ISSN: 2834-4928

Research Article

Journal of Electrical Electronics Engineering

An Empirical Study of Fuzz Stimuli Generation for Asynchronous Fifo And Memory Coherency Verification

Renju Rajeev[1]* and Xiaoyu Song [1]

[1]*Department of Electrical and Computer Engineering, Portland State University, Portland, Oregon 97201, USA*

***Corresponding Author**
Renju Rajeev, Department of Electrical and Computer Engineering, Portland State University, Portland, Oregon 97201, USA

Submitted: 2023, Aug 03; **Accepted:** 2023, Aug 23; **Published:** 2023, Sep 14

Citation: Rajeev, R., Song, X. (2023). An Empirical Study of Fuzz Stimuli Generation for Asynchronous Fifo And Memory Coherency Verification. *J Electrical Electron Eng, 2*(3), 302-306.

Abstract

Fuzz testing is a widely used methodology for software testing. It collects feedback of each run and uses it for generation of interesting stimuli in the future. This paper discusses the ability and process of fuzz stimuli generator for hardware verification. We chose an asynchronous FIFO and a memory coherency verification using fuzz [1]. Our results substantiate the effectiveness of fuzz testing in the hardware verification process.

1. Introduction

Pre-Silicon verification is an important effort, which usually is more than 70% of the VLSI design process. The behavioral model of the design written in a high-level language, which models the hardware, such as System Verilog (RTL), needs to be tested/verified in a presilicon/software environment to verify that it works as intended [2]. A verification plan is then developed for the design under test, with the intent for "good enough" verification. This guides the development of test cases, which are unique enough to verify the features required to be tested. Constrained Random Verification (CRV) methodology is a popular strategy, which is then used to generate the stimulus to cover the test plan. The stimuli that find bugs in the design is the most valuable since it results in enabling the analysis of the design bug and fixing it. The stimuli that utilize different complex parts of the design is also valuable, since it verifies that the design works as intended.

Fuzz testing is an effective method used in software testing [3]. Provided a framework to generate stimulus for hardware verification using a popular fuzz test generator AFL [4]. Fuzz test generation is powerful, but if run without controlling the stimulus generation, it can potentially generate a lot of stimuli that may not be useful to generate interesting scenarios. The AFL Fuzz stimuli generator works based on feedback from previous test runs. To collect this feedback, AFL needs to instrument the test program. Based on this feedback, AFL generates stimuli that targeted to cover previously unhit scenarios, resulting in automated stimulus generation for functional and other types of coverage (Line, toggle etc.). This effectively provides us with automated coverage guided stimulus generator. This can be applied to large RTL designs and effectively verify different logic blocks. The fuzzer can be guided/biased to generate stimulus targeting a specific logic block by writing coverpoints in the logic block.

2. Fuzz Testing Asynchronous FIFO

In hardware testing, it is important to fill up the different queues (eg: FIFOs) to hit interesting cases. CRV may not be able to do this without providing good enough constraints that target specific logic in the RTL design. Automating this will provide accelerated coverage closure. An asynchronous FIFO is chosen to study Fuzz stimuli generation for the following reasons.

- Relatively easy enough design to understand.
- Easy to generate states to cover by increasing the depth of the FIFO.
- Easy to add more states to cover by increasing the depth of the FIFO. During run time, each possible combination of the write pointer and read pointer of the memory in the FIFO is to be covered. To create the executable to run AFL, the tool Verilator is used to first convert the Systemverilog RTL of the FIFO to an equivalent C model. This C model is then compiled using the C compiler provided by AFL to create the executable file with fuzz instrumentation [5].

The Stimuli generated by the Fuzz generator is converted to transactions that are understood by the interface of the hardware being verified. For a FIFO, this is relatively easy since there are only two control signals (Write and Read), with four possible combinations. The Fuzz generated stimuli were able to cover all the coverpoints in a relatively fast manner. Since the design is relatively small, CRV is also able to hit most of the coverpoints with similar performance as Fuzz stimuli. Since the number of possible opcodes is 4 (Rd, Wr, RdWr, NoP), it is expected

that CRV is also good at generating good stimuli to hit most coverpoints. This proves that Fuzz generated stimuli is at-least as good as the existing popular and widely used stimuli generation methodology (CRV) for hardware verification.

3. Asynchronous FIFO Architecture

The FIFO's architecture chosen is the standard architecture. The depth of the FIFO used is 64. Fig 1 shows the architecture diagram of the Asynchronous FIFO. The input interface has the clocks, resets (Read and Write), Read, Write and Write Data pins. The output interface has the Full, Empty, and the Read data pins. Internally, it contains instances of a synchronizer module, which is a clock domain crossing module to synchronize the read pointer to write clock domain, and vice-versa. These synchronized pointers are then used by the full and empty generator modules to generate the write full, and the read empty.

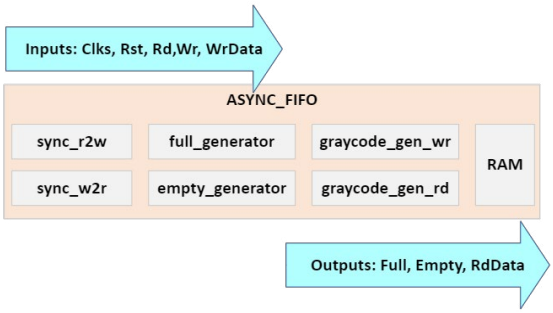

Figure 1: Async FIFO Architecture

4. Bug Insertion and Testing

A logic bug which causes the full signal to assert incorrectly is introduced. An assertion which checks the correct functionality for the full signal is available in the RTL. A cover property is coded to cover the case where the full signal is asserted. This cover property is intended to guide the fuzzer to generate the stimulus to assert full. In CRV based verification, a directed test to verify that the full and empty signal are asserted correctly is to be written by the Design Verification engineer. By using coverage guided fuzzing, the fuzzer will automatically generate such tests, and in the process, generate multiple interesting stimuli. This becomes particularly useful as the design gets bigger, and cover properties detailing interesting cases are coded as part of the design. The fuzzer can act as a bug hunter who is guided by cover properties, which automates generation of stimuli which executes different logic blocks. The fuzzer was able to find the bug and saves the stimulus. The following output from the fuzzer indicates the time taken by it to generate the stimulus which was able to find the bug (88 mins). In the process of finding the bug, it was also able to hit several cover properties coded. CRV was also able to find this bug, within similar run time. Fig 2 indicates that the fuzzer took 1 hour, 28 mins to find the first crash. Each crash is caused when the execution finds a bug.

```
process timing                                    overall results
      run time : 0 days, 1 hrs, 28 min, 56 sec        cycles done : 0
 last new path : 0 days, 0 hrs, 0 min, 0 sec          total paths : 692
last uniq crash : 0 days, 0 hrs, 0 min, 0 sec       uniq crashes : 1
 last uniq hang : none seen yet                       uniq hangs : 0
 cycle progress                       map coverage
 now processing : 59 (8.53%)            map density : 4.34% / 5.81%
paths timed out : 0 (0.00%)           count coverage : 3.26 bits/tuple
 stage progress                        findings in depth
 now trying : havoc                   favored paths : 68 (9.83%)
stage execs : 917/3264 (28.09%)        new edges on : 113 (16.33%)
total execs : 54.4k                   total crashes : 1 (1 unique)
 exec speed : 10.53/sec (zzzz...)      total tmouts : 0 (0 unique)
 fuzzing strategy yields                          path geometry
  bit flips : 7/1480, 2/1474, 0/1462                 levels : 3
 byte flips : 0/185, 0/179, 8/168                   pending : 687
arithmetics : 0/10.4k, 0/5959, 0/2116             pend fav : 64
 known ints : 0/837, 0/4171, 12/6397             own finds : 691
 dictionary : 0/0, 0/0, 0/0                       imported : n/a
      havoc : 639/13.2k, 0/0                     stability : 100.00%
       trim : 0.00%/50, 0.00%
```

Figure 2: AFL Fuzzer output screen for FIFO Fuzzing

The fuzz execution was run on the C model executable file, compiled using the C compiler provided by AFL. This enabled instrumenting the code to collect event coverage feedback by AFL.

Fig 3 shows the waveform indicating the states of the read and write pointer to cause the full signal to be asserted incorrectly.

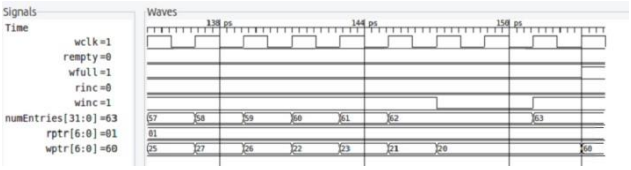

Figure 3: Fuzz Stimuli Which Found the Bug Inserted in the FIFO

5. Memory Coherency Verification Using Fuzz

The behavior of a memory system as described formally in the Memory Consistency Model (MCM) must be verified so that software programmers can rely upon it for parallel programming. We attempt to generate stimuli using fuzz. A memory model created in gem5 memory simulator is used for this. AFL is the fuzz generator used to generate the stimuli. A requirement for fuzz testing is that the fuzz generator should be provided feedback by the target being fuzzed about the coverage achieved by each stimulus generated, and the fuzz generator will use that to generate the future stimuli. AFL provides a compiler tool for C++, which compiles the source code which also instruments the executable file specifically for the feedback purpose. Hence it is important to compile the gem5 memory model implemented in C++ using the compiler provided by AFL. Once this is done, a system with four CPU cores with one dedicated cache per core(L1), one shared cache (L2), and a main memory of a specified size (64MB) can be configured, and run the stimuli generated by the fuzz generator (AFL). The file generated by AFL needs a conversion layer to convert it to a format understood by the memory model. We use a simple decoder implemented for this purpose, which reads the stimuli generated by AFL, and converts it into Command, Address and Data format.

The tests are written in python, which in turn executes the C++ model. Hence the fuzzing is also invoked using python. A python specific package, Python-afl is installed, and a wrapper script is implemented to invoke the python test run command line. Checks which are available in gem5 is enabled to generate an error if a bug is found out. The fuzzer's goal is to send stimulus so that these checks fail and cause the execution to crash. A bug in the state machine is intentionally introduced, and fuzzing is executed to see if it can find this bug [6].

6. Memory Model Using Gem5

The memory model used is the model provided by gem5, which is a widely used model for research. A configuration of 4 processors with one L1 and L2 caches using Two level MESI protocol, and main memory is used to run the fuzz generated stimuli. The model is implemented in C++, and the top-level tests wrappers are written in Python which configures the system and tester. The model is compiled using the C++ compiler provided by AFL to insert the instrumentation code in the executable, which is used by the AFL fuzz generator for collecting feedback of running a stimulus, for the purpose of generating subsequent stimuli. Fig 4 shows the memory model configuration used for running fuzz. Four processes are using a 64MB main memory. Each process has its own L1 Cache. A L2 Cache is shared among the four processes.

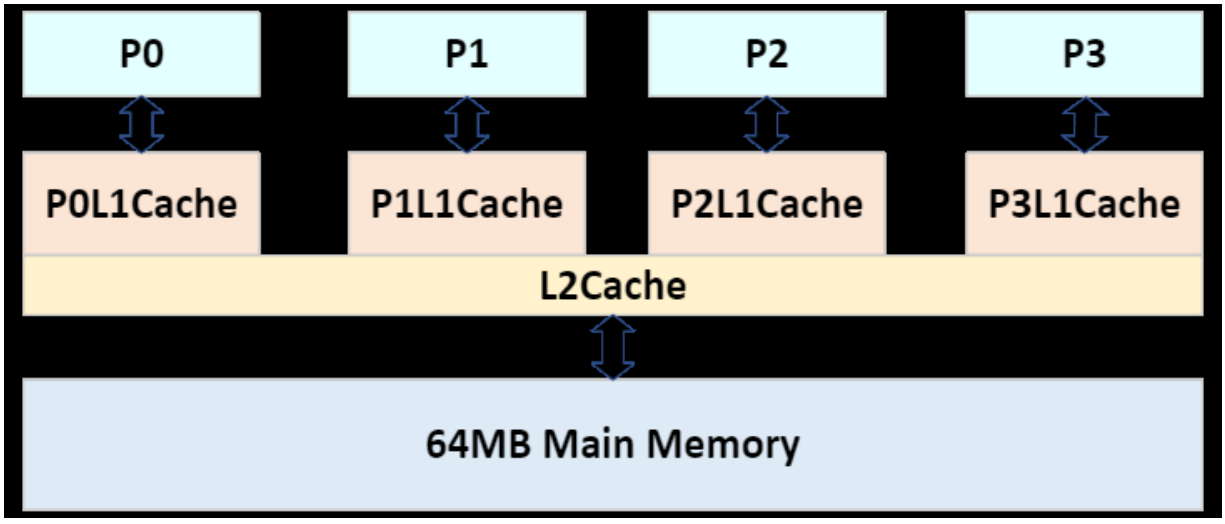

Figure 4: Memory Coherency Model

7. Bug Insertion and Testing

A bug in the L1 Cache's state machine is introduced, with a goal of finding the bug by fuzzing the model. The bug inserted is to not send an invalidation acknowledgement in the "S: Shared" state of the L1 Cache. This will cause a deadlock when an invalidation request is sent to the cache model. A check is implemented to crash the execution when it detects a deadlock. The fuzzer saves the stimuli which was able to cause any crash, which can be used to rerun the test with more debug options and root-cause the crash. A random test (CRV) which generates transactions (Reads/ Writes) per process is used to compare against Fuzz. AFL fuzzer was able to find the bug in 70 minutes,

running 209 executions. Fig 5 indicates that the fuzzer took 1 hr, 10 mins to find the first crash. Each crash is caused when the execution finds a bug. The 209th execution found the bug in this case. CRV took ~3hours with ~500 executions to find the bug. The state of the cache entry should change from Invalid → Shared → Invalid. The transition from Shared to Invalid can be caused by another processor writing to the same address and sending an Invalidation to all other processes. This Invalidation will get stuck since the bug inserted will cause the Invalidation acknowledgement not be sent.

AFL Iterative Algorithm for FIFO/memory test

1: **Find the stimuli to crash the target memory model** *in: Rd/Wr transactions (Addr, Data) out: Order of txns(Stimuli) to expose the bug.*

2: *//AFL dry run*

3: **Run the initial input** to make sure the target doesn't always crash

4: **while** (AFL not done)

5: Generate stimulus (Rd/Wr transaction on a process)

6: Send and observe the response from the DUT

7: Collect feedback from coverage instrumentation

8: If target crashed/timed out, save the stimuli, and restart the test.

9: Set AFL done when coverage is 100%

10: **end while**

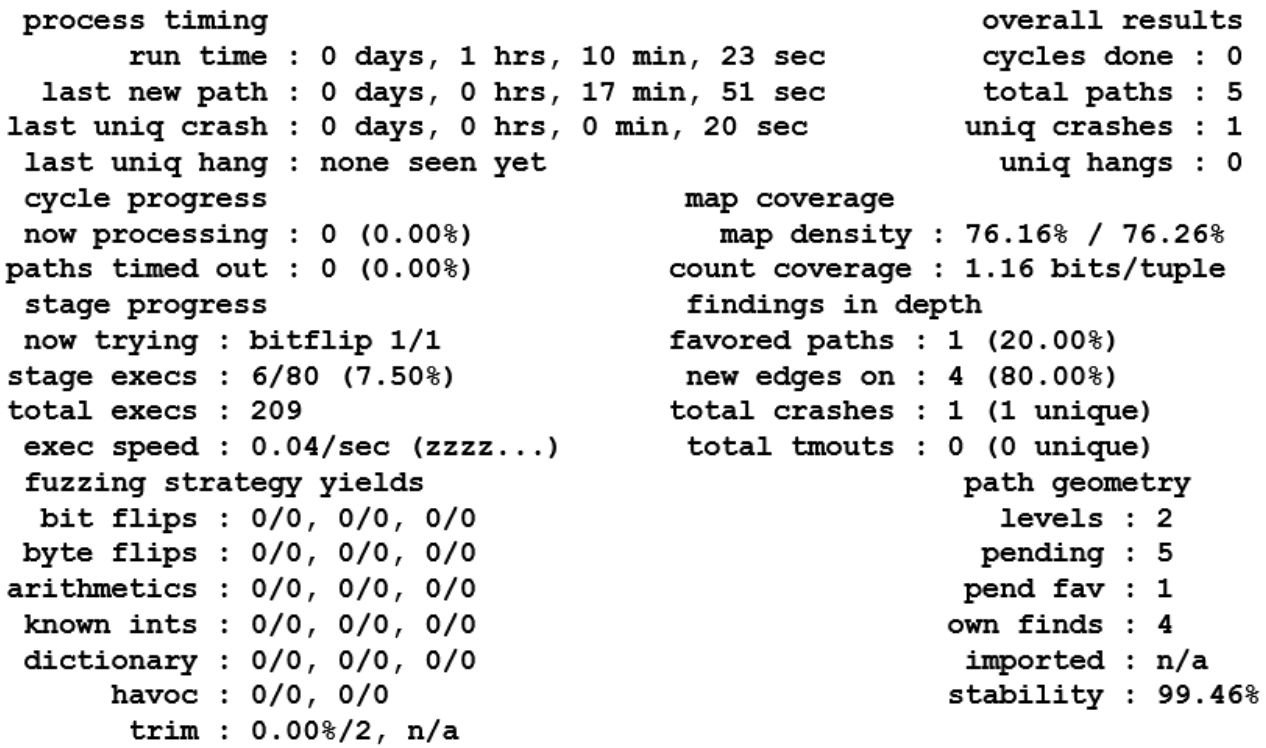

Figure 5: AFL Fuzzer Output Screen For Memory Model Fuzzing

8. Conclusion

In this paper, we have demonstrated the results of using coverage guided AFL fuzzer for hardware verification. The fuzzer was able to use coverage as a guidance to generate stimulus for verifying a FIFO and a memory coherency model. Both designs were inserted with a bug. Fuzz and CRV methodologies were applied to find the bugs [7,8]. For the smaller FIFO design, both methodologies were able to find the bug in a reasonable amount of time. This proves that Fuzz is at least as good as CRV. With Memory model testing, Fuzz was able to find the bug in about half the time taken by CRV. This proves that Fuzz's methodology of using coverage as a feedback mechanism can accelerate verification process.

Acknowledgment

"This article's(chapter's) publication was funded by the Portland State University Open Access Article Processing Charge Fund, administered by the Portland State University Library".

References

1. Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., & Wood, D. A. (2011). The gem5 simulator. ACM SIGARCH computer architecture news, 39(2), 1-7.
2. Laeufer, K., Koenig, J., Kim, D., Bachrach, J., & Sen, K. (2018, November). RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (pp. 1-8). IEEE.
3. Trippel, T., Shin, K. G., Chernyakhovsky, A., Kelly, G., Rizzo, D., & Hicks, M. (2022). Fuzzing hardware like software. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 3237-3254).
4. Nossum, V., & Casasnovas, Q. (2016). Filesystem fuzzing with american fuzzy lop. In Vault Linux Storage and Filesystems Conference.
5. Snyder, W. (2004). Verilator and systemperl. In North American SystemC Users' Group, Design Automation Conference.
6. Elver, M., & Nagarajan, V. (2016, March). McVerSi: A test generation framework for fast memory consistency verification in simulation. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 618-630). IEEE.
7. Holmes, J., Ahmed, I., Brindescu, C., Gopinath, R., Zhang, H., & Groce, A. (2020). Using relative lines of code to guide automated test generation for Python. ACM Transactions on Software Engineering and Methodology (TOSEM), 29(4), 1-38.
8. Luo, D., Li, T., Chen, L., Zou, H., & Shi, M. (2022). Grammar-based fuzz testing for microprocessor RTL design. Integration, 86, 64-73.