

POLITECNICO DI TORINO

Master's Degree in ELECTRONIC ENGINEERING



Master's Degree Thesis

Advanced techniques for testing delay faults

Supervisors:

Prof. Matteo SONZA REORDA

Prof. Riccardo CANTORO

Prof. Arnaud VIRAZEL

Prof. Patrick GIRARD

Candidate:

Gianmarco MONGELLI

Academic Year 2021-2022

Abstract

The new semiconductor technologies are characterised by an higher and higher frequency. So, testing delay faults of a circuit is becoming increasingly important. These types of faults affect the timing behaviour of the circuit and they can be divided in *Transition Delay Faults* (TDFs) and *Path Delay Faults* (PDFs). In this study the transition delay model is used because TDFs are much better supported by both standards and EDA tools than PDFs. The selected method for testing the delay faults is the functional test in the form of the Software-Based Self-Test (SBST). The main problem of this approach is that implementing an efficient test program could be not trivial for the engineer. So, this study aims to help the testing engineer to develop in a better way a test program increasing the observability of the injected faults, exploiting *Modelsim* and its functionalities because of the deep internal access the tool has on the DUT.

A flow is created and validated on a simple circuit, and then applied to the PULPINO processor. The flow works on the PULPINO environment compiling the self-test code and comparing the results generated by the golden simulation with the ones generated by each fault simulation corresponding to each signal of the fault list. If the results of the two simulations are not equal it means that the injected fault affects the behaviour of the processor, so the selected fault is observable.

The experiments show the difference between golden and faulty simulation in terms of time simulation, clock cycle of each instructions, type of instruction, address or content of the involved registers, etc. Therefore, the test engineer could exploit these results to understand how to improve the test program. The obtained results are compatible to the ones obtained using specific tools for testing, as Z01X and *Tetramax*.

Table of Contents

1	Introduction	1
2	Basics and backgrounds	4
2.1	Transition delay faults	5
3	User Manual of the environment	7
3.1	Flow description	8
3.2	Environment structure	8
3.3	How the environment works and how to use it	12
3.3.1	Importing the environment into a new computer file system and making it workable	12
3.3.2	Running the flow	13
3.3.3	Interpretation of reports	16
3.3.4	How to speed-up the execution time of the master script . .	18
3.3.5	The simulation script	20
3.4	Conclusion	21
4	Chronological steps to build the environment	22
4.1	Initial setting problems to solve	23
4.2	The tracer: comparison mechanism of the flow	23
4.2.1	Tracer instantiation in structural processor	26
4.2.2	Tracer instantiation in gate level processor	27
4.3	Validation of the FI flow	32
4.3.1	Analysis of the validation environment	34
4.3.2	FI script explanation	35
4.4	Fan-out problem and how to solve it	36
4.4.1	Modification of the tech library	37
4.4.2	Analysis of the validation environment of the fan-out solution	39
4.5	Validation of the new FI script	40
4.6	Application of the flow on the processor	41

5	Experimental results	42
5.1	Validation of the tracer on the gate-level processor model	42
5.2	Validation of the FI flow	44
5.2.1	Report_fi_check	45
5.2.2	Report_100 and report_constr_a_85_71	47
5.2.3	Report_100_mod_libr, report_85_mod_libr, check_libr and check_libr_85	49
5.3	Validation of fan-out solution	50
5.4	Experiments on the flow	54
5.4.1	Tracer_start_code	55
5.4.2	Tracer_division_code	56
5.4.3	Tracer_division_code_speed	56
6	Conclusions and perspectives	58
	Bibliography	61

Chapter 1

Introduction

The new advanced semiconductor technologies adopted in emerging applications are characterised by an increasing frequency and computational capabilities. Such technologies are extremely complex and sophisticated, leading to more frequent physical defects and a reduced operative lifetime [1].

So, testing the integrated circuits became more difficult. Most of the tested defects are modeled as delay faults because of the high frequency reached. These kinds of faults usually affect the timing behaviour of the circuit and they are divided in two categories: *Path Delay Faults* (PDFs) and *Transition Delay Faults* (TDFs). In these studies the second one will be considered. In order to test these kinds of faults two approaches could be used:

- *Design-for-Testability* (DFT) in which additional hardware modules are added to the Device Under Test (DUT). Some examples can be Logic BIST or scan chains. The problems of these solutions are that an overhead in terms of performance or timing can be introduced and the *overtesting* could be generated because some functionally untestable faults could be tested.
- *Functional testing*, in the form of Software-Based Self-Test (SBST). It is based on the execution of a test program containing a set of Self-Test Libraries (STLs) by the DUT. The results produced by the test program are then compacted into a signature that is compared against the golden circuit's one to look for the presence of structural faults [1].

This method is cheaper and more suitable than DFT and it is usually performed as in-field testing, when the device's safety has to be guaranteed throughout the operative lifetime.

In this research the selected method for testing delay faults is the functional test in the form of the SBST. Its *issue* is that developing an effective test program could be not trivial for the testing engineer. In fact, the implemented program should be able to reach an high Fault Coverage (FC) forcing as much as possible the propagation of faults to the Primary outputs (POs).

So, the idea to mitigate this problem is to provide to the test engineer an automatic way to get more information on the behaviour of the DUT (PULPINO processor as case study) for each specific fault injection (FI) performed. This is done improving the observability of the faults through a generic and systematic flow [1]. Obviously, the main objective is to get more information on the not detected faults trying to understand why these faults are not propagated to the POs.

Modelsim [2] tool is exploited to build the flow, thanks to its better internal accessibility in terms of modules and signals inside the DUT compared to other tools as *Z01X* [3] or Tetramax[4].

The development of the test flow just mentioned is the main purpose of this research and it will be described in detail in the rest of the manuscript.

This manuscript is composed of 6 sections, including the introduction (chapter 1). The other sections are: Chapter 2 in which the main topics of this study in particular on the TDFs are summarised, chapter 3 in which a deep explanation of how the flow and its environment from the point of view of the user is provided, chapter 4 in which all the basic steps to build the final flow are described, chapter 5 in which all the experiments are reported and analyzed in detail and chapter 6 in which the conclusions of the work and the obtained results are analyzed and the possible improvements that can be performed on the flow are made.

Anyway, it is important to distinguish the chapter 3 to the chapter 4 that could seem similar at the first impact. The chapter 3 aims to describe the created flow from the point of view of the user but not from a point of view of the designer. So, it does not describe in detail how the environment has been built while in the

chapter 4 a deep analysis of the environment and its step by step development are described.

Chapter 2

Basics and backgrounds

In the new semiconductor technologies the operating frequency gets higher and higher and the process variations affects increasingly these new technologies (Process variation happens when processes fail to follow a precise pattern) [5]. Hence, defects affecting the timing characteristics are more common and detecting these defects requires performing the test at speed . In fact, increasing the maximum delay of a combinational block in a sequential circuit may cause sampling wrong values, for example, in some memory elements [6]. This issue depends on the clock value that is set by the designer to a value conform to the length of the critical path (CP). The delay faults can affect the CP causing, thus, a wrong sampling on some signals of the processor. The delay faults are divided in two main models: Path Delay Faults (PDFs) and Transition Delay Faults (TDFs).

In these studies, as just said in the introduction, the behaviour of the PULPINO processor when *transition delay faults* (TDFs) are injected during the simulation, using *Modelsim* [2], is analyzed. So, this chapter aims to describe TDFs and their properties. Moreover, the transition delay fault model is chosen over the path delay one because TDFs are much better supported by both standards and Electronic Design Automation (EDA) [7] tools than PDFs [1].

2.1 Transition delay faults

Transition delay fault model is a special case of delay faults and it is one of the most widely used delay model in the common applications. Its concept is based on modelling the defect as a delay of a gate to pass from the value 0 to the value 1, that is called slow-to-rise (STR) or viceversa, slow-to-fall (STF). The number of faults in a circuit can be summarized by the formula $2(n + i)$ where n is the number of gates and i is the number of inputs [6]. In order to detect a TDF pairs of test vectors are needed, to generate a specific transition (e.g. 0010 - 0000 that is clearly a STF), and then propagate them on the output signals to observe a possible different behaviour on the circuit.

An example of transition delay fault behaviour on a simple circuit is shown in *Figure 2.1*.

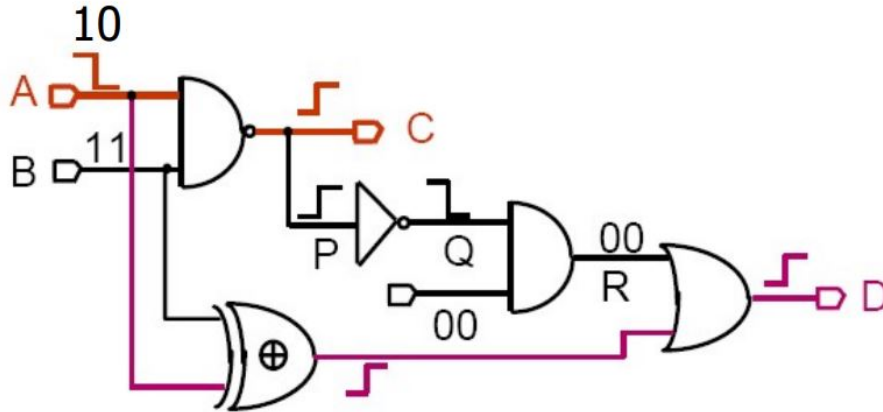


Figure 2.1: Example of transition model [1]

As can be seen, in this case, the used vector pair is 11 - 01. So, a STF is injected on the input A and if it is faulty the output C or D can detect the transition. The transition fault should be detected by the path that has D as output because it is the CP.

In the FI flow explained in the next chapter the fault is injected in this way: each

transition of the analyzed signal is delayed of a quantity equal to $T_{clk} + 10\%T_{clk}$. In this case, the clock period is 40 ns, so the adopted delay during the fault injection (FI) flow is 44 ns.

An important thing to consider that will be a problem during the development of the environment for the application of the flow is the fan-out. An example is shown in *Figure 2.2*.

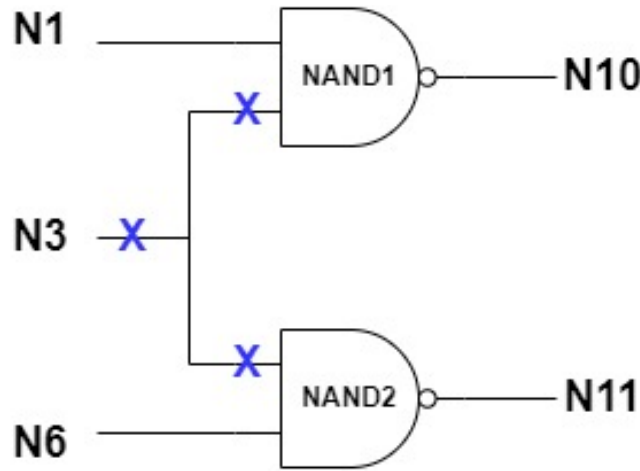


Figure 2.2: circuit with fan-out

It is important remembering that *Faults on a fanout stem are not equivalent to faults on the corresponding fan-out branches* [8]. These are the basic theoretical backgrounds related to this topic needed to understand this manuscript. Obviously, other technical basics are necessary to exploit and can understand the flow and its environment as, for example, *Verilog*, *SistemVerilog* and some scripting languages.

Chapter 3

User Manual of the environment

This chapter aims at better explaining how the PULPINO environment works and in deeply analyzing the files and folders inside it. This environment exploits the implemented flow shown in *Figure 3.1*.

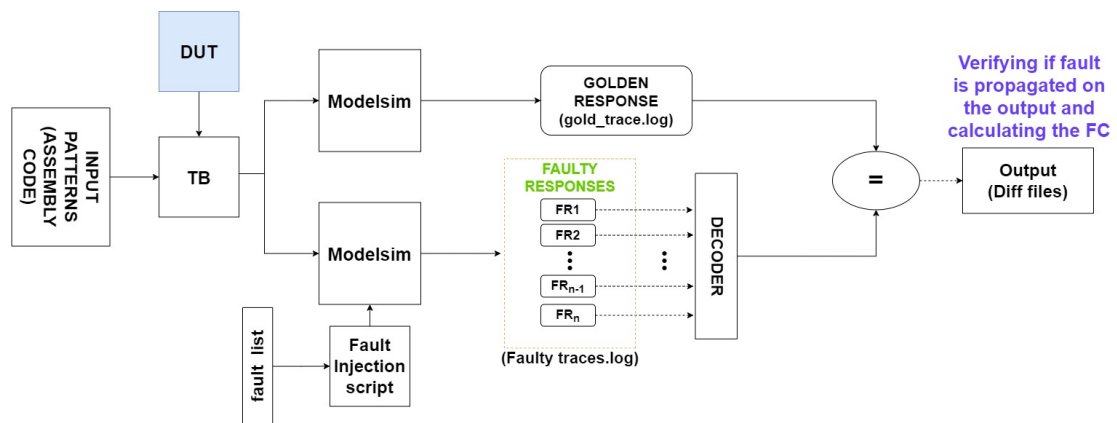


Figure 3.1: Fault injection flow

3.1 Flow description

The purpose of this flow is to understand if a difference exists between the golden simulation and the faulty simulation, for each fault injection (FI) performed, helping the user or, specifically, a testing engineer in his work. An assembly code is used to test the processor thanks to a given testbench. So, thanks to this sequence of operations, a given fault can be *observed* or not.

The golden simulation is represented by the upper branch of *Figure 3.1* and it simply tests the processor without any injections thanks to the assembly code while the faulty simulation is represented by the bottom branch. The latter is, basically, a loop in which the FI execution is performed for each fault in the fault list. At the end of the two branches, for each cycle, the output file of the golden simulation and the one related to the faulty simulation are compared.

These two files are the output of the tracer processor.

Trace is a hardware debug feature that allows the run-time behavior of IP to be monitored [9]. In simple words, it writes on the output file the original assembly code from the signals of the processor in order to verify, comparing the source assembly code and the output file of the tracer, the correctness of the processor's behaviour.

In this case, if the golden output and the faulty output are different it means that the processor is affected by a fault.

3.2 Environment structure

The internal structure of the following environment is shown in *Figure 3.2*.

The most important files and folders are:

- **asic**: a folder in which there are three main files to this purpose. They are:
 - **synopsys/out/final_riscv_core.v**: it is the *Verilog* netlist that describes the processor at gate level. It has been modified in order to integrate the tracer above only set inside the RTL level processor (usefulness and integration of the tracer will be better explained in the next

```

asic
ci
clean_all.sh
compile_pulpino.sh
compile_selftest.sh
doc
fault_analysis.sh
files
fpga
gate
ips
ipstools
LICENSE
NOTE
py_files
README_gianmarco.md
README.md
rtl
run_fsim.sh
run_selftest_sim_gui.sh
run_selftest_sim_nogui.sh
std_out.txt
sw
tb
tmax
transcript
vsim

```

Figure 3.2: internal structure of the PULPINO environment

chapter).

- `techlib/NangateOpenCellLibrary.v`, the version used in the flow, and the original version `techlib/NangateOpenCellLibrary_old.v`: they are both technology libraries including all the primary logic gates; the only difference between them is that the first one fixes the fan-out problem while the second one not (this last concept will be better explained in the next chapter).
- `clean_all.sh`: a *Bash* script used to clean the environment from all the build files of compilations and simulations.
- `compile_pulpino.sh`: a *Bash* script used to compile all the *Verilog* and *systemverilog* files necessary in order to build the PULPINO processor.
- `compile_selftest.sh`: a *Bash* script used to compile the assembly code to load into the processor.

- **fault_analysis.sh**: a *Bash* script that represents the flow conceptually described by the *Figure 3.1*.
- **files**: a folder in which three *Text* files are included. The most important one is **all_signals.txt**, because it contains all the faults to analyze, basically the fault list. In particular the type of the transition fault (STF or STR) and the name of the signal are specified in each row. The other two files are support files used by a *Python* script.
- **ips** and **RTL**: two folders which are composed by all the *SystemVerilog* files involved in the structural processor.
- **py_files**: a folder in which four *Python* script files have to be deeper analyzed. They are:
 - **modificaFile_original_library.py**: Its task is to modify the script *run.tcl*, in the path *vsim/tcl_files/*, in order to perform a correct fault injection (FI) on original library (old one) for each transition delay fault. It is invoked for each fault, as said before, in the master script (*fault_analysis.sh*) with four parameters: name of the FI program to be modified, name of the new FI program, type of transition fault, the name of the signal to test.
 - **mod_library.py**: it modifies the tech library in order to fix the fan-out problem. In this environment it has been just invoked once, so the library now is completed. In the case in which the tech library is changed, this script has to be run calling the new library, in *asic/techlib/* path, *NangateOpenCellLibrary_old.v* and checking if the script is compatible with the library. The idea of fixing the fan-out problem will be explained in the next chapter.
 - **modificaFile_modified_library.py**: it has the same function of *modificaFile_original_library.py* with the same parameters, but adapted and applied to the modified library.
 - **comment.py**: This script allows to comment or not the FI script on *run.tcl*. The parameters are the name of the FI program to be modified, the name

of the new FI program and y/n that specify if the FI has to be commented or not. It is used at the start of *fault_analysis.sh* script to obtain the golden response at the first simulation and all the faulty responses at the subsequent simulations.

- **post_processing.py**: It is a script that aims to recap the results obtain in the reports folder, analyzing if the faults are observed and their typology. Moreover, It provides some useful statistics. Only one parameter is passed to this script and it is the path related to the folder inside **reports** (e.g. *python py_files/post_processing.py ../reports/tracer_start_code/*) that has to be analyzed.
- **run_selftest_sim_gui.sh**: this script handles the simulation that is performed by *Modelsim* [2] with graphical interface.
- **run_selftest_sim_nogui.sh**: this script handles the simulation that is performed by *Modelsim* [2] without graphical interface.
- **sw**: two folders are really important here. They are:
 - **apps/riscv_tests/polito** in which the main assembly files are located. In particular **tests/simple.S** is very important because changing it we can test in different ways the processor.
 - **build/apps/riscv_tests/polito** in which all the simulation files are generated. In particular *trace_core_00_0_gold.log*, that is generated during the golden simulation (without FI), and *trace_core_00_0.log*, that is generated every time a new FI is performed on one of the faults within the fault list, are mainly important. They are both generated thanks to the tracer put inside the netlist.
- **tb**: The testbench files are included in this folder. The master one is **tb.sv** in which the processor is instantiated.
- **vsim**: It is composed of two folders that are:

- `tcl_files`: it contains the simulation scripts written in *tcl*. The most important ones are `run.tcl` and `config/vsim.tcl`. The former chooses the waveforms, applies the FI and set some parameters while the latter is called by *run.tcl* and it's important because here the most appropriate simulation parameters can be set by the user (e.g. the level of the optimization).
- `vcompile`: it contains the compilation scripts.

Outside the environment there also are three more important elements to be considered:

- `reports` that is a folder in which all the results of the fault simulations are stored. They are the results got by the *diff* bash command between golden output file and faulty output file of the simulation of the processor.
- `create_speed_environment.sh` that is a *Bash* script whose task is to replicate the original environment for n° times (this number can be modified) in order to speed up the fault simulation exploiting it.
- `fault_analysis_speed.sh` that is the master script and it simply runs the environment created by `create_speed_environment.sh`.

3.3 How the environment works and how to use it

In order to fully understand how to use this environment and how it works, it is crucial to follow some essential points taken step by step. They are explained in the next subsections.

3.3.1 Importing the environment into a new computer file system and making it workable

When the environment is extracted from *.zip*, several steps have to be performed in order to make it working.

First of all, it is essential to ensure that the specific toolchain, the one related to the testing processor in order to be able to perform the compilation, is installed and its path is correctly exported. However, it could be really useful the command "*which COMPILER_NAME*" to check if the file exists; If all is correctly set the compilation of the *SystemVerilog* files should run without errors.

The next step is the compilation of the assembly test program. Before running `compile_selftest.sh` it is important to modify the file `sw/ref/link.riscv.ld` including the correct path, otherwise the error shown in *Figure 3.3* will appear during the compilation.



```

Linking CXX executable rv_polito.elf
/opt/riscy_gnu_toolchain/bin/../lib/gcc/riscv32-unknown-elf/5.2.0/../../../../riscv32-unknown-elf/bin/ld: cannot
open linker script file /auto/gmongelli/pulpino/sw/ref/link.common.ld: No such file or directory
collect2: error: ld returned 1 exit status
make[3]: *** [apps/riscv_tests/polito/rv_polito.elf] Error 1
make[2]: *** [apps/riscv_tests/polito/CMakeFiles/rv_polito.elf.dir/all] Error 2
make[1]: *** [apps/riscv_tests/polito/CMakeFiles/rv_polito.dir/rule] Error 2
make: *** [rv_polito] Error 2
  
```

Figure 3.3: error during the selftest compilation

Additionally, before running the simulation, it is extremely important that no optimization is performed in the design. In order to ensure that, modifying, adding or removing some parameters inside the file `vsim/tcl_files/config/vsim.tcl` could be necessary. At this point the simulation can be performed and it should work well. These ones are the basic steps that should fix all the problems in order to exploit the environment, but it is important to know that other errors may occur caused by e.g. a different version of *Modelsim* or *Python*, etc...

3.3.2 Running the flow

Before running the flow, creating `report` (outside `pulpino`) and `tracer_analysis` (inside `report`) folders are essential, considering that the path used in the master script for storing the results is `reports/tracer_analysis`. Instead, once the all flow is completed a new `tracer_analysis` folder has to be created while the one just used has to be modified. This last step is obligatory between two different executions of the flow.

The master script that runs the flow, in *Figure 3.1*, is `fault_analysis.sh` (*Bash*

script) and its code is the one shown in *Listing 3.1*:

Listing 3.1: Master script describing the flow

```
1 ./clean_all.sh
2 ./compile_pulpino.sh
3 ./compile_selftest.sh
4
5 #golden simulation
6
7 echo "Golden simulation analysis"
8 python ./py_files/comment.py ./vsim/tcl_files/run.tcl ./vsim/
   tcl_files/trashed.tcl y
9 ./run_selftest_sim_nogui.sh
10 mv ./sw/build/apps/riscv_tests/polito/trace_core_00_0.log ./sw/build/
   apps/riscv_tests/polito/trace_core_00_0_gold.log
11 cp ./sw/build/apps/riscv_tests/polito/trace_core_00_0_gold.log ../
   reports/tracer_analysis/
12 python ./py_files/comment.py ./vsim/tcl_files/run.tcl ./vsim/
   tcl_files/trashed.tcl n
13
14 #Fault simulations
15 while IFS= read -r line; do
16     python ./py_files/modificaFile_modified_library.py ./vsim/
   tcl_files/run.tcl ./vsim/tcl_files/trashed.tcl $line;
17     rm ./vsim/tcl_files/trashed.tcl;
18     var=$(cat files/signal.txt)
19     if test -f "../reports/tracer_analysis/$var"; then
20         echo "$var exist"
21     else
22         echo "$var analysis"
23         ./run_selftest_sim_nogui.sh;
24         diff ./sw/build/apps/riscv_tests/polito/trace_core_00_0_gold.
   log ./sw/build/apps/riscv_tests/polito/trace_core_00_0.log >> ../
   reports/tracer_analysis/${var}
25     fi
26 done < ./files/all_signals.txt
```

This code can be differentiated in three main parts:

- The first part aims to *clean* all the files of the previous fault simulation (It is done in order to speed-up the simulation, it will be better explained later in this chapter) and to run again the *compilation* scripts. This work is performed by `clean_all.sh`, `compile_pulpino.sh` and `compile_selftest.sh`.
- As we can see in *Figure 3.1*, it is really important to compare the golden simulation with all the faulty simulations for this purpose. The second part of the code aims to generate the golden output file to be compared with the faulty output file for each signal inside the fault list (the output files are generated by the tracer as just said).

In order to generate it, `comment.py` script is really significant because it allows to comment(y option) or not(n option) the FI part of the simulation script(`run.tcl`). In this case, the FI part is commented and a simulation is run. Once the simulation ends and the output file is gotten, thanks to tracer, the FI part can be written again in `run.tcl`. To be noted that the file `trace_core_00_0_gold.log` is copied into the report folder, outside the pulpino environment.

- The last part of the code has the purpose to perform all faulty simulations in order to obtain the outputs to compare with golden simulation.

The fault list (`all_signals.txt`) is read line by line, where the line is composed by the signal to analyze and the kind of transition fault. So, the subsequent python script (`modificaFile_modified_library.py`) is called in order to modify the FI script, by replacing the old code with the new one related to the current signal of the line. During the execution of the flow an expected or an unexpected interruption could happen. In this situation, some faulty simulations have already been performed while other not. In order to avoid to reanalyze the same faults just simulated, a check is performed in the code by storing, cycle by cycle, the current output file to be generated in a variable `var` (`STF_RISCV_CORE_ex_stage_i_alu_i_int_div_div_i_U457_B1.txt` is an example of output report). If the output file of this fault exists in the report folder the simulation is skipped otherwise it is done and the comparison between golden and faulty report is performed.

After that the cycle ends and starts again. *The loop goes on until the all fault list is completely analyzed.* The most important thing the user can exploit in this script is the file representing the fault list (`files/all_signals.txt`). Thanks to it, different set of faults can be analyzed simply modifying the fault list file.

At the end of the flow execution, the `post_processing.py` can be run, with its parameter (e.g. `../reports/tracer_start_code/`), in order to summary all the information of the reports in a single file called `post_processing.csv` inside the folder `reports/*folder to analyze*/`.

3.3.3 Interpretation of reports

Inside `reports`, different folders obtained by several flow executions are located. Inside these files there are all the *text* files related at each fault in the fault list. An example is `STF_RISCV_CORE_ex_stage_i_alu_i_alu_ff_i_U38_A` where *STF* is the type of the transition fault while the rest of the file specifies the fault analyzed. These files derive from the *diff* command between the golden output, generated by the tracer during the golden simulation, and the faulty output, generated by the tracer during the faulty simulations. Let's analyze the structure of the output tracer shown in *Listing 3.2*, taking a look at a little part of its content, and then let's analyze the output of the *diff* command shown in *Listing 3.3*:

Listing 3.2: `trace_core_00_0_gold.txt` generated during the golden simulation

	Time	Cycles	PC	Instr	Mnemonic
1					
2					
3	19520000	471	000068f4	00008713	addi
	x14, x1, 0	x14=00000000	x1:00000000		
4	19560000	472	000068f8	00008793	addi
	x15, x1, 0	x15=00000000	x1:00000000		
5	19600000	473	000068fc	00008813	addi
	x16, x1, 0	x16=00000000	x1:00000000		
6	19640000	474	00006900	00008893	addi
	x17, x1, 0	x17=00000000	x1:00000000		
7	19680000	475	00006904	00008913	addi
	x18, x1, 0	x18=00000000	x1:00000000		

8	20000000	483 00006924 00008d13 addi
	x26, x1, 0	x26=00000000 x1:00000000
9	20040000	484 00006928 00008d93 addi
	x27, x1, 0	x27=00000000 x1:00000000
10	20080000	485 0000692c 00008e13 addi
	x28, x1, 0	x28=00000000 x1:00000000
11	20120000	486 00006930 00008e93 addi
	x29, x1, 0	x29=00000000 x1:00000000
12	20160000	487 00006934 00008f13 addi
	x30, x1, 0	x30=00000000 x1:00000000
13	20200000	488 00006938 00008f93 addi
	x31, x1, 0	x31=00000000 x1:00000000
14	20240000	489 0000693c 00101117 auipc
	x2, 0x101000	x2=0010793c
15	20280000	490 00006940 6c410113 addi
	x2, x2, 1732	x2=00108000 x2:0010793c

As we can see, the different columns are: Time, Cycles, PC, Instr, Mnemonic. Then the assembly instruction and the registers contents are specified.

Listing 3.3: example of a report file generated by *diff*

1	2237c2237	
2	<	294440000 7344 00001698 10098fb3 p.ff1
	x31, x19	x31=00000005 x19:1a8fbb20
3	—	
4	>	294440000 7344 00001698 10098fb3 p.ff1
	x31, x19	x31=00000004 x19:1a8fbb20

The row indicated by < is the one related the output golden tracer while > represents the faulty one. Obviously if these lines are written inside the report file it means that there is some difference between faulty and golden simulations for the targeted signal.

The last file to evaluate is `post_processing.csv` and an example of its content is reported in *Listing 3.4*:

Listing 3.4: an extraction of `post_processing` report

1	FAULT,OBSERVED,TYPE
2	.

```

3      .
4      .
5 STR_RISCV_CORE_ex_stage_i_alu_i_U1709_A1,yes,data register error
6 STF_RISCV_CORE_ex_stage_i_alu_i_U1835_A1,yes,data register error
7 STF_RISCV_CORE_ex_stage_i_alu_i_int_div_div_i_U333_ZN,yes,data
  register error
8 STR_RISCV_CORE_ex_stage_i_alu_i_int_div_div_i_U335_B1,no,none
9 STR_RISCV_CORE_ex_stage_i_alu_i_int_div_div_i_U339_B,no,none
10 STF_RISCV_CORE_ex_stage_i_alu_i_int_div_div_i_U338_B1,no,none
11      .
12      .
13      .
14 TOT N FAULTS,N FAULTS OBSERVED,% OBSERVED
15 696,93,13.36

```

The first part of this file is related to report some information fault by fault while the last part provides some statistics.

3.3.4 How to speed-up the execution time of the master script

The problem of the master script is that it could take a long time if the fault list is too big. For example, in our case the fault list has more or less 700 signals and the execution flow takes almost 3 days. In order to fix this problem, two scripts have to be used in sequence:

1. `create_speed_env.sh` in which, first of all, the fault list is split in ten smaller files with the same number of faults between them with the only exception of the last file that could have less faults. This number (n) has to be modified by the user if the fault list changed. Considering $t = \text{total number of faults in the fault list}$ and $n = \text{total number of faults in the smaller fault list}$, the calculation of n is

$$\lceil n = \frac{t}{10} \rceil$$

For example, in our case $t = 696$ so $n = 70$. The code of `create_speed_env.sh` is shown in *Listing 3.5*:

Listing 3.5: script for creating the speed environment

```
1 mkdir all_signals_division
2 split -l 70 ./pulpino/files/all_signals.txt all_signals_division/
  all_signals
3
4 for i in {0..9}
5 do
6     [ -e .pulpino${i} ] && rm -rf .pulpino${i}
7     cp -r pulpino .pulpino${i}
8 done
9
10 mv all_signals_division/all_signalsaa .pulpino0/files/all_signals
  .txt
11 mv all_signals_division/all_signalsab .pulpino1/files/all_signals
  .txt
12 mv all_signals_division/all_signalsac .pulpino2/files/all_signals
  .txt
13 mv all_signals_division/all_signalsad .pulpino3/files/all_signals
  .txt
14 mv all_signals_division/all_signalsae .pulpino4/files/all_signals
  .txt
15 mv all_signals_division/all_signalsaf .pulpino5/files/all_signals
  .txt
16 mv all_signals_division/all_signalsag .pulpino6/files/all_signals
  .txt
17 mv all_signals_division/all_signalsah .pulpino7/files/all_signals
  .txt
18 mv all_signals_division/all_signalsai .pulpino8/files/all_signals
  .txt
19 mv all_signals_division/all_signalsaj .pulpino9/files/all_signals
  .txt
20
21 rm -r all_signals_division
```

Thus, the number 70 in second row has to be modified with what is gotten by the previous formula if a new fault list has been created.

The second part of the code aims to create 10 temporary environments that

will exploit the 10 smaller fault lists just created.

2. `fault_analysis_speed.sh`. Its code is shown in *Listing 3.6*:

Listing 3.6: script for running the speed environment

```

1 for i in {0..9}
2 do
3     cd .pulpino${i}
4     ./fault_analysis.sh &> std_out.txt &
5     sleep 7m
6     cd ..
7 done

```

In our example ($t = 696$) the obtained execution time is around 12 hours. If a faster execution time is needed some further modifications have to be performed. If we consider a new parameter $c = \text{number of copies of the environment}$, the new resulting n is

$$\lceil n = \frac{t}{c} \rceil$$

and the two script must be modified in the following way:

- In the second line of `create_speed_env.sh` n obtained by the calculation should replace 70.
- In the fourth line of `create_speed_env.sh` 9 should be replaced with c .
- In `create_speed_env.sh`, the line `mv all_signals_division/all_signalsaX.pulpinoN/files/all_signals.txt` has to be added until this line is repeated c times, considering that X and N are replaced in the proper way. Another idea could be implementing a loop.
- In the first line of `fault_analysis_speed.sh` 9 should be replaced with c .

3.3.5 The simulation script

Another part of the flow that can be managed by the user is `run.tcl`. Waveforms can be added or removed, several parameters can be set, the FI injection script

can be modified with the subsequent adjustment in the python scripts involved in FI, etc...but one of the most important things for the proper functioning of the flow is to take care of this part of the code of `run.tcl` shown in *Listing 3.7*:

Listing 3.7: make sure simulation stops in case of infinite loops

```
1 when -fast {$now = @1 ms} {  
2     quit -f  
3 }
```

The number "1 ms" is necessary in order to prevent the infinite execution in case of loop simulation and it has to be greater than the effective execution time. So, if the execution time is equal to 2 ms the number to insert in the code, instead of 1 ms, should be > 2 ms.

3.4 Conclusion

The purpose of this flow is that the user, usually a testing engineer, could have more details on the faults and an help in writing the code. For example, it can be noticed a difference in a specific bit for a line code between the golden and a faulty simulation or more instruction in a specific part of the code respect that what we expected and so on... However, this flow can be further improved generating more statistics based on the reports.

Chapter 4

Chronological steps to build the environment

At the beginning of this thesis study a simple PULPINO processor environment was provided. This chapter aims at describing all the basic steps performed to get the environment able to implement the flow illustrated in *Figure 4.1*. This process is carefully described in chronological order in the following sections.

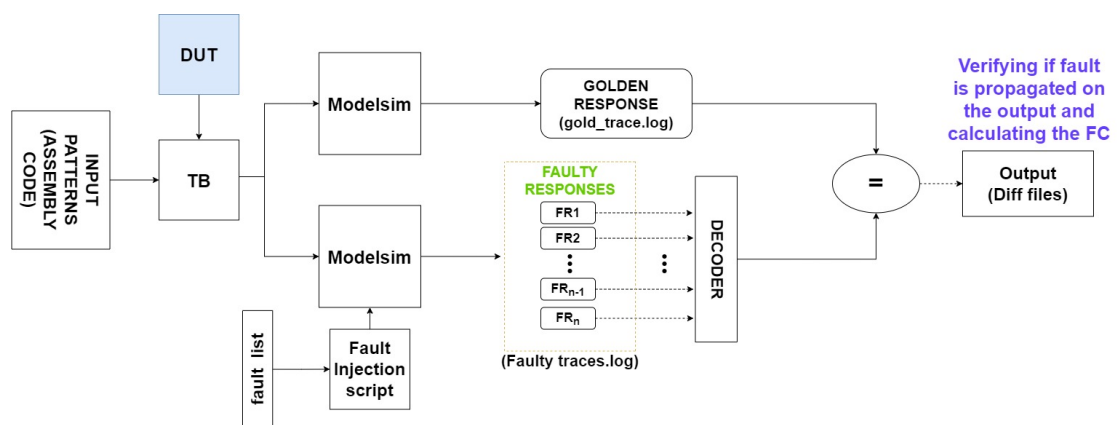


Figure 4.1: Fault injection flow

4.1 Initial setting problems to solve

At the beginning of a deeply analysis of the environment, during first compilations and simulations, some problems raised:

1. The *toolchain* of the compiler was not initially installed, so some errors stopped the process. The solution was simply to download and install it exporting its correct path.
2. In order to perform a correct compilation and simulation, other little modifications, related to some paths not correctly set, had to be done like writing the correct path inside `sw/ref/link.riscv.ld` as just described in the user manual chapter.
3. The initial folder was set to compile and simulate the RTL processor and not the *gate* level one. The problem was that the test process should analyze the gate level processor like in the tool used before Z01X [3], so by trying to inject faults on the signals related to the netlist with all the logic functions created after the synthesis. The solution was simply to modify the compiler script in order to compile the correct *Verilog* files.

At this point, the environment was ready to be modified in order to provide an helpful instrument to the testing engineer.

4.2 The tracer: comparison mechanism of the flow

After all parameters, scripts, etc... of the environment were set in the correct way, the problem to fix was finding a way to compare the faulty simulation with the golden one. The general idea was comparing two output simulation files somehow generated.

In reality, in the initial version of the environment, there was already a way to generate an output file from the simulation: the tracer, *a module able to map the execution time to the instructions currently executed by the processor core* [1]. So,

it is a module that can *reconstruct* the original assembly code from the signals of the processor and It can be used to generate the assembly code of the golden simulation and to compare it with each assembly code generated by the faulty simulations. In order to exploit it, two main problems should be fixed:

1. The tracer was included in the RISC_V_CORE structural but not in the RISC_V_CORE gate. Its instantiation in the structural processor is shown in *Listing 4.1*.

Listing 4.1: instantiation of the tracer in the structural processor

```

1  'ifndef VERILATOR
2  'ifdef TRACE_EXECUTION
3      riscv_tracer riscv_tracer_i
4      (
5          .clk          ( clk_i
6              always-running clock for tracing
7              .rst_n      ( rst_ni
8              .fetch_enable ( fetch_enable_i
9              .core_id      ( core_id_i
10             .cluster_id   ( cluster_id_i
11
12             .pc            ( id_stage_i.pc_id_i
13             .instr         ( id_stage_i.instr
14             .compressed    ( id_stage_i.is_compressed_i
15             .id_valid      ( id_stage_i.id_valid_o
16             .is_decoding   ( id_stage_i.is_decoding_o
17             .pipe_flush    ( id_stage_i.controller_i.pipe_flush_i ),
18             .mret          ( id_stage_i.controller_i.mret_insn_i ),
19             .uret          ( id_stage_i.controller_i.uret_insn_i ),
20             .ecall         ( id_stage_i.controller_i.ecall_insn_i ),
21             .ebreak        ( id_stage_i.controller_i.ebrk_insn_i ),
22             .rs1_value     ( id_stage_i.operand_a_fw_id
23             .rs2_value     ( id_stage_i.operand_b_fw_id
24             .rs3_value     ( id_stage_i.alu_operand_c
25             .rs2_value_vec  ( id_stage_i.alu_operand_b
26
27             .rs1_is_fp      ( id_stage_i.regfile_fp_a

```

```

28     .rs2_is_fp      ( id_stage_i.regfile_fp_b      ),
29     .rs3_is_fp      ( id_stage_i.regfile_fp_c      ),
30     .rd_is_fp        ( id_stage_i.regfile_fp_d      ),
31
32     .ex_valid        ( ex_valid                      ),
33     .ex_reg_addr     ( regfile_alu_waddr_fw         ),
34     .ex_reg_we        ( regfile_alu_we_fw           ),
35     .ex_reg_wdata     ( regfile_alu_wdata_fw        ),
36
37     .ex_data_addr    ( data_addr_o                  ),
38     .ex_data_req      ( data_req_o                  ),
39     .ex_data_gnt      ( data_gnt_i                  ),
40     .ex_data_we        ( data_we_o                  ),
41     .ex_data_wdata    ( data_wdata_o                ),
42
43     .wb_bypass        ( ex_stage_i.branch_in_ex_i    ),
44
45     .wb_valid         ( wb_valid                    ),
46     .wb_reg_addr     ( regfile_waddr_fw_wb_o        ),
47     .wb_reg_we        ( regfile_we_wb               ),
48     .wb_reg_wdata     ( regfile_wdata               ),
49
50     .imm_u_type       ( id_stage_i.imm_u_type        ),
51     .imm_uj_type      ( id_stage_i.imm_uj_type       ),
52     .imm_i_type       ( id_stage_i.imm_i_type        ),
53     .imm_iz_type      ( id_stage_i.imm_iz_type[11:0] ),
54     .imm_z_type       ( id_stage_i.imm_z_type        ),
55     .imm_s_type       ( id_stage_i.imm_s_type        ),
56     .imm_sb_type      ( id_stage_i.imm_sb_type       ),
57     .imm_s2_type      ( id_stage_i.imm_s2_type       ),
58     .imm_s3_type      ( id_stage_i.imm_s3_type       ),
59     .imm_vs_type      ( id_stage_i.imm_vs_type       ),
60     .imm_vu_type      ( id_stage_i.imm_vu_type       ),
61     .imm_shuffle_type ( id_stage_i.imm_shuffle_type  ),
62     .imm_clip_type    ( id_stage_i.instr_rdata_i[11:7] ),
63 );
64 `endif

```

2. In order to instantiate the tracer in the gate level processor, some specific signals of the netlist (corresponding on the ones of the structural processor connected to the tracer) should be connected to the port map of the tracer, but finding them in the netlist was really hard because *Synopsys* changed the name of a lot of signal in the new *Verilog* description.

4.2.1 Tracer instantiation in structural processor

In *Listing 4.1* all signals connected to port map at the structural level are shown. Different kind of signal categories can be distinguished:

- *Input/Output of RISC_V_CORE:*
 clk_i, rst_ni, fetch_enable_i, core_id_i, cluster_id_i, data_addr_o,
 data_req_o, data_gnt_i, data_we_o, data_wdata_o;
- *Signals of RISC_V_CORE:*
 ex_valid, regfile_alu_waddr_fw, regfile_alu_we_fw, regfile_alu_wdata_fw,
 wb_valid, regfile_waddr_fw_wb_o, regfile_we_wb, regfile_wdata;
- *Input/Output of ID_STAGE:*
 id_stage_i.pc_id_i, id_stage_i.is_compressed_i, id_stage_i.id_valid_o,
 id_stage_i.is_decoding_o, id_stage_i.instr_rdata_i;
- *Signals of ID_STAGE:*
 id_stage_i.instr, id_stage_i.operand_a_fw_id,
 id_stage_i.operand_b_fw_id, id_stage_i.alu_operand_c,
 id_stage_i.alu_operand_b, id_stage_i.regfile_fp_a,
 id_stage_i.regfile_fp_b, id_stage_i.regfile_fp_c, id_stage_i.regfile_fp_d
 and all instructions with "id_stage_i.imm";
- *Input/Output of EX_STAGE:*
 ex_stage_i.branch_in_ex_i;
- *Input/Output of CONTROLLER:*
 id_stage_i.controller_i.pipe_flush_i, id_stage_i.controller_i.mret_insn_i,

```
id_stage_i.controller_i.uret_insn_i, id_stage_i.controller_i.ecall_insn_i,
id_stage_i.controller_i.ebrk_insn_i;
```

4.2.2 Tracer instantiation in gate level processor

As just said, the most important issue in instantiating the tracer in the gate level processor was that the synthesizer changed some signal' names randomly when it created the *Verilog* netlist and, sometimes, it was really hard to find the corresponding signals to connect to the port map. In order to fix this problem, the idea was to divide the signals of port map in different categories and for each of them, to exploit a different method to find correct signals to connect to the port map. After doing it, the tracer was connected to the gate level processor as shown in *Listing 4.2*.

Listing 4.2: instantiation of the tracer in the gate level processor

```
1 riscv_tracer riscv_tracer_i
2 (
3   .clk          ( clk_i                      ), // i/o!
4   always-running clock for tracing
5   .rst_n        ( rst_ni                     ), // i/o!
6   .fetch_enable ( fetch_enable_i             ), // i/o!
7   .core_id      ( core_id_i                  ), // i/o!
8   .cluster_id   ( cluster_id_i               ), // i/o!
9
10  .pc            ( pc_tracer                  ),
11  .instr         ( instr                      ),
12  .compressed    ( is_compressed_id           ),
13  .id_valid      ( id_valid                   ),
14  .is_decoding   ( is_decoding                ),
15  .pipe_flush    ( id_stage_i_pipe_flush_dec  ),
16  .mret          ( id_stage_i_mret_insn_dec   ),
17  .uret          ( 1'b0                      ),
18  .ecall         ( id_stage_i_ecall_insn_dec  ),
19  .ebreak        ( id_stage_i_ebrk_insn      ),
20  .rs1_value     ( operand_a_fw_id            ),
```



```

21      .rs2_value      ( operand_b_fw_id                      ),
22      .rs3_value      ( alu_operand_c                       ),
23      .rs2_value_vec   ( alu_operand_b                      ),
24
25      .rs1_is_fp       ( 1'b0                               ),
26      .rs2_is_fp       ( 1'b0                               ),
27      .rs3_is_fp       ( 1'b0                               ),
28      .rd_is_fp        ( 1'b0                               ),
29
30      .ex_valid         ( id_stage_i_ex_valid_i              ),
31      .ex_reg_addr      ( {1'b0, regfile_alu_waddr_ex}       ), //
regfile_alu_waddr_ex è di 5 bit e l'MSB è sempre 0, regfile porta
b
32      .ex_reg_we        ( regfile_alu_we_ex                  ),
33      .ex_reg_wdata     ( regfile_alu_wdata_fw               ),
34
35      .ex_data_addr     ( data_addr_o                         ), // i/o!
36      .ex_data_req      ( data_req_o                         ), // i/o!
37      .ex_data_gnt      ( data_gnt_i                        ), // i/o!
38      .ex_data_we       ( data_we_o                         ), // i/o!
39      .ex_data_wdata    ( data_wdata_o                      ), // i/o!
40
41      .wb_bypass        ( branch_in_ex                       ),
42
43      .wb_valid         ( id_stage_i_wb_ready_i              ),
44      .wb_reg_addr      ( {1'b0, ex_stage_i_regfile_waddr_lsu} ), //
cercare su register file porta addr a
45      .wb_reg_we        ( ex_stage_i_regfile_we_lsu          ),
46      .wb_reg_wdata     ( ex_stage_i_lsu_rdata_i             ),
47
48      .imm_u_type       ( imm_u_type                         ), // da
qui in poi conseguenza di instr
49      .imm_uj_type      ( imm_uj_type                       ),
50      .imm_i_type       ( imm_i_type                        ),
51      .imm_iz_type      ( imm_iz_type[11:0]                  ),
52      .imm_z_type       ( imm_z_type                        ),
53      .imm_s_type       ( imm_s_type                        ),
54      .imm_sb_type      ( imm_sb_type                       ),

```

```

55 .imm_s2_type      ( imm_s2_type      ) ,
56 .imm_s3_type      ( imm_s3_type      ) ,
57 .imm_vs_type      ( imm_vs_type      ) ,
58 .imm_vu_type      ( imm_vu_type      ) ,
59 .imm_shuffle_type ( imm_shuffle_type ) ,
60 .imm_clip_type    ( instr_rdata_id [11:7] )
61 );

```

Now let's analyze each category and their method to select the right signals to connect to the port map (In this analysis it is explained how to derive the corresponding signal of the processor at gate level from the one of the processor at structural level):

- *Input/Output of RISC_V_CORE*: It is the simplest case. In fact, considering RISC_V_CORE as a black box, they are simply inputs or outputs of the processor. So, the synthesizer doesn't affect the name of these signals.
- *Signals of RISC_V_CORE*: for these signals it could be necessary to investigate a little bit more. For example, there are different cases and the most important ones are:
 - *ex_valid*. If it is searched inside the structural_processor it can be seen that it is connected to a port of ID_stage called *ex_valid_i*. So, if the name of this port in the gate_processor is searched, the corresponding signal can be found (*id_stage_i_ex_valid_i*). This algorithm is repeated for others signals like *regfile_alu_we_fw*, etc...
 - *regfile_waddr_fw_wb_o*. Same algorithm of before is used in ID_STAGE but a deep research is necessary in order to find how this signal is connected to a *waddr_a_i*. Sometimes synthesizer removes bits from some signals like in this case so, it is necessary find the missing bit that could be or a new signal or a '0'.
 - *regfile_alu_wdata_fw* in which the most important problem of instantiating tracer inside the gate level is faced. The original signal name connected to the port map of structural_RISC is translated in some nets

by the synthesizer as shown in *Listing 4.3*. In these cases finding correct signals to port is not easy. This action is performed by the processor a few times, but there could be other situations in which the translation nets is more frequent.

Listing 4.3: several nets associated to a signal

```

1  assign regfile_alu_wdata_fw = { n2640 , n2941 , ! n3452 , ! n3445 , !
n3438 , n3287 , n3272 , n3265 , n2859 , ! n3403 , ! n3396 , ! n3389 , ! n3382 , !
n3373 , n3218 , n3213 , n3206 , n3199 , n2548 , n2546 , n3178 , n2540 , n2465 ,
n2535 , n2463 , n2530 , n2462 , n2525 , n2523 , n2521 , n2519 , n2517  };
2

```

- *Input/Output of ID_STAGE*: the only problem is on *pc_id_i* that is translated in a composition between *pc_id* (31 bits) and *id_stage_i_hwloop_target[0]* while the other signals are easily found.
- *Signals of ID_STAGE*: *instr* is a mix between *instr_rdata_id* and some nets. All *imm* signals depend on *instr* signal. All *fp* signals are set to 0 because floating point unit is not active.

There are some cases in which finding the corresponding signal in gate level is not easy. One of these signals is *operand_a_fw_id*. In this case, the implemented solution is deriving it in the same way it is got in the STRUCTURAL_RISCV as shown in *Listing 4.4*.

Listing 4.4: operand_a_fw_id calculation

```

1  always @*
2  begin : operand_a_fw_mux
3      case (id_stage_i_operand_a_fw_mux_sel)
4          SEL_FW_EX:    operand_a_fw_id = regfile_alu_wdata_fw;
5          SEL_FW_WB:    operand_a_fw_id = ex_stage_i_lsu_rdata_i;
6          SEL_REGFILE:  operand_a_fw_id =
id_stage_i_regfile_data_ra_id;
7          default:      operand_a_fw_id =
id_stage_i_regfile_data_ra_id;

```

```

8         endcase
9     end
10

```

This code can be considered as a black box able to generate the interested signal providing him, as inputs, signals that can be found in a simpler way. This concept is explained in *Figure 4.2*. This figure explains all the connecting

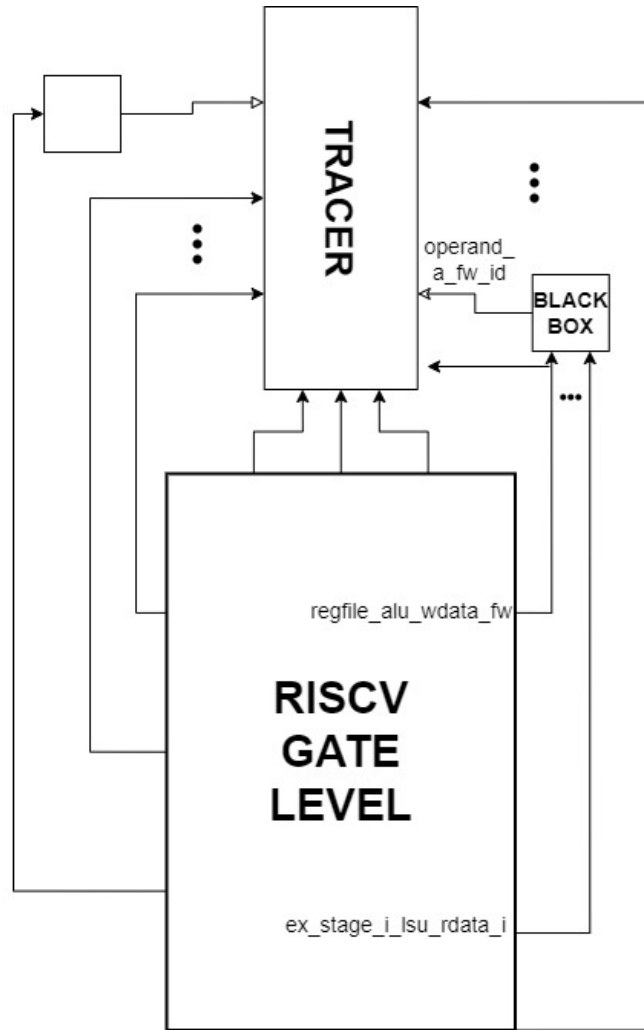


Figure 4.2: general scheme of how the tracer is connected to the RISC_V_GATE

process between RISC_V_GATE and the tracer. In fact, as we can see, some signals are connected directly from the processor to the tracer why others,

instead, have an intermediate block.

- *Input/Output of EX_STAGE*: the only case is *ex_stage_i.branch_in_ex_i* and the corresponding signal in the netlist is *branch_in_ex*.
- *Input/Output of CONTROLLER*: The name of structural processor signals and gate processor signals are similar.

The main problem of this approach is that when something changes in the processor, some signals or nets can change their name or the instantiating of some components can be redefined. In order to solve it an idea could be consider the processor as a black box and rewrite all the signals to be connected with the tracer as input or output.

After all signals of the port map were connected to the port map, a validation was needed to prove the correctness of this approach. That was done comparing the output of the tracer connected to the structural processor and the one of the tracer connected to the gate processor. It will be explained in the next chapter.

4.3 Validation of the FI flow

After creating the comparison instrument (the tracer) the flow of *Figure 4.1* was implemented. The validation of the flow was performed on a simple combinational circuit to assure its proper functioning. This operation was realised with the help of a slightly modified scheme as shown in *Figure 4.3*.

As said before, the DUT chosen for this purpose was a simple combinational circuit shown in *Figure 4.4* and the validation flow was applied on this simple circuit. Some differences can be seen comparing it with the one of *Figure 4.1*.

- DUT and TB are obviously different and the input patterns are provided by an ATPG and not by a code, as for the PULPINO flow.
- In this scheme, another comparison is included between the fault coverage provided by the ATPG and the one provided by the analysis of the DIFF files.

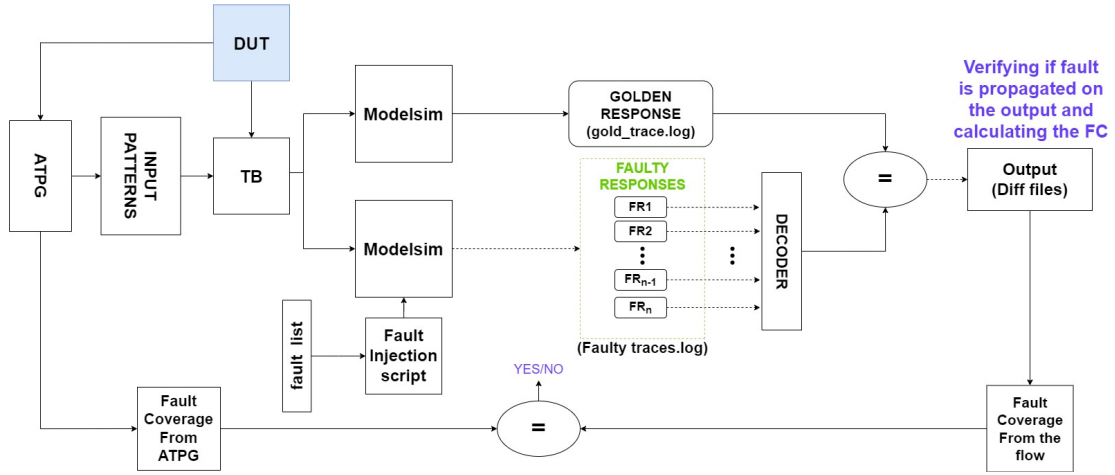


Figure 4.3: validation fault injection flow

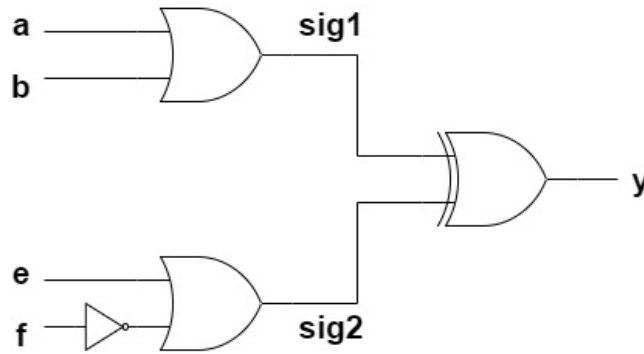


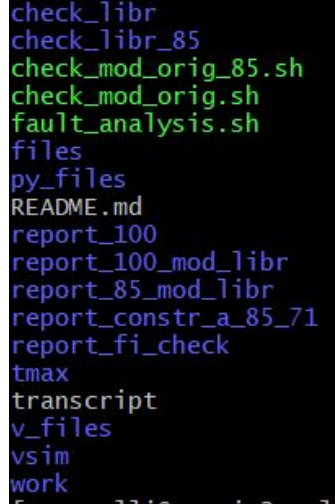
Figure 4.4: combinational circuit for validation flow

- The compared output files, in *Figure 4.3* are different respect than the ones compared in *Figure 4.1*. In fact, here, they are a simple transcription of the output behaviour of the circuit while, in the other scheme, they are the output files provided by the tracer.

The most important difference to analyze is the use of ATPG. In fact, thanks to the comparison between its fault coverage and the one provided by the output files the correctness of the flow was established (If they are equal it is validated while if they are different it is not). Let's analyze in detail the environment of the validation flow and how it works.

4.3.1 Analysis of the validation environment

The internal structure of the validation environment is shown in *Figure 4.5*.



```
check_libr
check_libr_85
check_mod_orig_85.sh
check_mod_orig.sh
fault_analysis.sh
files
py_files
README.md
report_100
report_100_mod_libr
report_85_mod_libr
report_constr_a_85_71
report_fi_check
tmax
transcript
v_files
vsim
work
```

Figure 4.5: internal structure of the validation environment

The most important folders and files for the validation are the ones described in the list below (some ones are not considered because they are useless for this purpose).

- `v_files` is a folder in which all the *Verilog* files are included as:
 - `pdt2002` that is the logic gate library.
 - `injection_module.v` that is the *Verilog* description of the circuit in *Figure 4.4* used to describe in detail its behaviour (written in a file) thanks to `tb_injection_module_rand.v`.
 - `atpg_injection_module.v` that is a module describing the same circuit using the tech library.It is simulated using two testbenches: `atpg_tb_injection_module.v`, that exploits patterns generated without constraints by the ATPG, and `atpg_tb_injection_module_constr_a.v`, that exploits patterns generated with a constraint.
- `tmax.tcl` that is an important script whose task is to generate the patterns using ATPG on TETRAMAX.

- all the `report*` folders are used for the report files.
- `files` folder where the fault list is included.
- `py_files` contains the script aims at modifying the simulation script for a correct FI.
- `tmax` contains the fault list and the patterns generated by TETRAMAX.
- `vsim` contains all the simulation files. Inserire simulation script description.
- `fault_analysis.sh` is the master script for running the flow.

Let's explore how the validation flow was used. After writing all the Verilog files to design the circuit in *Figure 4.4*, generating the patterns and the fault coverage with the ATPG was the next step. These patterns were used to write the testbench that, exploiting the flow, generated some outputs and a fault coverage (FC). The results of the comparison between these two FC were satisfying so the flow was validated (they were equal). Experiments and results of the validation process will be better explained in the next chapter.

4.3.2 FI script explanation

In all the flow there is one script that is implemented to inject a fault for each signal in the fault list. Specifically, It is a part of the TCL script `vsim/tcl_files/run.tcl` shown in *Listing 4.5*.

Listing 4.5: FI script

```
1 set forbiddenTime 0
2 set delay 44000
3
4 when -fast {/top_i/core_region_i/CORE/RISCV_CORE/ex_stage_i_alu_i/
   int_div_div_i/U406/A1'event and /top_i/core_region_i/CORE/
   RISCV_CORE/ex_stage_i_alu_i/int_div_div_i/U406/A1 = 1'h1} {
5     uivar forbiddenTime
6     uivar delay
7     if {$now != $forbiddenTime} {
```



```

8           force -freeze /tb/top_i/core_region_i/CORE/RISCV_CORE/
           ex_stage_i_alu_i/int_div_div_i/U406/A1 1'h0 -cancel $delay
9           set forbiddenTime [expr {$now + $delay}]
10        }
11    }

```

The general concept of a transition fault is that when there is an event on a specific signal it should be delayed of an amount of time. In fact, in the code there is an example in which when *U406/A1* has a rising edge it is delayed of 44 ns (because delay is set to 44000 and the resolution is in ps). In order to make this script work it is important to consider a "forbidden time". It is equal to the actual time plus the delay for injecting the transition fault because at that point the signal has a new transition and if the forbidden time was not set the script wrongly would try to stop the switch on this signal. This script will be changed after some modifications made on the tech library to solve the fan-out problem that will be explained in the next section.

4.4 Fan-out problem and how to solve it

At this time everything was ready for a correct execution of the flow and the beginning of the experiments on the processor. In reality, another problem caused by the fan-out should be fixed. In order to understand this issue, an example of *fan-out*, shown in *Figure 4.6*, is analyzed.

As can be seen the signal N3 is split in two branches that feed two different NAND gates and this ramification is called *fan-out*. The important thing to know is that *Faults on a fanout stem are not equivalent to faults on the corresponding fanout branches*. Using *Modelsim* [2], during a simulation of the circuit *Figure 4.6*, it was seen that if a fault is injected in one of the two branches, all the faults belonging to the fan-out are affected. Obviously, this behaviour contradicts the previous definition.

In order to solve the problem the idea was to modify the tech library in order to avoid this wrong behaviour.

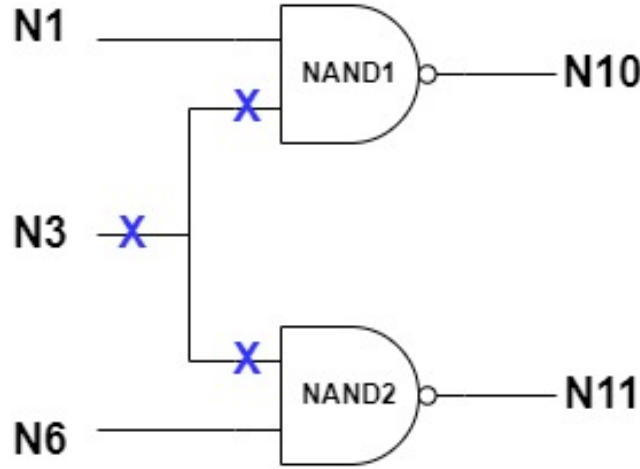


Figure 4.6: circuit with fan-out

4.4.1 Modification of the tech library

All the logic functions are included inside the tech library *asic/techlib/NangateOpenCellLibrary.v* and an example of an AND gate is shown in *Listing 4.6*.

Listing 4.6: verilog description of AND function inside the technology library

```

1 module AND2_X1 (A1, A2, ZN);
2   input A1;
3   input A2;
4   output ZN;
5   and(ZN, A1, A2);
6   specify
7     (A1 ==> ZN) = (0.1, 0.1);
8     (A2 ==> ZN) = (0.1, 0.1);
9   endspecify
10 endmodule

```

The problem was that the injection fault was performed through an external signal to the logic gate and not through its inputs. So, the idea to solve this problem could be finding a way to drive directly the inputs and outputs of each logic function when needed.

The implemented solution aimed at modifying each logic gate as shown in *Figure 4.7* (in this case an AND gate was considered). As can be seen the inputs or outputs

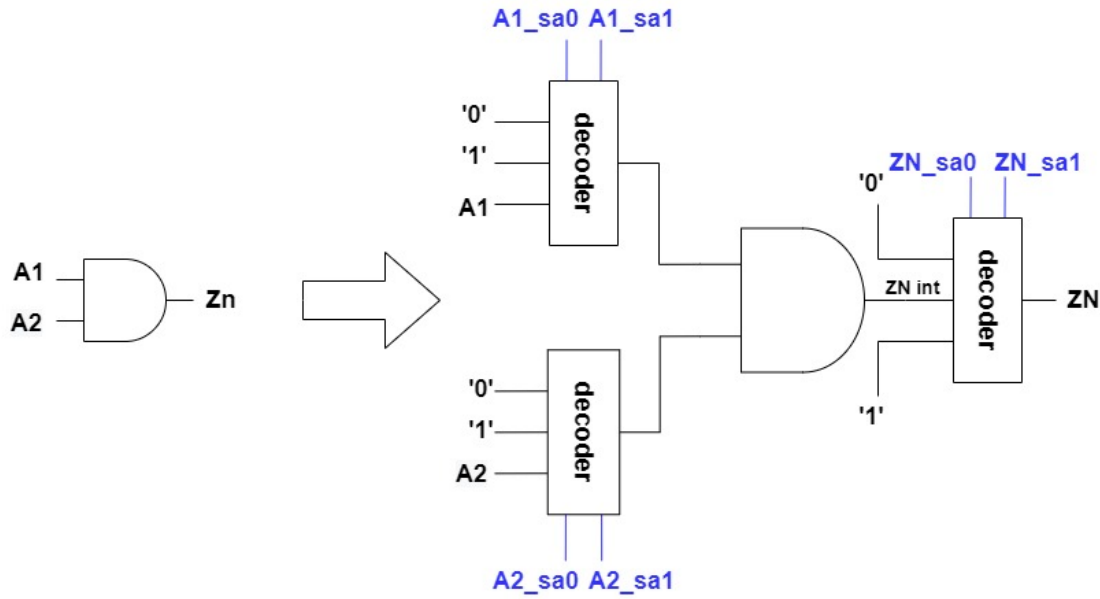


Figure 4.7: design modification on tech library (AND gate)

of the AND gate can be driven by some decoders and their selection signals. The Verilog code for this solution is the one shown in *listing 4.7*.

Listing 4.7: verilog description of AND function inside the technology modified library

```

1 module AND2_X1 (A1, A2, ZN);
2   input A1;
3   input A2;
4   output ZN;
5
6   wire A1_sa0 = 1'b0;
7   wire A1_sa1 = 1'b0;
8   wire A2_sa0 = 1'b0;
9   wire A2_sa1 = 1'b0;
10  wire ZN_sa0 = 1'b0;
11  wire ZN_sa1 = 1'b0;
12  wire ZN_int;
13
14  assign ZN = ZN_sa0 ? 1'b0 : ZN_sa1 ? 1'b1 : ZN_int;
15

```

```

16  and(ZN_int,
17      A1_sa0 ? 1'b0 : A1_sa1 ? 1'b1 : A1,
18      A2_sa0 ? 1'b0 : A2_sa1 ? 1'b1 : A2);
19
20  specify
21      (A1 => ZN) = (0.1, 0.1);
22      (A2 => ZN) = (0.1, 0.1);
23  endspecify
24
25  endmodule

```

In order to can modify each logic gate inside the tech library a *python* script called `mod_library.py` was implemented.

4.4.2 Analysis of the validation environment of the fan-out solution

Once the library was correctly modified, a check should be performed on its correctness. So an environment was created and its task was to test the new library on a circuit in *Figure 4.6*. The environment is very simple as shown in *Figure 4.8*.

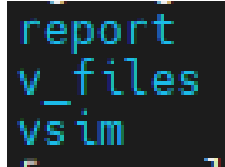


Figure 4.8: validate_fanout environment

Only three folders where created.

- **report** in which all the reports stored.
- **v_files** in which tb and Verilog description of the circuit are included.
- **vsim** in which the compilation and simulation scripts are included

The experimental results will be explained in the next chapter.

As the new library worked well the FI script should be adapted to the new logic gates.

4.5 Validation of the new FI script

The same validation environment of *Figure 4.3* was used, but some files were modified and other ones were added.

- The FI script `vsim/tcl_files/run.tcl` was modified in the way shown in *Listing 4.8* to adapt it to the new tech library.

Listing 4.8: new FI script

```

1  set forbiddenTime 0
2  set delay 44000
3
4  when -fast {/top_i/core_region_i/CORE/RISCV_CORE/
   ex_stage_i_alu_i/U701/A1'event and /top_i/core_region_i/CORE/
   RISCV_CORE/ex_stage_i_alu_i/U701/A1 = 1'h0} {
5      uivar forbiddenTime
6      uivar delay
7      if {$now != $forbiddenTime} {
8          force -freeze /tb/top_i/core_region_i/CORE/
   RISCV_CORE/ex_stage_i_alu_i/U701/A1_sa1 1'h1
9          force -freeze /tb/top_i/core_region_i/CORE/
   RISCV_CORE/ex_stage_i_alu_i/U701/A1_sa1 1'h0 $delay
10         set forbiddenTime [expr {$now + $delay}]
11     }
12 }
13
```

The difference with the previous one is that, here, the force commands are applied on the selection signals of the decoders (see *Figure 4.7*) and not directly on the signals of the fault list.

- In `py_files` a new simulation script was created for adapting the new library.
- New reports were created that confirmed the validity of the model. They will be analyzed in the next chapter.

4.6 Application of the flow on the processor

After the validation of the flow was confirmed on a simple circuit, the processor was ready to exploit it. So, Some experiments were performed to understand the behaviour of the processor using different code. One problem was the flow simulation time that for 700 faults in the fault list took almost three days. To fix it two scripts were created: One for creating 10 or more copy of the PULPINO environment and one for running in the same time all the copies of the environment (more details for using this option are explained in the user manual chapter). In this way the simulation time was reduced significantly. Anyway, this option could be not necessary because this flow should be usually exploited for a small number of faults related, for example, to a specific module.

Instead, the last work done was to create a python script able to resume all the results in a csv files. After that the environment was ready to be used or still improved.

Chapter 5

Experimental results

In this chapter, all the experimental results got in the different chronological steps for building the environment are deeply analyzed.

5.1 Validation of the tracer on the gate-level processor model

This validation process proves that the tracer works well in the gate-level processor. It is performed exploiting the output generated by the tracer during the simulation that is the one shown in *Listing 5.1*.

Listing 5.1: part of the output generated during the simulation thanks to the tracer

	Time	Cycles	PC	Instr	Mnemonic
1	18880000	455	00000080	03d0606f	jal
2	x0, 26684				
3	18960000	457	000068bc	30501073	csrrw
	x0, x0, 0x305				
4	19000000	458	000068c0	00000093	addi
	x1, x0, 0	x1=00000000			
5	19040000	459	000068c4	00008113	addi
	x2, x1, 0	x2=00000000	x1:00000000		

6	19080000	460 000068c8 00008193 addi
	x3, x1, 0	x3=00000000 x1:00000000
7	19120000	461 000068cc 00008213 addi
	x4, x1, 0	x4=00000000 x1:00000000
8	...	
9	...	
10	...	

This output is a representation of the assembly code got thanks to the signals of the processor connected to the tracer. In particular, the simulation time, the number of the cycle, the address in the program counter, the opcode of the instruction and its name, the registers involved in the operation and immediate numbers are reported.

If the output corresponds to the original test code written by the engineer it means that the processor works well. Obviously, the processor works well so the question should be: is the tracer properly connected with the different signals of the gate level processor? In order to answer it, the generated output of the gate level processor simulation is compared with the one generated by the structural processor and if they are equal, the tracer behaviour inside the gate level processor is validated. The comparison is performed thanks to the tool MELD and the result is satisfying unless for a portion of the program that is different between the two log files as shown in *Figure 5.1*.

This difference is not an error due to a wrong mapping of the tracer in the *Verilog* netlist, but it is caused by a part of the *SystemVerilog* code shown in *Listing 5.2*. How it can be seen, the *Ifdef* condition determines that if synthesis is performed, the first address of *PCCR_q* is always read otherwise the complete address is considered. This causes the differences in reading values of *csr* registers between 780 and 79F(32 count registers). Moreover, there is a bug at the starting point of 78c (12th count register) where the value read by register is unknown because the instruction tries to access in a part of memory that does not exist, considering that *PCCR_q* is composed of 11 cells of 32 bits.

Listing 5.2: different behaviors of gate and structural processors

Experimental results

x3=ffffffff				16184 000063ae 786021f3 csrrs	x3, x0, 0x786	x3=00000000
x3:ffffffff	PA:00000000	→	←	16185 000063b2 00302023 sw	x3, 0(x0)	x3:00000000
x2:ffffffff				16186 000063b6 78611073 csrrw	x0, x2, 0x786	x2:ffffffff
x3=ffffffff				16192 000063ba 787021f3 csrrs	x3, x0, 0x787	x3=00000000
x3:ffffffff	PA:00000000	→	←	16193 000063be 00302023 sw	x3, 0(x0)	x3:00000000
x2:ffffffff				16194 000063c2 78711073 csrrw	x0, x2, 0x787	x2:ffffffff
x3=ffffffff				16200 000063c6 788021f3 csrrs	x3, x0, 0x788	x3=00000000
x3:ffffffff	PA:00000000	→	←	16201 000063ca 00302023 sw	x3, 0(x0)	x3:00000000
x2:ffffffff				16202 000063ce 78811073 csrrw	x0, x2, 0x788	x2:ffffffff
x3=ffffffff				16208 000063d2 789021f3 csrrs	x3, x0, 0x789	x3=00000000
x3:ffffffff	PA:00000000	→	←	16209 000063d6 00302023 sw	x3, 0(x0)	x3:00000000
x2:ffffffff				16210 000063da 78911073 csrrw	x0, x2, 0x789	x2:ffffffff
x3=ffffffff				16216 000063de 78a021f3 csrrs	x3, x0, 0x78a	x3=00000000
x3:ffffffff	PA:00000000	→	←	16217 000063e2 00302023 sw	x3, 0(x0)	x3:00000000
x2:ffffffff				16218 000063e6 78a11073 csrrw	x0, x2, 0x78a	x2:ffffffff
x3=ffffffff				16224 000063ea 78b021f3 csrrs	x3, x0, 0x78b	x3=00000000
x3:ffffffff	PA:00000000	→	←	16225 000063ee 00302023 sw	x3, 0(x0)	x3:00000000
x2:ffffffff				16226 000063f2 78b11073 csrrw	x0, x2, 0x78b	x2:ffffffff
x3=ffffffff				16232 000063f6 78c021f3 csrrs	x3, x0, 0x78c	x3=xxxxxxxx
x3:ffffffff	PA:00000000	→	←	16233 000063fa 00302023 sw	x3, 0(x0)	x3:xxxxxxxx
x2:ffffffff				16234 000063fe 78c11073 csrrw	x0, x2, 0x78c	x2:ffffffff
x3=ffffffff				16240 00006402 78d021f3 csrrs	x3, x0, 0x78d	x3=xxxxxxxx
x3:ffffffff	PA:00000000	→	←	16241 00006406 00302023 sw	x3, 0(x0)	x3:xxxxxxxx
x2:ffffffff				16242 0000640a 78d11073 csrrw	x0, x2, 0x78d	x2:ffffffff
x3=ffffffff				16248 0000640e 78e021f3 csrrs	x3, x0, 0x78e	x3=xxxxxxxx
x3:ffffffff	PA:00000000	→	←	16249 00006412 00302023 sw	x3, 0(x0)	x3:xxxxxxxx

Figure 5.1: difference between the two outputs from the different versions of the processors

```

1      // look for 780 to 79F, Performance Counter Counter Registers
2      if (csr_addr_i[11:5] == 7'b0111100) begin
3          is_pccr      = 1'b1;
4
5          pccr_index = csr_addr_i[4:0];
6      `ifdef ASIC_SYNTHESIS
7          perf_rdata = PCCR_q[0];
8      `else
9          perf_rdata = PCCR_q[csr_addr_i[4:0]];
10     `endif
11     end

```

5.2 Validation of the FI flow

In order to validate the flow, the scheme in *Figure 5.2* is used. The reports generated are divided into different categories:

1. report_fi_check;
2. report_100 and report_constr_a_85_71;

3. report_100_mod_libr, report_85_mod_libr,
check_libr and check_libr_85.

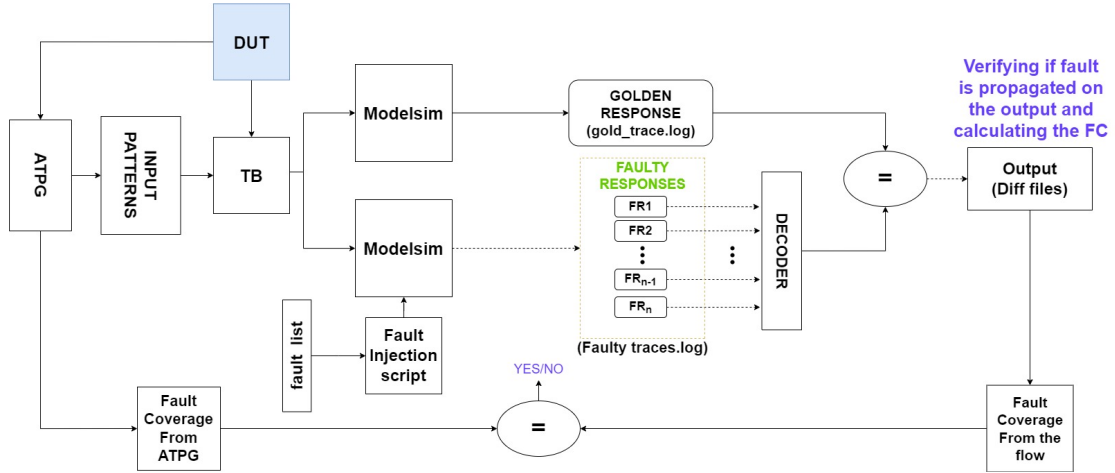


Figure 5.2: validation fault injection flow

As just said, in order to prove the flow, a simple circuit (shown in *Figure 5.3*) is designed, composed of two OR gates and one XOR gate.

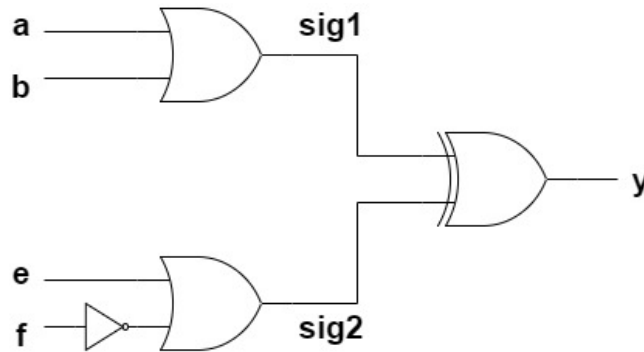


Figure 5.3: combinational circuit for validation flow

5.2.1 Report-fi-check

This folder is composed of reports given by an initial test of the flow in which random patterns are used instead of the ones generated by the ATPG. So, It is

not the proof of a well working flow, because a fault coverage to compare is not reported, but it can help us to understand the behaviour of the circuit and if the FI injection flow script, apparently, works correctly.

In this case, two *Verilog* files of `v_files` are used for compilation and then simulation: `injection_module.v` that aims to describe the simple circuit in *Figure 5.3* without using the tech library and a testbench `rand_tb_injection_module.v` in which random patterns are provided for the simulation. Thanks to this simulation, a detailed file of the golden simulation and all the diff files (the ones got by the difference between the golden output and every faulty outputs derived from each fault of the fault list) are created. The former is shown in *Listing 5.3*.

Listing 5.3: golden output for initial FI check

```

1      0 ps: a=0 b=0 e=0 f=0 -> sig1=0 sig2=1 -> y=1
2    90000 ps: a=1 b=0 e=0 f=0 -> sig1=1 sig2=1 -> y=0
3   180000 ps: a=0 b=0 e=0 f=1 -> sig1=0 sig2=0 -> y=0
4   270000 ps: a=0 b=1 e=0 f=1 -> sig1=1 sig2=0 -> y=1
5   400000 ps: a=0 b=0 e=1 f=1 -> sig1=0 sig2=1 -> y=1
6   440000 ps: a=0 b=1 e=1 f=1 -> sig1=1 sig2=1 -> y=0
7   500000 ps: a=0 b=0 e=1 f=1 -> sig1=0 sig2=1 -> y=1
8   560000 ps: a=1 b=0 e=1 f=1 -> sig1=1 sig2=1 -> y=0
9   610000 ps: a=0 b=0 e=1 f=1 -> sig1=0 sig2=1 -> y=1
10  640000 ps: a=0 b=1 e=1 f=1 -> sig1=1 sig2=1 -> y=0
11  710000 ps: a=0 b=0 e=1 f=1 -> sig1=0 sig2=1 -> y=1

```

This is the correct behaviour of the circuit with the random patterns. Simulation time and the behaviour of inputs, signals and outputs are reported here. Instead, an example of diff files is reported in *Listing 5.4*.

Listing 5.4: STF on input "a" diff file for initial FI check

```

1 3a4,5
2 > 180001 ps: a=1 b=0 e=0 f=1 -> sig1=1 sig2=0 -> y=1
3 > 224001 ps: a=0 b=0 e=0 f=1 -> sig1=0 sig2=0 -> y=0
4 10c12,14
5 < 640000 ps: a=0 b=1 e=1 f=1 -> sig1=1 sig2=1 -> y=0
6 —
7 > 610001 ps: a=1 b=0 e=1 f=1 -> sig1=1 sig2=1 -> y=0

```

8	>	640000	ps:	a=1 b=1 e=1 f=1 -> sig1=1 sig2=1 -> y=0
9	>	654001	ps:	a=0 b=1 e=1 f=1 -> sig1=1 sig2=1 -> y=0

Remembering that the rows with < are the golden ones while > represents the faulty ones, in this file, a STF (slow to fall) transition fault is injected on input "a" of the simple circuit shown in *Figure 5.3*. The fact that the file in *Listing 5.4* is not empty means that the injection causes a different behaviour of the simulation respect than the golden simulation. For example, in the last file, the first two rows in the faulty simulation are generated because the STF on input "a" delays the transition, so "a" remains stable for 44 ns and then the transition is performed. Moreover, as we can see, the input is correctly propagated on the output because in the first row "y" is equal to 1 and after 44 ns is set to 0.

Therefore, the FI script and the flow apparently seems to work properly but we have not obtained the definitive proof yet. In order to do that, as suggested by the validation flow in *Figure 5.2*, the ATPG should be used to generate the test vectors and a fault coverage (FC) to compare with the FC of the flow.

5.2.2 Report_100 and report_constr_a_85_71

The reports inside these folders validate the flow. The patterns generated thanks to *Tetramax* are the crucial point here. In fact, in the `tmax` folder we can found two stil files: `test_vectors.stil`, in which all the possible test vectors are generated and the FC is 100 % and `test_vectors_constr_a.stil`, in which a constraint on *a* (not all the possible patterns are generated) is set to 0 and the FC is 85.71 %.

The considered file for compilation and simulation inside `v_files` are the following ones: the injection module `atpg_injection_module.v` and the testbenches `atpg_tb_injection_module.v` and `atpg_tb_injection_module_constr_a`.

A FC of 100 % should be achieved with the patterns inside the first testbench while a FC of 85.71 % with the patterns inside the second testbench. Moreover, the tech library is considered now because the module of the simple circuit exploits it.

Both the folders have the same way to validate the flow, so let's analyze only the first one. In `report_100` the patterns are used in `atpg_tb_injection_module.v` in the way shown in *Listing 5.5*.

Listing 5.5: testbench with patterns of 100 % FC

```

1 //pattern0
2 #50000 a_i = 0; b_i = 0; e_i =0; f_i =1;
3 #50000 a_i = 1; b_i = 0; e_i =0; f_i =1;
4 //pattern1
5 #50000 a_i = 0; b_i = 0; e_i =1; f_i =0;
6 #50000 a_i = 1; b_i = 1; e_i =1; f_i =0;
7 //pattern2
8 #50000 a_i = 0; b_i = 0; e_i =0; f_i =1;
9 #50000 a_i = 0; b_i = 1; e_i =0; f_i =1;
10 //pattern3
11 #50000 a_i = 1; b_i = 1; e_i =1; f_i =0;
12 #50000 a_i = 0; b_i = 0; e_i =0; f_i =1;
13 //pattern4
14 #50000 a_i = 1; b_i = 0; e_i =0; f_i =1;
15 #50000 a_i = 0; b_i = 0; e_i =0; f_i =0;
16 //pattern5
17 #50000 a_i = 0; b_i = 1; e_i =0; f_i =1;
18 #50000 a_i = 1; b_i = 1; e_i =1; f_i =1;

```

Each 100 ns the output of the circuit is written on the golden simulation file shown in *Listing 5.6*.

Listing 5.6: golden output for validating the flow

```

1 101 ns:    y=1
2 201 ns:    y=0
3 301 ns:    y=1
4 401 ns:    y=0
5 501 ns:    y=1
6 601 ns:    y=0

```

In this case, differently to the one considered in *Listing 5.3*, only the time simulation and the outputs are reported on the output file. Comparing the golden output with a faulty output, obtained from the STR on input "a" as before, the result is the one in *Listing 5.7*.

Listing 5.7: STF on input "a" diff file for initial validating the flow

```

1 4,5c4,5
2 < 401 ns:    y=0
3 < 501 ns:    y=1
4 ———
5 > 401 ns:    y=1
6 > 501 ns:    y=0

```

As we can see, when the STF is delayed the diff output marks a difference, so the FI script works well. In order to validate it the FC should be 100 % and ,in fact, the percentage is achieved because all the files generated are not empty, therefore the output is propagated for each fault injected and the FC obtained by the flow is 100 %.

The same process is applied with the patterns generated by the testbench with the constraints and the FC, in this case 85.71 %, is matched again.

5.2.3 Report_100_mod_libr, report_85_mod_libr, check_libr and check_libr_85

In order to solve the fan-out problem (explained in the chapter 4), the tech library and the FI script have been changed. So, a new test is performed with these two new elements.

In `report_100_mod_libr` and `report_85_mod_libr`, the format of the reports is similar to the one in *Listing 5.7*. To achieve these results, the same flow of before is done. Moreover, the test show the same results of the one run before, therefore it means that the modified flow works in the correct way.

In order to prove that these results are in line with what we expect a comparison is performed using the two scripts `check_mod_orig` and `check_mod_orig_85` and the outputs are stored in `check_libr` and `check_libr_85`. Their size is 0, so, considering that the comparison is performed thanks the *diff* command, all the results are in compliance between them.

5.3 Validation of fan-out solution

We recall that the *fan-out* problem in *Modelsim* [2] is that, using the original library, if we have a branch and we inject a fault on one of the stems all the branches are affected by the fault. The modified library solves this problem and some experiments are conducted to validate it on the circuit shown in *Figure 5.4*.

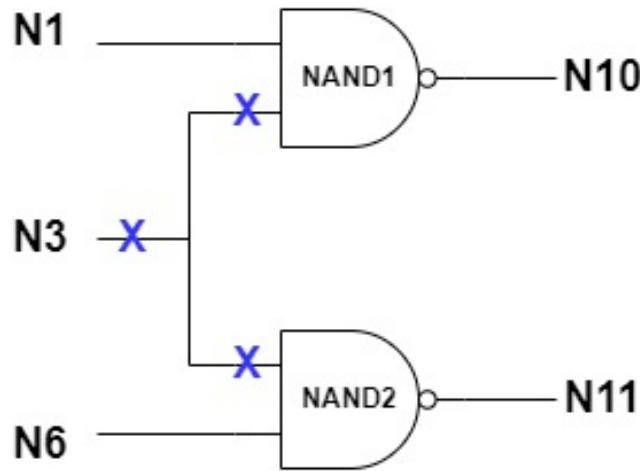


Figure 5.4: circuit with fan-out

All the files for validation are inside the folder `validate_fanout/reports`. There are four more reports inside this folder that correspond to four different simulations.

- `golden_sim_orig_lib` in which a golden simulation (without FI) with original technology library is performed.
- `faulty_sim_orig_lib` in which a faulty simulation (with FI) with original technology library is performed. A stuck-at-0 is applied on the first NAND on A2 (see *Figure 5.5*) in order to understand what happens on the branch.
- `golden_sim_mod_lib` in which a golden simulation (without FI) with modified technology library is performed.
- `faulty_sim_mod_lib` in which a faulty simulation (with FI) with modified technology library is performed. A stuck-at-0 is applied on the first NAND on A2 (see *Figure 5.5*) in order to understand what happens on the branch.

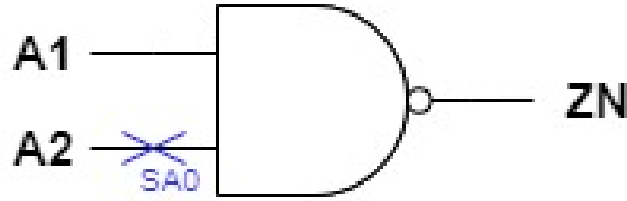


Figure 5.5: internal structure of the NAND

Each simulation generates three reports.

- the report of the behaviour of the circuit in *Figure 5.4*..
- the reports of the behaviour of the two NAND gates.

So, in total, there are 12 files to be compared to validate the fan-out solution. Analyzing the behaviour of the circuit and the two internal NAND gates during a golden simulation (shown in *Listing 5.8*) and comparing it with the simulation during the FI with a SA0 on NAND1/A2 (shown in *Listing 5.9*), It can be seen how the FI does not work correctly.

In fact, for example, in the second row of the entire circuit of the golden simulation N1, N3 and N6 are equal to 1. So, the outputs of the NAND gates N10 and N11 are equal to 0. The problem is that if we inject the fault (SA0 on NAND1/A2) also NAND2/A1 is affected and both the outputs are equal to 1, therefore all the branch driven by the input N3 is affected. Instead, N10 should be equal to 1 while N11 should be equal to 0.

Listing 5.8: reports of golden simulation with original library

```

1  \entire circuit
2  0 ns:    N1=0 N3=0 N6=0 -> N10=1 N11=1
3
4  |50 ns:   N1=1 N3=1 N6=1 -> N10=0 N11=0|
5
6  100 ns:   N1=0 N3=1 N6=0 -> N10=1 N11=1
7  150 ns:   N1=1 N3=1 N6=0 -> N10=0 N11=1
8
9  \NAND1
10 0 ns:    A1=0 A2=0 -> ZN=1

```



```

11 |
12 |50 ns:    A1=1 A2=1 -> ZN=0|
13 |
14 |100 ns:   A1=0 A2=1 -> ZN=1
15 |150 ns:   A1=1 A2=1 -> ZN=0
16 |
17 |\\NAND2
18 | 0 ns:    A1=0 A2=0 -> ZN=1
19 |
20 |50 ns:    A1=1 A2=1 -> ZN=0|
21 |
22 |100 ns:   A1=1 A2=0 -> ZN=1

```

Listing 5.9: reports of faulty simulation with original library

```

1 |\\entire circuit
2 | 0 ns:    N1=0 N3=0 N6=0 -> N10=1 N11=1
3 |
4 |50 ns:    N1=1 N3=0 N6=1 -> N10=1 N11=1|
5 |
6 |100 ns:   N1=0 N3=0 N6=0 -> N10=1 N11=1
7 |150 ns:   N1=1 N3=0 N6=0 -> N10=1 N11=1
8 |
9 |\\NAND1
10| 0 ns:    A1=0 A2=0 -> ZN=1
11|
12|50 ns:    A1=1 A2=0 -> ZN=1|
13|
14|100 ns:   A1=0 A2=0 -> ZN=1
15|150 ns:   A1=1 A2=0 -> ZN=1
16|
17|\\NAND2
18| 0 ns:    A1=0 A2=0 -> ZN=1
19|
20|50 ns:    A1=0 A2=1 -> ZN=1|
21|
22|100 ns:   A1=0 A2=0 -> ZN=1

```

Once, the library is modified the new golden simulation, shown in *Listing 5.10*, is

compared with the one in *Listing 5.8* and they are equal. This result proves that the new library does not affect the normal behaviour of the circuit. After that, the fault is injected and comparing the text file in *Listing 5.10* with the one in *Listing 5.11* it can be seen that all the branch is not affected on both the stems as before. In fact, in the faulty simulation, N10 is equal to 1 and N11 is equal to 0.

Listing 5.10: reports of golden simulation with modified library

```

1  \\entire circuit
2    0 ns:      N1=0 N3=0 N6=0 -> N10=1 N11=1
3
4  |50 ns:      N1=1 N3=1 N6=1 -> N10=0 N11=0|
5
6  100 ns:     N1=0 N3=1 N6=0 -> N10=1 N11=1
7  150 ns:     N1=1 N3=1 N6=0 -> N10=0 N11=1
8
9  \\NAND1
10   0 ns:      A1=0 A2=0 -> ZN=1
11
12 |50 ns:      A1=1 A2=1 -> ZN=0|
13
14 100 ns:      A1=0 A2=1 -> ZN=1
15 150 ns:      A1=1 A2=1 -> ZN=0
16
17 \\NAND2
18   0 ns:      A1=0 A2=0 -> ZN=1
19
20 |50 ns:      A1=1 A2=1 -> ZN=0|
21
22 100 ns:      A1=1 A2=0 -> ZN=1

```

Listing 5.11: reports of faulty simulation with modified library

```

1  \\entire circuit
2    0 ns:      N1=0 N3=0 N6=0 -> N10=1 N11=1
3
4  |50 ns:      N1=1 N3=1 N6=1 -> N10=1 N11=0|
5
6  100 ns:     N1=0 N3=1 N6=0 -> N10=1 N11=1

```

```

7 150 ns:    N1=1 N3=1 N6=0 -> N10=1 N11=1
8
9 \\NAND1
10 0 ns:     A1=0 A2=0 -> ZN=1
11
12 |50 ns:    A1=1 A2=1 -> ZN=1|
13
14 100 ns:    A1=0 A2=1 -> ZN=1
15 150 ns:    A1=1 A2=1 -> ZN=1
16
17 \\NAND2
18 0 ns:     A1=0 A2=0 -> ZN=1
19
20 |50 ns:    A1=1 A2=1 -> ZN=0|
21
22 100 ns:    A1=1 A2=0 -> ZN=1

```

In conclusion, after these tests are performed, the fan-out solution is validated.

5.4 Experiments on the flow

In this section three experiments on the PULPINO processor, exploiting the designed flow, are analyzed. These experiments are inside the external environment `pulpino_flow` inside the folder `reports` and they are divided in three categories in several folders.

- `tracer_start_code`
- `tracer_division_code`
- `tracer_division_code_speed`

All the folders have in common a report file for each fault in the fault list and a summary file that resume if the faults are observed, the type of error (in the case analyzed here the only error is "data register error" that is an error on one of the bits of some register), the percentage of the observed faults and the total number of faults. An example of summary file is shown in *Listing 5.12*.

Listing 5.12: an extraction of post_processing report

```

1 FAULT,OBSERVED,TYPE
2      .
3      .
4      .
5 STR_RISCV_CORE_ex_stage_i_alu_i_U1709_A1,yes,data register error
6 STF_RISCV_CORE_ex_stage_i_alu_i_U1835_A1,yes,data register error
7 STR_RISCV_CORE_ex_stage_i_alu_i_int_div_div_i_U335_B1,no,none
8 STR_RISCV_CORE_ex_stage_i_alu_i_int_div_div_i_U339_B,no,none
9 STF_RISCV_CORE_ex_stage_i_alu_i_int_div_div_i_U338_B1,no,none
10     .
11     .
12     .
13 TOT N FAULTS,N FAULTS OBSERVED,% OBSERVED
14 696,93,13.36

```

The summary file simply provides a recap of the all files generated by the flow during its execution. These files are generated by a comparison (diff command in bash) between the output of the tracer of the golden simulation and each one of the faulty simulations. An example is provided in *Listing 5.13* where "<" represents the golden file while ">" represents the faulty file. Here, for example, a difference on the register x31 can be noticed in the bit 0.

Listing 5.13: example of a report file generated by *diff*

```

1 2237c2237
2 <          2944440000          7344 00001698 10098fb3 p.ff1
   x31, x19          x31=00000005 x19:1a8fbb20
3 ———
4 >          2944440000          7344 00001698 10098fb3 p.ff1
   x31, x19          x31=00000004 x19:1a8fbb20

```

Let's analyze the three folders above one by one.

5.4.1 Tracer_start_code

This is the analysis done with the original assembly code provided at the beginning of this study. Moreover, a fault list and an analysis of them by Z01X is provided.

The benefit is that the resulting reports generated by the flow are conform to the ones generated by Z01X. In fact, all the signals of the division exploiting the FI are not observed as in the initial report because the bits of the register are masked at some point during the loop of the division implementation. Instead, the other faults are usually observed. In the summary report these are the recap results: "tot faults 696, observed faults 93, percentage observed faults 13.36 %"

5.4.2 Tracer__division__code

The same analysis of before is performed with a more part of assembly code, provided by a previous study, that should be able to make observable the FI on some faults belonging to the division module. This is possible because of this code, shown in *Listing 5.14*, that prevents the interested bit to be masked by the loop of the division. It means that, for example, if 10 loops must be performed by the division and the fault is injected on a bit at the second iteration, the result of this bit should be the same at the end of the division process.

Listing 5.14: code for observing the fault of the division module

```

1 li    x5,0xaaaaaaaa
2 li    x6,1
3 div   x7,x5,x6
4 sw    x7,0(sp)

```

At the end of the execution the result can be summarized in "tot faults 696, observed faults 311, percentage observed faults 44.68 %". It can be seen that the percentage of observed faults is increased because a lot of division module faults are observable thanks to the code of *Listing 5.14*.

5.4.3 Tracer__division__code__speed

Running the flow the problem can be the execution time that can increase a lot with a bigger size of the fault list. In order to improve the performance a speed environment exists and it is validated running the flow. The results are the same of the previous report that exploit the same assembly code, so this environment

can be used. The execution time is really decreased in fact in the previous FI it was almost 3 days while exploiting this environment is 12 hours.

Chapter 6

Conclusions and perspectives

The objective of this study was to find a way to guide the testing engineer to develop an effective test program. Above all, the main problem was trying to understand why some faults injected are not detected (ND), so not propagated on the POs. Therefore, the idea to fix this issue was to increase their observability, generating, in an automatic way, some output files that can provide to the testing engineer some useful additional information for guiding the test engineer to improve the test program.

As the experimental results prove, the objective has been achieved because a lot of information thanks to the implemented FI flow can be exploited. In fact, for each fault injected the output generated by the tracer establishes if the considered fault is observable or not. Moreover, the generated file could contain some lines that can be exploited to analyze why the PULPINO processor does not work well during a specific fault injection.

In *Listing 6.1* an example of output file generated by the flow is reported.

Listing 6.1: example of a report file generated by the flow

```
1 2237c2237
2 <          2944440000          7344 00001698 10098fb3 p.ff1
          x31, x19          x31=00000005 x19:1 a8fbb20
```


In conclusion, we can state that the developed flow executed on the PULPINO environment, as case study, can be really useful to understand how to improve the test code but it is only a starting point because more potential improvements can be made, above all. An example of potential improvement could be in terms of post-processing analysis of the outputs generated by the flow execution.

Bibliography

- [1] Riccardo Cantoro; Patrick Girard; Riccardo Masante; Sandro Sartoni; Matteo Sonza Reorda; Arnaud Virazel. «Self-Test Libraries Analysis for Pipelined Processors Transition Fault Coverage Improvement». In: 2021. URL: <https://ieeexplore-ieee-org.ezproxy.biblio.polito.it/document/9486711>.
- [2] Mentor Graphics Corporation. «ModelSim® User's Manual». In: vol. Software Version 10.1c. 2012. URL: https://www.microsemi.com/document-portal/doc_view/131619-modelsim-user.
- [3] Synopsys. «Z01X Functional Safety Assurance». In: *High-Speed Fault Simulation Solution for IEC 61508 and ISO 26262 Compliance*. 2022. URL: <https://www.rfc-editor.org/info/rfc8321>.
- [4] Synopsys. «TetraMAX® ATPG». In: *User Guide*. Vol. Version A-2007.12-SP4. June 2008. URL: <https://manualzz.com/doc/o/pfc01/test-pattern-validation-user-guide-tetramax-overview>.
- [5] «Definition – What is Process Variation?» In: URL: <https://tallyfy.com/process-variation/#:~:text=Process%20variation%20happens%20when%20processes,in%20transactional%20and%20production%20processes.&text=Simply%20put%2C%20we%20aim%20to,that%20the%20results%20are%20repeatable..>
- [6] Politecnico di Torino Matteo Sonza Reorda. «Other fault models». In: *01RK-ZOQ - Testing and fault tolerance*.

- [7] Jad G. Atallah. «EDA Tools Usage and Tutorial Authoring for Basic Electronic Circuits Education». In: September 2018. URL: <https://ieeexplore-ieee-org.ezproxy.biblio.polito.it/document/8629482/authors#authors>.
- [8] Politecnico di Torino Matteo Sonza Reorda. «Stuck-at faults». In: *01RKZOQ - Testing and fault tolerance*.
- [9] Ann Steffora Mutschler. «Using Processor Trace At The System Level». In: 2020. URL: <https://semiengineering.com/using-processor-trace-at-the-system-level/>.