Harnessing the power of property-based testing in FPGA design and verification

Christiaan Baaij, QBayLogic

E&A ELECTRONICS & APPLICATIONS '23

26 T/M 28 SEPTEMBER '23
JAARBEURS UTRECHT

Testing is hard

- Hardware, software, doesn't matter - same principles apply
  - *Does function X behave like Y?*
- Functions / components have a habit of becoming complex
  - *Did we test enough?*

Figure 1: average function in production code

Typical tests

```
assert_equal(3 + 5, 8)
assert_equal(1001 + 3, 1004)
assert_equal(1001 + (-1), 1000)
…
```

- Did you cover all edge cases?
- Are the properties you test for clear?
- Hard to think of a set of tests!

Computer aided testing

- We want computers to help us think of test cases

- Cue randomized testing:
  a. Define a property a design should adhere to

  b. Generate random input, test design upholds property

  c. On error: report broken design/property to user
     On success: repeat step b

Today

- Discuss challenges with randomized testing
- Discuss “Hedgehog” approach used by many Clash designers

Clash

1. BSD2 licenced **Haskell** to **Verilog/VHDL** compiler
   - **Haskell**: a functional programming language
   - **Verilog/VHDL**: Industry’s hardware description languages

2. A standard library for writing digital circuits

Clash in production:

- Myrtle.ai: neural network inference accelerators
- Google: R&D platform self-synchronizing computer networks
- LumiGuide: bicycle parking management ProRail train stations

Haskell

1. High-level, statically typed, compiled, general purpose language
2. Expressive type system
3. Functional: encourages thinking what, not how
   a. Separation of “pure” and “side-effect” code
4. Ecosystem with
   a. Build tools
   b. Package repositories
   c. Many high-quality libraries
5. Haskell in production:
   a. ShellCheck: Bash linter
   b. Sigma: Facebook spam filter
   c. Chordify: (online) audio analysis

Clash+Haskell

- Combining state of the art software tooling with hardware design *in the same language*

- Pioneer in constrained random testing:
  - 2006: QuickCheck
  - 2017: **Hedgehog** <= today's focus
  - 2023: Falsify

Design under test

```
myShiftL ::
  Bitvector 16 ->    Arg 1:  bits to be shifted
  Int ->             Arg 2:  number of positions to shift
  Bitvector 16       Result: shifted bits
```

```
clashi> myShiftL  0b0000_0000_0000_0010  3
0b0000_0000_0001_0000

clashi> myShiftL  0b0000_0000_0000_0010  8
0b0000_0010_0000_0000

clashi> myShiftL  0b0000_0000_0000_0010  0
0b0000_0000_0000_0010
```

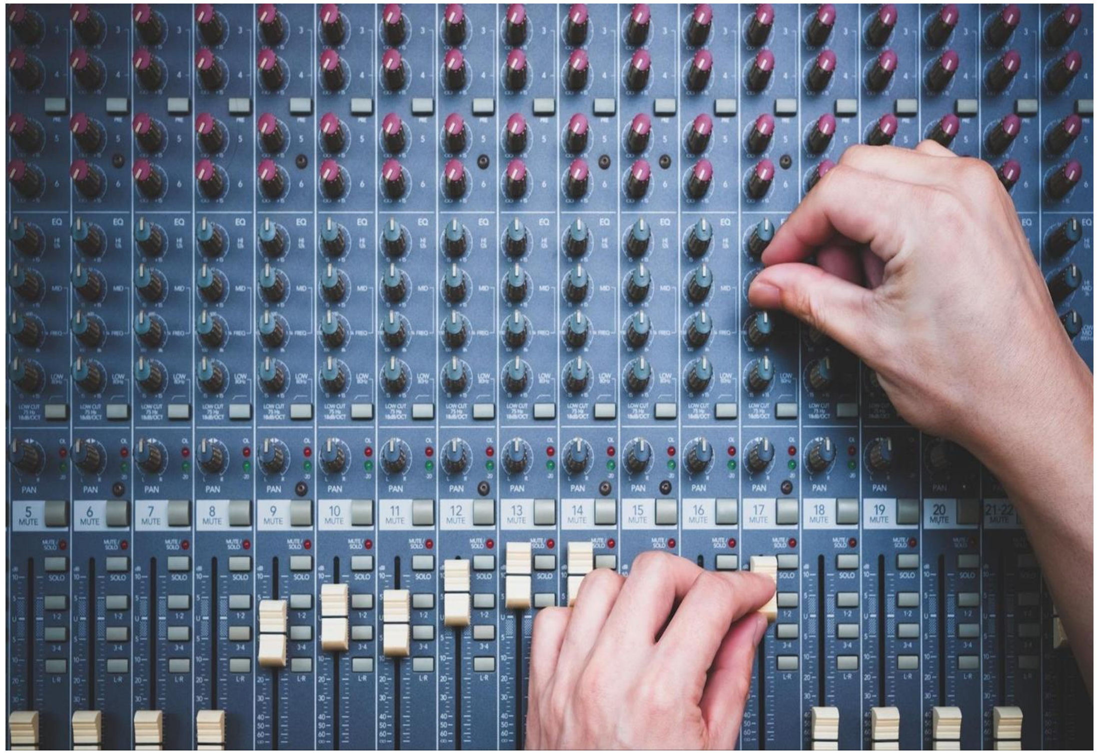
Dials

Our design has two dials:

- Bitvector to shift
- Number of positions to shift by

Our design will have two obvious bugs:

- Doesn’t work for negative shifts
- Doesn’t actually shift, but rotates



Test code

```
prop_idWithShiftL :: H.Property
prop_idWithShiftL = H.property $ do
    bv       <- H.forAll $ Gen.integral $ Range.linearFrom 0 0      maxBound
    shiftByN <- H.forAll $ Gen.integral $ Range.linearFrom 0 (-100) 100

    goldenShiftL bv shiftByN === myShiftL bv shiftByN
```

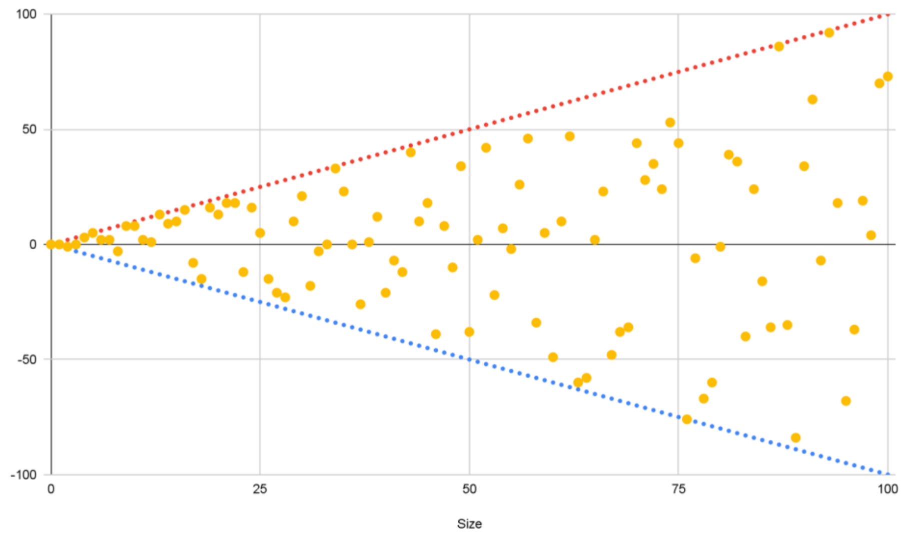
What do we expect?

- *Any* negative number fails
- We generate shifts from -100 to 100
- Maybe it will fail with:
  - -52
  - 0001_0011_1001_1000

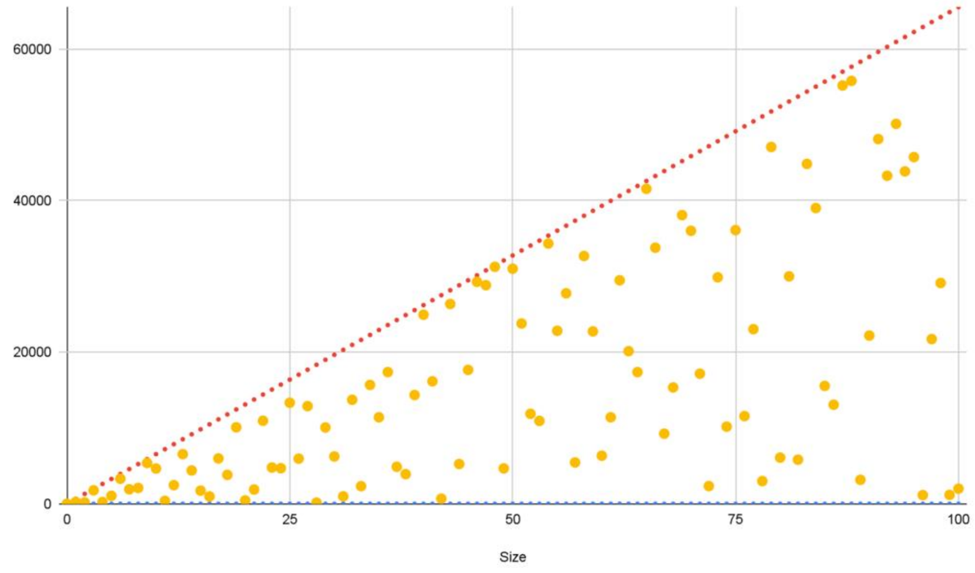
Growing inputs

Hedgehog doesn’t pick *just* at random, it turns the dials

number of positions to shift



bitvector value



What do we expect?

- *Any* negative number fails
- We generate shifts from -100 to 100
- Hedgehog will slowly grow these values
- Maybe it will fail with:
  - -52
  - 0001_0011_1001_1000

Actual error

```
┏━━ tests/Tests/HwAccel/Shifter.hs ━━━
    18 ┃ prop_idWithShiftL :: H.Property
    19 ┃ prop_idWithShiftL = H.property $ do
    20 ┃   bv       <- H.forAll $ Gen.integral $ Range.linearFrom 0 0      maxBound
       ┃   │ 0 b0000_0000_0000_0001
    21 ┃   shiftByN <- H.forAll $ Gen.integral $ Range.linearFrom 0 (-100) 100
       ┃   │ -1
    22 ┃  goldenShiftL bv shiftByN === myShiftL bv shiftByN
       ┃  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
       ┃   │ ━━━━ Failed (- lhs) (+ rhs) ━━━━
       ┃   │ - 0 b0000_0000_0000_0000
       ┃   │ + 0 b0000_0000_0000_0010
```

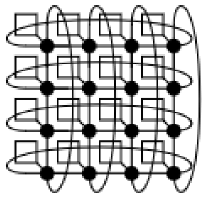
Shrinking!

- Once Hedgehog hits an error it will start shrinking inputs
- It will try to minimize as many “dials” as it can
- Result will be something very close to where your bug lives

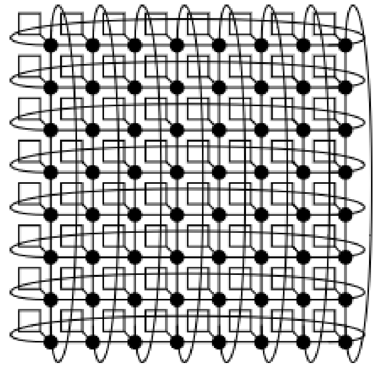
Real-life example: bittide project

- https://github.com/bittide/bittide-hardware
- Hardware support to enable a distributed system architecture (data centers) based on the idea of synchronous, ehead-of-time scheduling.
- System-on-Chip with many protocols and Network-on-Chip (NoC) like features.
- Example properties:
  - The NoC switch does not lose packets
  - Concatenating an A->B bus-protocol converter to an B->A protocol converter gives me an A->A component
  - Interconnect for AXI4 can properly route transactions for any valid memory map

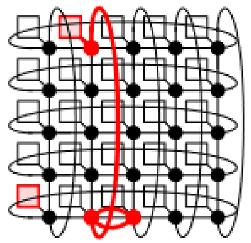
Example of Using PyH2 to Test Torus OCN



- 4x4 torus, minimal routing
- Passes directed tests
- Passes "standard" random tests



- 8x8 torus, minimal routing
- Passes directed tests
- Fails "standard" random test with
  - 100s of cycles
  - 1000s of packets



- PyH2 is used to test torus
- PyH2 spent about ~20 min trying many different network sizes, packets, and payloads

- **PyH2 used auto-shrinking to find a small failing test case which could be debugged in a few minutes**
  - single packet
  - zero payload
  - design to 5x5 Torus

```
always_comb @(*) begin
  if (pkt_dst_x < pos_x) begin
    west_dist = pos_x - pkt_dst_x;
    east_dist = last_col_id
      - pos_x + 'd1 + pkt_dst_x;
  end
  else begin
    west_dist = last_col_id
      + pos_x + 'd1 - pkt_dst_x;
    east_dist = pkt_dst_x - pos_x;
  end
end
```

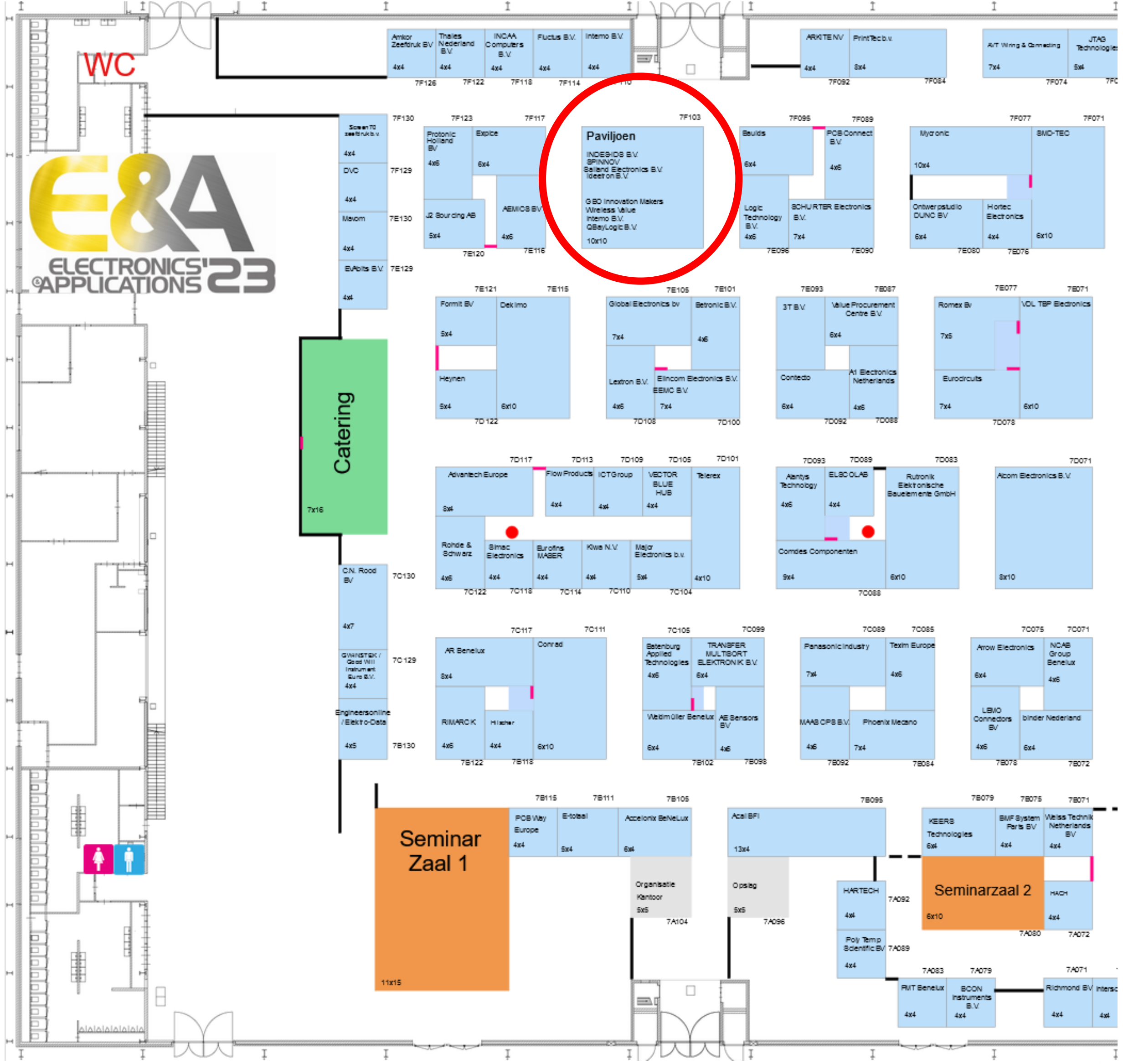
https://eri-summit.darpa.mil/docs/ERIsummit2019/posh/24POSH%20Princeton%20Website.pdf

Other test generation approaches

- Property-based testing versus constraint-random testing:
  - Property-based testing does not draw all random data before-hand, making it possible to leverage runtime information to guide random data generation
  - Can automatically shrink the failing test case to a minimal failing case once a bug is discovered
- Coverage-directed test generation (CDG) is complementary to PBT

Closing thoughts

- Higher confidence in functional correctness
- Higher likelihood you’ll meet that deadline
- Still.. not a silver bullet, you have to think about:
  - Properties
  - Generators
- This talk covered aspects of testing during the design development phase, later talks will cover testing in a completely different light: testing after manufacture.



Standnummer 7F103

christiaan@qbaylogic.com

26 T/M 28 SEPTEMBER '23 JAARBEURS UTRECHT

Harnessing the power of property-based testing in FPGA design and verification

BACKUP SLIDES

Clash’s features

- Inherited from Haskell
  - Extensive type system
  - Algebraic Data Types
  - Package management and build tools
  - Optimizing compiler
  - State-of-the-art testing libraries
  - REPL
  - Polymorphism, metaprogramming, higher-order functions, compile time evaluations, …

Clash’s features

- Clash the standard library
  - Multi-clock designs without accidental clock-domain crossings
  - Clear separation of *stateful* and *combinatorial* logic
  - Type level pipeline delay tracking
  - SVA/PSL support
  - Safe multiplication / subtraction / …
  - Cycle accurate simulation in Haskell

- Clash libraries:
  - \`clash-cores\`: pre-made cores, Xilinx primitives
  - \`clash-protocols\`: easy protocol composition