MLIR as Hardware Compiler Infrastructure

Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, Andrew Young, Richard Xia

2021-10-15

Verilog is a Terrible Language (for building or verifying hardware)

```
`timescale 1ns / 1ps
module uart_tx
  #(
    parameter
    CLK_FREQUENCY = 66_000_000,          // fpga clock frequency
    UART_FREQUENCY = 921_600             // UART clock frequency
    )
  (
   input       user_clk, rst_n, start_tx,
   input [7:0] data,
   output reg  tx_bit,
   output reg  ready,
   output reg  chipscope_clk
   );

  localparam
             TICKS_PER_BIT  = CLK_FREQUENCY / UART_FREQUENCY;
  localparam IDLE = 2'd0,
             INIT = 2'd1,
             TX   = 2'd2,
             DONE = 2'd3;

  reg [ 1:0]                 state, next_state;
  reg [ 2:0]                 bit_count;
  reg [ 7:0]                 data_buf;
  reg [11:0]                 clk_count;

  always @ (posedge user_clk or negedge rst_n) begin
    if (!rst_n)
      chipscope_clk <= 0;
    else if ((clk_count == TICKS_PER_BIT-1) |
             (clk_count == TICKS_PER_BIT>>1))
      chipscope_clk <= ~chipscope_clk;
    else
      chipscope_clk <= chipscope_clk;
  end
```

1. "Language Power"
   a. No Polymorphism
   b. No functional prog.
   c. No standard library
2. Behavioral Inference
   a. Flip-flops
   b. FPGA Block RAMS
   c. Finite-state Machines
3. Not a Single Source of Truth
   a. Clock/reset tree
   b. Floorplan
   c. UPF (power)
   d. Debug

```
always @ (posedge user_clk or negedge rst_n) begin
  state <= (!rst_n) ? IDLE : next_state;
end

always @ (posedge user_clk or negedge rst_n) begin
  if (!rst_n) begin
    tx_bit    <= 1; // should be pulled up on reset
    ready     <= 1;
    data_buf  <= 0;
    bit_count <= 0;
    clk_count <= 0;
  end
  else begin
    case (state)
      IDLE: begin
        /* ... */
      end
      INIT: begin
        /* ... */
      end
      TX: begin
        /* ... */
      end
      DONE: begin
        /* ... */
      end
      default: begin
        /* ... */
      end
    endcase
  end
end

always @ * begin
  case (state)
    IDLE:    next_state <= (start_tx)                    ? INIT : state;
    INIT:    next_state <= (clk_count == TICKS_PER_BIT-1) ? TX   : state;
    TX:      next_state <=
               ((bit_count == 7) & (clk_count == TICKS_PER_BIT-1)) ?
                 DONE : state;
    DONE:    next_state <= (clk_count == TICKS_PER_BIT-1) ? IDLE : state;
    default: next_state <= IDLE;
  endcase
end

endmodule
```

Code examples on this slide are Copyright 2018, Schuyler Eldridge, Apache 2.0: https://github.com/seldridge/verilog

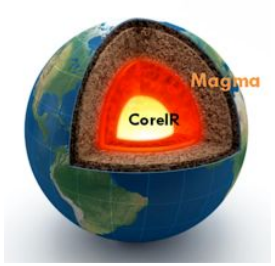
Pre-Cambrian Hardware Design: Everybody is Trying to Solve the Same Problems!

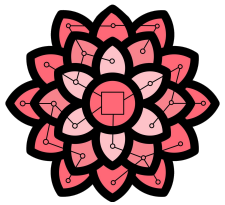
LLHD

FuseSOC

B-Lang

DFiant HDL





cocotb

edalize

VERILATOR

Dahlia

CHISEL

FIRRTL

SYSTEMC

Calyx

PyMTL

SpinalHDL
An alternative hardware description language

tcl
POWERED

Every language/effort is *reacting* to some issue with hardware:

- Industry has a cornucopia of Verilog/VHDL generators
- Verilog/VHDL are too low-level or not "powerful"
- Verilog/VHDL are the wrong abstraction
- Verilog/VHDL is hard to transform
- Verilog/VHDL doesn't capture all the information I need
- Verilog/VHDL simulation is a pain

These efforts all *replicate* similar approaches of "generators" (Figure 1) or "compilers" (Figure 2) and wind up re-solving the same problems:

- "Good" Verilog generation
- Compiler passes: CSE, DCE, Constant Propagation
- Fast, performant compiler infrastructure

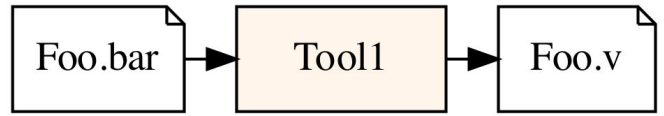


Figure 1: A Verilog Generator

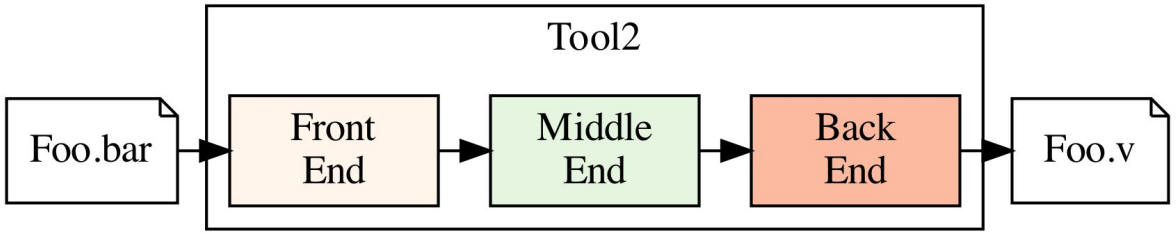
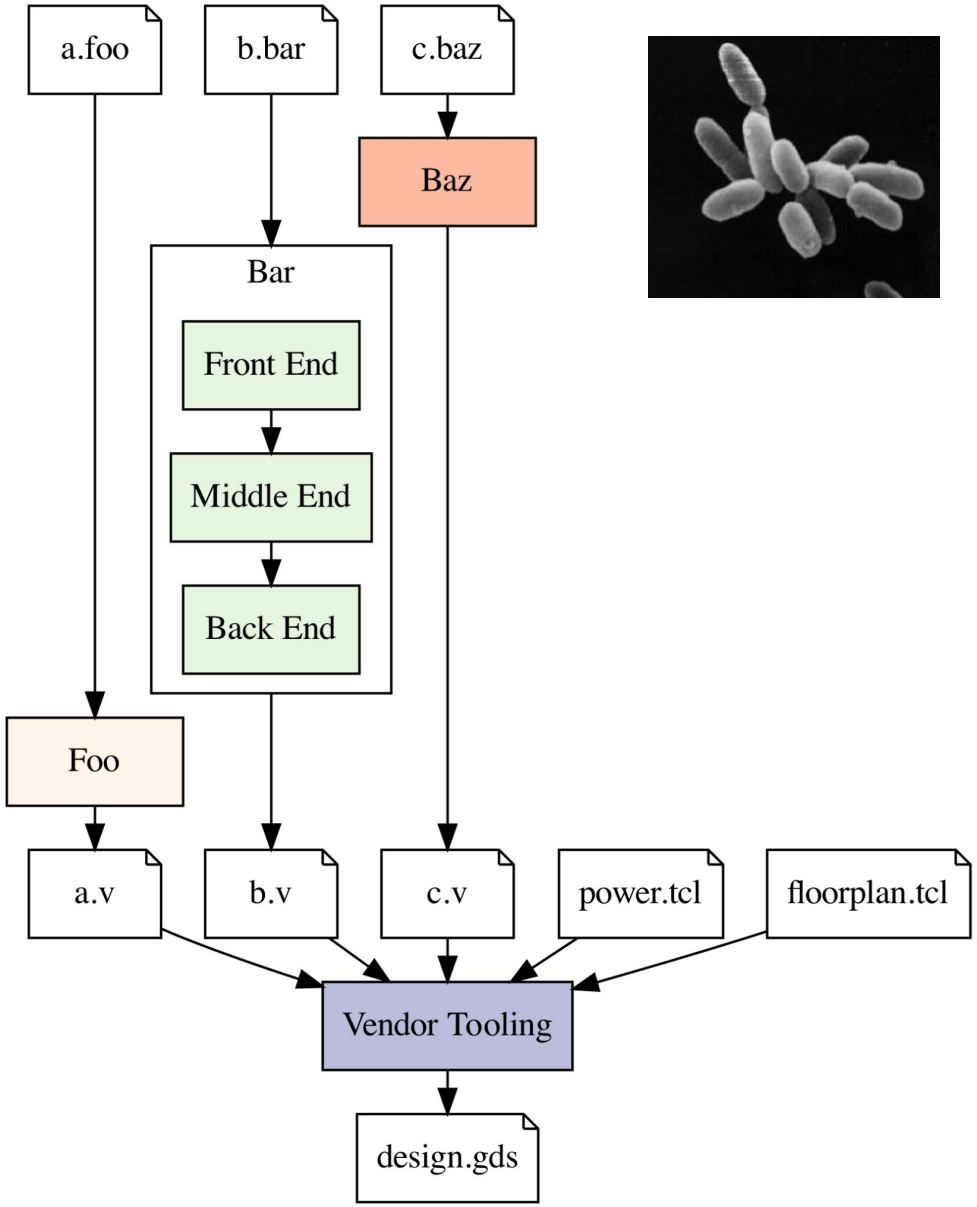
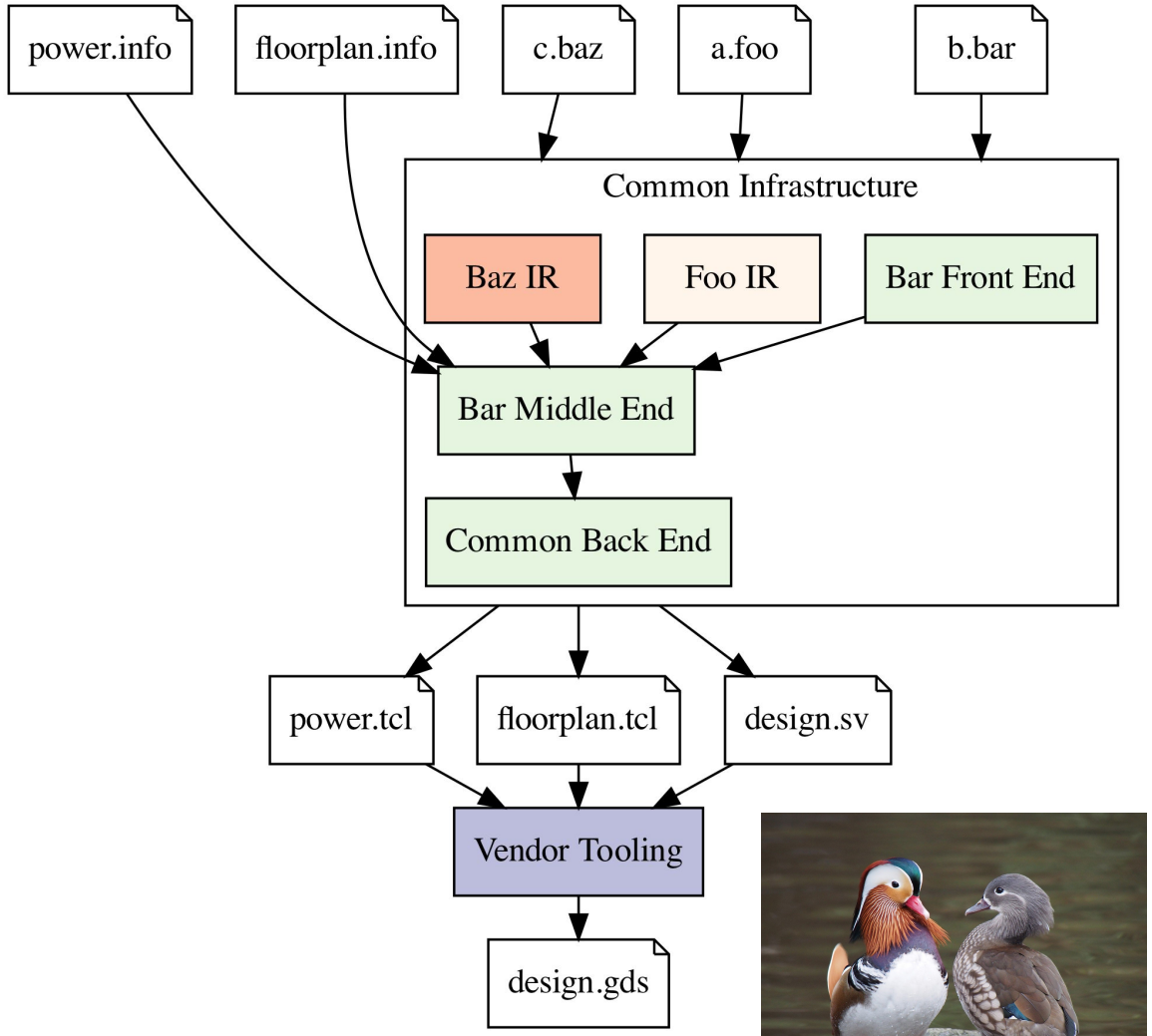


Figure 2: A Verilog Compiler-like Generator

A Cambrian Explosion of Hardware Design/Verification





MLIR (Multi-level IR) is Best-in-class Compiler Infrastructure

MLIR: A Compiler Infrastructure for the End of Moore's Law

Chris Lattner * Google
Mehdi Amini Google
Uday Bondhugula IISc
Albert Cohen Google
Andy Davis Google
Jacques Pienaar Google
River Riddle Google
Tatiana Shpeisman Google
Nicolas Vasilache Google
Oleksandr Zinenko Google

Abstract

This work presents MLIR, a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.

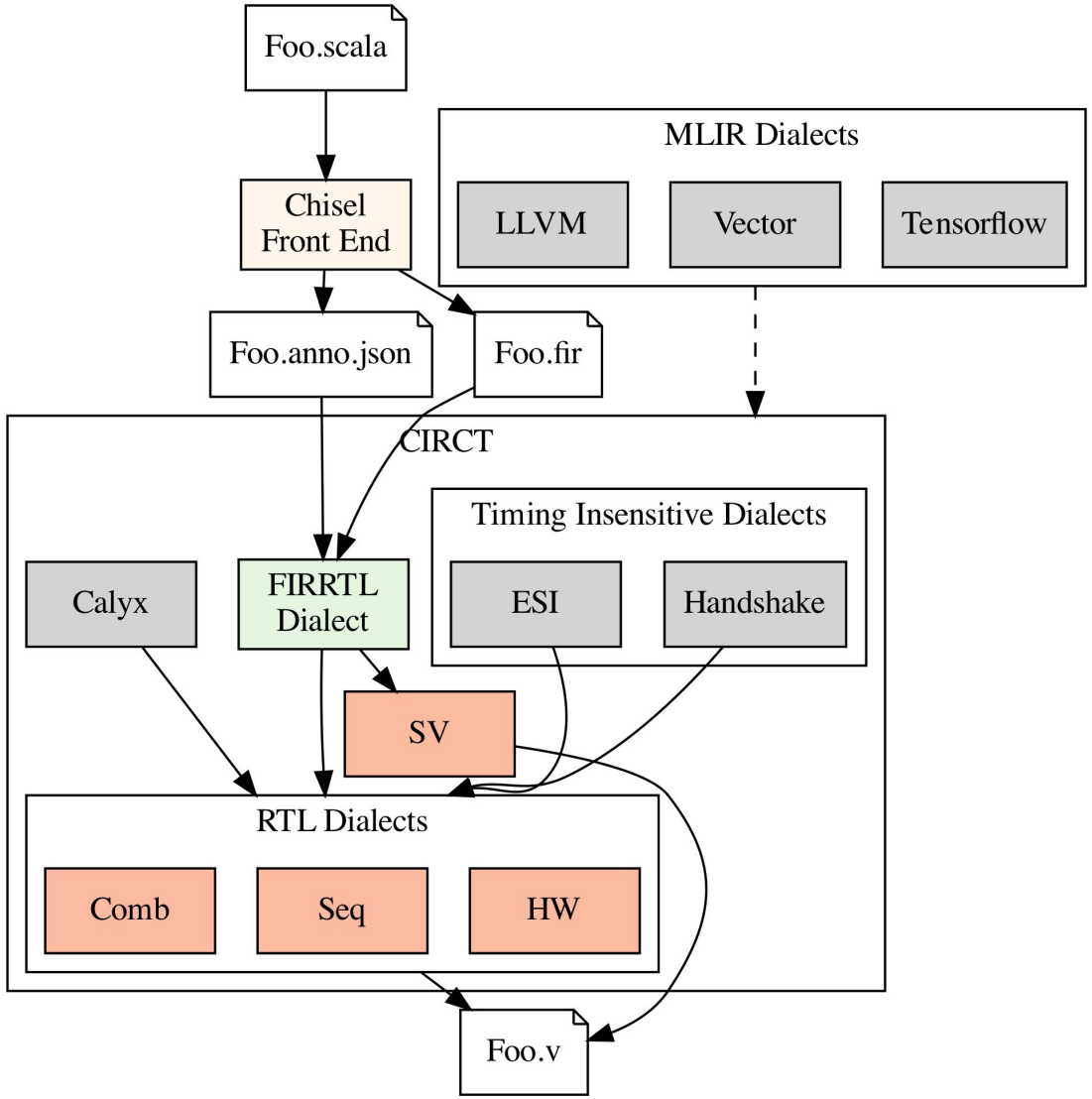
MLIR facilitates the design and implementation of code generators, translators and optimizers at different levels of abstraction and also across application domains, hardware targets and execution environments. The contribution of this work includes (1) discussion of MLIR as a research artifact, built for extension and evolution, and identifying the challenges and opportunities posed by this novel design point in design, semantics, optimization specification, system, and engineering. (2) evaluation of MLIR as a generalized infrastructure that reduces the cost of building compilers—describing diverse use-cases to show research and educational opportunities for future programming languages, compilers, execution environments, and computer architecture. The paper also presents the rationale for MLIR, its original design principles, structures and semantics.

https://mlir.llvm.org/

What is MLIR?

- *"Multi-level* IR"
- Part of the LLVM project
- Infrastructure for building new compilers
- Suitable for building new IRs
- Can handle *multiple* IRs at once
- Written in C++
- Extremely performant

CIRCT is *Hardware* Compiler Infrastructure



https://circt.llvm.org https://github.com/llvm/circt

What is CIRCT (pronounced "circuit")?

- **An LLVM incubator project**
- **Compiler infrastructure for generating Verilog**
- **A collection of hardware dialects**
  - FIRRTL Dialect
  - SystemVerilog Dialect
  - RTL Dialects
    - Combinational
    - Sequential
    - Hardware
  - Timing Insensitive Dialects
    - Elastic Silicon Interconnect (ESI)
    - Handshake
- **Eventually, a target for MLIR/LLVM**
- **A collection of tools for compiling circuits**
  - e.g., *firtool* is a drop in replacement for *firrtl*

Example: Chisel to Verilog through MLIR

Chisel to FIRRTL IR

```
class Foo extends Module {
  val a, b = IO(Input(UInt(2.W)))
  val c = IO(Output(UInt()))

  c := RegNext(a +& b)
}
```

Chisel Elaboration produces FIRRTL IR

```
circuit Foo :
  module Foo :
    input clock : Clock
    input reset : UInt<1>
    input a : UInt<2>
    input b : UInt<2>
    output c : UInt

    node _c_T = add(a, b)
    reg c_REG : UInt, clock with :
      reset => (UInt<1>("h0"), c_REG)
    c_REG <= _c_T
    c <= c_REG
```

SiFive

FIRRTL IR to FIRRTL Dialect

```
circuit Foo :
  module Foo :
    input clock : Clock
    input reset : UInt<1>
    input a : UInt<2>
    input b : UInt<2>
    output c : UInt

    node _c_T = add(a, b)
    reg c_REG : UInt, clock with :
      reset => (UInt<1>("h0"), c_REG)
    c_REG <= _c_T
    c <= c_REG
```

circt-translate -import-firrtl

```
module  {
  firrtl.circuit "Foo"   {
    firrtl.module @Foo(in %clock: !firrtl.clock, in %reset: !firrtl.uint<1>,
                       in %a: !firrtl.uint<2>, in %b: !firrtl.uint<2>,
                       out %c: !firrtl.uint) {
      %0 = firrtl.add %a, %b : (!firrtl.uint<2>, !firrtl.uint<2>) -> !firrtl.uint<3>
      %_c_T = firrtl.node %0  : !firrtl.uint<3>
      %c0_ui1 = firrtl.constant 0 : !firrtl.uint<1>
      %c_REG = firrtl.reg %clock  : !firrtl.uint
      firrtl.connect %c_REG, %_c_T : !firrtl.uint, !firrtl.uint<3>
      firrtl.connect %c, %c_REG : !firrtl.uint, !firrtl.uint
    }
  }
}
```

SiFive

FIRRTL Dialect to HW/Comb/SV Dialects

```
module  {
  firrtl.circuit "Foo"   {
    firrtl.module @Foo(in %clock: !firrtl.clock, in %reset: !firrtl.uint<1>,
                       in %a: !firrtl.uint<2>, in %b: !firrtl.uint<2>,
                       out %c: !firrtl.uint) {
      %0 = firrtl.add %a, %b : (!firrtl.uint<2>, !firrtl.uint<2>) -> !firrtl.uint<3>
      %_c_T = firrtl.node %0  : !firrtl.uint<3>
      %c0_ui1 = firrtl.constant 0 : !firrtl.uint<1>
      %c_REG = firrtl.reg %clock  : !firrtl.uint
      firrtl.connect %c_REG, %_c_T : !firrtl.uint, !firrtl.uint<3>
      firrtl.connect %c, %c_REG : !firrtl.uint, !firrtl.uint
    }
  }
}
```

`firtool -lower-to-hw`

```
module  {
  hw.module @Foo(%clock: i1, %reset: i1, %a: i2, %b: i2) -> (c: i3) {
    %false = hw.constant false
    %0 = comb.concat %false, %a : i1, i2
    %1 = comb.concat %false, %b : i1, i2
    %2 = comb.add %0, %1 : i3
    %c_REG = sv.reg  : !hw.inout<i3>
    %3 = sv.read_inout %c_REG : !hw.inout<i3>
    sv.alwaysff(posedge %clock)  {
      sv.passign %c_REG, %2 : i3
    }
    hw.output %3 : i3
  }
}
```

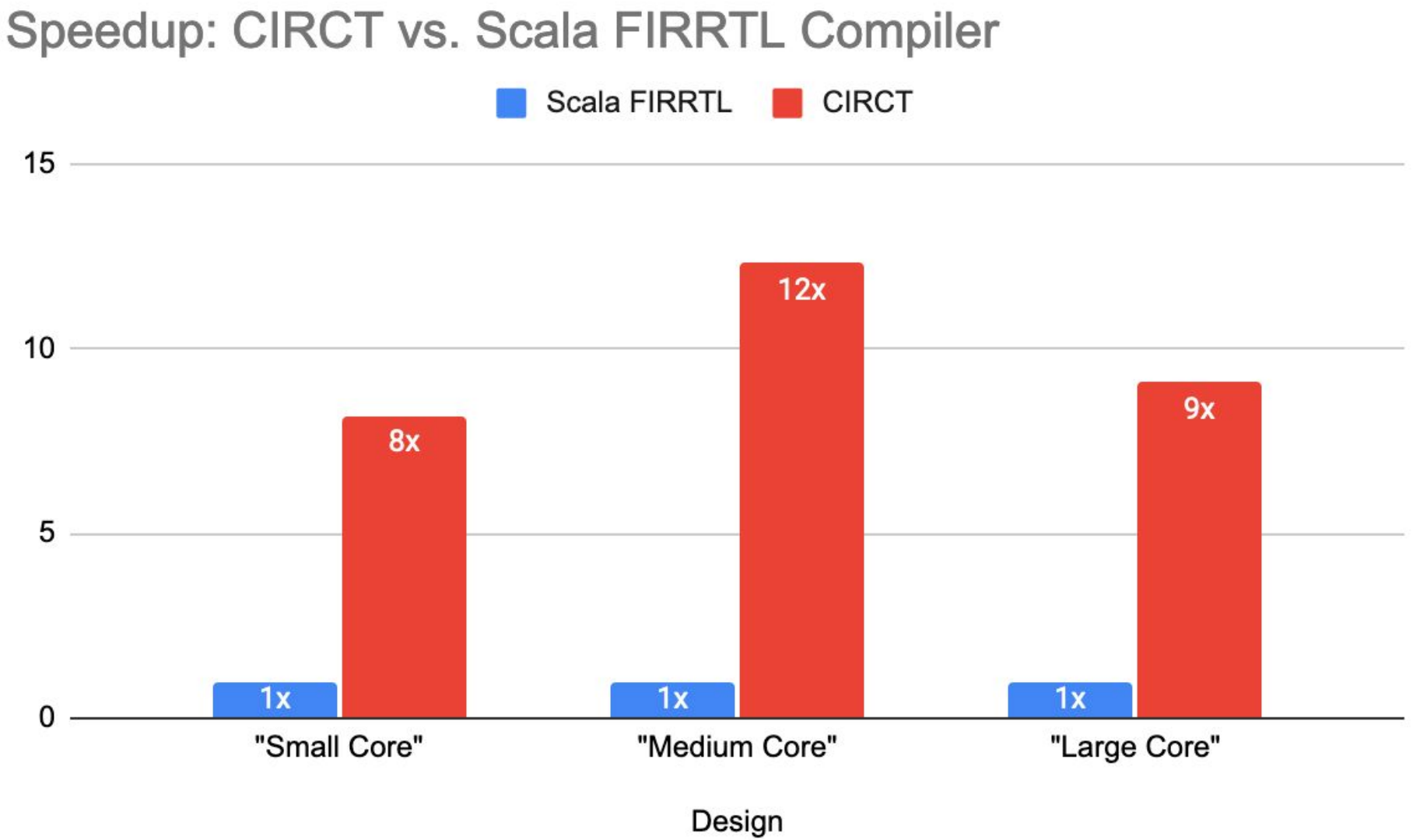
HW/Comb/SV Dialects to Verilog

```
module  {
  hw.module @Foo(%clock: i1, %reset: i1, %a: i2, %b: i2) -> (c: i3) {
    %false = hw.constant false
    %0 = comb.concat %false, %a : i1, i2
    %1 = comb.concat %false, %b : i1, i2
    %2 = comb.add %0, %1 : i3
    %c_REG = sv.reg  : !hw.inout<i3>
    %3 = sv.read_inout %c_REG : !hw.inout<i3>
    sv.alwaysff(posedge %clock)  {
      sv.passin %c_REG, %2 : i3
    }
    hw.output %3 : i3
  }
}
```

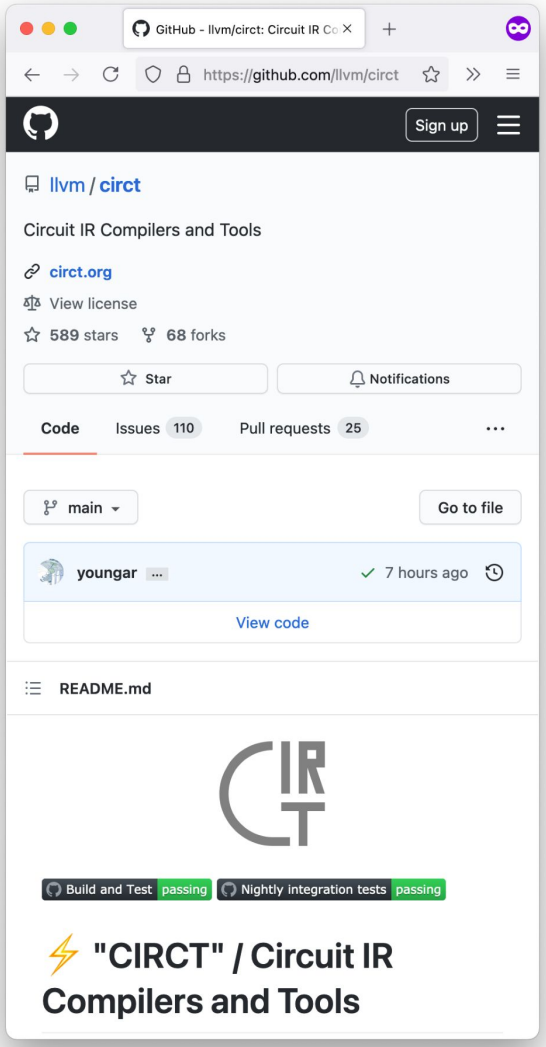
circt-opt -export-verilog

```
module Foo(
  input         clock, reset,
  input  [1:0] a, b,
  output [2:0] c);

  reg [2:0] c_REG;

  wire [2:0] _T = {1'h0, a} + {1'h0, b};
  always @(posedge clock)
    c_REG <= _T;
  assign c = c_REG;
endmodule
```

Early Results: An Order of Magnitude Faster FIRRTL Compiler



Come help us build next generation hardware infrastructure!



- **(Or consider using CIRCT if we've already done some work for you!)**
- **CIRCT is an open source project.**
  - We welcome contributors!
  - (We welcome your bug reports, too!)
  - CIRCT may be useful infrastructure for your product, project, or research
- **More information can be found on:**
  - GitHub:
    - github.com/llvm/circt
  - CIRCT Forums:
    - llvm.discourse.group/c/projects-that-want-to-become-official-llvm-projects/circt/40
  - Discord Server:
    - https://discord.gg/xS7Z362
  - We have open developer meetings on Wednesdays at 1100 PT

Acknowledgements

A large number of people at SiFive contribute to CIRCT: Andrew Lenharth, Andrew Young, Prithayan Barua, Fabian Schuiki, Hanchen Ye, Laura Gallo, Richard Xia, Aliaksei Chapyzhenka, Jack Koenig, Adam Izraelevitz, George Leontiev, Ram Sunder, Chris Lattner

A large number of people *outside* SiFive contribute to CIRCT: John Demme, Mike Urbach, Stephen Neuendorffer, Hideto Ueno, Chris Gyurgyik, Amalee Wilson, and many others...