Towards Optimized Arithmetic Circuits with MLIR

Louis Ledoux, Pierre Cochard, Florent de Dinechin
INSA Lyon, Inria, CITI, UR3720, 69621 Villeurbanne, France
{louis.ledoux, pierre.cochard, florent.de-dinechin}@insa-lyon.fr

***Abstract*—Numerical programs are typically conceived with real numbers in mind. However, programming languages operate at a lower abstraction level with fixed-width machine arithmetic. This abstraction gap limits the scope of legal arithmetic optimizations in compilers, in particular when targetting hardware.**

This work introduces a set of MLIR dialects that explicitly separate concerns between real-valued computation and low-level arithmetic representation. The RealArith dialect captures mathematical intent, enabling algebraic rewrites and approximation-aware transformations. The FixedPointArith dialect expresses quantized arithmetic with fine-grained control over bit widths. This separation enables arithmetic optimizations beyond those supported by conventional compilers. An example end-to-end lowering flow performs polynomial approximation, then generates fixed-point Horner-form architectures tailored for hardware synthesis. Early hardware results on signal processing benchmarks demonstrate the potential of this approach.

***Keywords*—MLIR, arithmetic optimization, fixed-point, polynomial approximation, high-level synthesis**

I. Introduction

Numerical programs are often written under the implicit assumption that operations behave like those over real numbers. However, modern compiler infrastructures, including MLIR, typically operate on machine-level formats such as fixed-width integers and floating-point numbers (IEEE754). These formats impose rigid evaluation semantics, limiting the set of legal arithmetic transformations.

For example, while addition is associative in real arithmetic, it is not in floating point. Similarly, expression fusion or algebraic rewrites that are mathematically valid may become unsafe or imprecise when executed with finite-precision types. These constraints hinder possible optimizations, particularly in domains like signal processing, scientific computing, and machine learning, where computations follow well-defined mathematical patterns such as matrix multiplication (GEMM), quantization, sparse accumulation, activation functions, or transcendental operations. These patterns are often amenable to algebraic simplification or approximation before being lowered to low-level arithmetic circuits, where further hardware-specific rewrites such as operator specialization and bitwidth tuning can be applied.

When compiling to hardware, such limitations can and should be relaxed. Hardware offers the freedom to implement arithmetic operators with arbitrary bit widths and optimized datapaths. Given the ability to tune precision and layout at the circuit level, designers can trade off accuracy, and area according to application needs. To fully exploit this flexibility, compiler flows must reason not only about bits, but also about the mathematical semantics of computation.

The Multi-Level Intermediate Representation (MLIR) [3] offers a promising foundation for building such flows. Originally developed to improve the compilation of machine-learning models, MLIR now also serves as a foundation for hardware-oriented compiler projects such as CIRCT[1] and Dynamatic[2]. Its extensible infrastructure enables modular modeling of programs across abstraction levels. In MLIR, these levels are described as *dialects*, each of which defines an Intermediate Representation (IR) with its own operations, types, and transformation rules. However, existing MLIR dialects remain tightly bound to machine arithmetic and lack a systematic way to express real-valued computation or reason about approximation.

This work introduces an arithmetic-aware MLIR flow that bridges high-level mathematical intent and hardware-oriented representation. Our contributions include two dialects with transformation passes and lowerings for arithmetic optimization and hardware generation:

- **RealArith** represents computations over real numbers, enabling semantic-preserving rewrites and symbolic approximation control.
- **FixedPointArith** expresses quantized arithmetic with precise control over fixed-point types and is designed to target hardware synthesis.

Building on these dialects, we implement a full lowering pipeline that transforms real-valued expressions into hardware-oriented fixed-point arithmetic. The pipeline performs approximation using external tools such as Sollya [1] and FloPoCo [2], generating Horner-form evaluators with precision-tuned datapaths. This enables the synthesis of optimized arithmetic accelerators directly from high-level mathematical IR. Our contributions are as follows:

(1) We design and implement the `RealArith` and `FixedPointArith` MLIR dialects to support multi-level arithmetic reasoning and transformation.
(2) We develop an approximation-aware lowering pipeline that translates real expressions into fixed-point arithmetic using Sollya and FloPoCo.
(3) We generate precision-tuned, pipelined Horner architectures, suitable for RTL and HLS-based synthesis.
(4) We demonstrate early hardware results on signal processing benchmarks, where our approach enables trade-offs between memory footprint and arithmetic complexity under an accuracy budget.

[1] https://circt.llvm.org/
[2] https://dynamatic.epfl.ch/

Manuscript received July 14, 2025; revised July 28, 2025; accepted August 13, 2025. Published September 2, 2025.
Issue category: Special Issue on DSD/SEAA 2025 on Works in Progress (WiP) Session, Salerno, Italy, Sept. 2025
Paper category: Short
DOI: doi.org/10.64552/wipiec.v11i1.90

II. Background

A. Multi-Level Intermediate Representation (MLIR)

MLIR provides an extensible infrastructure for building compiler pipelines with multiple abstraction levels. Its core abstraction, the *dialect*, allows different computational models to coexist, enabling progressive lowering from high-level semantics to hardware-ready representations. Projects such as CIRCT and Dynamatic extend MLIR for hardware synthesis, motivating our use of dialects to capture arithmetic intent across levels of precision and abstraction.

B. Functional Audio Stream (FAUST)

Faust [5] is a domain-specific language for real-time digital signal processing, making it well-suited to capture the associated mathematical intent. It adopts a functional programming model and supports multiple backends, including C++, LLVM IR, and hardware targets. To enable FPGA deployment, the Syfala toolchain [6] compiles Faust-generated C++ via Vitis HLS, producing hardware for real-time audio. More recently, an MLIR backend has been introduced, enabling lowering of Faust programs to hardware-oriented IR flows.

C. Floating-Point Cores (FloPoCo)

FloPoCo is an open-source tool for generating parameterized arithmetic cores, particularly optimized for FPGA targets. Internally, FloPoCo is structured around a clear separation of concerns between high-level arithmetic modeling, such as real-valued polynomial and piecewise approximations, low-level datapath construction, including components like bit heaps, and target-specific mapping to FPGA resources. While this philosophy guides its architecture, these layers are currently intertwined in implementation and not explicitly exposed, making them difficult to access or reuse from external tooling.

These internal abstractions align naturally with MLIR's dialect model. Our work seeks to expose each layer as an explicit dialect, making FloPoCo's arithmetic reasoning and circuit synthesis capabilities more accessible and interoperable within MLIR-based hardware flows.

D. Polynomial Approximations and Horner Architectures

Polynomial approximation is a classical technique that enables efficient evaluation of functions using only additions and multiplications. This topic has been well studied in the literature: textbooks [2], [4] detail both the mathematical foundations and implementation strategies, including range reduction and hardware-oriented considerations.

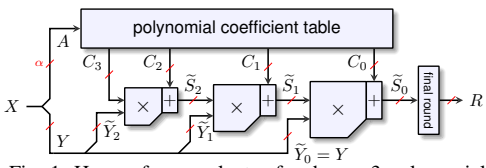


Fig. 1: Horner-form evaluator for degree-3 polynomials.

A univariate polynomial p of degree d over a real variable X has real-valued coefficients $C_i \in \mathbb{R}$. The Horner evaluation scheme is often used, since it involves only one multiplication per coefficient:

$$p(X) = C_0 + X(C_1 + \cdots + X(C_{d-1} + XC_d))). \quad (1)$$

Figure 1 represents a piecewise polynomial evaluator the fixed-point architecture for evaluating degree-3 polynomials using Horner's method. The input X is decomposed into two parts: the most significant α bits, denoted A, addresses a coefficient table holding 2^α polynomials. The remaining $w_X - \alpha$ bits, denoted Y, serve as the local offset for evaluation within the selected sub-interval.

In a fixed-point implementations (Figure 1), the smallest possible format of each coefficient C_i can be derived from the function and the accuracy constraints. Similarly, each Horner step may use a truncation $\widetilde{Y}_i$ of Y to minimize the size of the corresponding multiplier [2] – this is implemented in FloPoCo.

The segmentation parameter α introduces a trade-off: increasing α reduces the required polynomial degree and hence arithmetic cost, but exponentially increases memory usage due to the 2^α coefficient sets. This trade-off must be co-optimized based on implementation constraints. MLIR provides a suitable infrastructure to capture this trade-off explicitly, enabling lowering strategies or automated heuristics to tune the arithmetic / memory balance in the generated hardware.

III. End-to-End Compilation Flow

A. System Integration Overview

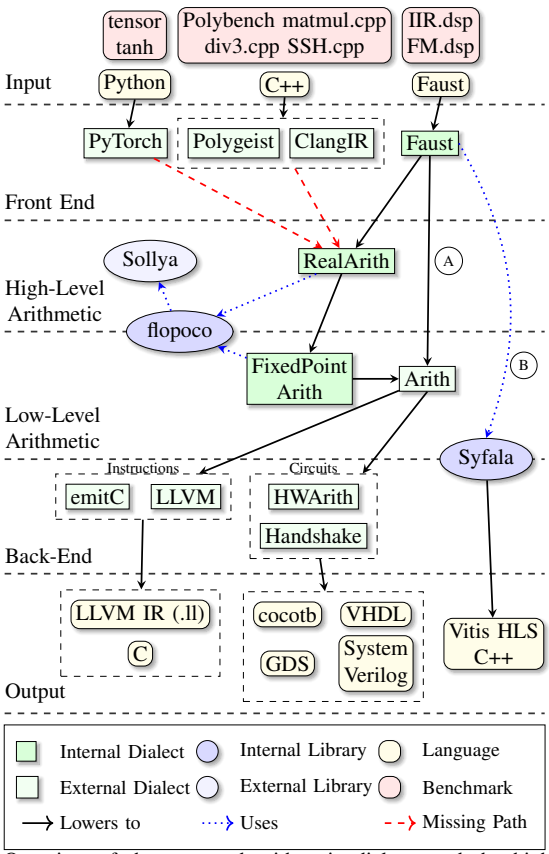


Fig. 2: Overview of the proposed arithmetic dialects and the high-level synthesis toolchain.

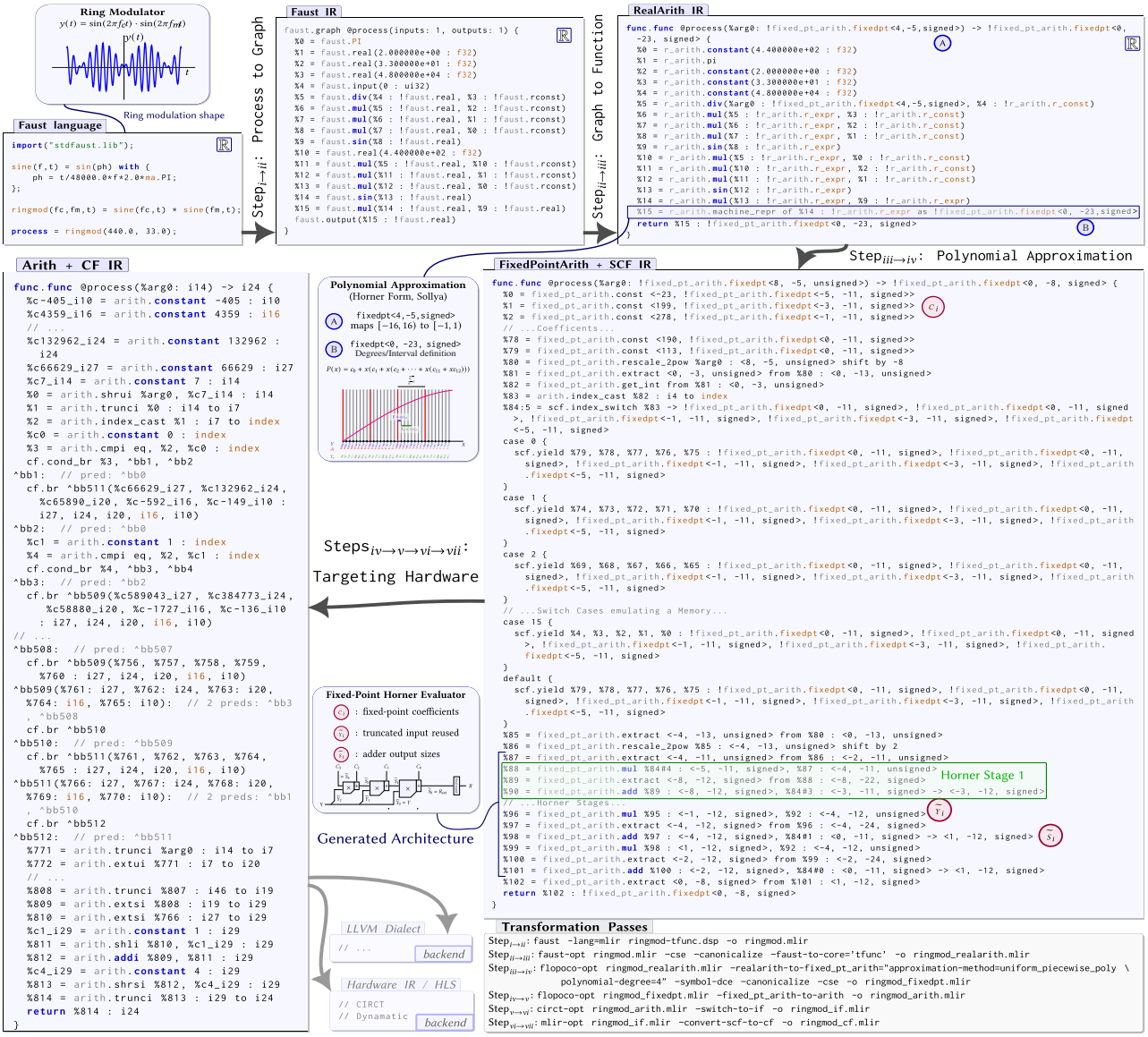


Fig. 3: From real-valued audio signals to synthesizable RTL: multi-intent arithmetic dialects and transformation passes.

Figure 2 illustrates the integration of our arithmetic-aware flow within a high-level synthesis toolchain. While the diagram emphasizes signal processing use cases, the flow is designed to accommodate a broader range of inputs, such as AI models or polyhedral C++ code. Figure 2-(A) and -(B) denote baseline that bypass the proposed optimizations.

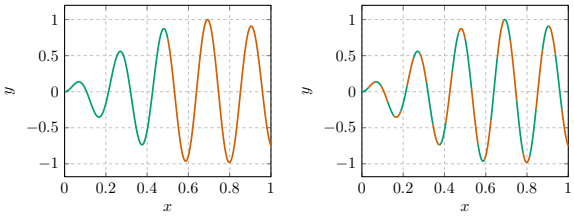
B. Multi-Level Arithmetic Dialects

We introduce two MLIR dialects that reflect distinct levels of arithmetic abstraction. The `RealArith` dialect operates over mathematical real numbers and supports symbolic expressions with both algebraic and transcendental operations. As illustrated by Figure 3-(B), it introduces the `machine_repr` operation, which defines the transition point from infinite-precision real arithmetic to a concrete fixed-point format suitable for implementation. The `FixedPointArith` dialect encodes quantized fixed-point arithmetic with operations. This dialect serves as an intermediate representation amenable to hardware synthesis and lowers directly to the core MLIR dialects (`Arith`).

C. Transformation and Pass Pipeline

Figure 3 illustrates the transformation pipeline across intermediate representations in our flow. The process begins from high-level real-valued expressions written in DSLs like Faust, which are then mapped to the `RealArith` dialect. Approximation is triggered by the insertion of a `machine_repr` operation, introduced by the `faust-opt` pass (Figure 3-(B)). This operation marks the boundary between real-valued computation and fixed-point implementation, as indicated by the absence of the $\mathbb{R}$ badge. The LSB of its return type determines the desired output precision.

This request is handled by a transformation pass that performs symbolic approximation using the Sollya library. The result is a fixed-point polynomial architecture expressed in Horner form. The corresponding evaluator is emitted with our `FixedPointArith` dialect, paired with `SCF` (Structured



(a) $\alpha = 1$, Degree=16 (b) $\alpha = 5$, Degree=4

Fig. 4: Uniform piecewise polynomial approximations.

Control Flow) Dialect to implement a case-based memory access over 2^α coefficient sets.

The generated architecture corresponds to the fixed-point Horner evaluator depicted in Figure 1. The fixed-point coefficients c_i are defined by the values yielded in each `scf.index_switch` case. Since each of the 2^α segments requires a polynomial of degree d, each case yields $d+1$ constants, totaling $(d+1)\cdot 2^\alpha$ coefficients. The truncated inputs $\widetilde{Y_i}$ are propagated through the stages, visible in the IR as shared `extract` and `rescale` operations. Adder output sizes $\widetilde{S_i}$ result from precision growth at each stage and correspond to the bitwidths seen in the intermediate addition results on lines `%88`–`%90`. Eventually, to target hardware synthesis tools, the IR is lowered to core dialects: `Arith`, `CF`, and `Builtin`. This includes scaling fixed-point into integers and converting control constructs into the `CF` dialect required by CIRCT or Dynamatic. Once in this form, downstream tools such as `dynamatic-opt` and `export-hdl` can be used to generate synthesizable HDL.

IV. Results Discussions

A. Experimental Setup and Methodology

We evaluate our compilation flow on a ring modulation algorithm expressed in Faust. The computation consists of the product of two sine waves, $y(t) = \sin(2\pi f_1 t)\cdot\sin(2\pi f_2 t)$, chosen for its relevance to real-time audio processing and the presence of a nonlinear transcendental function. Two baselines are considered (see two first rows of Table I). The first uses Syfala, which compiles Faust-generated C++ through Vitis HLS to produce synthesizable RTL (see Figure 2-Ⓑ). The second bypasses our proposed optimizations by lowering Faust-generated MLIR directly to the `arith` dialect in floating point, and applies the `mlir-opt -test-math-polynomial-approximation` pass to expand transcendental functions into `f32`-based polynomial approximations(see Figure 2-Ⓐ). Figure 4 shows two of our polynomial approximation configurations after range reduction to $[0,1)$ of a subset of a full period. The visible alternation of segments reflects the piecewise scheme. The degree-16 case needs to store $2^1\cdot(16+1) = 34$ coefficients, likely wider than the $2^5\cdot(4+1) = 160$ narrower ones of the degree-4 case.

B. Hardware Results

Table I reports hardware usage across methods on the xc7z020-1clg400c FPGA. DSP usage increases with polynomial degree due to deeper pipelines and wider coefficients, which result in more arithmetic stages and facilitate automated DSP inference. Lower-degree piecewise configurations trade arithmetic for memory by increasing the number of segments. Both 32-bit floating-point baselines show high DSP usage due to mantissas being mapped to dedicated multipliers.

TABLE I: Resource usage across methods on xc7z020 FPGA.

Method (output precision)	Poly. degree	Hardware resources		
		LUT	FF	DSP
faust-syfala (32 bits)	*	3,765	3,142	32
faust-mlir (32 bits)	*	14,638	8,226	27
Uniform Piecewise Poly. Approx. (10 bits)	4	2,868	4,172	4
	5	3,209	4,460	5
	14	1,252	1,157	50
	18	1,531	1,369	63
Uniform Piecewise Poly. Approx. (24 bits)	3	127,441	209,121	5
	4	32,042	56,004	8
	5	17,608	30,399	11
	6	9,712	16,489	15
	8	4,965	7,886	24
	9	5,228	8,229	27

** Not applicable in this case.*

While these early results demonstrate the feasibility of our flow, we note that baseline paths required manual construction due to gaps in existing MLIR hardware lowering support. A detailed and systematic evaluation of baseline strategies, as well as more precise comparisons across numeric formats, will be the subject of future work.

V. Conclusions and Future Work

This work presents a multi-level arithmetic-aware MLIR flow that connects high-level mathematical semantics to low-level hardware representations in an end-to-end pipeline.

Preliminary results on a signal processing benchmark show promising trade-offs between memory footprint and arithmetic complexity. However, the current state of end-to-end MLIR hardware support poses challenges for establishing robust baselines. With the dialect and lowering infrastructure now in place, future work will focus on introducing hardware-level optimizations – such as bitheap-based arithmetic synthesis [2, ch. 7] – as well as supporting a wider range of approximation schemes, including table-based methods and non-uniform segmentation strategies. We also plan to extend evaluation to larger workloads in signal processing, linear algebra, and AI, where the benefits of semantic-aware arithmetic compilation are expected to be more pronounced.

References

[1] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *International Congress on Mathematical Software*, vol. 6327. Heidelberg, Germany: Springer, September 2010, pp. 28–31.

[2] F. de Dinechin and M. Kumm, *Application-Specific Arithmetic*. Springer, 2024.

[3] C. Lattner *et al.*, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[4] J.-M. Muller, *Elementary functions, algorithms and implementation, 3rd Edition*. Birkhaüser Boston, 2016.

[5] Y. Orlarey, D. Fober, and S. Letz, "FAUST : an Efficient Functional Approach to DSP Programming," in *NEW COMPUTATIONAL PARADIGMS FOR COMPUTER MUSIC*, E. D. FRANCE, Ed., 2009, pp. 65–96.

[6] M. Popoff, "Audio DSP to FPGA Compilation: The Syfala Toolchain Approach," Ph.D. dissertation, Univ Lyon, INSA Lyon, Inria, May 2023.