Generating Hardware Block Floating-Point implementations with Metalibm Lugdunum

Nicolas Brunie,

Metalibm Workshop, March 12th 2018

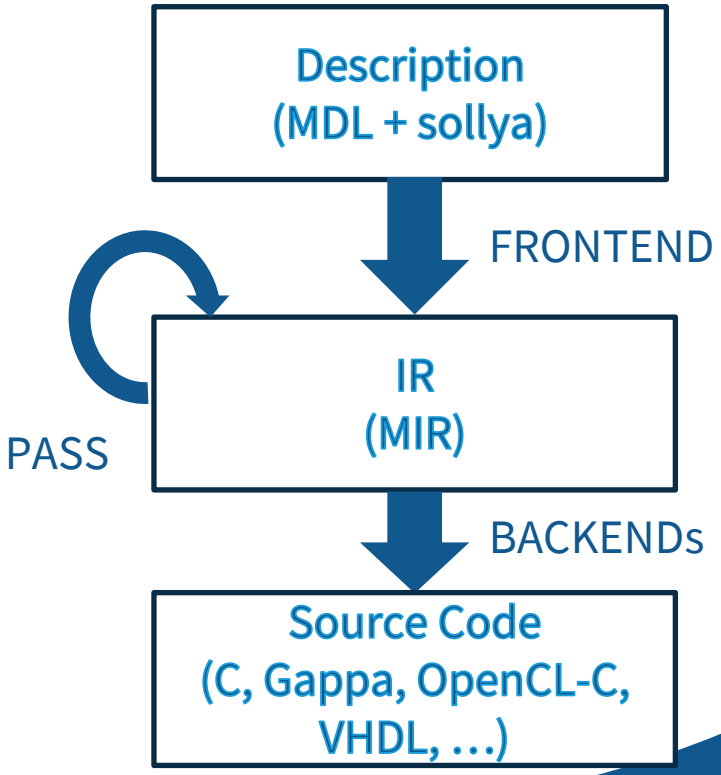
www.kalrayinc.com

Outline

1. Metalibm (Lugdunum) in general
2. RTL generation in Metalibm
3. Fixed-Point description
4. Fixed-Point middle-end
5. Demonstration
6. THE big announcement
7. Conclusion

Metalibm Generation Scheme

- Set of tools for the generation, validation, profiling of arithmetic kernel code
- Metalibm Description Language
  - Rich and Verbose
  - Based on pythonsollya for numerics
- Intermediate Representation processing
  - Validation, Optimization, Vectorization
  - Easy to manipulate, extend
- TestBench (validation and profiling) generation
  - Debug messages
  - Numeric emulation and checking
  - Benchmarking
- Backends for Code generation



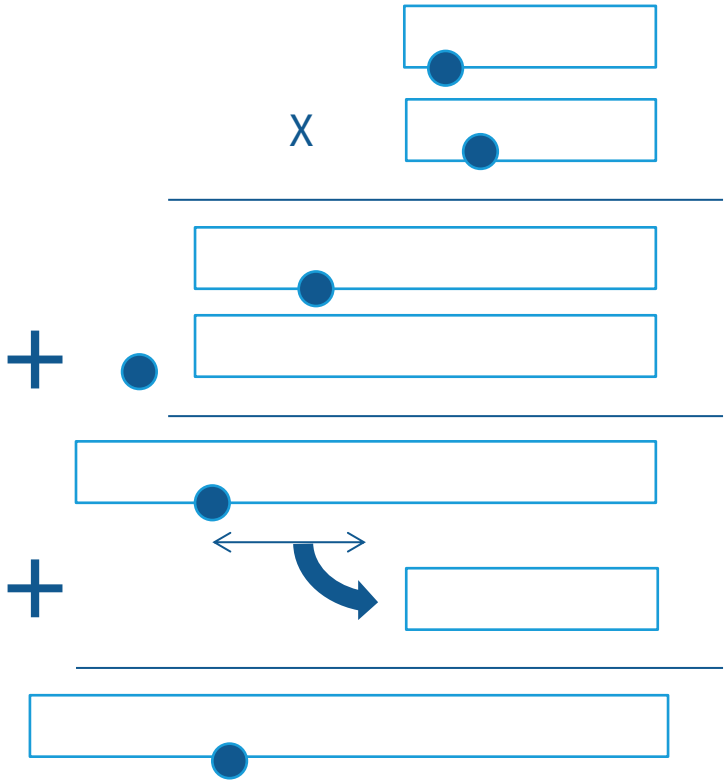
Demo

RTL generation capabilities

- Introducing Entity object
- MDL extended: New Meta-operations
  - Signal, Process, …
- VHDL Backend
- TestBench Generation Capabilities
  - Wrapping entity
  - Simulation ready
- Debug capabilities
  - Generate script to extract internal signals from entities
- Intermediate processing passes
  - Legalization
  - Pipelining

Fixed-Point operator description

- Extra fixed-point formats
  - Virtual format, reliying on physical support format
- FixedPointPosition meta-operation
  - Abstract position relative to the final point position
  - Can be injected into operation flow
  - Resolved during code generation
  - Multiple Specifiers:
    - FromLSBToLSB, FromMSBToLSB, FromPointToLSB, FromPointToMSB
- Easy to build operator by combining multiple sub-operators (e.g: dot-products)



Fixed-Point middle-end

- Legalize operations
  - Resolve FixedPointPosition value
  - Min-Max legalization
  - Determine a format for each undefined node
    - Enforcing “no overflow” rule
    - Limiting width to the minimal required
    - Introducing conversions when required
- Optimizations
  - Constant propagation
  - Pipelining
  - …

Demo

THE Big Announcement !

- Kalray's Metalibm is now **open-source**
- Available on **https://github.com/kalray/metalibm**

  Feel free to try it out, fork it and contribute back
- Thanks to all involved: **Florent**, **Hugues**, **Marc**, **Julien**, **GuillaumeS**, …

Conclusion

- Set of tools to build RTL operators
  - Implementation description
  - Code generation
  - Validation and Testing
- Future works
  - Abstract MDL from VHDL concepts
  - Extend fixed-point optimization
  - Push pass architecture to software side

Last year in Metalibm, 1/2

- Extending **x86 Backend** (SSEs, AVX, AVX2 …) thanks to Hugues
  - Clean vector format definition (SSE / AVX based on Virtual vector format)
  - New optimization pass for vector promotion (based on implementation availability)
    - Goal: avoid converting back and forth between standard and machine specific registers
    - Instantiated for x86 (m128, m256) and Kalray (wp)
  - New optimization: Select Compare to Comparison translation (x86)
- Adding support for **multi-word arithmetic**
  - Meta-blocks to simplify support of double double, triple double single single
  - Meta-block for Leading Zero Count (e.g. useful for vector implementation)
- Wrapped with (python)**CGPE** , thanks to Hugues
- Adding internal **benchmarking** capabilities (based on ReadTimeStamp node)
- **Sub-vector support**, thanks to Guillaume G.
- **Meta-functions**: expm1, new log, rsqrt, atan, tanh, sinh
- Cleaning **non-regression tests**
- Metalibm is now compatible with **python3**

Last year in metalibm, 2/2

Adding VHDL code generation capability

- ML_Entity
- TestBench generation
- Pipelining
- Clean pass-based optimization pipeline
- Fixed-Point format and optimization pass (legalizer, datapath sizing)
- FixedPointPosition placeholder

New test generation mechanism

- IEEE-754 arithmetic (special values)
- Weighted probability for directed random generation

On going:

- Refactor software pipeline: dynamic organization around passes
- Automatic micro-bench generation for micro-architectural profiling and target annotation
- Code cleaning (enforcing PEP8, linter in integration, authorized to fail (pylint **-15.77 / 10** …)