Hardware.jl – An MLIR-based Julia HLS Flow

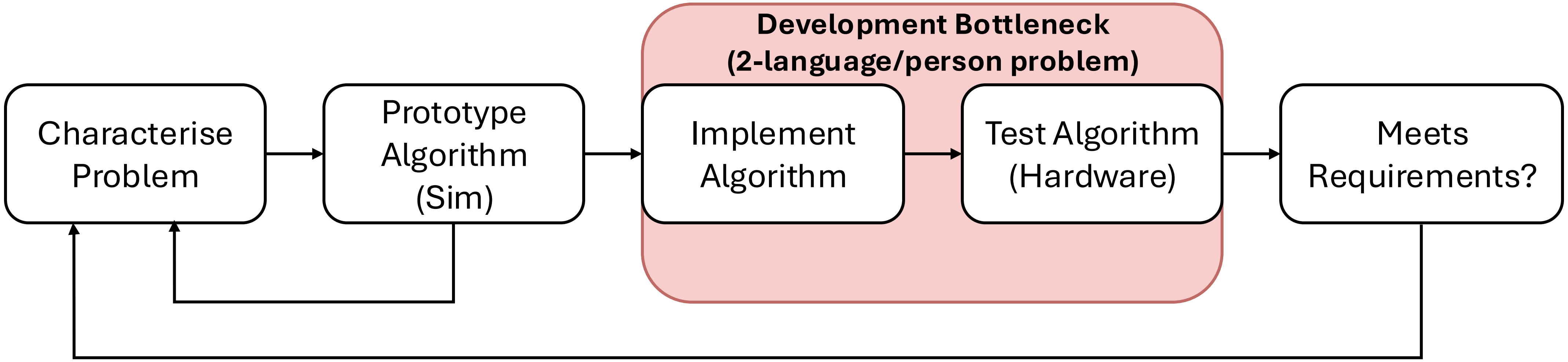
Benedict Short, Ian McInerney, John Wickerson

Imperial College London

Motivation

The Engineer's Workflow

- Most electrical, mechanical and aeronautical engineers follow a systematic workflow:



- The bottleneck is becoming more problematic as algorithms grow in complexity
- e.g. Real-time control systems in electric motors offload algorithms to FPGAs and ASICs

Why are the current solutions insufficient?

Existing tools fall into one of two categories:

1. Traditional tools, e.g. Vitis:
   - Abstraction level too low for engineers
   - Doesn’t eliminate the two-language problem

2. High-level alternatives:
   - MATLAB HLS: slow, unoptimized designs
   - Julia HLS tools: limited functionality/optimisations, rely on deprecated toolchains with fragile infrastructure

Why use Julia for design-entry?

- Popular within the scientific computing community
    - Maths friendly
    - Dynamically Typed
    - Open Source

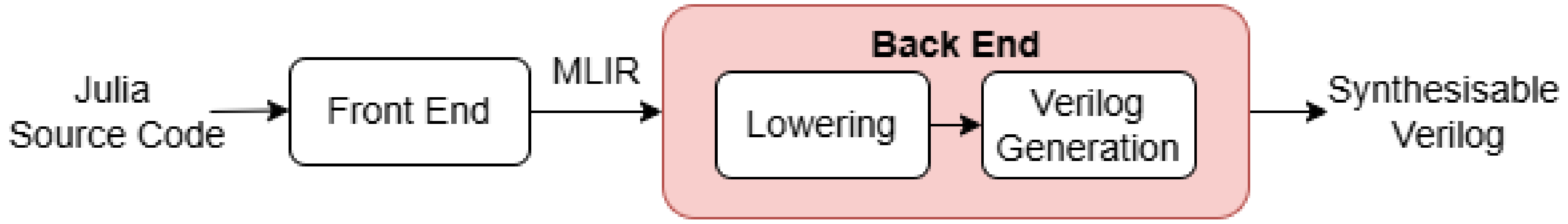
- Has a ‘Pure Julia’ ecosystem – easy to synthesise!
    - Packages like SciML would allow us to synthesise ODE solvers and optimisation algorithms directly into accelerators

- HLS friendly:
    - Native support for high-level abstract types, e.g. AbstractArray
    - Extract intrinsics directly to preserving high-level information, e.g. inherent parallelism during matrix operations

Our Solution: Hardware.jl

High-Level Aims & Overview

- Directly synthesise hardware from pure Julia source code
- Facilitate quick end-to-end accelerator design
- Reusable and easy to maintain (key priority)
- Take advantage of Julia’s extensibility

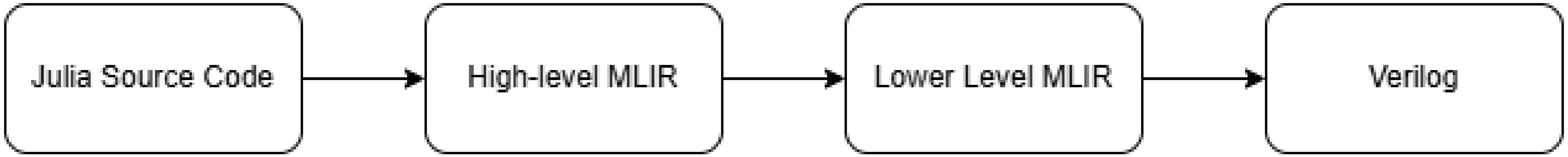


Why use MLIR?

- Dialects capture high-level semantics directly
  - Avoid emulation and premature lowering!
  - e.g. map AbstractArrays to ML dialects for first-class linear algebra support
- Allows for innovation at new levels of abstraction
  - Don’t reinvent the wheel
  - Inherit powerful analyses and robust infrastructure
- Stable IR
  - Julia actively upgrades LLVM, but existing HLS tools (e.g. AMD Vitis) are built on old versions
  - MLIR is significantly more stable between versions, making the tool easier to maintain

How do we extract program information?

- Two main types of program information to extract:
    1. Low-level SSA-type IR designed to run on a fetch-execute processor (emulation)
    2. High-level intrinsics for hardware specific optimisations (e.g. matrices)


$$\begin{tikzcd}\boxed{\text{Julia Source Code}} \arrow[r] & \boxed{\text{High-level MLIR}} \arrow[r] & \boxed{\text{Lower Level MLIR}} \arrow[r] & \boxed{\text{Verilog}}\end{tikzcd}$$

How do we generate Hardware from MLIR?

- CIRCT library:
  - Keeps the pipeline in MLIR
  - Permissively licensed and open-source
  - Abstracts away difficult problems that have been solved, e.g. generating valid Verilog for different EDA tools

- Custom HLS Backend:
  - Currently written in C++, long-term goal to move the tool to Julia
  - Chains together passes to lower into Verilog
  - Different lowering routes to generate different types of hardware

But Julia is Dynamically Typed…

- Julia implements advanced type-inference passes that aim to maximise type-stability to improve run-time performance
  - Re-use Julia's robust and performant infrastructure
  - e.g. Union splitting turns dynamic dispatch into static dispatch
- We place restrictions on the source code, in line with those from the JuliaC AOT Compiler
  - Anticipate that users will have evaluated the feasibility of the AOT compiler beforehand
- Enforce static typing on an IR Code level
  - Avoids exponentially increasing designs

Evaluation and Long-Term Vision

Metrics and Benchmarks

- Metrics:
  - Resource usage (design efficiency)
  - Design Latency
  - ‘Synthesisability’ of arbitrary programs

- Benchmarks:
  - Evaluate on an AMD Xilinx platform
  - Evaluate against existing HLS tools, the AOT and JIT Julia Compiler
  - Linear algebra/mathematical programs (e.g. PID controller)

Short- & Long-term Goals:

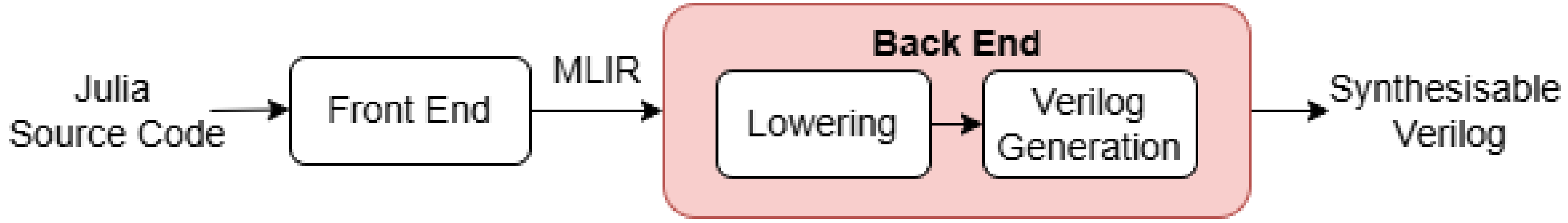
Short-Term:

Front end:
- Standard Control-Flow
- Floating Point and Integer Operations
- Matrix Operations

Back end:
- Matrix Operations via TOSA and scalability
- Support for Fixed Point operations
- Dynamic vs Static Scheduling

Long-Term:

- Implement the Back End in Julia and leverage access to numerical libraries
- Co-simulation against a Julia model
- Automatically generate Bindings for Compiled Julia (full black-box accelerator design)


Julia Source Code
Front End
MLIR
Back End
Lowering
Verilog Generation
Synthesisable Verilog