TOWARDS FUNCTIONAL COVERAGE-DRIVEN FUZZING FOR CHISEL DESIGNS

Andrew Dobis, Tjark Petersen
& Martin Schoeberl.

- Technical University of Denmark (DTU) -
Department of Applied Mathematics & Computer Science

DTU

OUTLINE

- Motivation: Why use Functional Coverage as a metric?
- Background: Chisel & Fuzzing.
- Current solutions: Fuzzing for Chisel designs using software fuzzers and using custom hardware fuzzing.
- Our solution: Use hardware-oriented coverage metric to drive fuzzing of a Chisel Design.
- Initial Experiments: Using the fuzzer on the Leros accumulator ALU.
- **Note:** This solution is a work in progress and is currently still in development.

MOTIVATION

MOTIVATION: WHY USE FUNCTIONAL COVERAGE?

- Fuzzing is often associated to software testing.

- Current work has been done on Fuzzing for Digital Circuits, but none using **Functional Coverage** as a driving metric.

- Goal:
  - **Explore** the effects of using a metric that inherently contains data about the Device Under Test (DUT).
  - **Evaluate** its impact on the fuzzing efficiency.

BACKGROUND

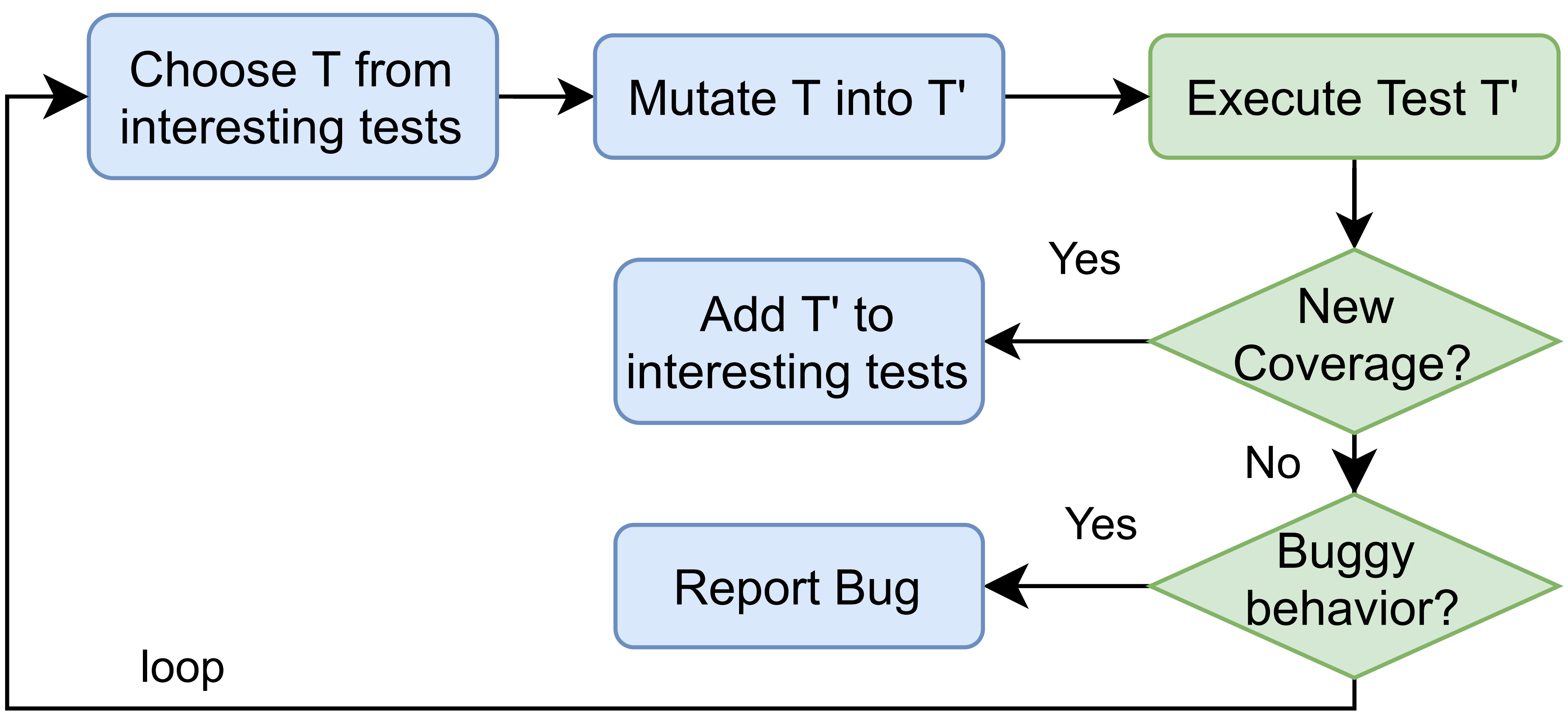
BACKGROUND: CHISEL AND VERIFICATION

- Chisel: Hardware Construction Language (HCL) embedded in Scala. Allows for high-level description of digital circuits using Object-Oriented and Functional programming. Can generate Verilog as a final output.

- Functional Coverage: Hardware-centric coverage metric based around the use of a *Verification Plan(VP)*. Functional Coverage gives a quantitative measure of the testing progress, telling us "*which features of the DUT have been tested?*".

BACKGROUND: COVERAGE DRIVEN MUTATION-BASED FUZZING

- Automatic randomized input generation.
- Inputs are based on a set of valid initial inputs(*seeds*).
- Seeds are then mutated depending on the coverage result that they generate. If the new inputs are "interesting", they are added to the seeds.



CURRENT SOLUTIONS

OVERVIEW OF THE CURRENT SOLUTIONS

- American Fuzzy Lop (AFL), 2013:
  - Software coverage driven mutation-based fuzzer.
  - uses edge coverage, a form of branch coverage.
  - Their mutation techniques are used in our solution.
- RFuzz, 2020:
  - Coverage-driven mutation based buzzer for RTL designs.
  - Leverages FPGAs to accelerate their solution.
  - Employs intelligent techniques for fast memory initialization.
  - Metric is also edge coverage.

CURRENT SOLUTIONS: CONTINUATION

- Fuzzing Hardware like Software, 2021:
    - Translates DUT hardware into a software model.
    - Uses existing software fuzzers for the fuzzing.

- All of these solutions rely on the same metric to guide fuzzing.
    - Why not use a more complex metric?
    - What will the impact on the performance look like?

OUR SOLUTION:
FUNCTIONAL COVERAGE TO DRIVE FUZZING

OUR SOLUTION: OVERVIEW

- Functional Coverage metric being used is from ChiselVerify.
- Fuzzer functions in 5 phases:
    - **Interpret** user-defined input files as bit-streams and load them into the queue.
    - **Select** next file from queue.
    - **Mutate** file, first with deterministic then non-deterministic mutation passes.
    - **Run** test and **retrieve** coverage results.
    - **Compare** results to previous ones, **determine** if test was interesting and **add** it to the corpus. Repeat.

OUR SOLUTION: DEFINING A TEST

- Main difference between Hw fuzzing and Sw fuzzing:
    - Defining tests with timing.

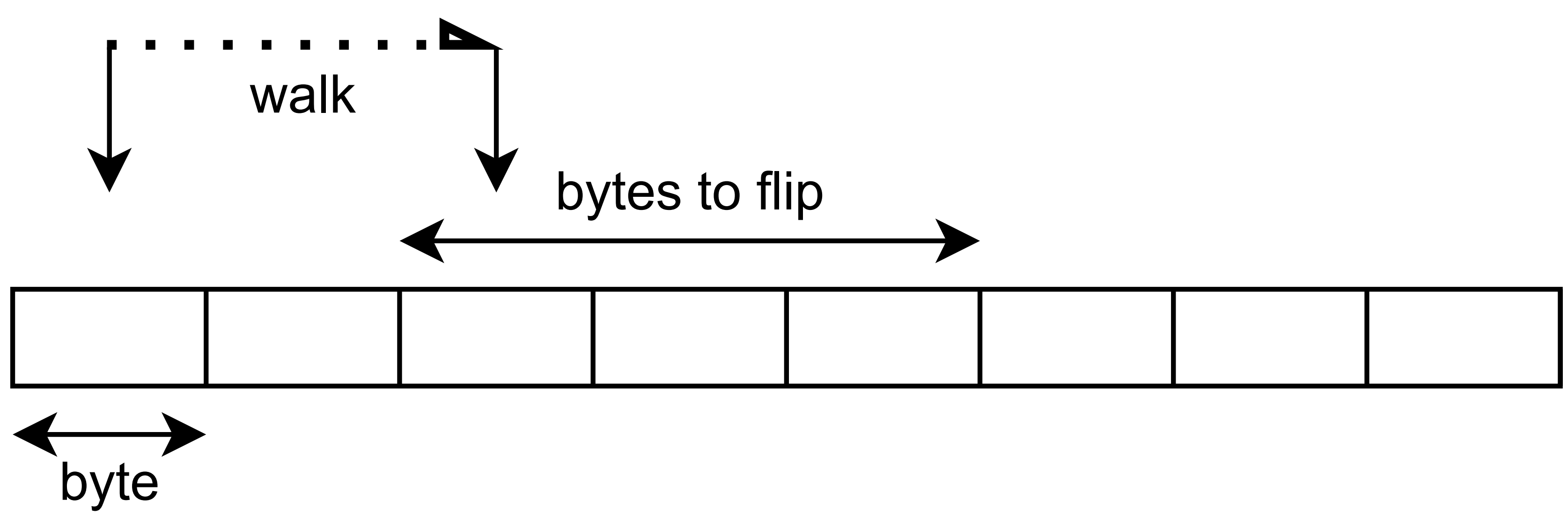
- <u>Input</u>: Given a DUT with two 32b and one 64b input.
    - `input_size = 32 * 2 + 64 = 128 bits`
    - A single cycle of inputs is a bit-string of `input_size` length.
    - Each line in the input file is a cycle's worth of inputs.
    - <u>ex: </u>`0x00FF00'FFFFFF'0000FFFF00007FFF`

OUR SOLUTION: MUTATION ENGINE

- 1st attempt: Direct use of AFL's engine using the JNI.
    - Problem: Compilation time was too long.
        - Need to add more dependencies(scala-jni).
- Solution: Reimplement subset of AFL's engine in Scala.
    - Use some deterministic mutation passes.
    - After that use non-deterministic passes.
    - So far only deterministic passes have been implemented.

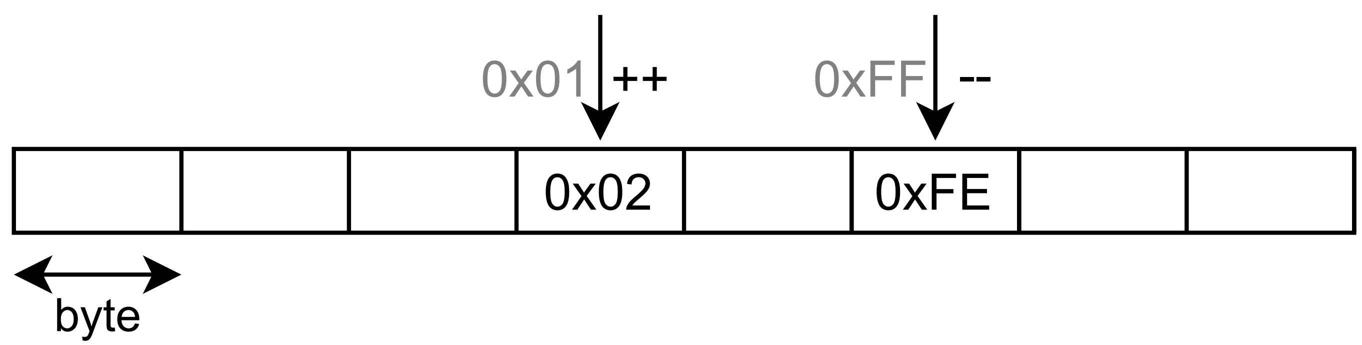
OUR SOLUTION: MUTATION ENGINE

- Deterministic mutation passes:
  - **Walking bit-flips:** Sequentially walk through each input string line **bit-by-bit** and **flip** either 1, 2 or 4 bits/pass.
  - **Walking byte-flips:** Same idea but **byte-by-byte**.

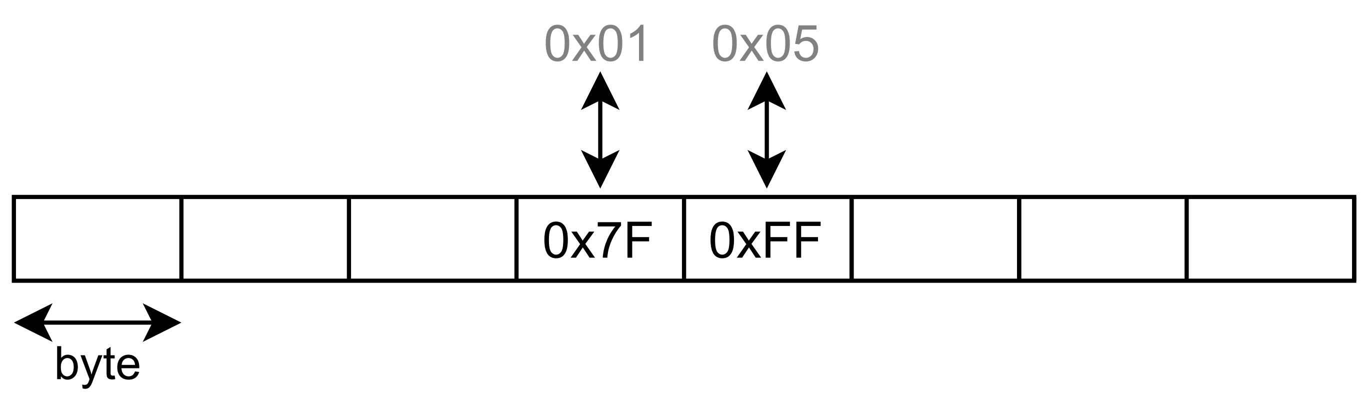


OUR SOLUTION: MUTATION ENGINE

- **Simple arithmetic:** **Add** or **subtract** values to each bit line. Done with multiple incrementation or decrementations of single bytes in the string.



- **Known integers:** **Replace** bytes in the string with predefined "interesting" integers like `0xFF` or `0x7F`.



INITIAL EXPERIMENTS

INITIAL EXPERIMENTS: USE CASE

- Note: Our solution is a work in progress.
- Use case: Leros accumulator ALU:
  - `Input op<3b>, din<8b>; Output out<8b>`
  - Goal: Check for every operation using the most interesting operands.
- What we need to do:
  - 1) Create verification plan for functional coverage.
  - 2) Create input seed file.
  - 3) Run fuzzer.

INITIAL EXPERIMENTS: SETUP FUZZER

- Verification plan:

```
val cr = new CoverageReporter(dut)
cr.register(
    cover("op", dut.input.op)(
        bin("nop", 0 to 0),
        //[...] Bins for each operation
        bin("shr", 7 to 7)),
    cover("din", dut.input.din)(
        bin("0xF", 0 to 0xF),
        //[...] Cover all ranges
        bin("0xFFFF", 0xFFF to 0xFFFF)),
    cover("accu", dut.output.accu)(
        //[...] Same as din
    cover("ena", dut.input.ena)(
        bin("disabled", 0 to 0),
        bin("enabled", 1 to 1)))
```

- Create Input seed: `110 00100000 001 00011001 000 00000000`
- Call fuzzer: `Fuzzer(dut, cr)("output.txt", "seed.bin")`

EVALUATION: IMPACT OF FUNCTIONAL COVERAGE ON EFFICIENCY

- Only initial tests have done so far and results aren't conclusive enough.

- Current work is being done on using the same fuzzer with edge coverage in order to compare the results to functional coverage.

- We **expect** functional coverage to lead to a converging fuzzer in less iterations than with edge coverage. However, the current efficiency of ChiselVerify's FC may also lead to slower fuzzing cycles.

CONCLUSION

CONCLUSION

- Work-in-progress paper is a sketch of how to support testing and verification of digital designs described in Chisel with fuzzing.

- Basis for more detailed performance evaluation when all mutation techniques are added.

- Current work is being done on extending the fuzzing methods for constrained random code generation.

REFERENCES

- *Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen.* ***Rfuzz****: Coverage-directed fuzz testing of rtl on fpgas. In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 1–8, 2018.*
- *Michal Zalewski.* ***American fuzzy lop****. https://github.com/google/AFL.*
- *Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks.* ***Fuzzing hardware like software.*** *CoRR, abs/2102.02308, 2021.*
- *Michal Zalewski.* ***Binary fuzzing strategies****: what works, what doesn't. https://lcamtuf.blogspot.com/2014/08/ binary- fuzzing- strategies- what-works.html.*

GETTING STARTED USING CHISELVERIFY

- **Current Project repository:**

  https://github.com/chiselverify/chiselverify/

- **Project Wiki (Good way to get started):**

  https://github.com/chiselverify/chiselverify/wiki/

- ChiselVerify is published on Maven. To use it, add following line to your `build.sbt`:

```
libraryDependencies += "io.github.chiselverify" % "chiselverify" % "0.1"
```

QUESTIONS?