Interval Analysis on the RISC-V for mobile robotics

Design of a custom ISA extension

Pierre Filiol
Luc Jaulin
Jean-Christophe Le Lann
Théotime Bollengier

Contents

1. Motivations
2. Fundamentals of RISC-V
3. Strategy Used in This Work
4. Contours of the Extension
5. Hardware Implementation
6. Toolchain Adaptation
7. Validation and Results
8. Conclusion

Contents

1 Motivations

2 Fundamentals of RISC-V

3 Strategy Used in This Work

4 Contours of the Extension

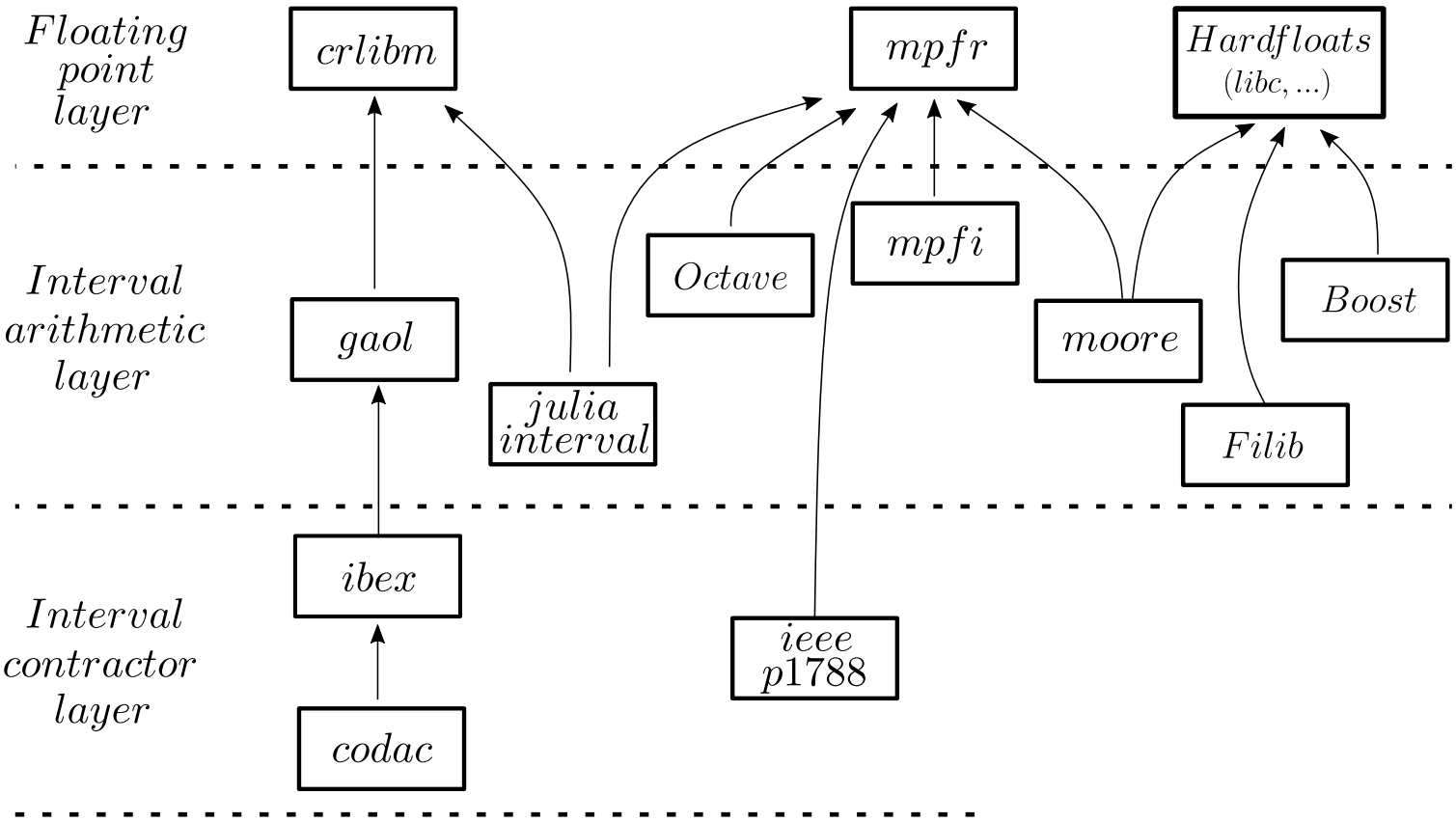
5 Hardware Implementation

6 Toolchain Adaptation

7 Validation and Results

8 Conclusion

The Software Ecosystem



The most popular libraries are very diverse in terms of

- Programming languages
- Abstraction level
- Design trade-off
- Architecture targeted

The Challenges of Intervals Software

- Implementing intervals correctly in software is **notoriously hard**.

A lot of the difficulties comes from IEEE-754 floating-point

- Rounding
- Precision
- Overall performances

About Rounding

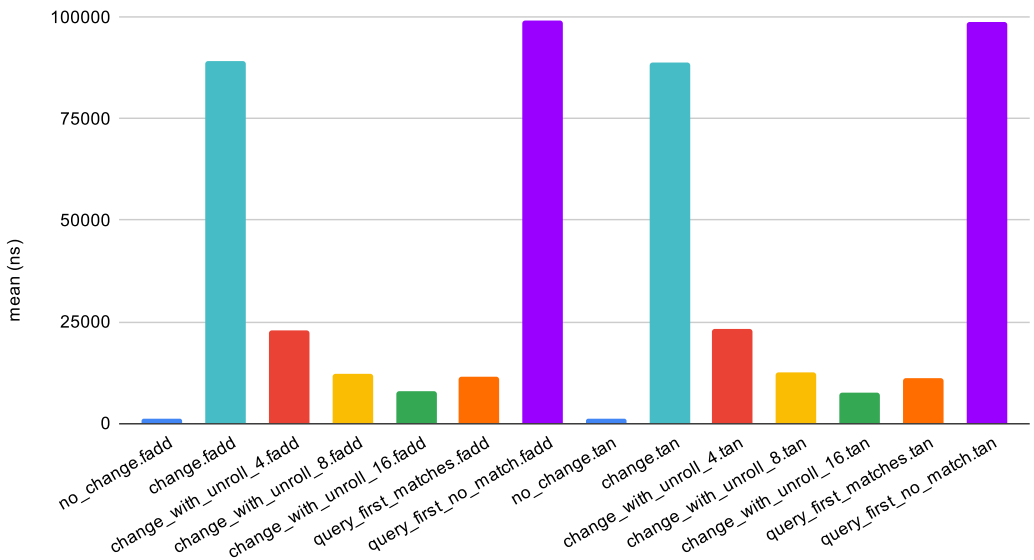
- To ensure that the true value is included in the result, intervals must be correctly rounded.

- This involves frequent switches between multiple IEEE-754 rounding modes: Round-up (RU) and Round-Down (RD).

- **Example of the addition**:

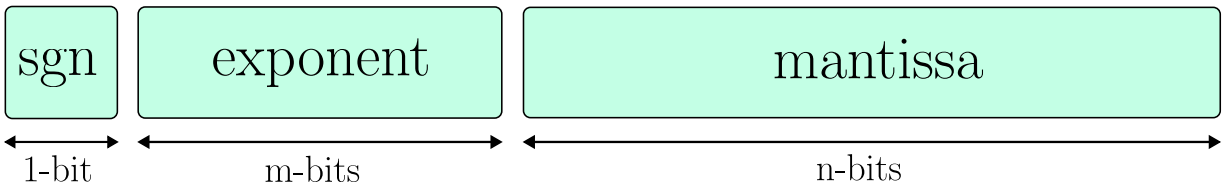
$$[x] + [y] = [\lfloor \underline{x} + \underline{y} \rfloor, \lceil \bar{x} + \bar{y} \rceil] \qquad \text{where } \lfloor . \rfloor \text{ and } \lceil . \rceil \text{ are RD/RU}$$

- **This is a performance killer on x86/arm platforms (up to 70× slower)**



About precision

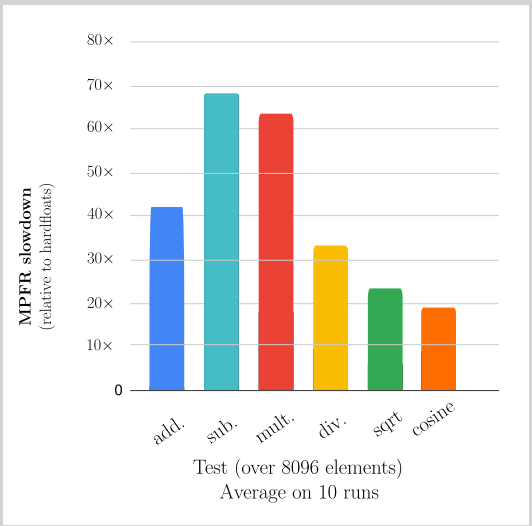
- IEEE-754 floating-points available in multiple formats:



type	m	n
float64	11	53
float32	8	24
float16	5	11

Interval libraries handle it in multiple ways

- Using **softfloat** (MPFR, CRlibm, ...)



- Using **hardfloat** (FPU, GPU, ...)
  - Limited to predefined precisions (f64, f32, f16)
  - "One size fits all" approach

The needs of embedded robotics

- Embedded robotics plays with specific rules:
  - **Time-critical applications**
  - **Varying precision requirements**
  - **Need guarantees, even at the cost of pessimism**

In this presentation

- Design of a dedicated hardware architecture for interval analysis.
- Targeted at mobile robotics.
- Implemented as a RISC-V custom extension.

Contents

1 Motivations

2 Fundamentals of RISC-V

3 Strategy Used in This Work

4 Contours of the Extension

5 Hardware Implementation

6 Toolchain Adaptation

7 Validation and Results

8 Conclusion

An open and accessible standard

Openness

The RISC-V standard has a specific approach

- The ISA is open and free of charge.
- Being RISC-V compliant = Implementing the instruction set.

RISC-V processors are shipped in two ways:

- As a **manufactured chip** (founder approach).
- As an *Intellectual Property* (IP) **softcore** (fabless approach) for synthesis on ASIC/FPGA.

Hardware shipped as software is handy

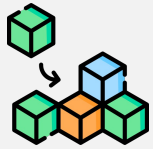
As researchers, softcores gives us the opportunity to **experiment**, **benchmark** & **modify** a processor.

Figure: Payed and closed-source

Figure: Payed support and open-source

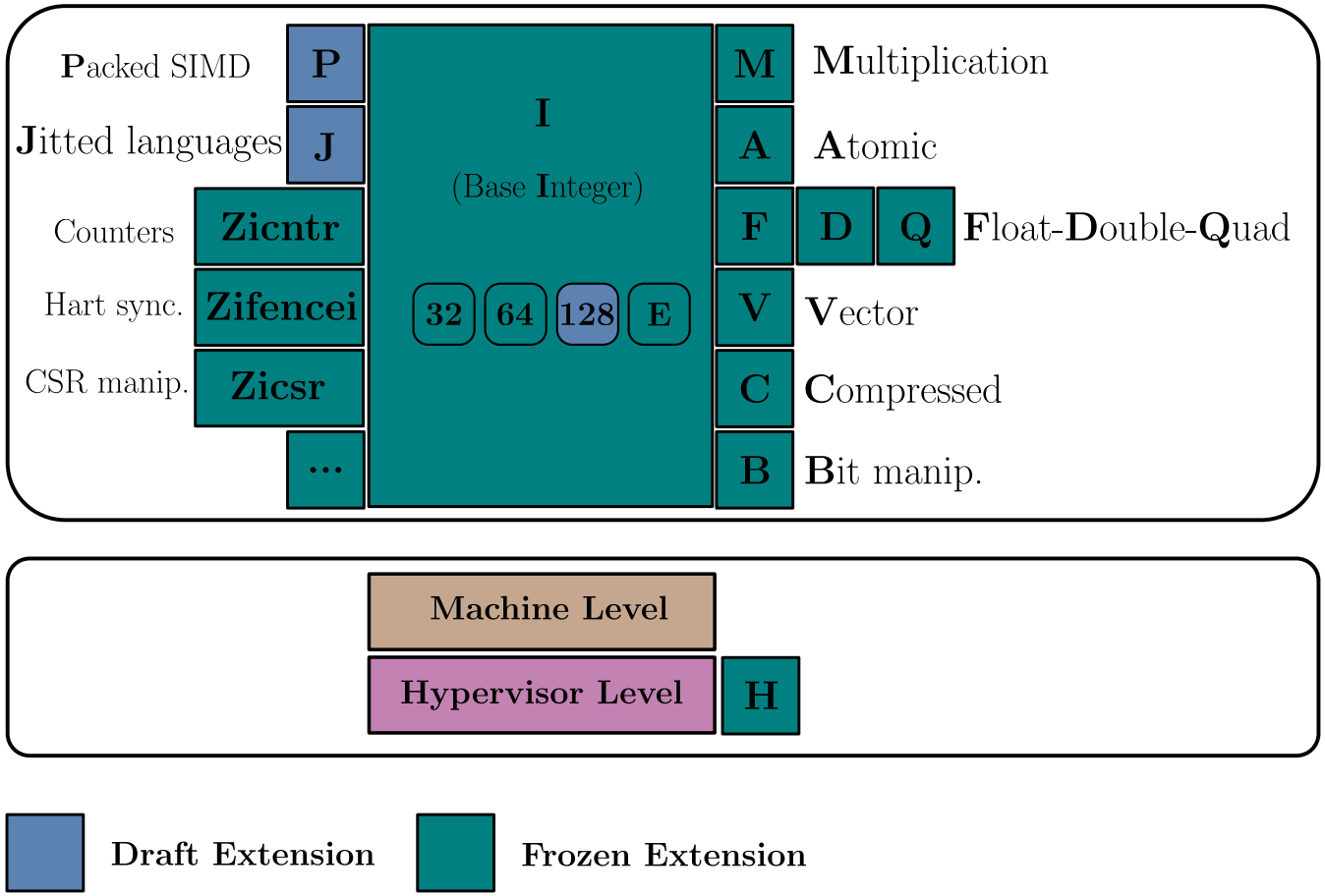
Figure: Free and open-source

A Modular Approach to ISA

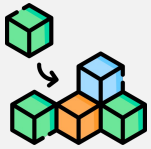


Modularity

- Instructions are regrouped in purpose-oriented modules called extensions.



Pay for what you use

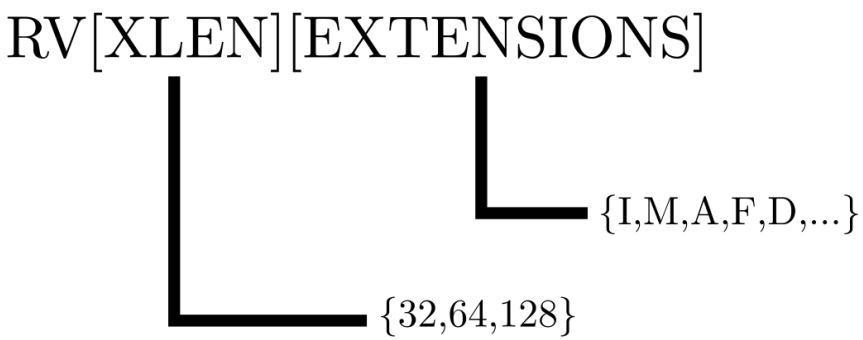


Modularity

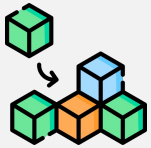
▶ Extension I defines the minimal subset of instructions and is mandatory to every design.

▶ The other extensions are optional and implemented when the application requires it.

▶ The extensions available in a particular RISC-V design are described through the following nomenclature:



Example: RV32-IMAF



Modularity

RV32I — Base Integer

ADD ADDI AND ANDI BEQ
SUB SRL OR ORI BNE
SLL SRLI XOR XORI BGE
SLLI SLTU SRA LUI BGEU
SLT SLTIU SRAI AUIPC BLT
SLTI LBU LB SB BLTU
FENCE LHU LH SH JAL
FENCE.TSO PAUSE LW SW JALR
ECALL EBREAK

RV32A — Atomic operations

LR.W SC.W AMOAND.W AMOOR.W AMOXOR.W
AMOADD.W AMOMIN.W AMOMAX.W AMOMINU.W AMOMAXU.W
AMOSWAP.W

RV32M — Hardware multiplication/division

MULH DIV MUL REM REMU
MULHU DIVU
MULHSU

Zicsr — Control state registers

CSRRW CSRRS CSRRC CSRRWI CSRRSI
CSRRCI

RV32F — Floating-points (single precision)

FLW FMADD.S FMSUB.S FADD.S FSUB.S
FSW FNMADD.S FNMSUB.S FMUL.S FDIV.S
FSQRT.S FSGNJ.S FSGNJN.S FSGNJX.S FMIN.S
FCLASS.S FCVT.S.W FCVT.W.S FEQ.S FMAX.S
FMV.W.X FCVT.S.WU FCVE.WU.S FLT.S FLE.S
FMV.X.W

Zifencei — Hart synchronization

FENCE.I

Sometimes You Want More

Extensibility 

- Standard extensions are meant for general-purpose computing.
- Intervals are too niche to be part of the official standard.
- Do not worry! The RISC-V standard has you covered.

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

- The RISC-V GCC toolchain supports the addition of new instructions.

Contents

1 Motivations

2 Fundamentals of RISC-V

3 Strategy Used in This Work

4 Contours of the Extension

5 Hardware Implementation

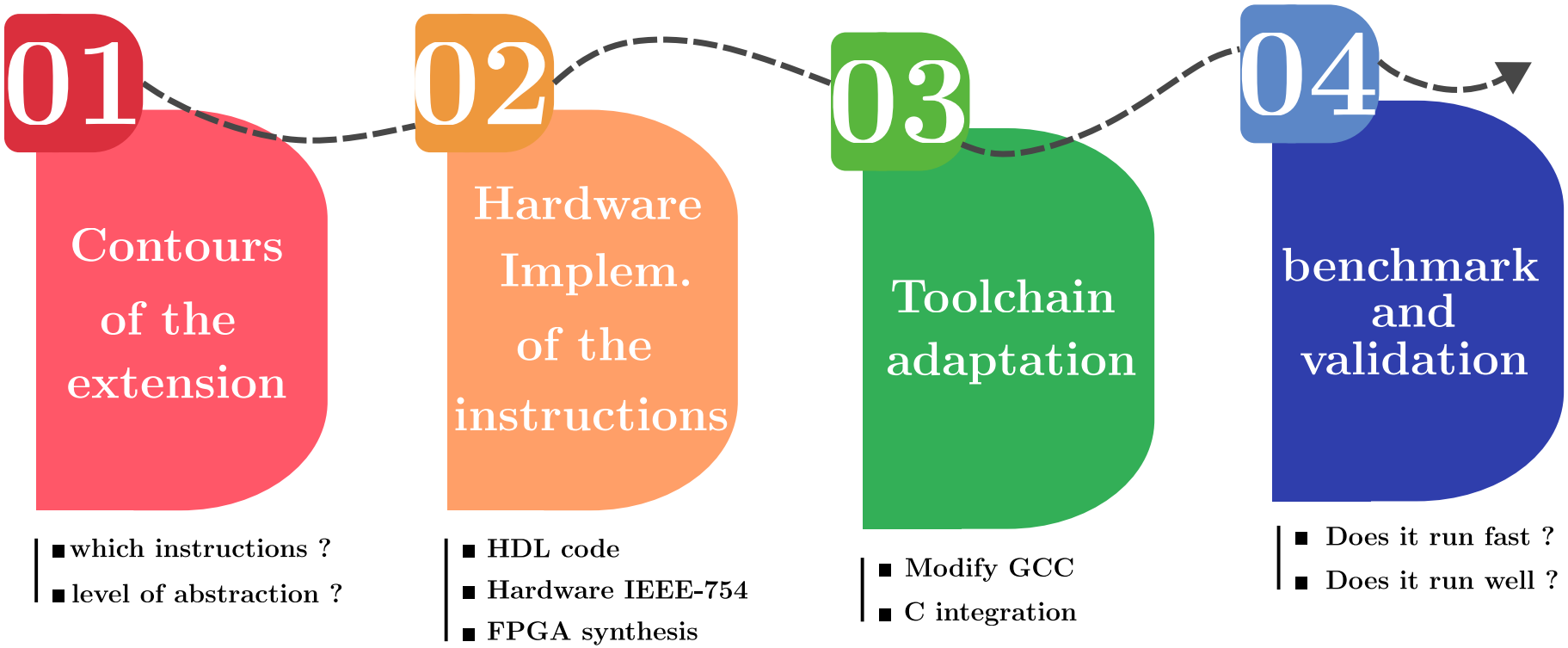
6 Toolchain Adaptation

7 Validation and Results

8 Conclusion

Workflow

▶ **Goal**: Design a custom RISC-V extension called **xinterval** to accelerate intervals at the operator level.



Contents

1 Motivations

2 Fundamentals of RISC-V

3 Strategy Used in This Work

4 Contours of the Extension

5 Hardware Implementation

6 Toolchain Adaptation

7 Validation and Results

8 Conclusion

A Contractor-centric Approach

▶ In embedded robotics, intervals are used in many recurring problems.

Examples

- Localization
- Robust control
- Optimization
- ...

▶ Still, the workflow is always the same:

1. Translate the situation as a *Constraint Satisfaction Problem* (CSP).
2. Derive the relevant contractors with a forward-backward methodology (e.g HC4R).
3. Use the contractor as it is or use a *Branch and Bound* strategy such as SIVIA.

An example - CSP

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

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

- Let us build a contractor for this problem using HC4R

An Example - HC4R

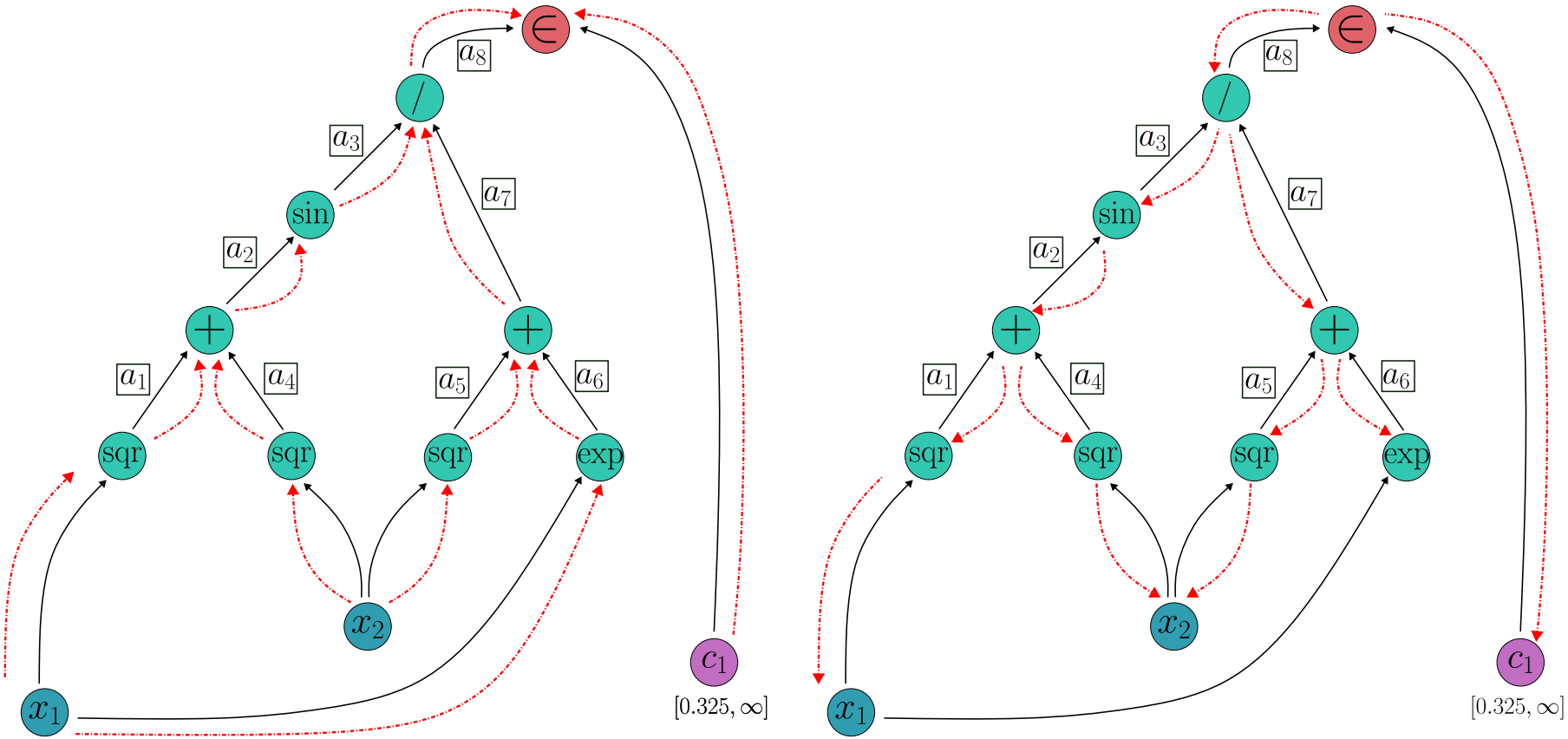


Figure: Forward (left) and backward (right) propagations of HC4R

An Example - Forward-Backward Contractor for $\mathcal{H}$

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 1: Contractor $\overrightarrow{\mathcal{C}_{\mathcal{H}}}$

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 $[x_1],[x_2]$

Algorithm 2: Contractor $\overleftarrow{\mathcal{C}_{\mathcal{H}}}$

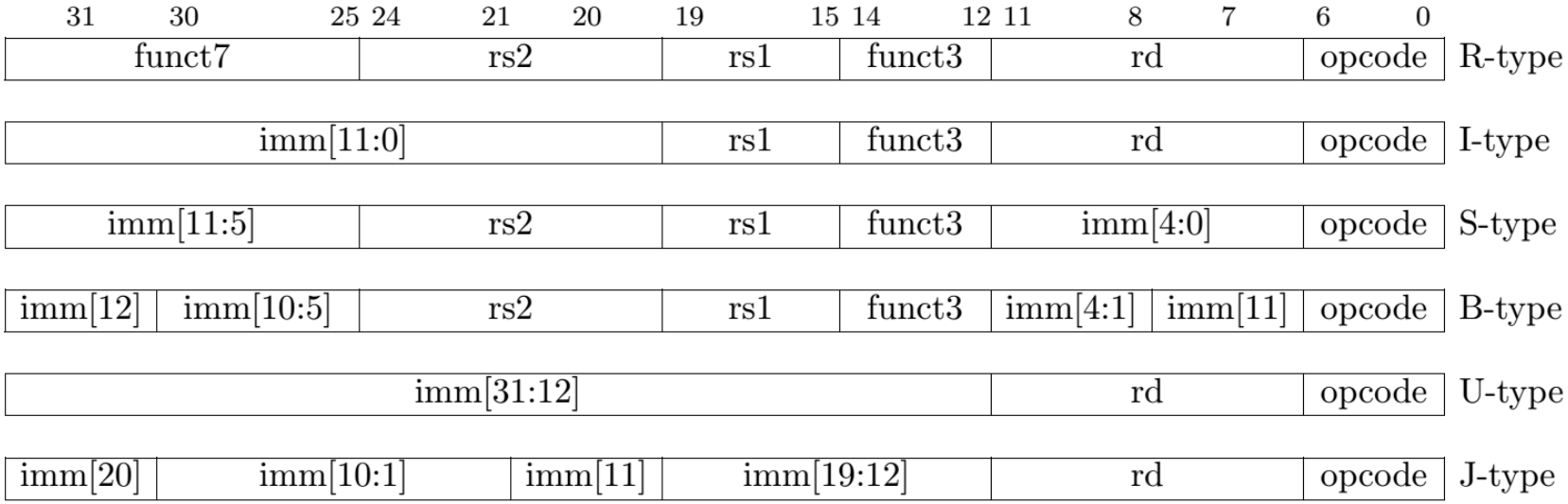
Strategy for **xinterval**

▶ The aforementionned workflow is intuitive for the programmer and accommodate most situations.

Strategy

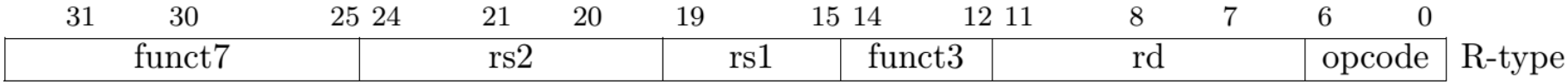
We need **xinterval** to accelerate the forward and backward contractors of the most recurring elementary operators on intervals.

Problem: How to translate this to RISC-V compliant instructions?



The Format of **xinterval** Instructions

- All the instructions from **xinterval** use the R-Type format (register to register).



  - **opcode**: The opcode number. We use the *custom-0* one (0xB) which is dedicated to custom extensions.
  - $\mathrm{r_d}$: The destination register.
  - $\mathrm{rs_1}$, $\mathrm{rs_2}$: The source registers (up to two arguments).
  - $\mathrm{funct_3}, \mathrm{funct_7}$: 3-bits and 7-bits identifiers. The pair must be unique to each instruction.

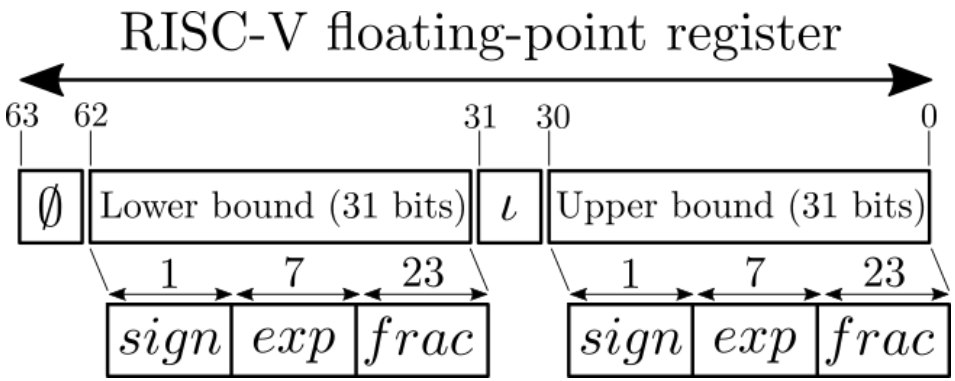
Example: itv.add (interval addition)

The assembly instruction **itv.add f3 f1 f2** performs the addition of the two intervals contained in floating-point registers **f1** and **f2** and stores the result in floating-point register **f3**.

Fitting Intervals Into Floating-point Registers

- Intervals are stored in the 64-bit registers dedicated to double precision floating-points.

- The following interval model is used:



It contains:

- A 1-bit flag for emptyness
- A 1-bit flag for iota
- Two custom-float bounds

Publication about the iota flag

Filiol and al. "A new interval arithmetic to generate the complementary of contractors." Acta Cybernetica, vol. 26, no. 4, 21 Mar. 2024, pp. 817–838, https://doi.org/10.14232/actacyb.300840.

A tour of **xinterval** instructions

register content interpretation	
reg interval	reg integer
reg float32	reg float64

- **Instructions to create intervals**

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	✓	✓	✓	$\{r_d\} = \text{itv}(\{rs_1\}, \{rs_2\}) \cap \iota$	Create interval from float bounds + iota

- **Instructions to extract data from an interval**

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

A Tour of **xinterval** Instructions

register content interpretation	
reg interval	reg integer
reg float32	reg float64

- **Instructions for set operations on intervals**

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

- **Instructions for arithmetic operations on intervals**

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

A Tour of **xinterval** Instructions

register content interpretation	
reg **interval**	reg **integer**
reg **float32**	reg **float64**

- **Instructions for real functions on intervals**

asm	$\mathbf{r_d}$	$\mathbf{rs_1}$	$\mathbf{rs_2}$	effect	description
itv.sqr	✓	✓	✗	$\{r_d\} = \mathrm{sqr}(\{rs_1\})$	square of the selected itv
itv.sqr_revh	✓	✓	✗	$\{r_d\} = \mathrm{itv}(-\sqrt{lb(\{rs_1\})}, \sqrt{ub(\{rs_1\})})$	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

Contents

1 Motivations

2 Fundamentals of RISC-V

3 Strategy Used in This Work

4 Contours of the Extension

5 Hardware Implementation

6 Toolchain Adaptation

7 Validation and Results

8 Conclusion

Implementing **xinterval** Instructions in Hardware

- Porting **xinterval** instructions in hardware.

How to do it ?

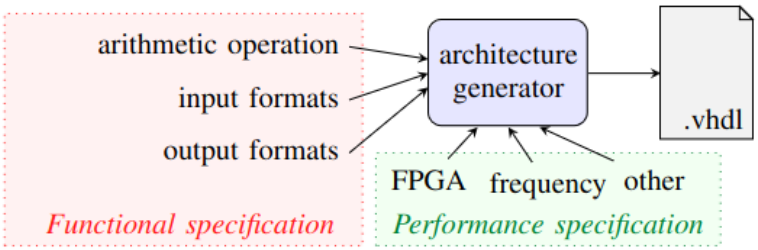
- Implementation using the VHDL language.
- Synthesis on FPGA.
- Choice of a low to middle-end chip (XC7Z020-1CLG400C).

This is challenging

- Heavily IEEE-754 reliant code (hard).
- Specificities of the FPGA target.

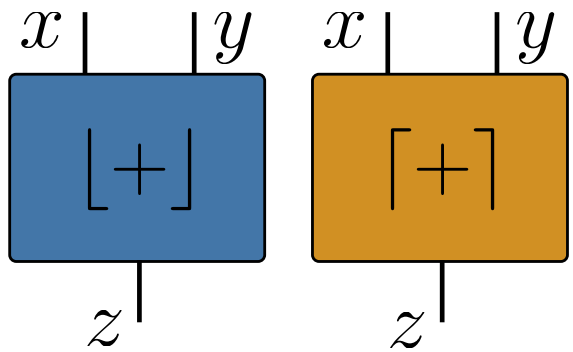
How we Handled Hardware Floats?

- Use of the FloPoCo project (INRIA).




```
1
2 *** Final report ***
3 Output file: flopoco.vhdl
4 Target: Kintex7 @ 100 MHz
5 |---Entity RightShifterSticky24_by_max_26_F100_uid4
6 |      Not pipelined
7 |---Entity IntAdder_27_F100_uid6
8 |      Pipeline depth = 1
9 |---Entity Normalizer_Z_28_28_28_F100_uid8
10 |      Not pipelined
11 |---Entity IntAdder_33_F100_uid11
12 |      Not pipelined
13 Entity FPAdd_7_23_F100_uid2
14    Pipeline depth = 1
```

- This is very practical:
  - Custom float format.
  - Selectable rounding mode.
  - Automatic pipelining (heuristic)
  - Huge catalog of operators
- We use this convention:

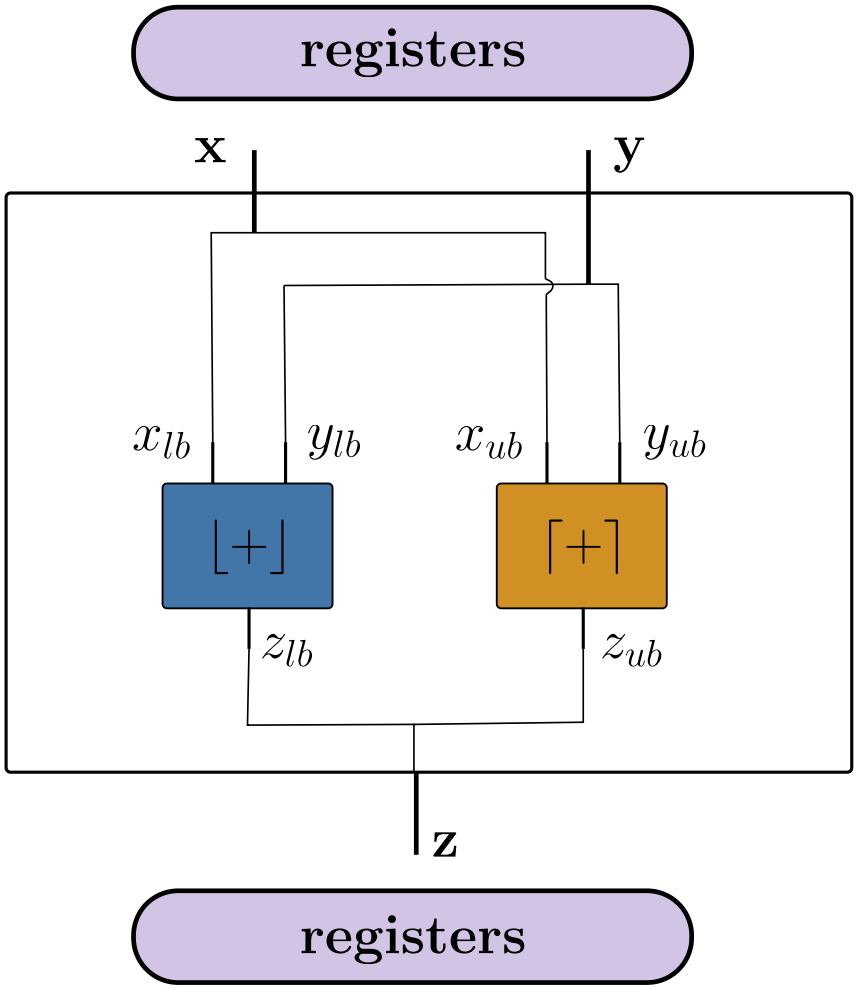


More Information

Dinechin, Pasca. "Designing custom arithmetic data paths with FloPoCo" (2011)

A Simple Example: Addition With **itv.add**

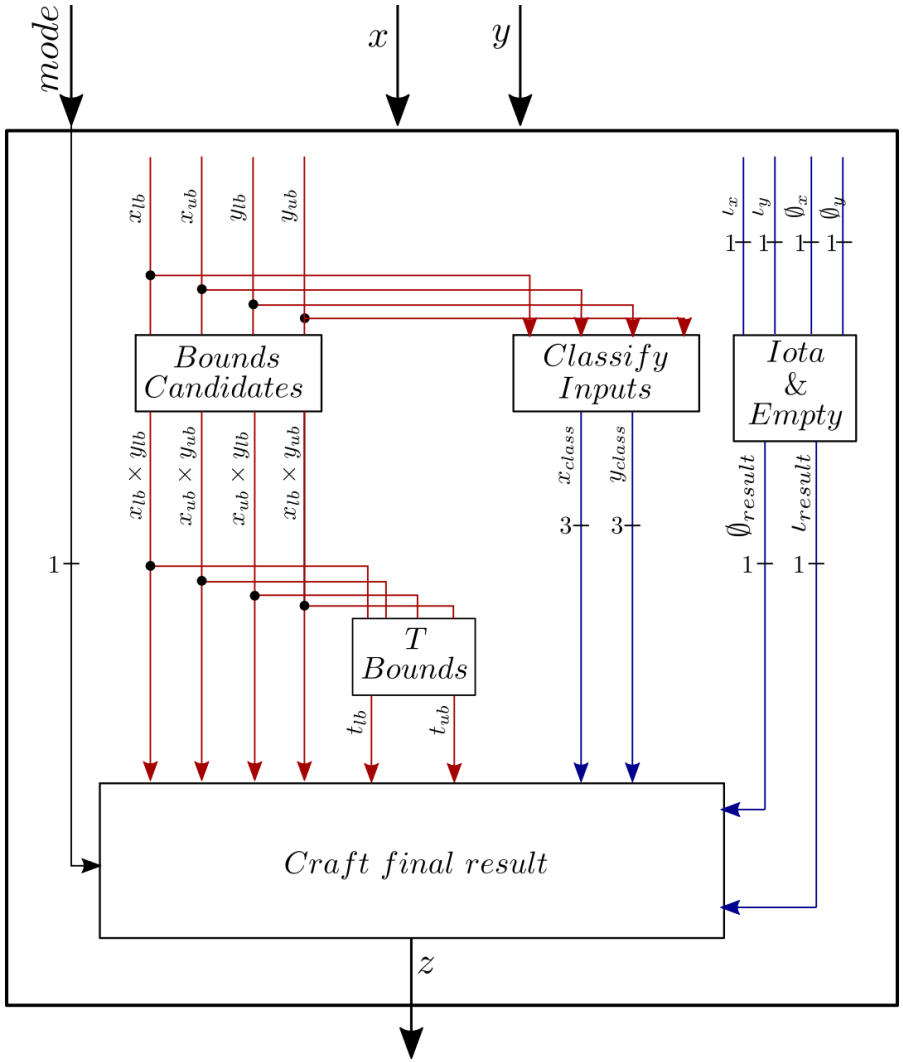
- The circuit looks like this:



- Benefits:
  - Bounds handled concurrently.
  - Different roundings at the same time.
  - No context switch.
  - Directly from and to registers.
  - Timing controlled precisely.

A Bit More Complex: Multiplication With **itv.mul**

- The circuit looks like this:



- **Bounds candidates**:
  - 4 multiplications in parallel (possible bounds).
- **Classify inputs**:
  - Bound selection criteria.
- **Craft result**:
  - Selected bounds re-rounding.
  - Package result.

Summary of the Results Achieved

- Numbers given for XC7Z020-1CLG400C @ 100MHz

	LUTs	Reg.	Muxes	DSP	BRAM	Cycles
ItvAdd	1334	880	0	0	0	7
ItvMul	1115	1002	7	8	0	8
ItvDiv	3399	2893	52	2	0	15
ItvSqrt	896	720	0	0	0	12
ItvExp	650	805	558	2	0	10
ItvLog	1279	1673	48	6	2	12
ItvSinCos	6105	5123	0	20	2	37
ItvAsinAcos	5320	4125	327	0	0	31

More information on this topic

Filiol and al. "Efficient Hardware Primitives for Interval Contractors in Robotics and Integration to a Custom RISC-V ISA Extension". Published in Newcas2025

Contents

1 Motivations

2 Fundamentals of RISC-V

3 Strategy Used in This Work

4 Contours of the Extension

5 Hardware Implementation

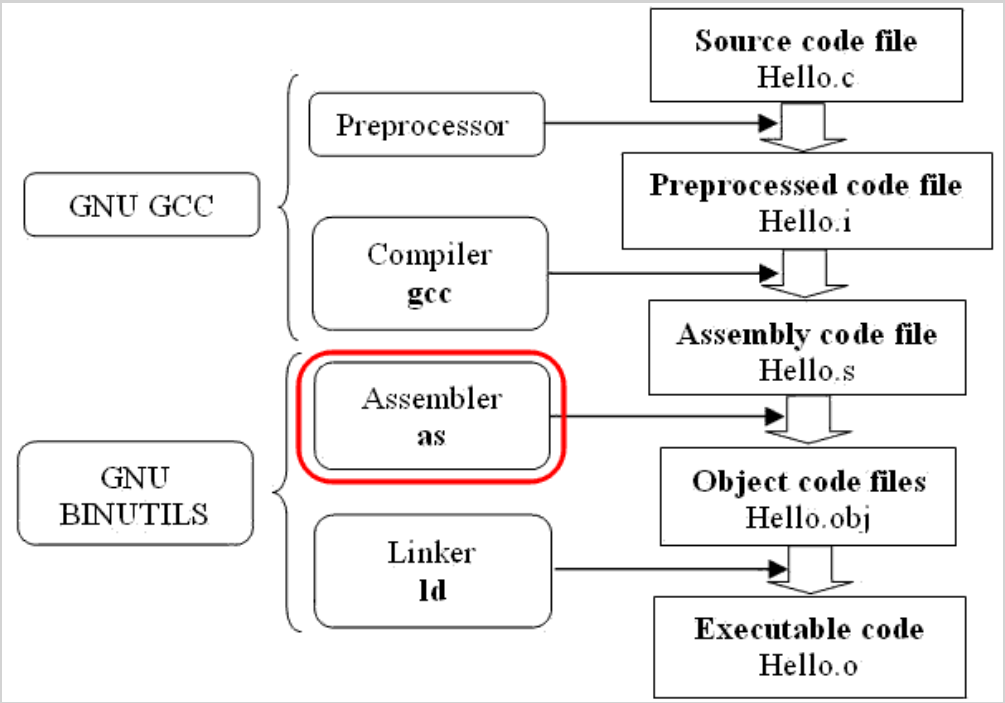
6 Toolchain Adaptation

7 Validation and Results

8 Conclusion

Adding compiler support for **xinterval**

Solution: Declare the instructions in the binutils layer



More information on this topic

Filiol and al. "Efficient Hardware Primitives for Interval Contractors in Robotics and Integration to a Custom RISC-V ISA Extension". Published in Newcas2025

Using the New **xinterval** Instructions From C

- Let's bind assembly instructions from C.

```c
/* interval as double (64 bits ) */
typedef itv_t double ;

/* Inlining of xinterval instruction itv.set.i */
inline itv_t __attribute__((always_inline))
itv_set_i(itv_t itv1, itv_t itv2) {
  itv_t result;
  asm("itv.set.i %0, %1, %2" : "=f"(result) : "f"(itv1), "f"(itv2));
  return result;
}

/* Inlining of xinterval instruction itv.sub */
inline itv_t __attribute__((always_inline))
itv_sub(itv_t itv1, itv_t itv2) {
  itv_t result;
  asm("itv.sub %0, %1, %2" : "=f"(result) : "f"(itv1), "f"(itv2));
  return result;
}
```

Listing: Addition backward contractor using *xinterval*

Implementing contractors

- Implementing accelerated contractors becomes easy.

Example: Backward addition

Suppose that we have the contrainst $\mathbf{x + y = z}$.
The associated backward contractor is:

$$\overleftarrow{\mathcal{C}_+} : [x'] = ([z] - [y]) \cap [x]$$

```
1 /* Backward ctc for x1 */
2 itv_t addbwctc1(itv_t x1, itv_t x2, itv_t x3) {
3     itv_t sub = itv_sub(x3, x2);
4     itv_t inter = itv_set_i(x1, sub);
5     return inter;
6 }
```

Listing: Addition backward contractor using *xinterval*

- **Other libraries can be easily ported to use xinterval!**

Contents

1 Motivations

2 Fundamentals of RISC-V

3 Strategy Used in This Work

4 Contours of the Extension

5 Hardware Implementation

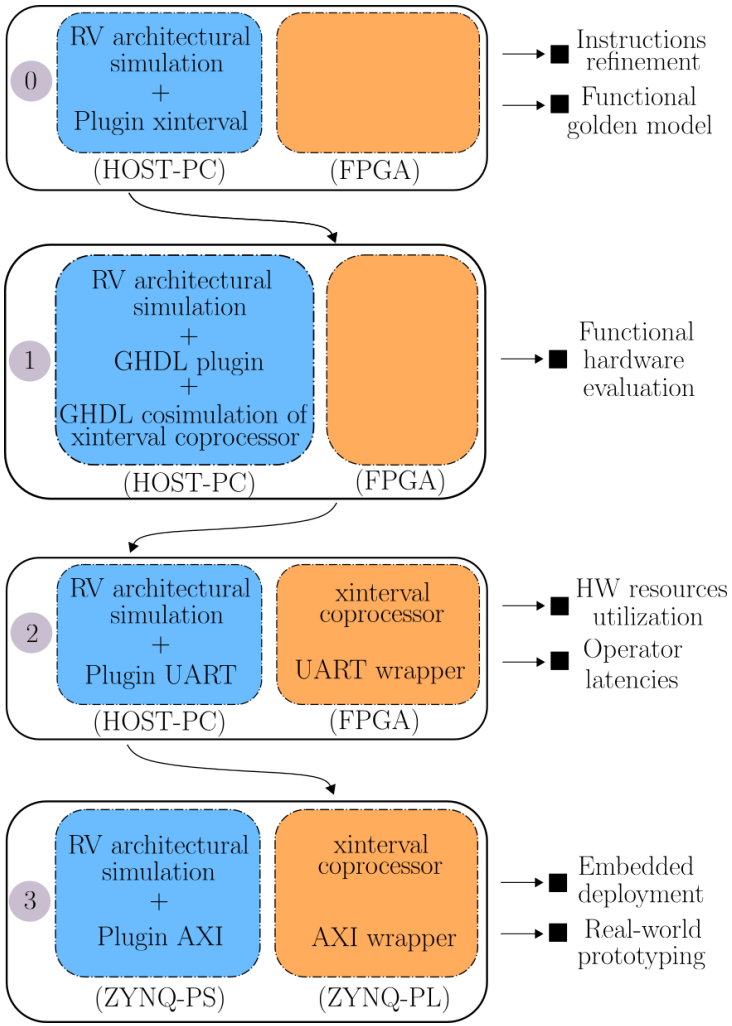
6 Toolchain Adaptation

7 Validation and Results

8 Conclusion

An Iterative Development and Validation Flow

- A strategy based on gradual replacement of software by hardware.



- **Hybrid approach**:
  - Instrumented emulation.
  - Hardware (**xinterval** core).
- **Comparison between**:
  - Interval analysis implemented in non-accelerated RISC-V.
  - Interval analysis implemented with **xinterval**.
- **Targeted at robotics applications**:
  - Capable of running contractors/SIVIA at each step of the design.

Reference Testbenches

Benchmark 1: **ring**

$$\mathcal{H}: \begin{cases} X: (x_1, x_2) \\ D: ([-10, 10], [-10, 10]) \\ C: \mathrm{f}(X) \in [16,\ 25] \end{cases} \quad \text{where } \mathrm{f}(X) = (x_1 - 1)^2 + (x_2 - 2)^2$$

Features: Addition, Subtraction, Square

Benchmark 2: **waves**

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

Features: Addition, Square, Exponential, Sine

Reference Testbenches

Benchmark 3: **donut**

$$\mathcal{H}: \begin{cases} X: (x_1, x_2) \\ D: ([-3,3],[-3,3]) \\ C: \mathrm{f}(X) \in [\text{-}0.2,\ 0.2] \end{cases} \quad \text{where } \mathrm{f}(X) = e^{x_1 \times x_2} - \sin(x_1 - x_2)$$

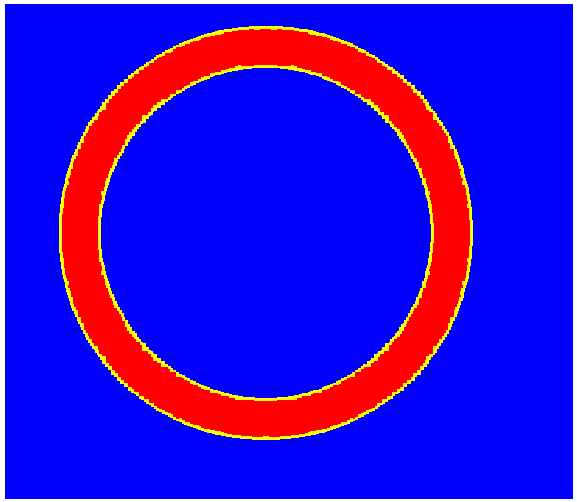
Features: Subtraction, Multiplication, Exponential, Sine

Benchmark 4: **vortex**

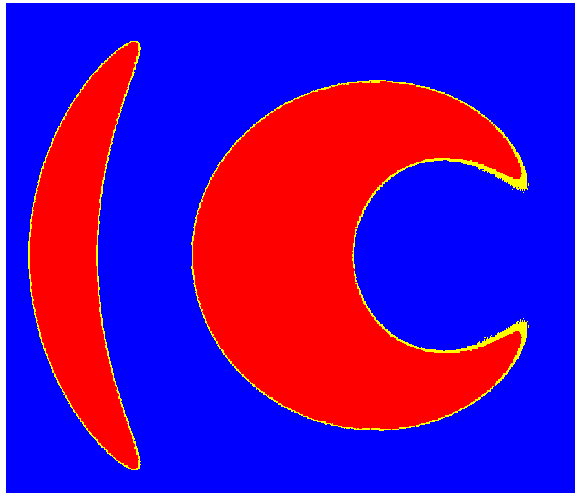
$$\mathcal{H}: \begin{cases} X: (x_1, x_2) \\ D: ([-3.5,2],[-3,3]) \\ C: \mathrm{f}(X) \in [0.325,\ +\infty] \end{cases} \quad \text{where } \mathrm{f}(X) = (x_2 - 5)\cos(4\sqrt{(x_1-4)^2} + {x_2}^2 - x_1 \sin(2\sqrt{{x_1}^2 + {x_2}^2})$$

Features: Addition, Subtraction, Multiplication, Square, Square-root, Sine, Cosine

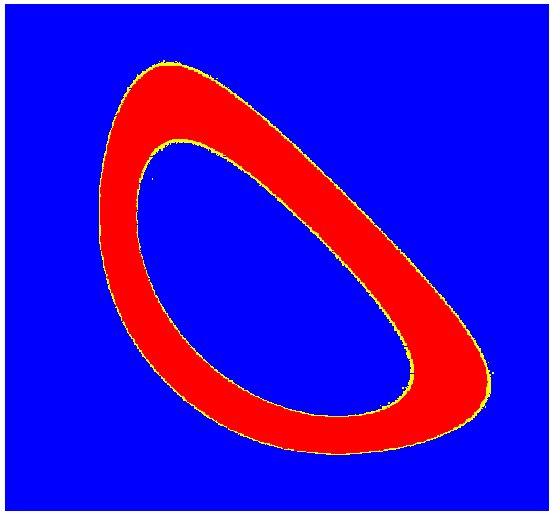
Running Everything on the FPGA



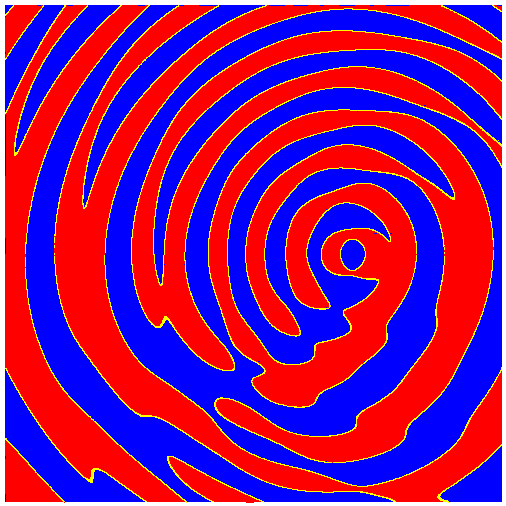
(a) ring



(b) waves



(c) ring



(d) waves

Performance Estimations

- We obtained the following preliminary results

Benchmark	Speedup *
ring	$\times 3.2$
waves	$\times 7.3$
donut	$\times 8.2$
vortex	$\times 7.8$

Table: Speed-up estimate relative to a RISC-V non xinterval-accelerated implementation.

Contents

1 Motivations

2 Fundamentals of RISC-V

3 Strategy Used in This Work

4 Contours of the Extension

5 Hardware Implementation

6 Toolchain Adaptation

7 Validation and Results

8 Conclusion

Conclusion

Some remarks about this study

- It works! (that's cool).
- Still a proof of concept.
- Much room for improvement (IEEE-754 operators, FPGA, ...).
- Low learning curve for a programmer.
- Allow porting of existing libraries.

Thank you!