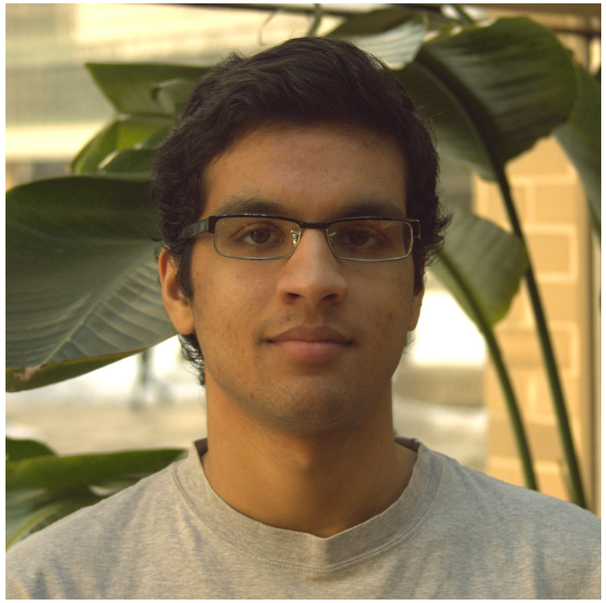
magma + CIRCT

Raj Setaluri — setaluri@cs.stanford.edu
Lenny Truong — lenny@cs.stanford.edu
8/18/2021

Who we are



Raj Setaluri
@rsetaluri

Lenny Truong
@leonardt

Generates Circuits

Abstracts Circuits

Scala Metaprograms Chisel

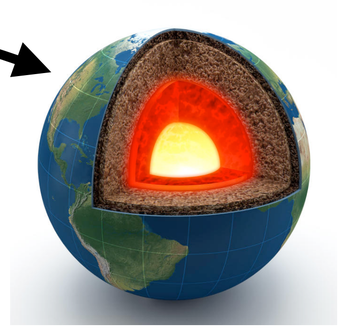
Scala

Chisel

Generates Circuits

Abstracts Circuits

Python Metaprograms Magma



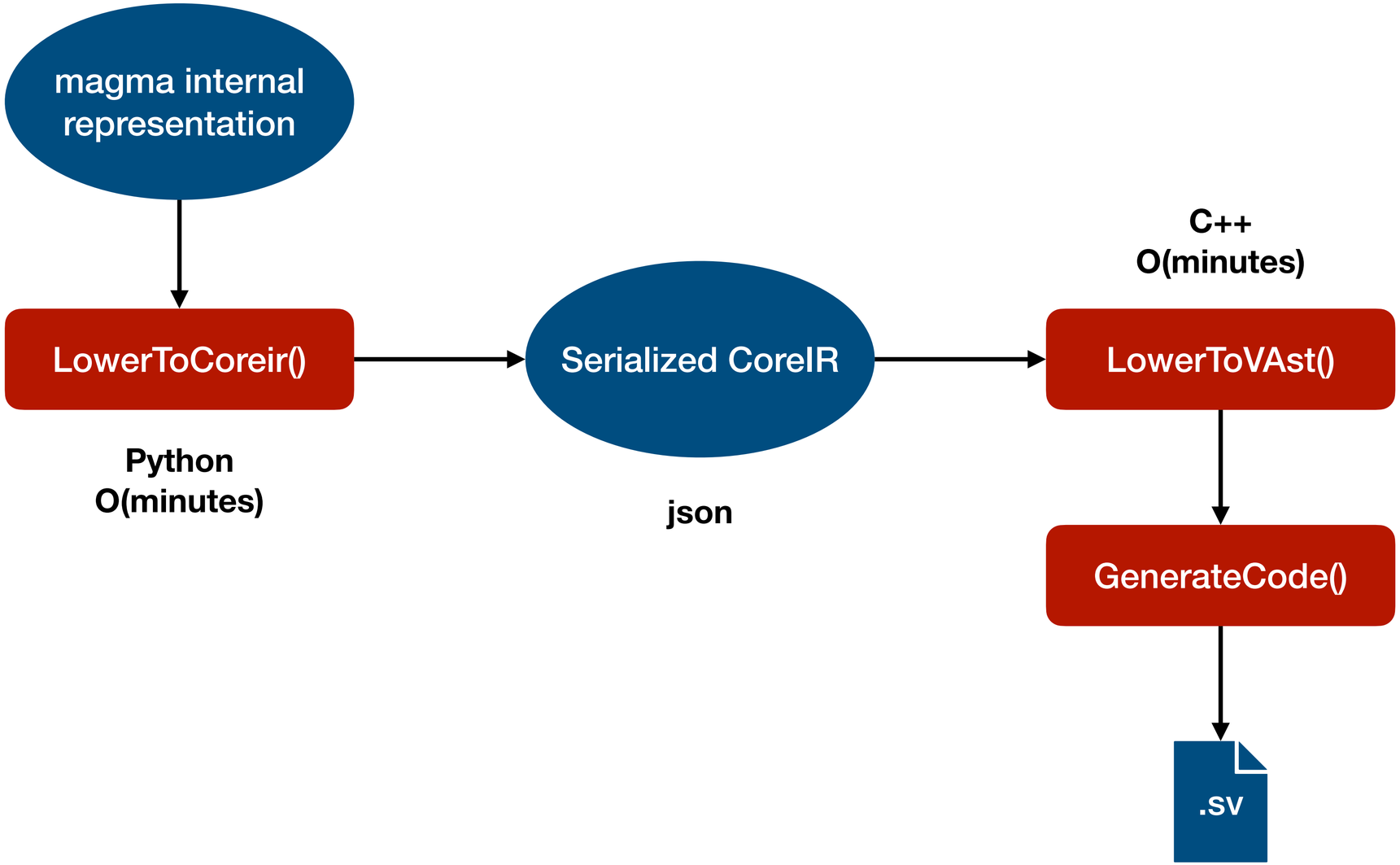
Python

Magma

magma @ Facebook

- Used magma for a major block in a production SoC
  - 5mm^2 at 5nm technology (~150M-200M transistors)
- Committed to continued use of magma, assuming
  - Continued runtime performance improvements (frontend + backend)
  - Continued feature improvements

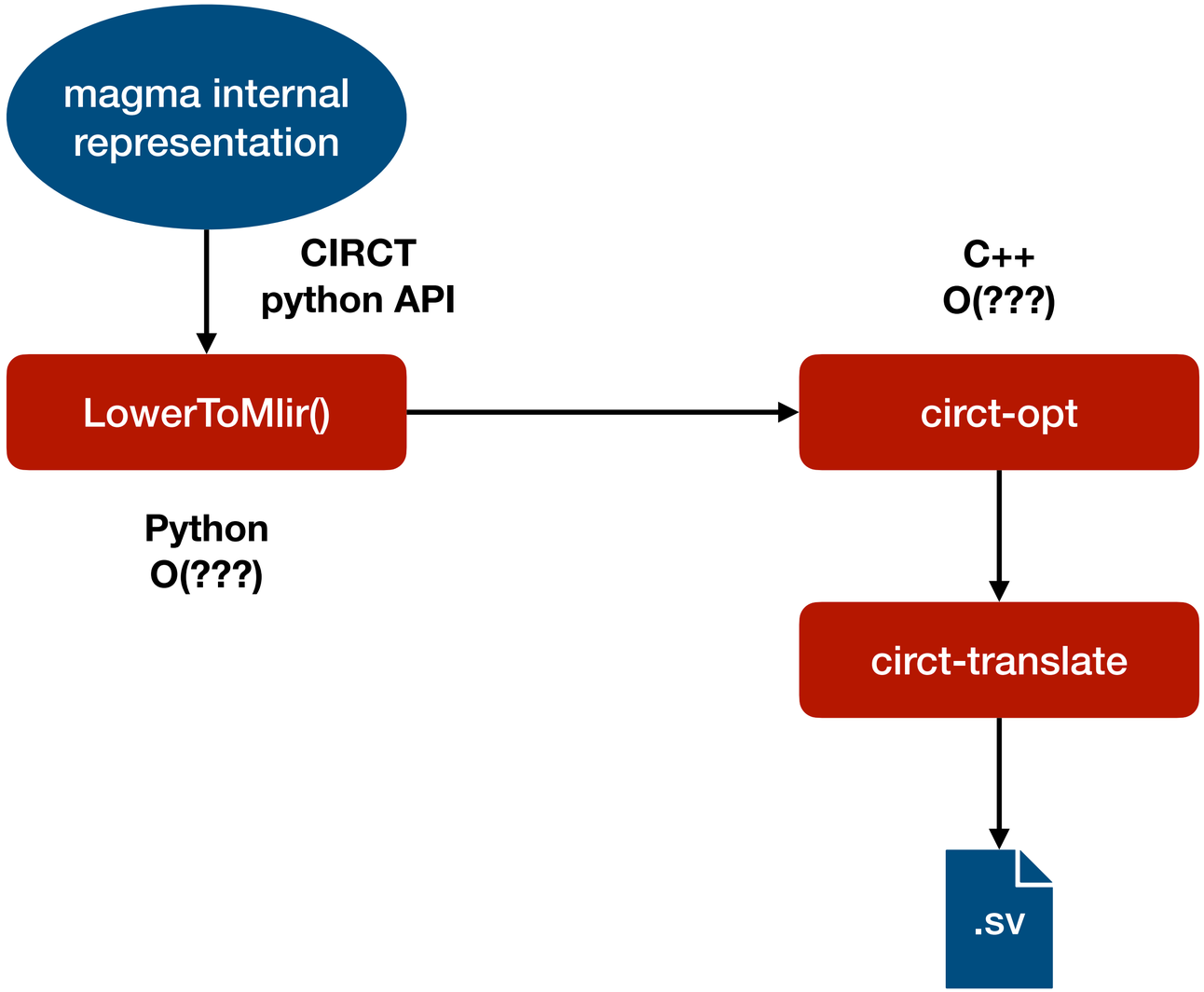
magma compiler



magma compiler issues

- Slow!
  - Round-trip serialized format because python C-API is flaky
  - CoreIR C++ is not parallel
- Had to roll own Verilog-AST library for customized code-generation

magma compiler v2 vision



magma compiler v2 requirements

- Avoid regression in generated Verilog
- Support all major magma abstractions (see later)
- High performance
    - O(minutes) —> O(1 minute)? O(seconds)?

magma+CIRCT blocker Q's

- Does `EmitVerilog` support our custom Verilog generation?
    - Inlining of comb. operators & concats
    - Persistence of (specially tagged) named wires
- Do we need to write our own dialect(s)?
    - We have complex type system (Sum/Product/Enum/Array)
    - Strict SMT-adherent comb. operators
    - Generator-as-first-class-citizen for many primitives (e.g. mux, register, memory)
- `circt-opt` & `circt-translate` performance?

Do we need to create our own dialect?

- ‘HW’ dialect
    - Structural module def + instancing
    - Array, Struct, Union, InOut types
- ‘SV’ dialect
    - Wires, reg’s, assignment, always_*
- ‘comb’ dialect
    - Strict operators (e.g. `add N :: (Bits[N], Bits[N]) -> Bits[N]`)
- **Decision**: No, use HW, SV, comb dialects

Do we need to create our own dialect?

```
!Inputs = type !hw.struct<I0: i32, I1: i32, CIN: i1>
!Outputs = type !hw.struct<O: i32, COUT: i1>

hw.module @Top(%I: !Inputs) -> (%O: !Outputs) {
  // "Explode" doesn't work?
  %0 = hw.struct_extract %I["I0"] : !Inputs
  %1 = hw.struct_extract %I["I1"] : !Inputs
  %2 = hw.struct_extract %I["CIN"] : !Inputs

  // Construct sum operands by zero-extending.
  %zero_1 = hw.constant 0 : i1
  %zero_32 = hw.constant 0 : i32
  %3 = comb.concat %zero_1, %0 : (i1, i32) -> (i33)
  %4 = comb.concat %zero_1, %1 : (i1, i32) -> (i33)
  %5 = comb.concat %zero_32, %2 : (i32, i1) -> (i33)

  // Perform sum.
  %6 = comb.add %3, %4, %5 : i33

  // Extract sum and carry out from from result and re-pack.
  %7 = comb.extract %6 from 0 : (i33) -> (i32)
  %8 = comb.extract %6 from 32 : (i33) -> (i1)
  %9 = hw.struct_create (%7, %8) : !Outputs
  hw.output %9 : !Outputs
}
```

types-struct.mlir

```
hw.module @Foo(%I: i1) -> (%O: i1) {
  hw.output %I : i1
}

hw.module @Top(%I: i1) -> (%O: i1) {
  %Foo_inst0_O = hw.instance "Foo_inst0" @Foo(%I) : (i1) -> (i1)
  hw.output %Foo_inst0_O : i1
}
```

core-structural.mlir

```
hw.module @Top(%I: i64) -> (%O16_0: i16, %O32_0: i32) {
  %I16_0 = comb.extract %I from 16 : (i64) -> i16
  %I16_1 = comb.extract %I from 32 : (i64) -> i16
  %I24_0 = comb.extract %I from 20 : (i64) -> i24
  %I32_0 = comb.extract %I from 24 : (i64) -> i32

  %0 = comb.and %I16_0, %I16_1 : i16
  %1 = comb.add %0, %I16_0 : i16
  %2 = comb.mul %1, %I16_1 : i16

  hw.output %2, %I32_0 : i16, i32
}
```

core-comb.mlir

Do we need to write our own SV codgen?

- SV dialect has:
  - ‘sym’ construct for persisting wires
  - ‘ifdef’ operator for conditional Verilog
  - ‘if’/‘casez’ operators
  - ‘verbatim’ operator backdoor —> want to minimize use
- **Decision**: No, use `EmitVerilog`; contribute passes as needed!

```
hw.module @Top(%I0: i1, %I1: i1) -> (%O: i1) {
  %x = sv.wire sym @_0 : !hw.inout<i1>
  %0 = comb.or %I0, %I1 : i1
  sv.assign %x, %0 : i1
  %x_o = sv.read_inout %x : !hw.inout<i1>
  hw.output %x_o : i1
}
```

maintain-wires.mlir

```
module Top(
  input  I0, I1,
  output O);

  wire x;       // <stdin>:3:10

  assign x = I0 | I1;   // <stdin>:4:10, :5:5
  assign O = x; // <stdin>:6:10, :7:5
endmodule
```

maintain-wires.sv

Muxes

- magma Mux is a generator:
  - `Mux T N :: (Array[N, T], Bits[clog2(N)]) -> T`
  - N need not be power of 2

Muxes

1. Flatten T into bits (ala `hw.bitcast`)

2. Stamp out CoreIR primitive:

    - `Mux w N :: (Array[N, Bits[w]], Bits[clog2(N)) -> Bits[w]`

3. Code generate if-else for `Mux w N` directly (not elaborated into mux-tree)

Muxes

```
module Mux11xBit (
    input I0,
    ...,
    input [3:0] S,
    output O
);
reg [0:0] coreir_commonlib_mux11x1_inst0_out;
always @(*) begin
if (S == 0) begin
    coreir_commonlib_mux11x1_inst0_out = I0;
end else if (S == 1) begin
    coreir_commonlib_mux11x1_inst0_out = I1;
end else if (S == 2) begin
    coreir_commonlib_mux11x1_inst0_out = I2;
end else if (S == 3) begin
...
end else begin
    coreir_commonlib_mux11x1_inst0_out = I10;
end
end

assign O = coreir_commonlib_mux11x1_inst0_out[0];
endmodule
```

Mux11xBit.sv

Muxes

- N = power of 2 restriction?
- if vs. case vs. ternary
    - `comb.mux` seems to output ternary currently
- **Decision**: ???
    - Ideally output a polymorphic `mux<N, T>` op
        - Let IR-passes do the rest
    - Know this is a well debated topic, happy to contribute

Registers

- magma register is a generator `Reg T en Opt[init] Opt[RST]`
  - `T`: type of value
  - `en`: whether or not there is an enable signal
  - `init`: initial value
  - `RST`: reset type ∈ (sync, async) x (negedge, posedge)

Registers

1. Flatten T into bits (ala `hw.bitcast`)
2. Elaborate enable/reset signals into a mux
3. Stamp out CoreIR primitive:
    - `Reg w init :: (Bits[w], Bit) -> Bits[w]`
4. Pattern match (Reg, Mux) to generate clock-gated register

Registers

```
module coreir_reg #(
    parameter width = 1,
    parameter clk_posedge = 1,
    parameter init = 1
) (
    input clk,
    input [width-1:0] in,
    output [width-1:0] out
);
  reg [width-1:0] outReg=init;
  wire real_clk;
  assign real_clk = clk_posedge ? clk : ~clk;
  always @(posedge real_clk) begin
    outReg <= in;
  end
  assign out = outReg;
endmodule
```

coreir_reg.sv

Registers

- 3 main issues:
    - Clock and Reset sensitivity
    - Specifying initial value
    - Enable signal generation
- **Decision**: Start with `sv.alwaysff`; migrate to `seq.compreg`

Registers

```
hw.module @Register_8_init_0_rst_0_en(
    %I: i8, %clk: i1, %rst: i1, %en: i1) -> (%O: i8) {
  %init_value = hw.constant 0 : i8
  %reset_value = hw.constant 0 : i8

  %reg = sv.reg {name = "reg0"} : !hw.inout<i8>
  sv.initial {
    sv.bpassign %reg, %init_value : i8
  }
  %reg_O = sv.read_inout %reg : !hw.inout<i8>
  %reg_I = comb.mux %en, %I, %reg_O : i8
  sv.alwaysff(posedge %clk) {
    sv.passign %reg, %reg_I : i8
  } (syncreset : posedge %rst) {
    sv.passign %reg, %reset_value : i8
  }
  hw.output %reg_O : i8
}
```

core-register-alwaysff.mlir

```
module Register_8_init_0_rst_0_en(
  input  [7:0] I,
  input        clk, rst, en,
  output [7:0] O);

  reg [7:0] reg0;       // <stdin>:3:13

  initial       // <stdin>:4:5
    reg0 = 8'h0;        // <stdin>:5:16, :6:7
  wire [7:0] _T = reg0; // <stdin>:8:10
  wire [7:0] _T_0 = en ? I : _T;        // <stdin>:9:10
  always @(posedge clk) begin   // <stdin>:10:5
    if (rst)    // <stdin>:10:5
      reg0 <= 8'h0;     // <stdin>:13:16, :14:7
    else        // <stdin>:10:5
      reg0 <= _T_0;     // <stdin>:11:7
  end // always @(posedge)
  assign O = _T;        // <stdin>:16:5
endmodule
```

core-register-alwaysff.sv

Mixed-direction types

- magma allows mixed direction types, e.g.
    - `Product[x=In(Bit), y=Out(Bit)]`
- Ostensibly, not possible in HW due to “functional” style

Mixed-direction types

- Options:
  - Always flatten struct ports in magma
  - Flatten only mixed-direction struct ports in magma
  - New HW concepts (seems hairy…)
    - `!hw.struct<hw.in<i1>, hw.out<i1>>`
    - `%p = hw.port : (hw.in<i1>, hw.out<i1>)`