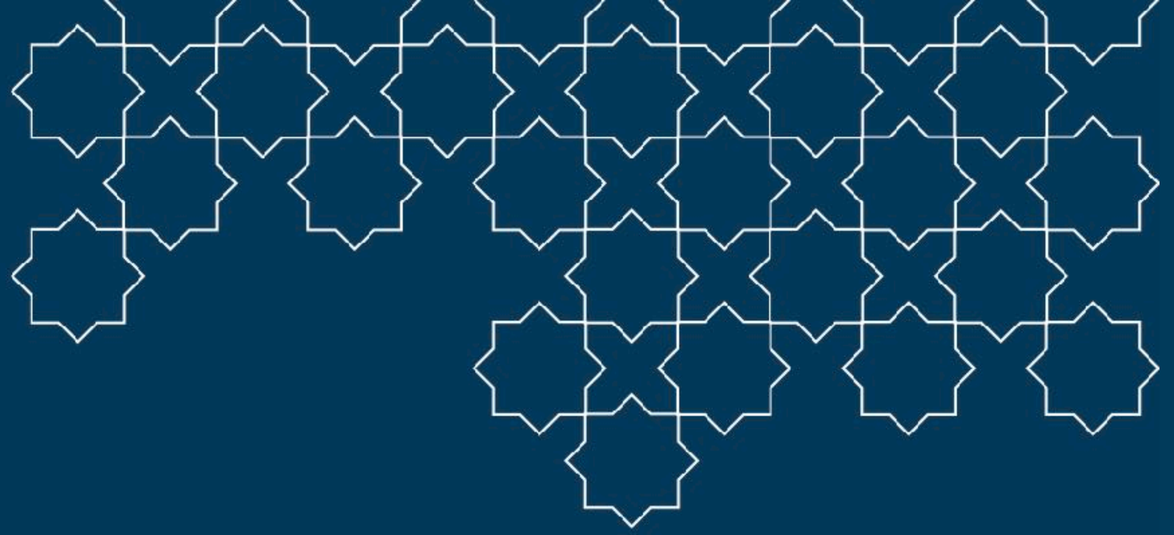



# Fault

## Open-Source EDA's Missing DFT Toolchain

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

# About us

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

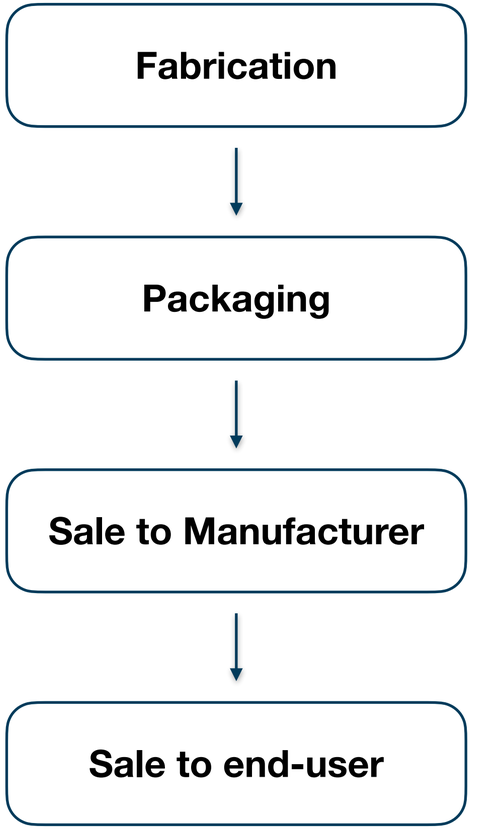
- Mohamed Gaber
  - Graduate Student, The American University in Cairo
  - Senior EDA Engineer, Efabless Corporation

- Mohamed Shalan
  - 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
    - Timing variations
    - Shorts or opens where they aren’t expected

- 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 10
    - If you don’t catch the error early, 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 and perform all necessary design changes to enable testing are broadly known as **design-for-testing**, or DFT tools

# Status Quo Ante

- 2019

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

- Projects like Yosys, Icestorm, etc breaking barriers to FPGA design
  - And soon, ASIC design, with pioneers like the OpenROAD project

- But crucially, no DFT utilities!

# Our Contribution

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

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

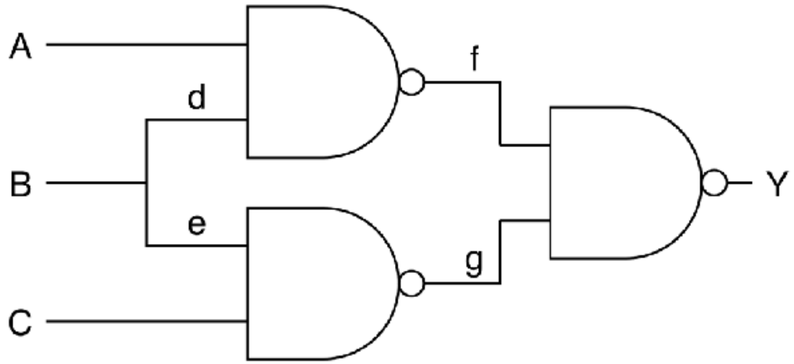
- Simply called "Fault."

# Architecture

# The Fault Model

- We need to model the possible defects that can occur in manufacturing somehow

- Simple but effective model: the **stuck-at** fault model
  - Any input or output to internal cells or macros are possible **fault sites**
  - 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
  - Doesn’t model some defects, but still very

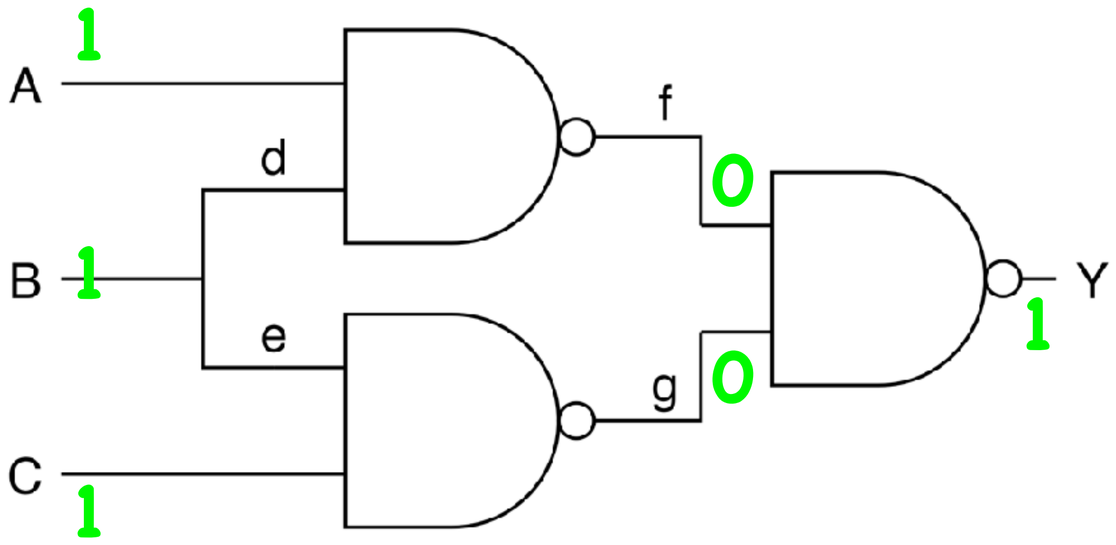


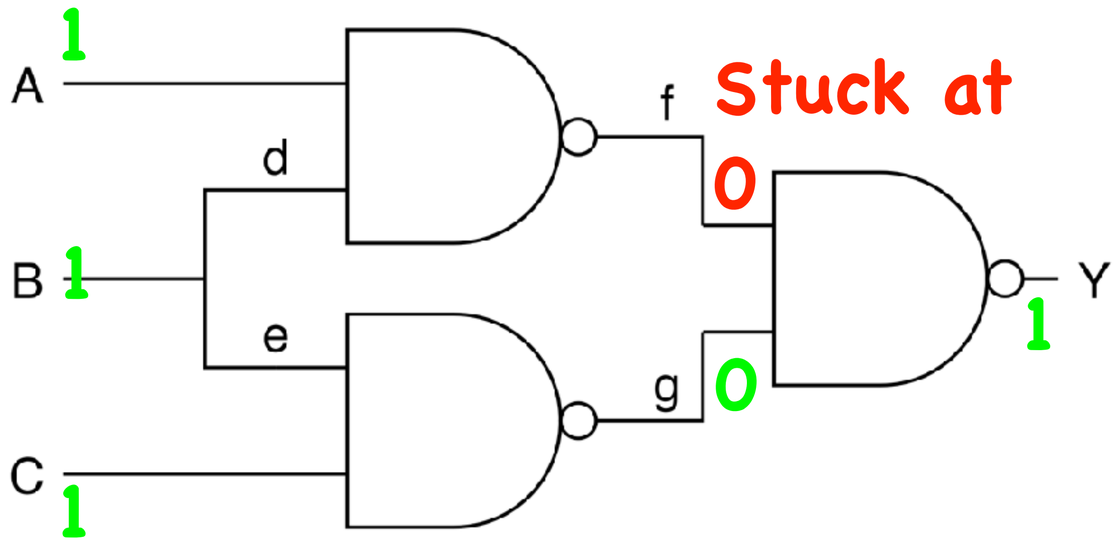
# The Fault Model

- Nodes where fault sites can occur should be…
  - **Controllable**
    - A value at a internal node can be set to a specific value using inputs
  - **Observable**
    - A fault at an internal node should propagate to the outputs

- How do we observe a fault? (aka the terminology section)
  - Using a **test vector**: which is a set of inputs to the circuit that somehow propagates a fault to an output
  - A test vector that can cause a certain fault to be observable is said to **cover** a fault.
  - The percentage of faults covered by a test vector or a set of test vectors is termed

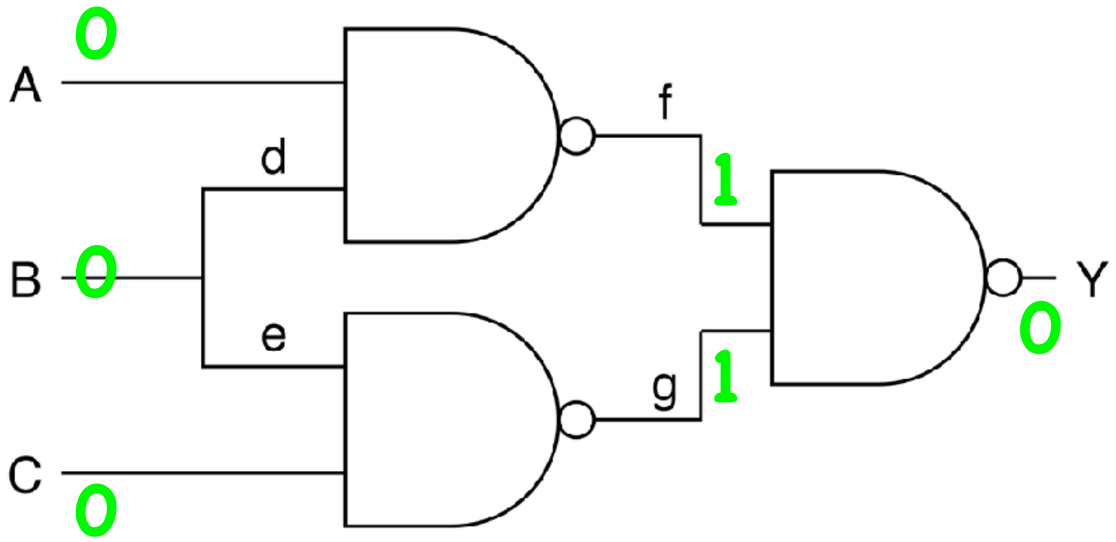
# Controllability and Observability

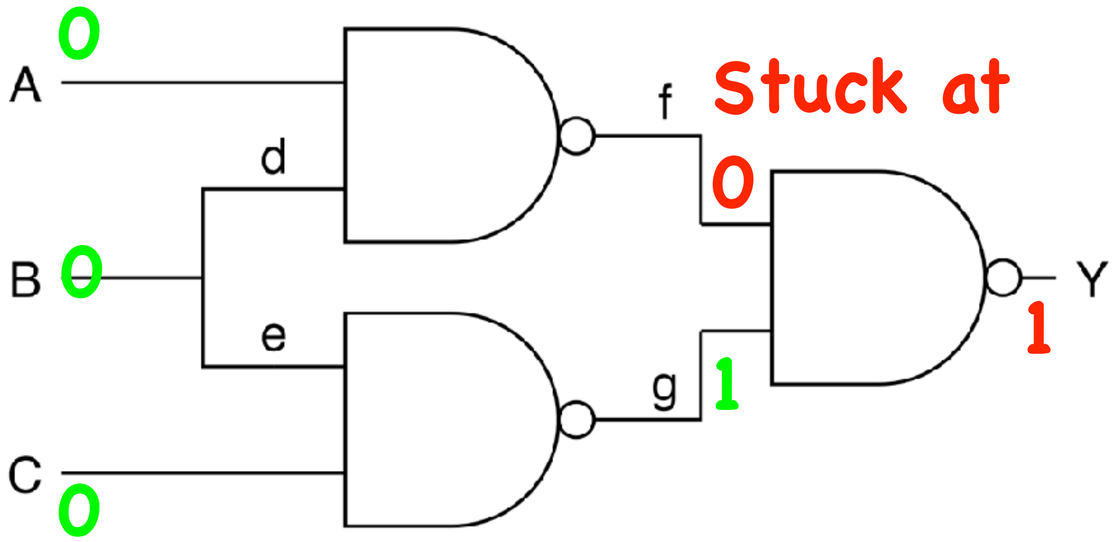



Test Vector $111_2$ does not cover $f_{s\text{-}a\text{-}0}$.
(No difference in output.)

# Controllability and Observability




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

# 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

# Automatic Test Pattern Generation

- Good ATPG tools try to optimize two metrics
  - Coverage of the most amount of possible defects
  - in the least total test time

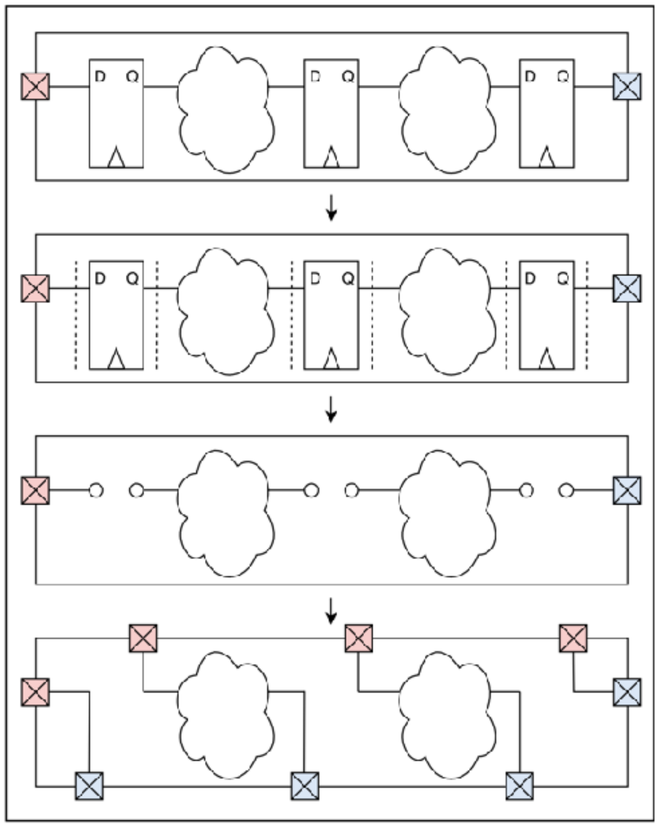
- This translates to:
  - Coverage of the **largest** amounts of fault sites
  - In the **least** amount of test vectors

- To keep test vector count down, the simplest thing you can do is check for redundant vectors and discard them, which we call compaction

# 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

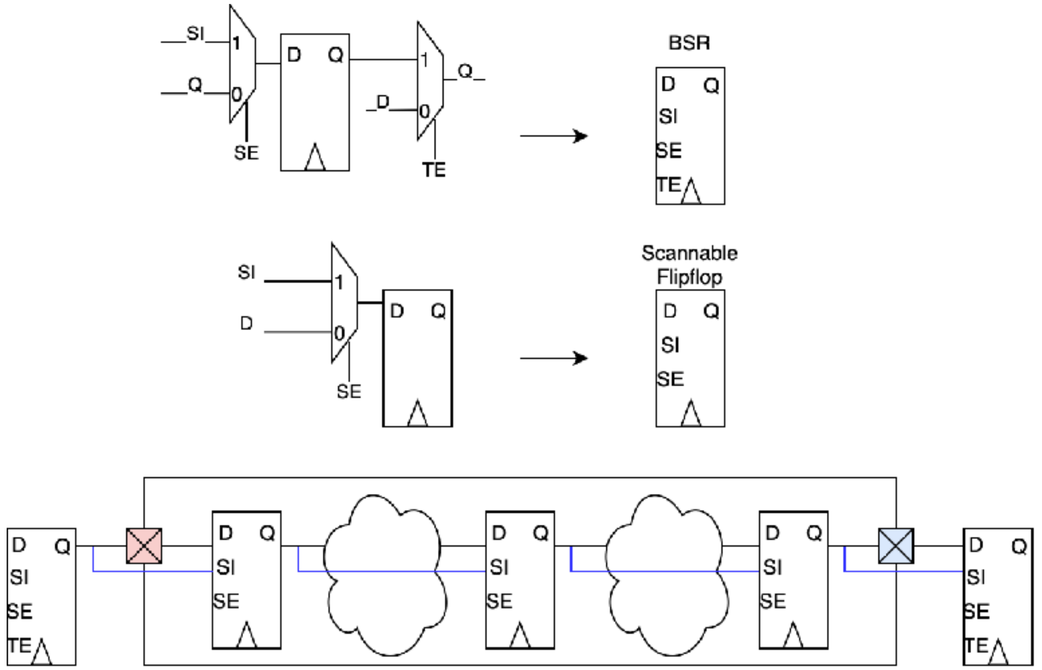


# Scan Insertion

- AKA “Scan chaining”, “Scan chain stitching”, …

- Create a daisy-chain **out of every single register in the circuit**
  - How? By converting them into **scannable flip-flops**
    - Some SCLs provide dedicated scannable flip-flops, some don’t
    - You can use multiplexers at an area penalty

- Serial interface to the daisy-chain
  - `tsi` → shift in
  - `tso` → shift out
  - `tck` → test clock
  - `shift` → enable shift mode
  - `test` → enable test mode



# Applying tests to a scan chain

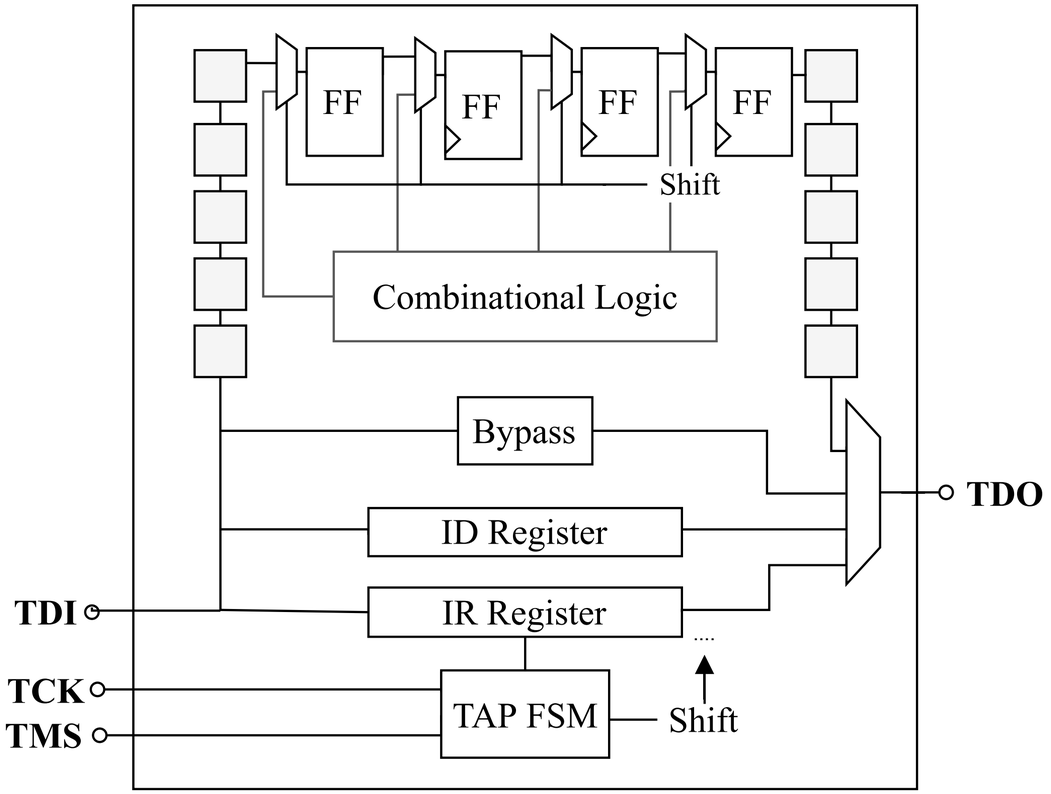
- Application of a test pattern becomes
  - Scan in test vector
  - Wait one clock cycle
  - Scan out result
  - Compare with expected, validating the combinational components

- Takes $2N + 1$ clock cycles where $N$ is the length of the scan chain

- What about sequential elements?
  - A simple test at the beginning: scan in any pattern and then scan it out immediately
  - If you get the same result, you've proven every flip-flop works

# TAP Controller Insertion

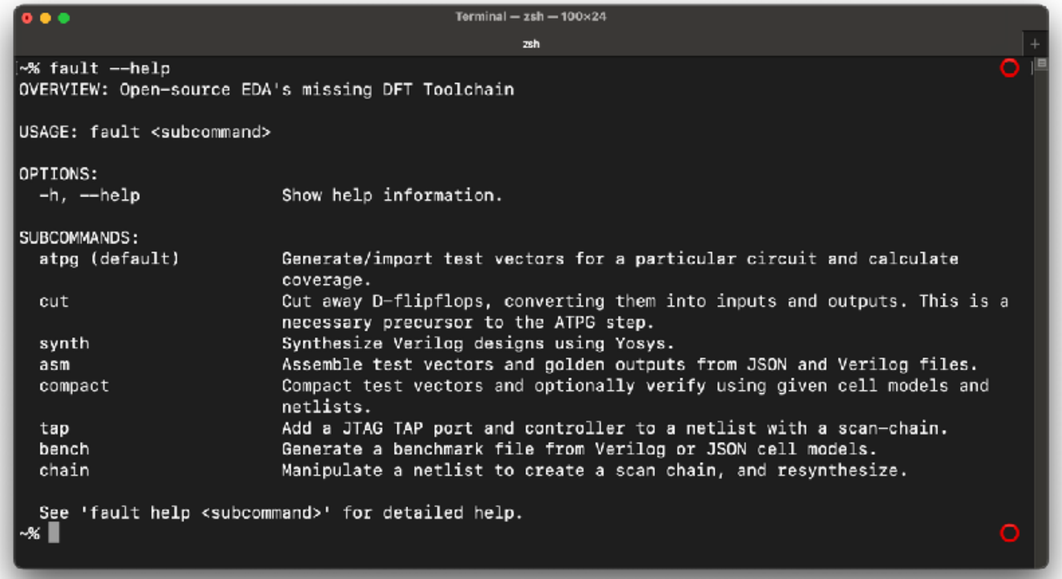
- The JTAG standard specifies the use of a test-access port (TAP) that can be used to communicate with the scan chain
- This necessitates a TAP controller, for which we used an open source one
  - https://www.opencores.org/projects/jtag/
- In practice, communication and running tests is done through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



# 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

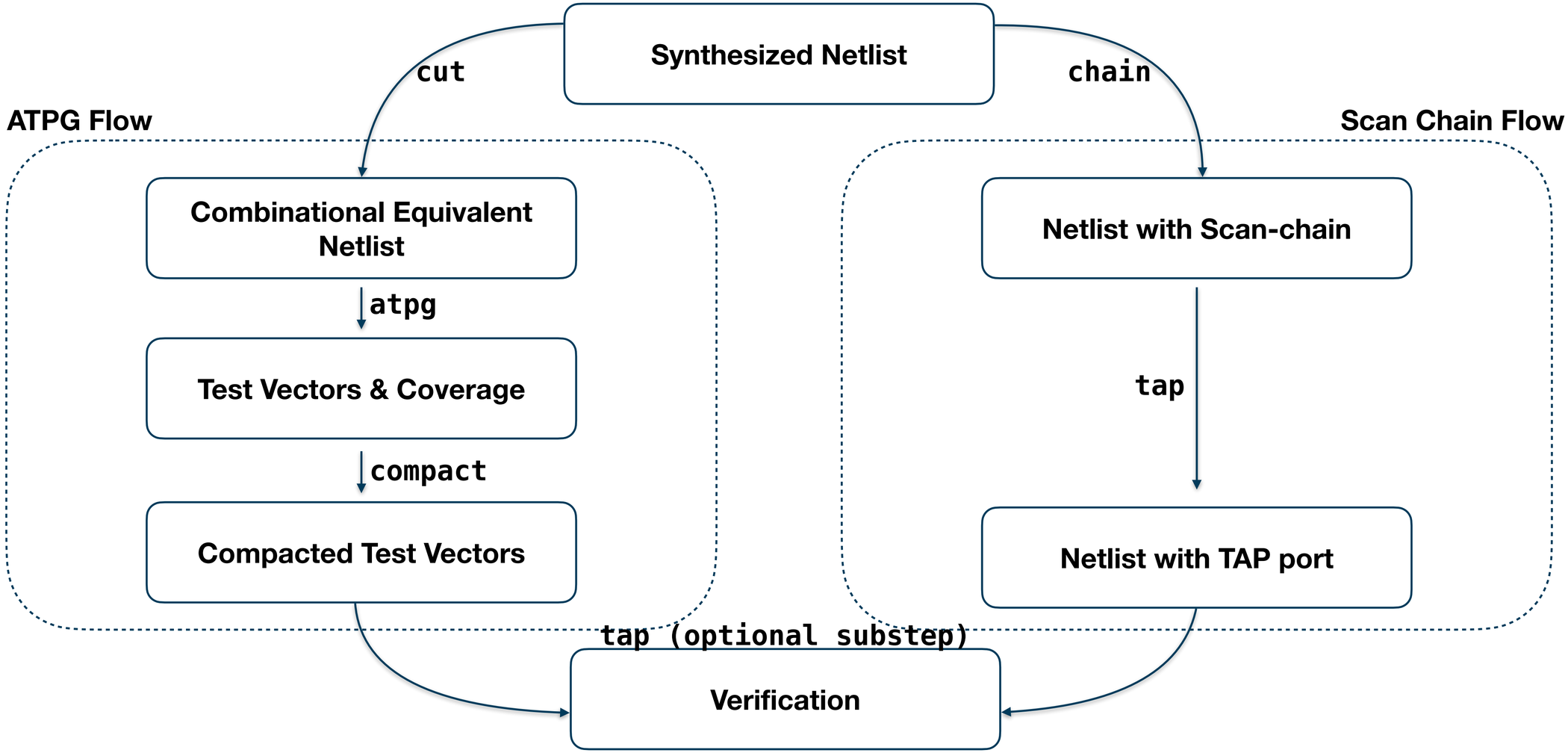
- **`atpg/main`** - uses a combinational netlist and a PRNG to generate test patterns
  - Can also calculate coverage test patterns generated by external tools

- **`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


Synthesized Netlist
cut
chain
ATPG Flow
Scan Chain Flow
Combinational Equivalent Netlist
atpg
Test Vectors & Coverage
compact
Compacted Test Vectors
Netlist with Scan-chain
tap
Netlist with TAP port
tap (optional substep)
Verification

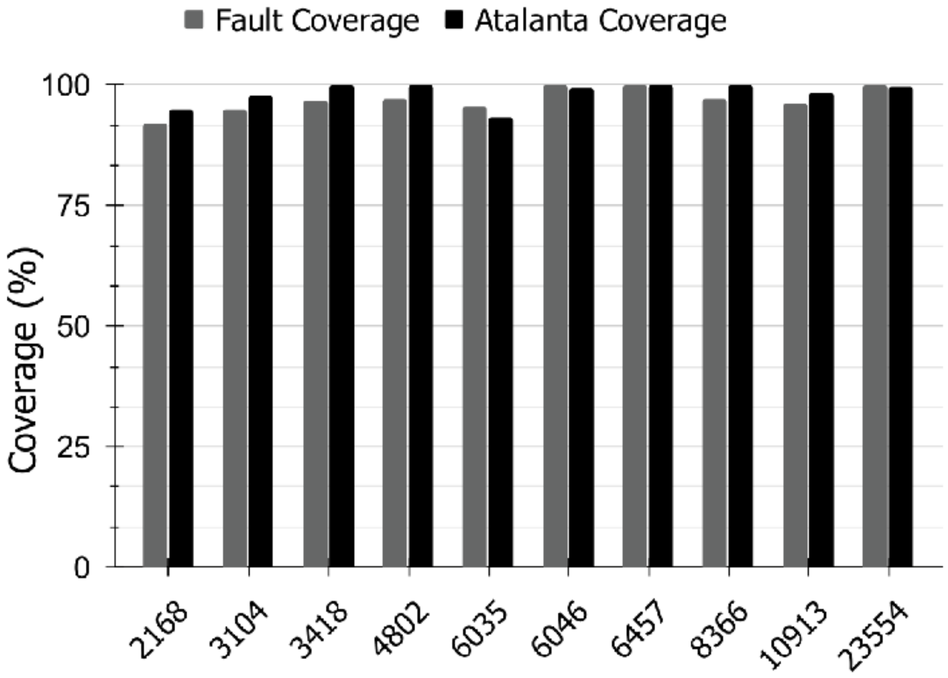
# Testing & Results

# 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

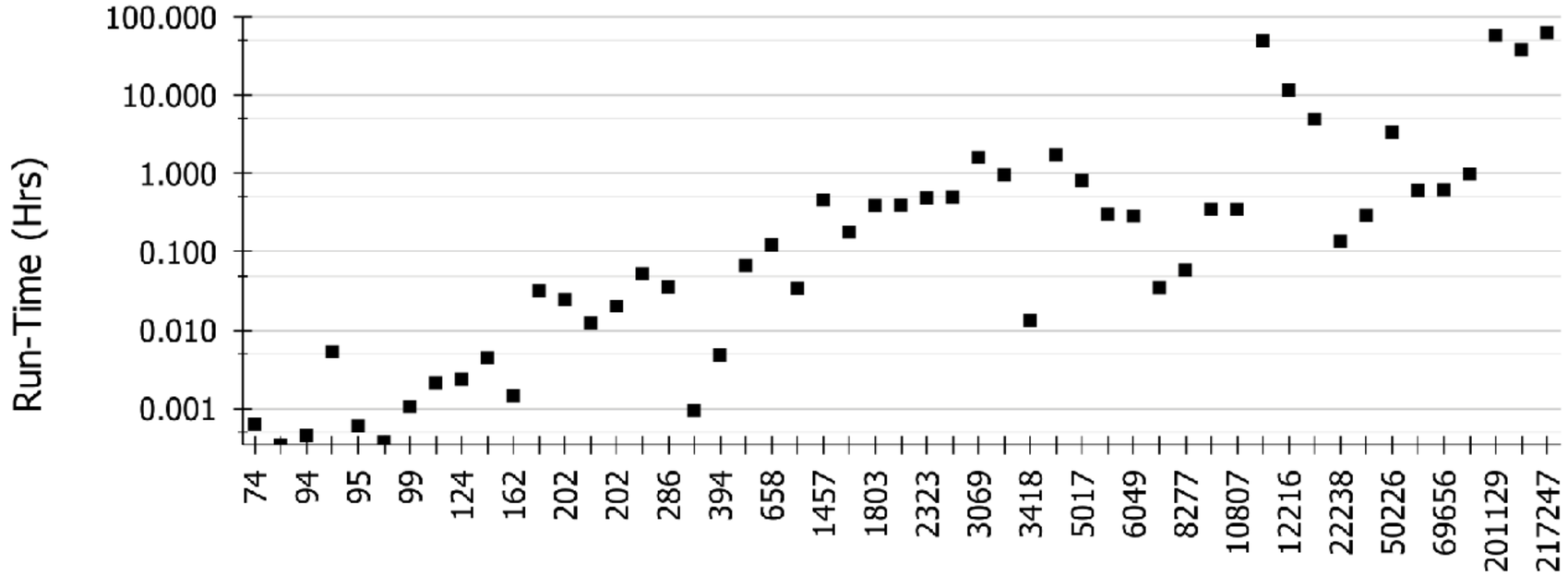
- Tests run on an Intel Xeon-based platform running Ubuntu 18.04 LTS with 32 GB of RAM across 10 threads

# Results: Coverage

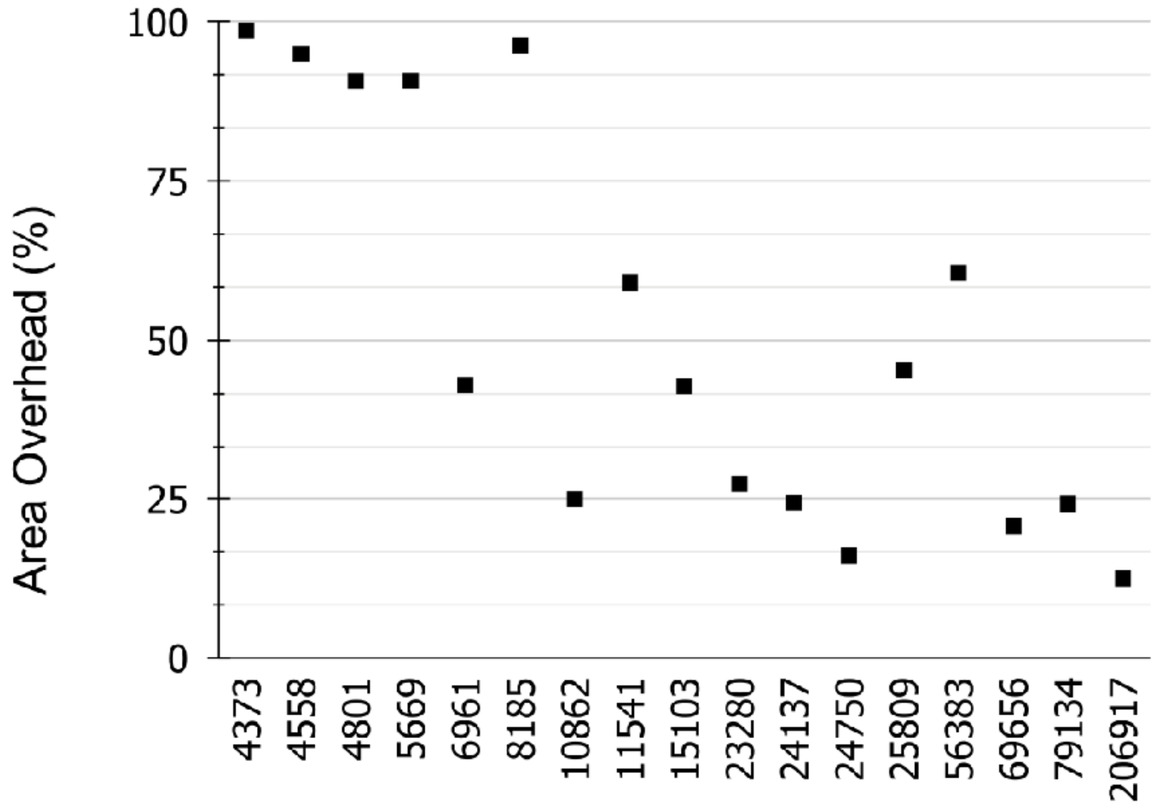



**Fault average: 96.6%**

# 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

# WIP Features

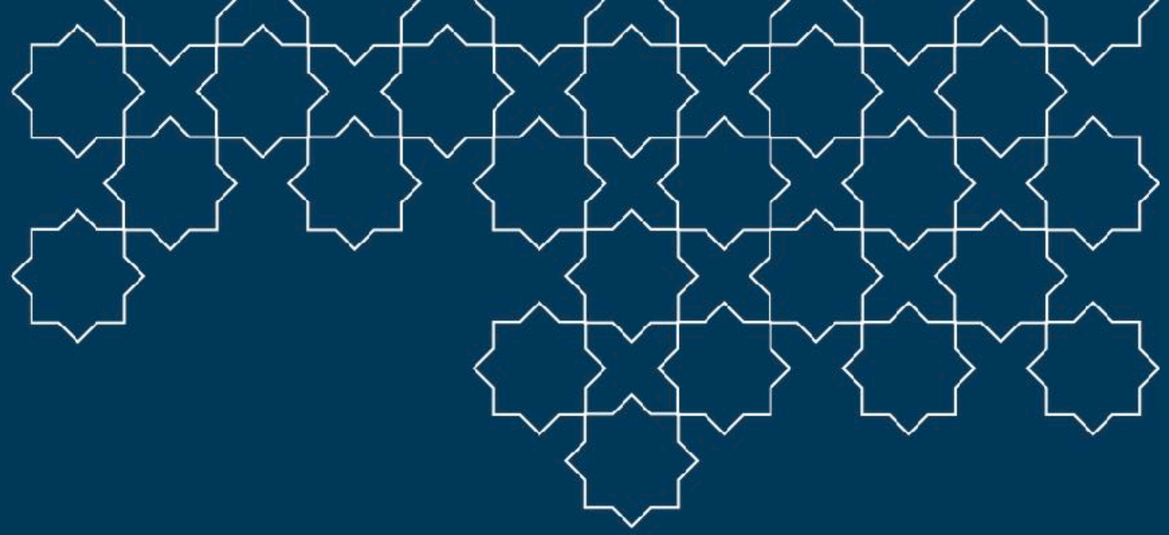
- Improve ATPG (WIP)
  - Fault relies on Pseudorandom test generation; long run-time.
  - Objectives: A slight boost to coverage and a massive reduction in run-time
    - Maybe use the upcoming presentation...?
  - GSoC internship secured!

- Support multiple scan-chains (WIP)
  - Fault currently supports inserting a TAP controller for exactly one scan chain
  - Doesn't scale

# Future Work

- Support test-vector compression
  - TAP controllers can have hardware on-chip that accepts lossy-compressed test vectors
  - Saves time when the test-vector is serially input (that channel is slow)

- Physical-aware scan insertion
  - Fault modifies the netlist - manipulating the layout directly can lead to time and area savings
  - Timing and area advantages if we instead manipulate the physical layout to insert the requisite cells



# ✨ Thank you!