New Perspectives on Core In-field Path Delay Test

New Perspectives on Core In-field Path Delay Test

Riccardo Cantoro, Dario Foti,
Sandro Sartoni and Matteo Sonza Reorda
Politecnico di Torino
{riccardo.cantoro, sandro.sartoni, matteo.sonzareorda}@polito.it
s251581@studenti.polito.it

Lorena Anghel and Michele Portolan
Univ Grenoble Alpes, CNRS, Grenoble INP,
TIMA, 38000 Grenoble, France

***Abstract*—Path Delay fault test currently exploits DfT-based techniques, mainly relying on scan chains, widely supported by commercial tools. However, functional testing may be a desirable choice in this context because it allows to catch faults at-speed with no hardware overhead and it can be used both for end-of-manufacturing tests and for in-field test. The purpose of this article is to compare the results that can be achieved with both approaches. This work is based on an open-source RISC-V-based processor core as benchmark device. Gathered results show that there is no correlation between stuck-at and path delay fault coverage, and provide guidelines for developing more effective functional test.**

***Index Terms*—software-based self-test, software test library, on-line test, path-delay test, safety, scan test, LOC**

I. Introduction

Semiconductor companies are developing new advanced technologies that require more complex and sophisticated manufacturing processes. The complexity is related to the shorter channel length that allows high working frequencies and dense designs. This advantage, however, comes at the price of more frequent physical defects and shorter device lifespan. Some of these defects cannot be tested by adopting the traditional stuck-at fault model anymore, as they can only be modeled by delay fault models. Path delay faults, however, although already known in academia, are not supported as extensively as stuck-at faults by EDA tools. This poses new challenges in the matter of testing.

Delay faults are usually addressed by employing Design for Testability (DfT) solutions. The most commonly adopted ones rely on the use of scan and *Built-In Self-Test* (BIST). Their main advantage is that they are based on mature technology, currently supported by most commercial tools; on the other hand, however, they introduce additional hardware that may lead to an area overhead and timing performance decrease. Moreover, routing of clock signals may be further complicated by the insertion of the scan and BIST circuitry.

Functional solutions, mainly in the form of *Software-Based Self-Test* (SBST) [1], could be considered as well. They are reliable, affordable, and applicable even in situations where accessibility is reduced. Test is performed by running a Self-Test Library (STL), hence at-speed test, crucial in delay testing, can be conducted. Only functional patterns are applied; as a consequence, overtesting is avoided and power consumption is reduced. Finally, functional solutions are suitable to be applied for in-field test, too. Currently, STLs are commonly provided by several semiconductor companies as companion products of their CPUs or SoCs wherever they are used in safety-critical applications where comprehensive in-field testing is required [2]–[7]. Unfortunately, no commercial tool currently supports functional path delay test of sequential circuits.

The main goal of this work is to report the results of a detailed analysis on the effectiveness of functional path delay test, by means of an SBST approach on an open-source pipelined RISC-V CPU, and to compare them against the results obtained with a DfT approach. To do that, we developed a flow to perform functional delay fault simulation on sequential circuits based on commercial tools integrated through Bash/Python scripts, while the DfT-based analysis is carried out by using standard functionalities offered by ATPG and fault simulation tools. To perform the functional fault simulation process, we evaluated a set of test programs originally developed for stuck-at faults. Some insights on testability and test generation can be evinced, highlighting some limitations in the currently available test flow for path delay faults, and suggesting new ideas for improving the effectiveness of functional test. To the best of our knowledge, this is the first work that reports quantitative results on the path delay fault coverage that can be achieved resorting to a functional approach on a pipelined CPU core.

This paper is organized as follows: in Section II, a background on the path delay fault model, commercial tools, and related works is given. Section III describes the flow we developed to perform functional path delay test. Section IV gives details about the case study, while the achieved results, as well as some considerations, are presented in Section V. Finally, Section VI draws the conclusions.

II. Background

A. Path Delay test

In a generic synchronous system, input and output signals are synchronized with respect to a specific periodic clock signal. All signals are supposed to assume a steady value within a certain time frame marked by clock signal edges, indicated as *clock period*. The *Path Delay* fault (PDF) can be defined as the modelling of a defect, distributed along a path, that affects the nominal propagation delay of that path causing a specific state transition to happen later than the sampling edge of the clock. A circuit can pass a delay test if it produces correct outputs when specific inputs are given at the maximum

978-1-7281-9113-3/20/$31.00

working frequency. Since delay testing requires to generate and catch a state transition, test patterns for PDFs must be composed of pairs of test vectors to be applied in succession.

Nowadays, many commercial tools specialized in fault simulation are available. Such tools can fully handle stuck-at faults, the analysis of which is the de-facto standard in the testing industry. Moreover, some of them also support gate delay faults, such as transition delays. Path delays, on the other hand, are still not as popular as stuck-at and transition ones; even though they are supported by EDA tools, they imply the presence of DfT modules, generally by means of classic scan chain protocols, such as Launch-on-Shift (LOS) and Launch-on-Capture (LOC), thus restricting the effort to the combinational part of circuit[1]. Unfortunately, to the best of our knowledge, no commercial tool fully supports sequential fault simulation of very long sequences of patterns, such as SBST programs.

B. Related Works

Numerous works on path delay faults can be found in literature. To start off, [8] reports a comprehensive overview of the state-of-the-art on delay faults. Our work is also complementary to the ones presented in [9], [10], as they elaborate on the insertion of monitors at the end of the most critical paths in order to make sure their arrival time does not exceed the nominal clock period. The analyses these papers perform are mainly based on aging effects. Works [11]–[13] focus on the generation of test programs for path delay faults, either by means of identifying functional constraints or by using evolutionary tools. These approaches, however, are tested on non-pipelined processors, thus avoiding the problem of propagating detected delay faults through the pipeline stages. Moreover, custom fault simulation tools are used in all previous works. [14], [15] present two different methods for generating test programs for computational blocks in pipelined processors. Such works aim at testing specific combinational datapath modules inside the CPU, thus not dealing with the complexity of exciting and observing faults inside a pipelined module. Finally, the authors of [16] present a test pattern generation method based on a graph representation of pipelined processors. However, obtaining a graph model for internal modules of a CPU may not be an easy task when dealing with complex processor cores; moreover, an ad-hoc fault simulation has been adopted.

Despite the presence of several works on the functional test of path delay faults on processor cores, a thorough comparison between results achieved by adopting DfT solutions against those achieved by means of a purely functional approach (such as SBST to fully pipelined CPUs) is still missing.

[1]Such tools allow the user to change the number of *capture cycles* during LOC, which corresponds to the case when the fault effect is propagated in the sequential logic for some clock cycles before capturing the results into the scan chains; however, the number of capture cycles is limited, thus making this approach not suitable for an SBST scenario, in which fault effects may be latent for a significant amount of clock cycles.

III. Functional Fault Simulation Flow

The flow we developed for path delay functional test is based on a set of commercial tools coordinated together by means of Python and Bash scripts and it is devised for functional testing, meaning that the netlist is supposed to either not use any scan chains or to not have such hardware at all. Before launching the fault simulation process, few preliminary steps are needed. First, the device under test (DUT) has to be synthesized such that the combinational cells of the gate-level netlist are separated from the sequential ones, thus producing a *combinational netlist*, containing only combinational elements, as well as a *top-level netlist* that consists of the combinational and sequential cells. The latter netlist is then employed in a logic simulation, in which the test program is simulated to record the golden responses from the combinational and top-level circuits into a *patterns* list. Lastly, it is necessary to extract a list of combinational paths from the combinational netlist with the most stringent timing requirements to be stored into a *path definition* list. Such task is performed by means of a Static Timing Analysis (STA) tool and the paths produced are those that will be tested when running the fault simulation. It is worth mentioning that STA tools are not able to recognize *false paths* that could introduce untestable faults: a pruning of those paths from the path definition list may be needed, e.g., as detailed in [17]. Once these steps are cleared, the fault simulation process is launched. To start a combinational-level fault simulation is performed, where paths from the path definition list are tested with the patterns stored in the patterns list. This allows to identify all PDFs that produce a difference on a primary output (PO) or pseudo primary output (PPO) port and the clock period when this happens. If available, more accurate results can be achieved by enabling the no fault-dropping option, i.e., by never deleting faults, even when detected, from the active fault list. This allows us to consider, in the following steps, the propagation of fault effects through the sequential logic not just for the first pattern, but for the whole test set. The last step consists of propagating the effects of detected faults throughout the sequential logic and check whether it reaches an observable point. We modeled this sequential-level fault simulation by means of translating each detected fault, together with its possible propagation endpoint and the time instant at which it reaches the sequential element, into a bit-flip applied to faulty path endpoint at the aforementioned time instant.

IV. Case Study

This section introduces the chosen DUT benchmark as well as a set of test programs that will be used to conduct path delay fault simulations. Such test programs were originally designed for stuck-at fault testing and they are representative of the current state-of-the-art in test program generation [18], which mainly focuses on this fault model.

A. Processor Core

The adopted benchmark device is an open-source single-core SoC platform based on a 32-bit RISC-V core, developed

by ETH Zurich and Università di Bologna, named PULPino [19]. In this work, PULPino was configured to use the RI5CY core, an in-order, single-issue core with 4 pipeline stages. The experiments we performed focused only on the core that was isolated from other boundary components such as peripherals. The processor core was synthesized using the 45nm Silvaco Open Cell library (former Nangate Open Cell library) [20], with a clock period of 5ns and a total area (equivalent gates) of $51,001.65$. The hierarchy was flattened and then processed after synthesis to allow us to focus on the combinational logic. The list of paths was obtained from the synthesized core after the STA process. In order not to focus too much on a single module of the CPU, a maximum of 10 most critical paths per endpoint were extracted, with a total of 17,738 paths withing a slack range of $[0.37 : 4.95]$ns. As the fault list minimization was not among the objectives of this work, no pruning of functionally untestable faults was performed. The only faults that were identified and removed by the fault simulation tool are the structurally untestable faults.

B. Test Programs

A total of five stuck-at fault (SAF) oriented test programs were considered. A brief explanation of their characteristics is given in this paragraph.

Program 1 is a medium-sized test program composed of a series of macros, each one targeting a separate functionality of the core. The code includes dedicated test macros for each core unit, that allows to reach above 90% of SAF coverage on crucial modules like the ALU, multiplier, and register file. The macros are composed of blocks of instructions followed by blocks of store instructions.

Program 2, instead, is a short test program organized as multiple couples of instructions, where arithmetic operations are immediately followed by a store operation. The approach was to explicitly target the register file, multiplier, ALU, and the Control and Status Register (CSR), while other modules were covered as a side effect. Random values were used to test arithmetic units, while the register file were tested using the method presented in [21]. Other parts of the design, such as hardware loops, and CSR registers, were tested with custom algorithms.

Program 3 is the shortest among the set of programs. The code was randomly generated by a suitable script and it is structured by groups of instructions composed of two register loads, one arithmetic operation and then one store instruction. This program also includes instructions for vector operation, hardware interrupts, and performance counters reading.

Program 4 is the longest one. The approach of the program 4 is also based on macros. At the beginning of each macro, the register file is cleared; then, new values are loaded in registers that will be used as source operands in the subsequent couple of instructions (any instruction, except for load or store ones, can be used here). Lastly, a store instruction is issued, to save the previously computed result. This program is very interesting as it covers the most important modules in the design, reaching above 85% of fault coverage.

The last program was obtained by randomly generating instructions targeting the ALU module, only. Its structure is based on groups of instructions where the register file is loaded with immediate random values, then a random number of arithmetic instructions (such as add, mul, div, and rem) are performed also in random order and finally all the registers values are stored back in the memory.

The goal pursued with our experiments was to evaluate the path delay fault coverage achieved by each program and compare the functional fault coverage with the coverage achieved by current scan-based benchmark test procedures.

V. Fault simulation data

In this section, we analyze the results gathered on the various fault simulation experiments. It is worth noting that the results obtained in this paper are not tool dependent, as each step has been cross-validated with similar EDA tools from different vendors.

The presented experiments have been run on 5 cores of an Intel Xeon CPU E5-2680 v3. The whole functional simulation flow required indicatively 48 to 72 hours for each program.

In order to set a reference for the following analysis, we analyzed the combinational logic of the RI5CY core only, i.e., assuming that its input/output signals are fully controllable/observable with an ATPG process. The ATPG engine produced a PDF coverage of 38.79%, with 13,762 detected faults, and 21,714 untestable faults. This is the ideal upper-bound for the achievable fault coverage that can hardly be achieved even when adopting scan-based — i.e., LOC — tests, due to constraints imposed by the test procedure.

Furthermore, in order to perform a more realistic estimation of the coverage, we launched an ATPG process targeting LOC. To do so, a scan version of the core netlist was produced. This choice was driven by the need to have comparable results, since we use the same fault list for both scan-based and functional tests. The scan chain featured 1,300 MUX D flip-flops. LOC analysis led to the following conclusion: 12,389 (34.92%) — corresponding to 90% of the faults detected by the previous reference experiment — of the faults can be detected with LOC, while 23,087 have been marked as untestable. The higher number of untestable faults may be attributed to the constraints imposed by LOC. A comparative summary of these experiments is reported in Table I.

TABLE I
COMBINATIONAL ATPG VS. LOC COMPARISON

	Detected faults	Untestable faults	Total faults	FC%	FC% wrt reference
Comb. ATPG	13,762	21,714	35,476	38.79%	100.00%
LOC	12,389	23,087	35,476	34.92%	90.02%

A. Combinational-level fault simulation

Fault simulation results of the test programs on the combinational logic are summarized in Table II.

For each test program, the table reports information about the amount of test patterns corresponding to its execution, the

TABLE II
COMBINATIONAL-LEVEL FAULT SIMULATION RESULTS

	Progr. 1	Progr. 2	Progr. 3	Progr. 4	Random	Cumulative
Test patterns	64,502	36,394	17,269	118,098	32,416	268,679
Detected faults	6,816	6,973	6,856	7,554	6,573	8,085
Fault coverage%	19.21	19.66	19.33	21.29	18.53	22.79
FC% wrt refer.	49.5	50.67	49.81	54.89	47.76	58.75

number of detected faults, the fault coverage, and the fault coverage normalized with respect to the reference experiment. The last column, *Cumulative*, reports the aforementioned data in a cumulative fashion, as if all test programs could be collapsed into a single program.

The fault coverage values of Program 1, 2, and 3 are differing by just few decimal digits, although Program 3 is faster. This is of particular importance when considering that the test programs have been devised with different techniques and structures being also very heterogeneous in terms of duration and number of instructions. It also suggests that no correlation between SAF and PDF functional coverages may exist.

B. Sequential-level fault simulation

We injected faults detected at the combinational level after removing those affecting specific ports like POs, clock or enable signals (detected by implication), as their effect is instantly noticeable on the whole system's behaviour. Moreover, in order to decrease this fault simulation cost, we grouped together all faults producing a bit-flip on the same PPO during the same clock cycle (equivalent faults). Table III presents results of the sequential-level fault simulation.

TABLE III
FUNCTIONAL FAULT SIMULATION RESULTS

Parameter	Program 1	Program 2	Program 3	Program 4	Random
Injected	6,484	6,772	6,629	7,067	6,333
Det. by Simulation	4,797	5,420	4,781	6,660	5,011
Det. by Implication	332	201	227	487	206
Fault Coverage%	14.45	15.84	14.11	20.14	14.12
FC% wrt reference	37.27	40.84	36.39	51.93	37.91
Prop. Coefficient%	75.25	80.61	73.04	94.61	76.23

In the set of injected faults, those that caused an observable misbehaviour on POs are indicated as *Detected by Simulation*. Such faults, together with those detected by implication, were used to compute the fault coverage. In the table, we also reported the fault coverage with respect to the reference experiment. Finally, the *Propagation Coefficient* is the percentage of faults that have been successfully propagated to a PO among those faults detected at combinational level.

It is possible to see that program 4 achieved the best results in terms of both faults detected at combinational level $(7,554)$ and propagation coefficient (94.61%). This result may be partly due to the longer duration of the program 4. Program Random shows very interesting results as well. Despite the large gap in terms of fault coverage for SAF and TDF, the functional test results are comparable to what has been achieved with other test programs. This brings us again to the conclusion that no correlation among the considered fault models coverage exists.

C. Test programs effectiveness

Finally, we have performed an analysis of the effectiveness of the test programs to understand how delay faults on paths with different slacks are covered. Moreover, we wanted to see if there is a correlation between detected faults and the associated path slack value.

The distribution of the considered PDFs with respect to slack intervals is presented in Table IV. Each line takes into account a 0.5ns step interval; if a path owns a slack ranging in that interval, it will be counted in the row. The absolute number of faults per interval is reported in the first column, while in the other columns, we report the fault coverage achieved by combinational ATPG patterns, LOC patterns and functional test programs, respectively. Interestingly, all faults belonging to the slack interval [0.0-1.5]ns were not detected by any test method. These numbers are in accordance with data shown in previous publications [22], [23]. Delay faults untestability was not further investigated as it is not a focus of the current paper and will be analyzed in future works.

TABLE IV
FAULT COVERAGE PER SLACK RANGE

Slack intervals [ns]	Total faults	Comb. ATPG FC%	LOC FC%	Functional test programs FC%
[0.0 - 0.5]	4,672	0.0	0.0	0.0
[0.5 - 1.0]	3,228	0.0	0.0	0.0
[1.0 - 1.5]	640	0.0	0.0	0.0
[1.5 - 2.0]	248	79.4	67.3	7.2
[2.0 - 2.5]	388	66.0	56.7	7.0
[2.5 - 3.0]	422	64.6	56.6	6.6
[3.0 - 3.5]	4,670	5.8	5.2	0.5
[3.5 - 4.0]	7,176	12.7	6.9	0.3
[4.0 - 4.5]	3,968	56.1	41.3	3.6
[4.5 - 5.0]	10,064	95.0	93.2	75.1

In order to identify the most critical faults and how those are covered by the test methods considered in this work, we have analyzed the faults affecting the arithmetic blocks within the core. Table V reports such information. We highlight the fact that not all faults from the fault list traverse one of these modules; in some cases, one path could traverse more than one arithmetic block.

TABLE V
FAULT COVERAGE PER MODULE

Module name	Total faults	ATPG comb. FC%	ATPG scan FC%	Functional fault sim. FC%
id_stage_i_add_531	160	100.0	98.1	0.0
alu_i_int_div_div_i_sub_100	12	50.0	50.0	33.3
alu_i_int_div_div_i_add_100	446	25.8	25.8	25.8
ex_stage_i_mult_i_add_109_2	1,580	0.0	0.0	0.0
ex_stage_i_mult_i_mult_109	1,580	0.0	0.0	0.0
cs_registers_i_add_775	132	100.0	100.0	0.0
load_store_unit_i_mult_add_463_aco	1,884	72.4	72.0	21.8
load_store_unit_i_add_463_aco	1,994	73.5	73.0	24.6
r1589	868	100.0	85.8	0.0
ex_stage_i_alu_i_add_168	6,960	0.0	0.0	0.0
ex_stage_i_alu_i_add_182	6,960	0.0	0.0	0.0

Furthermore, we analyzed the effort required to propagate each fault from a PPO to a PO. As a reminder, each fault is possibly detected at combinational level. In the positive case, the difference with respect to the fault-free circuit at a given clock cycle may possibly propagate to a PO; in such a case, the fault is detected. As an indication of the testability of a fault, we can measure how many times the fault produces a difference on a PPO before being detected. As an example, Table VI reports about the results measured by analyzing Program 4. Most of the faults (nearly 80%) are immediately detected, as soon as they produce a difference on a PPO. Only a subset composed of about 20% of the faults require a significant number of differences on PPOs at different clock cycles before being detected. Currently, we are unable to state whether we could speed-up the detection of the latter subset of faults by a more careful design of the test programs, or whether their detection strictly requires longer test programs.

TABLE VI
NUMBER OF DETECTED FAULTS VS. NUMBER OF DIFFERENCES ON PPOS

Differences on PPOs	1	10	30	64
Fault detected	5,712	6,552	6,918	7,146
(%)	79.9	91.7	96.8	100.0

Our results show that further efforts are required to identify suitable techniques to generate effective test programs able to achieve a higher PDF fault coverage. These results can be achieved first of all by increasing the number of PDFs that produce a difference at combinational level and, further to that, by increasing the number of faults whose effects can be propagated up to the POs, that is, by bettering the propagation coefficient. With our experiments, we clearly show that the correlation between the ability of a test program to detect PDFs, and the ability to detect SAFs and TDFs is quite low. Table VII collects the fault coverage percentages for each program on the three fault models mentioned in this work.

TABLE VII
FAULT COVERAGE SUMMARY

Parameter	Program 1	Program 2	Program 3	Program 4	Random
Clock cycles	64,527	36,500	17,308	181,370	32,455
SAF FC%	86.77	81.79	81.37	82.97	59.44
TDF FC%	41.90	44.21	63.16	61.90	24.41
PDF FC%	14.45	15.84	14.11	20.14	14.12

VI. CONCLUSIONS

In this work we evaluate the effectiveness of functional techniques based on the SBST paradigm applied to the path delay fault model in a pipelined CPU core and compare the achieved results with those provided by scan-based techniques.

Results show that a large portion of faults was detected once they produce a difference on an endpoint. Another important conclusion is that more than a half of the faults detected by scan tests was also detected with functional approaches. Identifying functionally untestable faults is still an open issue; moreover, some of the most critical faults are not functionally observable until they create several differences at combinational level.

We are currently working at devising solutions to effectively generate functional test programs able to achieve higher fault coverage figures with respect to PDFs.

REFERENCES

[1] M. Psarakis *et al.*, "Microprocessor software-based self-testing," *IEEE Design Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.

[2] Hitex. [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs

[3] STMicroelectronics. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/application_note/02/1a/91/78/e4/15/4d/35/CD00290100.pdf/files/CD00290100.pdf/jcr:content/translations/en.CD00290100.pdf

[4] Cypress Semiconductor. [Online]. Available: https://www.cypress.com/file/249196/download

[5] Renesas Electronics. [Online]. Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html

[6] Microchip Technology Inc. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf

[7] ARM. [Online]. Available: https://developer.arm.com/technologies/functional-safety

[8] J. Mahmod *et al.*, "Special session: Delay fault testing - present and future," in *2019 IEEE 37th VLSI Test Symposium (VTS)*, 2019, pp. 1–10.

[9] Z. Ghaderi *et al.*, "Sensible: A highly scalable sensor design for path-based age monitoring in fpgas," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 919–926, 2017.

[10] A. Sivadasan *et al.*, "Nbti aged cell rejuvenation with back biasing and resulting critical path reordering for digital circuits in 28nm fdsoi," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 997–998.

[11] Wei-Cheng Lai *et al.*, "Test program synthesis for path delay faults in microprocessor cores," in *IEEE International Test Conference*, 2000, pp. 1080–1089.

[12] P. Bernardi *et al.*, "On the automatic generation of test programs for path-delay faults in microprocessor cores," in *12th IEEE European Test Symposium (ETS'07)*, May 2007, pp. 179–184.

[13] K. Christou *et al.*, "A novel sbst generation technique for path-delay faults in microprocessors exploiting gate- and rt-level descriptions," in *26th IEEE VLSI Test Symposium (vts 2008)*, April 2008, pp. 389–394.

[14] N. Hage *et al.*, "On testing of superscalar processors in functional mode for delay faults," in *30th IEEE International Conference on VLSI Design and 16th IEEE International Conference on Embedded Systems (VLSID)*, 2017, pp. 397–402.

[15] C. H. . Wen *et al.*, "On a software-based self-test methodology and its application," in *23rd IEEE VLSI Test Symposium (VTS'05)*, 2005, pp. 107–113.

[16] V. Singh *et al.*, "Instruction-based self-testing of delay faults in pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1203–1215, Nov 2006.

[17] J. Chen *et al.*, "Identification of testable representative paths for low-cost verification of circuit performance during manufacturing and in-field tests," in *32nd IEEE VLSI Test Symposium (VTS)*, April 2014, pp. 1–6.

[18] P. Bernardi *et al.*, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744–754, 2016.

[19] ETH Zurich and Università di Bologna. [Online]. Available: https://github.com/pulp-platform/pulpino

[20] Silvaco. [Online]. Available: https://www.silvaco.com/products/nangate/FreePDK45_Open_Cell_Library/

[21] D. Sabena *et al.*, "A new sbst algorithm for testing the register file of vliw processors," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 412–417.

[22] N. Ahmed *et al.*, "Timing-based delay test for screening small delay defects," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 320–325.

[23] X. Fu *et al.*, "Testable path selection and grouping for faster than at-speed testing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 2, pp. 236–247, 2012.