propagation approach". In: *Artificial Intelligence* 58.1 (1992), pp. 71–112. ISSN:

0004-3702. DOI: https://doi.org/10.1016/0004-3702(92)90005-I. URL: https://www.sciencedirect.com/science/article/pii/000437029290005I.

[20] Frédéric Benhamou et al. "CLP(Intervals) Revisited." In: Jan. 1994, pp. 124–138.

[21] Djamila Sam-Haroud et al. *Constraint Consistency Techniques for Continuous Domains*. 1995. URL: https://infoscience.epfl.ch/handle/20.500.14299/238126.

[22] Luc Jaulin et al. *Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics*. http://www.springer.com/engineering/computational+ 1-4471-1067-5. Springer London Ltd, Aug. 2001, p. 398. URL: https://hal.science/hal-00845131.

[23] Gilles Chabert et al. "Contractor programming". In: *Artificial Intelligence* 173.11 (2009), pp. 1079–1100. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2009.03.002. URL: https://www.sciencedirect.com/science/article/pii/S0004370209000381.

[24] Frédéric Benhamou et al. "Revising Hull and Box Consistency." In: Jan. 1999, pp. 230–244.

[25] Luc Jaulin et al. "Nonlinear State Estimation Using Forward-Backward Propagation of Intervals in an Algorithm". In: *Scientific Computing, Validated Numerics, Interval Methods*. Ed. by Walter Krämer et al. Boston, MA: Springer US, 2001, pp. 191–201. ISBN: 978-1-4757-6484-0. DOI: 10.1007/978-1-4757-6484-0_16. URL: https://doi.org/10.1007/978-1-4757-6484-0_16.

[26] David Waltz. "Generating Semantic Descriptions From Drawings of Scenes With Shadows". In: (Oct. 2004).

[27] J. Cleary. "Logical arithmetic". In: *Future computing systems* 2.2 (1987), pp. 125–149.

[28] L. Jaulin et al. "Set inversion via interval analysis for nonlinear bounded-error estimation". In: *Automatica* 29.4 (1993), pp. 1053–1064.

[29] James L. Crowley et al. "Principles and techniques for sensor data fusion". In: *Signal Processing* 32.1 (1993). Intelligent Systems for Signal and Image Understanding, pp. 5–27. ISSN: 0165-1684. DOI: https://doi.org/10.1016/0165-1684(93)90034-8. URL: https://www.sciencedirect.com/science/article/pii/0165168493900348.

[30] J.J. Leonard et al. "Simultaneous map building and localization for an autonomous mobile robot". In: *Proceedings IROS '91:IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*. 1991, 1442–1447 vol.3. DOI: 10.1109/IROS.1991.174711.

[31] U.D Hanebeck et al. "Localization of fast mobile robots based on an advanced angle-measurement technique". In: *Control Engineering Practice* 4.8 (1996), pp. 1109–1118. ISSN: 0967-0661. DOI: https://doi.org/10.1016/0967-0661(96)00110-4. URL: https://www.sciencedirect.com/science/article/pii/0967066196001104.

[32] Michael Drumheller. "Mobile Robot Localization Using Sonar". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-9.2 (1987), pp. 325–332. DOI: 10.1109/TPAMI.1987.4767907.

[33] Sergey P. Shary. *Non-traditional intervals and their use. Which ones really make sense?* 2022. arXiv: 2202.03058 [math.NA]. URL: https://arxiv.org/abs/2202.03058.

[34] Ralph Kearfott. "Interval Computations: Introduction, Uses, and Resources". In: *Euromath Bulletin* 2 (Jan. 1996).

[35] E. Kaucher. "Algebraische Erweiterungen der Intervallrechnung unter Erhaltung der Ordnungs- und Verbandsstrukturen". In: *Grundlagen der Computer-Arithmetik*. Ed. by Rudolf Albrecht et al. Vienna: Springer Vienna, 1977, pp. 65–79. ISBN: 978-3-7091-8471-4. DOI: 10.1007/978-3-7091-8471-4_7. URL: https://doi.org/10.1007/978-3-7091-8471-4_7.

[36] Svetoslav Markov. "A Complete Axiomatization of Kaucher Interval Arithmetic". In: (2015).

[37] Wolfgang Enger. "Interval Ray Tracing --- a divide and conquer strategy for realistic computer graphics". In: *The Visual Computer* 9.2 (Feb. 1992), pp. 91–104. ISSN: 1432-2315. DOI: 10.1007/BF01901274. URL: https://doi.org/10.1007/BF01901274.

[38] URL: https://standards.ieee.org/ieee/1788/4431/.

[39] "IEEE Standard for Interval Arithmetic". In: *IEEE Std 1788-2015* (2015), pp. 1–97. DOI: 10.1109/IEEESTD.2015.7140721.

[40] "IEEE Standard for Interval Arithmetic (Simplified)". In: *IEEE Std 1788.1-2017* (2018), pp. 1–38. DOI: 10.1109/IEEESTD.2018.8277144.

[41] URL: https://standards.ieee.org/ieee/754/6210/.

[42] URL: https://cs.upc/~robert/mirror/interval-comp/intsoft.html.

[43] Nathalie Revol et al. "Testing interval arithmetic libraries, including their IEEE-1788 compliance". In: *LNCS*. Vol. 13827. Gdansk, Poland: Springer, Sept. 2022, pp. 428–440. DOI: 10.1007/978-3-031-30445-3_36. URL: https://inria.hal.science/hal-03674743.

[44] G. Alefeld et al. "The basic properties of interval arithmetic, its software realizations and some applications". In: *Computers Structures* 67.1 (1998), pp. 3–8. ISSN: 0045-7949. DOI: https://doi.org/10.1016/S0045-7949(97)00150-8. URL: https://www.sciencedirect.com/science/article/pii/S0045794997001508.

[45] Werner Hofschuster et al. "C-XSC 2.0 -- A C++ Library for Extended Scientific Computing". In: *Numerical Software with Result Verification*. Ed. by René Alt et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 15–35. ISBN: 978-3-540-24738-8.

[46] O. Knüppel. "PROFIL/BIAS---A fast interval library". In: *Computing* 53.3 (Sept. 1994), pp. 277–287. ISSN: 1436-5057. DOI: 10.1007/BF02307379. URL: https://doi.org/10.1007/BF02307379.

[47] Hervé Brönnimann et al. "The Boost Interval Arithmetic Library". In: *Real Numbers and Computers*. Lyon, France, 2003, pp. 65–80. URL: https://inria.hal.science/inria-00348711.

[48] Goualard Frederic. "GAOL, Not just another interval library". In: *University of Nantes, France* (2005).

[49] Jordan Ninin et al. "Overview of the IBEX library". In: *Journée SMAI-MODE*. toulouse, France, Mar. 2016. URL: https://hal.science/hal-04600450.

[50] Le Bars Fabrice Rohou Simon Desrochers Benoit. "The CODAC library". In: *Acta cybernetica* 26.4 (Mar. 2024), pp. 871–887. URL: https://hal.science/hal-04805950.

[51] Michael Lerch et al. "FILIB++, a fast interval library supporting containment computations". In: *ACM Trans. Math. Softw.* 32.2 (June 2006), pp. 299–324. ISSN: 0098-3500. DOI: 10.1145/1141885.1141893. URL: https://doi.org/10.1145/1141885.1141893.

[52] Nathalie Revol et al. "Motivations for an arbitrary precision interval arithmetic and the MPFI library". In: *Reliable Computing* 11.4 (Aug. 2005), pp. 275–290. DOI: 10.1007/s11155-005-6891-y. URL: https://inria.hal.science/inria-00544998.

[53] Laurent Fousse et al. "MPFR: A multiple-precision binary floating-point library with correct rounding". In: *ACM Trans. Math. Softw.* 33.2 (June 2007), 13–es. ISSN: 0098-3500. DOI: 10.1145/1236463.1236468. URL: https://doi.org/10.1145/1236463.1236468.

[54] Marco Nehmeier. "libieeep1788: A C++ Implementation of the IEEE interval standard P1788". In: *2014 IEEE Conference on Norbert Wiener in the 21st Century (21CW)*. 2014, pp. 1–6. DOI: 10.1109/NORBERT.2014.6893854.

[55] Walter F. Mascarenhas. *Moore: Interval Arithmetic in Modern C++*. 2016. arXiv: 1611.09567 [cs.MS]. URL: https://arxiv.org/abs/1611.09567.

[56] Oliver Heimlich. "Summer Workshop For Interval Methods". In: *Interval Arithmetic In GNU Octave*. ENS, 2016, pp. 27–28.

[57] Evgeniya Vorontsova. "Interval Computations in Julia programming language". In: *Summer Workshop on Interval Methods (SWIM) 2019, ENSTA*. Paris, France, July 2019. URL: https://hal.science/hal-02321950.

[58] Neil Henning. *How costly is changing the hardware rounding mode?* Apr. 2023. URL: https://www.neilhenning.dev/posts/howcostlyischangingroundingmode/.

[59] Jonathan Eckstein. "Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5". In: *SIAM Journal on Optimization* 4.4 (1994), pp. 794–814. DOI: 10.1137/0804046. eprint: https://doi.org/10.1137/0804046. URL: https://doi.org/10.1137/0804046.

[60] David A. Bader et al. "Parallel Algorithm Design for Branch and Bound". In: *Tutorials on Emerging Methodologies and Applications in Operations Research: Presented at Informs 2004, Denver, CO*. Ed. by H. J. G. New York, NY: Springer New York, 2005, pp. 5-1–5-44. ISBN: 978-0-387-22827-3. DOI: 10.1007/0-387-22827-6_5. URL: https://doi.org/10.1007/0-387-22827-6_5.

[61] J Mohan. "Experience with two parallel programs solving the traveling salesman problem". In: None. IEEE Computer Society Press,Silver Spring, MD, Dec. 1982. URL: https://www.osti.gov/biblio/5214576.

[62] Audrey Woods. *The death of Moore's law: What it means and what might fill the gap going forward*. URL: https://cap.csail.mit.edu/death-moores-law-what-it-means-and-what-might-fill-gap-going-forward.

[63] N. Wirth. "A plea for lean software". In: *Computer* 28.2 (1995), pp. 64–68. DOI: 10.1109/2.348001.

[64] Rupp Karl. *Microprocessor Trend Data*. https://github.com/karlrupp/microprocessor-trend-data. 2022.

[65] John L. Hennessy et al. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[66] Mason Woo et al. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[67] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6 (Mar. 2008), pp. 40–53. DOI: 10.1145/1401132.1401152.

[68] John E. Stone et al. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[69] URL: https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c?ncid=no-ncid.

[70] URL: https://modal.com/gpu-glossary/device-hardware/streaming-multiprocessor.

[71] Sylvain Collange et al. “Chapter 9 - Interval Arithmetic in CUDA”. In: *GPU Computing Gems Jade Edition*. Ed. by Wen-mei W. Hwu. Applications of GPU Computing Series. Boston: Morgan Kaufmann, 2012, pp. 99–107. ISBN: 978-0-12-385963-1. DOI: https://doi.org/10.1016/B978-0-12-385963-1.00009-5. URL: https://www.sciencedirect.com/science/article/pii/B9780123859631000095.

[72] Sylvain Collange et al. “A GPU interval library based on Boost.Interval”. In: (July 2008).

[73] NVIDIA. *7. Single Precision intrinsics*. Sept. 2025. URL: https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/group__CUDA__MATH__INTRINSIC__SINGLE.html.

[74] NVIDIA. *Parallel Thread Execution ISA Version 9.0*. Sept. 2025. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/#type-information-for-instructions-and-operands.

[75] Zsolt Bagóczki et al. “A Parallel Interval Arithmetic-based Reliable Computing Method on a GPU”. In: *Acta Cybernetica* 23 (June 2017), pp. 491–501. DOI: 10.14232/actacyb.23.2.2017.4.

[76] Hongzhen Zhang et al. *Accelerating Deterministic Global Optimization via GPU-parallel Interval Arithmetic*. July 2025. DOI: 10.48550/arXiv.2507.20769.

[77] Luca Feranti et al. *IntervalLinearAlgebra.jl: linear algebra done rigorously*. Sept. 2021. DOI: 10.5281/zenodo.5363563. URL: https://github.com/juliaintervals/IntervalLinearAlgebra.jl.

[78] Evgeniya Vorontsova. “Interval Computations in Julia programming language”. In: *Summer Workshop on Interval Methods (SWIM) 2019, ENSTA*. Paris, France, July 2019. URL: https://hal.science/hal-02321950.

[79] Wilfrid da Silva et al. “GPU-accelerated Interval Arithmetic to solve the Apollonius Problem applied to a Stereo Drift Chamber”. working paper or preprint. Jan. 2024. URL: https://hal.science/hal-04413938.

[80] Ruud van der Pas. “Using OpenMP to Parallelize Interval Algorithms”. In: *SCAN 2008*. Sept. 2008.

[81] Anthony P. Leclerc. "Interval analysis: parallel methods for global optimization-Interval Analysis: Parallel Methods for Global Optimization". In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas et al. Boston, MA: Springer US, 2009, pp. 1709–1717. ISBN: 978-0-387-74759-0. DOI: 10.1007/978-0-387-74759-0_299. URL: https://doi.org/10.1007/978-0-387-74759-0_299.

[82] Nathalie Revol et al. "Parallel Implementation of Interval Matrix Multiplication". In: *Reliable Computing* 19.1 (2013), pp. 91–106. URL: https://inria.hal.science/hal-00801890.

[83] David Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163. URL: https://doi.org/10.1145/103162.103163.

[84] "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Std 754-1985* (1985), pp. 1–20. DOI: 10.1109/IEEESTD.1985.82928.

[85] Milos Ercegovac. "Digital Arithmetic". In: (2004). DOI: 10.1016/B978-1-55860-798-9.X5000-3. URL: https://doi.org/10.1016/b978-1-55860-798-9.x5000-3.

[86] Jean-Michel Muller et al. *Handbook of Floating-Point Arithmetic, 2nd edition*. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9. Birkhäuser Boston, 2018, p. 632.

[87] Florent de Dinechin et al. *Application-Specific Arithmetic*. Springer, 2024. URL: https://link.springer.com/book/10.1007/978-3-031-42808-1.

[88] Florent de Dinechin et al. "Designing Custom Arithmetic Data Paths with FloPoCo". In: *IEEE Design & Test of Computers* 28.4 (July 2011), pp. 18–27.

[89] Cristian KLEIN. "High-Quality and High-Throughput Random Number Generation in FPGAs". In: (2008).

[90] Naohito Nakasato et al. "High Performance High-Precision Floating-Point Operations on FPGAs Using OpenCL". In: *2018 International Conference on Field-Programmable Technology (FPT)*. 2018, pp. 262–265. DOI: 10.1109/FPT.2018.00049.

[91] Joel S. Emer et al. "A Characterization of Processor Performance in the vax-11/780". In: *SIGARCH Comput. Archit. News* 12.3 (Jan. 1984), pp. 301–310. ISSN: 0163-5964. DOI: 10.1145/773453.808199. URL: https://doi.org/10.1145/773453.808199.

[92] S.P. Morse et al. "The Intel 8086 Microprocessor: a 16-bit Evolution of the 8080". In: *Computer* 11.6 (1978), pp. 18–27. DOI: 10.1109/C-M.1978.218219.

[93] David A. Patterson et al. “The case for the reduced instruction set computer”. In: *SIGARCH Comput. Archit. News* 8.6 (Oct. 1980), pp. 25–33. ISSN: 0163-5964. DOI: 10.1145/641914.641917. URL: https://doi.org/10.1145/641914.641917.

[94] George Radin. “The 801 minicomputer”. In: *SIGPLAN Not.* 17.4 (Mar. 1982), pp. 39–47. ISSN: 0362-1340. DOI: 10.1145/960120.801824. URL: https://doi.org/10.1145/960120.801824.

[95] David A. Patterson et al. “RISC I: A Reduced Instruction Set VLSI Computer”. In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 443–457.

[96] John Hennessy et al. “MIPS: A microprocessor architecture”. In: *ACM SIGMICRO Newsletter* 13 (Dec. 1982), pp. 17–22. DOI: 10.1145/1014194.800930.

[97] Dave Ditzel. *The RISC Journey from One to a Million Processors*. URL: https://microarch.org/micro55/media/keynote_ditzel.pdf.

[98] John Hennessy et al. *Computer Architecture - A Quantitative Approach*. Jan. 2007.

[99] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Order Number: 325462-077US. Intel Corporation. Santa Clara, CA, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[100] Arm Ltd. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile (DDI 0487A.a)*. Defines AArch64 (64-bit) and AArch32 (32-bit) execution states with backward compatibility. Arm Ltd. Cambridge, UK, 2013. URL: https://developer.arm.com/documentation/ddi0487/latest.

[101] Unknown. “Intel and AMD renew patent cross-licensing pact”. In: *EE Times* (2001). Discusses the renewal of the longstanding x86 cross-license agreement. URL: https://www.eetimes.com/intel-and-amd-renew-patent-cross-licensing-pact/.

[102] Dmytro Spasiuk. “The history of Apple Silicon: how Apple’s ARM processors changed the computer industry”. In: *ITC.ua* (2024). Discusses Apple’s evolution from early ARM usage through custom ARM-based Apple Silicon and transition from Intel x86, situating Apple Silicon within ARM ISA adoption and evolution. URL: https://itc.ua/en/articles/the-history-of-apple-silicon-how-apple-s-arm-processors-changed-the-computer-industry/.

[103] Andrew Waterman et al. “RISC-V: An Open Standard Instruction Set Architecture”. In: *2011 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2011, pp. 1–2. DOI: 10.1109/ISPASS.2011.5764039.

[104] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Version 2.0*. Tech. rep. UCB/EECS-2014-54. The canonical specification for the base RISC-V ISA. EECS Department, University of California, Berkeley, 2014. URL: https://github.com/riscv/riscv-isa-manual.

[105] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.7*. Tech. rep. UCB/EECS-2015-49. Defines RISC-V privileged ISA and system-level specification. EECS Department, University of California, Berkeley, 2015. URL: https://github.com/riscv/riscv-isa-manual.

[106] Anisha Sharma. *How Next-Gen Chips Are Unlocking RISC-V's Customization Advantage*. RISC-V International ecosystem blog. 2025. URL: https://riscv.org/ecosystem-news/2025/07/how-next-gen-chips-are-unlocking-risc-vs-customization-advantage/.

[107] Sifive. *SiFive Int8 Matrix Multiplication Extensions Specification - Version 1.1*. Sept. 2023.

[108] Alperen Bolat et al. "Microarchitecture Design and Benchmarking of Custom SHA-3 Instruction for RISC-V". In: *arXiv:2508.20653 [cs.CR]* (2025). URL: https://arxiv.org/abs/2508.20653.

[109] Muhammad Sabih et al. *Hardware/Software Co-Design of RISC-V Extensions for Accelerating Sparse DNNs on FPGAs*. arXiv:2504.19659 [cs.AR]. 2025. URL: https://arxiv.org/abs/2504.19659.

[110] Thinh Hung Phạm et al. *CIDPro: Custom Instructions for Dynamic Program Diversification*. arXiv:1809.01221 [cs.CR]. 2018. URL: https://arxiv.org/abs/1809.01221.

[111] Simon Rohou et al. "The Codac Library". In: *Acta Cybernetica* 26.4 (Mar. 2024), pp. 871–887. DOI: 10.14232/actacyb.302772. URL: https://hal.science/hal-04805950.

[112] E.H. Moore. *Introduction to a form of general analysis*. Vol. 2. Yale University Press, 1910.

[113] T.S. Blyth. *Lattices and Ordered Algebraic Structures*. Springer, ISBN 1-85233-905-5, 2005.

[114] B. A. Davey et al. *Introduction to Lattices and Order*. (ISBN 0521784514): Cambridge University Press, 2002.

[115] P. Filiol et al. *Codes associated with the paper entitled: A new interval arithmetic to generate the complementary of contractors*. www.ensta-bretagne.fr/jaulin/iota.html. 2022.

[116] R.E. Moore et al. *Introduction to Interval Analysis*. Philadelphia, PA: SIAM, 2009.

[117] N. Revol. “Introduction to the IEEE 1788-2015 Standard for Interval Arithmetic”. In: *10th International Workshop on Numerical Software Verification - NSV 2017* ().

[118] IEEE Microprocessor Standards Commitee. “IEEE 1788-2015 Standard for Interval Arithmetic”. In: *https://standards.ieee.org/ieee/1788/4431/* (2015).

[119] RISC-V standard. “37.9 - Non-standard extension names”. In: 2025-05-08, pp. 626–627. URL: https://docs.riscv.org/reference/isa.

[120] RISC-V standard. “35 - RV32/64G Instruction Set Listings”. In: 2025-05-08, pp. 608–609. URL: https://docs.riscv.org/reference/isa.

[121] W. Waite W. Cody. “Software Manual for the Elementary Functions”. In: ed. by Prentice-Hall. 1980.

[122] W. J. Cody. “Implementation and testing of function software”. In: *Problems and Methodologies in Mathematical Software Production*. Berlin, Germany, 1982.

[123] Florent de Dinechin et al. “Fixed-point trigonometric functions on FPGAs”. In: *SIGARCH Comput. Archit. News* 41.5 (June 2014), pp. 83–88. ISSN: 0163-5964. DOI: 10.1145/2641361.2641375. URL: https://doi.org/10.1145/2641361.2641375.

[124] Florent De Dinechin et al. “Hardware Implementations of Fixed-Point Atan2”. In: *2015 IEEE 22nd Symposium on Computer Arithmetic*. 2015, pp. 34–41. DOI: 10.1109/ARITH.2015.23.

[125] Krste Asanović et al. “The Rocket Chip Generator”. In: 2016. URL: https://api.semanticscholar.org/CorpusID:5364470.

[126] Colin Schmidt. *RISC-V “Rocket Chip” Tutorial*. University Lecture. 2024. URL: https://riscv.org/wp-content/uploads/2024/12/riscv-rocket-chip-tutorial-bootcamp-jan2015.pdf.

[127] Jack Koenig. “A Hardware Accelerator for Computing an Exact Dot Product”. PhD thesis. 2018. URL: https://people.eecs.berkeley.edu/~krste/papers/koenig-ms.pdf.

[128] Theresa Tao. “A Tightly Integrated Generic Instruction RISC-V Accelerator (TIGRA) for the Rocket Core”. PhD thesis. 2021.

[129] Stefano Di Mascio et al. “Extending the noel-V platform with a RISC-V vector processor for space applications”. In: *Journal of Aerospace Information Systems* 20.9 (Sept. 2023), pp. 565–574. DOI: 10.2514/1.i011097.

[130] Tristan Gingold. *GHDL*. Version 6.0.0. 2017. URL: http://ghdl.free.fr/.

[131] Unai Martinez-Corral. *VHDL DPI/FFI based on GHDL's implementation of VHPIDIRECT*. URL: https://github.com/umarcor/ghdl-cosim.

[132] C. Dawson et al. “The Verilog Procedural Interface for the Verilog Hardware Description Language”. In: *Proceedings of the 1996 IEEE International Verilog HDL Conference (IVC '96)*. USA: IEEE Computer Society, 1996, p. 17. ISBN: 0818674296.

[133] “IEEE Standard VHDL Language Reference Manual - Procedural Language Application Interface”. In: *IEEE Std 1076c-2007 (Amendment to IEEE Std 1076-2002)* (2007), pp. 1–212. DOI: 10.1109/IEEESTD.2007.4299594.

[134] Zhiwei Tang. “The Zynq-7000 SoC on UltraScale Architecture”. In: Jan. 2021, pp. 231–236. ISBN: 978-3-030-51555-3. DOI: 10.1007/978-3-030-51556-0_34.

This page intentionally left blank.

Appendices

This page intentionally left blank.

A

Top Level view ot the NVIDIA Hooper Architecture

Figure A.1: *Architecture of the NVIDIA H100 (Hooper) - Credit to NVIDIA Hooper Whitepaper* [69]

B

Implementing Addition Contractors in *Libctc_no_acc*

Listing 14: *An Overview of the Interval Layer in Libctc_no_acc*

```
/* Addition for intervals - without xinterval */
interval_t itv_add(interval_t x, interval_t y) {
    uint32_t iota_add = ((x.iota | y.iota) > 0) ? 1 : 0;

    /* Omitted early exit logic due to empty input */

    float new_lb = x.lb + y.lb;
    float new_ub = x.ub + y.ub;
    interval_t res = {.lb = new_lb, .ub = new_ub, .empty = 0, .iota = iota_add};
    return res;
}

/* Subtraction for intervals - without xinterval */
interval_t itv_sub(interval_t x, interval_t y) {
    uint32_t iota_sub = ((x.iota | y.iota) > 0) ? 1 : 0;

    /* Omitted early exit logic due to empty input */

    float new_lb = x.lb - y.ub;
    float new_ub = x.ub - y.lb;
    interval_t res = {.lb = new_lb, .ub = new_ub, .empty = 0, .iota = iota_sub};
    return res;
}

/* Intersection for intervals - without xinterval */
interval_t itv_intersection(interval_t interval_1, interval_t interval_2) {
    uint32_t iota_inter = ((interval_1.iota | interval_2.iota) > 0) ? 1 : 0;

    /* Omitted early exit logic due to empty/disjoint input */

    float new_lb = (interval_1.lb <= interval_2.lb) ? interval_2.lb : interval_1.lb;
    float new_ub = (interval_1.ub <= interval_2.ub) ? interval_1.ub : interval_2.ub;
    interval_t res = {.lb = new_lb, .ub = new_ub, .empty = 0, .iota = iota_inter};
    return res;
}
```

Listing 15: *Implementing Addition Contractors in Libctc_no_acc*

```
/* Addition forward contractor */
interval_t addfwctc(interval_t x, interval_t y) {
    return itv_add(x,y);
}

/* Addition backward contractor 1 */
/* Given that Z = X + Y */
/* This function return Xc such as : */
/* Xc = X   (Z - Y) */
interval_t addbwctc1(interval_t x, interval_t y, interval_t z) {
    interval_t subtraction = itv_sub(z, y);
    interval_t intersected = itv_intersection(x, subtraction);
    return intersected;
}

/* Addition backward contractor 2 */
/* Given that Z = X + Y */
/* This function return Xc such as : */
/* Yc = Y   (Z - X) */
interval_t addbwctc2(interval_t x, interval_t y, interval_t z) {
    interval_t subtraction = itv_sub(z, x);
    interval_t intersected = itv_intersection(y, subtraction);
    return intersected;
}
```

C

Implementing Addition Contractors in *Libctc_acc*

Listing 16: *An Overview of the Interval Layer in Libctc_acc*

```
/* inline function which makes use of xinterval instruction itv.add */
/* inputs loaded from double registers */
/* output stored in double register */
inline interval_t __attribute__((always_inline)) itv_add(interval_t itv1, interval_t
↪  itv2) {
    interval_t result;
    asm("itv.add %0, %1, %2" : "=f"(result) : "f"(itv1), "f"(itv2));
    return result;
}

/* inline function which makes use of xinterval instruction itv.sub */
/* inputs loaded from double registers */
/* output stored in double register */
inline interval_t __attribute__((always_inline)) itv_sub(interval_t itv1, interval_t
↪  itv2) {
    interval_t result;
    asm("itv.sub %0, %1, %2" : "=f"(result) : "f"(itv1), "f"(itv2));
    return result;
}

/* inline function which makes use of xinterval instruction itv.ext.i */
/* inputs loaded from double register */
/* output stored in integer register */
inline int __attribute__((always_inline)) itv_intersection(interval_t itv) {
    int result;
    asm("itv.ext.i %0, %1" : "=r"(result) : "f"(itv));
    return result;
}
```

Listing 17: *Implementing Addition Contractors in Libctc_acc*

```
/* Addition forward contractor */
interval_t addfwctc(interval_t x, interval_t y) {
    return itv_add(x, y);
}

/* Addition backward contractor 1 */
/* Given that Z = X + Y */
/* This function return Xc such as : */
/* Xc = X   (Z - Y) */
interval_t addbwctc1(interval_t x, interval_t y, interval_t z) {
    interval_t subtraction = itv_sub(z, y);
    interval_t intersected = itv_intersection(x, subtraction);
    return intersected;
}

/* Addition backward contractor 2 */
/* Given that Z = X + Y */
/* This function return Xc such as : */
/* Yc = Y   (Z - X) */
interval_t addbwctc2(interval_t x, interval_t y, interval_t z) {
    interval_t subtraction = itv_sub(z, x);
    interval_t intersected = itv_intersection(y, subtraction);
    return intersected;
}
```