Fault: A Python Framework for Agile Hardware Verification using Executable Specifications, Formal Methods, and Metaprogramming

Lenny Truong - lenny@stanford.edu

StanfordAHA Intel Annual Review 10/15/2018

https://github.com/leonardt/fault

AHA
Agile Hardware Center

intel

Photo of San Andreas Fault by John Wiley

Who am I?

Lenny Truong - lenny@cs.stanford.edu

PhD Advisor: Pat Hanrahan

Research Focus: Hardware Languages and Verification

Background: Programming Languages and Compilers

- Past member of the Intel Labs Programming Systems Lab/Group
- Past member of SEJITS group at the UC Berkeley ASPIRE Lab (Undergraduate advisor: Armando Fox)

2018-Present

2016-Present

AHA
Agile Hardware Center

2015-2016

intel LABS

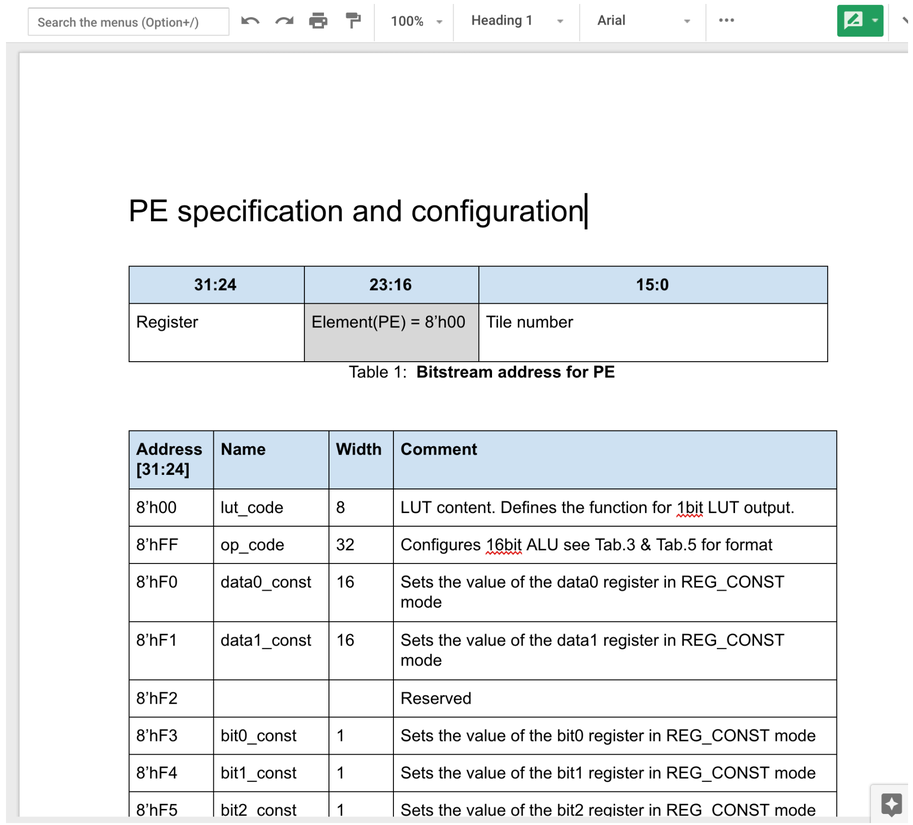
2014-2015

ASPIRE
UC BERKELEY

Retrospective: CGRA Tapeout 1

- Human readable **only** specification format (Google Doc)
- Entirely open source flow for initial design and verification
- Test benches written in C++

The Specification Problem



31:24	23:16	15:0
Register	Element(PE) = 8'h00	Tile number

Table 1: **Bitstream address for PE**

Address [31:24]	Name	Width	Comment
8'h00	lut_code	8	LUT content. Defines the function for 1bit LUT output.
8'hFF	op_code	32	Configures 16bit ALU see Tab.3 & Tab.5 for format
8'hF0	data0_const	16	Sets the value of the data0 register in REG_CONST mode
8'hF1	data1_const	16	Sets the value of the data1 register in REG_CONST mode
8'hF2			Reserved
8'hF3	bit0_const	1	Sets the value of the bit0 register in REG_CONST mode
8'hF4	bit1_const	1	Sets the value of the bit1 register in REG_CONST mode
8'hF5	bit2_const	1	Sets the value of the bit2 register in REG_CONST mode

Human readable **only** specification format (Google Doc)

The Specification Problem

- **Task**: Verify that the flags output of the PE matches the spec

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flag_sel				irq_en		acc_en	n/a	n/a	Signed	Operation					

Table 5: **PE configuration: op_code[15:0]**

4’hB	N != V
4’hC	~Z & (N == V)
4’hD	Z | (N != V)
4’hE	lut_code[{bit2,bit1,bit0}]
4’hF	comp_res_p (see “Predicate 1b”)

Table 6: **flag_sel options**

C - ‘carry out’ flag
V - ‘overflow’ flag
Z - res(Result 16b in Tab7) == 0
N - res[16] (“Result 16b” in Tab7), same as sign(res)
lut_code[***] - value stored in LUT
comp_res_p - see “Predicate 1b” in Tab7

HW name	Inputs	Result 16b	Predicate 1b	HW name
Minimum Set of Operations				
ADD	a,b - int16 or u_int16 d_p - 1bit	res = a + b + d_p	(a + b + d_p) >=2^16	Addition
SUB	a,b - int16 or u_int16	res = a + ~b + 1	(a + ~b + 1) >=2^16	Subtraction-Addition

The Specification Problem

- **Issue 1**: Document based specifications can be ambiguous

What does carry mean for non addition/subtraction?

C - ‘carry out’ flag
V - ‘overflow’ flag
Z - res(Result 16b in Tab7) == 0
N - res[16] (“Result 16b” in Tab7), same as sign(res)
lut_code[***] - value stored in LUT
comp_res_p - see “Predicate 1b” in Tab7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flag_sel				irq_en		acc_en	n/a	n/a	Signed	Operation					

Table 5: **PE configuration: op_code[15:0]**

4’hB	N != V
4’hC	~Z & (N == V)
4’hD	Z | (N != V)
4’hE	lut_code[{bit2,bit1,bit0}]
4’hF	comp_res_p (see “Predicate 1b”)

Table 6: **flag_sel options**

HW name	Inputs	Result 16b	Predicate 1b	HW name
Minimum Set of Operations				
ADD	a,b - int16 or u_int16 d_p - 1bit	res = a + b + d_p	(a + b + d_p) >=2^16	Addition
SUB	a,b - int16 or u_int16	res = a + ~b + 1	(a + ~b + 1) >=2^16	Subtraction-Addition

The Specification Problem

- **Issue 2**: In an agile world, the specification is constantly evolving

Specified output of the compute unit has changed

HW name	Inputs	Result 16b	Predicate 1b	HW name
LSHFT	a - int16 or u_int16 b[3:0] - uint16	a << b[3:0]	*(a + b) >=2^16*	Shift left
MULT_2	a,b - int16 or u_int16	(a*b)[15:0]	0~~(a*b+c)>=2^16~~	Multiply
MULT_1	a,b - int16 or u_int16	(a*b)[23:8]	0~~(a*b+c)>=2^24~~	Multiply middle

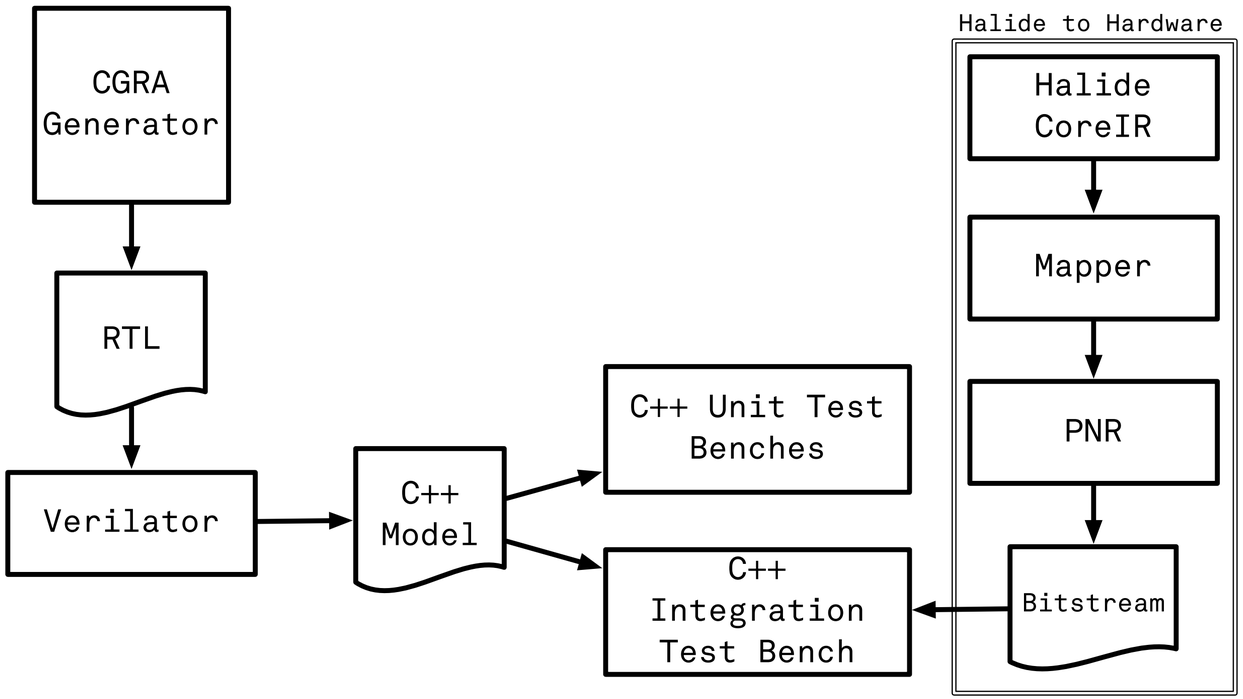
- Because the specification is only human readable, the test bench has to be manually updated any time the specification changes

The Tapeout Environment Problem

We don’t control the world

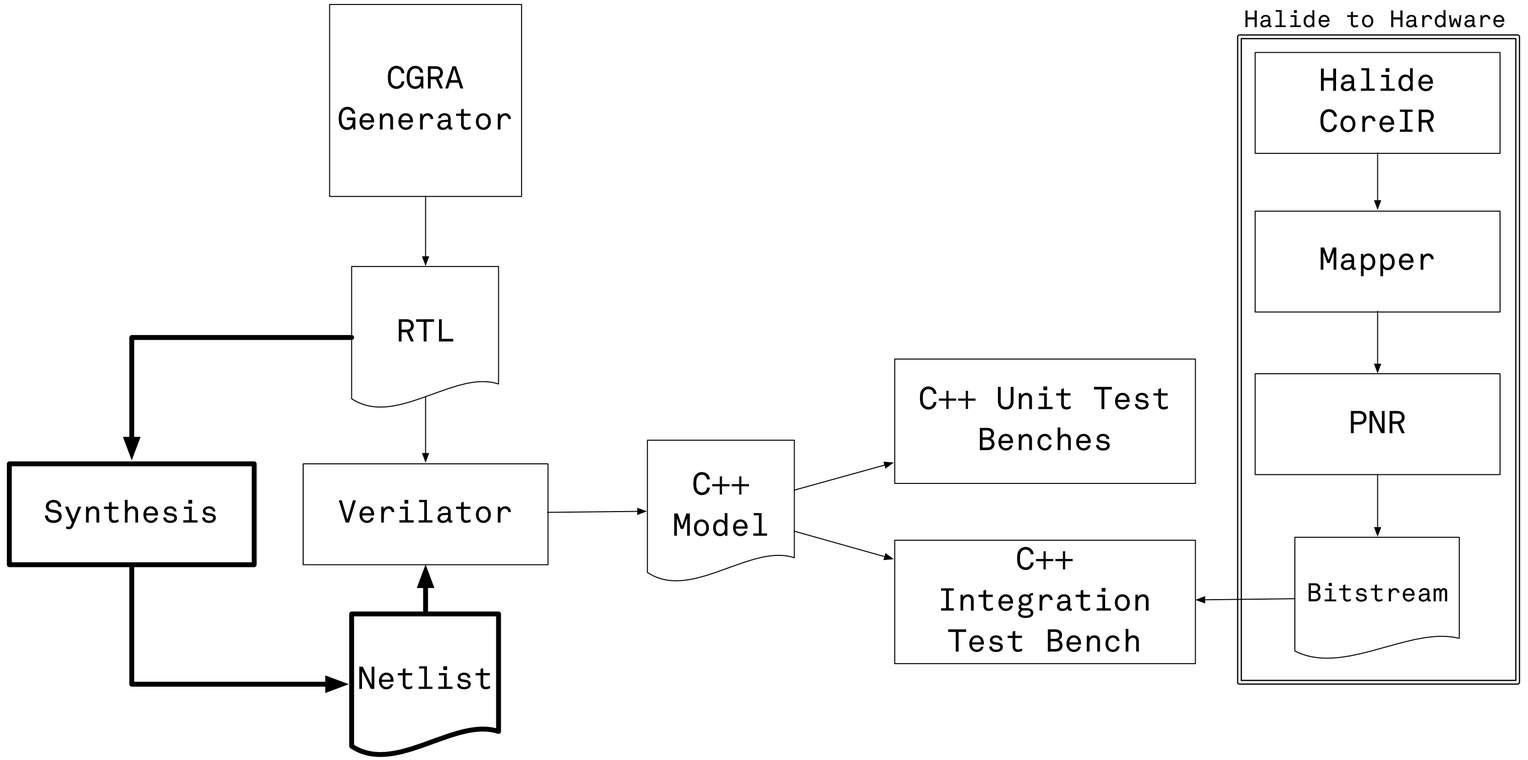
The Tapeout Environment Problem

- Initial design and verification done using entirely open source toolchain



The Tapeout Environment Problem

- Hoped to run the post synthesis design through the same flow



The Tapeout Environment Problem

- **Issue 1**: The open source simulator we used was incompatible with post-synthesis verilog

```
%Error: <some_file>.v:4802: Unsupported: Verilog 1995 reserved word not implemented: weak0
%Error: <some_file>.v:4802: syntax error, unexpected ',', expecting IDENTIFIER or '{'
%Error: <some_file>.v:4802: Unsupported: Verilog 1995 reserved word not implemented: weak1
%Error: <some_file>.v:5798: syntax error, unexpected '(', expecting IDENTIFIER
%Error: <some_file>.v:5805: syntax error, unexpected '(', expecting IDENTIFIER
```

- **Issue 2:** Our test benches were not portable to a different simulator

The Static Test Bench Problem

“To reduce design costs, we need to stop building chip instances, and start making chip generators instead.”

Shacham, Ofer, et al. "Rethinking digital design: Why design must change." IEEE micro 30.6 (2010): 9-24.

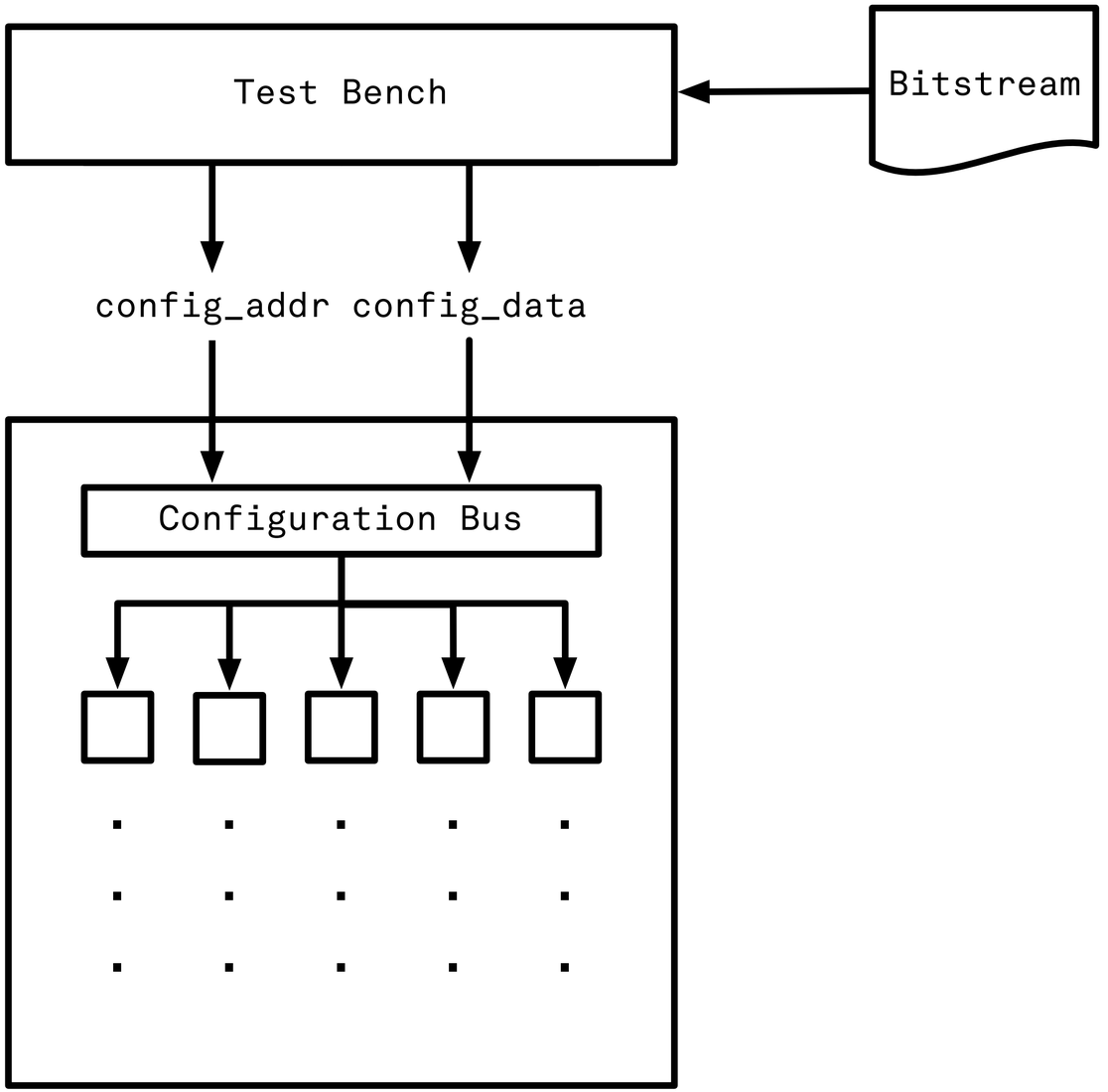
The Static Test Bench Problem

"To reduce **verification** costs, we need to stop **testing** chip instances, and start **testing** chip generators instead."

Corollary

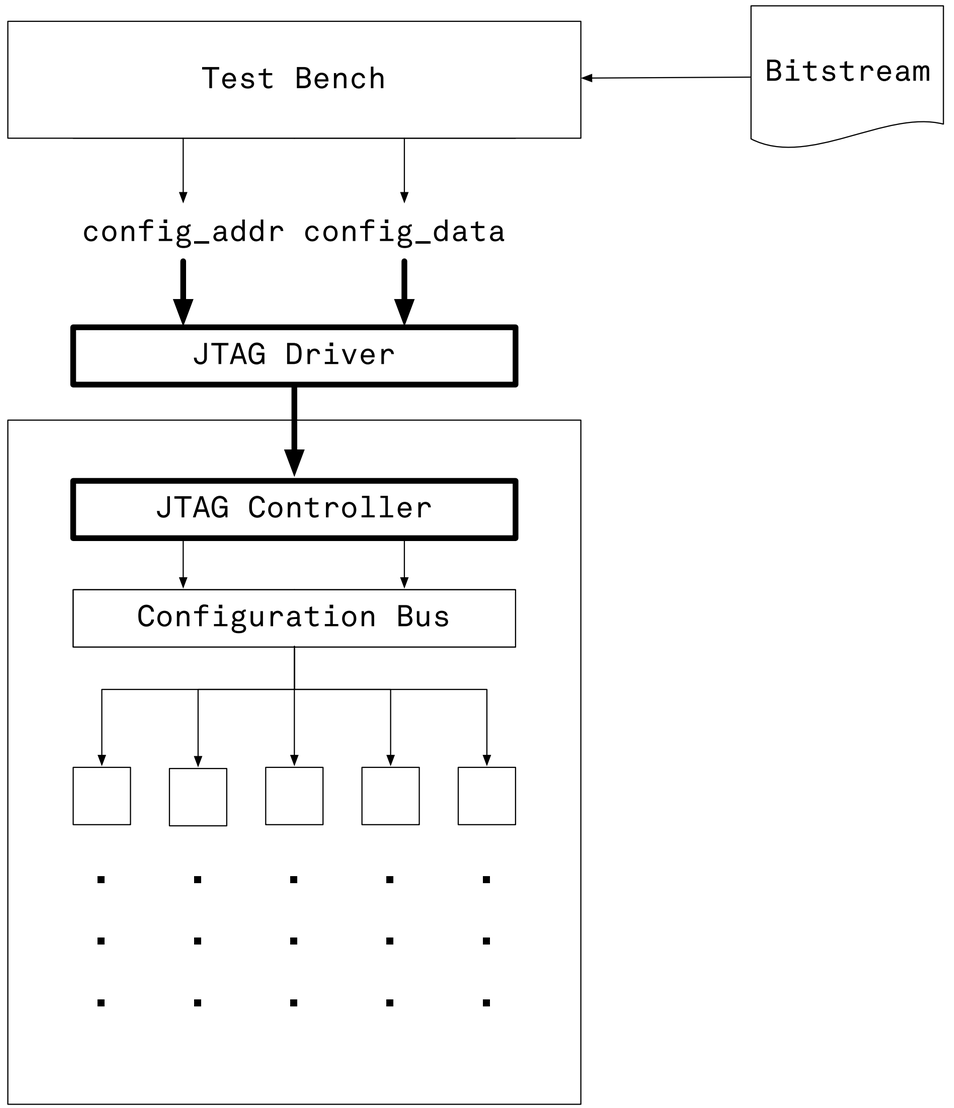
The Static Test Bench Problem

- Initial testing of the CGRA was done using direct access to the configuration bus



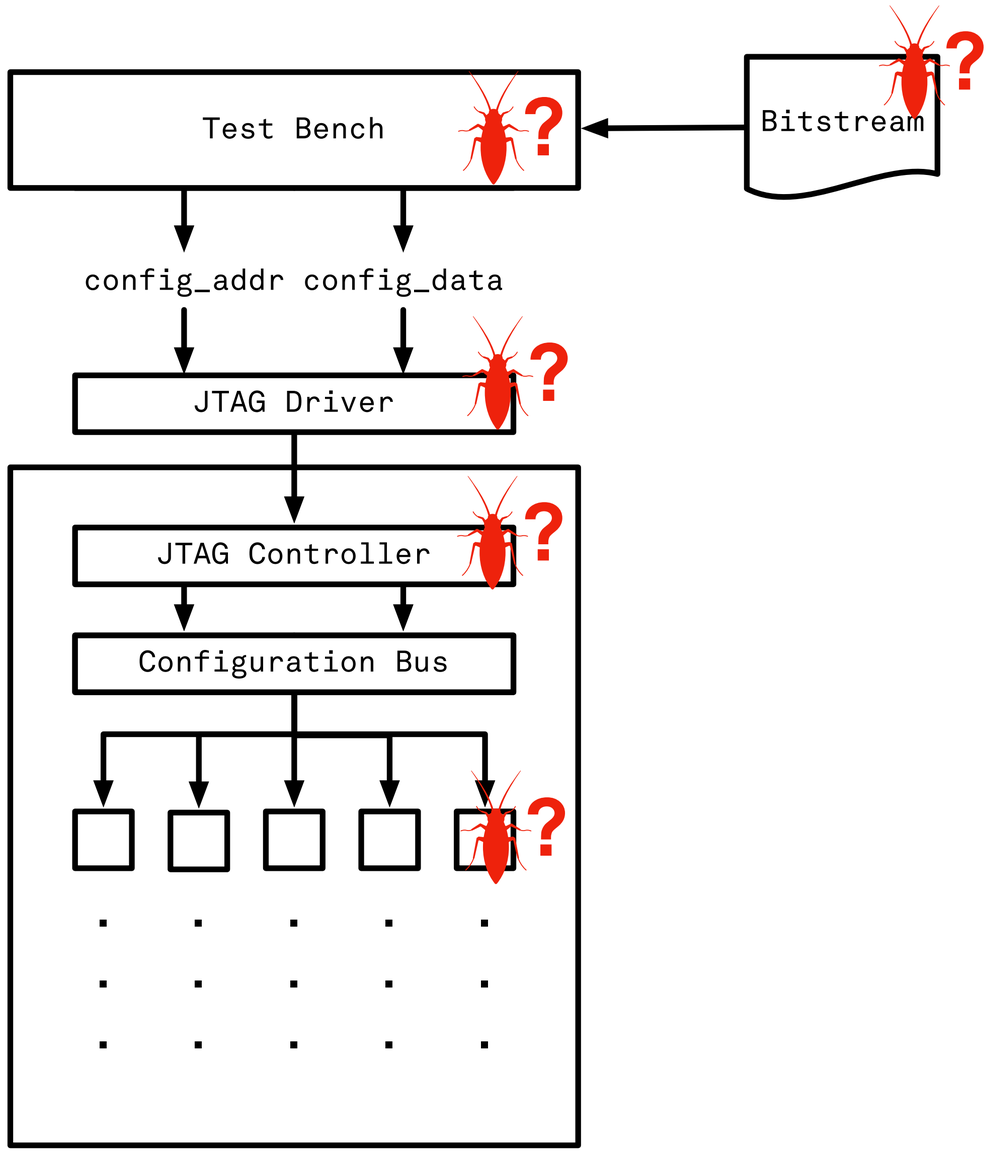
The Static Test Bench Problem

- Added a JTAG controller to the design and a JTAG driver to the test bench



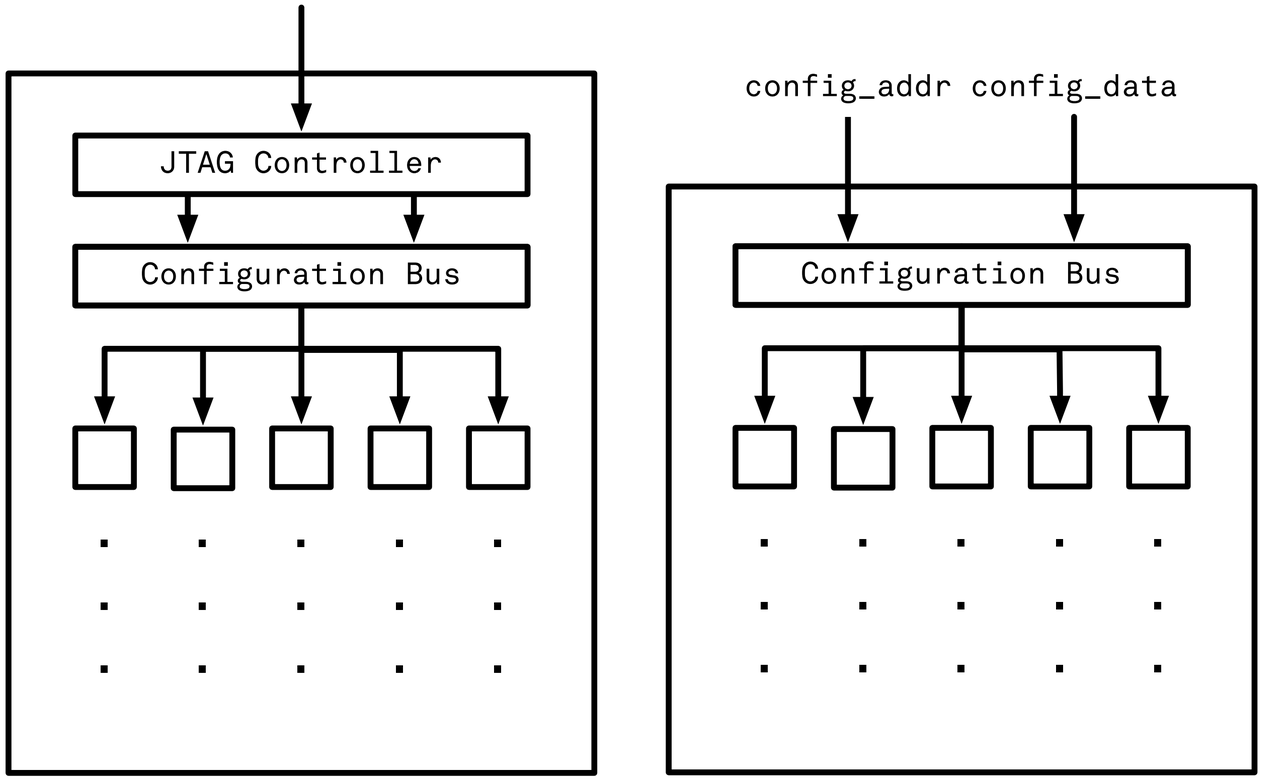
The Static Test Bench Problem

- Eventually, discovered a bug, needed to localize it



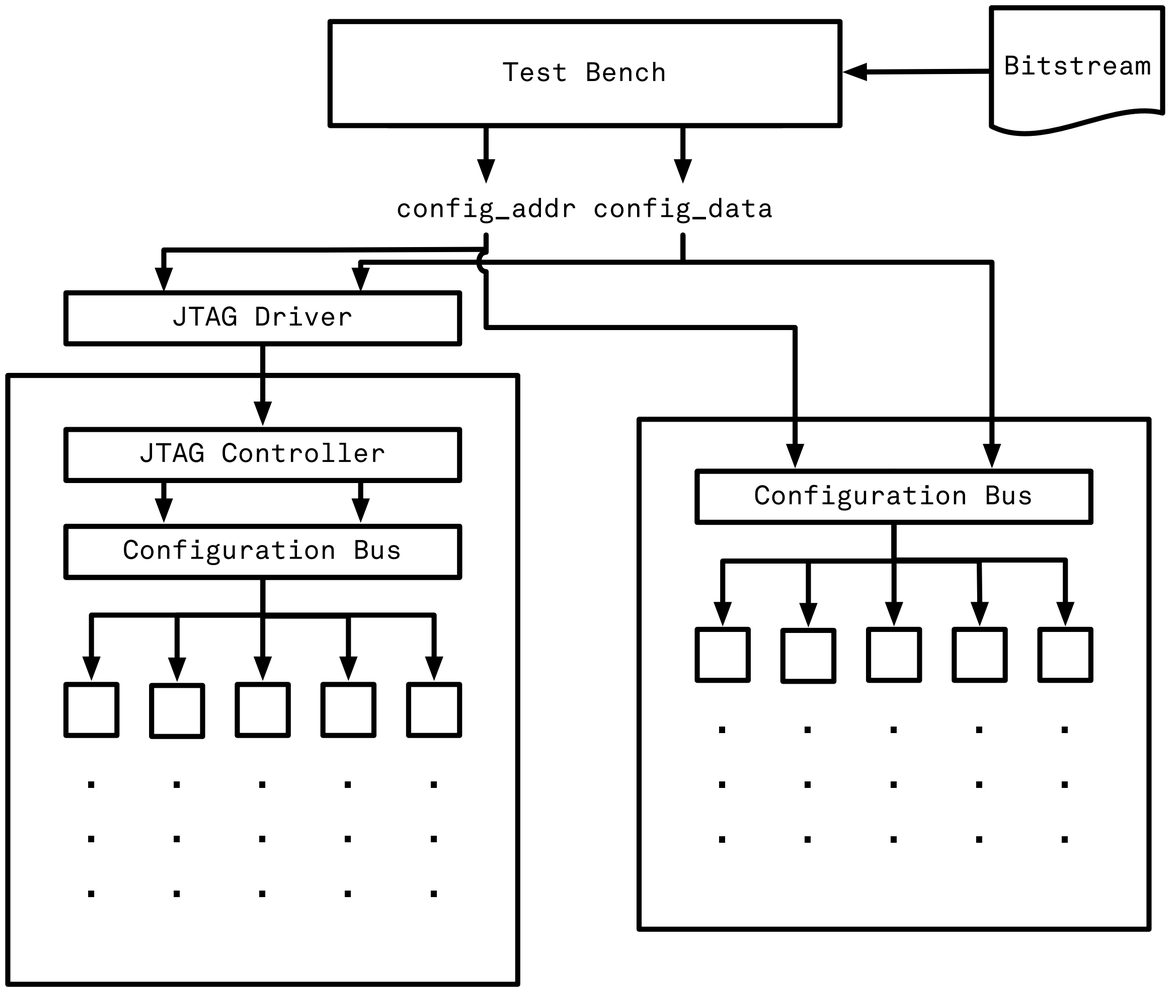
The Static Test Bench Problem

- Because the JTAG was introduced as generator parameter, we could easily construct two variants of the design



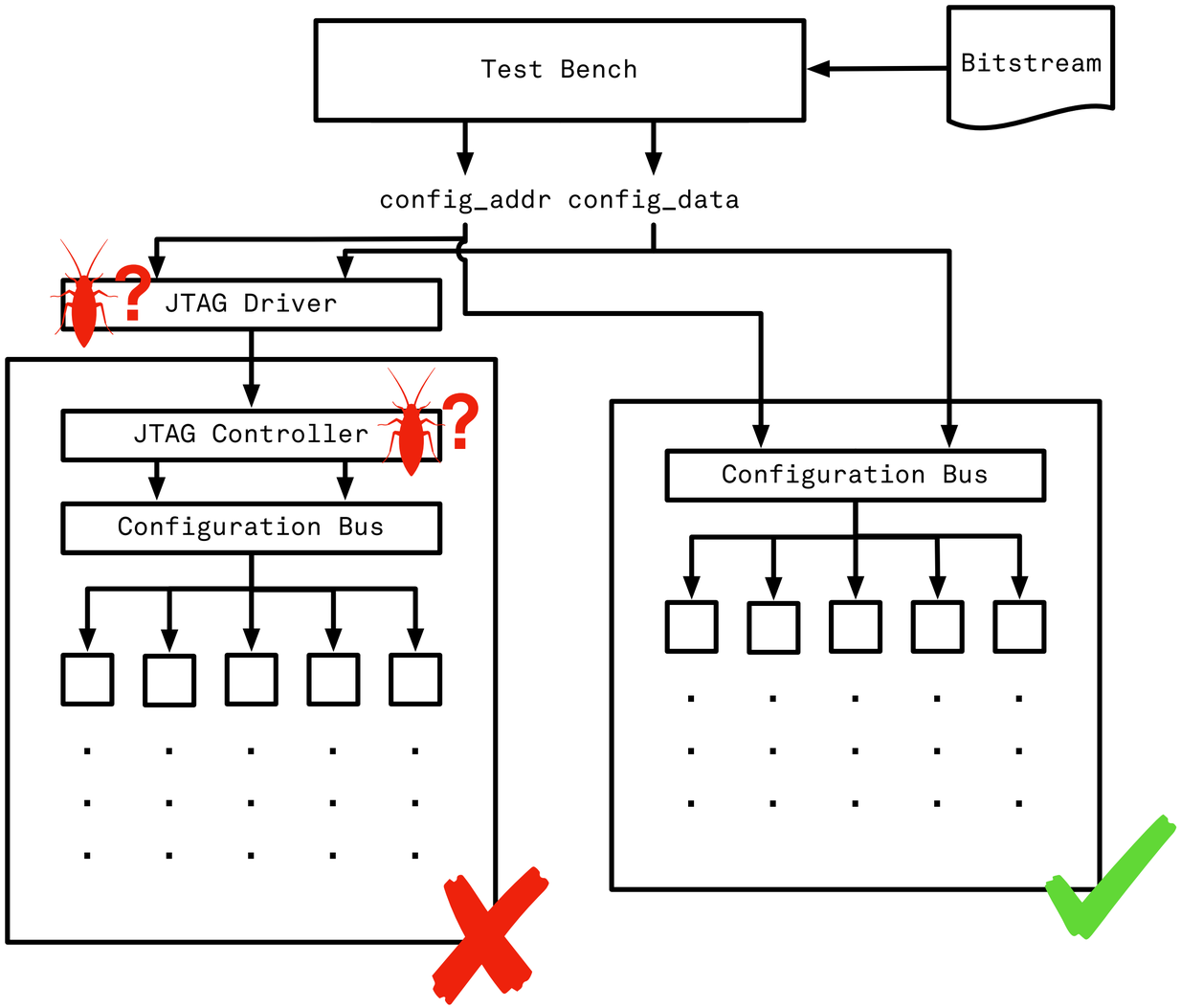
The Static Test Bench Problem

- But now we needed a test bench that worked with both variants



The Generator Testing Problem

- This allowed us to localize bugs to the JTAG Driver/ Controller
- Used #ifdef style macros in our test bench code



Moving Forward: Better Tools for Tapeout 2

- ~~Human readable **only** specification format (Google Doc)~~
  - Specifications are machine readable, executable, and formal
- ~~Entirely open source flow for initial design and verification~~
  - Infrastructure for constructing portable test benches
- ~~Test benches written in C++~~
  - Test benches written in a language with rich metaprogramming facilities (Python)

Fault

https://github.com/leonardt/fault

- Python/magma based framework for agile hardware verification
- Define specifications as Python objects
  - Machine readable, executable, and formal
- Testing abstractions are portable across execution engines
- Parametrization, introspection, and metaprogramming features for testing generators*
- Planned support for UVM components, constrained random and assertions provided as Python libraries

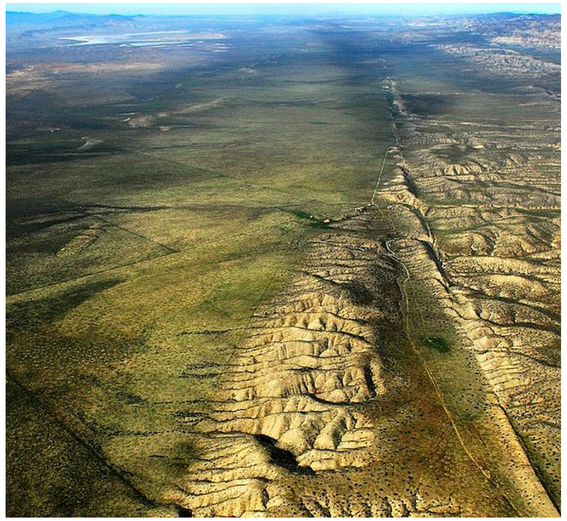
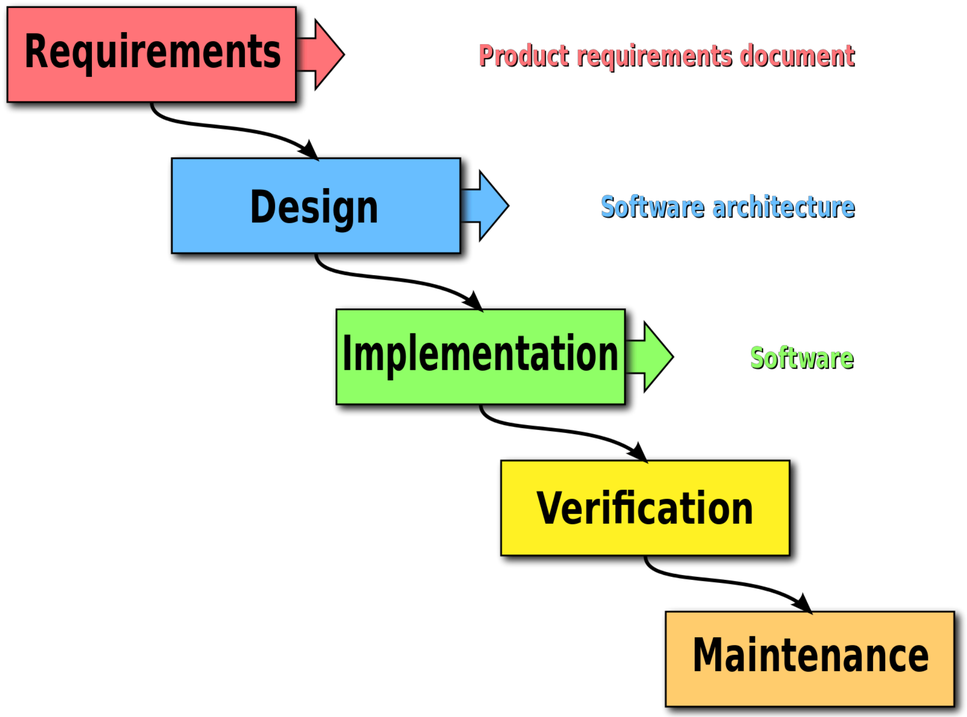


Photo of San Andreas Fault by John Wiley

*** The instances they produce, not the generator programs themselves**

What is Agile Testing in Software?

Waterfall Model

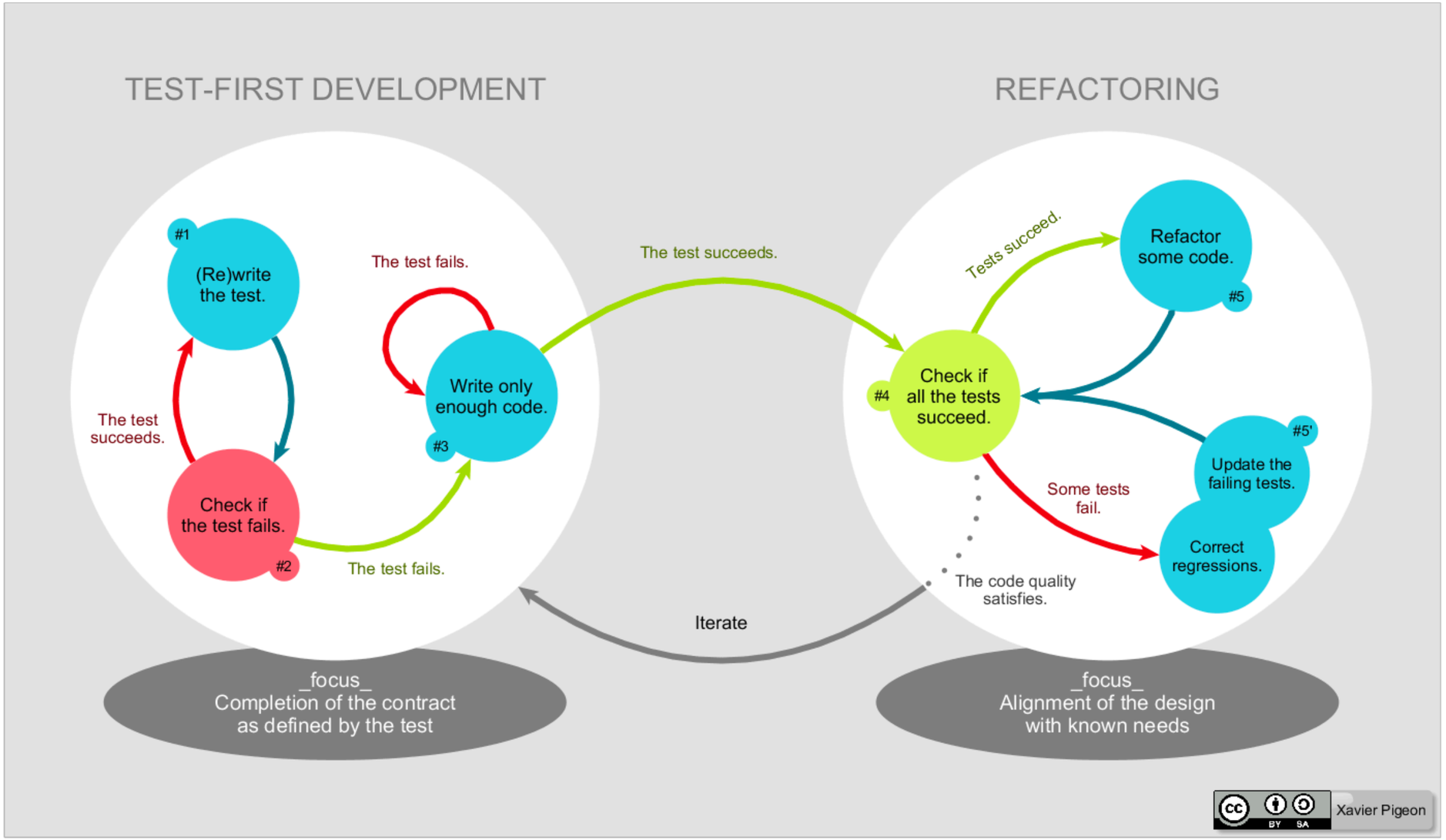


Peter Kemp / Paul Smith

Agile Model

- Testing is not a separate phase, it is as important as coding features
- Test Driven Development

Test Driven Development

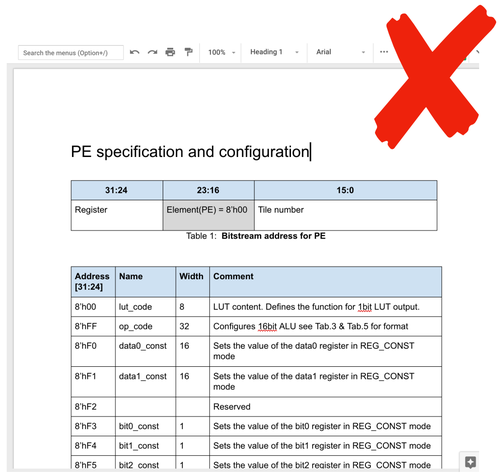


Observation: If we’re going to write the test first, we need to write the specification first

Hardware Specifications as Python Objects

Machine readable

- Python’s rich set of introspection capabilities enable the construction of programs that consume the specification
- Avoid have to change components as the specification evolves



PE specification and configuration

31:24	23:16	15:0
Register	Element(PE) = 8'h00	Tile number

Table 1: **Bitstream address for PE**

Address [31:24]	Name	Width	Comment
8'h00	lut_code	8	LUT content. Defines the function for 1bit LUT output.
8'hFF	op_code	32	Configures 16bit ALU see Tab.3 & Tab.5 for format
8'hF0	data0_const	16	Sets the value of the data0 register in REG_CONST mode
8'hF1	data1_const	16	Sets the value of the data1 register in REG_CONST mode
8'hF2			Reserved
8'hF3	bit0_const	1	Sets the value of the bit0 register in REG_CONST mode
8'hF4	bit1_const	1	Sets the value of the bit1 register in REG_CONST mode
8'hF5	bit2_const	1	Sets the value of the bit2 register in REG_CONST mode

Executable

- Provides a gold model for verification of hardware and software
- Users can interact with the specification to answer questions

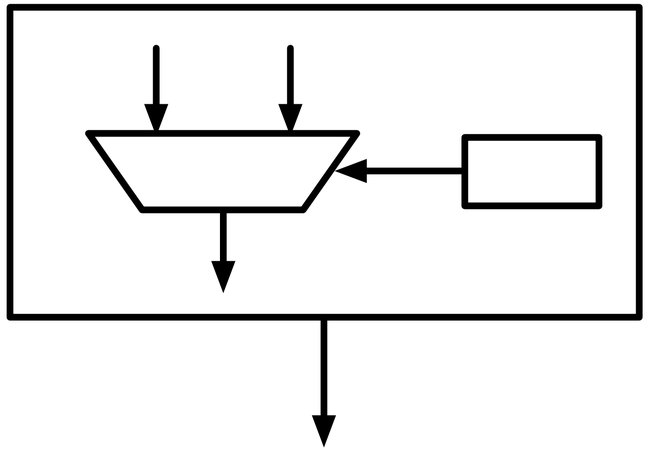
```python
class _SimpleCB(ParentCls):
    def __init__(self):
        super().__init__()
        self.reset()

    def reset(self):
        self.out = fault.UnknownValue
        self.read_data = fault.UnknownValue
        self.configure(CONFIG_ADDR, BitVector(0, 32))

    def configure(self, addr, data):
        self.config[addr] = data

    def __call__(self, *args):
        assert len(args) == num_tracks
        select = self.config[CONFIG_ADDR]
        select_as_uint = select.as_uint()
        if select_as_uint in range(num_tracks):
            return args[select_as_uint]
        return BitVector(0, width)
```

Formal

- Input specification to formal verification tools

Testing with an Executable Specification

DUT



Executable Specification

```
class _SimpleCB(ParentCls):
    def __init__(self):
        super().__init__()
        self.reset()

    def reset(self):
        self.out = fault.UnknownValue
        self.read_data = fault.UnknownValue
        self.configure(CONFIG_ADDR, BitVector(0, 32))

    def configure(self, addr, data):
        self.config[addr] = data

    def __call__(self, *args):
        assert len(args) == num_tracks
        select = self.config[CONFIG_ADDR]
        select_as_uint = select.as_uint()
        if select_as_uint in range(num_tracks):
            return args[select_as_uint]
        return BitVector(0, width)
```

```
cb_tester = ConnectionBoxTester(dut)
cb_tester.reset()
cb_tester.configure(addr, data)
cb_tester.poke(dut.inputs, inputs)
cb_tester.eval()
```

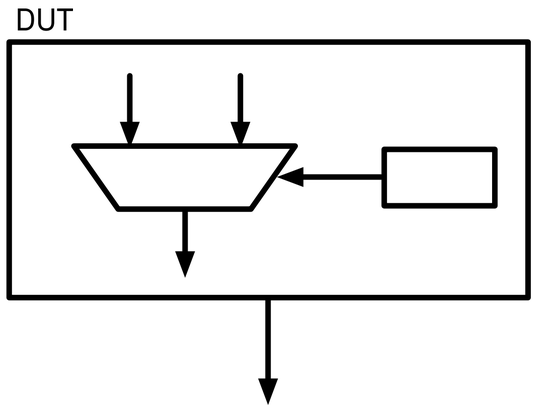
```
cb_spec = ConnectionBoxSpec()
cb_spec.reset()
cb_spec.configure(addr, data)
output = cb_spec(inputs)
```

```
cb_tester.expect(dut.output, output)
```

Testing Abstraction Pattern: UVM Driver

- Tester methods match specification interface, driver lowers method calls to low level signals on input ports

```
class ConfigurationTester(fault.Tester):
    def configure(self, addr, data):
        self.poke(self.clock, 0)
        self.poke(self.reset_port, 0)
        self.poke(self.circuit.config_addr, addr)
        self.poke(self.circuit.config_data, data)
        self.poke(self.circuit.config_en, 1)
        self.step()
```



Executable Specification

```
class _SimpleCB(ParentCls):
    def __init__(self):
        super().__init__()
        self.reset()

    def reset(self):
        self.out = fault.UnknownValue
        self.read_data = fault.UnknownValue
        self.configure(CONFIG_ADDR, BitVector(0, 32))

    def configure(self, addr, data):
        self.config[addr] = data

    def __call__(self, *args):
        assert len(args) == num_tracks
        select = self.config[CONFIG_ADDR]
        select_as_uint = select.as_uint()
        if select_as_uint in range(num_tracks):
            return args[select_as_uint]
        return BitVector(0, width)
```

```
cb_tester = ConnectionBoxTester(dut)
cb_tester.reset()
cb_tester.configure(addr, data)
cb_tester.poke(dut.inputs, inputs)
cb_tester.eval()
```

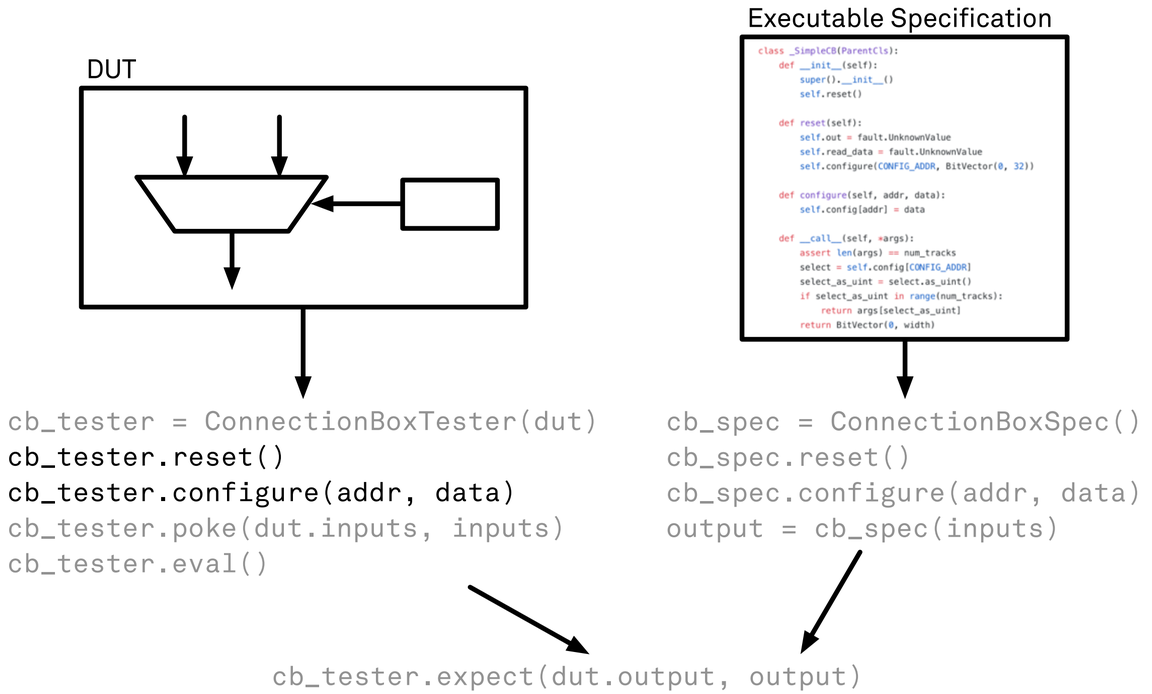
```
cb_spec = ConnectionBoxSpec()
cb_spec.reset()
cb_spec.configure(addr, data)
output = cb_spec(inputs)
```

```
cb_tester.expect(dut.output, output)
```

Modular Testing

- Leverage patterns like inheritance to reuse test components

```
class ConnectionBoxTester(ConfigurationTester,
                          ResetTester):
    pass
```



Formal Verification for the Masses*

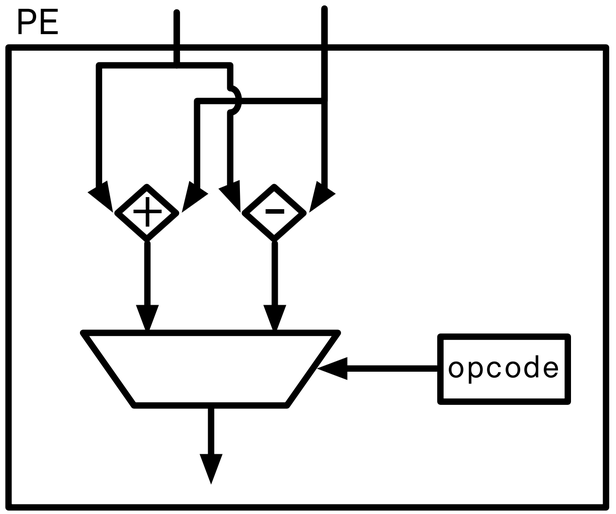
https://github.com/cristian-mattarei/CoSA

- Fault integrates with CoSA (CoreIR Symbolic Analyzer)
- Executable specifications are written using the bit_vector Python library which provides an executable implementation of the SMT-LIB BitVector primitives
- Python embedding means formal specifications are modular and composable

* Quote from Rick Bahr

Formal Verification for the Masses

- Start with an executable specification



Executable Specification

```
[operator.add, operator.sub]
```

```
tester.reset()
tester.configure(0, opcode)
tester.poke(pe.I0, I0)
tester.poke(pe.I1, I1)
tester.eval()
```

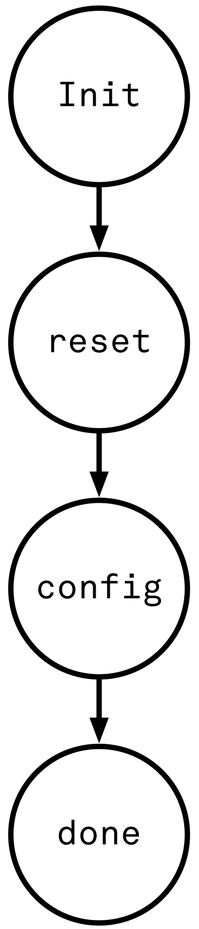
```
pe = ops[opcode]
output = pe(I0, I1)
```

```
tester.expect(dut.output, output)
```

Formal Verification for the Masses

- Produce an explicit transition system to configure the state of the device

```
class ConfigurationChecker(fault.Checker):
    def configure(self, addr, data):
        self.poke(self.clock, 0)
        self.poke(self.reset_port, 0)
        self.poke(self.circuit.config_addr, addr)
        self.poke(self.circuit.config_data, data)
        self.poke(self.circuit.config_en, 1)
        self.step()

checker = PEChecker(pe)
checker.reset()
checker.configure(0, opcode)
```




```
# INIT
I: self.reset = 0_1
I: self.config_en = 0_1
I: conf_done = 0_1
I: self.config_addr = 0_1
I: self.config_data = 0_2

# STATES


# S0 -> S0a: reset
S0: self.config_addr = 0_1
S0: self.config_data = 0_2
S0: self.config_en = 0_1
S0: self.reset = 0_1
S0: conf_done = 0_1

S0a: self.config_addr = 0_1
S0a: self.config_data = 0_2
S0a: self.config_en = 0_1
S0a: self.reset = 1_1
S0a: conf_done = 0_1

S1: self.config_addr = 0_1
S1: self.config_data = 0_2
S1: self.config_en = 0_1
S1: self.reset = 0_1
S1: conf_done = 0_1

# S1a -> S2a: config
S1a: self.config_addr = 0_1
S1a: self.config_data = 0_2
S1a: self.config_en = 1_1
S1a: self.reset = 0_1
S1a: conf_done = 0_1

S2: self.config_addr = 0_1
S2: self.config_data = 0_2
S2: self.config_en = 1_1
S2: self.reset = 0_1
S2: conf_done = 0_1

S2a: self.config_addr = 0_1
S2a: self.config_data = 0_2
S2a: self.config_en = 0_1
S2a: self.reset = 0_1
S2a: conf_done = 1_1

# TRANSITIONS
I -> S0
S0 -> S0a
S0a -> S1
S1 -> S1a
S1a -> S2
S2 -> S2a
S2a -> S2a
```

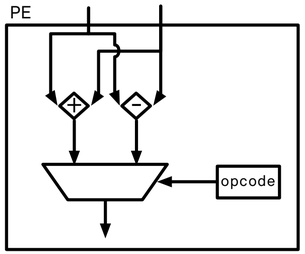
Formal Verification for the Masses

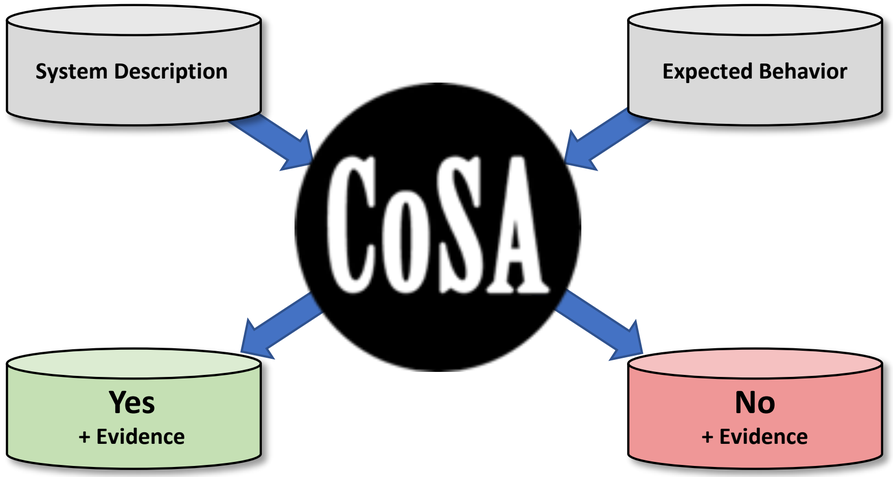
- Produce an expected formula based on the configuration

```
checker = PEChecker(pe)
checker.reset()
checker.configure(0, opcode)
checker.expect(pe.out, op(checker.peek(pe.I0),
                          checker.peek(pe.I1))
```

```
formula: (conf_done = 1_1) -> ((self.I0 + self.I1) = self.O)
```

Formal Verification for the Masses



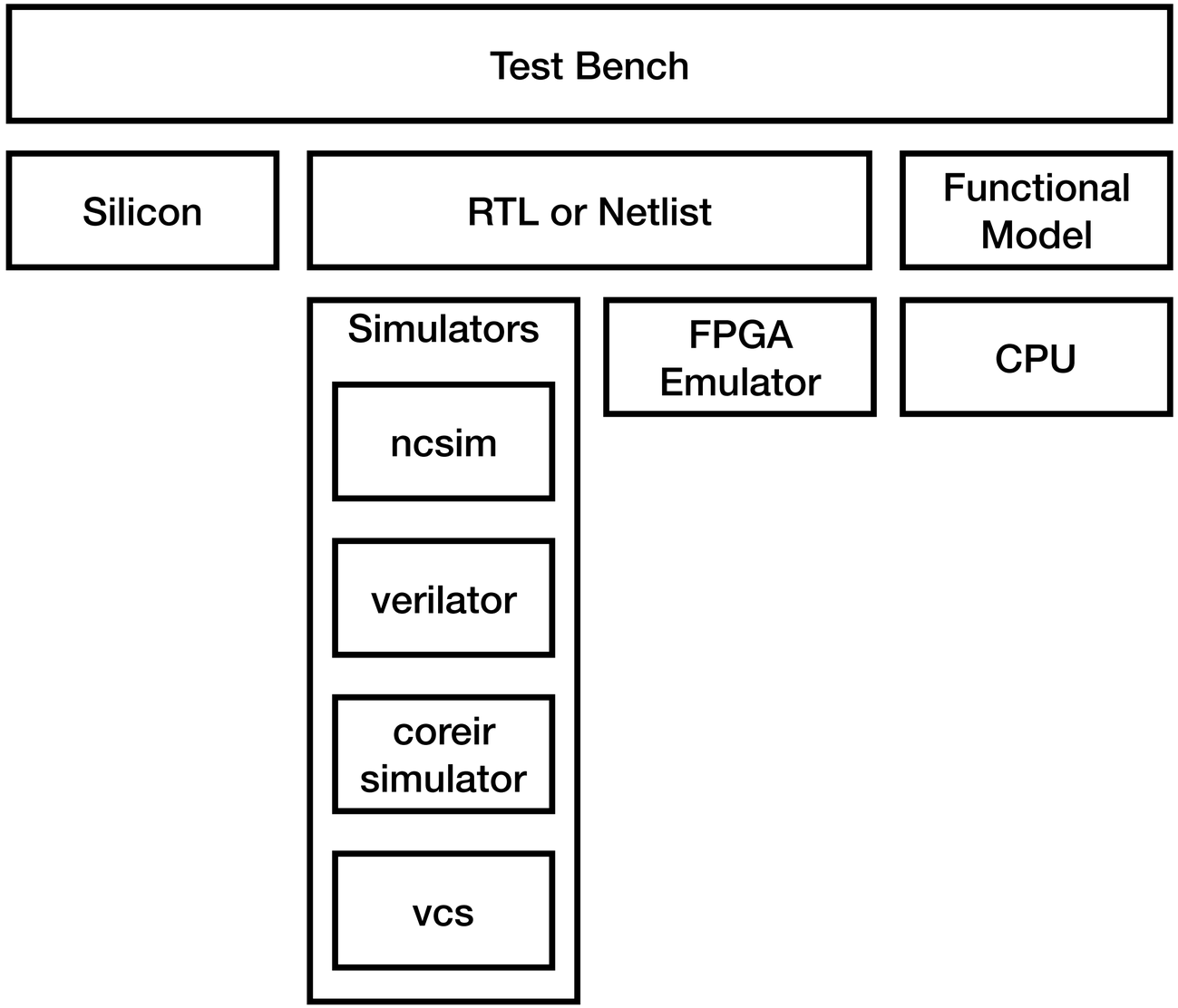



```
# INIT
I: self.reset = 0_1
I: self.config_en = 0_1
I: conf_done = 0_1
I: self.config_addr = 0_1
I: self.config_data = 0_2

# STATES


# S0 -> S0a: reset
S0: self.config_addr = 0_1
S0: self.config_data = 0_2
S0: self.config_en = 0_1
S0: self.reset = 0_1
S0: conf_done = 0_1

S0a: self.config_addr = 0_1
S0a: self.config_data = 0_2
S0a: self.config_en = 0_1
S0a: self.reset = 1_1
S0a: conf_done = 0_1

S1: self.config_addr = 0_1
S1: self.config_data = 0_2
S1: self.config_en = 0_1
S1: self.reset = 0_1
S1: conf_done = 0_1

# S1a -> S2a: config
S1a: self.config_addr = 0_1
S1a: self.config_data = 0_2
S1a: self.config_en = 1_1
S1a: self.reset = 0_1
S1a: conf_done = 0_1

S2: self.config_addr = 0_1
S2: self.config_data = 0_2
S2: self.config_en = 1_1
S2: self.reset = 0_1
S2: conf_done = 0_1

S2a: self.config_addr = 0_1
S2a: self.config_data = 0_2
S2a: self.config_en = 0_1
S2a: self.reset = 0_1
S2a: conf_done = 1_1

# TRANSITIONS
I -> S0
S0 -> S0a
S0a -> S1
S1 -> S1a
S1a -> S2
S2 -> S2a
S2a -> S2a
```

formula: (conf_done = 1_1) -> ((self.I0 + self.I1) = self.O)

Test Portability

- Portable test benches foster innovation in the simulation
  - Easily migrate to a new simulator
- Easily scale verification efforts as compute demand increases
- Reuse verification effort during bringup



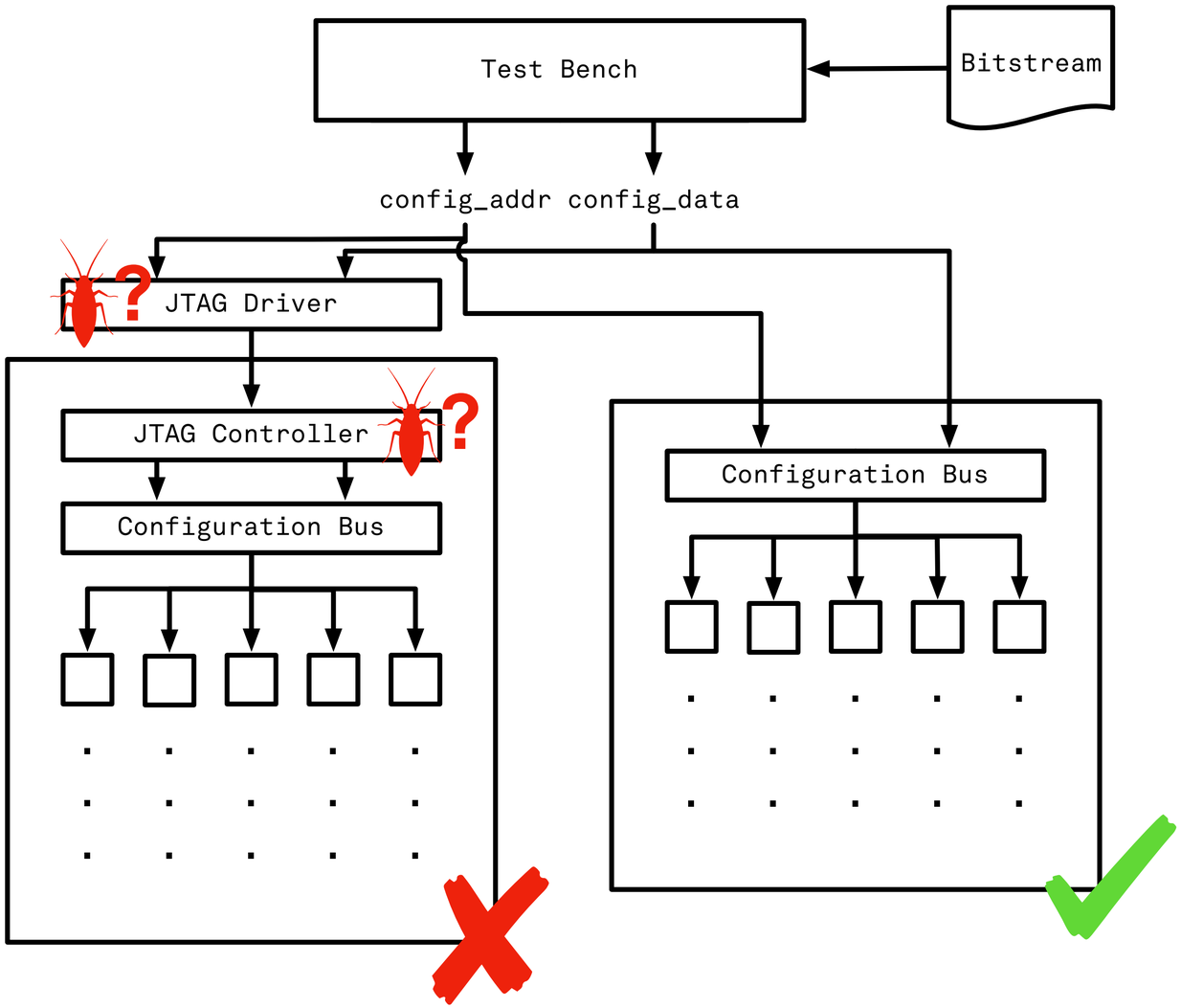
Portability Example: Using Different Simulators

```
...
cb_tester.expect(dut.output, output)

cb_tester.compile_and_run(target="verilator")
cb_tester.compile_and_run(target="coreir”)
```

Simply specify the target when executing the test

Metaprogramming Test Benches

- Leverage flexibility in generators to improve debugging productivity



Metaprogramming Test Benches

- For each instruction in the bit stream, execute the appropriate configuration

```
for (int i = 0; i < {len(config_data_arr)}; i++) {{
    {run_config}
}}
```

- How this is done depends on whether the chip has been generated using the JTAG or not

Metaprogramming Test Benches

- Test bench generator accepts the same parameter as the circuit generator

```
if (args.use_jtag):
    run_config += f"""
        jtag.write_config(config_addr_arr[i],config_data_arr[i]);
    """
else :
    run_config += f"""
        {top_name}->config_data_in = config_data_arr[i];
        {top_name}->config_addr_in = config_addr_arr[i];
        next({top_name})
    """
```

Use JTAG Driver

Directly poke configuration bus

Conclusion

- Fault: Python/magma based framework for agile hardware verification
  - Specifications are machine readable, executable, and formal
  - Test benches are portable
  - Test benches are metaprogrammed in Python

https://github.com/leonardt/fault

Photo of San Andreas Fault by John Wiley

Thanks!