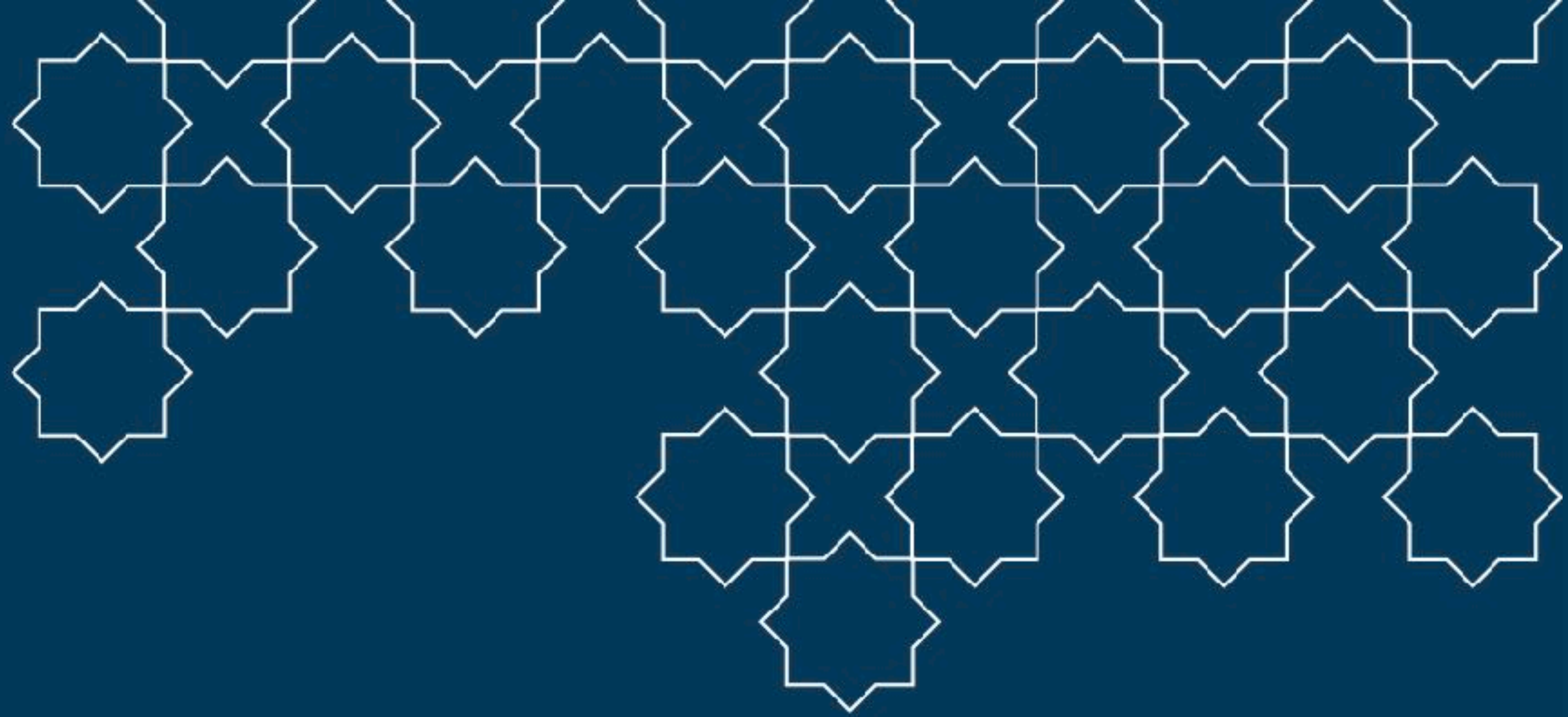
The American University in Cairo

School of Sciences and Engineering



Fault

Open-Source EDA's Missing DFT Toolchain

Mohamed Gaber · Free Silicon Conference 2024 · June 21st, 2024

About the Authors

- Manar Abdelatty
  - BSc in Computer Engineering, The American University in Cairo
  - Ph.D. Student, Brown University

- Mohamed Gaber
  - Graduate Student, The American University in Cairo
  - Senior EDA Engineer, Efabless Corporation

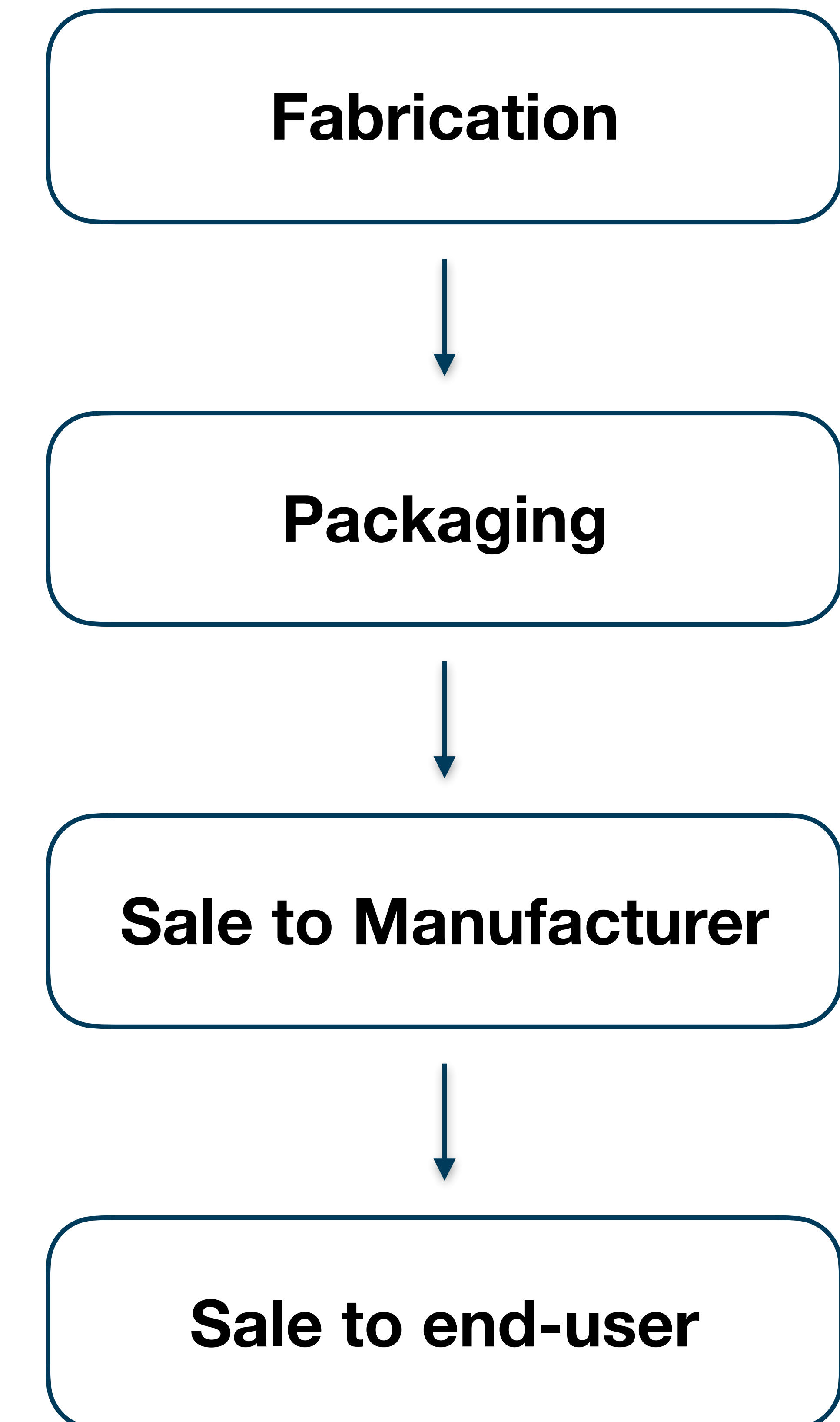
- Mohamed Shalan
  - Associate Professor, The American University in Cairo
  - Head of EDA/IC Design, Efabless Corporation

Background

- Chip fabrication is inherently imperfect
  - Defects in EDA tools
  - Defects in manufacturing

- Manufacturing variations
  - Shorts or opens where they aren't expected
  - Timing variations

- The **sooner** you catch the error, the better
  - "Rule of ten" - at every step, up to and including shipping to an end user, the cost of remedy is multiplied by
    - If you don't, you or your customer are recalling the device at great cost



Background

- The solution: test immediately after fabrication, discarding bad chips
  - Apply inputs (called **test vectors**) to a design under test (DUT)
  - Compare results with a known-good golden model (GM)
  - A mismatch indicates a bad chip

- The tools to generate these patterns are broadly known as **design-for-testing**, or DFT tools
  - Synopsys TestMAX DFT
  - Cadence Modus

- Good DFT tools try to optimize two metrics
  - **Total test time**, the lower the better
  - **Coverage** of possible faults in a chip

Status Quo Ante

- Extremely robust proprietary DFT utilities
  - Available at nominal cost to research institutions
  - Cost-prohibitive to startups

- Projects like Yosys, OpenROAD and the Google/Skywater Open Source PDK breaking barriers to ASIC design
  - Derivative projects including OpenLane

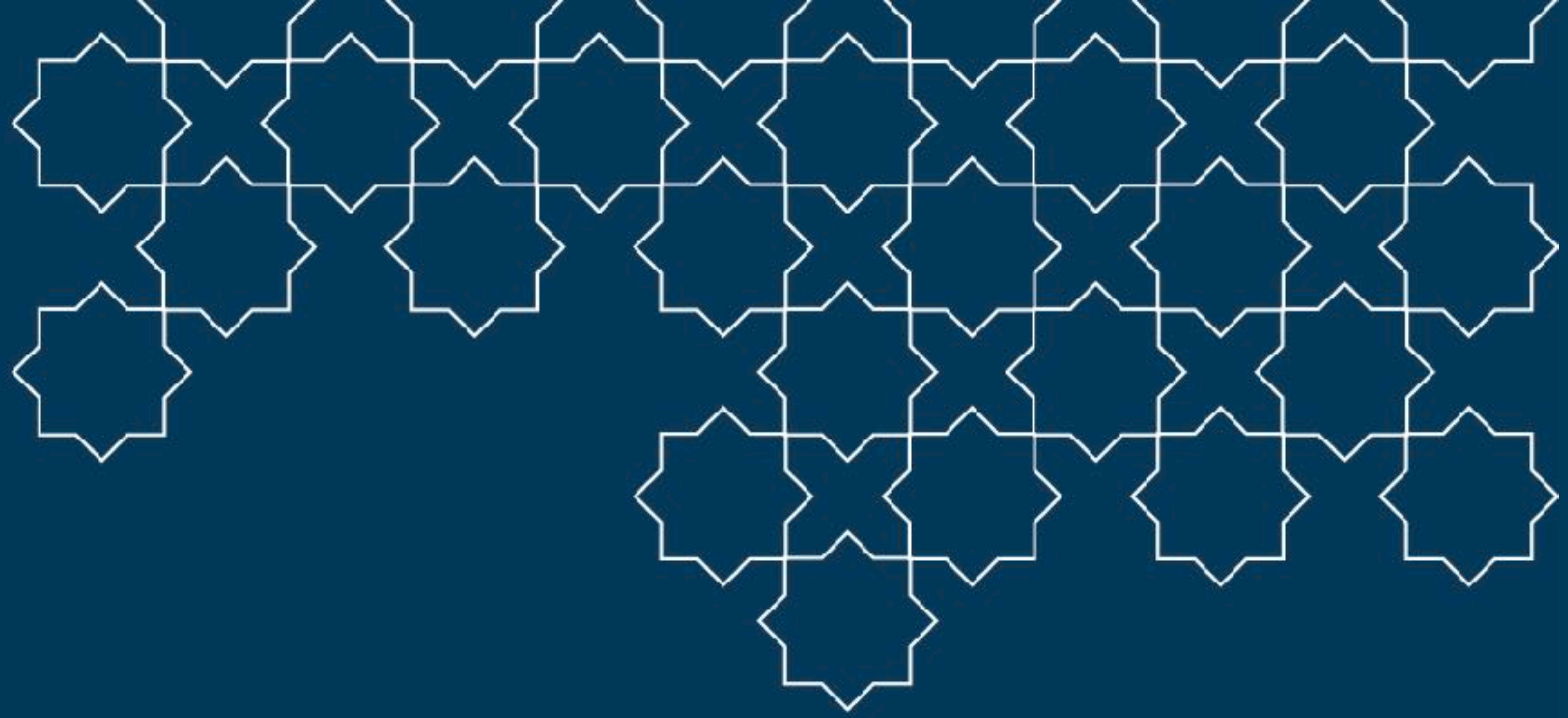
- But crucially, no DFT utilities!

Our Contribution

- An open-source DFT toolchain leveraging multiple existing open-source utilities

- Provides major steps for DFT implementation
  - Automatic Test Pattern Generation (ATG) and Compaction
  - Scan-chain Stitching
  - Insertion of a IEEE 1149.1 TAP controller
  - Verification at all steps

- Simply called "Fault."



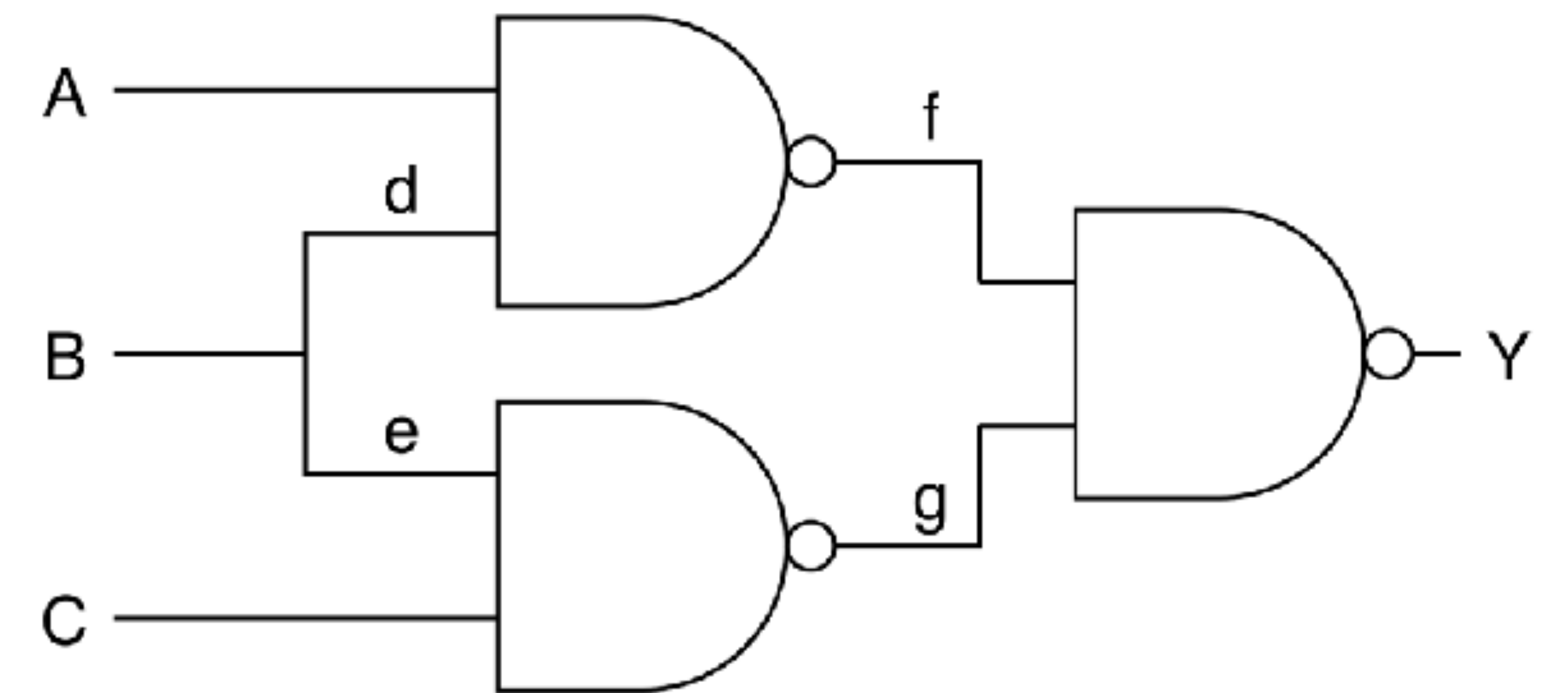
Architecture

The Fault Model

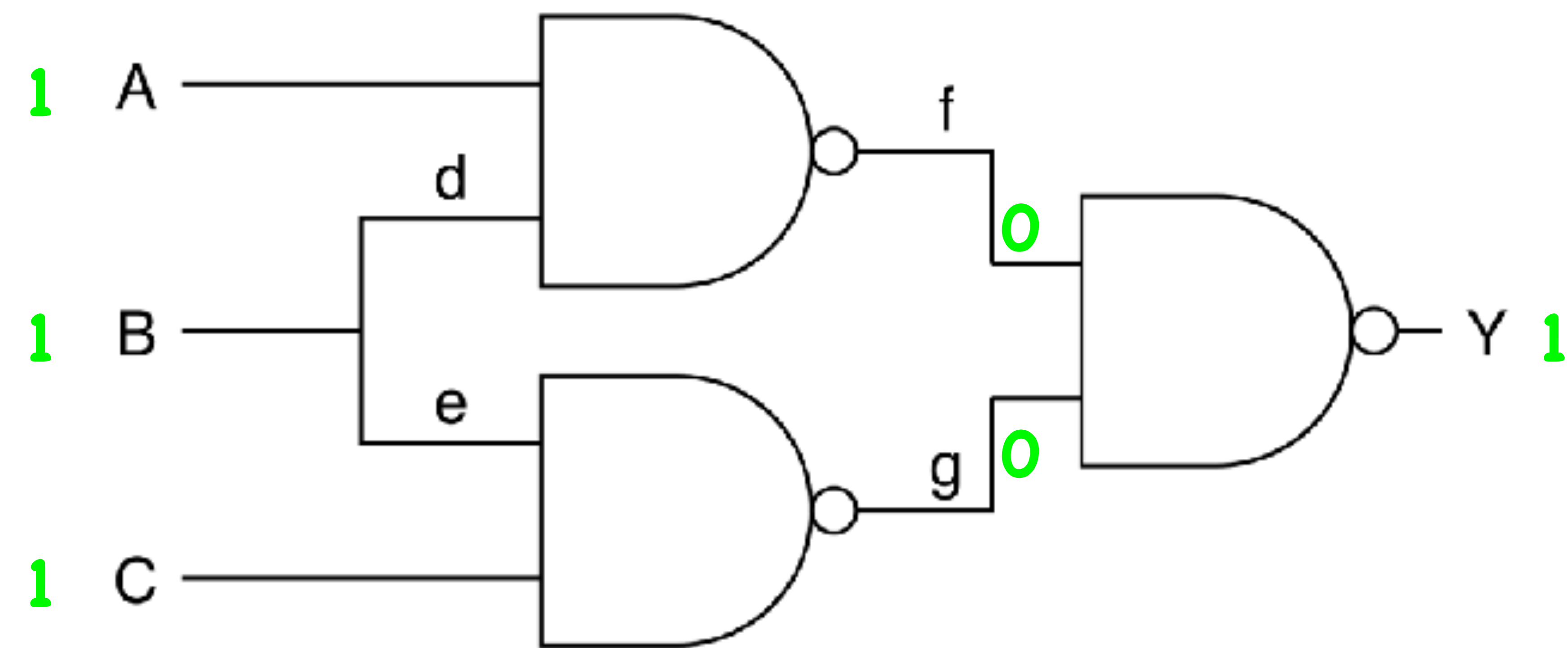
- It is impossible to simulate all defects because there's just too many of them
  - This necessitates using a model for the faults somehow

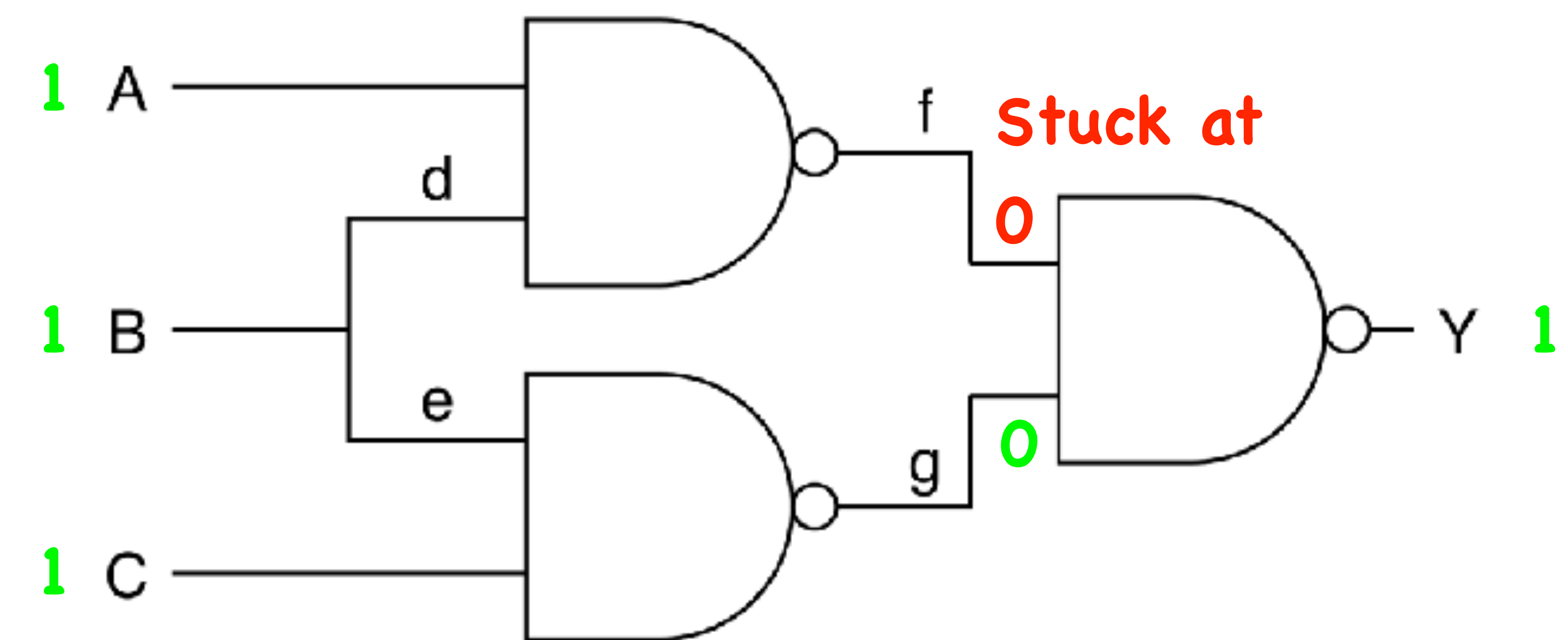
- There are many models, but none of them can model the behavior of all possible defects

- Simple but effective model: the **stuck-at** Fault model
  - Assume any input or output to internal cells or macros are **fault sites**
  - Assume any fault site can be bridged to 1 or 0
    - **Stuck at 0**, **Stuck at 1**
    - Both are two separate faults that need to be covered
  - If we can find a test vector that causes the output to be different when this fault site is bridged to 1, we can say that this fault-site is covered



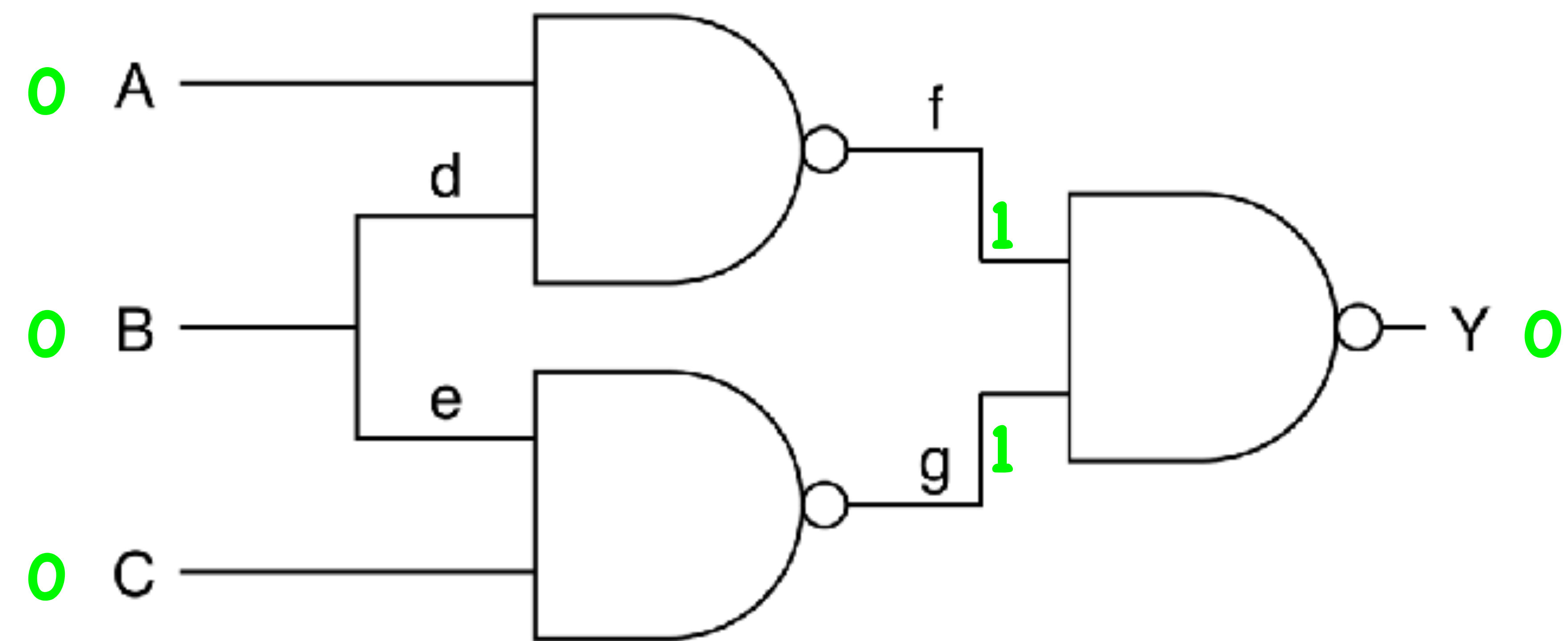
Finding Test Vectors

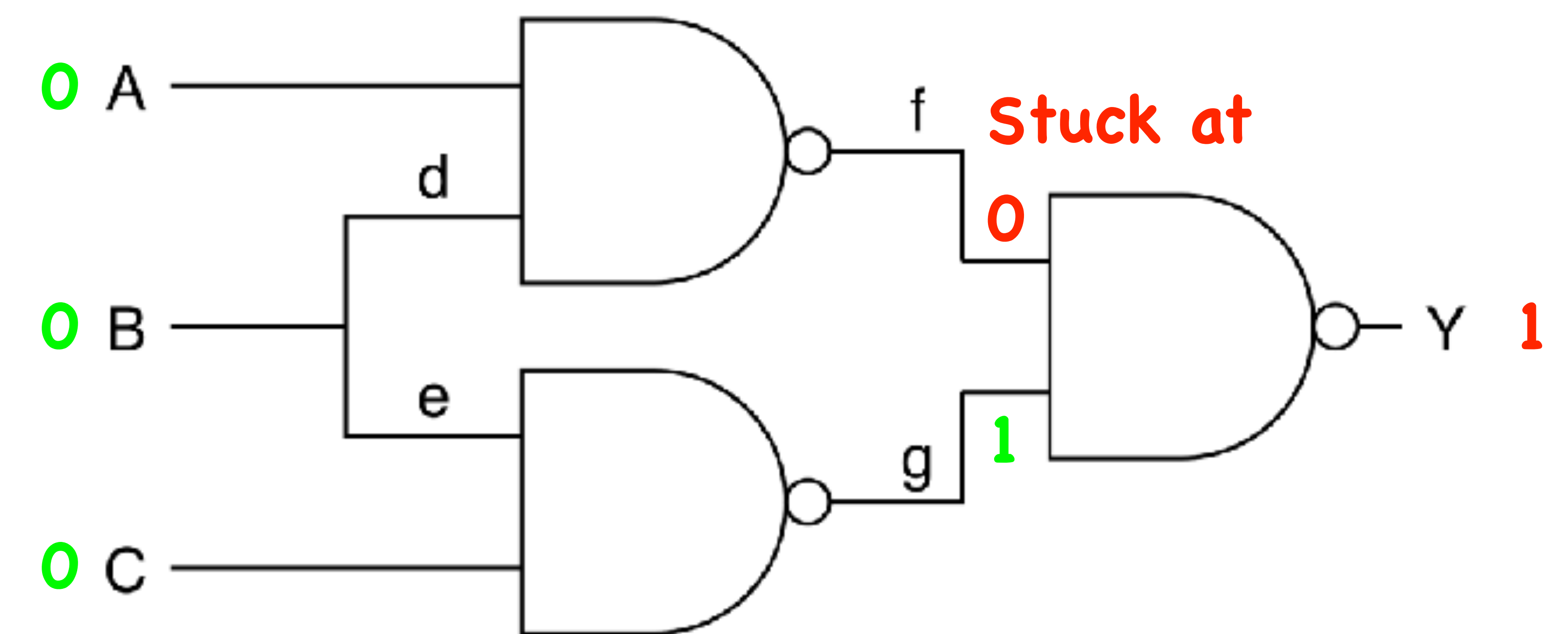




Test Vector 111_2 does not cover $f_{s\text{-}a\text{-}0}$.
(No difference in output.)

Finding Test Vectors





Test Vector 000_2 covers $f_{s\text{-}a\text{-}0}$.
(Difference in output.)

Automatic Test Pattern Generation

```
covered = set()
coverage = 0

while coverage < 95%:
  vector = generateTestVector()
  for fault in faultSites:
    gold = simulate(vector, None)
    true = simulate(vector, fault)
    if gold != true:
      covered.add(fault)
  coverage = len(covered) / len(faultSites)
```

Generating test vectors is NP-hard.

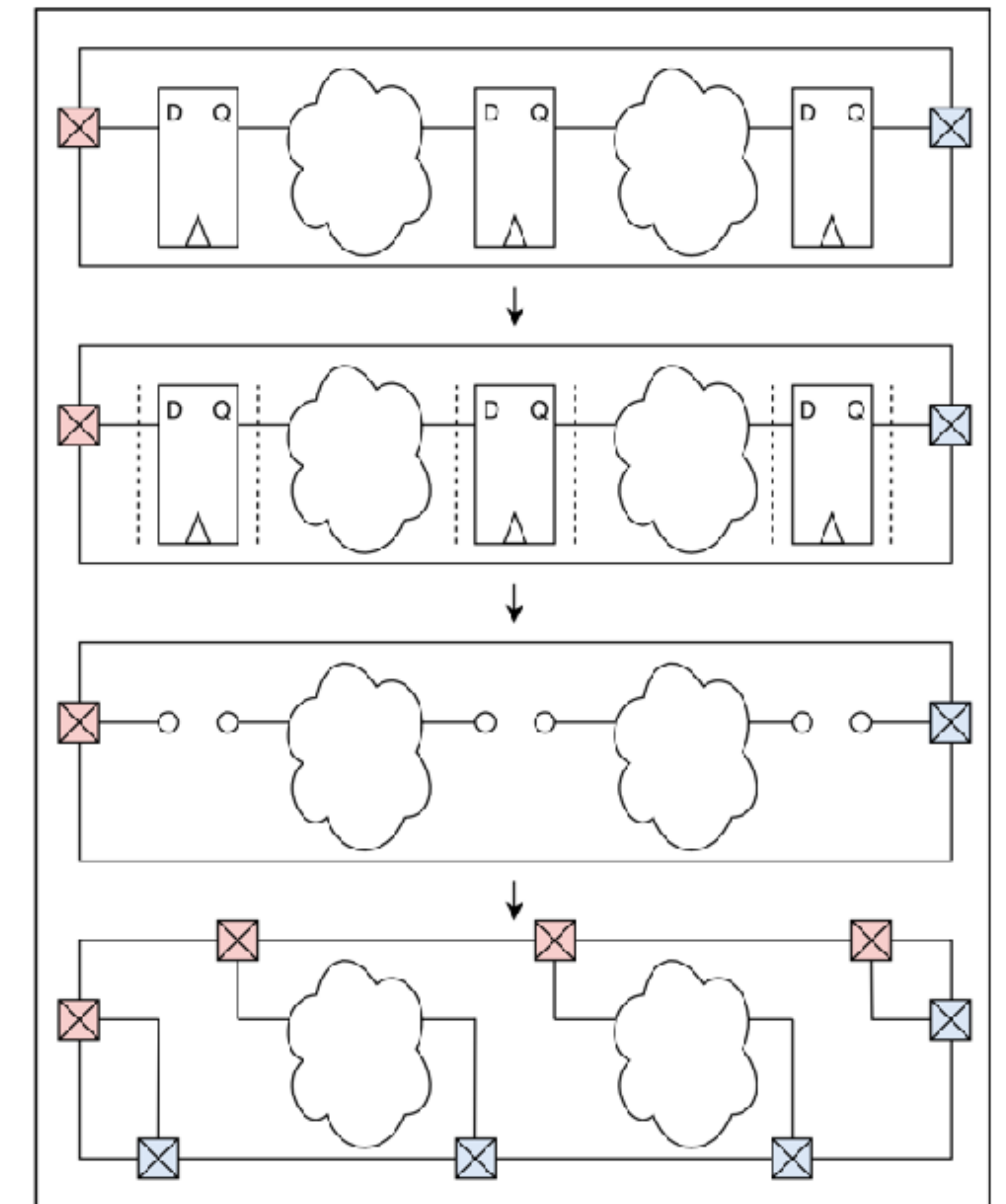
In practice, it can be done using one of many algorithms, such as:

- D Algorithm
- PODEM
- Pseudorandom Test Generation

What about Sequential Circuits?

- Approach demonstrated so far only works for combinational circuits
  - How do you generate patterns for sequential circuits?
  - **You don't.**

- You modify the circuit so that all register inputs are outputs to your circuit, and all register outputs are inputs to your circuit.
  - This generates patterns for all the combinational parts of the circuit
  - But now you need to load the data into the registers directly somehow

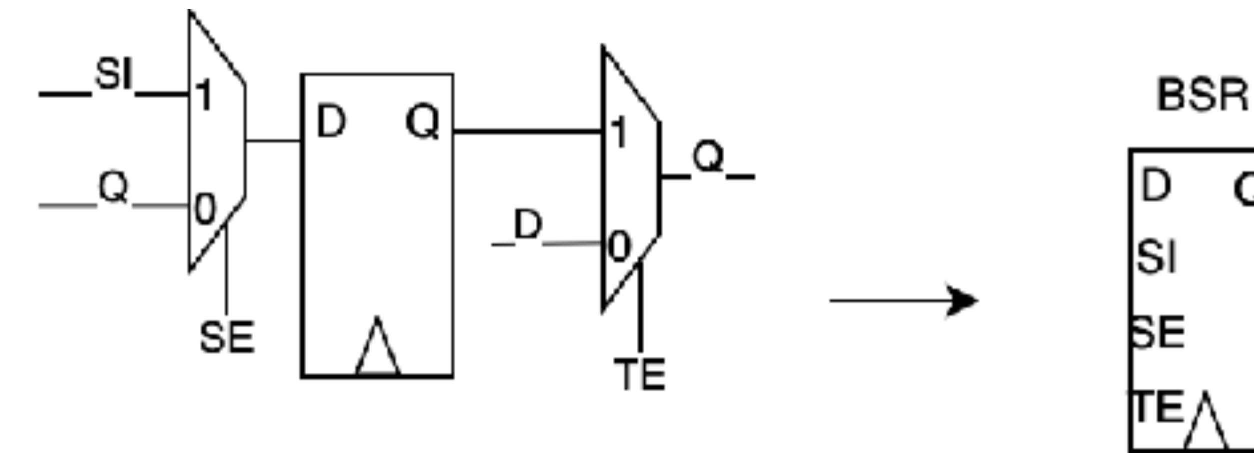


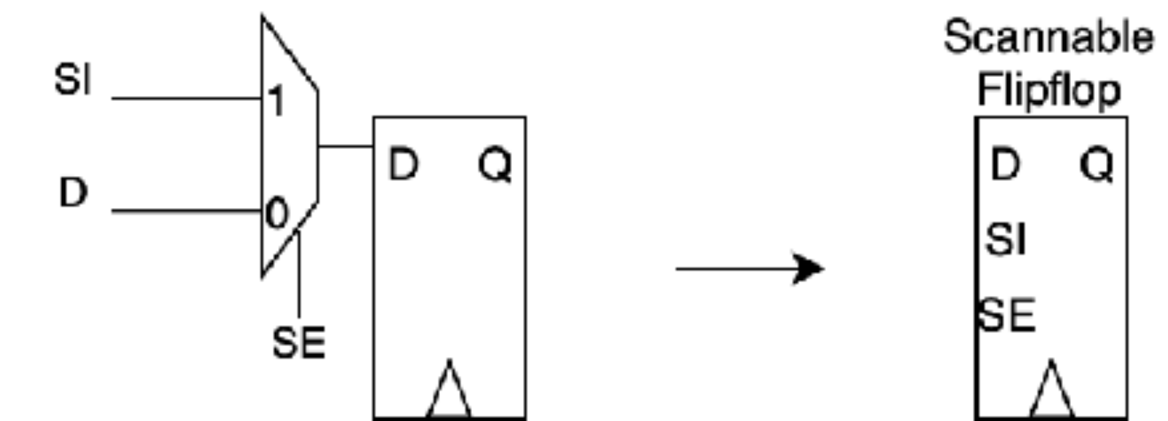
Scan-chain Stitching

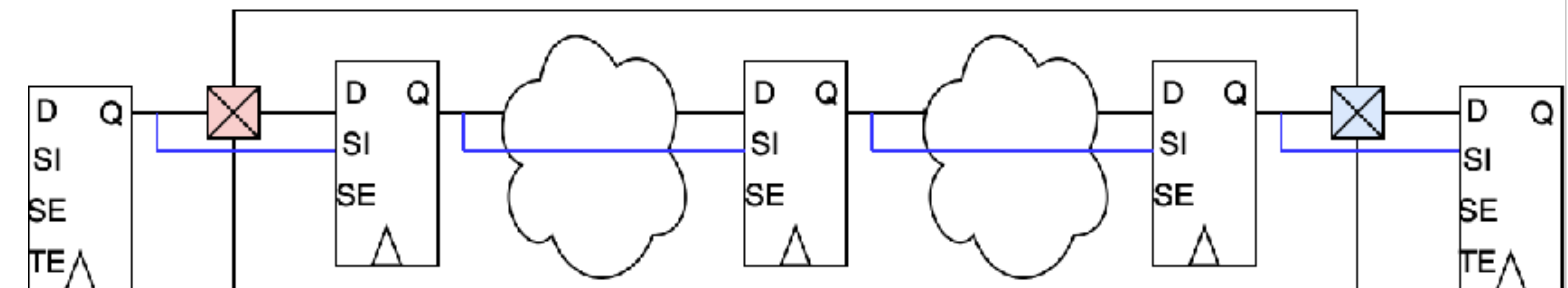
- Adding **boundary scan registers**
  - i.e. registers for all inputs and outputs, entirely bypassed when not in test mode

- Create a daisy-chain **out of every single register in the circuit**
  - How? By converting them into **scannable flipflops**
  - Serial interface to the daisy-chain
    - `tsi` → shift in
    - `tso` → shift out
    - `tck` → test clock
    - `shift` → enable shift mode
    - `test` → enable test mode

- Application of a test pattern becomes
  - Scan in test vector
  - Wait one clock cycle
  - Scan out result
  - Compare with expected



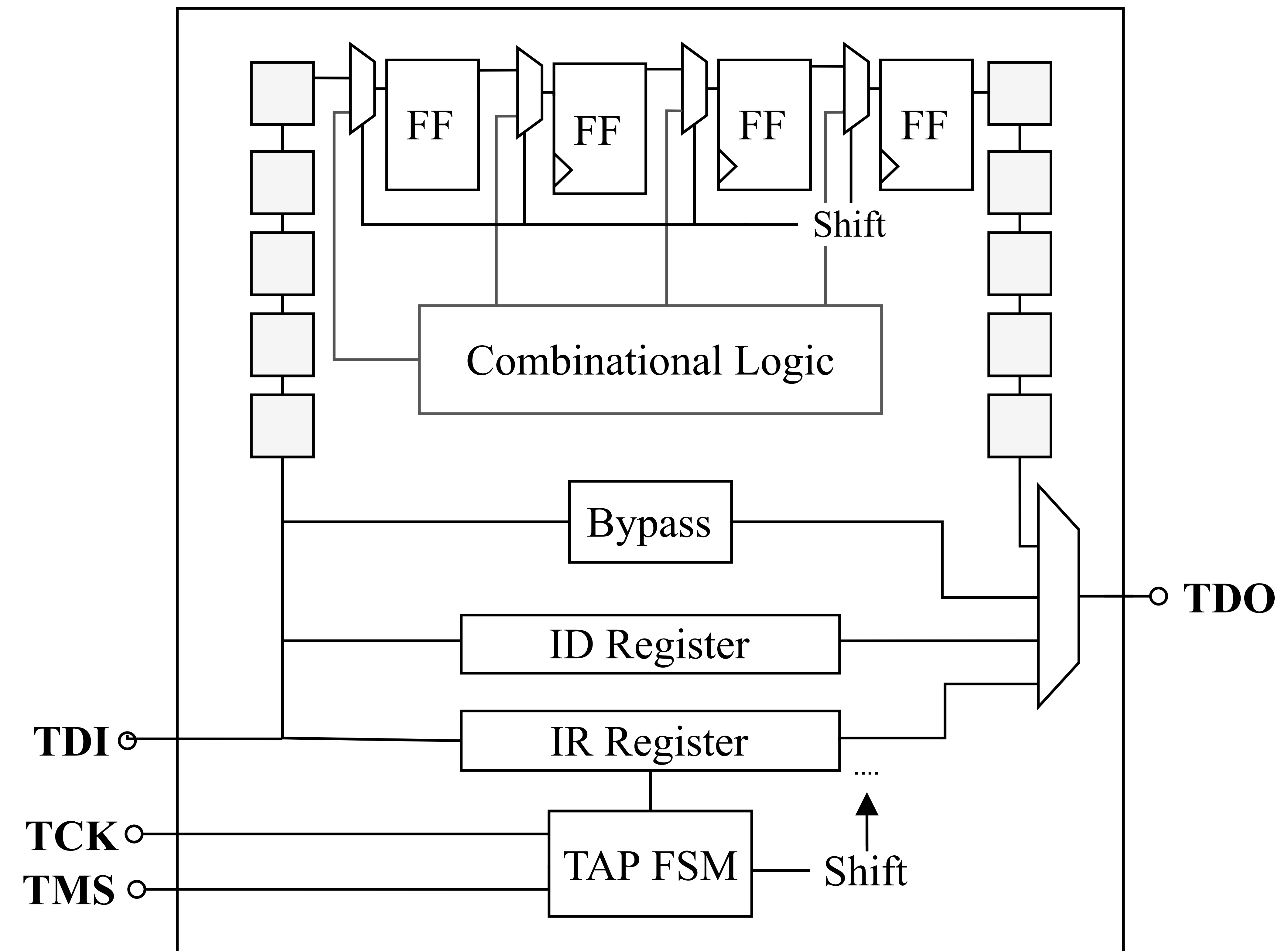


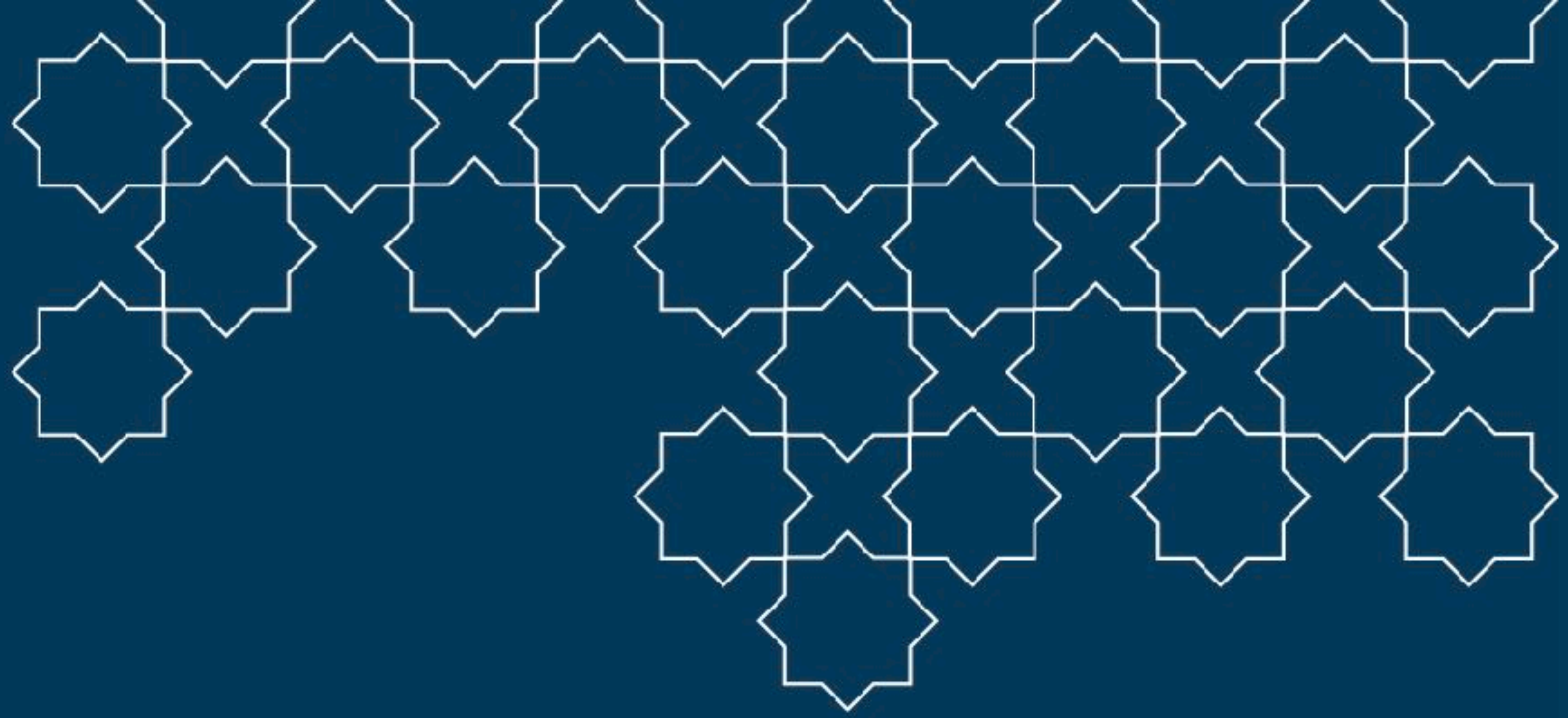


💡 With this arrangement, testing the registers themselves is as simple as scanning in any random pattern and scanning it out, and checking that it maintains its integrity.

TAP Controller Insertion

- The JTAG standard specifies the use of a test-access port (TAP) that can be used to communicate with the scan chain
- This necessitates a TAP controller, for which we used an open source one
  - https://www.opencores.org/projects/jtag/
- In practice, communication and running tests is done through the very same TAP controller
  - Fault verifies that the test:
    - Can be run with the TAP controller
    - Behaves as expected



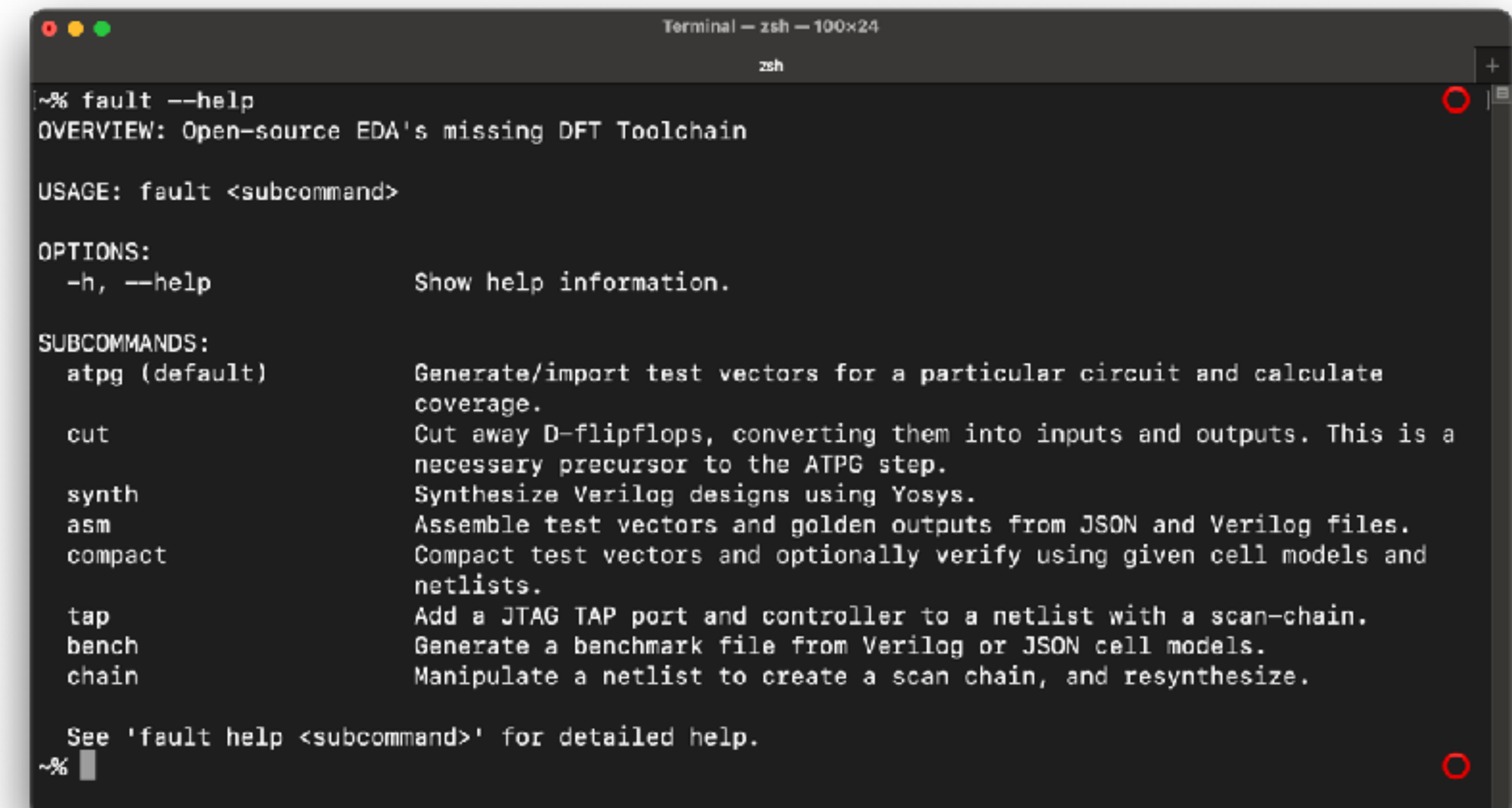


Implementation

Implementation

- Commandline-tool written Swift 5.4, for macOS or Linux
  - PythonKit by Google Inc.
    - Used to interface with the **Pyverilog** library by Shinya Takamaeda-Yamazaki and Contributors for Netlist manipulation
  - Yosys by Claire Wolf and Contributors
    - Used for re-synthesizing after Netlist manipulations
  - IcarusVerilog by Stephen Williams and Contributors
    - Used for ATPG simulation and verification

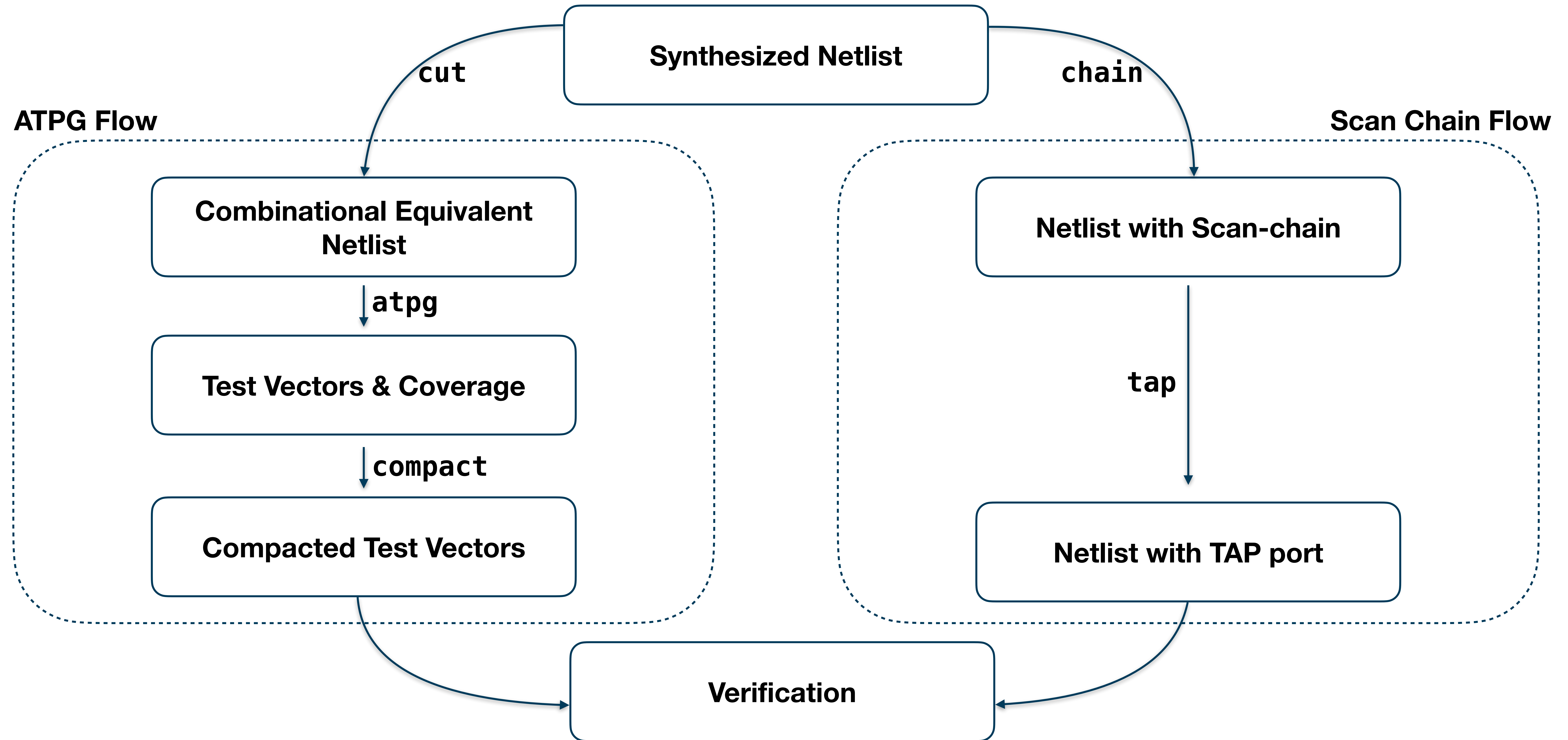
- Docker image provided to make setup easier

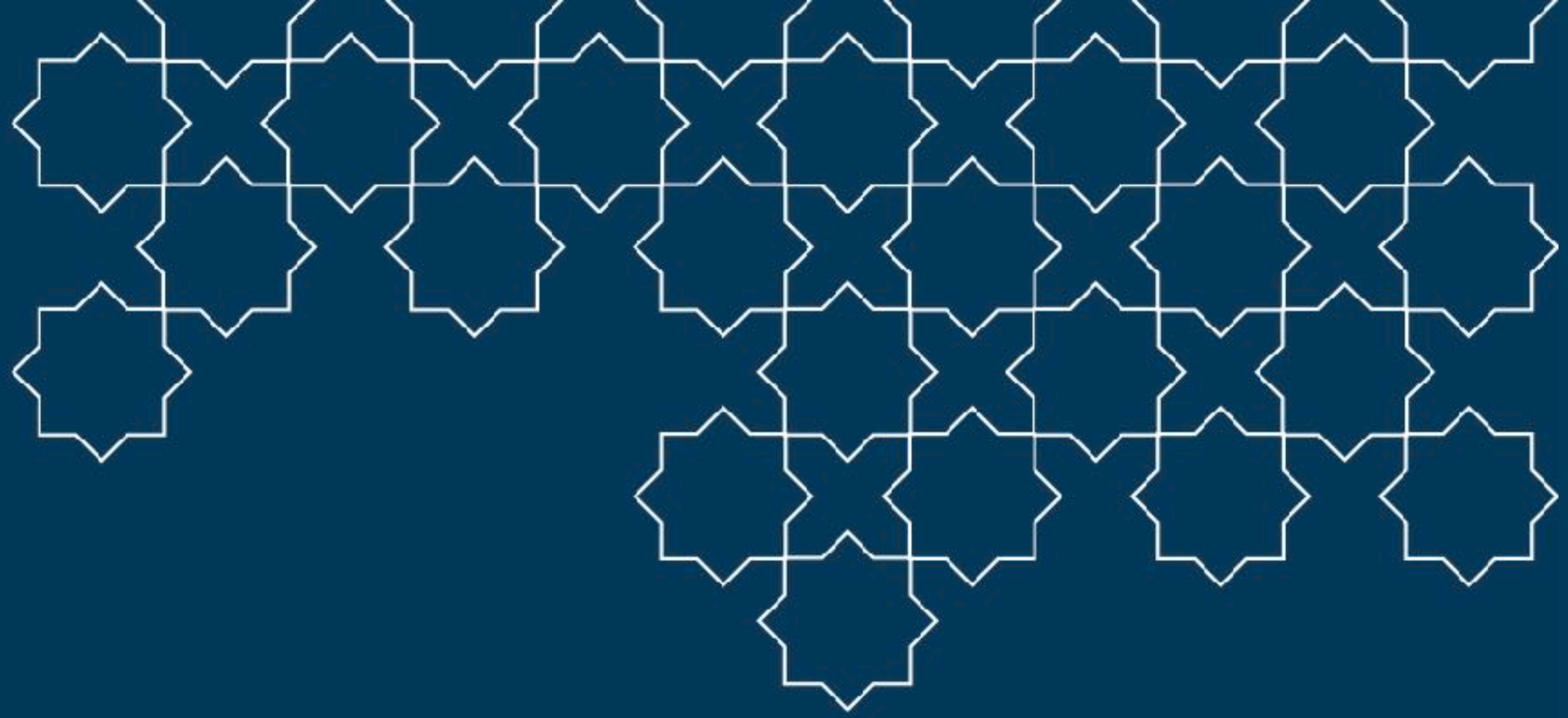


Subcommands

- **`synth`** - optional utility to synthesize the design -- in practice, use your own netlists
- **`cut`** - converts a sequential netlist to a combinational one compatible with ATPG
- **`atpg/main`** - uses a combinational netlist and a PRNG to generate test patterns
  - Can also calculate coverage test patterns generated by external tools
- **`compact`** - removes redundant test vectors
- **`chain`** - creates a scan chain out of the netlist, including adding boundary scan registers
  - verifies the scan chain using test patterns from atpg
  - can also add BSRs around hard macros instantiated in the design
- **`tap`** - adds a tap controller to a final design, verifying using test patterns from atpg afterwards

Fault Flow





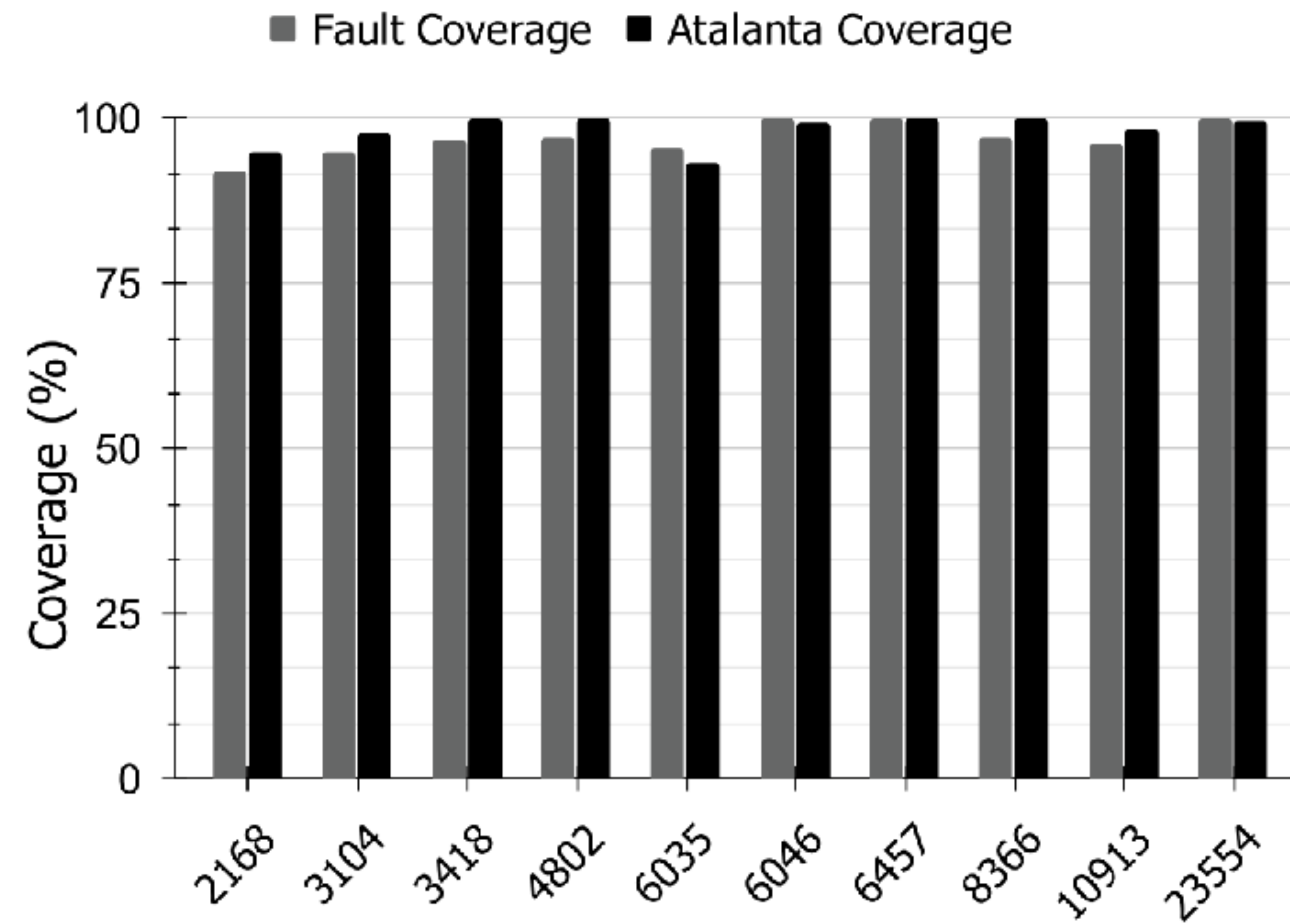
Testing, Results

Testing

- Evaluated ATPG's performance using a number of open-source designs frequently used as benchmarks
  - For comparison, we used **Atalanta** by Virginia Polytechnic Institute & State University
    - Source-available (not open-source) ATPG

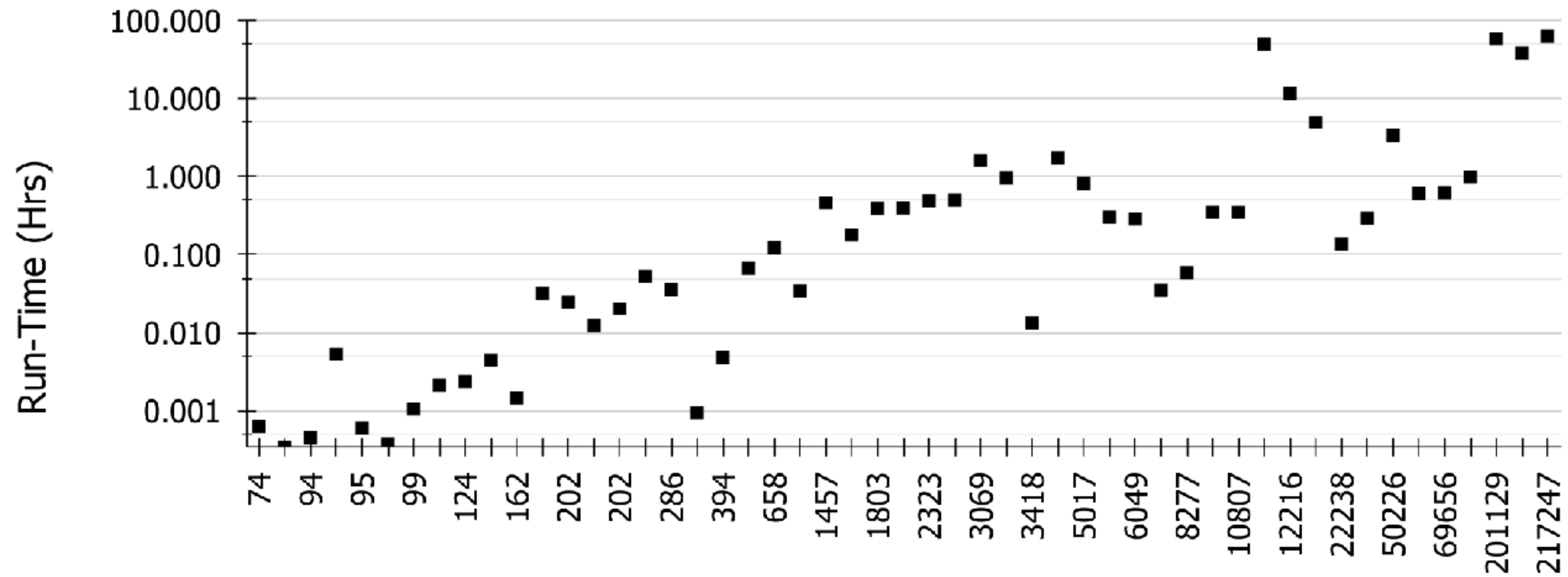
- Tests run on an Intel Xeon-based platform running Ubuntu 18.04 LTS with 32 GB of RAM across 10 threads

Results: Coverage

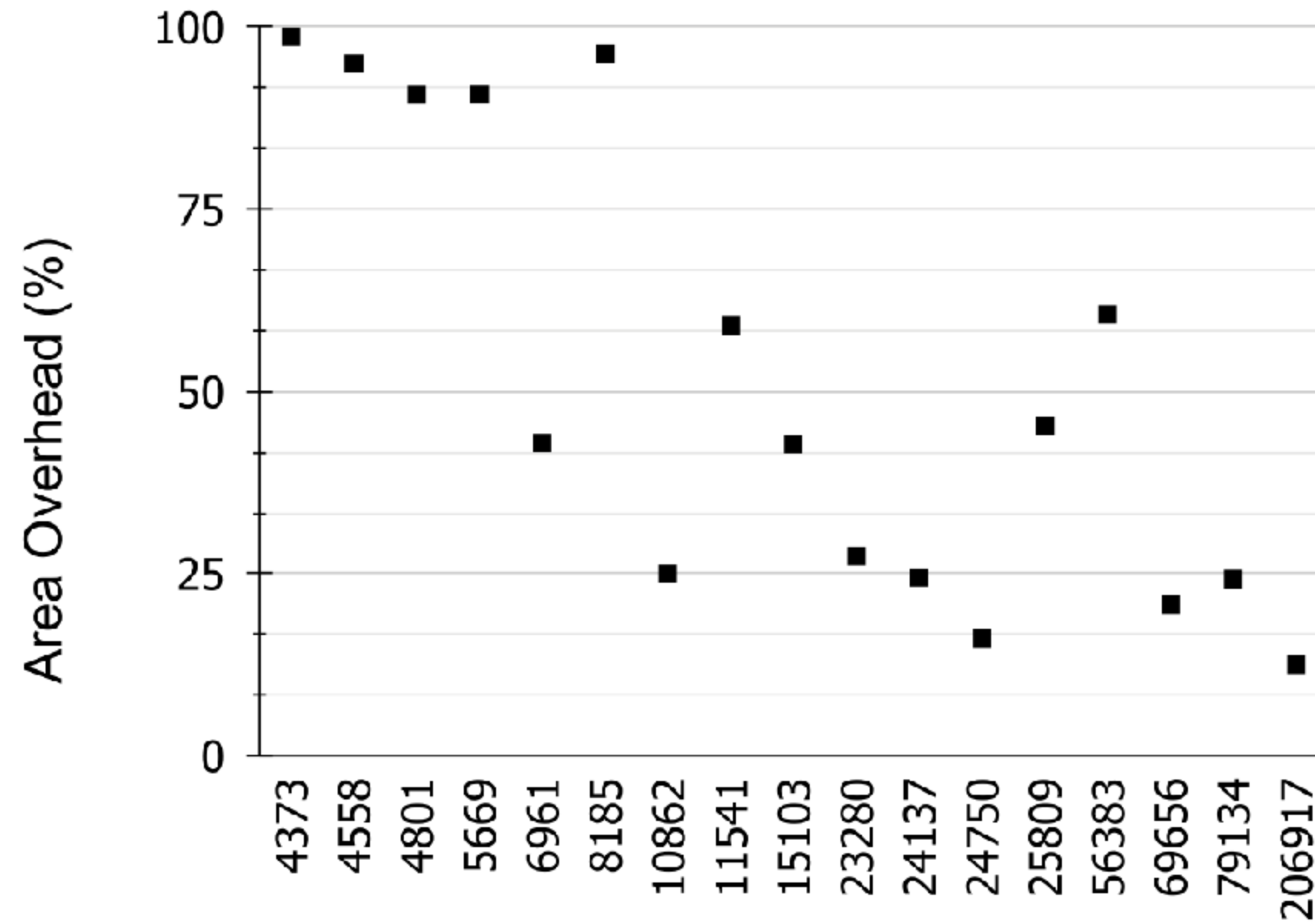


Fault average: 96.6%

Results: ATPG Runtime



Results: Area Overhead



Conclusion and Future Work

Conclusion

- We have presented a complete, open-source DFT solution that can
  - Generate test vectors for netlists
  - Stitch scan-chains into netlists
  - Generate and verify the addition of a test-access port into a chained netlist

- It is freely available on https://github.com/AUCOHL/Fault
  - We still provide Docker images for x86-64

- We've taped out a number of Fault-based designs
  - Caravel with Fault DFT SPM https://platform.efabless.com/projects/26
  - A complex proprietary design at Efabless

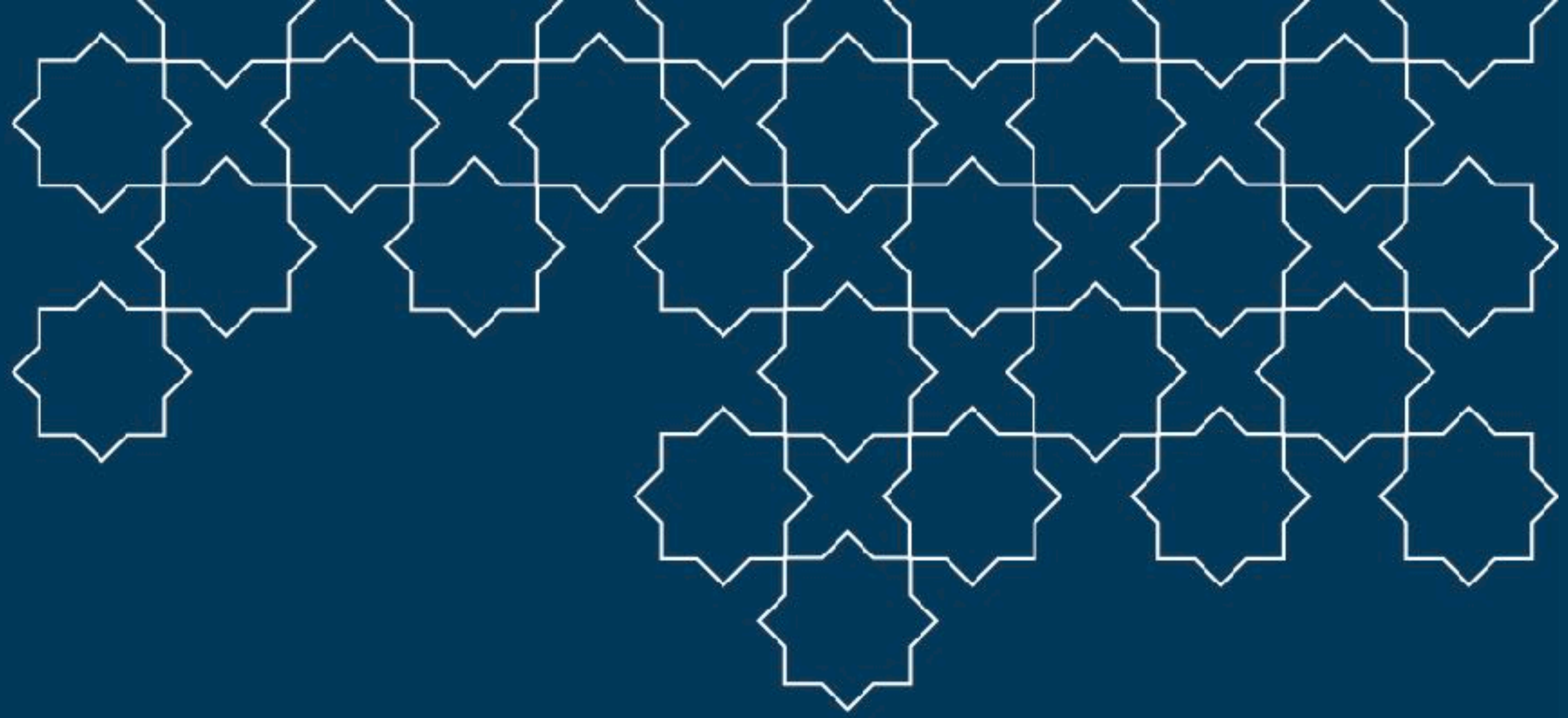
Future Work

- Implement a proper ATPG algorithm
  - Slight boost to coverage, massive reduction in run-time
  - GSoC internship secured!
- Support test-vector compression
  - For a design with N flipflops, you have to wait 2N + 1 clock cycles to run a single test
  - Again, the less physical wall clock time the cheaper
- Support multiple scan-chains
  - Fault currently supports inserting a TAP controller for exactly one scan chain

Future Work

- Placement-aware chaining
  - Currently we operate on the Netlist
  - Placement can be inefficient in comparison with direct layout manipulation

- More robust test set
  - We currently have two simple tests, a combinational one and a macro-based one



✨ Thank you!