Building Reusable and Reliable Hardware with Metaprogramming in Magma and Fault

Lenny Truong

Stanford University

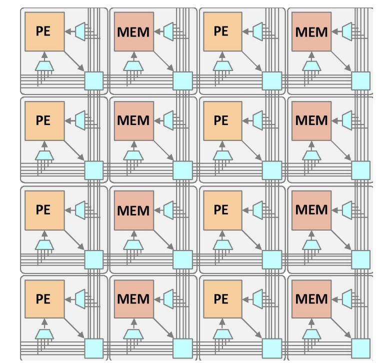
Generators

```
cfg_addr_width = 32
cfg_data_width = 32
height = 16
width = 8
mem_params = {
    "width": 32
    "depth": 256
}
```

Parameters

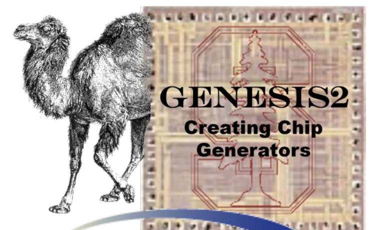
Generator

Program



Hardware, Compiler, Tests, OS Driver, …

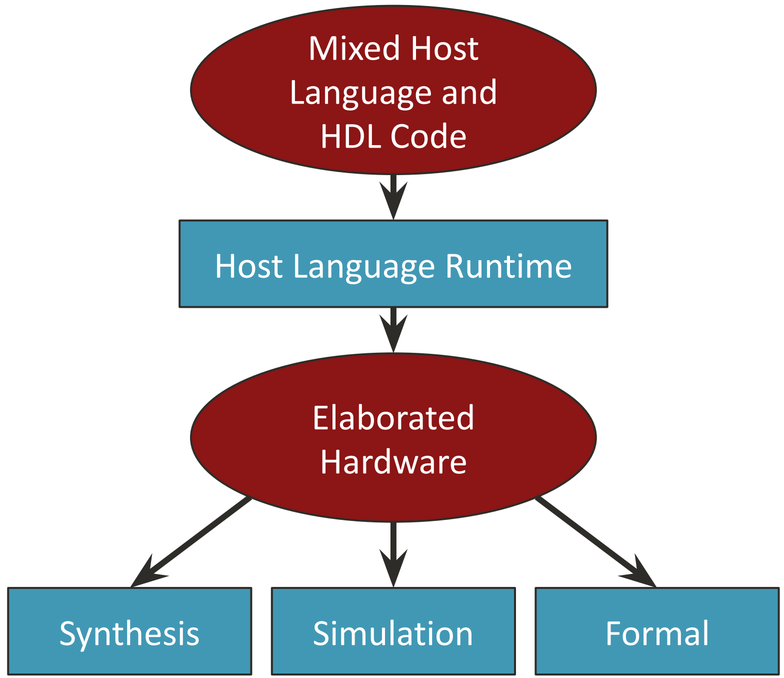
Evolution of Generator Technology



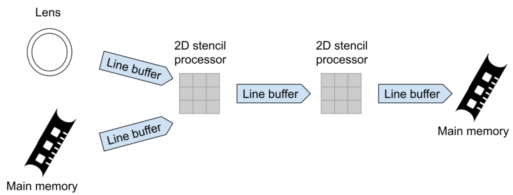
Generate Statements → **String Templates** → **Libraries and DSLs**

Generators as Metaprograms

- **Hardware** is described as a **program**
  - HDL is an embedded DSL
- **Hardware generators** are **program generators**
  - Metaprograms

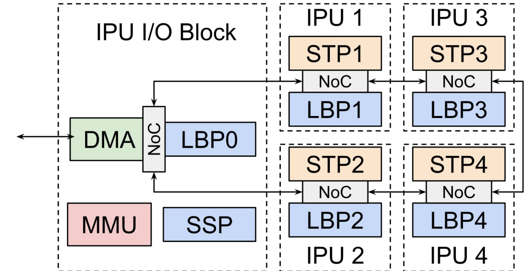


Hardware DSLs are Generators



High-level Hardware Description

Domain-specific Compiler



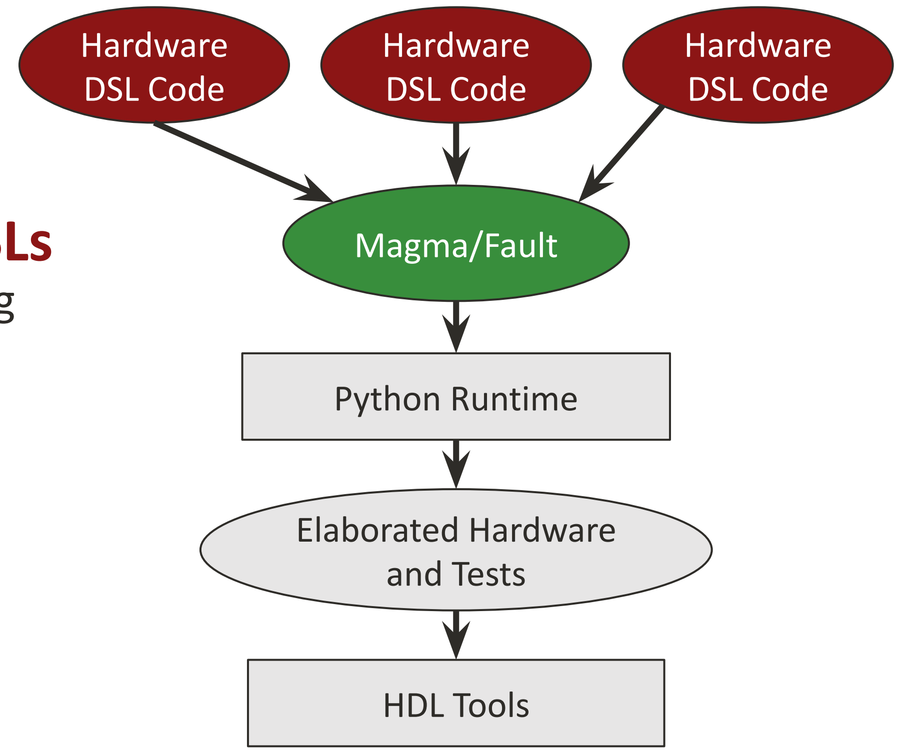
RTL, Compiler, Tests

- Sophisticated metaprograms require experts
- **DSLs amortize the cost of metaprograms**
    - User sees high-level abstractions
    - Compiler (metaprogram) analyzes, transforms, and generates code

https://en.wikipedia.org/wiki/Pixel_Visual_Core

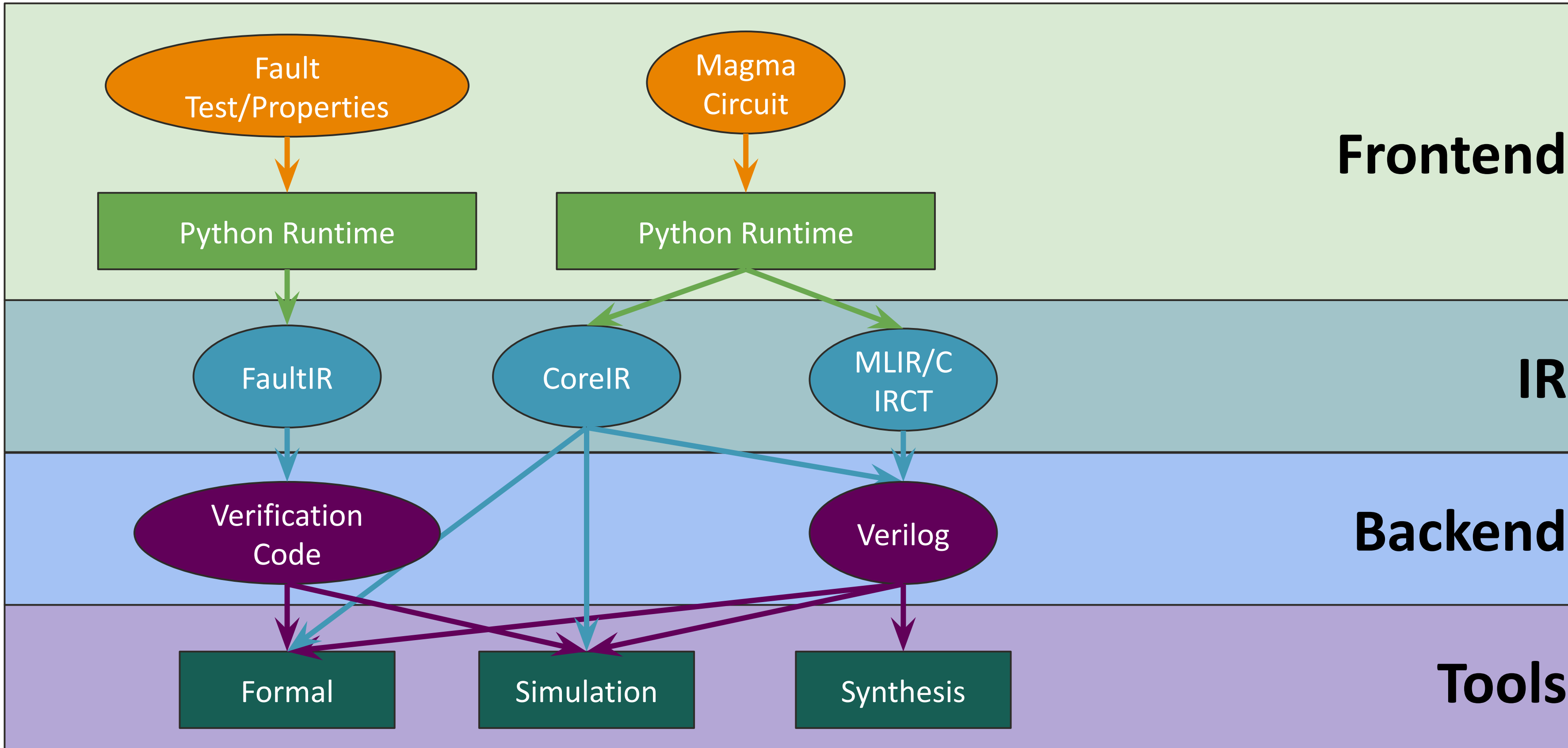
Magma/Fault Hardware DSL Platform

- **General Purpose HDL** hosts **hDSLs**
  - Leverages Python metaprogramming
  - **magma** – hardware construction
  - **fault** – hardware verification
  - Formal semantics
  - Language independent compiler IR

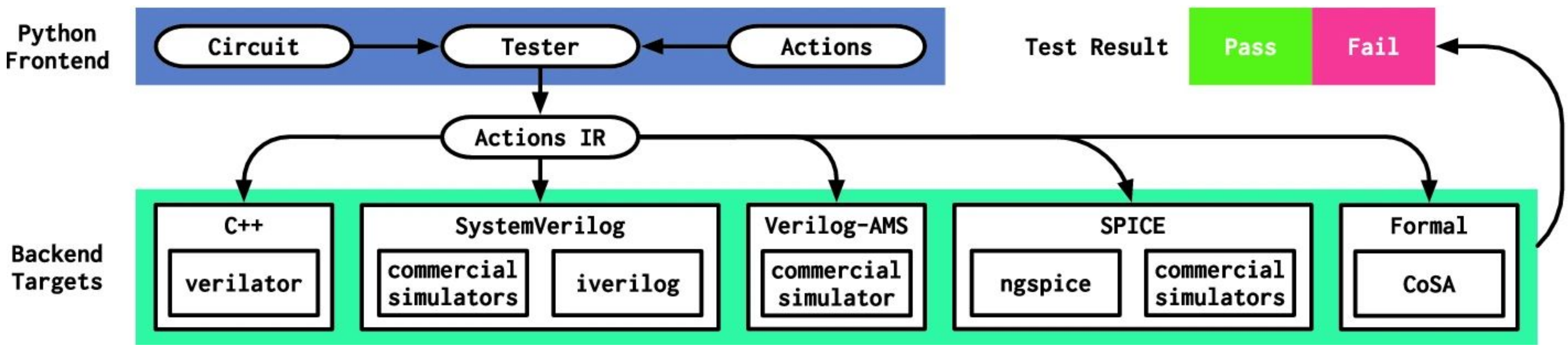


Truong and Hanrahan. “A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity.” SNAPL 2019

Magma/Fault System Architecture



Fault System Architecture



Truong, Lenny, et al. "fault: A python embedded domain-specific language for metaprogramming portable hardware verification components." *CAV 2020*

Demo Goals

- Introduce magma/fault syntax
- Introduce staged metaprogramming architecture
  - Run Python, produce design and test, run target program (e.g. simulator)
- Demonstrate design and verification metaprogramming features

CLI

```
> cd /aha/fault-micro/
> ls
advanced_pe.py  simple_alu.py  simple_pe.py
> python simple_alu.py
> ls build
SimpleALU.v  SimpleALU_driver.cpp
```

Python Imports

```python
import magma as m
import fault as f
import hwtypes as ht

import operator
```

simple_alu.py – Basic Magma

```python
class SimpleALU(m.Circuit):
    io = m.IO(
        a=m.In(m.UInt[16]),
        b=m.In(m.UInt[16]),
        c=m.Out(m.UInt[16]),
        opcode=m.In(m.Bits[2])
    )

    io.c @= m.mux(
        [io.a + io.b, io.a - io.b, io.a * io.b, io.b ^ io.a],
        io.opcode
    )
```

Testing the SimpleALU

```python
ops = [operator.add, operator.sub, operator.mul, operator.xor]
tester = f.Tester(SimpleALU)
for i, op in enumerate(ops):
    tester.circuit.opcode = i
    tester.circuit.a = a = ht.BitVector.random(16)
    tester.circuit.b = b = ht.BitVector.random(16)
    tester.eval()
    tester.circuit.c.expect(op(a, b))

tester.compile_and_run("verilator", flags=["-Wno-fatal"],
                       directory="build")
```

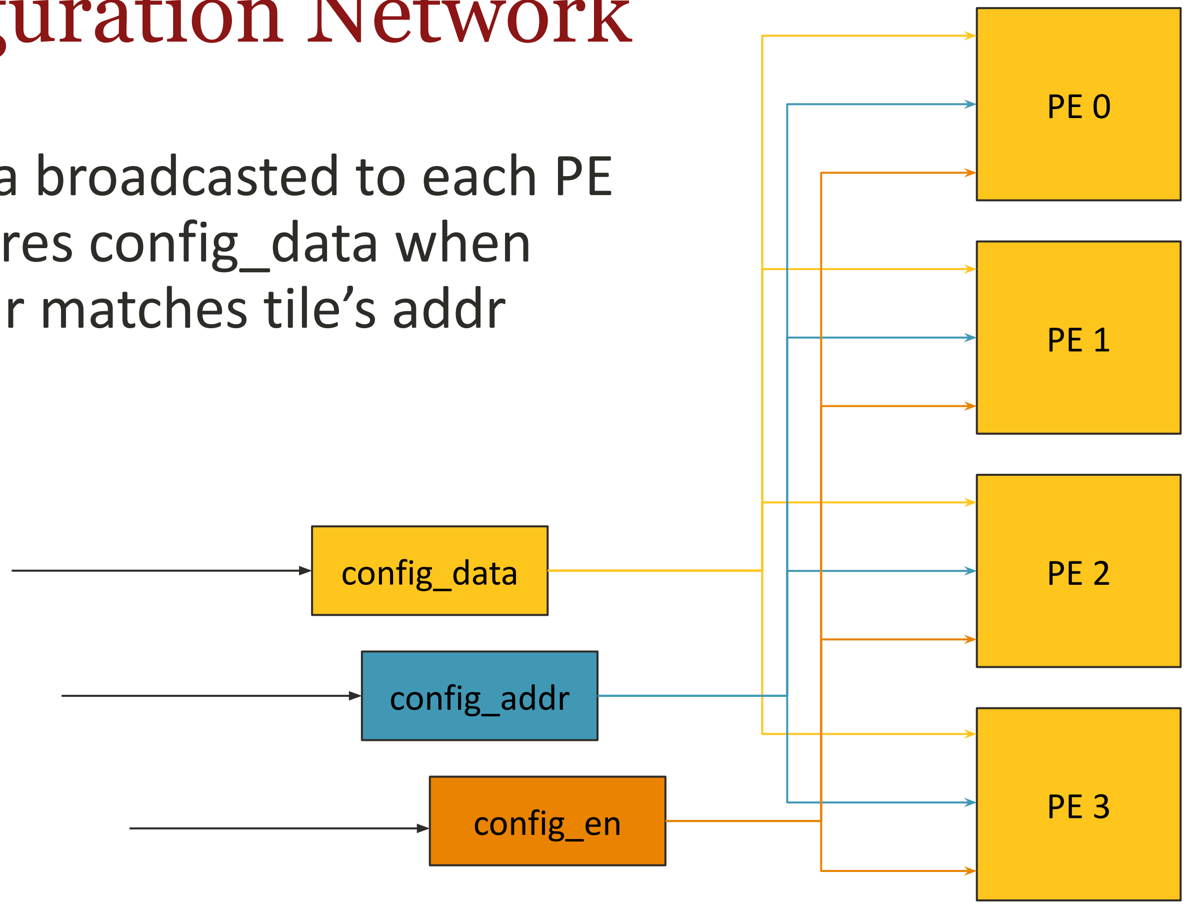
Optimizing Add/Sub

```
class OptALU(m.Circuit):
    io = m.IO(
        a=m.In(m.UInt[16]),
        b=m.In(m.UInt[16]),
        c=m.Out(m.UInt[16]),
        opcode=m.In(m.Bits[2])
    )

    sum_ = io.a + m.mux([io.b, -io.b], io.opcode[0])
    io.c @= m.mux(
        [sum_, sum_, io.a * io.b, io.b ^ io.a],
        io.opcode
    )
```

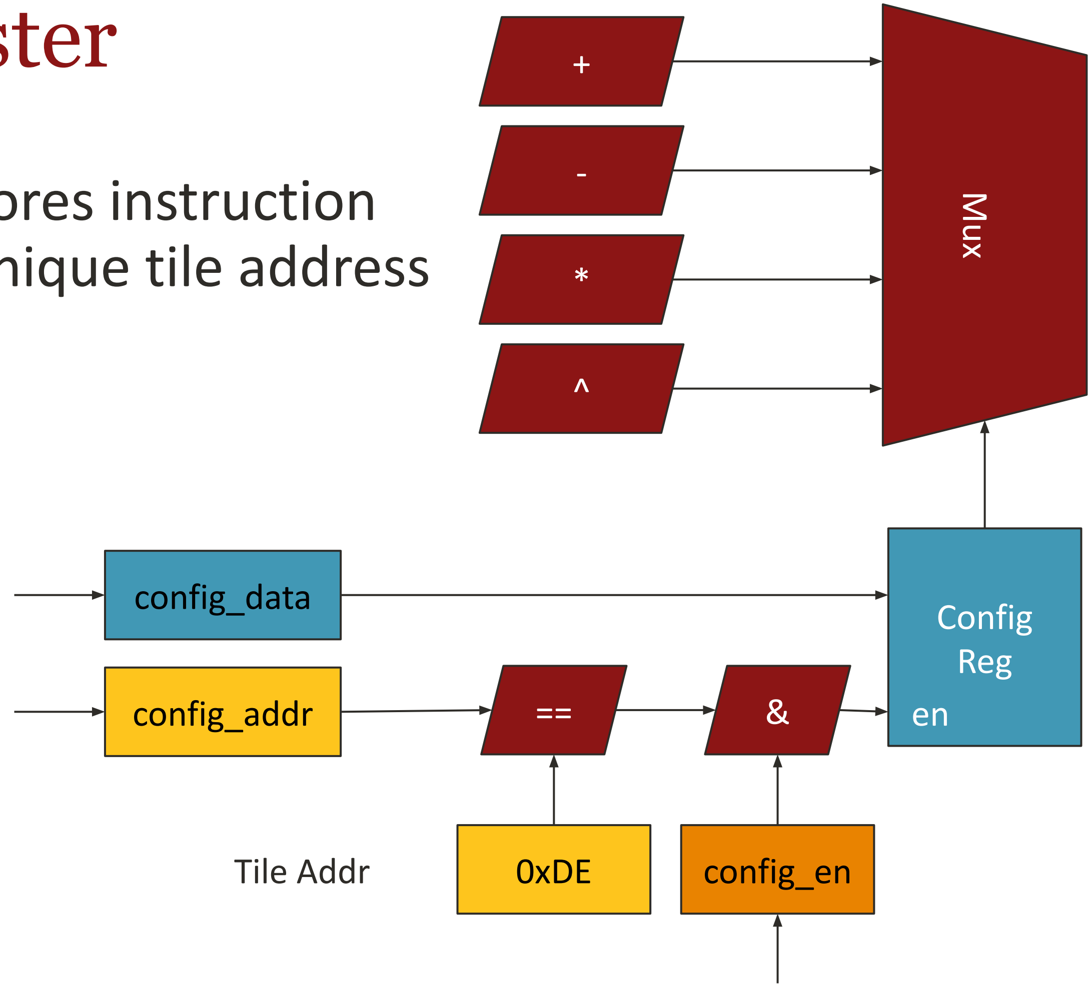
PE Configuration Network

- config_data broadcasted to each PE
- PE only stores config_data when config_addr matches tile's addr



PE Config Register

- configuration reg stores instruction
- each instance has unique tile address



simple_pe.py – Generate Configuration Logic

```python
class SimplePE(m.Generator2):
    def __init__(self, addr):
        self.io = io = m.IO(
            a=m.In(m.UInt[16]), b=m.In(m.UInt[16]), c=m.Out(m.UInt[16]),
            config_addr=m.In(m.Bits[8]), config_data=m.In(m.Bits[2]), config_en=m.In(m.Enable)
        ) + m.ClockIO(has_reset=True)

        opcode = m.Register(m.Bits[2], has_enable=True, reset_type=m.Reset)()(
            io.config_data,
            CE=io.config_en & (io.config_addr == addr)
        )

        io.c @= m.mux(
            [io.a + io.b, io.a - io.b, io.a * io.b, io.b ^ io.a],
            opcode
        )
```

Config Tester – Reusable Test Components

```
class ConfigurationTester:
    def __init__(self, circuit, config_addr_port, config_data_port,
                 config_en_port):
        self.config_addr_port = config_addr_port
        self.config_data_port = config_data_port
        self.config_en_port = config_en_port

    def configure(self, addr, data):
        self.poke(self.config_addr_port, addr)
        self.poke(self.config_data_port, data)
        self.poke(self.config_en_port, 1)
        self.step(2)
        self.poke(self.config_en_port, 0)
```

ResetTester – Type Introspection

```python
class ResetTester:
    def __init__(self, circuit):
        for port in circuit.interface.ports.values():
            if isinstance(port, m.Reset):
                self.reset_port = port
                break

    def reset(self):
        self.poke(self.reset_port, 1)
        self.step(2)
        self.poke(self.reset_port, 0)
        self.step(2)
```

PETester – Composition

```
class PETester(f.SynchronousTester, ResetTester, ConfigurationTester):
    def __init__(self, circuit, clock, config_addr_port, config_data_port,
                 config_en_port):
        f.SynchronousTester.__init__(self, circuit, clock)
        ResetTester.__init__(self, circuit)
        ConfigurationTester.__init__(self, circuit, config_addr_port,
                                     config_data_port, config_en_port)

    def check_op(self, addr, instr, op):
        tester.configure(addr, instr)
        tester.circuit.a = a = ht.BitVector.random(16)
        tester.circuit.b = b = ht.BitVector.random(16)
        tester.step(2)
        tester.circuit.c.expect(op(a, b))
```

Using the PETester

```
ops = [operator.add, operator.sub, operator.mul,
       operator.xor]
addr = 0xDE
PE = SimplePE(addr)
tester = PETester(
    PE, PE.CLK, PE.config_addr, PE.config_data,
    PE.config_en
)
for i, op in enumerate(ops):
    tester.check_op(addr, i, op)
tester.reset()
tester.check_op(addr, 0, ops[0])
```

Generating a PE from Instruction/Op Mapping

```
class AdvancedPE(m.Generator2):
    def __init__(self, addr, instr_op_map):
        n_cfg_bits = max(x.bit_length() for x in instr_op_map.keys())
        self.io = io = m.IO(
            a=m.In(m.UInt[16]), b=m.In(m.UInt[16]), c=m.Out(m.UInt[16]),
            config_addr=m.In(m.Bits[8]), config_data=m.In(m.Bits[n_cfg_bits]),
            config_en=m.In(m.Enable)
        ) + m.ClockIO(has_reset=True)

        opcode = m.Register(
            m.Bits[n_cfg_bits], has_enable=True, reset_type=m.Reset
        )()(io.config_data, CE=io.config_en & (io.config_addr == addr))
        curr = None
        for instr, op in instr_op_map.items():
            next = op(self.io.a, self.io.b)
            if curr is not None:
                next = m.mux([curr, next], opcode == instr)
            curr = next
        self.io.c @= curr
```

Testing the PE Generator

```python
addr = 0xDE
ops = m.common.ParamDict({
    0xDE: operator.add, 0xAD: operator.sub,
    0xBE: operator.mul, 0xEF: operator.xor
})
PE = AdvancedPE(addr, ops)
tester = PETester(
    PE, PE.CLK, PE.config_addr, PE.config_data, PE.config_en
)
for inst, op in ops.items():
    tester.check_op(addr, inst, op)

tester.reset()
tester.check_op(addr, 0xDE, ops[0xDE])
```

More Magma/Fault

- **DISCLAIMER:** magma has undergone significant changes since the latest commit used in the AHA docker
- https://github.com/phanrahan/magma/tree/master/examples
- Linear Feedback Shift Register
    - Python libraries, higher-order circuits (functional programming patterns)
    - https://github.com/phanrahan/magma/blob/master/examples/lfsr.py
    - https://github.com/phanrahan/magma/blob/master/examples/tests/test_lfsr.py
- Batchers Odd-Even Sorting Network
    - Advanced higher-order circuits and recursion
    - https://github.com/phanrahan/magma/blob/master/examples/odd_even_sort.py
    - https://github.com/phanrahan/magma/blob/master/examples/tests/test_odd_even_sort.py
- riscv_mini
    - 3-stage processor design
    - https://github.com/phanrahan/magma/tree/master/examples/riscv_mini