Towards Multi-Level Arithmetic Optimizations

The Real Number Illusion

- Most numerical code is written with **real numbers** in mind.
- Yet, compilers only obey low-level **machine numbers**: floats and ints.
- This discourages ("unsafe-math") or misses many **optimization opportunities**, especially in **machine learning**, **linear algebra**, or **signal processing**.

Arithmetic Optimizations

... beyond existing low-level rewrites in current compilers:

- Operator specialization: squarers, constant multipliers
- Expression fusion: $\frac{1}{\sqrt{x^2+y^2}}$ or $\sin(\omega t + \phi)$
- Optimizations tailored to target semantics

Useful when compiling to hardware [Lah+18; Ugu+20; For+22; DK24], but not only.

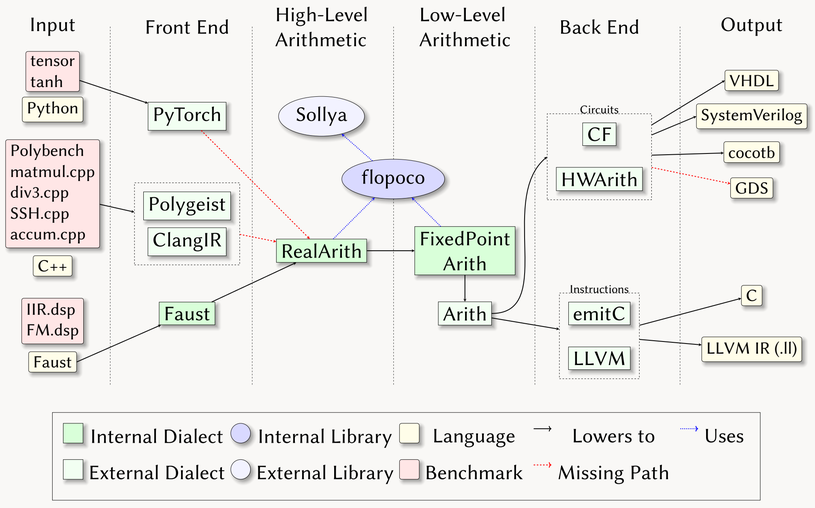
Semantics First, Bits Later

Separation of concerns thanks to MLIR:

- Higher levels capture "mathematical intent",
- Lower levels deal with "machine numbers".

Our **contributions**:

- `real_arith` and `fixed_pt_arith` dialects
- Polynomial approximation lowering from real expressions
- Precision-tuned Horner architecture derived from dialect-level ops
- End-to-end MLIR flow evaluated on signal processing workloads

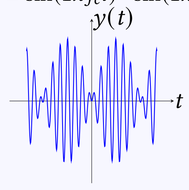


Overall contribution: proposed dialects (in dark green), integrated in a High-Level Synthesis ecosystem.

Example: End-to-End Audio compilation flow

Ring Modulator

$y(t) = \sin(2\pi f_c t) \cdot \sin(2\pi f_m t)$

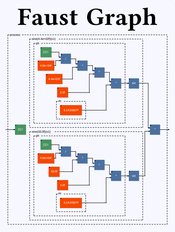


Faust language

```
import("stdfaust.lib");

sine(f,t) = sin(ph) with {
    ph = t/48000.0*f*2.0*ma.PI;
};

ringmod(fc,fm,t) = sine(fc,t) * sine(fm,t);

process = ringmod(440.0, 33.0);
```

Ring modulation shape

Faust Graph



Internal Graph

Step$_{i \to ii}$: Process to Graph

Faust IR

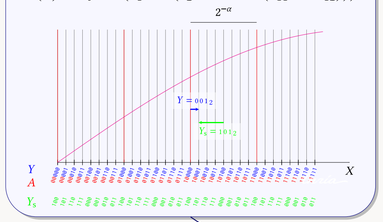
```
faust.graph @process(inputs: 1, outputs: 1) {
  %0 = faust.PI
  %1 = faust.real(2.000000e+00 : f32)
  %2 = faust.real(3.300000e+01 : f32)
  %3 = faust.real(4.800000e+04 : f32)
  %4 = faust.input(0 : ui32)
  %5 = faust.div(%4 : !faust.real, %3 : !faust.rconst)
  %6 = faust.mul(%5 : !faust.real, %2 : !faust.rconst)
  %7 = faust.mul(%6 : !faust.real, %1 : !faust.rconst)
  %8 = faust.mul(%7 : !faust.real, %0 : !faust.rconst)
  %9 = faust.sin(%8 : !faust.real)
  %10 = faust.real(4.400000e+02 : f32)
  %11 = faust.mul(%5 : !faust.real, %10 : !faust.rconst)
  %12 = faust.mul(%11 : !faust.real, %1 : !faust.rconst)
  %13 = faust.mul(%12 : !faust.real, %0 : !faust.rconst)
  %14 = faust.sin(%13 : !faust.real)
  %15 = faust.mul(%14 : !faust.real, %9 : !faust.real)
  faust.output(%15 : !faust.real)
}
```

Step$_{ii \to iii}$: Graph to Function

Polynomial Approximation
(Horner Form, Sollya)

- A: `fixedpt<4,-5,signed>` → maps $[-16, 16)$ to $[-1, 1)$
- B: `fixedpt<0, -23, signed>` → 12th-degree polynomial

$P(x) = c_0 + x(c_1 + x(c_2 + \cdots + x(c_{11} + xc_{12})))$



Polynomial Transformation

RealArith IR

```
func.func @process(%arg0: !fixed_pt_arith.fixedpt<4,-5,signed>) -> !fixed_pt_arith.fixedpt<0,
  -23, signed> {
  %0 = r_arith.constant(4.400000e+02 : f32)
  %1 = r_arith.pi
  %2 = r_arith.constant(2.000000e+00 : f32)
  %3 = r_arith.constant(3.300000e+01 : f32)
  %4 = r_arith.constant(4.800000e+04 : f32)
  %5 = r_arith.div(%arg0 : !fixed_pt_arith.fixedpt<4,-5,signed>, %4 : !r_arith.r_const)
  %6 = r_arith.mul(%5 : !r_arith.r_expr, %3 : !r_arith.r_const)
  %7 = r_arith.mul(%6 : !r_arith.r_expr, %2 : !r_arith.r_const)
  %8 = r_arith.mul(%7 : !r_arith.r_expr, %1 : !r_arith.r_const)
  %9 = r_arith.sin(%8 : !r_arith.r_expr)
  %10 = r_arith.mul(%5 : !r_arith.r_expr, %0 : !r_arith.r_const)
  %11 = r_arith.mul(%10 : !r_arith.r_expr, %2 : !r_arith.r_const)
  %12 = r_arith.mul(%11 : !r_arith.r_expr, %1 : !r_arith.r_const)
  %13 = r_arith.sin(%12 : !r_arith.r_expr)
  %14 = r_arith.mul(%13 : !r_arith.r_expr, %9 : !r_arith.r_expr)
  %15 = r_arith.machine_repr of %14 : !r_arith.r_expr as !fixed_pt_arith.fixedpt<0, -23,
    signed>
  return %15 : !fixed_pt_arith.fixedpt<0, -23, signed>
}
```

(A) (B)

Step$_{iii \to iv}$: Polynomial Approximation

FixedPointArith IR

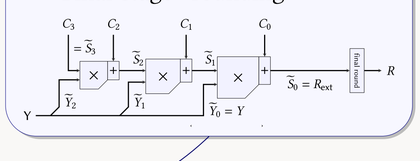
```
func.func @process(%arg0: !fixed_pt_arith.fixedpt<4, -5, signed>) -> !fixed_pt_arith.fixedpt
  <23, 0, signed> {
  %0 = fixed_pt_arith.const <2, !fixed_pt_arith.fixedpt<0, -2, signed>>
  %1 = fixed_pt_arith.const <-4170, !fixed_pt_arith.fixedpt<11, -2, signed>>
  %2 = fixed_pt_arith.const <2132970, !fixed_pt_arith.fixedpt<20, -2, signed>>
  %3 = fixed_pt_arith.const <1183, !fixed_pt_arith.fixedpt<9, -2, signed>>
  %4 = fixed_pt_arith.const <-303000, !fixed_pt_arith.fixedpt<17, -2, signed>>
  %5 = fixed_pt_arith.const <-70, !fixed_pt_arith.fixedpt<5, -2, signed>>
  %6 = fixed_pt_arith.const <13004, !fixed_pt_arith.fixedpt<12, -2, signed>>
  %7 = fixed_pt_arith.const <-3, !fixed_pt_arith.fixedpt<0, -2, signed>>
  %8 = fixed_pt_arith.const <-261, !fixed_pt_arith.fixedpt<7, -2, signed>>
  %9 = fixed_pt_arith.rescale_2pow %arg0 : <4, -5, signed> shift by -4
  %10 = fixed_pt_arith.mul %8 : <7, -2, signed>, %9 : <0, -9, signed>
  %11 = fixed_pt_arith.extract <8, -4, signed> from %10 : <8, -11, signed>
  %12 = fixed_pt_arith.add %11 : <8, -4, signed>, %7 : <0, -2, signed> -> <7, -4, signed>
  // ...
  // Intermediate Horner stages
  // ...
  %31 = fixed_pt_arith.mul %30 : <19, -4, signed>, %9 : <0, -9, signed>
  %32 = fixed_pt_arith.extract <20, -4, signed> from %31 : <20, -13, signed>
  %33 = fixed_pt_arith.add %32 : <20, -4, signed>, %0 : <0, -2, signed> -> <20, -4, signed>
  %34 = fixed_pt_arith.extend %33 : <20, -4, signed> to <23, -4, signed>
  %35 = fixed_pt_arith.extract <23, 0, signed> from %34 : <23, -4, signed>
  return %35 : !fixed_pt_arith.fixedpt<23, 0, signed>
}
```

C_i $\widetilde{Y}_i$ $\widetilde{S}_i$

Horner Stage 1

Fixed-Point Horner Evaluator

- C_i: fixed-point coefficients
- $\widetilde{Y}_i$: truncated input reused
- $\widetilde{S}_i$: adder output sizes
- Final stage: rounding



Generated Architecture

Step$_{iv \to v}$: Fixed-Point scaling

Step$_{v \to vi}$: Instructions

Step$_{v \to vii}$: Datapaths

LLVM Dialect

```
// ...
```
backend

Hardware IR / HLS

```
// CIRCT
// Dynamatic
```
backend

Arith IR (simplified)

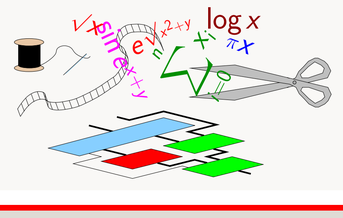
```
func.func @process(%arg0: i10) -> i24 {
  %c2_i3 = arith.constant 2 : i3
  %c-4170_i14 = arith.constant -4170 : i14
  // ...
  %0 = arith.extsi %arg0 : i10 to i20
  %1 = arith.muli %0, %c-261_i20 : i20
  %c7_i20 = arith.constant 7 : i20
  %2 = arith.shrsi %1, %c7_i20 : i20
  %3 = arith.trunci %2 : i20 to i13
  %4 = arith.extsi %3 : i13 to i14
  // ...
  %65 = arith.trunci %64 : i28 to i24
  return %65 : i24
}
```

Transformation Passes

Step$_{i \to ii}$: `faust -lang=mlir ringmod-tfunc.dsp -o ringmod.mlir`
Step$_{ii \to iii}$: `faust-opt ringmod.mlir --cse --canonicalize --faust-to-core='tfunc' -o ringmod_realarith.mlir`
Step$_{iii \to iv}$: `flopoco-opt ringmod_realarith.mlir --realarith-to-fixed_pt_arith -o ringmod_fixedpt.mlir`
Step$_{iv \to v}$: `flopoco-opt ringmod_fixedpt.mlir --realarith-to-fixed_pt_arith -o ringmod_arith.mlir`
Step$_{v \to vi}$: `flopoco-opt ringmod_arith.mlir --arith-to-llvmir -o ringmod_llvm.mlir`
Step$_{v \to vii}$: `flopoco-opt ringmod_arith.mlir --arith-to-hw -o ringmod_hw.mlir`

Open-Source Tools

- **Sollya** [SMC10]
- **ScaleHLS** [Ye+22]
- **Dynamatic** [Che+22], `https://dynamatic.epfl.ch/`
- **SODA-OPT** [Ago+22], `https://github.com/pnnl/soda-opt`
- **CIRCT** `https://circt.llvm.org/`
- **Faust** `https://faust.grame.fr`
- **FloPoCo** `https://www.flopoco.org`



Bibliography

[Ago+22] N. B. Agostini et al. "An MLIR-based compiler flow for system-level design and hardware acceleration". In: *41st IEEE/ACM International Conference on Computer-Aided Design*. 2022, pp. 1–9.

[Che+22] J. Cheng, L. Josipović, G. A. Constantinides, and J. Wickerson. "Dynamic Inter-Block Scheduling for HLS". In: *Field Programmable Logic and Applications*. 2022.

[DK24] F. de Dinechin and M. Kumm. *Application-Specific Arithmetic*. Springer, 2024.

[For+22] L. Forget, G. Harnisch, R. Keryell, and F. De Dinechin. "A single-source C++20 HLS flow for function evaluation on FPGA and beyond.". In: *Highly Efficient Accelerators and Reconfigurable Technologies*. ACM, 2022.

[Lah+18] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen. "Are we there yet? A study on the state of High-Level Synthesis". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (2018), pp. 898–911.

[SMC10] S. Chevillard, M. Joldeş, and C. Lauter. "Sollya: An Environment for the Development of Numerical Codes". In: *International Congress on Mathematical Software*. 2010.

[Ugu+20] Y. Uguen, F. de Dinechin, V. Lezaud, and S. Derrien. "Application-Specific Arithmetic in High-Level Synthesis Tools". In: *Transactions on Architecture and Code Optimization* 17.1 (2020).

[Ye+22] H. Ye et al. "ScaleHLS: A new scalable High-Level Synthesis framework on Multi-Level Intermediate Representation". In: *High Performance Computer Architecture*. 2022.

Pierre Cochard[1], Florent de Dinechin[1], Luc Forget, Louis Ledoux[1]

`pierre.cochard@insa-lyon.fr, florent.de-dinechin@insa-lyon.fr, dev@alias.lforget.fr, louis.ledoux@insa-lyon.fr`

[1]INSA Lyon, Inria