SiFive

CIRCT: Lifting hardware development out of the 20th century

Andrew Lenharth and Chris Lattner
LLVM Developer Meeting
November 17, 2021

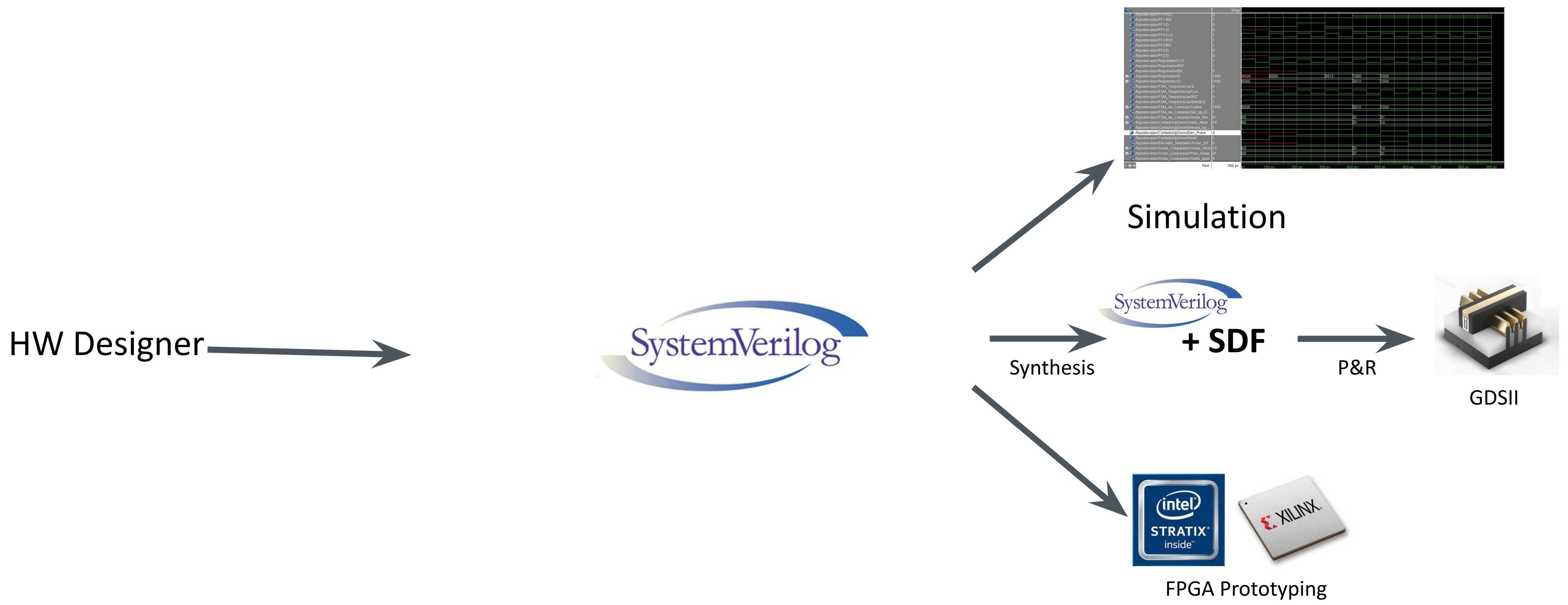
This talk shares the work of many amazing folks in the CIRCT community!

Agenda

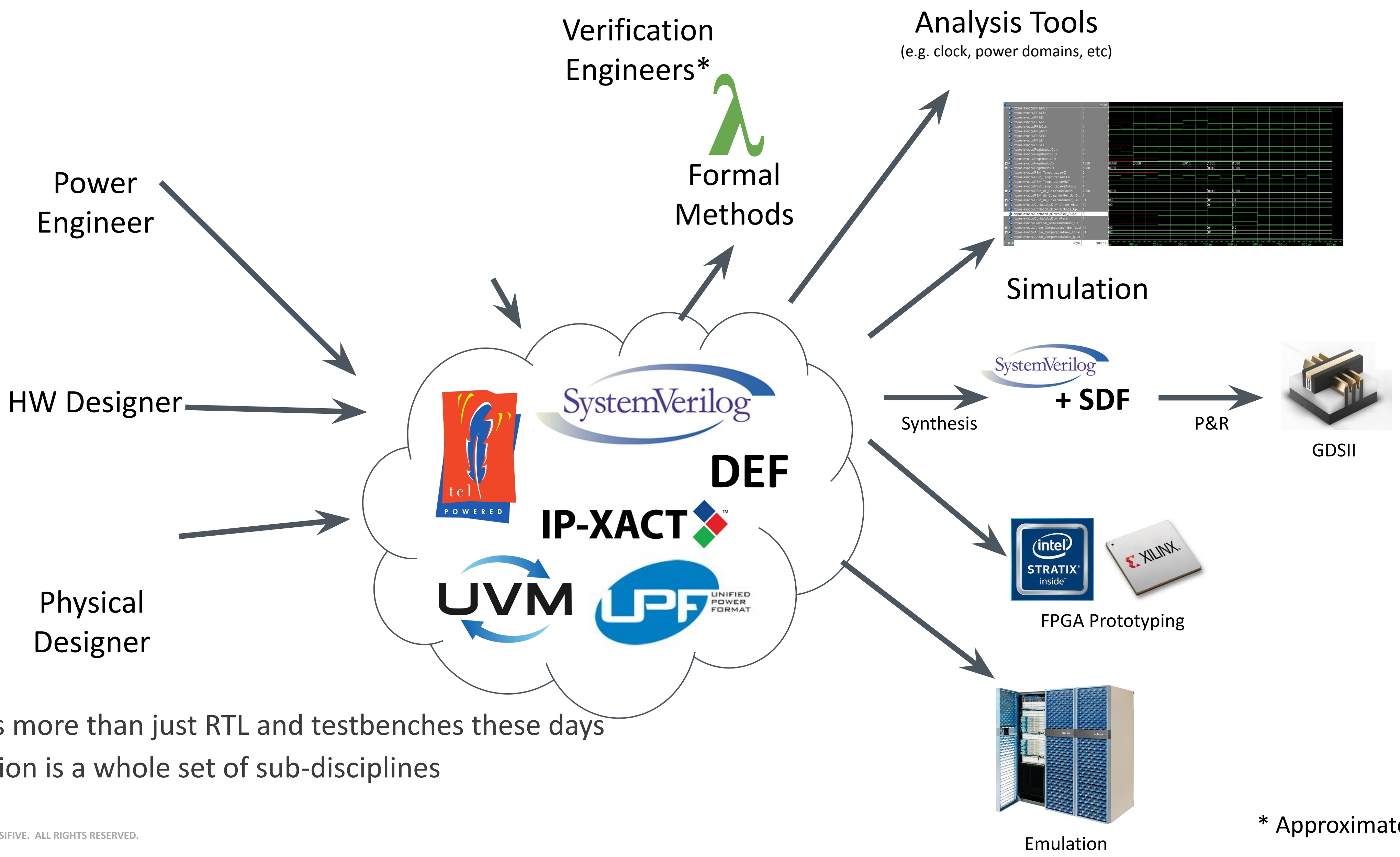
- Intro to Hardware Design
- Emerging Approaches
- Intro to CIRCT
- Hardware vs Software IRs
- Pushing the Boundaries of MLIR
- CIRCT in Production
- Future Directions

Introduction to Classical Hardware Design

Making a chip is easy, right?



Hardware design is a team sport; not one thing

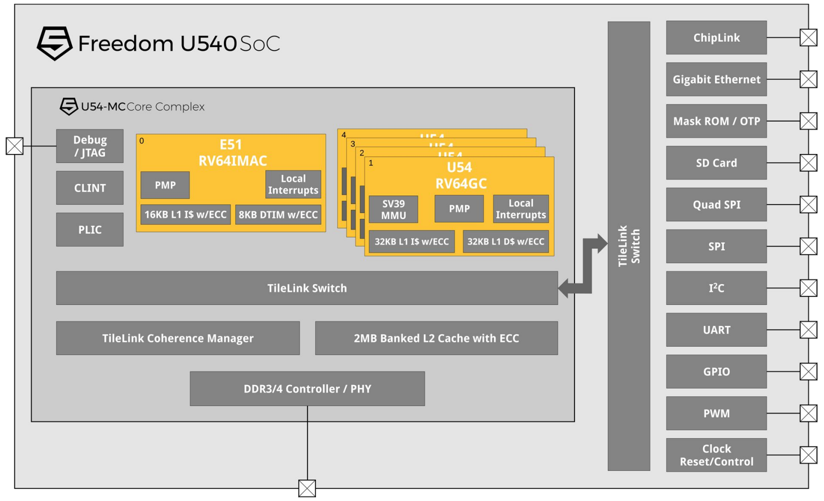


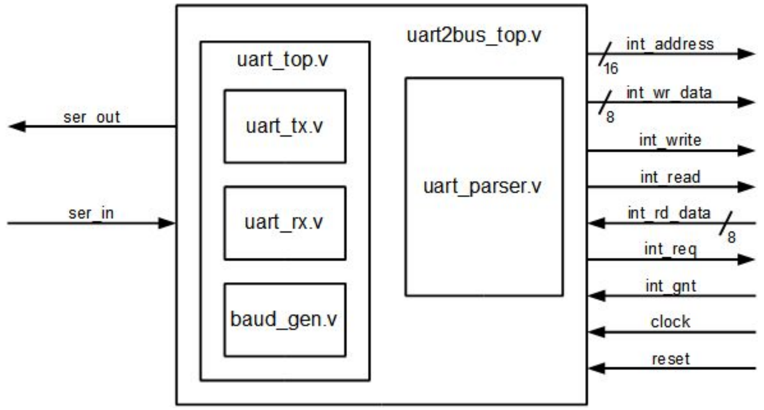
Design is more than just RTL and testbenches these days

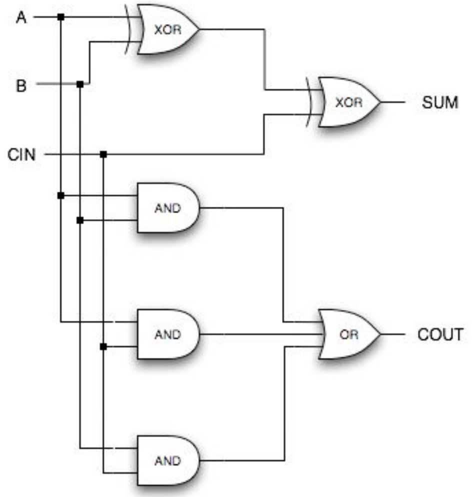
Verification is a whole set of sub-disciplines

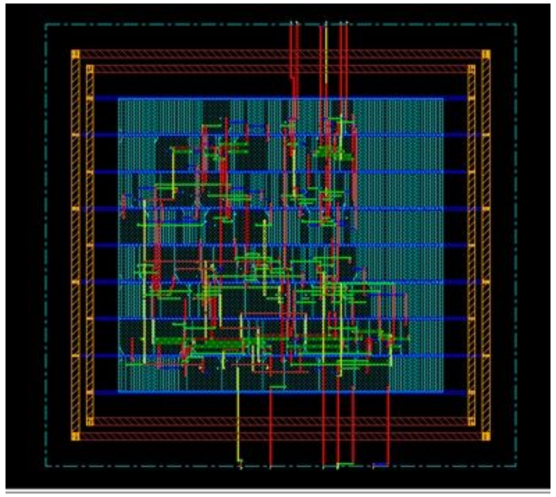
* Approximately to scale

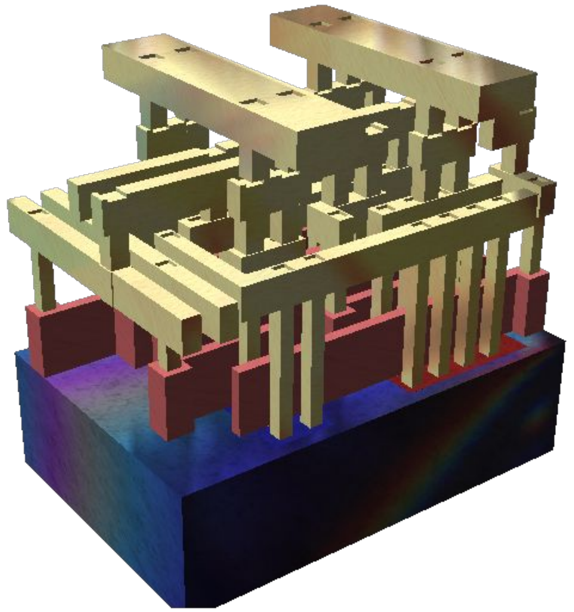
Hardware is inherently Hierarchical and Multi-Level











None of the tools or engineers can reason about the full stack.

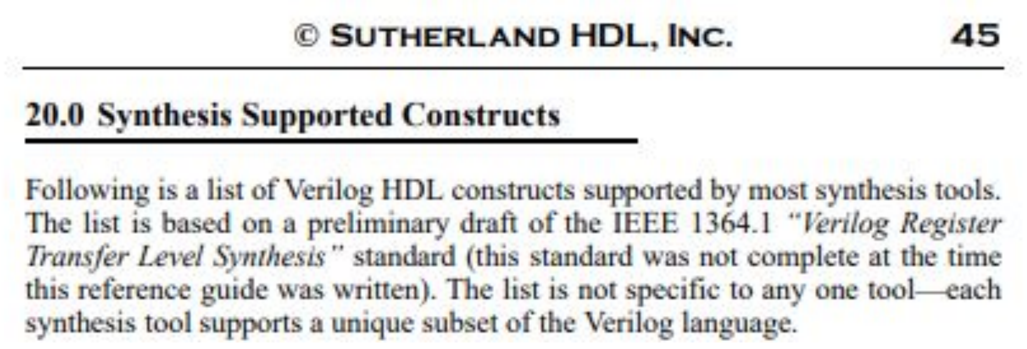
[System]Verilog at the center of things has "issues"

Huge language, with surprising gaps

- e.g. weak metaprogramming

Extremely Verbose

Supported subsets vary by tool



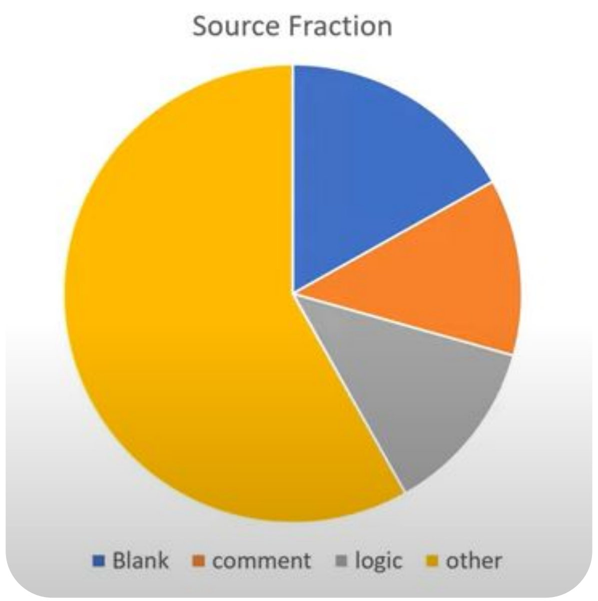
© SUTHERLAND HDL, INC. 45

20.0 Synthesis Supported Constructs

Following is a list of Verilog HDL constructs supported by most synthesis tools. The list is based on a preliminary draft of the IEEE 1364.1 *"Verilog Register Transfer Level Synthesis"* standard (this standard was not complete at the time this reference guide was written). The list is not specific to any one tool—each synthesis tool supports a unique subset of the Verilog language.

System Verilog acts as an ***IR*** between tools

- .. and it doesn't capture power, PD, SoC assembly , ...



	Keywords	Pages of Standard (Lang-only)
System Verilog	250 (+ some)	1315
C++ 20	97 (+ some)	591
Python 3.7	35	170

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!

Free/OSS tools are becoming available in this space!

Proprietary tools are very expensive and not hackable:

- Barrier to personal experimentation and learning

Open alternatives are rising up to fill the gap:

- Part of the larger "Open Hardware" movement!

open source hardware

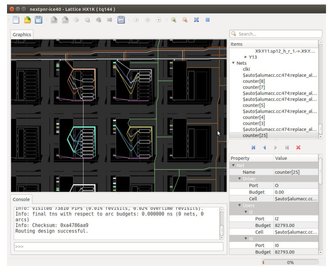
SymbiFlow

Yosys

ICARUS

VERILATOR

OpenROAD



nextpnr

Problem: none are tackling the representational issues!

Emerging Approaches in Hardware Design

Many approaches to raising design abstraction

Research is producing new HW design models and abstraction approaches

CHISEL

LLHD

Magma

Dahlia

Calyx

SpinalHDL

XLS

migen

bluespec

Aetherling

Incorporating language + type system + compiler tech:

… often directly inspired by software

See also: ASPLOS LATTE'21 Workshop

One typical approach: Verilog "Generators"

Challenges with Verilog:

"We need more metaprogramming (parameterization) to enable reuse of modules"

"Verilog has a weak type system, I want to catch bugs at compile time"

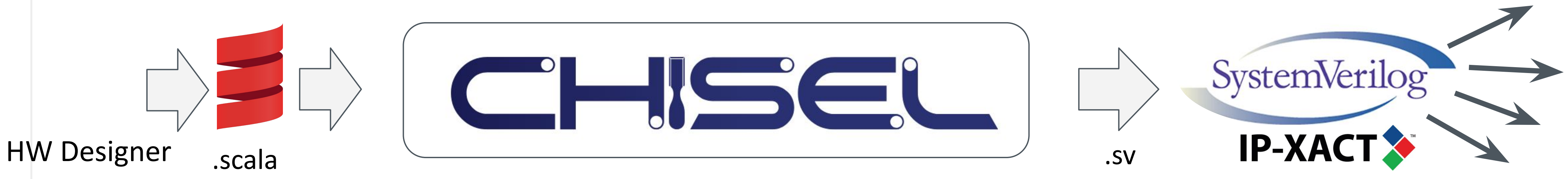
"I want to express complex parameterization in json or another file format"

"I want to be able to build tooling for my designs without having to parse Verilog"

Solution: Don't write Verilog, write a ***program*** to generate Verilog!

- Everything from a Perl script up to a generator *framework*
- Compare to TableGen, PerfectShuffle, Bison, ...

"Chisel" Generator Framework (our running example)



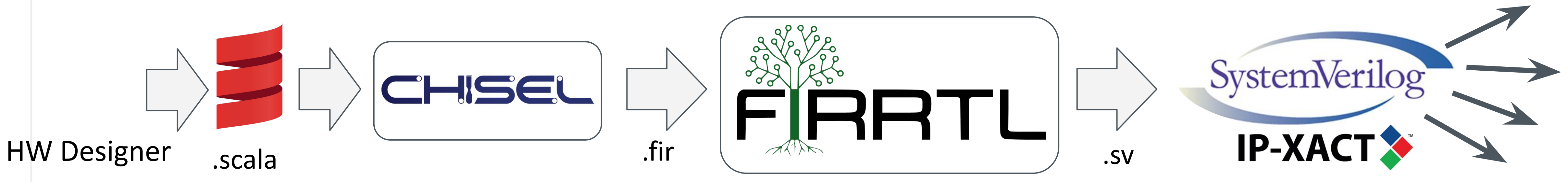
Scala API for generating SystemVerilog:

- 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



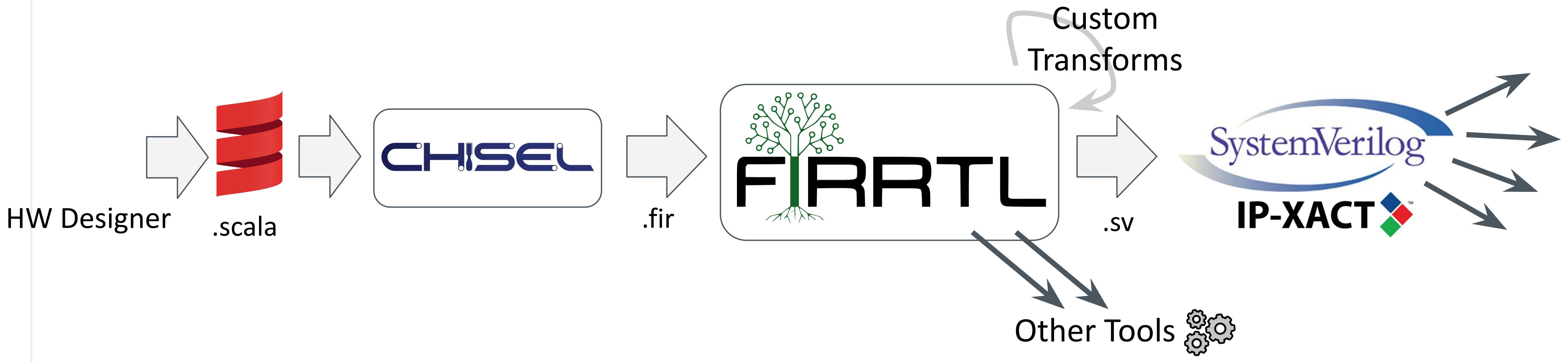
"One Shot" Lowering to Verilog was too complicated, so FIRRTL was introduced:

- 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 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, ...

OSS tools are great, but not as great as they could be!

Wonderful ecosystem of Open Source 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:

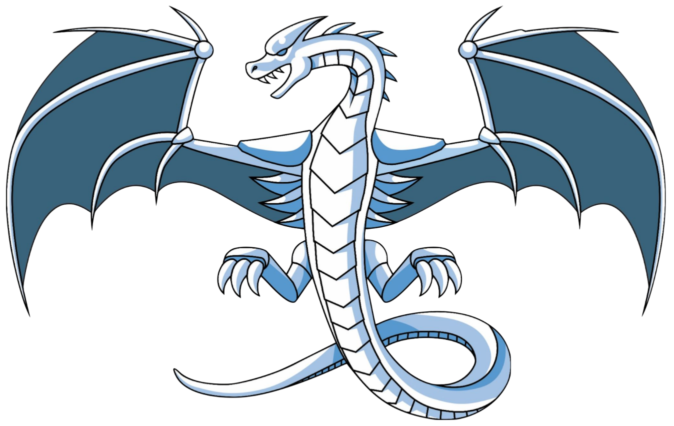
- Little shared code slows progress, each tool is missing features
- Features, quality of results, and user experience trails proprietary tools
- Poor SystemVerilog compatibility harms interoperation

Problem: no one is tackling the IR / representational issues!

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!

Introduction to the LLVM CIRCT Project

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

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 has useful libraries and tech!

CIRCT dialects + implementations:

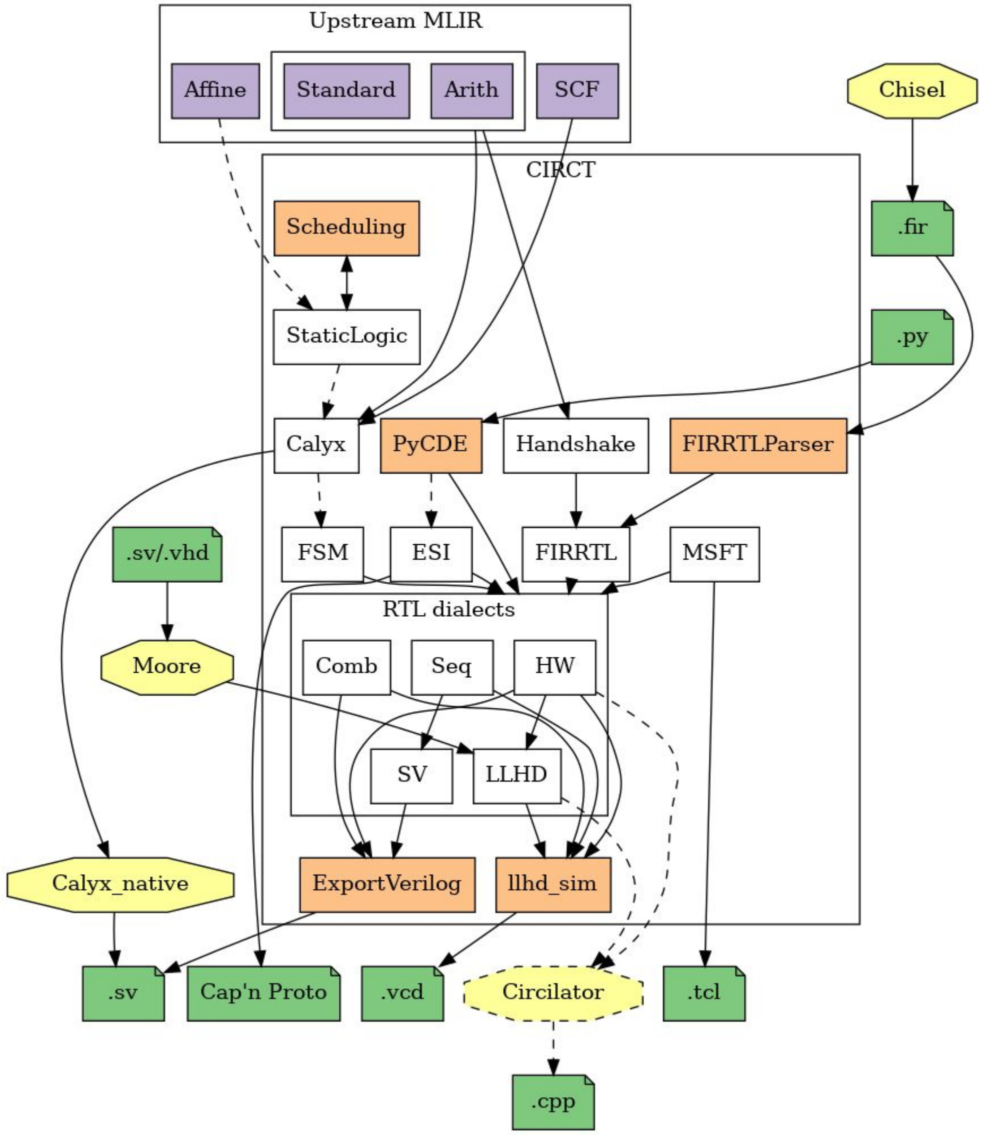
- Full FIRRTL implementation
- Active work on Calyx, LLHD, ESI, Handshake
- Excellent SystemVerilog generation pipeline

Users of CIRCT Libraries:

- Firtool / Chisel at SiFive
- In progress: Magma and Moore projects

Currently focused on "frontend" issues:

- Tools that want to generate [System]Verilog
- Not (yet!) simulation, synthesis, P&R, etc



CIRCT is pushing the limits of MLIR!

Hardware is implicitly parallel and turns into physical devices:

- Hardware IRs have some different needs than Software IRs

CIRCT clients are pushing huge designs:

- Great opportunity to improve MLIR compile time

Verilog is a human readable text file:

- Generating source code has different concerns than generating a .o file

Extending MLIR is fun, let's talk about it!

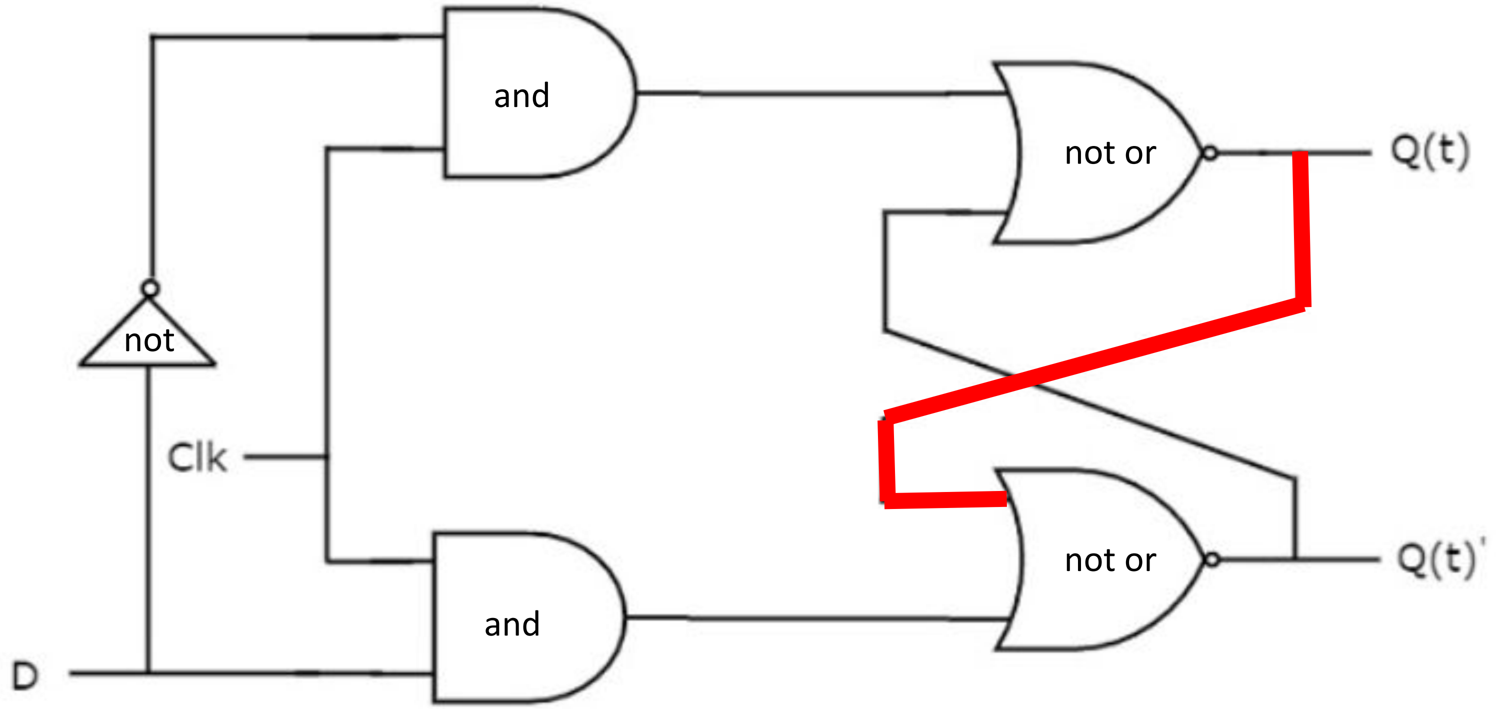
Hardware vs Software IR Challenges

Problem: Logic loops

Circuits can have feedback - DAGs are not enough

- Often hidden in abstractions

Convergence (one hopes) semantics v.s. interleaved sequences



Solution: MLIR now supports Graph Regions

Ops can allow non-dominating uses in regions:

```
hw.module @FF(%clk : i1, %D: i1)
          -> (Q: i1, Qn: i1) {
  %c1 = hw.constant 1 : i1
  %Dn = comb.xor %D, %c1 : i1
  %t1 = comb.and %Dn, %clk : i1
  %t2 = comb.and %D, %clk : i1
  %u1 = comb.or %t1, %o2 : i1
  %u2 = comb.or %t2, %o1 : i1
  %o1 = comb.xor %u1, %c1 : i1
  %o2 = comb.xor %u2, %c1 : i1
  hw.output %o1, %o2 : i1, i1
}
```

Input MLIR

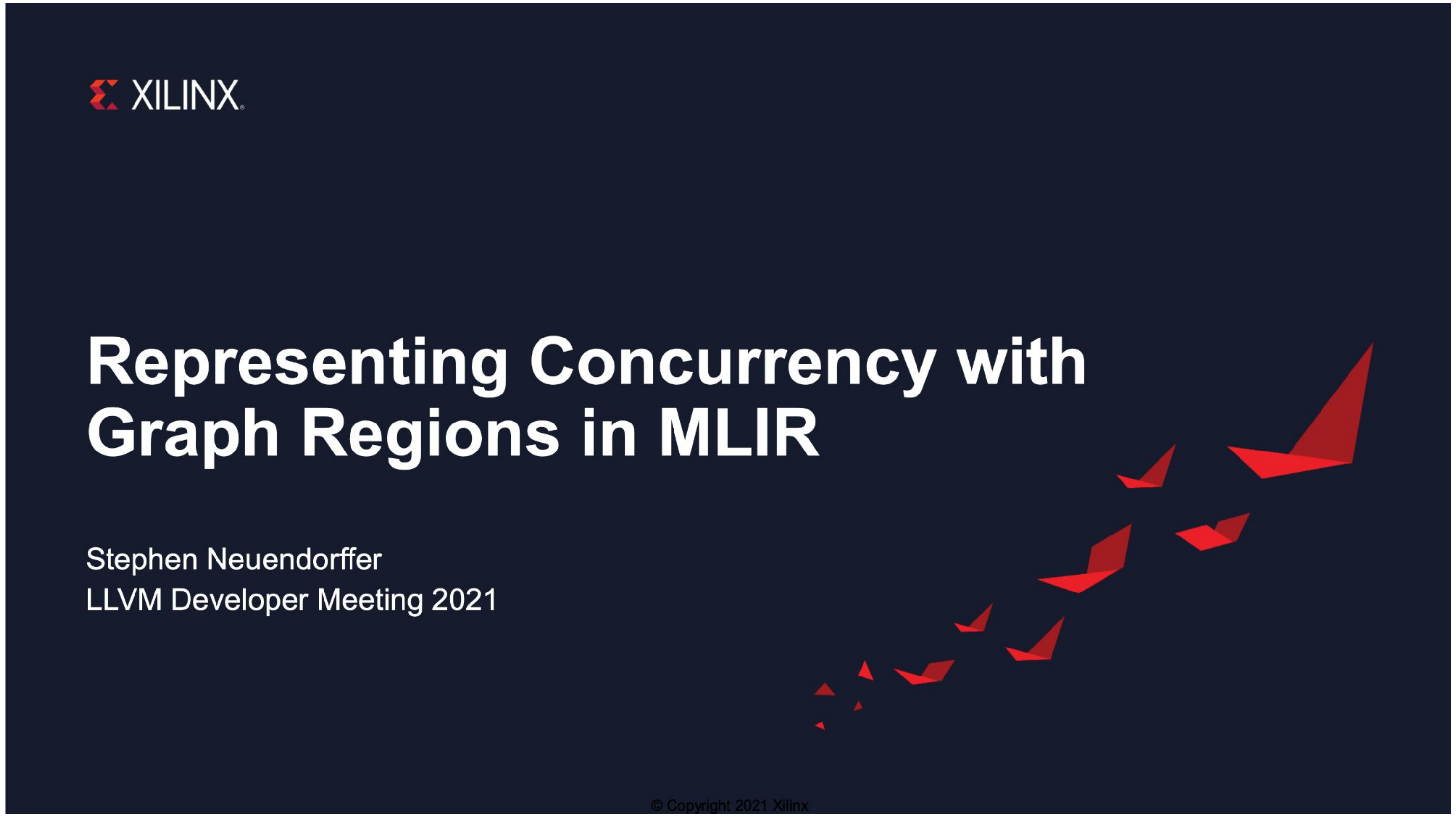
```
module FF(input  clk, D,
          output Q, Qn);

  wire _T;

  wire _T_0 = ~D & clk | ~_T;
  assign _T = D & clk | ~_T_0;
  assign Q = ~_T_0;
  assign Qn = ~_T;
endmodule
```

Output Verilog

This minor extension opens many use-cases!

- General graphs are common in many domains

More info: see Stephen Neuendorffer's Quick Talk

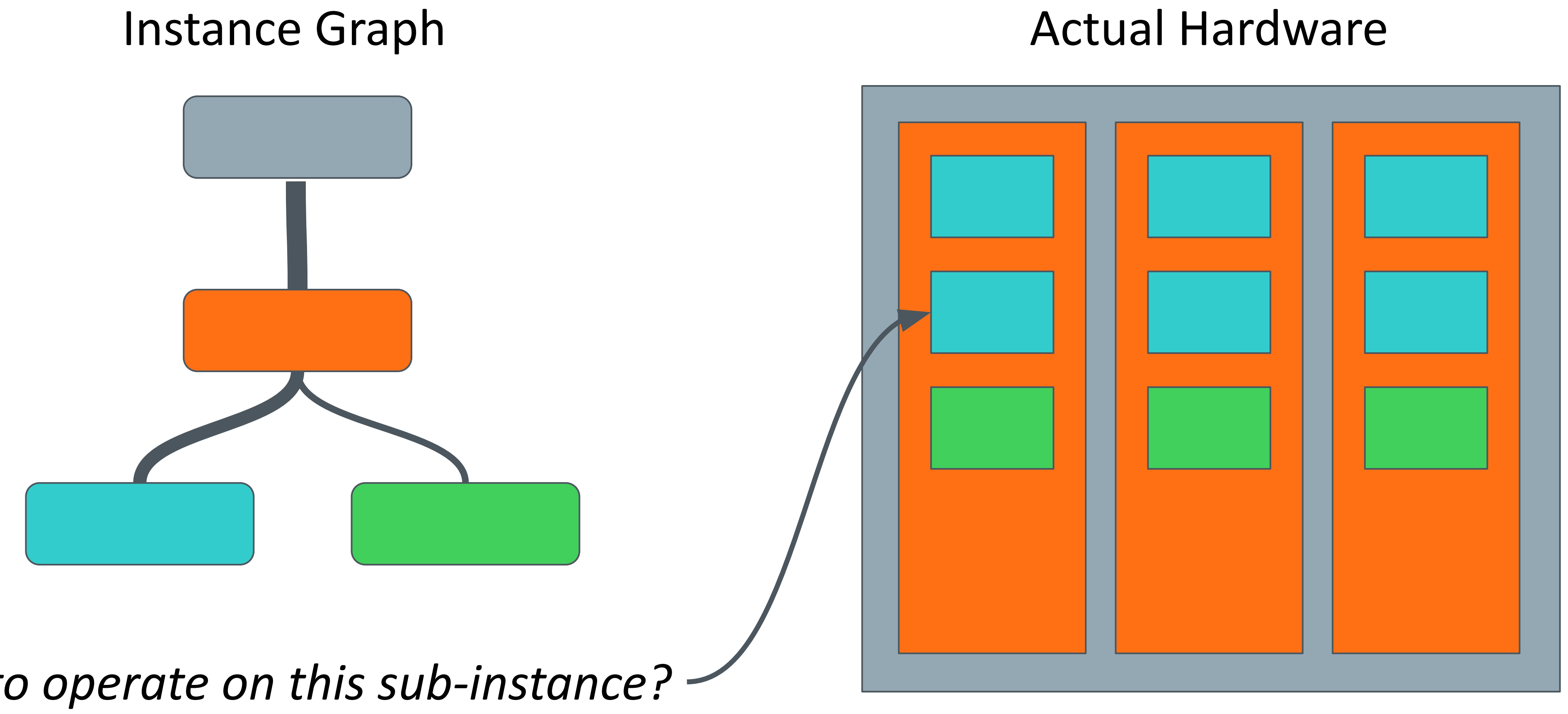

XILINX.

Representing Concurrency with Graph Regions in MLIR

Stephen Neuendorffer
LLVM Developer Meeting 2021

© Copyright 2021 Xilinx

Problem: Instance graph doesn't represent hardware



Instance graph is an abstraction, used to reduce memory usage / compile time

- Physical design and many other things need to break this abstraction

Solution: first-class path specification

Context-sensitivity encapsulated in a path specification

Paths are passed to context-aware ops

```
firrtl.nla @nla [@FooNL::baz, @BazNL::bar, @BarNL::w]

firrtl.module @FooNL() {
  firrtl.instance baz {circt.nonlocal = @nla} @BazNL()
  firrtl.instance baz2 @BazNL
}
firrtl.module @BazNL() {
  firrtl.instance bar {circt.nonlocal = @nla} @BarNL()
  firrtl.instance bar2 @BarNL()
}
firrtl.module @BarNL() {
  %w = firrtl.wire {circt.nonlocal = @nla} : !firrtl.uint<1>
}
```

Problem: Verification of production code

Verification needs to be run on the unmodified code being used for synthesis

- Verification code needs to be "on the side"

This has big implications throughout the language and compilers

```
module synchcounter(
  input clk,reset,
  output [3:0] count );
  reg [3:0] icount;
  always @(posedge clk)
  begin
    if (reset)
      icount <= 4'b0000;
    else
      count <= count + 1;
  end
  assign count = icount;
endmodule
```

```
module Design(
  input clk,reset;
  output [3:0] count;
);
  wire [3:0] out1, out2;
  synccounter m1 (.count(out1), ...);
  synccounter m2 (.count(out2), ...);
  assign count = out1 + out2;
endmodule
```

Design

```
// This might not even be seen
// when compiling the other code.
module testbench();
assign property(Design.m1.icount
             == Design.m2.icount);
…
force Design.m2.clk = `0;
endmodule
```

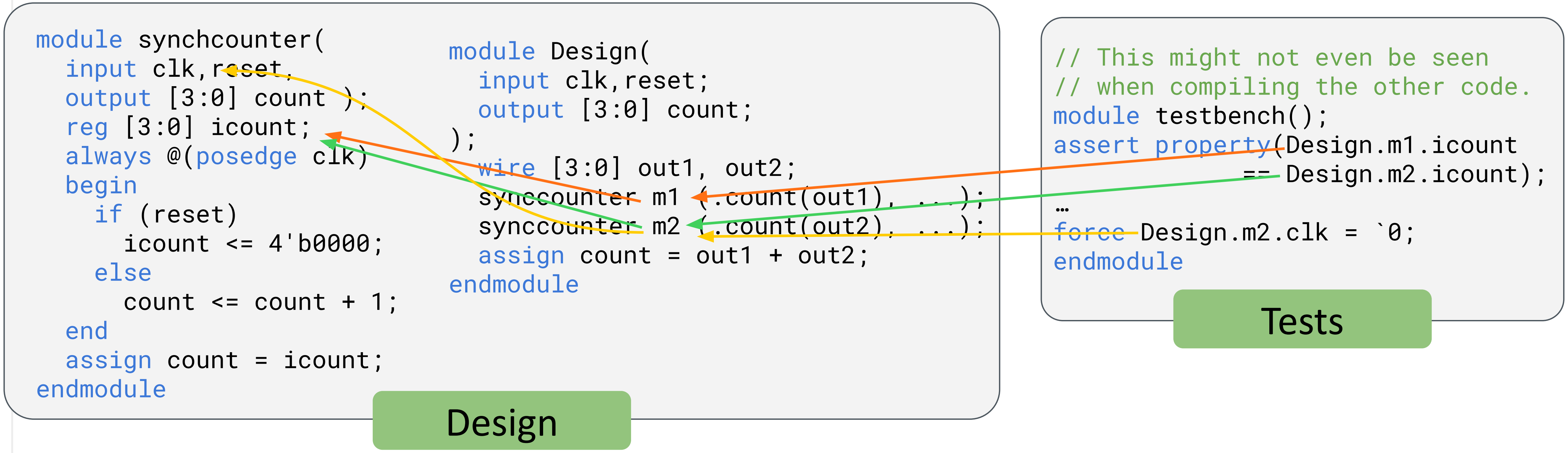
Tests

Problem: Verification of production code

Verification needs to be run on the unmodified code being used for synthesis

- Verification code needs to be “on the side”

This has big implications throughout the language and compilers



```
module synchcounter(
  input clk,reset,
  output [3:0] count );
  reg [3:0] icount;
  always @(posedge clk)
  begin
    if (reset)
      icount <= 4'b0000;
    else
      count <= count + 1;
  end
  assign count = icount;
endmodule
```

```
module Design(
  input clk,reset;
  output [3:0] count;
);
  wire [3:0] out1, out2;
  synccounter m1 (.count(out1), ...);
  synccounter m2 (.count(out2), ...);
  assign count = out1 + out2;
endmodule
```

```
// This might not even be seen
// when compiling the other code.
module testbench();
assert property(Design.m1.icount
             == Design.m2.icount);
…
force Design.m2.clk = `0;
endmodule
```

Solution: Naming and “invisible” instances

Inner symbol tables on each module, optional symbol on many things

- Ports complicate everything (not operation results)

Binds transform an instance to an instance-at-a-distance

- May be in other files or modules

```
sv.bind #hw.innerNameRef<@AB::@b1>
hw.module.extern @EMod(%a: i1, %b: i2)
hw.module @AB(%a: i1, %b: i2) {
  hw.instance "yo" sym @b1 @EMod(a: %a: i1, b: %b: i2) -> () {bind=1}
}
```

MLIR Bind

```
module AB(
  input       a,
  input [1:0] b);
endmodule
```

Verilog

```
bind AB EMod yo (
  .a (a),
  .b (b)
);
```

Verilog

Problem: Need to emit many custom text files

SiFive's builds generates many text files (json, xml, yaml, etc) with design metadata

- … and this is generated from high level FIRRTL IR half way through the pipeline

Metadata file needs to refer to modules, instances, memories etc

- … whose names will change as the compiler lowers the IR!

```
[{
   "module_name": "FIRRTLMem_1_1_0_59_512_1_1_1_0_1_a",
   … //metadata
   "hierarchy": "TOPMod.system.something.another.whatnot.SiFive_magic_ram"
 },
 {
   "module_name": "FIRRTLMem_1_1_0_54_256_1_1_2_0_1_a",
   … //metadata
   "hierarchy": "TOPMod.system.something.different.thingy.frontend.SiFive_nifty_array"
 }]
```

“useful” json

These are emitted instance and module names

Solution: Verbatims with explicit output files

Just allow passes to store the output in the IR

Verbatim nodes, with text substitution, including of MLIR symbols

- late binding of names
- "emit to this filename" attribute on any top-level thing

```
sv.verbatim "[ \"{{0}\", \"{{1}}\" ]" {
  output_file = #hw.output_file<"../foo.json">,symbols = [@Mod1, @Mod2]
}
```

Problem: Pervasive code generators

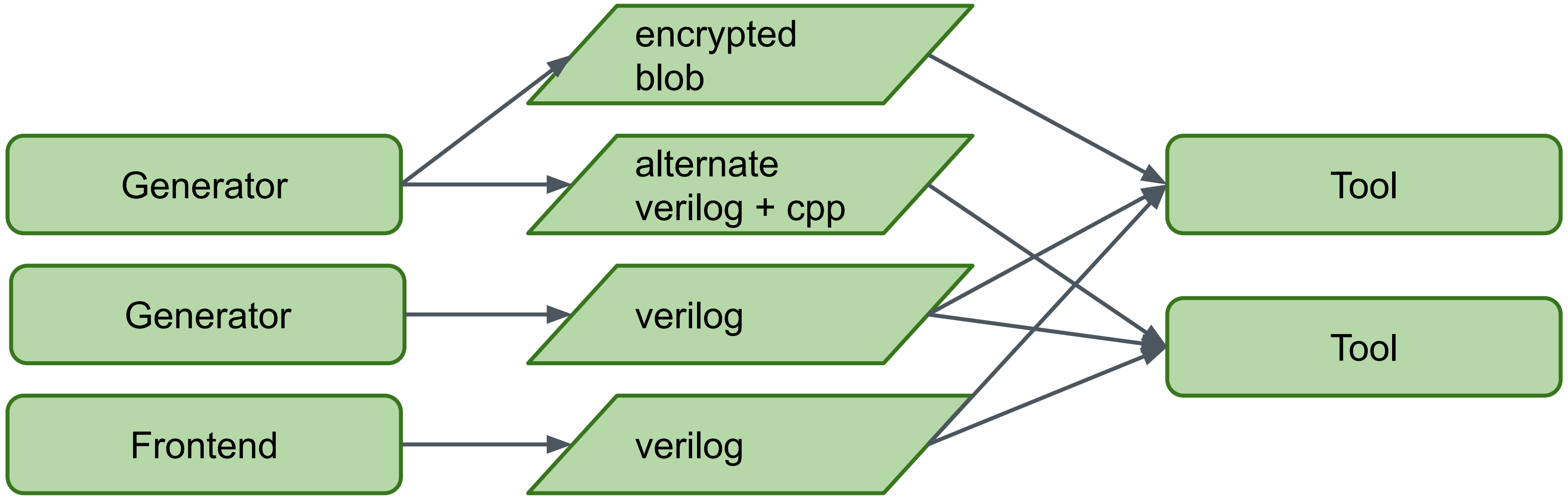
Need single source of truth in face of external generation and transformation

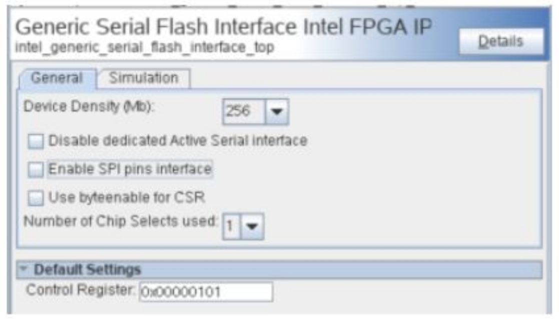
- Avoid lots of brittle duct-tape
- Remove sources of errors

Common approaches are less than ideal

- encode large portions of the design in the build system 😟
- run manually and check in generated output 😠

Better language support for meta-programming might reduce need?




Generic Serial Flash Interface Intel FPGA IP
intel_generic_serial_flash_interface_top

Device Density (Mb): 256
☐ Disable dedicated Active Serial interface
☐ Enable SPI pins interface
☐ Use byteenable for CSR
Number of Chip Selects used: 1
Default Settings
Control Register: 0x00000101

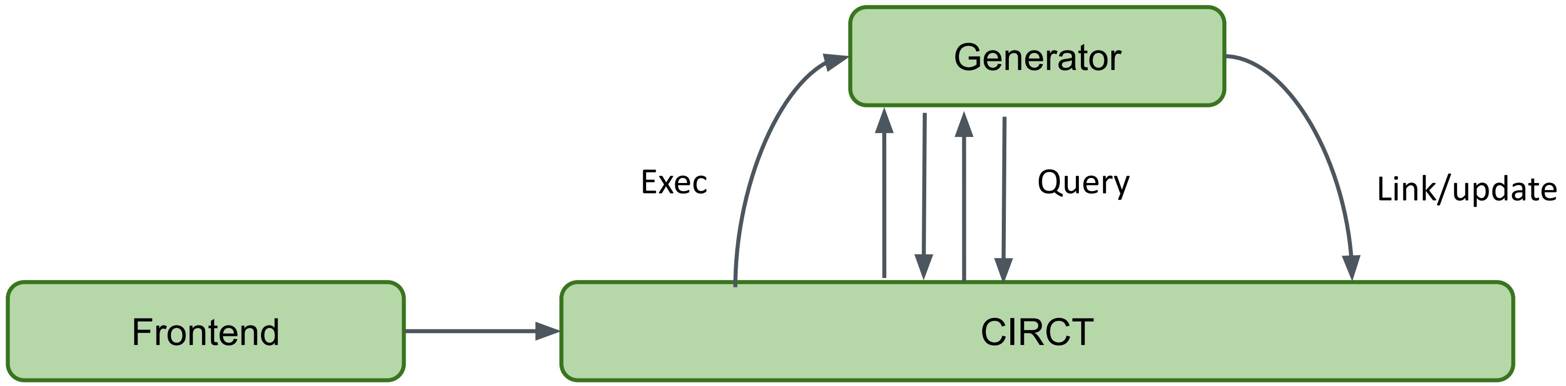
Invoking arbitrary code for "generated modules"

Work-in-progress!

Encode resolvable external components

Looks build system-like. Compiler is forking processes!

- Callout to resolve or refine when needed



```
hw.generator.schema @MEMORY, "Simple-Memory", ["ports", "write_latency", "read_latency"]

hw.module.generated @genmod1, @MEMORY() -> (data: i32, addr: i8, enable: i1) attributes
{write_latency=1, read_latency=1, ports=["read","write"]}
```

Pushing the Boundaries of MLIR

Problem: Source code generation is hard!

We need "high quality" Verilog output

- SiFive sells IP: generated Verilog is a product!
- Generated Verilog is input to test benches

Many challenging issues:

- Generating `ifdef's (#ifdef)
- Generating macro defines / uses
- Comment generation
- Source code formatting, indentation, line wrapping
- Insulating a frontend from these problems

Avoid obviously silly output!

```
// RANDOM may be set to an expression that produces
// a 32-bit random unsigned value.
`ifndef RANDOM
  `define RANDOM {$random}
`endif
...
module TileLinkMonitor_2(
  input        clk, rst, io_in_a_ready, io_in_a_vali
  input [2:0] io_in_a_bits_opcode,
  input [5:0] io_in_a_bits_source,
  ...

 `ifndef SYNTHESIS
    initial begin
      automatic logic [31:0] _T_36;

      `INIT_RANDOM_PROLOG_
      `ifdef RANDOMIZE_REG_INIT
        if (~rst) begin
          automatic logic [31:0] _T_46 = `RANDOM;
          a_first_counter = _T_46[2:0];
        end
        _T_36 = `RANDOM;
```

SystemVerilog dialect represents textual constructs

IR directly models source level concerns:

- Multiple forms of `ifdefs`
- "Verbatim" expressions and statements
- Language concepts like initial/always blocks

PrettifyVerilog pass "optimizes" text:

- Merging redundant if's/ifdefs
- Sinking logic into narrowest scope

Front-ends get full control, less complexity:

- Full power of [System]Verilog available
- Tedium / complexity delegated to CIRCT

```
sv.verbatim "// RANDOM may be set to an expression
sv.ifndef "RANDOM" {
  sv.verbatim "`define RANDOM {$random}"
}

hw.module @TileLinkMonitor_2(%clk: i1, %rst: i1,
    %io_in_a_ready: i1, %io_in_a_valid: i1, ... {
...

 sv.ifndef "SYNTHESIS" {
   sv.initial {
     sv.verbatim "`INIT_RANDOM_PROLOG_"
     sv.ifdef.procedural "RANDOMIZE_REG_INIT" {
       %true = hw.constant true
       %6464 = comb.xor %reset, %true : i1
       sv.if %6464 {
         %RANDOM_3973 = sv.verbatim.expr.se "`RANDO
         %10244 = comb.extract %RANDOM_3973 from 0
         sv.bpassign %rob_state, %10244 : i3
         ...
     }
   }
 }
}
```

Logic IR is easy to analyze; Exporter handles syntax

comb/seq dialects handle logic expressions

- comb: combinational logic (add, mux, etc)
- seq: sequential logic (reg, mem, etc)

Easy to analyze, transform, peephole etc:

- normal SSA data flow graph
- conceptually similar to `std` dialect or LLVM IR

Syntax issues managed by ExportVerilog:

- Precedence, indentation, wrapping, etc
- Temporary variable insertion
  - Multiple-use expression
  - Long line breaking
  - Verilog exprs aren't always composable

```
hw.module @arith(%a: i8, %b: i8) -> (b: i4) {
  %1 = comb.add %a, %b : i8
  %2 = comb.sub %a, %b : i8
  %3 = comb.mul %1, %1, %2 : i8
  %4 = comb.extract %3 from 2 : (i8) -> i4
  hw.output %4 : i4
}
```

Input MLIR

```
module arith(
  input  [7:0] a, b,
  output [3:0] b_0);

  // Temporary due to multiple uses.
  wire [7:0] _T = a + b;

  // Parentheses inserted.
  wire [7:0] _T_0 = _T * _T * (a - b);

  // Extracts may only be from a temporary.
  assign b_0 = _T_0[5:2];
endmodule
```

Generated Verilog

Challenge: Emitting Verilog for Diverse+fragmented ecosystem

Many tools consume [System]Verilog, but no consistent definition of what that means!

- Products need to be able to decide what is right for their users

Several axes of control:

- Language subsets: e.g. "just Verilog", verboten language features
- Conceptual: Verilog "Interfaces", vs structured ports, vs flattened ports
- Formatting: indentation, line wrapping, other "clang-format" sorts of issues

Approach: encapsulate complexity into shared infra

- Serve many different clients
- Let frontend or end-user decide what they want

Solution: `circt::LoweringOptions` framework

Key design points:

- Serialized into a string - like a "target triple"
- String stored as an attribute on the builtin.module
- `circt::LoweringOptions` class provides type-safe API
- Lowering passes eliminates unsupported things

```
module attributes {circt.loweringOptions = "disallowPackedArrays"} {

hw.module @array_create_get_default(%arg0: i8, ...) {
  sv.initial {
    %three_array = hw.array_create %arg2, %arg1, %arg0 : i8
    %2 = hw.array_get %three_array[%sel] : !hw.array<3xi8>

    %cond = comb.icmp eq %2, %arg2 : i8
    sv.if %cond {
```

```
struct LoweringOptions {
  LoweringOptions(mlir::ModuleOp m);
  void parse(StringRef options,
             ErrorHandlerT callback);
  void setAsAttribute(mlir::ModuleOp m);

  /// If true, eliminate packed arrays for tools
  /// that don't support them (e.g. Yosys).
  bool disallowPackedArrays = false;

  /// If true, do not emit SystemVerilog locally
  /// scoped "automatic" or logic declarations -
  /// emit top level wire and reg's instead.
  bool disallowLocalVariables = false;

  /// This is the target width of lines in an
  /// emitted Verilog source file in columns.
  unsigned emittedLineLength;
  ...
```

Example: Lowering packed / 2D arrays

```
$ firtool circt/test/Dialect/SV/hw-legalize-modules-packed-arrays.mlir -verilog \
```

```
initial begin
  automatic logic [2:0][7:0] _T =
     {{arg2}, {arg1}, {arg0}};

  if (_T[sel] == arg2)
    ...
```

```
-lowering-options=""
```

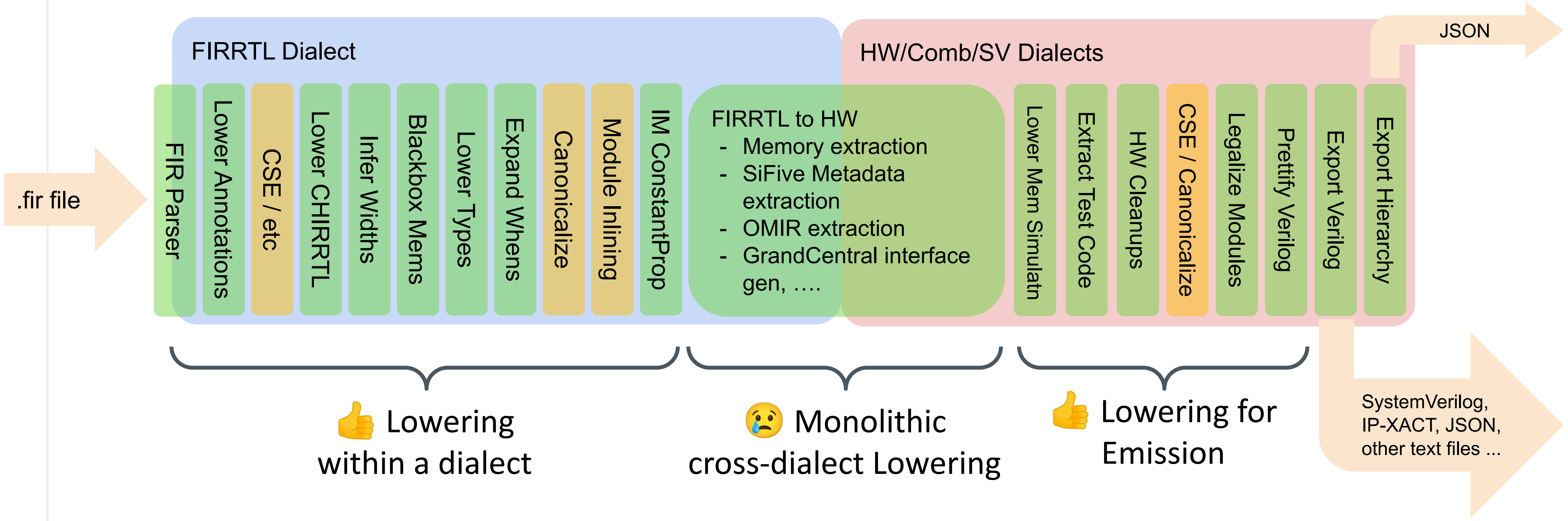
```
initial begin
  casez (sel)
    2'b00: casez_tmp = arg0;
    2'b01: casez_tmp = arg1;
    2'b10: casez_tmp = arg2;
    default: casez_tmp = 8'bx;
  endcase

  if (casez_tmp == arg2)
    ...
```

```
-lowering-options="disallowPackedArrays"
```

Tools and users have full control!

- Complexity encapsulated into circt, not in front-ends / generators
- Testing tools can override this on the command line

Problem: Monolithic lowering passes

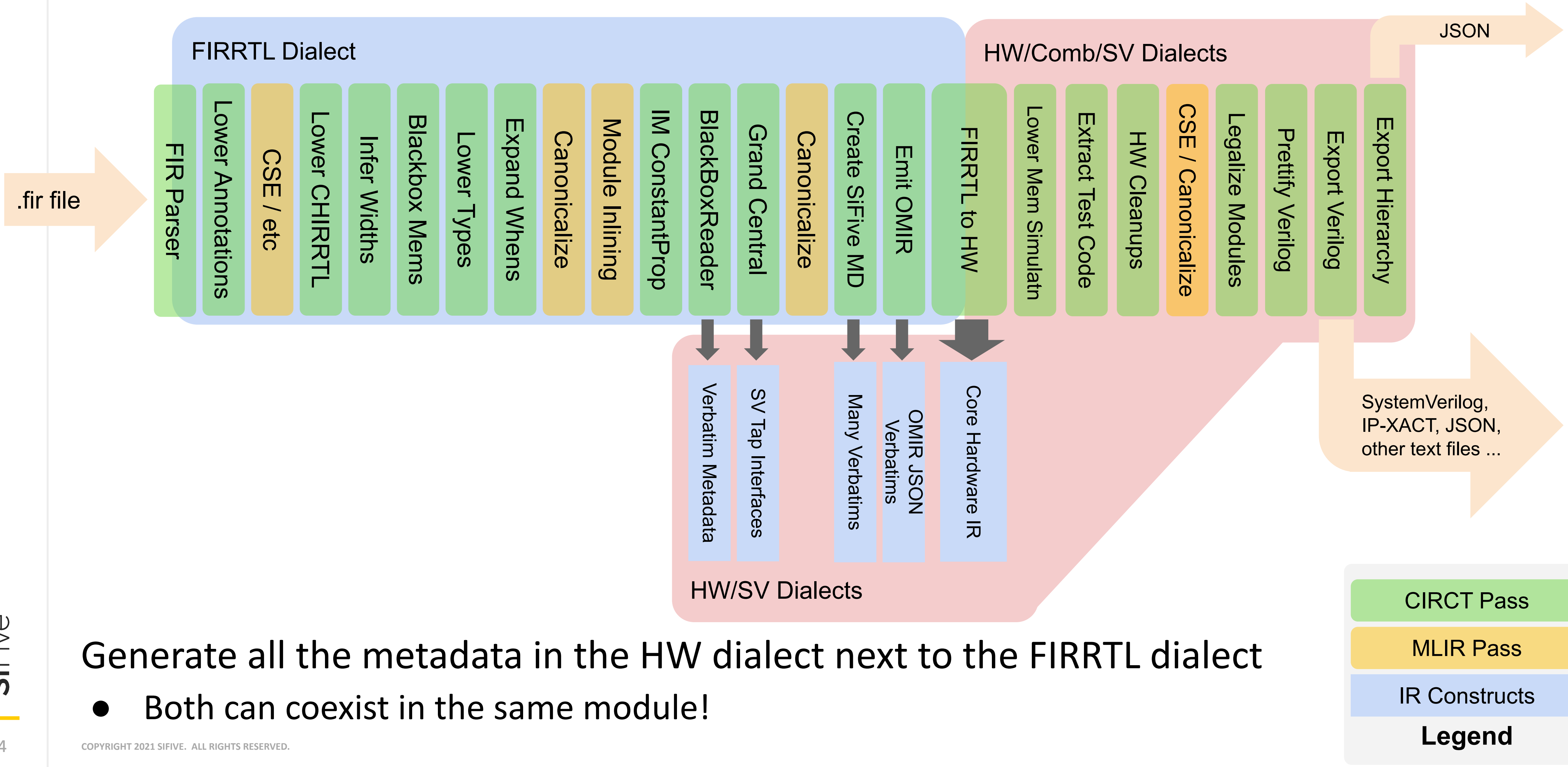


Problems with monolithic lowering passes

- Too many concerns handles in one codebase
- Difficult to extend and scale for new problem domains
- Difficult to test each component

Legend
CIRCT Pass
MLIR Pass
IR Constructs

Solution: 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!

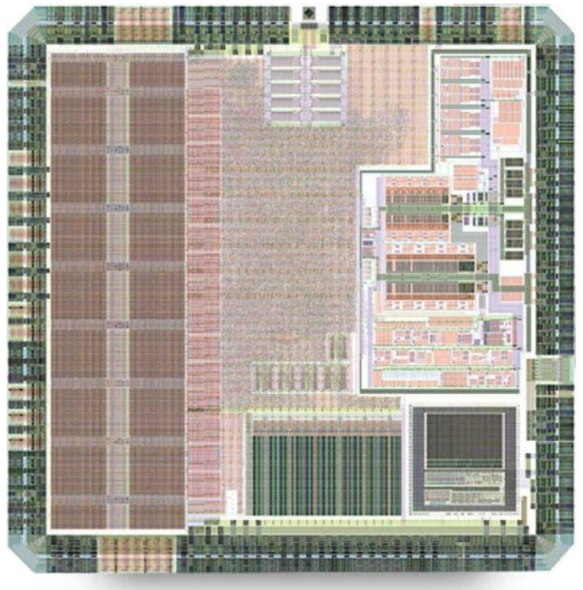
Challenge: Very large designs

Hardware is "big" - many billions of gates

Designs keep getting larger as we push towards higher complexity designs

- Common example: **500MB** of .fir file input generating **290MB** of SystemVerilog
- >5GB designs are not uncommon

How do we continuously accelerate productivity?



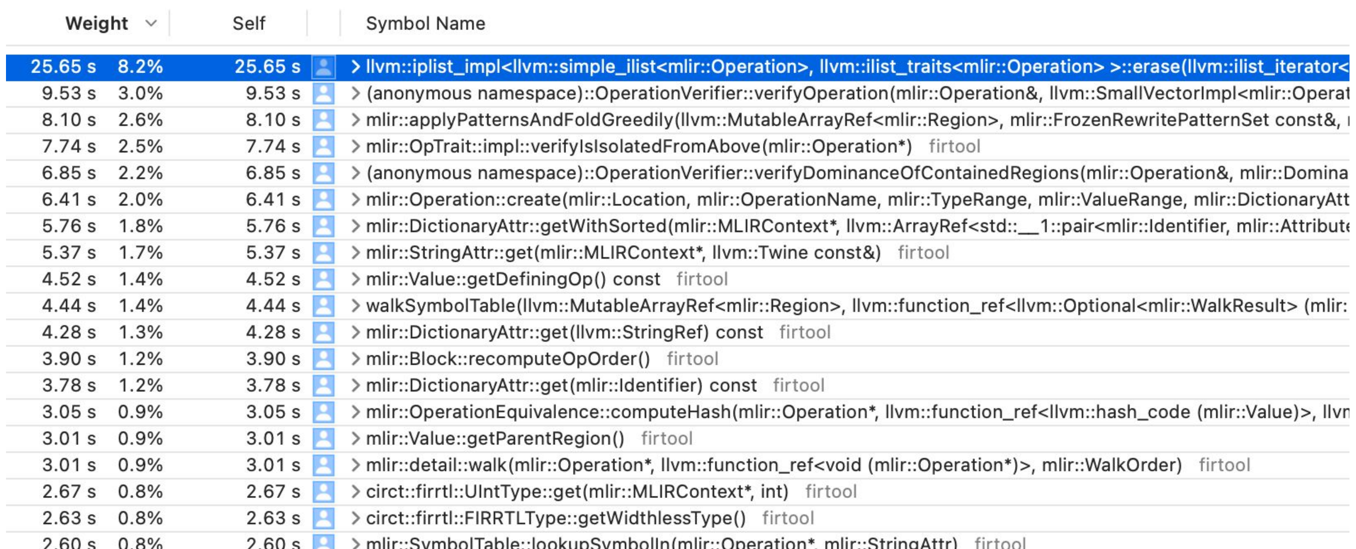
~Solution: Profile, Speed up and Parallelize all the things

Improve MLIR itself:

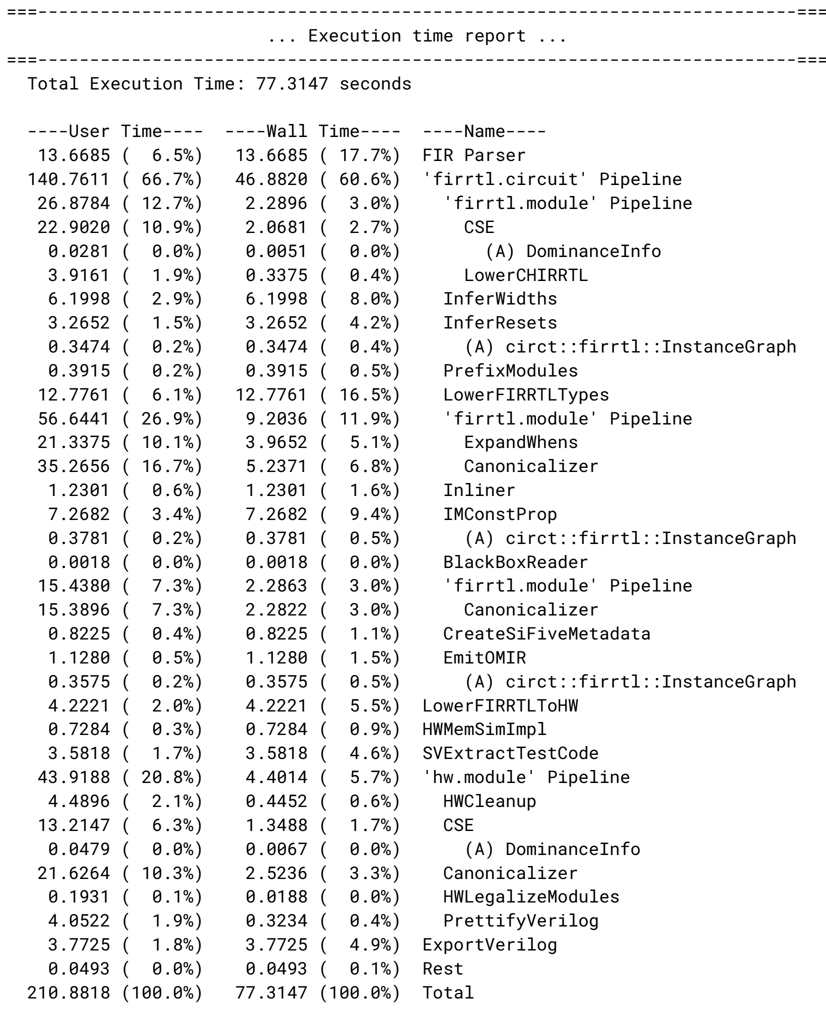
- Data structures, memory usage, implementation details
- Core algorithms like the verifier

Parallelize individual components of the stack:

- A few passes are trivial parallel "function passes"
- Most passes require ad-hoc parallel for loops etc
- Parallelizing the parser and printer was a huge win



Weight		Self	Symbol Name
25.65 s	8.2%	25.65 s	llvm::iplist_impl<llvm::simple_ilist<mlir::Operation>, llvm::ilist_traits<mlir::Operation> >::erase(llvm::ilist_iterator<
9.53 s	3.0%	9.53 s	(anonymous namespace)::OperationVerifier::verifyOperation(mlir::Operation&, llvm::SmallVectorImpl<mlir::Operat
8.10 s	2.6%	8.10 s	mlir::applyPatternsAndFoldGreedily(llvm::MutableArrayRef<mlir::Region>, mlir::FrozenRewritePatternSet const&,
7.74 s	2.5%	7.74 s	mlir::OpTrait::impl::verifyIsIsolatedFromAbove(mlir::Operation*) firtool
6.85 s	2.2%	6.85 s	(anonymous namespace)::OperationVerifier::verifyDominanceOfContainedRegions(mlir::Operation&, mlir::Domina
6.41 s	2.0%	6.41 s	mlir::Operation::create(mlir::Location, mlir::OperationName, mlir::TypeRange, mlir::ValueRange, mlir::DictionaryAtt
5.76 s	1.8%	5.76 s	mlir::DictionaryAttr::getWithSorted(mlir::MLIRContext*, llvm::ArrayRef<std::__1::pair<mlir::Identifier, mlir::Attribute
5.37 s	1.7%	5.37 s	mlir::StringAttr::get(mlir::MLIRContext*, llvm::Twine const&) firtool
4.52 s	1.4%	4.52 s	mlir::Value::getDefiningOp() const firtool
4.44 s	1.4%	4.44 s	walkSymbolTable(llvm::MutableArrayRef<mlir::Region>, llvm::function_ref<llvm::Optional<mlir::WalkResult> (mlir:
4.28 s	1.3%	4.28 s	mlir::DictionaryAttr::get(llvm::StringRef) const firtool
3.90 s	1.2%	3.90 s	mlir::Block::recomputeOpOrder() firtool
3.78 s	1.2%	3.78 s	mlir::DictionaryAttr::get(mlir::Identifier) const firtool
3.05 s	0.9%	3.05 s	mlir::OperationEquivalence::computeHash(mlir::Operation*, llvm::function_ref<llvm::hash_code (mlir::Value)>, llvm
3.01 s	0.9%	3.01 s	mlir::Value::getParentRegion() firtool
3.01 s	0.9%	3.01 s	mlir::detail::walk(mlir::Operation*, llvm::function_ref<void (mlir::Operation*)>, mlir::WalkOrder) firtool
2.67 s	0.8%	2.67 s	circt::firrtl::UIntType::get(mlir::MLIRContext*, int) firtool
2.63 s	0.8%	2.63 s	circt::firrtl::FIRRTLType::getWidthlessType() firtool
2.60 s	0.8%	2.60 s	mlir::SymbolTable::lookupSymbolIn(mlir::Operation*, mlir::StringAttr) firtool



```
===-------------------------------------------------------------------------===
                        ... Execution time report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 77.3147 seconds

  ----User Time----  ----Wall Time----  ----Name----
   13.6685 (  6.5%)   13.6685 ( 17.7%)  FIR Parser
  140.7611 ( 66.7%)   46.8820 ( 60.6%)  'firrtl.circuit' Pipeline
   26.8784 ( 12.7%)    2.2896 (  3.0%)    'firrtl.module' Pipeline
   22.9020 ( 10.9%)    2.0681 (  2.7%)      CSE
    0.0281 (  0.0%)    0.0051 (  0.0%)        (A) DominanceInfo
    3.9161 (  1.9%)    0.3375 (  0.4%)      LowerCHIRRTL
    6.1998 (  2.9%)    6.1998 (  8.0%)    InferWidths
    3.2652 (  1.5%)    3.2652 (  4.2%)    InferResets
    0.3474 (  0.2%)    0.3474 (  0.4%)      (A) circt::firrtl::InstanceGraph
    0.3915 (  0.2%)    0.3915 (  0.5%)    PrefixModules
   12.7761 (  6.1%)   12.7761 ( 16.5%)    LowerFIRRTLTypes
   56.6441 ( 26.9%)    9.2036 ( 11.9%)    'firrtl.module' Pipeline
   21.3375 ( 10.1%)    3.9652 (  5.1%)      ExpandWhens
   35.2656 ( 16.7%)    5.2371 (  6.8%)      Canonicalizer
    1.2301 (  0.6%)    1.2301 (  1.6%)    Inliner
    7.2682 (  3.4%)    7.2682 (  9.4%)    IMConstProp
    0.3781 (  0.2%)    0.3781 (  0.5%)      (A) circt::firrtl::InstanceGraph
    0.0018 (  0.0%)    0.0018 (  0.0%)    BlackBoxReader
   15.4380 (  7.3%)    2.2863 (  3.0%)    'firrtl.module' Pipeline
   15.3896 (  7.3%)    2.2822 (  3.0%)      Canonicalizer
    0.8225 (  0.4%)    0.8225 (  1.1%)    CreateSiFiveMetadata
    1.1280 (  0.5%)    1.1280 (  1.5%)    EmitOMIR
    0.3575 (  0.2%)    0.3575 (  0.5%)      (A) circt::firrtl::InstanceGraph
    4.2221 (  2.0%)    4.2221 (  5.5%)  LowerFIRRTLToHW
    0.7284 (  0.3%)    0.7284 (  0.9%)  HWMemSimImpl
    3.5818 (  1.7%)    3.5818 (  4.6%)  SVExtractTestCode
   43.9188 ( 20.8%)    4.4014 (  5.7%)  'hw.module' Pipeline
    4.4896 (  2.1%)    0.4452 (  0.6%)    HWCleanup
   13.2147 (  6.3%)    1.3488 (  1.7%)    CSE
    0.0479 (  0.0%)    0.0067 (  0.0%)      (A) DominanceInfo
   21.6264 ( 10.3%)    2.5236 (  3.3%)    Canonicalizer
    0.1931 (  0.1%)    0.0188 (  0.0%)    HWLegalizeModules
    4.0522 (  1.9%)    0.3234 (  0.4%)    PrettifyVerilog
    3.7725 (  1.8%)    3.7725 (  4.9%)  ExportVerilog
    0.0493 (  0.0%)    0.0493 (  0.1%)  Rest
  210.8818 (100.0%)   77.3147 (100.0%)  Total
```

This is good, but we are a long ways away from "great"

`llvm/Support/Threading.h` isn't great:

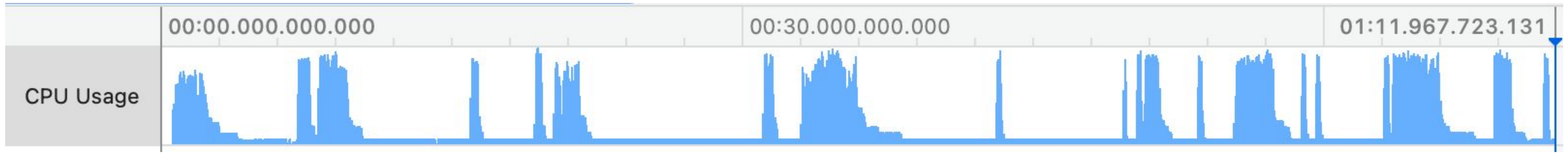
- High constant factor costs
- No support for hierarchical, graph-based parallelism, or future/promise-based approaches
- Few concurrent data structures

Some algorithms are difficult to parallelize:

- e.g. lattice updates in interprocedural constant propagation

Further research is required for 100x improvements:

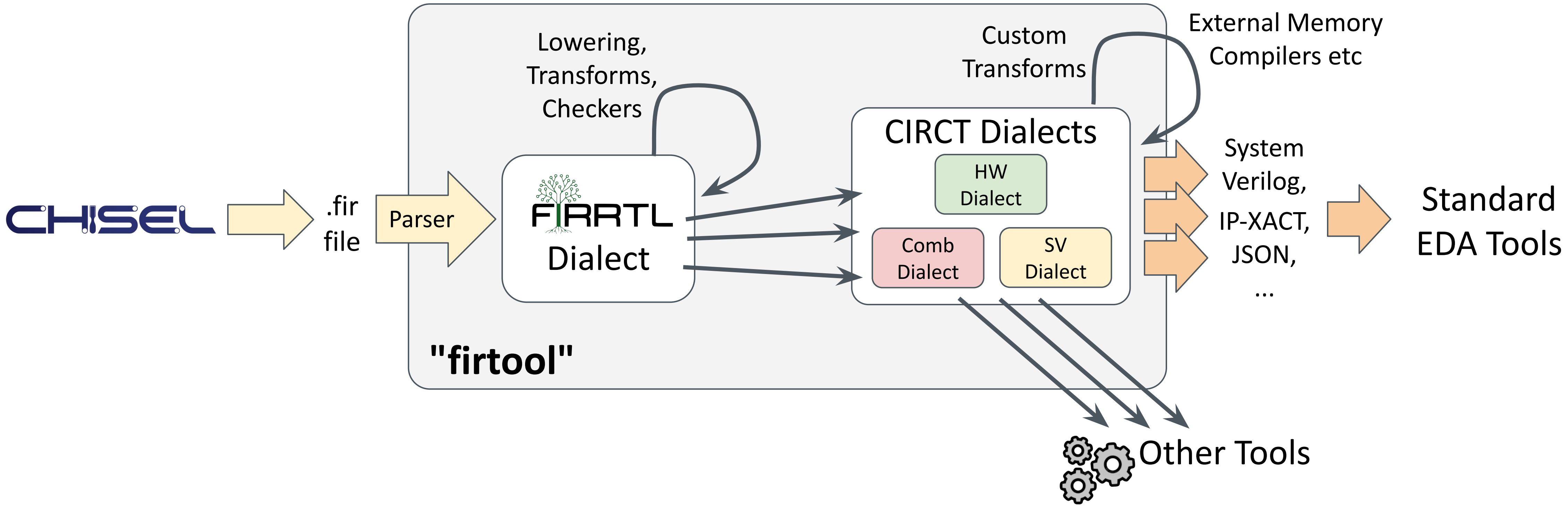
- Incorporate caching and distribution into the compiler
- Change Chisel to be less monolithic



Current utilization on 8 core Intel MacBook Pro is poor

"firtool": Experience with a production CIRCT tool

"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)
- Generates ~1500 SystemVerilog, ~300 IP-XACT, ~200 yaml, and ~100 json files

Validating correctness with formal methods

Formal equivalence of random circuits and real-world designs

- In whole and in parts
- Reference v.s. firtool
- OSS and commercial tools

Primary failure modes:

- Alternate library implementations
  - Memories: I'm looking at you
- State element changes

```
circuit top_mod :
  module top_mod :
    input inp_db: SInt<18>
    output _tmp53: UInt<18>
    wire tmp49: SInt<18>
    tmp49 <= dshr(inp_db, tail(asUInt(inp_db), 11))
    _tmp53 <= xor(tmp49, asSInt(UInt<13>(2856)))
```

firrtl

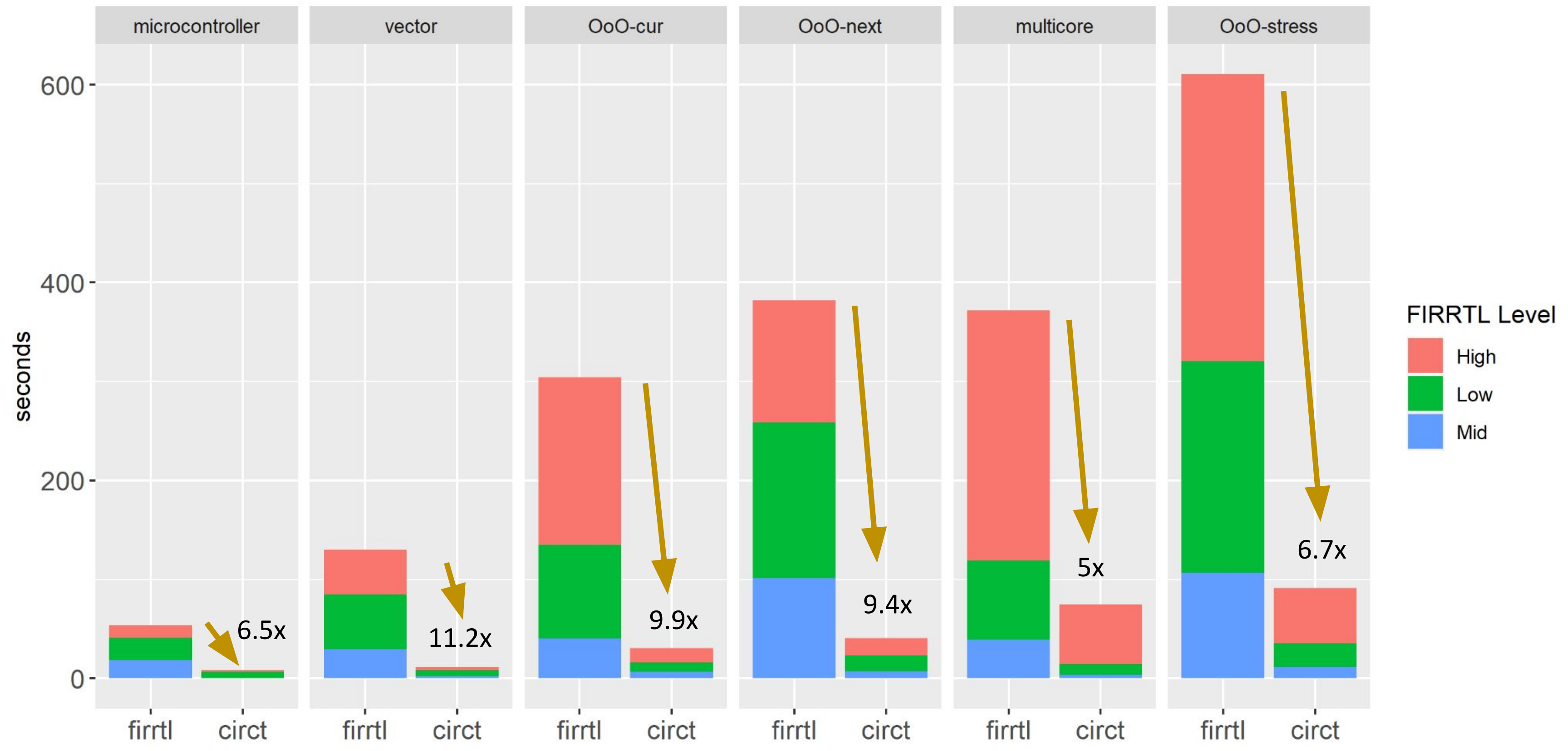
```
module top_mod(input  [17:0] inp_db,
               output [17:0] _tmp53);
  wire [17:0] _GEN_0 = inp_db;
  wire [17:0] tmp49 = $signed(inp_db) >>> _GEN_0[6:0];
  assign _tmp53 = $signed(tmp49) ^ 18'shb28;
 endmodule
```

reference

```
module top_mod(input  [17:0] inp_db,
               output [17:0] _tmp53);
  assign _tmp53 = $signed(inp_db) >>> $signed(inp_db[6:0]) ^ 18'hB28;
endmodule
```

firtool

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

"firtool" entering production unblocks further progress

Memory and CPU usage reductions enable increased design complexity

- But design size is growing faster than build machines
- Need better representations and distributed builds to keep going

Enables workflow simplifications

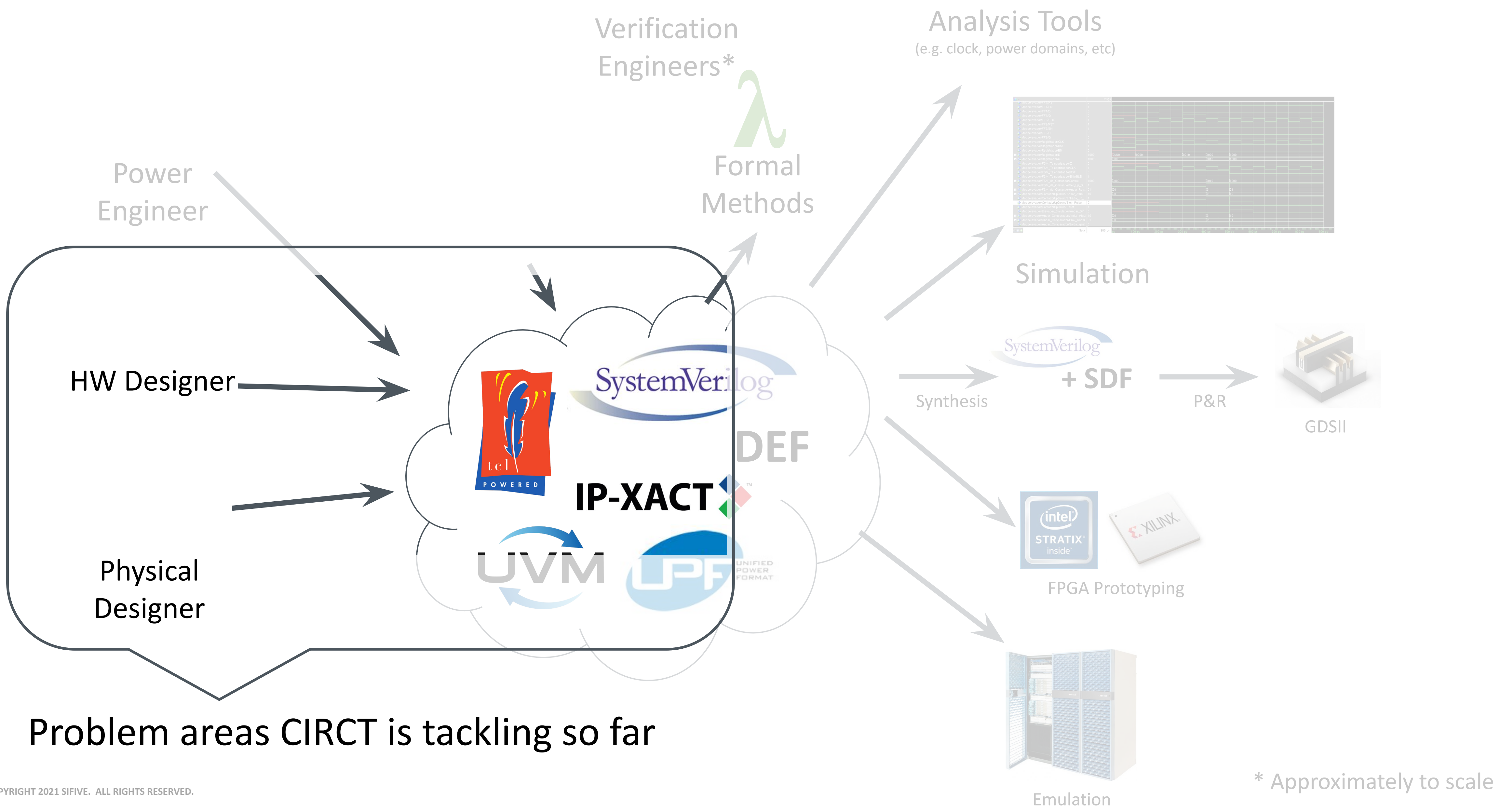
- Reduce indirection through temporary jsons in build flow
- Incorporate higher-level information into IR

Good tools have avalanching productivity gains!

#1 reaction is impatience to migrate to the new tools

CIRCT Frontiers and Future Directions

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

Software is a huge problem for Hardware design!

Without software, Silicon is just "expensive sand":

- Drivers, firmware, framework integration, compilers, etc
- Hardware teams build huge amount of SW for verification

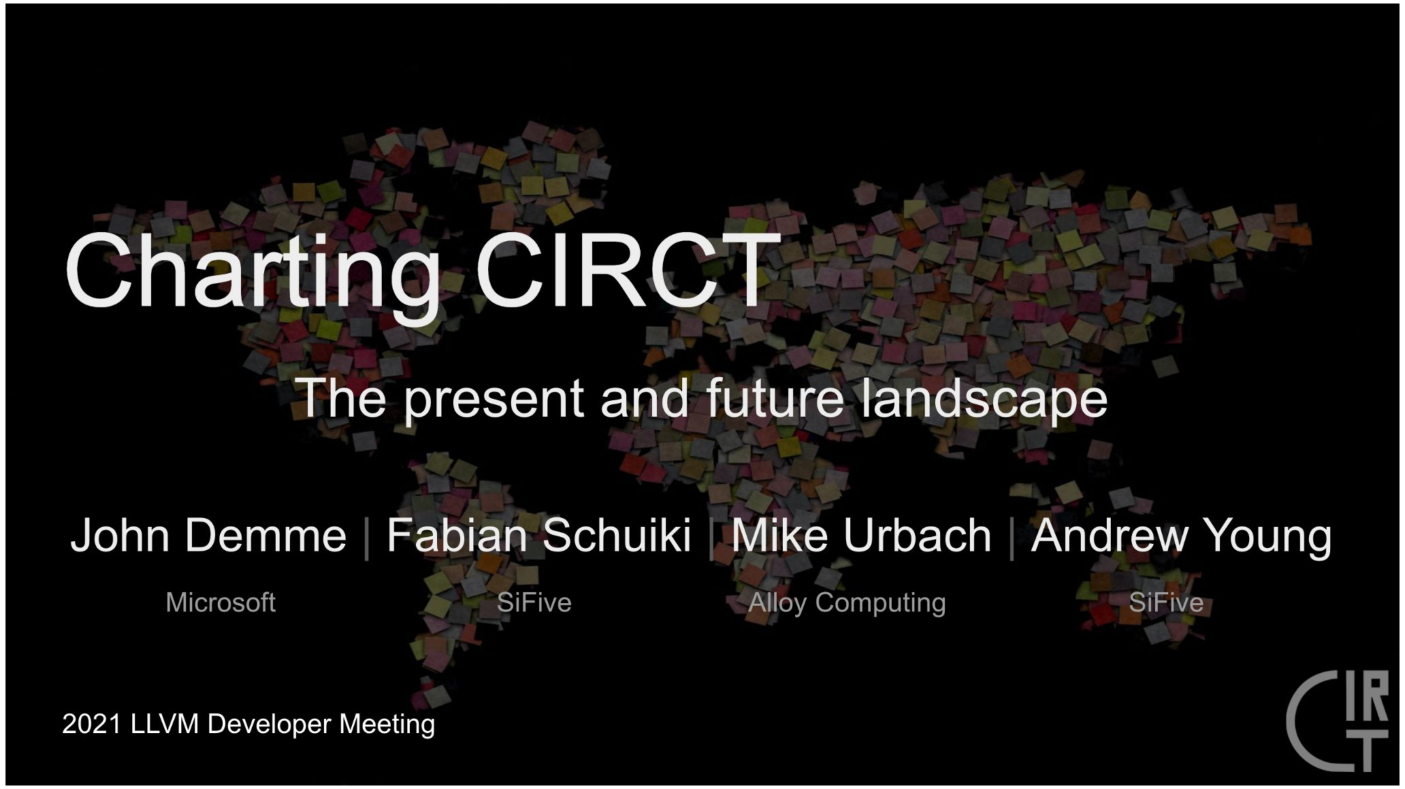
No single source of truth for HW and SW in a design!

Sand castle on Rehoboth Beach about to be washed away … photo and castle by Andy West

"If only there was a compiler framework which could represent both hardware and software…"

MLIR

Many active community projects!


Charting CIRCT

The present and future landscape

John Demme | Fabian Schuiki | Mike Urbach | Andrew Young

Microsoft SiFive Alloy Computing SiFive

2021 LLVM Developer Meeting

CIRCT: Lifting hardware development out of the 20th century

The future is built by an open and collaborative community:

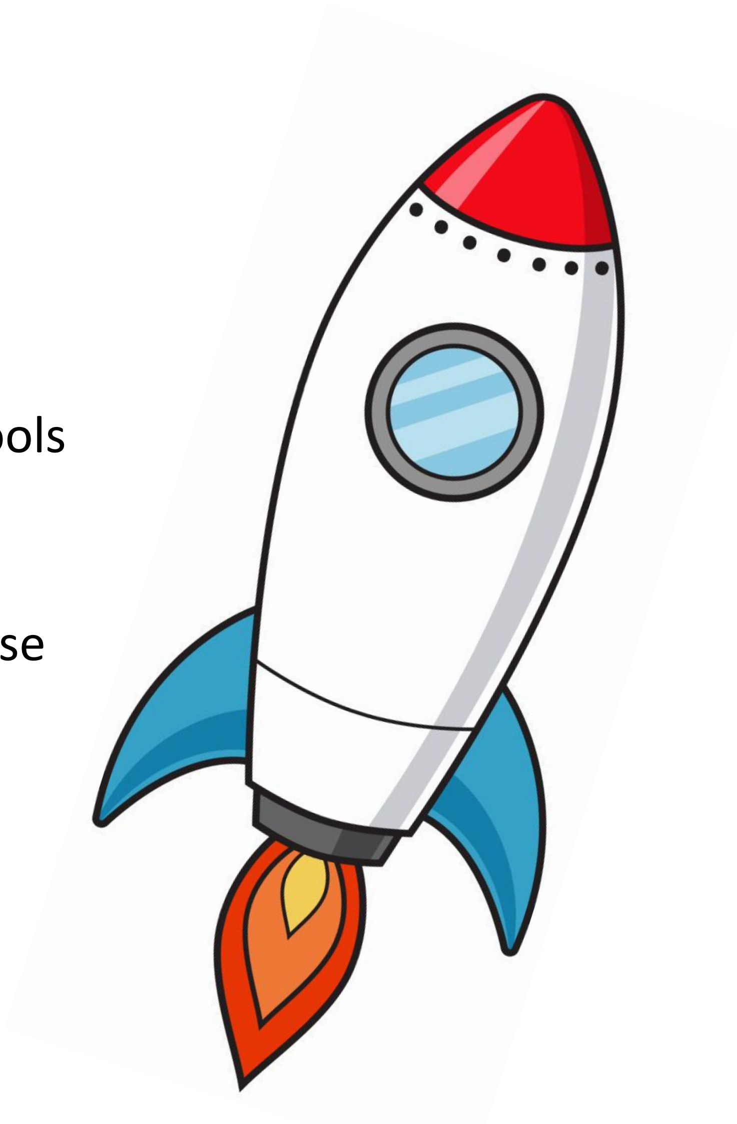
- Pulling together the small group of passionate HW tool engineers

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