Publishing Magma at a Hardware Venue

We need your help

DAC Submission Rejected

- Productivity story fell flat (lack of experimental results)
  - Hard to collect quantitative data (e.g. lines of code is a poor metric)
  - Dropping this line of argument
- Need a better comparison with other systems
  - Chisel, HLS
- One reviewer wanted a better description of compiler and optimizations
- **Overall** - Needs more convincing research contributions

A New Story

Research Contributions:

- A new language for hardware generators based on metaprogramming
  - Generate, introspect, and modify hardware descriptions programmatically
  - Novel use of Python metaclasses to provide an idomatic embedding
- A new hardware type system, syntax, and semantics tailored for DSLs
  - Support for multiple interpretations (simulation, hardware, formal)
- A new verilog code generation algorithm for improved readability
  - Consumes a generic structural hardware IR that can be reused by languages

Next Generation Generators with Magma

- Genesis2 embeds hardware in Perl using verilog strings templating
  - Enabled programmatic construction of hardware based on parameters
- Chisel embeds hardware in Scala using a DSL
  - Extends Genesis2's approach with an improved type system and support for object oriented and functional programming patterns
- Magma embeds hardware in Python using a DSL
  - Extends Chisel's approach with support for metaprogramming
    - Although possible in Scala/Chisel, this is not discussed in their paper

Metaprogramming Hardware

- *Metaprogramming* involves writing programs that treat other programs as data
  - Generate, examine, and modify code
- Embedding hardware as a program allows the use of metaprogramming
  - Generate a program -> Generate hardware

Embedding Hardware in Python with Magma

- Circuits are classes
- All circuits have an `io` variable that defines the interface
- Interface ports are instances type magma types (values) that provide operators
- `@=` is used to wire ports

```
class FullAdder(m.Circuit):
    io = m.IO(
        a=m.In(m.Bit), b=m.In(m.Bit), cin=m.In(m.Bit),
        sum_=m.Out(m.Bit), cout=m.Out(m.Bit)
    )

    io.sum_ @= io.a ^ io.b ^ io.cin
    io.cout @= (io.a & io.b) | (io.b & io.cin) | (io.a & io.cin)
```

A Simple Generator

- Circuits describe hardware (a program)
- A function that returns a circuit is a hardware (program) generator

```
def AdderN(n):
    class Adder(m.Circuit):
        io = m.IO(
            A=m.In(m.UInt[n]), B=m.In(m.UInt[n]), CIN=m.In(m.Bit),
            SUM=m.Out(m.UInt[n]), COUT=m.Out(m.Bit)
        )
        curr_cin = io.CIN
        for i in range(n):
            next_sum, curr_cin = FullAdder()(io.A[i], io.B[i], curr_cin)
            io.SUM[i] @= next_sum
        io.COUT @= curr_cin
    return Adder
```

Issue: Calling Versus Instancing Generators

- In the previous example, there is no relationship between the generator (a normal Python function) and the generated circuit, that is

```
assert isinstance(AdderN(4), AdderN)) is False
```

- This prevents us from introspecting circuits to see what generated them
- **Solution:** Magma provides a `Generator` class that uses metaprogramming (metaclasses) to maintain the relationship between Generators and their instances

Magma Generator Classes

```
class Adder(m.Generator2):
    def __init__(self, n: int):
        self.io = io = m.IO(
            A=m.In(m.UInt[n]), B=m.In(m.UInt[n]), CIN=m.In(m.Bit),
            SUM=m.Out(m.UInt[n]), COUT=m.Out(m.Bit)
        )
        curr_cin = io.CIN
        for i in range(n):
            next_sum, curr_cin = FullAdder()(io.A[i], io.B[i], curr_cin)
            io.SUM[i] @= next_sum
        io.COUT @= curr_cin

assert isinstance(Adder(4), Adder)) is True
```

Using Introspection

```
class AttachQueueToReadyValid(m.Generator):
    def __init__(self, ckt):
        self.io = io = m.IO(**ckt.io.ports)  # copy ports
        inst = ckt() # instance circuit to wrap
        for name in ckt.interface.ports:
            inst_port = getattr(inst, name)
            intf_port = getattr(io, name)
            if isinstance(inst_port, m.ReadyValid):
                T = type(inst_port.data)
                queue = Queue(32, T)()
                if inst_port.is_consumer():
                    inst_port @= queue.deq
                    queue.enq @= intf_port
                else:
                    queue.enq @= inst_port
                    intf_port @= queue.deq
            else:
                m.wire(inst_port, intf_port)
```

Modifying Circuits

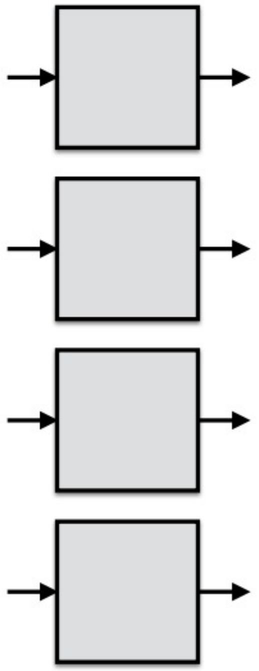
- Register file (circuit/class definition) is modified at runtime (after class is instanced)

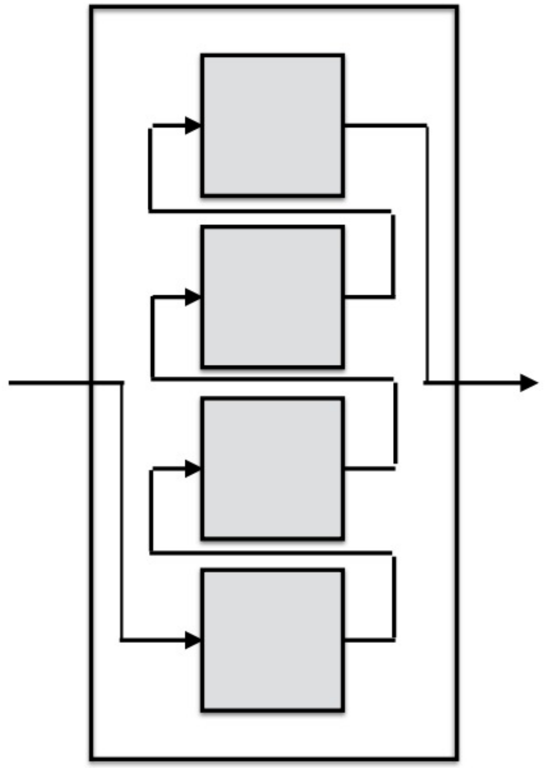
```
regs = RegFileBuilder(name="reg_file", height=32, width=x_len)
io.rdata1 @= m.mux([
    0,
    regs[io.raddr1]  # adds read port
], io.raddr1.reduce_or())
io.rdata2 @= m.mux([
    0,
    regs[io.raddr2]  # adds read port
], io.raddr2.reduce_or())
wen = m.bit(io.wen) & io.waddr.reduce_or()
regs.write(io.waddr, io.wdata, enable=m.enable(wen))  # adds write port
```

Higher Order Circuits

- Functions that takes a circuit instance and returns a new circuit instance
- User controls behavior using port names
- Implementation uses reflection

```
shift_register = fold(
    [m.Register(m.Bit)() for _ in range(8)],
    foldargs={'I': 'O'}  # I is driven by O
)
```





Metaprogramming with Magma Summary

- Magma's embedding provides full access to Python's metaprogramming facilities
    - Users can dynamically generate, inspect, and modify circuits (classes)
- Magma's use of metaclasses maintains idiomatic Python introspection
    - Avoids cognitive overhead of learning a separate language

Comparison with Chisel

- Python and Scala provide different metaprogramming capabilities
    - Cannot modify Scala classes at runtime (would violate static typing)
- Lack of metaclasses forces Chisel to use deviate from idomatic Scala
    - Chisel requires the user to use wrapper methods for DSL types
- Higher order patterns require boilerplate code

Chisel Module Pattern

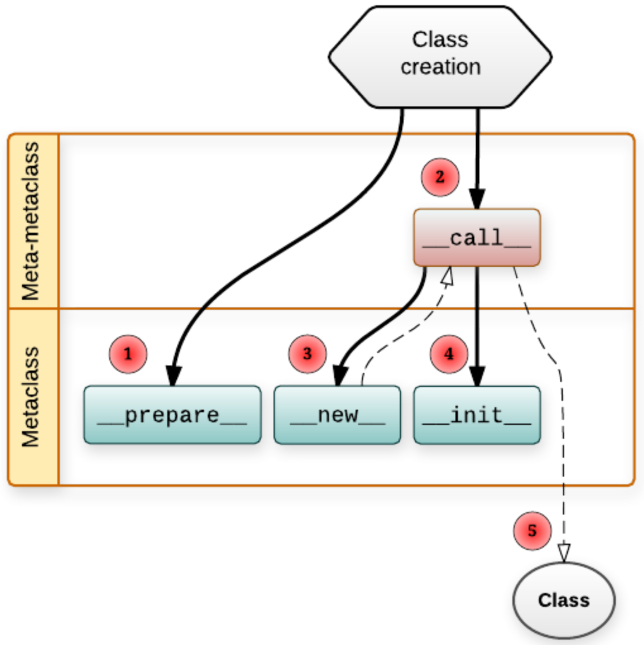
- Requires user to wrap Module instantiation (`new Foo`) with `Module(...)`

```
class Foo extends Module {...}
class Bar extends Module {
    ...
    instance = Module(new Foo)
    ...
}
```

- Deviation from standard Scala can be a source of confusion
    - stackoverflow.com/questions/40532039/chisel3-functional-module-mux4

Avoiding `Module(...)` with Metaclasses

- Chisel uses the `Module(...)` wrapper to inject code around object instantiation
    - E.g., the method updates a global variable tracking the current definition
- Magma uses the Python metaclass pipeline
    - Metaclass methods called when a subclass is defined, no extra user code required



A Scala Alternative from SpinalHDL

Uses a macro to insert "postInit" hook (runs after the subclass constructor code)

- MainTransformer.scala

Used to use DelayedInit (deprecated Scala feature)

- https://www.scala-lang.org/api/current/scala/DelayedInit.html
- https://github.com/scala/bug/issues/4330
- https://contributors.scala-lang.org/t/delayedinit-or-oncreate-any-solution/1748/43

A Note on Class Definition vs Instantiation

- Magma's language implementation inserts hooks during subclass definition
- Chisel and SpinalHDL implementation inserts hooks during class instantiation
  - In Chisel, calling `Module(new Foo)` twice will define `Foo` twice

Chisel Wires, Types, and Values

- Requires users to wrap type instantiation with `Wire(...)`
  - Types: Vec, Bundle, UInt, SInt, and Bool

```
class BigBundle extends Bundle {
  val myVec = Vec(5, SInt(23.W))
  val flag = Bool()
  val f = new MyFloat
}
```

  - Values: Wire, Reg, Input, Output, and Mem

```
val myVecWire = Wire(Vec(5, SInt(23.W)))
val myBundleWire = Wire(new BigBundle)
```

- Deviation from standard Scala can be a source of confusion
  - stackoverflow.com/questions/40816397/syntax-about-chisel-vec-wire

Chisel Higher Order Module

```scala
// Provides a more specific interface since generic Module
// provides no compile-time information on generic module's IOs.
trait MyAdder {
    def in1: UInt
    def in2: UInt
    def out: UInt
}

class Mod1 extends RawModule with MyAdder {
    val in1 = IO(Input(UInt(8.W)))
    val in2 = IO(Input(UInt(8.W)))
    val out = IO(Output(UInt(8.W)))
    out := in1 + in2
}

class Mod2 extends RawModule with MyAdder {
    val in1 = IO(Input(UInt(8.W)))
    val in2 = IO(Input(UInt(8.W)))
    val out = IO(Output(UInt(8.W)))
    out := in1 - in2
}

class X[T <: BaseModule with MyAdder](genT: => T) extends Module {
    val io = IO(new Bundle {
        val in1 = Input(UInt(8.W))
        val in2 = Input(UInt(8.W))
        val out = Output(UInt(8.W))
    })
    val subMod = Module(genT)
    io.out := subMod.out
    subMod.in1 := io.in1
    subMod.in2 := io.in2
}
```

Design of the Magma Language

- Extensible static type system based on algebraic data types
- Semantics based on SMT theory of bitvectors
- Structural syntax for general composition of hardware components
- Functional syntax for combinational circuits
- Class syntax for sequential circuits
- Coroutine syntax for finite state machines

The Magma Type System

- Types are dynamically generated and checked at Python runtime
  - Avoids dependent typing (Chisel does this too)
  - Get an error message at the source level, rather than in the generated Verilog
- Standard algebraic data types except no infinite recursion
  - User can compose primitive types to build more complex structures
  - Leverage all of PL type theory (don't need to reinvent the wheel)
- Primitive operators based on SMT
  - Easy integration with formal verification flows
  - Consistent with hwtypes executable implementation for multiple interpretations
- `MagmaProtocol` encapsulates hardware behavior of user defined types

MagmaProtocol

- Interface for interpreting user defined types as a core magma types
- Develop new types while reusing hardware semantics of magma types
- Example: Used by Peak to interpret sum types as Bits

```python
class MagmaProtocolMeta(type):
    def _to_magma_(cls):
        """Retrieve underlying magma type"""

    def _qualify_magma_(cls, direction: Direction):
        """Qualify the underlying type"""

    def _flip_magma_(cls):
        """Flip the underlying type"""

    def _from_magma_value_(cls, val: Type):
        """Create an instance from a value"""
```

Comparison with Chisel

- Similar core type system (algebraic data types)
- Primitives are not designed around SMT semantics
    - E.g. FIRRTL add primitive produces an extra carry bit unlike SMT `bvadd`
- Lacks `MagmaProtocol` style extensibility

Magma Syntax

- Core structural syntax (instance and wire circuits)
    - Expressive (captures general hardware from transistors to PCBs)
    - General composition mechanism for DSL components
- Syntax extensions that support multiple interpretations
    - Functional syntax for combinational circuits
        - `if` and `return` lowered into muxes using SSA
    - Class syntax for sequential circuits
        - Conditional assignment to state elements (registers/memories)
    - Coroutine syntax for finite state machines
        - Structured control flow lowered into finite state machine transitions

Core Structural Syntax

```
class ALU(m.Circuit):
    io = m.IO(
        a=m.In(m.UInt[16]),
        b=m.In(m.UInt[16]),
        opcode=m.In(m.Bits[2]),
        c=m.Out(m.UInt[16])
    )
    # create instance
    mux = m.Mux(4, m.UInt[16])()
    # wire up inputs
    mux.I0 @= io.a + io.b
    mux.I1 @= io.a - io.b
    mux.I2 @= io.a * io.b
    mux.I3 @= io.a / io.b
    mux.S @= io.opcode
    # Wire up output
    io.c @= mux.O
```

Functional Syntax (combinational)

```python
@m.combinational()
def alu(a: m.UInt[16], b: m.UInt[16], opcode: m.Bits[2]) -> \
        m.UInt[16]:
    if opcode == 0:
        return a + b
    elif opcode == 1:
        return a - b
    elif opcode == 2:
        return a * b
    return a / b
```

Class Syntax (sequential)

```
@m.sequential()
class ALU:
    def __init__(self):
        self.opcode = m.Register(m.Bits[2])()

    def __call__(a: m.UInt[16], b: m.UInt[16],
                 opcode: m.Bits[2], config_en: m.Bit) -> \
            m.UInt[16]:
        if config_en:
            self.opcode = opcode
        if self.opcode.prev() == 0:
            return a + b
        elif self.opcode.prev() == 1:
            return a - b
        elif self.opcode.prev() == 2:
            return a * b
        return a / b
```

Coroutine Syntax (fsms)

```python
@m.coroutine()
class UART:
    def __init__(self):
        self.message = m.Register(T=m.Bits[8], init=0)()
        self.i = m.Register(T=m.UInt[3], init=7)()
        self.tx = m.Register(T=m.Bit, init=1)()

    def __call__(self, run: m.Bit, message: m.Bits[8]) -> m.Bit:
        while True:
            self.tx = m.bit(1)  # end bit or idle
            yield self.tx.prev()
            if run:
                self.message = message
                self.tx = m.bit(0)  # start bit
                yield self.tx.prev()
                while True:
                    self.i = self.i - 1
                    self.tx = self.message[self.i.prev()]
                    yield self.tx.prev()
                    if self.i == 7:
                        break
```

Comparison with Chisel

- Similar core syntax (structural abstraction)
- Chisel uses the `when` statement which is not designed multiple interpretations
- Lacks coroutine syntax for FSMs

Why Support Multiple Interpretations?

- Useful for building higher level DSLs like PEak
    - Generate hardware, software simulation, and formal spec from a single source

Comparison with HLS

- Magma is at a lower level of abstraction (could be a target for an HLS compiler)
    - Tailored for RTL design where clock timing is explicit

Verilog Code Generation

- Consumes a generic structural hardware IR that can be reused by other languages
- Inline wire pass removes auto generated intermediates to closely match user code

FullAdder Example

```python
class FullAdder(m.Circuit):
    io = m.IO(
        a=m.In(m.Bit), b=m.In(m.Bit), cin=m.In(m.Bit),
        sum_=m.Out(m.Bit), cout=m.Out(m.Bit)
    )

    io.sum_ @= io.a ^ io.b ^ io.cin
    io.cout @= (io.a & io.b) | (io.b & io.cin) | (io.a & io.cin)
```

With Wire Inlining

```
module FullAdder (
    input a,
    input b,
    input cin,
    output sum_,
    output cout
);
assign sum_ = (a ^ b) ^ cin;
assign cout = ((a & b) | (b & cin)) | (a & cin);
endmodule
```

Without Wire Inlining

```
module corebit_xor (
    input in0,
    input in1,
    output out
);
  assign out = in0 ^ in1;
endmodule

module corebit_or (
    input in0,
    input in1,
    output out
);
  assign out = in0 | in1;
endmodule

module corebit_and (
    input in0,
    input in1,
    output out
);
  assign out = in0 & in1;
endmodule

module FullAdder (
    input a,
    input b,
    input cin,
    output sum_,
    output cout
);
wire magma_Bit_and_inst0_out;
wire magma_Bit_and_inst1_out;
wire magma_Bit_and_inst2_out;
wire magma_Bit_or_inst0_out;
wire magma_Bit_or_inst1_out;
wire magma_Bit_xor_inst0_out;
wire magma_Bit_xor_inst1_out;
corebit_and magma_Bit_and_inst0 (
    .in0(a),
    .in1(b),
    .out(magma_Bit_and_inst0_out)
);
corebit_and magma_Bit_and_inst1 (
    .in0(b),
    .in1(cin),
    .out(magma_Bit_and_inst1_out)
);
corebit_and magma_Bit_and_inst2 (
    .in0(a),
    .in1(cin),
    .out(magma_Bit_and_inst2_out)
);
corebit_or magma_Bit_or_inst0 (
    .in0(magma_Bit_and_inst0_out),
    .in1(magma_Bit_and_inst1_out),
    .out(magma_Bit_or_inst0_out)
);
corebit_or magma_Bit_or_inst1 (
    .in0(magma_Bit_or_inst0_out),
    .in1(magma_Bit_and_inst2_out),
    .out(magma_Bit_or_inst1_out)
);
corebit_xor magma_Bit_xor_inst0 (
    .in0(a),
    .in1(b),
    .out(magma_Bit_xor_inst0_out)
);
corebit_xor magma_Bit_xor_inst1 (
    .in0(magma_Bit_xor_inst0_out),
    .in1(cin),
    .out(magma_Bit_xor_inst1_out)
);
assign sum_ = magma_Bit_xor_inst1_out;
assign cout = magma_Bit_or_inst1_out;
endmodule
```

Comparison with Chisel

```
module FullAdder(
  input   clock,
  input   reset,
  input   io_a,
  input   io_b,
  input   io_cin,
  output  io_sum,
  output  io_cout
);
  wire  _T = io_a ^ io_b; // @[FullAdder.scala 16:18]
  wire  _T_2 = io_a & io_b; // @[FullAdder.scala 18:20]
  wire  _T_3 = io_b & io_cin; // @[FullAdder.scala 18:36]
  wire  _T_4 = _T_2 | _T_3; // @[FullAdder.scala 18:28]
  wire  _T_5 = io_a & io_cin; // @[FullAdder.scala 18:54]
  assign io_sum = _T ^ io_cin; // @[FullAdder.scala 16:10]
  assign io_cout = _T_4 | _T_5; // @[FullAdder.scala 18:11]
endmodule
```

Conclusion

Research Contributions:

- A new language for hardware generators based on metaprogramming
    - Generate, introspect, and modify hardware descriptions programmatically
    - Novel use of Python metaclasses to provide an idomatic embedding
- A new hardware type system, syntax, and semantics tailored for DSLs
    - Support for multiple interpretations (simulation, hardware, formal)
- A new verilog code generation algorithm for improved readability
    - Consumes a generic structural hardware IR that can be reused by languages