


# Circuit IR for Compilers and Tools

## Heterogeneous Compilation in MLIR
## Hot Chips 34 Tutorial

Andrew Lenharth (SiFive)
John Demme (Microsoft)

# Demo:
# Creating hardware for ML

PyTorch to SystemVerilog with cosimulation

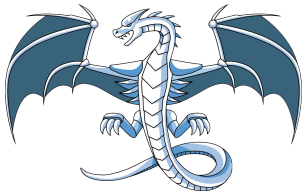
# CIRCT: Circuit IR for Compilers and Tools


**MLIR**-based tech for **HW Design and Verification**

- Composable toolchain for hardware design / EDA processes
- Focuses: High quality, usability, performance

**Modular** library based design to power next-gen ecosystem:

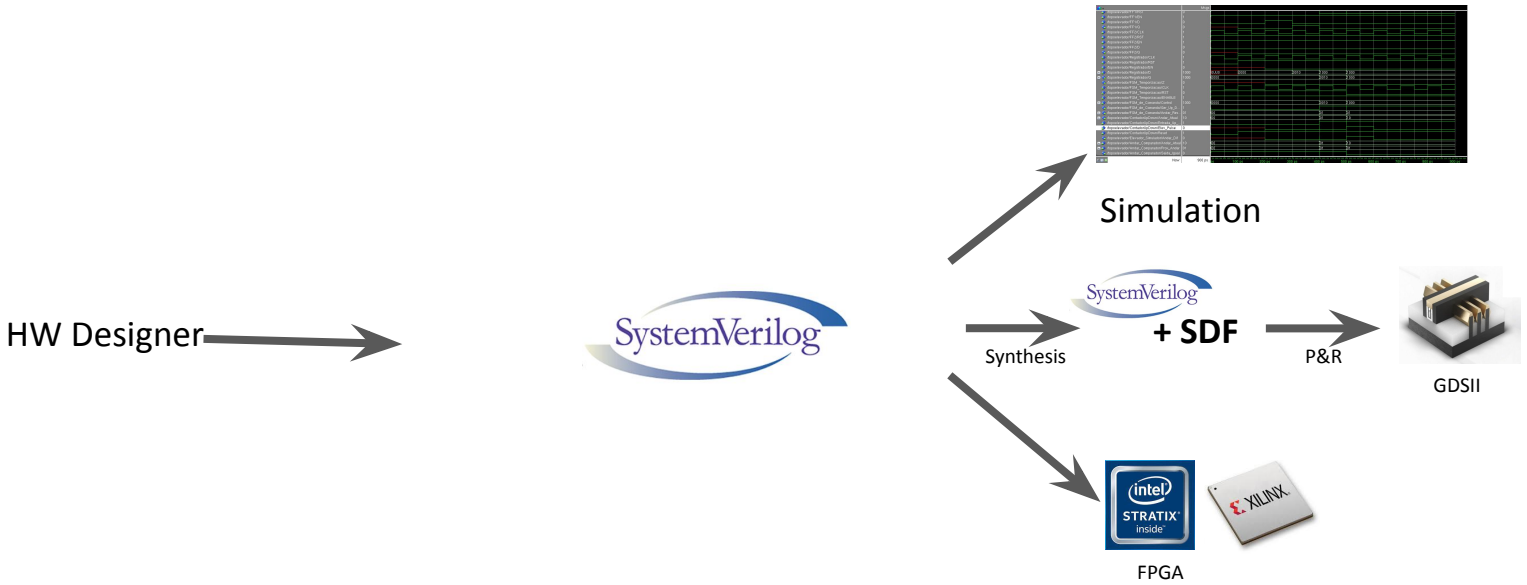
- Drive an innovation explosion for HW (like LLVM did for SW)



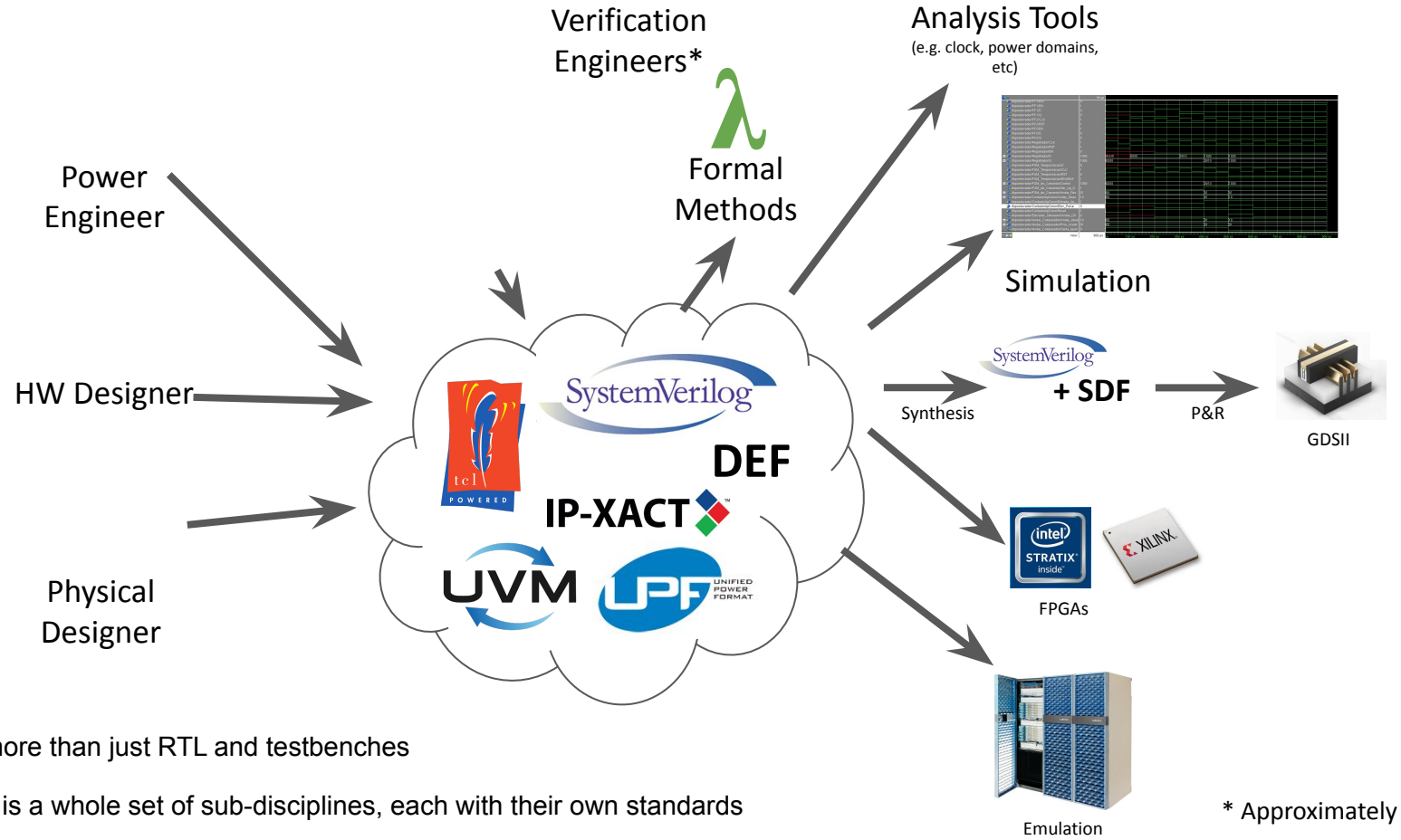
**LLVM Incubator Project:**

**https://circt.llvm.org**

# Making a chip is easy, right?



# Hardware design is a team sport; not one thing




Design is more than just RTL and testbenches

Verification is a whole set of sub-disciplines, each with their own standards

* Approximately to scale

# All aspects of a chip need specification

Pervasive redundancy, no single source of truth, little consistency

Each aspect of the design has different (sub-)languages

- Many languages are vendor or tool-dependent
- Specs are not orthogonal: reuse abstractions (despite poor abstraction capability)
- Redundancy: multiple sources of truth

These IRs are loosely coupled to the original design intent

- Long turn around and lots of effort to make changes
- Fragile layering

Designs become a mess of scripts, TCL, vendor-specific files, and duct-tape

*Lots of interacting tools and development flows!*

# Tools are great, but not as great as they could be!

Wonderful ecosystem of tools, but:

- Not always using best practices in software / compiler engineering
- Monolithic designs connected by unfortunate standards like Verilog
- Each framework / tool / tech stack is its own technology island

Each has a small developer community:

- Little shared code slows progress, each tool is missing features
- Features, quality of results, and user experience trails proprietary tools
- Poor SystemVerilog compatibility harms interoperation

**Problem:** no one is tackling the IR / representational issues!
**Problem:** duplication of effort on “uninteresting” parts!

Clipart by Pedro Neves

# "Library based design" in LLVM enabled a technology explosion!

OpenCL, CUDA, HLS, JIT'd database query engines, new languages like Swift, Rust, Julia, ….

**We need this for hardware design!**

# CIRCT is Tackling Interesting Problems

See LLVM 2021 Developer's Conference Keynote:

- Non-DAG logic (feedback)
- Hierarchical path specification
- SV Verification constructs
- Side-band design information
- Code generator integration
- Human-readable source code generation
- Interoperating with diverse, fragmented, and idiosyncratic tools
- Very large designs

# CIRCT has useful libraries and tech!

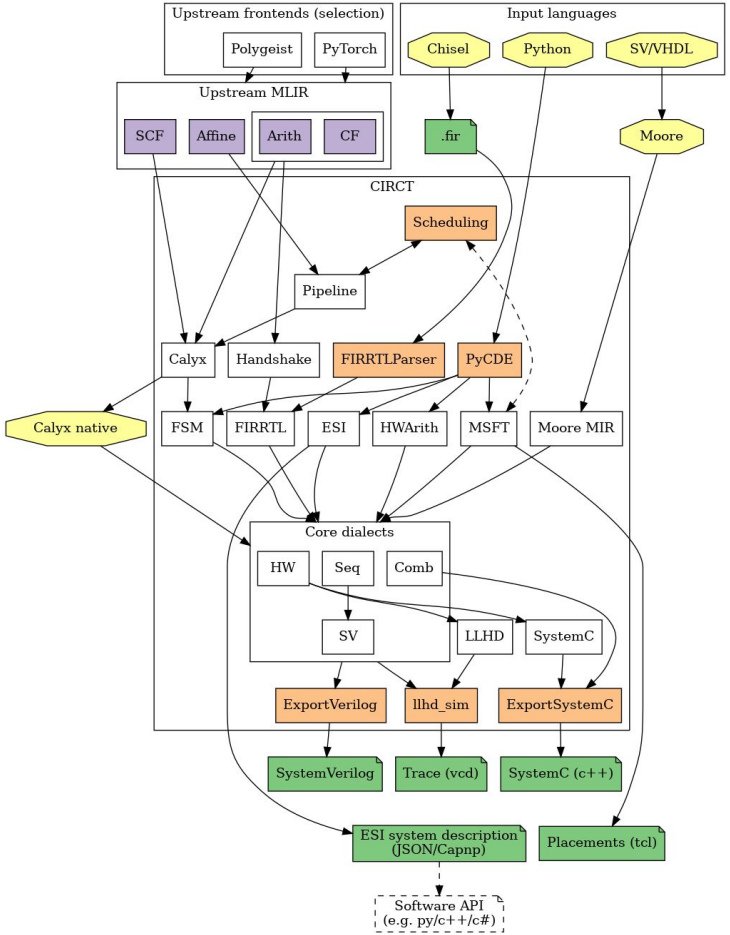
CIRCT dialects + implementations:

- Full, production FIRRTL implementation
- Core dialects stable
- Excellent SystemVerilog generation pipeline
- Active work on **all** dialects

Used both in production and research projects

"Frontend" use-cases stable and well-used:

- Tools that want to generate [System]Verilog
- Early interest in simulation, synthesis, P&R, etc

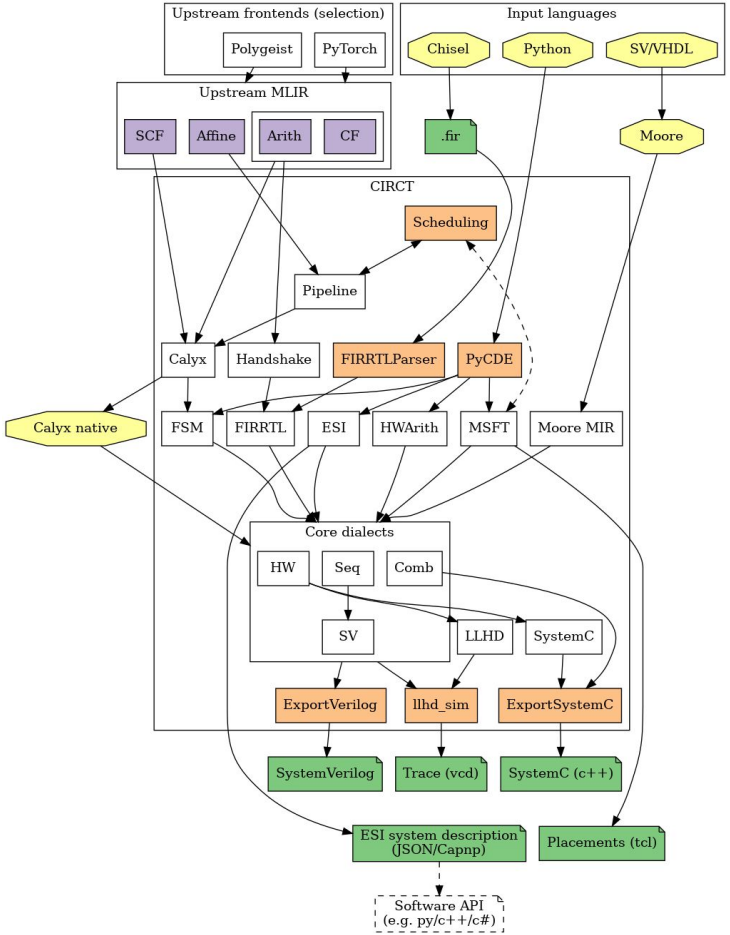


# Active CIRCT sub-projects

**Details later**
Core dialects
Emitting SystemVerilog
HLS flows
PyCDE (CIRCT Design Entry)
Elastic Silicon Interconnect
FIRRTL / Chisel

**Honorable mention**
Fast simulation
Scheduling framework
FSM
SystemC
FPGA physical design / high level constructs

+ several more



# Honorable mentions (selection of projects we won't be discussing further)

| | |
|---|---|
| Fast simulation | Keep high-level dialect ops for **fast functional simulation**. (Linear algebra runs a lot faster when compiled directly rather than in RTL simulation.) Lower to RTL to get cycle-level timing then correlate. |
| Scheduling framework[1] | HLS and other high-level lowerings need a scheduling system. The CIRCT scheduling framework abstracts away specific schedulers into a diverse set of "problem" models to suit numerous uses. |
| FSM | Abstraction for finite state machines, allowing easier reasoning and manipulation. Target-agnostic, generating efficient code for simulation and target hardware (e.g. ASIC, AMD/Intel FPGA). |
| SystemC | Models SystemC constructs and enables generating SystemC implementations. Work includes automatically generating systemC models from core dialects. |
| FPGA physical design[2] | Specify the placements of instances in a design instance hierarchy. Allows PD-conscious designers to define application-specific heuristics then produce a set of constraints with correct instance paths. |
| High level constructs[2] | Design systems at the level of **unscheduled** FSMs, data pipelines, broadcasts, systolic arrays, etc. Compiler can schedule/pipeline *correctly*, while physically optimizing. "**Placement-first pipelining**"™ |

* Various levels of stability/maturity, tending towards the experimental/infant end.

[1] "How to Make Hardware with Maths: An Introduction to CIRCT's Scheduling Infrastructure" [ Video ] [ Slides ] 2022 European LLVM Developer's Meeting, *Julian Oppermann, Technical University of Darmstadt*

[2] "Using CIRCT for FPGA Physical Design" [ Paper ] [ Video ] LATTE'22, John Demme and Aaron Landy (Microsoft)

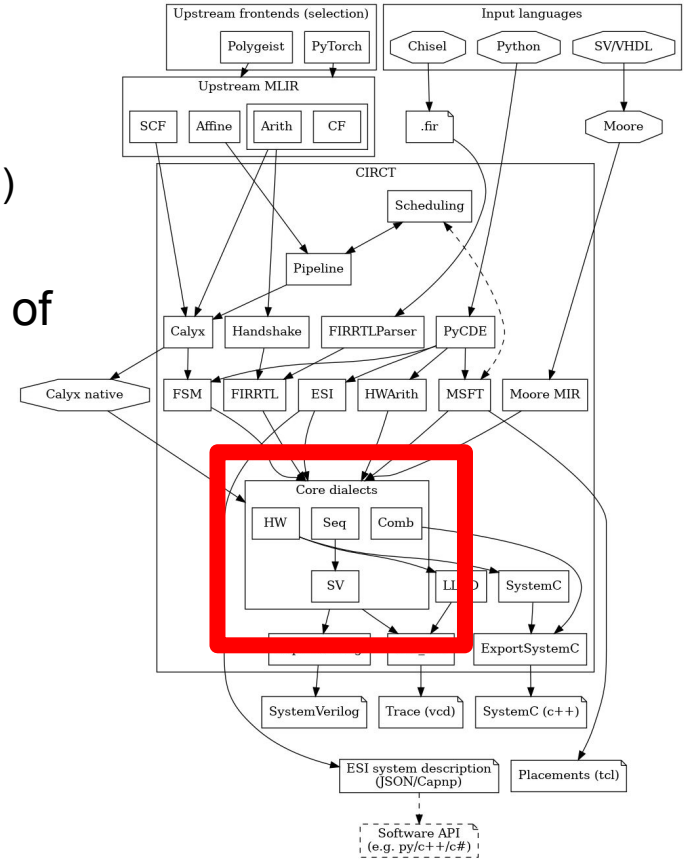
# Core dialects: the common denominator

**HW**: core abstractions
- Operations like module, instance (of a module)
- Also contains standard data types (int, array, struct, etc.)
- Status: **complete** and **stable**

**Comb**inational: computational ops without a sense of cycles or time
- Operations like add, shift, multiply, etc.
- Status: **complete** and **stable**

**Seq**uential: contains clocked storage elements
- Introduces a sense of time measured in cycles.
- Status: **incomplete** but **stable**

**SV: S**ystem **V**erilog weirdness



# Exporting SystemVerilog: ~~the good~~, the bad, and the ugly.

TL;DR: No tool implement the whole “standard” and no two in the same way – corner cases abound!
CIRCT abstracts away (most of) the pain

Goals: readable yet compatible with most tools. Requires supporting tool-specific SV “variants”.

## Tool specific annoyances:

- Token parse length
- Register syntactic form for some power tools
- Async reset register syntactic form for some synthesis and lint tools
- Automatic logic compatibility, or lack thereof
- No multi-dimensional arrays, no unpacked arrays, no structs (!)

## Tool specific optimizations:

- Muxes - so much complexity, compounded by weak pattern matching in some synthesis tools
- Vendor-specific annotations needed to get desired output

# Demo details

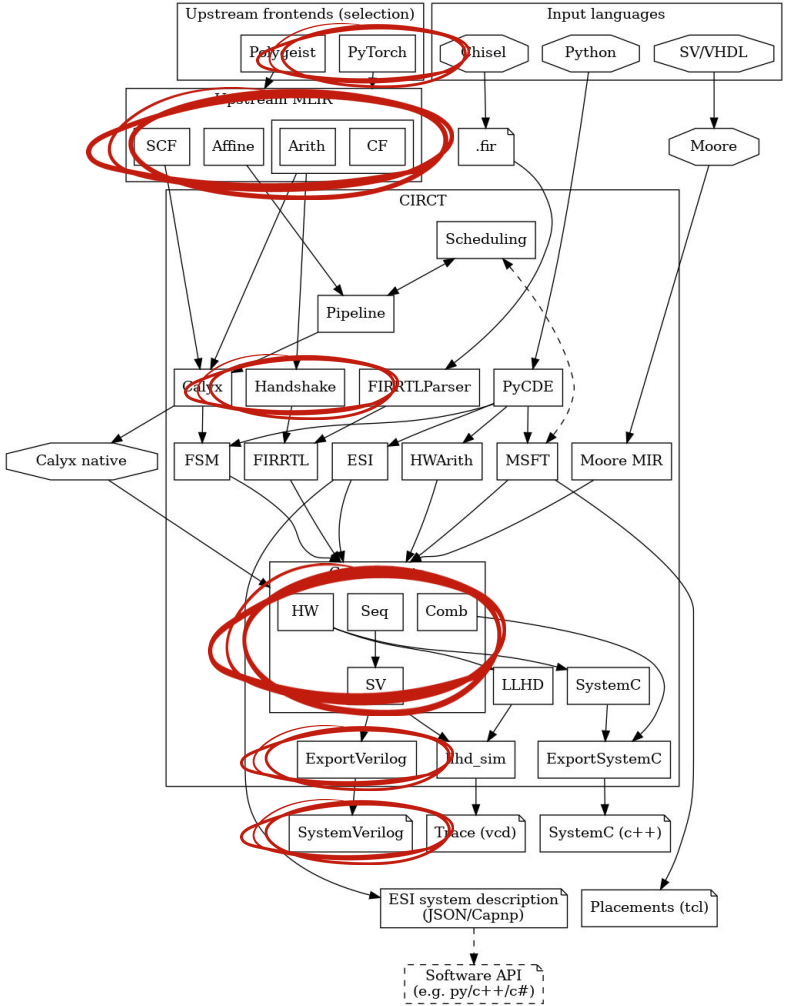
## PyTorch kernel (vector dot product)

```python
import torch


class DotModule(torch.nn.Module):
    def forward(self, a, b):
        return torch.matmul(a, b)
```

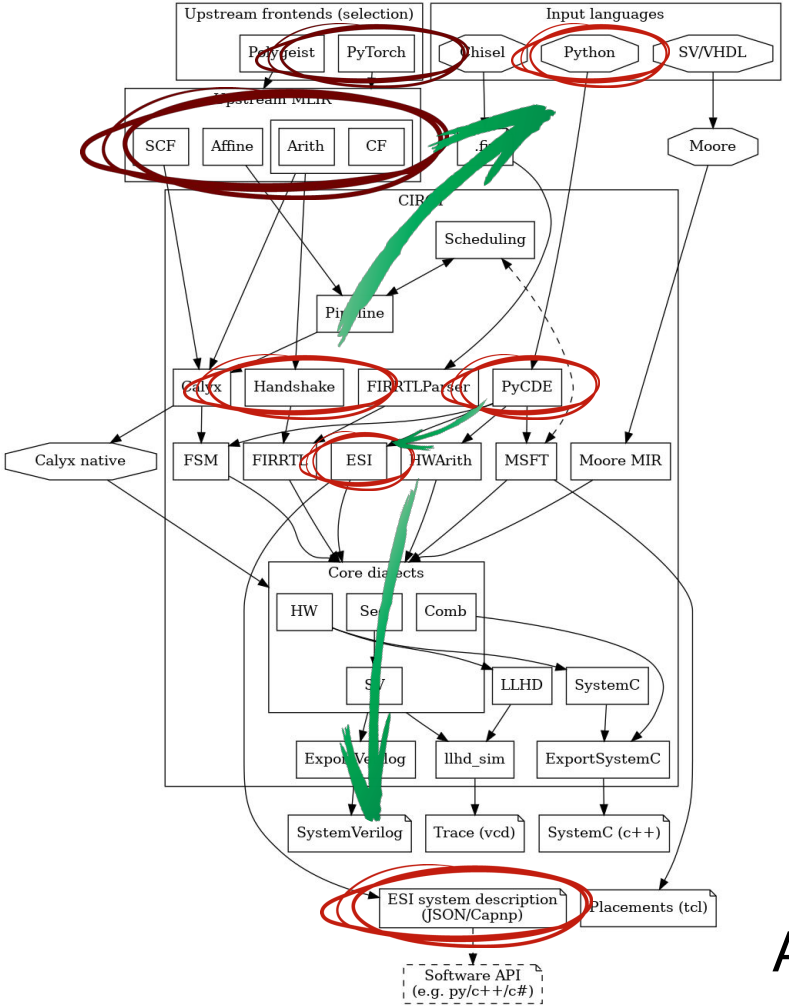
## Compiling with MLIR

```python
import torch
import torch_mlir

from dot import DotModule

shape = torch_mlir.TensorPlaceholder([5], torch.int32)

module = torch_mlir.compile(
    DotModule(), [shape, shape], output_type="linalg-on-tensors"
)

print(module)
```

# PyTorch code




# Step 1:
# Get to SystemVerilog




# Step 2: Assemble the system

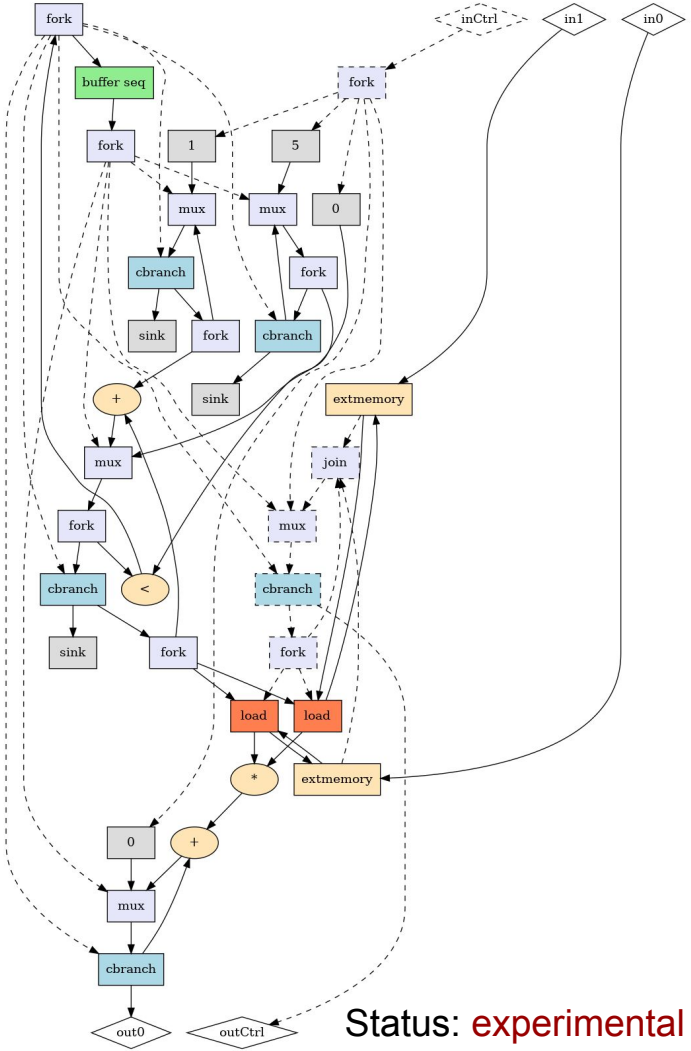
# Handshake: a dynamic dataflow IR

Dynamically-scheduled dataflow. Rather than a pre-computed schedule, tokens run through a dataflow graph at runtime.

Contains operations necessary to implement control constructs via dataflow. (e.g. `fork`, `join`, `conditional branch`, `conditional merge`, etc.)

Lowered to valid/ready semantics per operation.

The handshake implementation of the PyTorch dot product is on the right.




Status: experimental

# PyCDE: **C**IRCT **D**esign **E**ntry Python API

**System assembly**

Used to stitch together IP blocks and host connectivity with either:

- ESI (used in the demo)
- Ad-hoc wiring

The **demo** imports the PyTorch kernel module, wraps the raw wire ports in ESI then specifies ESI cosim as the external interface.

**Future work**: have the HLS lowering produce ESI "channel" ports to eliminate manual wrapping.

**Design entry**

CIRCT operations with *a lot* of syntactic sugar.
**NOT** a Python-based HDL!

Replace ad-hoc "generator" scripts which spit out RTL code. Gets all the goodness of the full CIRCT stack. (optimizations, SV compatibility, etc.)

Has an instance hierarchy browser API through which physical placements can be given, typically with a user-defined heuristic. (Used inside of Microsoft to physically optimize a ~60% FMax increase of an internal FPGA design.)

Enter any CIRCT operation – including the aforementioned high-level constructs.

Status: experimental

```python
@module
class HandshakeToESIWrapper:
  # Control Ports

  ## Generic ports always present
  clock = Input(types.i1)
  reset = Input(types.i1)

  ## Go signal
  go = InputChannel(types.i1)

  ## Done signal
  done = OutputChannel(types.i1)

  # Input 0 Ports

  ## Channels from Memory
  in0_ld_data0 = InputChannel(types.i32)

  ## Channels to Memory
  in0_ld_addr0 = OutputChannel(types.i64)

  # Input 1 Ports

  ## Channels from Memory
  in1_ld_data0 = InputChannel(types.i32)

  ## Channels to Memory
  in1_ld_addr0 = OutputChannel(types.i64)

  # Output 0 Ports

  ## Channels to Host
  result = OutputChannel(types.i32)
```

```python
@generator
 def generate(ports):
   # Typedefs
   ctrl_channel_type = types.channel(types.i1)
   i32_channel_type = types.channel(types.i32)
   i64_channel_type = types.channel(types.i64)

   # Instantiate the top-level module to wrap with backedges for most ports.
   wrapped_top = top(clock=ports.clock, reset=ports.reset)

   # Control Ports

   ## Go signal
   _, in_ctrl_valid = ports.go.unwrap(wrapped_top.inCtrl_ready)
   wrapped_top.inCtrl_valid.connect(in_ctrl_valid)

   ## Done signal
   out_ctrl_channel, out_ctrl_ready = ctrl_channel_type.wrap(1, wrapped_top.outCtrl_valid)
   wrapped_top.outCtrl_ready.connect(out_ctrl_ready)
   ports.done = out_ctrl_channel

   # Input 0 Ports

   ## Channels from Memory
   in0_ready = wrapped_top.in0_ldData0_ready & wrapped_top.in0_ldDone0_ready

   in0_ld_data0_data, in0_ld_data0_valid = ports.in0_ld_data0.unwrap(in0_ready)
   wrapped_top.in0_ldData0_data.connect(in0_ld_data0_data)
   in0_valid = Mux(in0_ready, types.i1(0), in0_ld_data0_valid)
   wrapped_top.in0_ldData0_valid.connect(in0_valid)
   wrapped_top.in0_ldDone0_valid.connect(in0_valid)

   ## Channels to Memory
   in0_ld_addr0_channel, in0_ld_addr0_ready = i64_channel_type.wrap(
      wrapped_top.in0_ldAddr0_data, wrapped_top.in0_ldAddr0_valid)
   wrapped_top.in0_ldAddr0_ready.connect(in0_ld_addr0_ready)
   ports.in0_ld_addr0 = in0_ld_addr0_channel

   # Input 1 Ports

   ## Channels from Memory
   in1_ready = wrapped_top.in1_ldData0_ready & wrapped_top.in1_ldDone0_ready

   in1_ld_data0_data, in1_ld_data0_valid = ports.in1_ld_data0.unwrap(in1_ready)
   wrapped_top.in1_ldData0_data.connect(in1_ld_data0_data)
   in1_valid = Mux(in0_ready, types.i1(0), in1_ld_data0_valid)
   wrapped_top.in1_ldData0_valid.connect(in1_valid)
   wrapped_top.in1_ldDone0_valid.connect(in1_valid)

   ## Channels to Memory
   in1_ld_addr0_channel, in1_ld_addr0_ready = i64_channel_type.wrap(
       wrapped_top.in1_ldAddr0_data, wrapped_top.in1_ldAddr0_valid)
   wrapped_top.in1_ldAddr0_ready.connect(in1_ld_addr0_ready)
   ports.in1_ld_addr0 = in1_ld_addr0_channel

   # Output 0 Ports
   out0_channel, out0_ready = i32_channel_type.wrap(wrapped_top.out0_data, wrapped_top.out0_valid)
   wrapped_top.out0_ready.connect(out0_ready)
   ports.result = out0_channel
```

Hand-written PyCDE wrapping code. In the future, this will not be necessary.

# IP composability: Elastic Silicon Interconnect

Our FPGA system assembly dialect

“Plumbing” is tedious and error-prone.

- Endianness, off-by-one cycle bugs, type mismatches, etc. CDCs!
- Too low-level!

Solution: raise the level of abstraction!

- Compiler technology to the rescue

# ESI: static type safety, high-level types

Untyped buses (i.e. bundle of wires) are very brittle!

- When the data types on the wires change, consumers misinterpret the data -> bug!
- Static type safety has been **wildly successful** in software at preventing bugs.

ESI has (plans for) a rich, hardware-centric type system to enable modeling accurately.

- Structs, array, enums, ints, etc. (table stakes) currently.
- Coming: Variable-length types. “Data windows” to specify source, destination port bandwidth.

Allows ESI to build rich, system-tailored software APIs automatically.

- Extend static type safety into software.
- Same API regardless of hardware bridge – PCIe, network, **simulation**, etc.

“Elastic Silicon Interconnects: Abstracting Communication in Accelerator Design” J. Demme, LATTE’21 [paper] [talk]

Status: experimental

# ESI: latency-insensitive model

Communications *modeled* as latency-insensitive "channels".

- Gives the compiler flexibility in IP-to-IP comms. (e.g. pipeline links, optimize bus widths, etc.)
- Build host or inter-device communication bridges anywhere.

HLS and some newer HDLs provide easy host connectivity (which is great), BUT:

- Can create vendor lock-in.
- Since they need control of the top-level, creates "language lock-in".
- Each HDL/HLS needs to add support for every FPGA board (many-to-many problem).

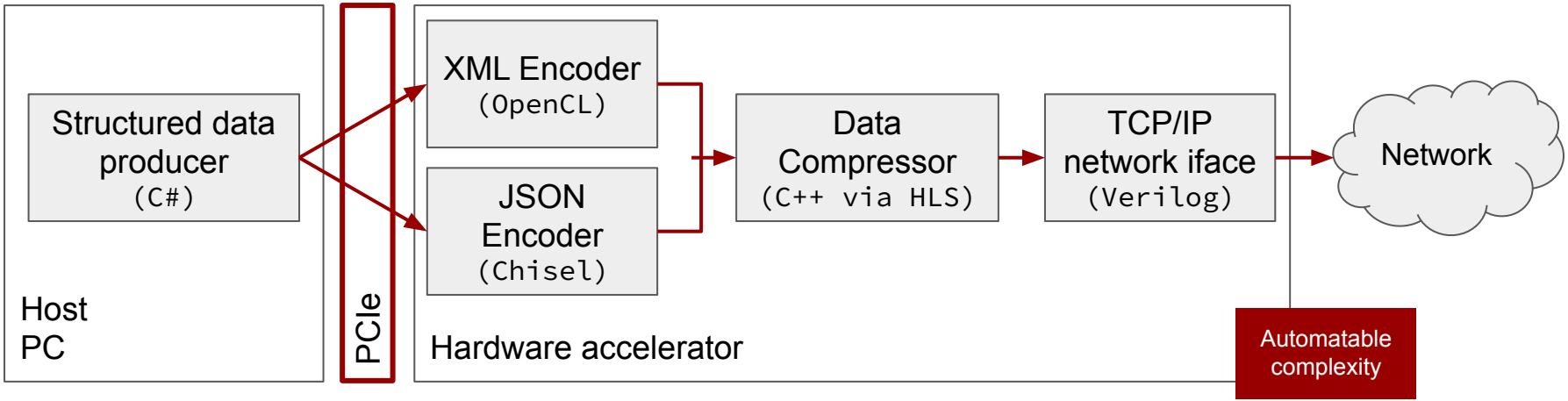
ESI provides a language and vendor agnostic system assembly system:

- HDLs/HLS produce IP blocks with ESI interfaces.
- Board vendors provide an ESI board support package.
- Use ESI to assemble a language heterogeneous system and wire it to any board!

"Elastic Silicon Interconnects: Abstracting Communication in Accelerator Design" J. Demme, LATTE'21 [paper] [talk]

Status: experimental

# ESI: latency-insensitive model



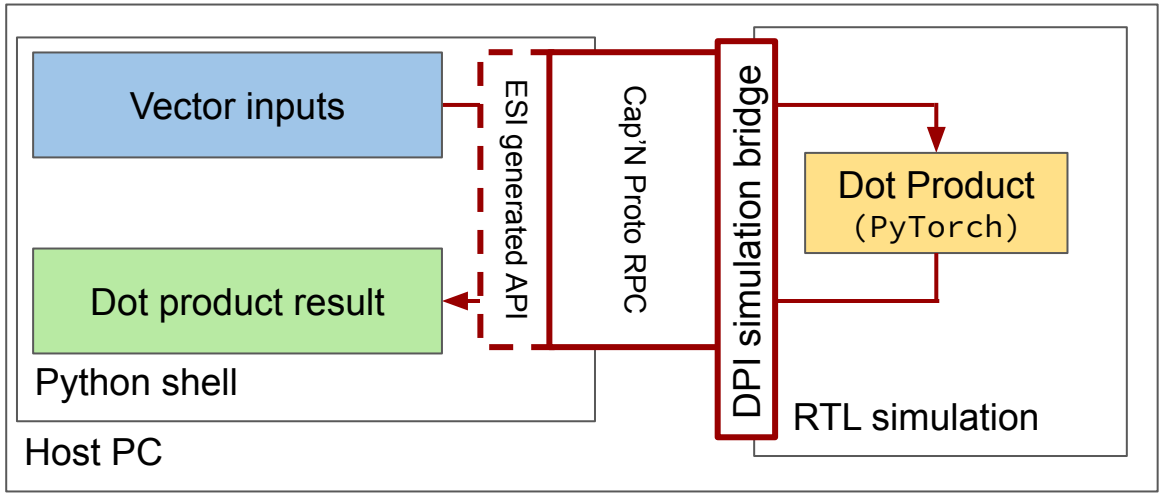

ESI provides a language and vendor agnostic system assembly system:

- HDLs/HLS produce IP blocks with ESI interfaces.
- Board vendors provide an ESI board support package.
- Use ESI to assemble a language-heterogeneous system and wire it to any board!

"Elastic Silicon Interconnects: Abstracting Communication in Accelerator Design" J. Demme, LATTE'21 [paper] [talk]

Status: experimental

# Demo ESI system

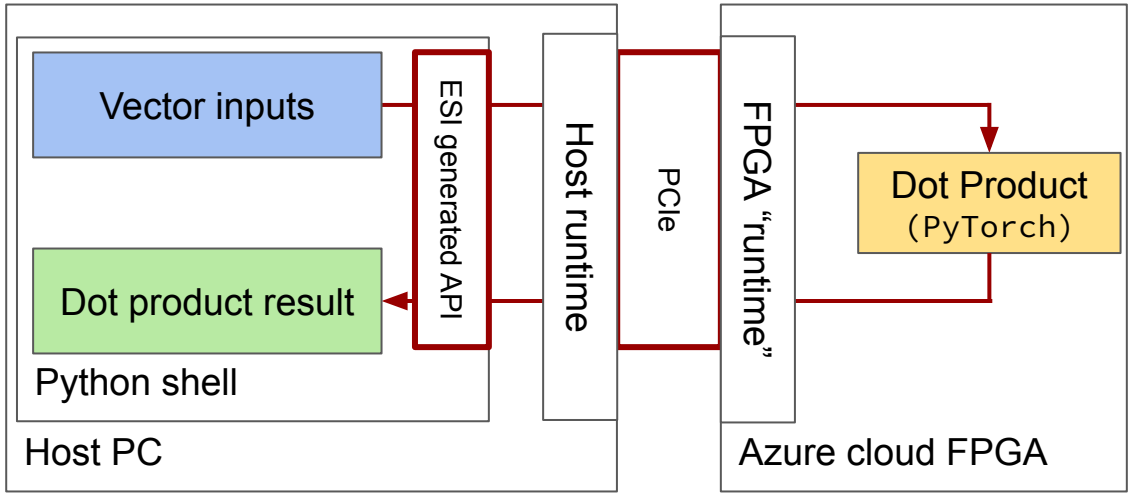



(simplified)

1. Vector inputs sent from Python to (hand-built, future auto) Python API.
2. Transferred via co-simulation RPC / DPI link to the PyTorch kernel.
3. Results transferred back through same links.

Status: experimental

# Coming soon to a cloud FPGA ~~near~~ *in a region close to* you!




# Mere person-weeks away!

```python
@esi.ServiceDecl
class HandshakeServices:
  go = esi.FromServer(types.i1)
  done = esi.ToServer(types.i1)
  read_mem = esi.ToFromServer(to_server_type=types.i64,
                              to_client_type=types.i32)
  result = esi.ToServer(types.i32)

@module
class Top:
  clock = Input(types.i1)
  reset = Input(types.i1)

  @generator
  def generate(ports):
    DotProduct(clock=ports.clock, reset=ports.reset)
    esi.Cosim(HandshakeServices,
              ports.clock, ports.reset)
```

```python
@module
class DotProduct:
  """An ESI-enabled module which only communicates with
     the host and computes dot products."""
  clock = Input(types.i1)
  reset = Input(types.i1)

  @generator
  def generate(ports):
    # Get the 'go' signal from the host.
    go = HandshakeServices.go("dotprod_go")

    # Instantiate the wrapped PyTorch dot product module.
    wrapped_top = HandshakeToESIWrapper(clock=ports.clock,
                                        reset=ports.reset,
                                        go=go)

    # Connect up the channels from the pytorch module.
    HandshakeServices.done("dotprod_done", wrapped_top.done)
    HandshakeServices.result("result", wrapped_top.result)

    # Connect up the memory ports.
    port0_data = HandshakeServices.read_mem(
            "port0", wrapped_top.in0_ld_addr0)
    wrapped_top.in0_ld_data0.connect(port0_data)
    port1_data = HandshakeServices.read_mem(
            "port1", wrapped_top.in1_ld_addr0)
    wrapped_top.in1_ld_data0.connect(port1_data)
```

Uses ESI “services”.
Won’t have time to go into today.

# ESI system assembly through PyCDE – cosim edition

```python
import torch
import numpy as np
from esi_cosim import HandshakeCosimBase, get_cosim_port
from dot import DotModule

class DotProduct(HandshakeCosimBase):
  pytorch_dot = DotModule()

  def run(self, a, b):
    self.memories[0] = a
    self.memories[1] = b
    self.go()
    return self.read_result()

  def run_checked(self, a, b):
    print(f"Computing dot product of {a} and {b}")
    result = self.run(a, b)
    tensor_a = torch.IntTensor(a)
    tensor_b = torch.IntTensor(b)
    dot = self.pytorch_dot.forward(tensor_a, tensor_b)
    print(f"from cosim: {result}, from pytorch: {dot}")

def rand_vec():
  return [np.random.randint(0, 100) for _ in range(5)]

cosim = DotProduct(get_cosim_port())

cosim.run_checked(rand_vec(), rand_vec())
```

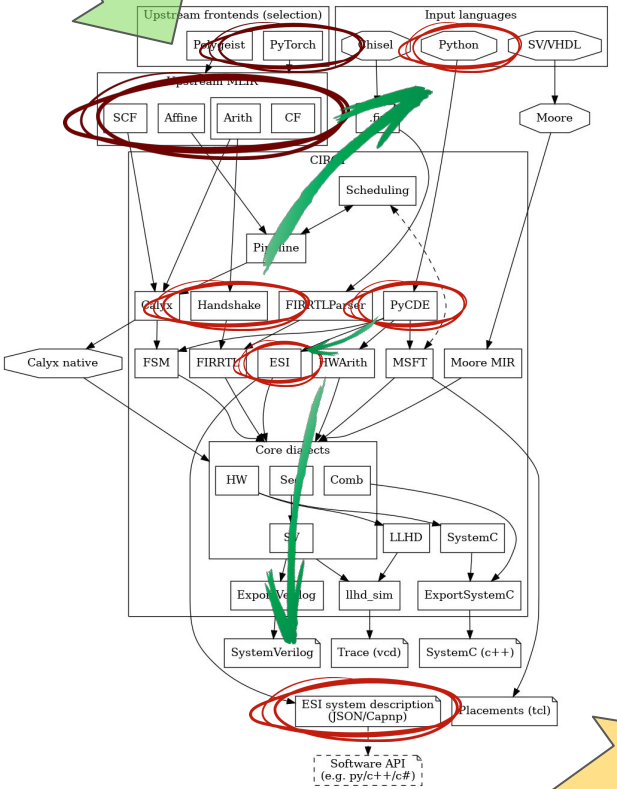
# Using the system in Python

```python
class HandshakeCosimBase(CosimBase):

  def __init__(self, port):
    super().__init__("PyCDESystem/schema.capnp", f"{os.uname()[1]}:{port}")
    self.done = self.openEP(1001,
                            sendType=self.schema.I1,
                            recvType=self.schema.I1)
    self.memory_ports = [
        self.openEP(1003, sendType=self.schema.I64, recvType=self.schema.I32),
        self.openEP(1004, sendType=self.schema.I64, recvType=self.schema.I32)
    ]
    self.result = self.openEP(1002,
                              sendType=self.schema.I32,
                              recvType=self.schema.I1)
    self.go_chan = self.openEP(1005,
                               sendType=self.schema.I1,
                               recvType=self.schema.I1)
    self.memories = [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

  def go(self):
    self.go_chan.send(self.schema.I1.new_message(i=False))

    while self.readMsg(self.done, self.schema.I1) is None:
      self.service_memories()
      time.sleep(0.01)

  def read_result(self):
    result = None
    while result is None:
      result = self.readMsg(self.result, self.schema.I32)
      time.sleep(0.01)
    return result.i

  def service_memories(self):

    def service(mem, port):
      addr = self.readMsg(port, self.schema.I64)
      if addr is not None:
        port.send(self.schema.I64.new_message(i=mem[addr.i]))

    for port, mem in zip(self.memory_ports, self.memories):
      service(mem, port)
```

Hand-built API. In the future, this will be built automatically.

## PyTorch kernel (vector dot product)

```python
import torch


class DotModule(torch.nn.Module):
    def forward(self, a, b):
        return torch.matmul(a, b)
```




```
$ ./cosim_demo.sh
##############################################################
## Generating the ESI system with PyCDE.
##    Outputs: SystemVerilog and Cosimulation schema.
##############################################################

... done.


##############################################################
## Compiling the SystemVerilog to simulation with Verilator
##############################################################

make: Entering directory '/code/hot-chips-2022-pytorch-circ
ccache g++  -I.  -MMD -I/code/circt/ext/share/verilator/inc
[...]
make: Leaving directory '/code/hot-chips-2022-pytorch-circt

... done.


##############################################################
## Running the Verilator simulation.
##############################################################

... started.


##############################################################
## Connecting to the simulation via ESI cosim, and dropping
##############################################################

Computing dot product of [42, 90, 24, 13, 38] and [21, 41,
from cosim: 5418, from numpy: 5418
>>> cosim.run_checked(rand_vec(), rand_vec())
Computing dot product of [19, 52, 11, 75, 80] and [21, 59,
from cosim: 15485, from numpy: 15485
>>>


Killing simulation.
... killed.
```

# Demo “caveats”


- Doesn’t produce efficient HW.
- Only works for a subset of PyTorch kernels.
- A few days’ worth of ad-hoc scripting (e.g. wrapping the kernel).

**BUT**

The PyTorch/HLS flow doesn’t have major funding (interns, students, and part-time).

Demonstrates the potential of CIRCT!

# Chisel/FIRRTL Compiler

# "Chisel" Generator Framework


## Scala API for generating SystemVerilog:

- Support high abstraction design, type checking to detect errors, application of SW techniques

## ***SiFive*** uses Chisel pervasively:

- All SiFive RISC-V Cores and several SoC's built with Chisel
- We have many extensions and custom things built into and around it

# Chisel is a *compiler* built on the "FIRRTL" IR

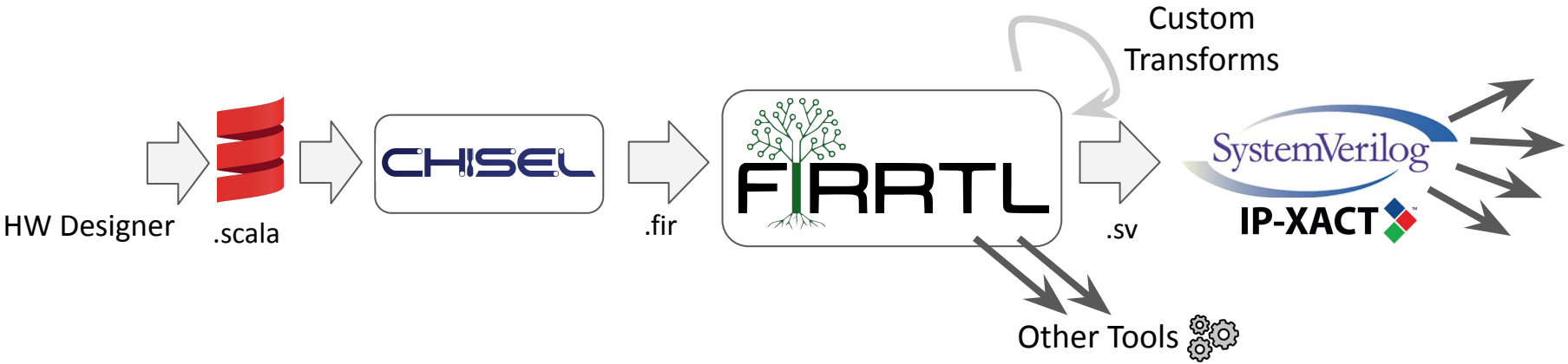

"One Shot" Lowering to Verilog was too complicated, so FIRRTL was introduced:

- Progressive (multi-level) lowering of complex types and operations
- Analyses like "width inference", a form of dataflow-based type checking
- Correct generation of Verilog text is… more complicated than it should be

The "*imperative code that builds a graph*" model crosscuts domains:

- e.g. Imperative Keras Python API ⇒ TensorFlow graph
- Well designed IRs make it much easier to analyse and transform the design!

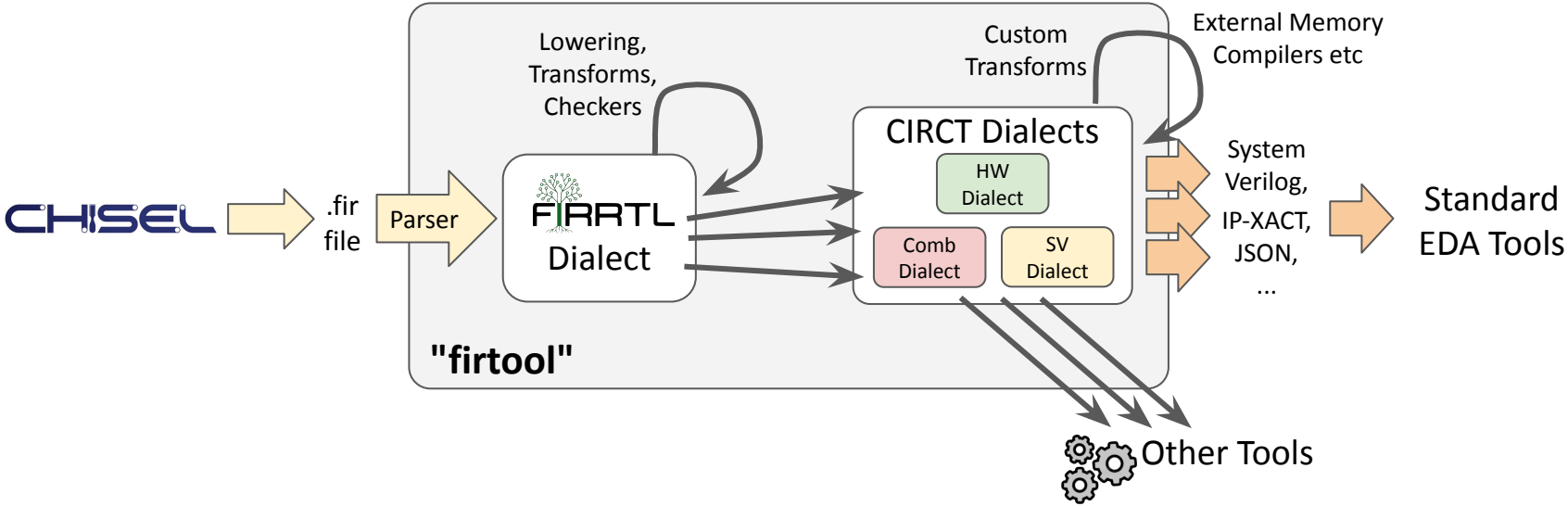
# A compiler IR for hardware enabled many new tools!




Building **robust** tools is **fast and cheap** using the FIRRTL IR:

- DFT/Scan chain insertion, time-multiplexing transformations, Host / Target clock decoupling for pause-able models, module hierarchy transforms (e.g. for power domains), run-time fault injection, circuit obfuscation, etc
- Custom checks: clock domain crossing, clock/reset synchronization safety, width inference checks, ...
- Simulators: ESSENT Simulator, FireSim AWS F1 FPGA Accelerated System Simulator, ...

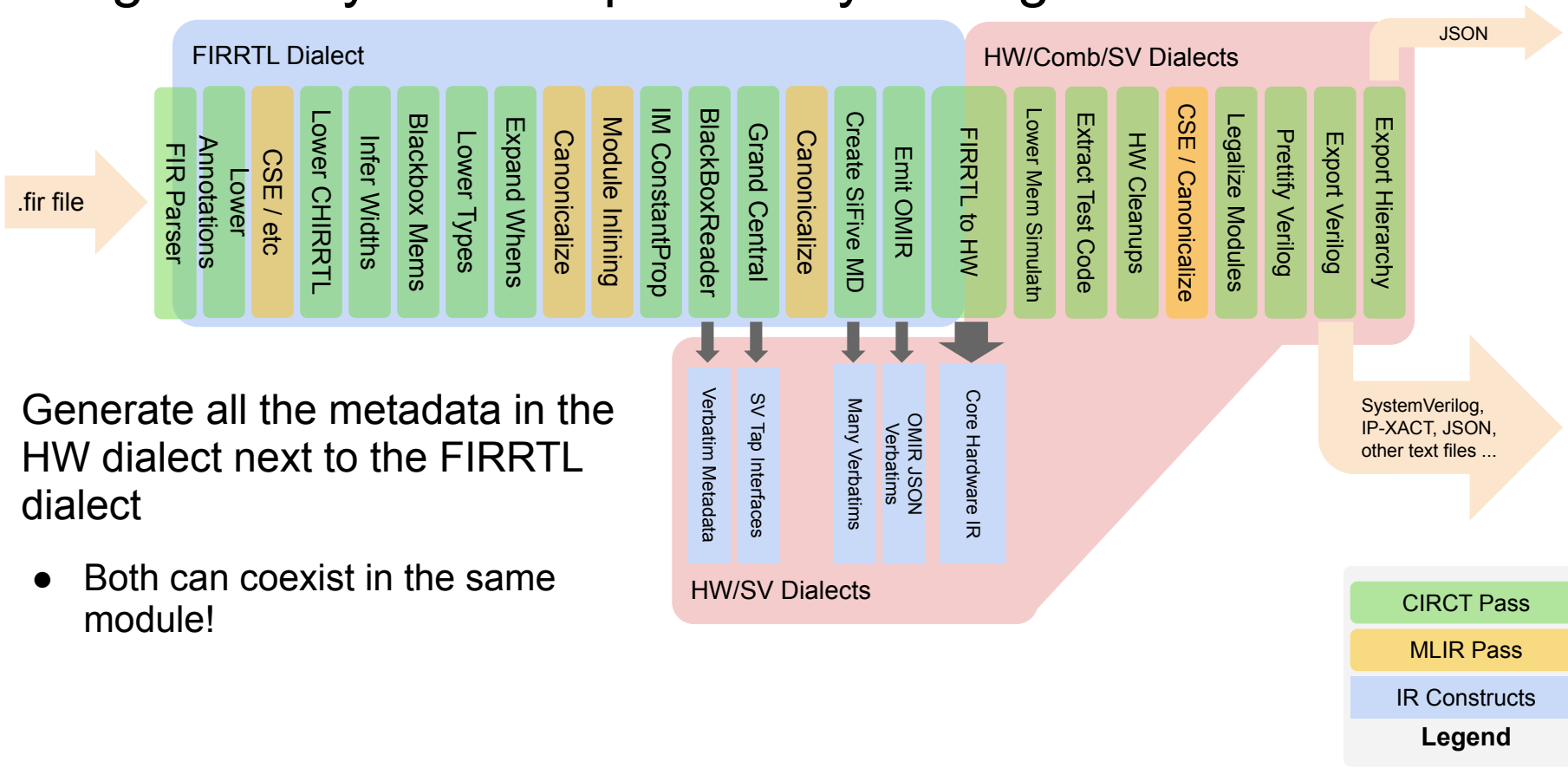
# "firtool" is an implementation of FIRRTL compiler




**Drop in replacement for the Scala FIRRTL compiler:**

- Lives entirely in the CIRCT project, heavily builds (and often drives) its infrastructure work
- Production quality for SiFive flows (among others)
- Will be the standard Chisel FIRRTL compiler in future chisel release

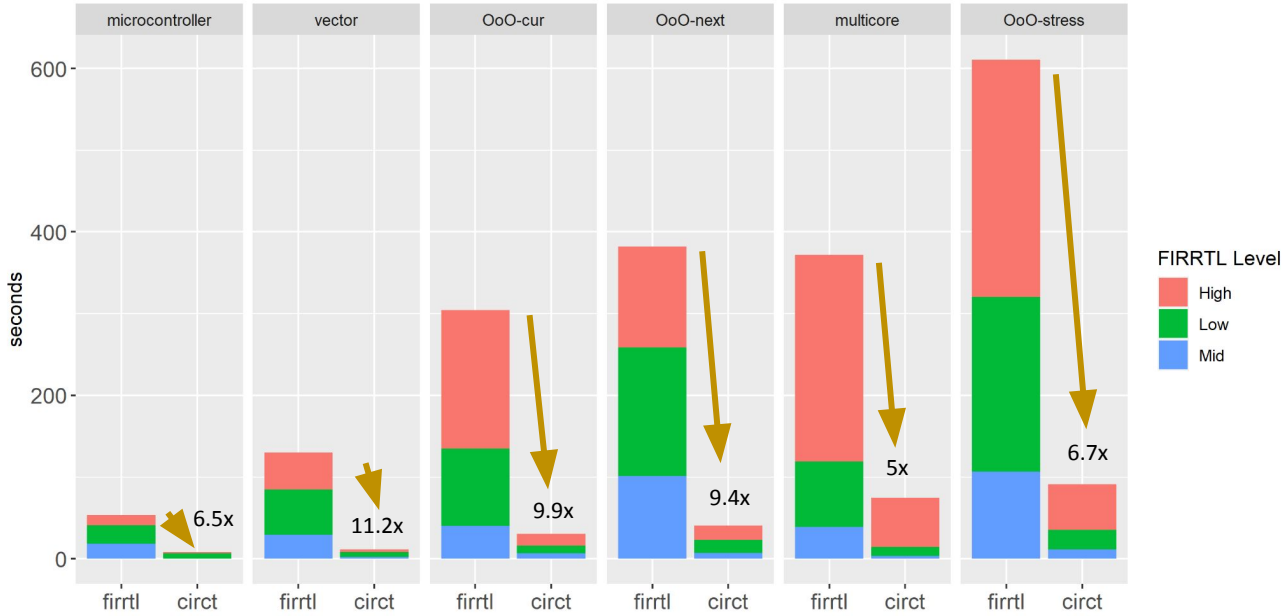
# Progressively lower in passes by mixing dialects




Generate all the metadata in the HW dialect next to the FIRRTL dialect

- Both can coexist in the same module!

# MLIR/CIRCT rapidly accelerates designer iteration cycle




This cuts **>10 minutes** out of iteration cycle for large config of our OoO core!

- Directly drives **increased designer** and **verif productivity**, faster design space exploration

# Chisel/FIRRTL Demo

```
// SPDX-License-Identifier: Apache-2.0
import chisel3._
import chisel3.stage.ChiselStage
object NoSourceInfo {
 implicit val noInfo =
chisel3.internal.sourceinfo.UnlocatableSourceInfo
}
import NoSourceInfo._

/** A totally generic ALU */
class ALUGeneric[A <: Bits with Num[A]]
                (gen: => A, ops: Seq[(A, A) => UInt])
                extends RawModule {
  val a, b = IO(Input(gen))
  val width = a.getWidth
  val f_lo, f_hi = IO(Output(UInt(width.W)))
  val opcode = IO(Input(UInt(util.log2Up(ops.size).W)))
  val result = if (ops.size == 1) {
    ops(0)(a, b).asUInt
  } else {
    val _ops = ops.zipWithIndex.map{ case (f, op) => (op.U, f(a, b)) }
    util.MuxLookup(opcode, 0.U, _ops).asUInt
  }
  f_hi := result(width * 2 - 1, width)
  f_lo := result(width - 1, 0)
}
```

```
object Ops {
  val Ops8 = Seq(
    (a: SInt, b: SInt) => (a % b).asUInt,
    (a: SInt, b: SInt) => (a * b).asUInt,
    (a: SInt, b: SInt) => (a + b).asUInt,
    (a: SInt, b: SInt) => (a - b).asUInt,
    (a: SInt, b: SInt) => (a / b).asUInt,
    (a: SInt, b: SInt) => (b.abs() ## a.abs()).asUInt,
    (a: SInt, b: SInt) => a.max(b).asUInt,
    (a: SInt, b: SInt) => a.min(b).asUInt
  )
}
class ALUGeneric_S8_8op extends ALUGeneric(SInt(8.W),
Ops.Ops8)
println(ChiselStage.emitChirrtl(new ALUGeneric_S8_8op))
```

Scala Source

# FIRRTL for ALU

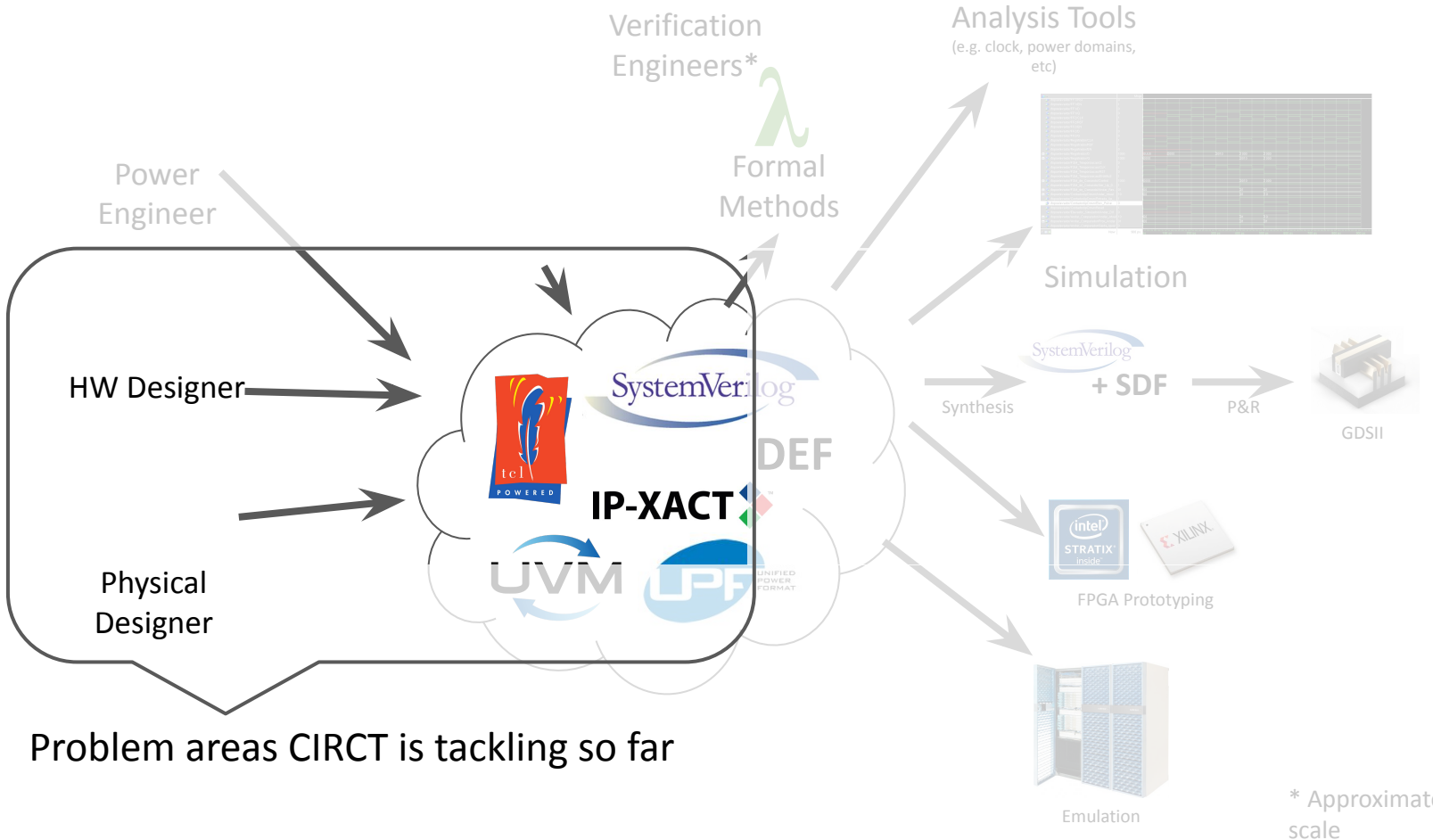
```
circuit ALUGeneric_S8_8op :
  module ALUGeneric_S8_8op :
    input a : SInt<8>
    input b : SInt<8>
    output f_lo : UInt<8>
    output f_hi : UInt<8>
    input opcode : UInt<3>

    node _result_T = rem(a, b)
    node result_defaultx = asUInt(_result_T)
    node _result_T_1 = mul(a, b)
    node _result_T_2 = asUInt(_result_T_1)
    node _result_T_3 = add(a, b)
    node _result_T_4 = tail(_result_T_3, 1)
    node _result_T_5 = asSInt(_result_T_4)
    node _result_T_6 = asUInt(_result_T_5)
    node _result_T_7 = sub(a, b)
    node _result_T_8 = tail(_result_T_7, 1)
    node _result_T_9 = asSInt(_result_T_8)
    node _result_T_10 = asUInt(_result_T_9)
    node _result_T_11 = div(a, b)
    node _result_T_12 = asUInt(_result_T_11)
    node _result_T_13 = lt(b, asSInt(UInt<1>("h0")))
    node _result_T_14 = sub(asSInt(UInt<1>("h0")), b)
    node _result_T_15 = tail(_result_T_14, 1)
    node _result_T_16 = asSInt(_result_T_15)
    node _result_T_17 = mux(_result_T_13, _result_T_16, b)
    node _result_T_18 = lt(a, asSInt(UInt<1>("h0")))
    node _result_T_19 = sub(asSInt(UInt<1>("h0")), a)
    node _result_T_20 = tail(_result_T_19, 1)
    node _result_T_21 = asSInt(_result_T_20)
    node _result_T_22 = mux(_result_T_18, _result_T_21, a)
    node _result_T_23 = cat(_result_T_17, _result_T_22)
    node _result_T_24 = lt(a, b)
    node _result_T_25 = mux(_result_T_24, b, a)
    node _result_T_26 = asUInt(_result_T_25)
    node _result_T_27 = lt(a, b)
    node _result_T_28 = mux(_result_T_27, a, b)
    node _result_T_29 = asUInt(_result_T_28)
    node _result_T_30 = eq(UInt<1>("h1"), opcode)
    node _result_T_31 = mux(_result_T_30, _result_T_2, result_defaultx)
    node _result_T_32 = eq(UInt<2>("h2"), opcode)
    node _result_T_33 = mux(_result_T_32, _result_T_6, _result_T_31)
    node _result_T_34 = eq(UInt<2>("h3"), opcode)
    node _result_T_35 = mux(_result_T_34, _result_T_10, _result_T_33)
    node _result_T_36 = eq(UInt<3>("h4"), opcode)
    node _result_T_37 = mux(_result_T_36, _result_T_12, _result_T_35)
    node _result_T_38 = eq(UInt<3>("h5"), opcode)
    node _result_T_39 = mux(_result_T_38, _result_T_23, _result_T_37)
    node _result_T_40 = eq(UInt<3>("h6"), opcode)
    node _result_T_41 = mux(_result_T_40, _result_T_26, _result_T_39)
    node _result_T_42 = eq(UInt<3>("h7"), opcode)
    node result = mux(_result_T_42, _result_T_29, _result_T_41)
    node _f_hi_T = bits(result, 15, 8)
    f_hi <= _f_hi_T
    node _f_lo_T = bits(result, 7, 0)
    f_lo <= _f_lo_T
```

# Verilog for ALU

```
// Generated by CIRCT sifive/1/10/0-72-gd4dbdb1fd
module ALUGeneric_S8_8op(
  input  [7:0] a,
               b,
  input  [2:0] opcode,
  output [7:0] f_lo,
               f_hi);

  wire [7:0]  result_defaultx;
  wire [15:0] result;   // Mux.scala:80:57

  assign result_defaultx = $signed(a) % $signed(b);
  wire _result_T_24 = $signed(a) < $signed(b);
  wire [7:0][15:0] _GEN = {{{8'h0, _result_T_24 ? a : b}},
                           {{8'h0, _result_T_24 ? b : a}},
                           {{$signed(b) < 8'sh0 ? 8'h0 - b : b, $signed(a) < 8'sh0 ? 8'h0 - a : a}},
                           {{7'h0, $signed({a[7], a}) / $signed({b[7], b})}},
                           {{8'h0, a - b}},
                           {{8'h0, a + b}},
                           {{{8{a[7]}}, a} * {{8{b[7]}}, b}},
                           {{8'h0, result_defaultx}}};     // Mux.scala:80:{57,60}
  assign result = _GEN[opcode]; // Mux.scala:80:57
  assign f_lo = result[7:0];    // Mux.scala:80:57
  assign f_hi = result[15:8];   // Mux.scala:80:57
endmodule
```

# Call To Action

# So far, we are just scratching the surface!




Problem areas CIRCT is tackling so far

# Still early days: many open frontiers yet to be explored!

Standardized dialects for key HW design features:

- SoC assembly (IP-XACT) and power modeling (UPF) dialects

Libraries for key ecosystem features:

- "VLang" - Clang-like Verilog parser
- Formal verification tools, high performance simulators

Physical design "backend" technologies:

- Floor planning, synthesis, place and route algorithms, …
- Technology specific MLIR dialects (e.g. iCE40 FPGA, Skywater PDK, TSMC 5, …)

New design approaches:

- New approaches for MLIR-based high level synthesis (HLS)
- New generator frameworks that expose and utilize these capabilities!
- Integrate first class verification system into the design flow

# Fostering collaboration in HW tools community

Weekly Open Design Meeting:

- Topic: Broad discussions about hardware design tools, challenges, and technologies
- Flexible format: spontaneous discussions, invited talks, discussions about patches, etc

Public Zoom meeting, recorded:

- Meeting notes include videos
- History goes back to May 2020

Both industrial and academic attendees:

- ~20-40 people/week

Everyone is welcome to attend, lurk, or present

- Use / knowledge of CIRCT is not required!

# CIRCT: Lifting hardware development out of the 20th century

The future is built by an open and collaborative community:

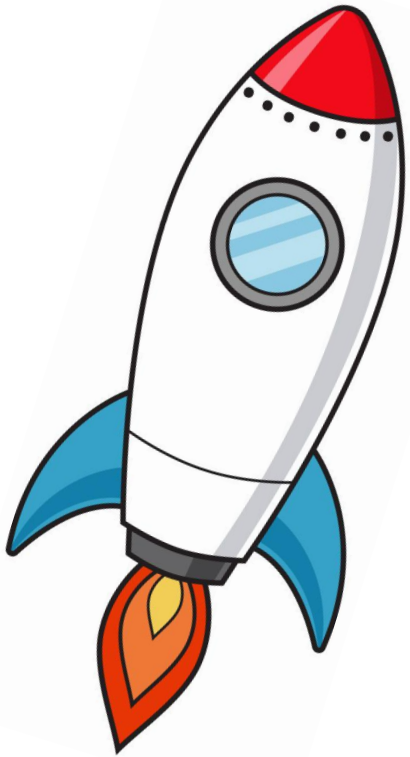
- Pulling together the small group of passionate HW tool engineers

The future is built from large amounts of shared code:

- Extended, improved, and leveraged across the ecosystem in many tools

The future has high quality implementations:

- Fast compile times, great Clang-like error messages, hackable code base



**Join us!**

**https://circt.llvm.org**

# CIRCT wants YOU!

https://circt.llvm.org/

https://github.com/llvm/circt

Discourse discussion board

Weekly discussions Wed. @ 11am PT

**Join us** in disrupting the hardware world!

“If you want to go fast, go alone.
If you want to go far, go together.”

## Credits

Mike Urbach (SiFive) wielded the duct tape for the demo. (and ~~some~~ most of the zip ties.)

All the CIRCT contributors!