Magma In Production

Lenny Truong — lenny@cs.stanford.edu

10/28/20

Magma @ Facebook

- Used magma for a major block in a production SoC
  - 5mm2 at 5nm technology (around 150M-200M transistors)
- Committed to continued use of the technology for next generation of the chip
  - Migrating leaf Verilog block to magma (responsible for 30% of the area)
    - Vector ALU (8x8 dot product) with multiple math modes (4, 8, 16 bit)
- Providing data for research paper publication (DAC 2021)

Experience Report

Design

- Magma works well for low-level design with strict performance requirements
  - Core structural abstractions, operators, and types are sufficient for productively constructing a sophisticated vector architecture
  - Python embedding improves productivity/flexibility by enabling the use of parametrization, loops, conditionals, and introspection
  - Quality of generated hardware matches handwritten verilog
    (data will be included in forthcoming publication, pending FB approval)

Verification

- fault works well for unit testing (lightweight, operate at Python/magma level)
- `m.inline_verilog` feature provides stop-gap solution for verification sign-off
  - Enables use of industry standard coverage and assertion methodology

```
class Main(m.Circuit):
    io = m.IO(I=m.In(m.Bit), O=m.Out(m.Bit)) + m.ClockIO()
    io.O <= FF()(io.I)
    m.inline_verilog("""
assert property (@(posedge CLK) {I} |-> ##1 {O});
""", O=io.O, I=io.I)
```

Roadmap

Key Focus Areas

- Design Productivity: Improve ergonomics, retain performance control
  - Zero cost abstractions (simplify common wiring patterns)
  - Improve ergonomics for describing control logic
- Verification Productivity: Scaling infrastructure beyond unit tests
  - UVM abstractions: sequences, monitors, drivers, scoreboards, …
  - Unified interface to constrained random, formal methods, and coverage

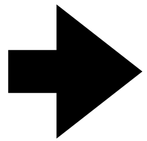
Zero Cost RTL Abstractions

- Maintain cycle-level timing semantics
  - Combinational syntax — avoid writing mux logic structurally (use if statements instead)
  - Sequential syntax — simplify register/memory interaction (behave like python variables/objects)
  - Coroutine syntax — simplify control logic (generate FSMs from program control flow)

Combinational+Sequential Syntax

```
mem.write(
    m.mux([wdata_a, wdata_b], wen_b),
    m.mux([waddr_a, waddr_b], wen_b),
    wen_a | wen_b
)
```

Structural



```
if wen_b:
    mem[wdata_b] = waddr_b
elif wen_a:
    mem[wdata_a] = waddr_a
```

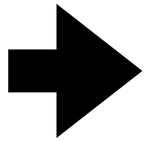
Combinational+Sequential

- Conditional statements lowered to muxes (avoid having to write structurally
- Special handling of registers/memory (implicit management of ports)

Coroutine Syntax

```
elif mem_state == MemState.WRITE:
    if io.w.valid:
        mem_wen0 @= True
        io.w.ready @= True
    if write_counter.COUT:
        io.aw.ready @= True
        mem_state @= MemState.WRITE_ACK
elif mem_state == MemState.WRITE_ACK:
    if io.b.ready:
        mem_state @= MemState.IDLE
```

Classic RTL FSM design pattern

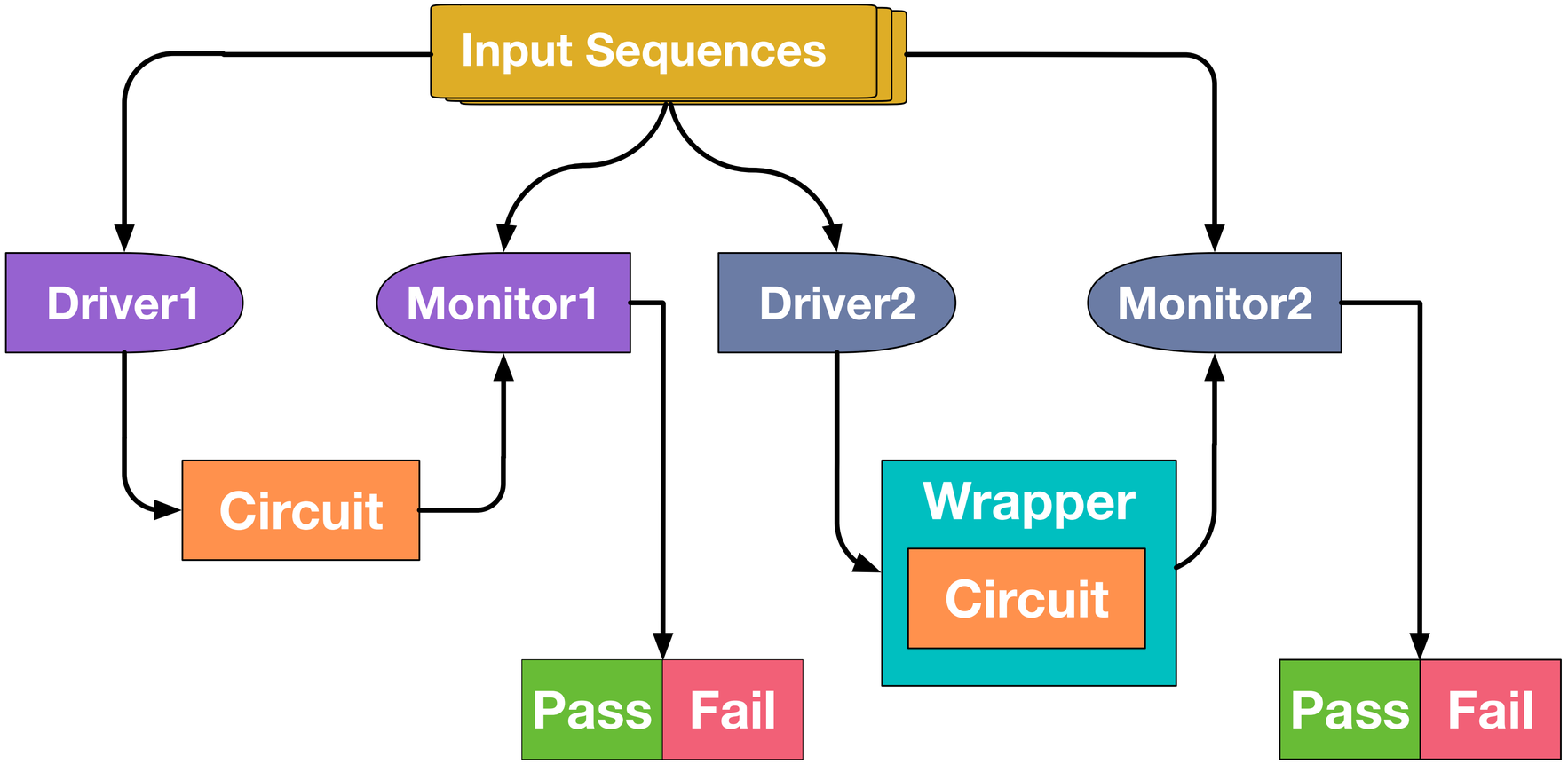


```
elif mem_state == MemState.WRITE:
    if io.w.valid:
        mem_wen0 @= True
        io.w.ready @= True
    if write_counter.COUT:
        io.aw.ready @= True
        yield
        while ~io.b.ready:
            yield
```

Coroutine yield (pause for a cycle)

- Use program state to abstract FSM transition logic

UVM Style Testing Infrastructure



https://github.com/StanfordAHA/garnet/blob/master/tests/test_peak_core/test_peak_core_and_tile.py

Fault Properties

- Write properties (assert, cover, assume) at the Python level using fault
  - Better error handling than writing SVA (catch at the Python level rather than in the generated verilog)
  - Use generator techniques (parameters, loops, conditionals, introspection)

```
class Main(m.Circuit):
    io = m.IO(write=m.In(m.Bit), read=m.In(m.Bit)) + m.ClockIO()
    f.assert_(io.write | f.implies | f.delay[1:2] | io.read,
              on=f.posedge(io.CLK))
```

Constrained Random, Formal, Coverage

- Properties + formal circuit model fed directly to our formal tools (pono)
- Generate constrained random inputs based on assumptions and use to verify assertions are not violated in simulation
- Promote use of property driven coverage metrics (versus toggle, line, etc…)
  - Can capture functional coverage as properties (reuse infrastructure)

Conclusion

- **Tech Transfer:** Industrial team at Facebook is building production hardware with magma
- **Experience Driven Research Directions:**
  - Improve design ergonomics without sacrificing performance control
  - Scaling verification infrastructure beyond unit tests