POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Evolutionary Techniques for Maximizing the Processor Stress

Supervisors

Matteo SONZA REORDA

Nikolaos DELIGIANNIS

Riccardo CANTORO

Candidate

Chenghan ZHOU

July 2023

Table of Contents

List of Tables III

List of Figures IV

Acronyms V

1 Introduction 1
- 1.1 Problem To Be Solved . . . 1
- 1.2 Brief Summary . . . 2
- 1.3 Thesis Structure . . . 2

2 Background 4
- 2.1 Burn-In Test . . . 4
- 2.2 Stimuli Generation Strategy . . . 6
- 2.3 Evolutionary Algorithm . . . 7
- 2.4 MicroGP3 Tool-Kit Introduction . . . 8

3 Implementation 10
- 3.1 RI5CY Stimuli Generation . . . 13
  - 3.1.1 Node Pair Extraction Strategy . . . 13
  - 3.1.2 Evolutionary Algorithm Constraint . . . 14
  - 3.1.3 Evolutionary Algorithm Setting And Population Setting . . . 20
- 3.2 IBEX Stimuli Generation . . . 22
  - 3.2.1 Node Pair Extraction Strategy . . . 22
  - 3.2.2 Evolutionary Algorithm Constraint . . . 24
  - 3.2.3 Evolutionary Algorithm Setting And Population Setting . . . 32

4 Results 33
- 4.1 The Result Of Switching Activity . . . 33
- 4.2 The Result Of Running Time . . . 36

5 Discussion And Conclusions 37

Bibliography 39

List of Tables

3.1 adder instruction group classification 15
3.2 adder instruction group classification 17
3.3 LSU instruction group 20
3.4 Operator list 22
3.5 IBEX adder instruction group 25
3.6 IBEX compressed decoder instruction group 27
3.7 IBEX decoder instruction group 28
3.8 IBEX operator list 32

4.1 stress efficiency of RI5CY stimuli on different functional unit 34
4.2 stress efficiency of IBEX stimuli on different functional unit 35
4.3 Table of result 36

List of Figures

2.1 Bathtub curve which represents the reliability of a product during its life cycle [5] . . . 5
2.2 Scenario of failure rate in the early stage with and without BI testing [7] . . . 5
2.3 Typical BI process [1] . . . 6
2.4 Typical BI process . . . 7
2.5 Genetic algorithm workflow . . . 9
2.6 MicroGP3 framework . . . 9

3.1 Flowchart of "run.py" . . . 11
3.2 Flowchart of "ugp3.evaluator.py" . . . 12
3.3 flow chart of parser . . . 23
3.4 relax mode . . . 24
3.5 insane mode . . . 24
3.6 instruction fetch stage structure . . . 27

4.1 QuestaSim GUI simulation interface . . . 34
4.2 dis-assembler interface . . . 35

Acronyms

IC
Integrated Circuit

PPM
Parts Per Million

DUT
Device Under Test

SBST
Software Based Self Test

ATPG
Automatic Test Pattern Generation

BI
Burn-In

SE
Stress Efficiency

IF
Instruction Fetch

ID
Instruction Decoder

LSU
Load-Store Unit

ISA

Instruction Set Architecture

Chapter 1

Introduction

As predicted by Moore's Law, the density of components integrated on ICs (Integrated Circuit) has significantly increased over the past few decades, resulting in smaller and more compact embedded systems. However, this progress has come with increased manufacturing complexity and difficulty. In particular, reliability is of utmost importance for embedded systems used in fields such as military and aerospace equipment, where their performance can have life-or-death consequences. Some of these applications must comply with high reliability standards, currently set to less than 1 PPM(Parts Per Million) failing [1]. Therefore, testing and evaluation of ICs and embedded systems are critical for all stakeholders involved in the manufacturing process, from design to market.

In order to improve the reliability of the product, functional testing is applied to products that have already been manufactured, but there may be some defects in the product that cannot be detected in this way. These products may perform well in functional testing or other end-of-manufacturing tests without device depreciation. In general, defects can be classified into two groups: time zero defects which can be detected by production tests without any stress, and latent defects which require stress to accelerate to the failing state in order to be detected [2].To further enhance the reliability of the product, burn-in testing is usually used as a complement to functional testing mentioned above.

1.1 Problem To Be Solved

The main idea of the commonly used burn-in test is to age the DUT (Device Under Test) by applying voltage or heating to the circuit in a climatic chamber, i.e. static burn-in. Dynamic burn-in is a screening simulating application conditions close to real use. It is also expected to help understand characteristics variations beforehand as commercial product failure recently comes up more frequently [3].

Several methods have been utilized for generating stimuli, including SBST (Software Based Self Test) and ATPG (Automatic Test Pattern Generation). However, this thesis describes a state-of-art method for generating stimuli, using a genetic algorithm, to achieve maximum switching activity for dynamic burn-in tests. This approach leverages gate level circuit description and other interconnection distribution information to generate stimuli that satisfy specific requirements based on multi-point stress metrics. The aim of this research is to explore the effectiveness of this new method and compare the properties when it is applied to 2 RISC-V based processors (i.e. RI5CY and IBEX) and additional information of the circuit.

1.2 Brief Summary

The main works and activities of this thesis could be summarized as follow:

- Study of the theory and the rationale of genetic algorithm;
- Study how to use the genetic algorithm tool-kit MicroGP3;
- Study the concept of multi-point stress metrics and the related papers;
- Implement a MicroGP3 generation workflow on RI5CY processor with random node pairs;
- Implement a full work flow on a real IBEX processor from node list generation with QestaSim to; generation of maximum toggle activity stimuli, but with additional information of layout description.

1.3 Thesis Structure

This thesis is divided into chapters which contain the following information:

- Chapter 2 provides an introduction to burn-in testing and reviews recent research in the field. It covers basic concepts and theories, as well as popular implementation methods. The chapter also includes a brief introduction to the evolutionary algorithm and the ugp3 tool-kit.
- Chapter 3 outlines the workflow of the burn-in test system and describes in detail the implementation methods and processes of the stimulus generation system using RI5CY and IBEX processors as DUTs.
- Chapter 4 presents the results of the developed GA-based maximum toggle activity burn-in stimuli generator. The chapter compares the running time and multi-point stress metrics which is designed to indicate the toggling activity of the circuit.

- Chapter 5 summarizes the conclusions drawn from the results obtained in Chapter 4.

Chapter 2

Background

At the end of the electronic device manufacturing process, all products must undergo functional testing to ensure their reliability. In most cases (over 90%), defects can be detected through methods such as functional testing, wafer testing, and initial electrical testing [4]. However, some defects cannot be directly detected by these methods, even if the test patterns and algorithms are perfectly designed. Typically, these defects are only detected as functional failures after the device has undergone a certain degree of degradation. If these weaknesses can't be detected before they goes to the market, the products usually have a very high infant morality, which means that they may produce some failures during their early stage, and this issue may costs a lot for enterprises. The curve which represents the relationship between failure rate and time is shown in figure 2.1

Based on the figure, the observed failure rate can be divided into three parts. Initially, the failure rate is dominated by the infant mortality failure, which is usually caused by latent manufacturing faults. This high failure rate in the early stage of the product life cycle decreases rapidly over time [6]. In the later stage, the main reasons for failure are random failure and wear-out failure, which gradually become more prominent as the product continues to age and undergo usage. By using the Burn-In testing before the product goes to the market, the infant morality failure rate will be obviously lower compared to the scenario without it, as show in figure 2.2 [7].

2.1 Burn-In Test

BI(Burn-In) test is a well know procedure in the microelectronics industry. It was first used to screen defects in low-volume immature parts in the sixties and was incorporated in the military standards by 1968 [8]. Nowadays, burn-in test has already become a main countermeasure against the infant-morality [9]. The

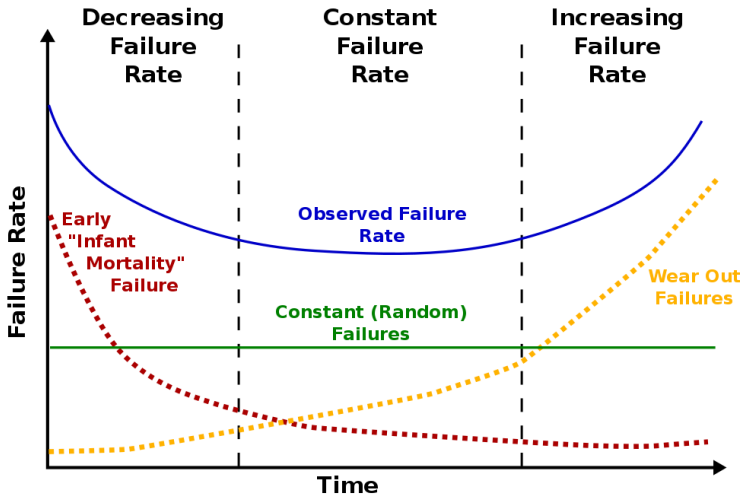

Figure 2.1: Bathtub curve which represents the reliability of a product during its life cycle [5]

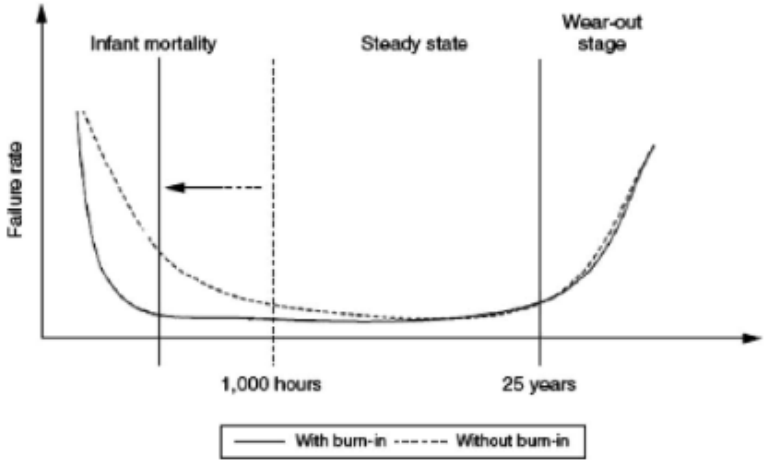

Figure 2.2: Scenario of failure rate in the early stage with and without BI testing [7]

BI process typically encompasses different stress and test steps that are shown in figure 2.3 and these steps are executed in a loop. By aging products, different types of defects that probably occur in the early life stages can be detected, such as resistive contacts/vias, resistive opens, resistive bridges, gate oxide shorts, improper implants and silicide breaks [1]. In general, burn-in test can be divided into two categories: static BI and dynamic BI. According to their different application

scenarios, they can be applied at either the wafer level or the package level.

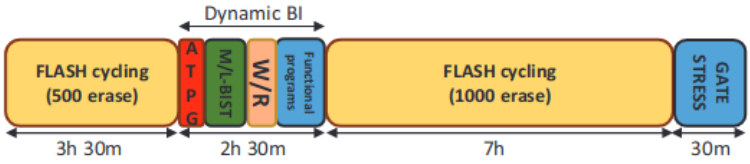

Figure 2.3: Typical BI process [1]

The static BI test is a method of subjecting materials to static stresses such as voltage and temperature in a climatic chamber. This involves heating the chips up to their specification limits using tunable voltage regulators mounted on the cold part of the test equipment to introduce voltage margins. To ensure precise control over the behavior of the test, the Arrhenius equation, which describes the relationship between temperature and chemical reaction, has been employed in this process. And for dynamic burn-in test, it involves multiple not only external stress such as factors mentioned above, but also internal stress effect that is produced by activating during the BI phase the different operational modes of the device under test (DUT) [10]

Up until recently, the most commonly applied BI procedure was static BI, during which the DUTs are exposed to a fixed and elevated temperature for an extended period of time without any application or stimulus during the test. This is achieved by placing the circuits into a climatic chamber that is able to heat the DUTs according to their specification limits. A drawback of static BI is that the circuit is not exercised. As the circuits' feature size continues to scale down and their structural and architectural complexity increases, so does the complexity and the cost of the BI test, rendering it unaffordable. BI test can be very time consuming, since its duration can be in the order of hours (especially for new technologies) and thus it can become a bottleneck for the whole manufacturing process.

2.2 Stimuli Generation Strategy

For BI process, switching activity measurement is performed through logic simulation by counting the number of transitions observed at gate-level for both state transitions and glitches according to equation 2.1 [1]. But in this thesis, we don't care about the effect of glitches.

$$SW_i^{t_eval} = \#transitions_i + \gamma \cdot \#gliches_i \tag{2.1}$$

Figure 2.4 illustrates the behavior under evaluation, which involves a node pair α and β produced based on gate-level description and other layout information.

During the sampling time window W, an algorithm generates stimuli to drive these nodes from their initial state to their reverse state and back. The behavior is then recorded as an SE (stress efficiency) function, as shown in equation 2.2, with possible values of 1, 2, 3, 4. Our objective is to generate functional sequences for each node pair that maximize the SE function's value of 4, indicating that both nodes underwent both transitions.

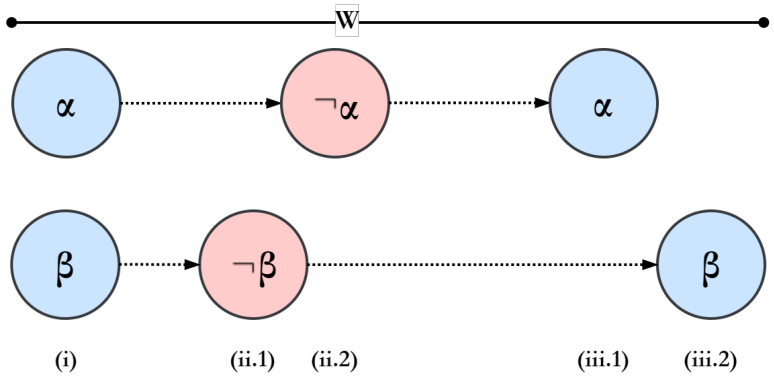

Figure 2.4: Typical BI process

$$SE(seq) := \sum_{i \in \{\alpha,\beta\}} T(i, seq) \mid_{init} \tag{2.2}$$

2.3 Evolutionary Algorithm

Evolutionary algorithms are a family of search algorithms that are inspired by biological evolution and natural selection. They are used to solve optimization problems by iteratively generating and improving candidate solutions.

The basic idea behind evolutionary algorithms is to create a population of candidate solutions, and then use selection, reproduction, and mutation to evolve the population towards better solutions. The process of evolution is modeled by creating new solutions through recombination and mutation of the existing solutions, and then selecting the best solutions for the next generation.

The most commonly used types of evolutionary algorithms are genetic algorithms, evolutionary strategies, and evolutionary programming. Genetic algorithms model the process of natural selection by using the concepts of fitness, crossover, and mutation to generate new candidate solutions. Evolutionary strategies and programming are similar to genetic algorithms, but differ in the way they handle the creation and mutation of candidate solutions.

To understand the process of GA algorithm, first of all, we have to define some basic terminals and glossaries:

- Individual: This is the basic unit that is manipulated by the algorithm. It can be used to represent every target that is going to be optimized. During the GA process, different individuals can generate new genotypes through gene exchange or mutation, which will result in individuals with new traits.
- Population: This is a set of individuals that are still alive (or haven't been removed from tournament). Mutation and crossover operations only occur on individuals contained in the population.
- Selection: This is a primary method for selecting individuals in a population, where individuals that are relatively poorer than others according to the evaluation standard are removed from the population through this process. And the standard of quality evaluation of each individual is indicated by the fitness value which should be defined by the engineer.
- Elitist: Individuals that is obviously better than others, it has less effect from aging (automatically removed from the population due to its existence for too long).

The genetic algorithm workflow can be visualized in Figure 2.5. It begins with a population of randomly generated individuals that are assessed based on their fitness function. Individuals with higher fitness scores are selected as parents for the subsequent generation, and genetic operators such as crossover and mutation are applied to create new offspring. The process of selection, crossover, and mutation is repeated for multiple generations until the stopping criteria are met [11] [12]. In this thesis, the fitness value is measured using the SE function that we proposed in the previous section, which is a multiple-point stress metric inspired by [10].

2.4 MicroGP3 Tool-Kit Introduction

MicroGP3 is an optimization tool that utilizes evolutionary algorithms to solve problems. Its framework is depicted in Figure 2.6, which includes configuration files used to define some basic configuration and the log files which is going to be preserved (ugp3.settings.xml), population settings for defining the parameters and operations of the evolutionary algorithm (ugp3.population.settings.xml), and constraints that specify the features required for each individual (ugp3.constraint.xml). For the sake of simplicity, the implementation of these files are elaborated in the chapter 3.

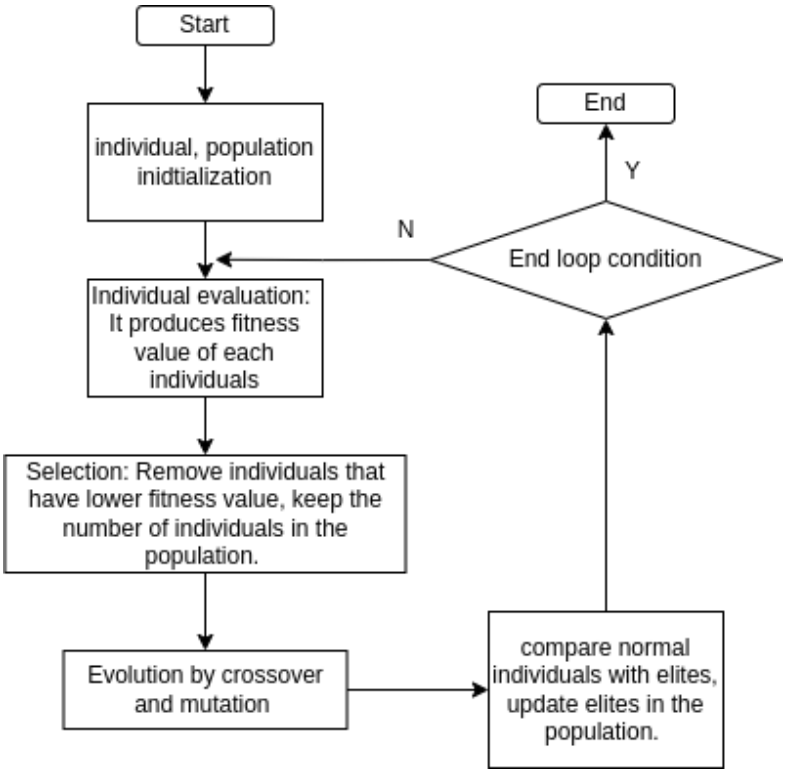

Figure 2.5: Genetic algorithm workflow

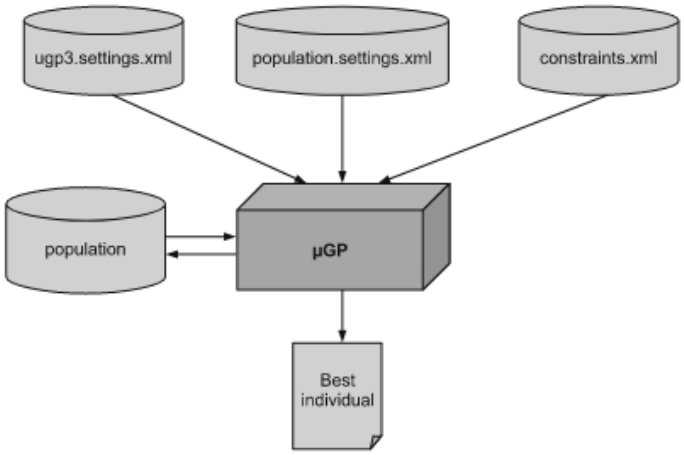

Figure 2.6: MicroGP3 framework

Chapter 3

Implementation

As the optimization process is time-consuming, it is essential to use concurrent threads to do this job. The structure of our project can be described as list 3.1. In the main directory, which is named "./ugp3" contains a public constraint file, the pair map extracted from gate level description and a run.py which is used to start the whole MicroGP3 optimization process.

Listing 3.1: MicroGP3 main folder structure

```
ugp3(main directory)
    -run.py
    -ugp3.constraint.xml
    -pair.map
    -run_dirN
        -sbst
            -individual_N.S
            -link.ld
            -Makefile
            -init_state.txt
        -ugp3.settings.xml
        -ugp3.population.settings.xml
        -ugp3.evaluator.xml
```

Since the optimization process is time-consuming, we have implemented parallel processing using the "multiprocessing" package of Python to optimize multiple node pairs defined in the "pair.map" file. The "run.py" script assigns pairs to different "run_dirN" directories (assuming the number of "run_dirN" directories is N, meaning N pairs are assigned to N "run_dirN" directories). Under each "run_dirN" directory, the "ugp3.evaluator.py" script concurrently evaluates M individuals. Therefore, the total number of concurrent threads used is $N \times M$. At the end of each batch run, N optimized individuals are generated simultaneously. And the limitation of this strategy only related to the capability of the processor. The workflow of "run.py" is shown in figure 3.1

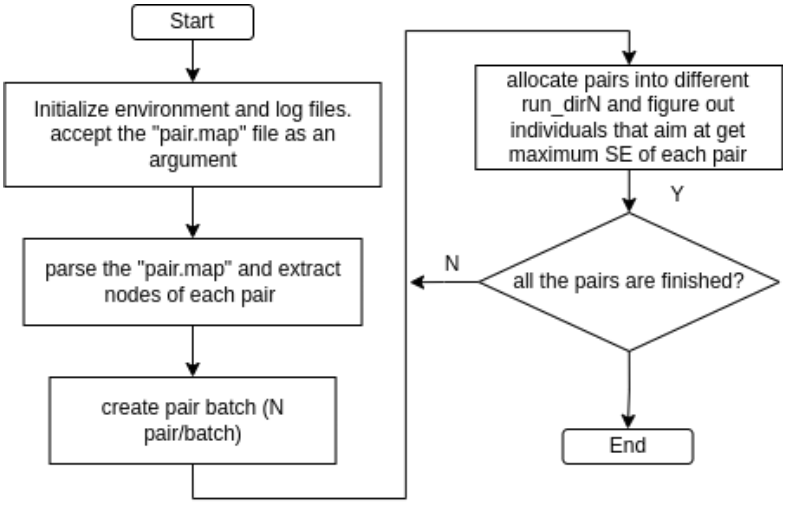

Figure 3.1: Flowchart of "run.py"

In addition, the shared constraint file also included in the main directory. For different processor, the epilogue and prologue are different. Except that, some instructions supported by the processor are different as well, so, the implementation of this file will be discussed separately later.

In the "run_dirN" directory, you can find the configuration files "ugp3.settings.xml" and "ugp3.population.settings.xml", as well as the external evaluator invoked by MicroGP3. The evaluator's primary function is to read the "individualsToEvaluate.txt" file generated by MicroGP3 and evaluate the individuals stored in the "/sbst" folder through QuestaSim. then it records the result of stress efficiency that returned from QuestaSim to "fitnessvalue.txt" which is regarded as the interface to tell the MicroGP3 which individual is better.

As the evaluator isn't provided by the MicroGP3, we designed this part by ourselves. The flowchart of ugp3.evaluator.py is shown in figure 3.2. Similar to the "run.py", this part of our system is almost the same for different processor during the experiment process. The only differences are the methods to run QuestaSim and the configurations of parallel threads of evaluation. For the RI5CY processor, the number of CPU cores that we can use is 60, therefore M in this experiment is 6 and N is 10, which means 10 pairs can be optimized in parallel. For each of these pairs, 6 individuals that are generated by MocroGP3 are evaluated concurrently. Similarly, for the IBEX processor, the number of cores that we can use is 84, so M=7 and N=12. Since QuestaSim cannot run in a single workspace, we invoke the "compile_n_testbenches.sh" script and the "make compile/questa/gate-dir DIRNAME=..." command respectively to generate the simulator workspace. The scripts used to run QuestaSim are shown in 3.2 and 3.3. The res is the result that

is extracted from the content printed on the terminal by regular expressions defined after the “grep” command.

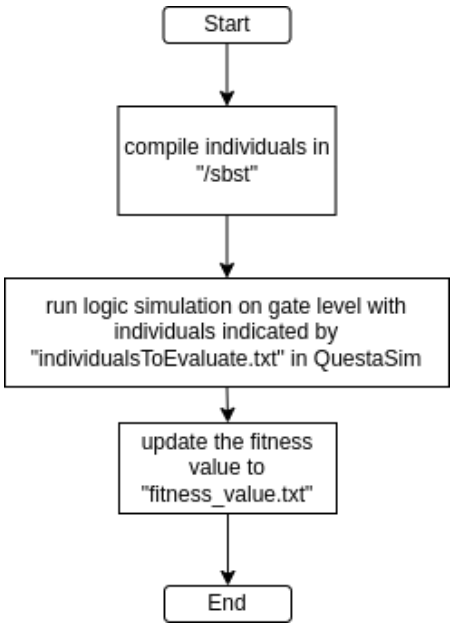

Figure 3.2: Flowchart of "ugp3.evaluator.py"

Listing 3.2: RI5CY simulator run function

```
res = os.popen("bash run_gate_nogui.sh -d %s -i %s -t %s -a %s -b %s
    <<< nn | grep -E 'STRESS EFFICIENCY=' | grep -Eo \"[0-9]+\"" % (
    run_dir, individual, target_module, net1, net2)).read()
```

Listing 3.3: IBEX simulator run function

```
res = os.popen("make lsim/gate/nogui-mps-pair-ugp3 FIRMWARE=%s
    PROBE_PAIR=%s DIRNAME=%s <<< nn | grep -Eo '(Current MAX SE:
    [0-9]+)|(Maximum SE achieved) [0-9]+' | grep -Eo  [0-9]+" % (
    individual, net_pair, run_dir)).read()
```

In both experiments, we gradually optimized the running scripts and made minor adjustments based on the experimental simulation environment and our operating habits. As a result, the command that triggers the program to start running has been slightly modified. In the case of the RI5CY experiment, the QuestaSim simulation is triggered by the bash script "run_gate_nogui.sh", which accepts the QuestaSim run directory, individual, the name of the functional unit, and the pair that we are going to measure. However, the "pair.map" file and the functional unit that contains the node pair in this file are not explicitly indicated. Therefore, in the "run.py" script for RI5CY, we have to provide the pair map and

the target functional unit for research. The corresponding command is shown in Listing 3.4.

Listing 3.4: RI5CY MicroGP3 process run command

```
python3 run.py {pair.map} {name_of_functional_unit}
```

But in IBEX experiment, we used a makefile to trigger the QuestaSim simulation. Additionally, since the path of the node in QuestaSim is already included in the pair.map, only the "pair.map" needs to be specified in "run.py" for IBEX. The corresponding command is shown in Listing 3.5.

Listing 3.5: IBEX MicroGP3 process run command

```
python3 run.py {pair.map}
```

3.1 RI5CY Stimuli Generation

3.1.1 Node Pair Extraction Strategy

Since we only have the gate level description of the RI5CY processor and it lacks distance information, we used QuestaSim to extract nodes contained in the targeted functional unit. To create node pairs, we developed a script that randomly combines these nodes and generates a pair map for different functional units. If the number of nodes is odd, the remaining node is appended to the first pair, which means that there will be a triplet inside the pair map. And in the following process, this triplet's maximum stress efficiency is 6 rather than 4. code of pair generator is shown in listing 3.6

Listing 3.6: Gate level description random node pair generator

```
import os
import sys
import random

def main():
    # with open("")
    if sys.argv[1]=="-h":
        print("\n**********************  HELP DOC **********************")
        print("param2: map file name RI5CY_xxxx_enumerate.map")
        print("param3: out file name RI5CY_XXXX.map\n")

    elif len(sys.argv)==3:
        with open(sys.argv[1], "r") as enumerate_map:
            enumerate_map_list=enumerate_map.readlines()
        with open(sys.argv[2], "w") as output:
            while len(enumerate_map_list)>0:
```

```
17                  # get net couple
18                  if len(enumerate_map_list)%2==0:
19                      pair=random.sample(enumerate_map_list, 2)
20                      enumerate_map_list.remove(pair[0])
21                      enumerate_map_list.remove(pair[1])
22                  elif len(enumerate_map_list)%2==1:
23                      pair=random.sample(enumerate_map_list, 3)
24                      enumerate_map_list.remove(pair[0])
25                      enumerate_map_list.remove(pair[1])
26                      enumerate_map_list.remove(pair[2])
27
28                  # parse pair string
29                  p_lst=[]
30                  for p in pair:
31                      p_lst.append(p.rstrip("\n").split(",")[1])
32                  if len(p_lst)==2:
33                      output.write("{};{}\n".format(p_lst[0], p_lst[1])
    )
34                  elif len(p_lst)==3:
35                      output.write("{};{};{}\n".format(p_lst[0], p_lst
    [1], p_lst[2]))
36
37      else:
38          print("please input filename (RI5CY_xxxx_enumerate.map) and
    ouput file name")
39
40  if __name__=="__main__":
41      main()
```

3.1.2 Evolutionary Algorithm Constraint

"Ugp3.constraints.xml" defines the format of each individual, with three different constraints for each targeted functional unit based on their behavior.

Adder

The type definition contains two blocks. The first block (name=riscv_reg) is the register list that we use to store data used as an operand. The second block (name=adder_instructions_g1 to adder_instructions_g7) contains instructions that relate to the adder. To reduce the ugp3 processing time and search space, instructions should be as minimal as possible.

RI5CY's ISA includes RV32I, RV32C, RV32M, RV32F, and PULP extension instructions. Adder-related instructions are those that contain comparison, subtraction, addition between register and register or register and immediate. Satisfied instructions mentioned in the RI5CY user manual and RISCV specification can be

classified into 7 groups from “adder_instructions_g1” to “adder_instructions_g7” according to their instruction formats. For example, p.abs compares the contents of the source register to 0 and stores -rs or rs according to the result. This process contains comparison and should be added to the satisfied instruction list. We put it in group 7 due to its format “p.abs rd, rs1”. Formats of different groups are shown in table 3.1.

Table 3.1: adder instruction group classification

Group	Format
adder_instructions_g1	xxx rd, rs1, rs2
adder_instructions_g2	xxx rd, rs1, imm[-2048, 2047]
adder_instructions_g3	xxx rd, rs1, rs2 imm[0, 31]
adder_instructions_g4	xxx rd, rs1, imm[0, 31]
adder_instructions_g5	xxx rd, rs1, imm[-32, 31]
adder_instructions_g6	xxx rd, rs1, imm[0, 63]
adder_instructions_g7	xxx rd, rs1

The constraint consists of only one section (id=main) that contains two subsections. The first subsection (id=load_subsection) initializes all registers, while the second subsection (id=main_subsection) covers all possible instruction scenarios. To ensure maximum diversity, different registers are initialized with a random number ranging from 0x00000000 to 0xffffffff. To avoid interference from other subroutines, we use only 7 tx registers and 12 sx registers. In the second subsection, expressions in macros for different formats are set according to the format of each group. Since the calculation process of random values stored in registers could update registers to a new random value, the initialization process occurs only once in each individual to simplify testing and save time. Therefore, the parameter of the macro should be set to maxOccurs=1, minOccurs=1, and averageOccurs=1. To increase the possibility of getting maximum stress efficiency and shorten testing time, 2 to 10 instructions should be arranged in each individual with maxOccurs=10, minOccurs=10, and averageOccurs=10.

Decoder

The RI5CY decoder translates the instruction memory contents into control signals and data that are transmitted to the execution and control units. To fully exercise the decoder, all instructions should be added to "ugp3.constraint.xml". However, since the constraint contains load and store instructions, we need to be careful not to interfere with the data and addresses stored in the registers. To explore the addresses that maximize switching activity for storing and loading instructions,

we defined as many addresses as possible in the item "data_mem_address" in the "typeDefinitions".

To address this, we separated the addresses stored in the registers into two groups: "riscv_reg" for normal data registers, and "riscv_reg_data_addr" for addresses defined in "data_mem_address". To save time during the evaluation process, we generated only addresses with the lowest 2 bytes equal to 0 in "data_mem_address". This way, the constraint limits the store and load instructions to specific data memory and prevents unexpected access during simulation and execution.

Listing 3.7: registers used in decoder constraints

```
<item type="constant" name="riscv_reg">
    <value>t0</value>
    <value>t1</value>
    <value>t2</value>
    <value>t3</value>
    <value>s0</value>
    <value>s1</value>
    <value>s2</value>
    <value>s3</value>
    <value>s4</value>
    <value>s5</value>
    <value>s6</value>
    <value>s7</value>
</item>
<item type="constant" name="riscv_reg_data_addr">
    <value>t4</value>
    <value>t5</value>
    <value>t6</value>
    <value>s8</value>
    <value>s9</value>
    <value>s10</value>
    <value>s11</value>
</item>
```

Table 3.2 contains the mnemonic codes for 16 groups of instructions based on their format, excluding branch and jump instructions and some instructions related to the processor's control register to avoid program control issues.

"load_subsection" for initializing registers and "main_subsection" for stress testing the processor with instruction expressions. Both of these subsections appear only once with parameters maxOccurs="1" and minOccurs="1". In each of the macros of the "main_subsection", their expression imitates the format of instructions. For example the expression of group 1 which is shown in listing 3.8. Its rd, rs1, and rs2 are registers that store random data or all 0s (in this case initialization is located at the end of the program). In the first run of this process, as seen in the real-time rundir_X.log, there were many 0s and 2s, which is likely due to the instruction searching space. To overcome this issue, we optimized the

Table 3.2: adder instruction group classification

Group	Format
adder_instructions_g1	xxx rd, rs1, rs2
adder_instructions_g2	xxx rd, rs1, imm12 [-2048, 2047]
adder_instructions_g3	xxx rd, rs1, rs2, imm5 [0,31]
adder_instructions_g4	xxx rd, rs1, imm5 [0,31]
adder_instructions_g5	xxx rd, rs1, imm6 [-32, 31]
adder_instructions_g6	xxx rd, rs1, imm6 [0, 63]
adder_instructions_g7	xxx rd, rs1
adder_instructions_g8	xxx rd, imm20
adder_instructions_g9	xxx rd, imm12(rs)
adder_instructions_g10	xxx rd, imm12(rs!)
adder_instructions_g11	xxx rd, rs2(rs1)
adder_instructions_g12	xxx rs3, rs2(rs1!)
adder_instructions_g13	xxx rd, rs1, imm5(is3), imm5(is2)
adder_instructions_g14	xxx rd, rs1, imm[0,1]
adder_instructions_g15	xxx rd, rs1, imm[0,2]
adder_instructions_g16	xxx rd, rs1, imm[-0x20, 0x1f]

parameters that govern the selection of instructions in the "main_subsection" by fixing the number of instructions to 10 (i.e., maxOccurs="10" minOccurs="10" averageOccurs="10"). This way, the appearance of excessive 0s and 2s in the program was reduced. Additionally, we increased the diversity of individuals in the population by enlarging the fitnessHole from 0.8 to 0.9 in the population settings. This allowed MicroGP3 to explore a larger space of potential solutions, resulting in a greater variety of individuals being generated and tested.

Listing 3.8: constraint example

```
<expression>
    <param ref="ins_g1"/> <param ref="rd"/>, <param ref="rs1"/>, <
    param ref="rs2"/>
</expression>
```

Load And Store Unit

RI5CY processor has a Load-Store Unit (LSU) which is responsible for managing data transfers to and from memory. This is the only way to manage data transfers. Therefore, instructions related to LSU are stored and loaded in its Instruction Set Architecture (ISA). RI5CY is a byte-addressable processor with word alignment, which means that the memory address that can be accessed must have its lowest

2 significant bits set to 00, indicating that the address ends with 4, 8, c, or 0. However, RI5CY is able to perform misaligned access, meaning access that is not aligned with natural word boundaries. Despite this, the type definition of data_mem_address limits the data memory address that can be accessed from 0x200000 to 0x240000, according to the data memory boundary.

During the initial run of this process, we used the full memory address range with the expectation of finding the best solution. However, due to the large size of the constant typeDefinition, the optimization process of stimuli took too much time, and the number of improved individuals was very high. This is not acceptable for a normal test stimuli generation process. Therefore, we reduced the number of memory addresses to 10 patterns that can toggle nodes in the LSU as much as possible. These patterns are shown in Listing 3.9. By doing this, the MicroGP3 is able to skip evaluating patterns that contain consecutive similar addresses.

Listing 3.9: data memory address

```
<item type="constant" name="data_mem_address">
    <value>0x200000</value>
    <value>0x20abcd</value>
    <value>0x21f0f0</value>
    <value>0x21a0a0</value>
    <value>0x21cc00</value>
    <value>0x22ffff</value>
    <value>0x22aaaa</value>
    <value>0x231111</value>
    <value>0x23beef</value>
    <value>0x23ffff</value>
</item>
```

For stimuli generation process of LSU, MicroGP3 presents a challenge in preventing unexpected memory access, which can lead to fatal errors during simulation. Without a mechanism in place to detect or avoid such errors before running, these faulty stimuli are unusable. For instance, loading r1 with the correct address during initialization and subsequently storing a random value in memory can cause a load instruction to retrieve an out-of-range value into r1, rendering it unsuitable for storing addresses. To address this issue, we separated registers into two groups: "riscv_reg" for storing random data and "riscv_reg_data_addr" for storing legal addresses, as illustrated below.

Listing 3.10: data memory address

```
<item type="constant" name="riscv_reg">
    <value>t0</value>
    <value>t1</value>
    <value>t2</value>
    <value>t3</value>
    <value>s0</value>
```

```
    <value>s1</value>
    <value>s2</value>
    <value>s3</value>
    <value>s4</value>
    <value>s5</value>
    <value>s6</value>
</item>
<item type="constant" name="riscv_reg_data_addr">
    <value>t4</value>
    <value>t5</value>
    <value>t6</value>
    <value>s7</value>
    <value>s8</value>
    <value>s9</value>
    <value>s10</value>
    <value>s11</value>
</item>
```

In the third part of the type definition, we classified load and store instructions into eight groups based on their format and function. These instructions have four different types of formats:

- xxx rd, imm12(rs)
- xxx rd, imm12(rs!)
- xxx rd, rs2(rs1)
- xxx rd, rs2(rs1!)

To ensure maximum toggling activity in the shift register of LSU, it is important to initialize the bytes with random values before executing load instructions, as the source data of these instructions come from memory, and if a memory cell is empty, the corresponding node in the shift register won't toggle. Thus, to address this issue, we have divided load and store instructions into 8 groups based on their format and function, with 4 groups for load instructions (lsu_instructions_g13 to lsu_instructions_g16) and 4 groups for store instructions (lsu_instructions_g9 to lsu_instructions_g12).

Comparing to the previous constraint, the new constraint consists of two sections. The first section (di=init) contains only one subSection (di=load_subsection), which defines a single macro (id=instruction_ld). This macro initializes 19 different random values, with values assigned to data address registers taken from "data_mem_address" and other values assigned randomly from the range of 0x00000000 to 0xFFFFFFFF. The reason for this constraint is to ensure that the initialization position is not placed at the end of the assembly, which could lead to unexpected load and store operations and fatal errors.

Table 3.3: LSU instruction group

Group	Format
lsu_instructions_g9	xxx rd, imm12(rs)
lsu_instructions_g10	xxx rd, imm12(rs!)
lsu_instructions_g11	xxx rd, rs2(rs1)
lsu_instructions_g12	xxx rd, rs2(rs1!)
lsu_instructions_g13	xxx rd, imm12(rs)
lsu_instructions_g14	xxx rd, imm12(rs!)
lsu_instructions_g15	xxx rd, rs2(rs1)
lsu_instructions_g16	xxx rd, rs2(rs1!)

The section with id = "main" contains all possible formats and instructions that may appear in the assembly code. The store instructions do not require any additional initialization or manipulation. However, for load expressions, a random value must first be loaded into memory. Therefore, in each of these expressions, there is an "sw" before the "load" instruction, and this store-load pair uses the same address. An example of these two different expressions is shown in 3.11 and 3.12.

Listing 3.11: store instruction constraint

```
<expression>
    <param ref="ins_gX"/> <param ref="rd"/>, 0(<param ref="rs"/>)
</expression>
```

Listing 3.12: load instruction constraint

```
<expression>
    sw <param ref="rd"/>, 0(<param ref="rs"/>)
    <param ref="ins_gX"/> <param ref="rd"/>, 0(<param ref="rs"/>)
</expression>
```

3.1.3 Evolutionary Algorithm Setting And Population Setting

The population setting is designed according to the structure and operations that we want to apply on the constraint. The constraint of decoder and adder contain the operation of vertices and subsections.The structure is shown below

- section(main)
  - subSection(load_subsection)
  - subSection(main_subsection)

Macros defined in the subsection will appear in the individuals generated by the algorithm. The population setting configuration defines the behavior of these elements (subsection, macro) manipulated by genetic algorithms. To operate on these two subsections, there are two operators towards subGraph. It should be noted that onePointImpreciseCrossover and twoPointImpreciseCrossover actually do the same thing compared to the precise version because there aren't any "outerLabel" parameters in the constraint. Similarly, "combinatorial" parameters do not appear in the constraint; therefore, inverOverCrossover operator doesn't work. However, we didn't remove them because they didn't have any effect on the final result. During the experiment with a superset of satisfied operators, we could know if there are any other possibilities to optimize the result. So in this experiment, we just removed some operators that are illegal to the current constraint.

Similarly, structure of LSU constraint are listed below:

- section(init)
  - subSection(load_subsection)
- section(main)
  - subSection(main_subsection)

In this way, the location of register initialization and main segment is fixed. This constraint ensures correct operations on the address of store in the individual. Since the expected algorithm behavior of each functional unit is very similar, we used the same operators for them. The operators that work in the genetic process are shown in Table 3.4.

To generate the optimum individual in this project, we increased the diversity of its population. We adjusted two parameters in this file to achieve this – "tournamentFitnessHole" and "cloneScalingFactor". During every tournament process, fitnessHole was assigned to 0.9 which means that 90% of tournament comparisons invoke diversity as standard to choose the best individuals. CloneScalingFactor was assigned to 0 which indicates that fitness value of those individuals generated by cloning will be 0 and naturally being removed from the current tournament. We also tried modifying their values during the experiment to get a better result.

Finally, in the "evaluation" segment, we use ugp3.evaluator.py as our external evaluator. We name the generated individuals X_mps.S and store them in the sbst_asm folder. We use fitness_value.txt as the fitness value for this process.

Table 3.4: Operator list

onePointCrossover
twoPointCrossover
singleParameterAlterationMutation
insertionMutation
removalMutation
replacementMutation
scanMutationCONSTANT
alterationMutation
subGraphInsertionMutation
subGraphRemovalMutation
subGraphReplacementMutation
allopatricDifferential
inverOverCrossover
onePointImpreciseCrossover
twoPointImpreciseCrossover
uniformCrossover
localSearch

3.2 IBEX Stimuli Generation

3.2.1 Node Pair Extraction Strategy

The IBEX pair extraction strategy is a bit different from RI5CY. The goal of this experiment is to detect the short wires that might occur during its early stage between two nearest nodes. First of all, as in the experiment of RI5CY, we need the gate level description of the processor and then generate nodes that are included in the specific functional unit. However, rather than providing adder module description, the source code of IBEX directly fuses its 32 bit adder with other ALU functional modules. Therefore, the first thing we have to do is to modify the RTL description of IBEX (we created a new RTL description file named ibex_adder.sv, and invoked this module in ibex_alu_modified.sv, then substitude this file with the old ibex_alu.sv), so that we can extract all the related nodes of adder by using QuestaSim conveniently, the tcl script we used in QuestaSim GUI to generate the node is shown in listing 3.13.

Listing 3.13: functional unit's node extraction

```
1 toggle report -all -instance {functional_unit} -file {
    node_list_report.txt}
```

We obtained the physical layout information report for the processor, ibex_top.def,

using Cadence. Based on this report, we designed a parser (its flowchart is shown in Figure 3.3) to determine node distances. The parser has three distance calculation modes: "relaxed", "accurate", and "insane".

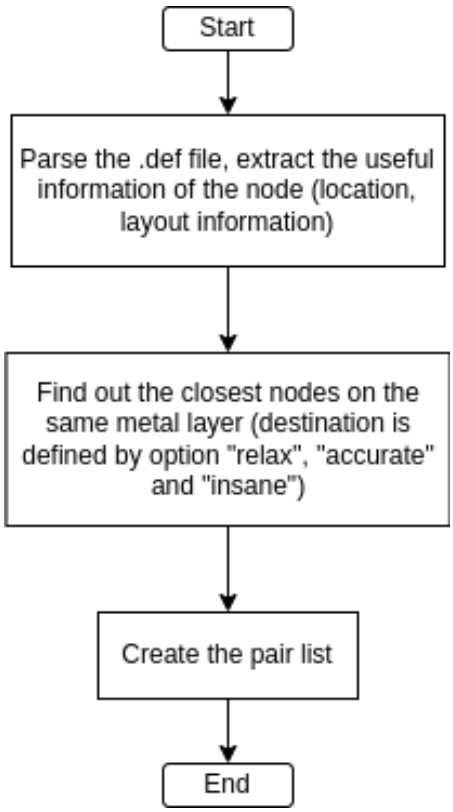

Figure 3.3: flow chart of parser

The relax mode sets the position of the node to its starting point, the distance of two node we defined is the euclidean distance between their starting points on the same metal layer, Figure 3.4 shows an example of this scenario. Assume N0, N1, N3 on the same metal layer, N2 on another layer. Obviously, for N0 the relationship of distance between nodes is $L0-2 < L0-1 < L0-3$, but since N2 and N0 aren't on the same layer, the closest node is N1, thus N0-N1 is a node pair that can be added to the list.

However, it should be noted that the "relaxed" mode cannot accurately measure the distance between wires, which is where a short circuit may occur. The "insane" mode, it's more precise than the previous one. It splits the wire into several segments and compares the starting point of each segment. The two nodes with the shortest distance defined in this way are the easy-to-short-circuit node pairs. The two starting points with the shortest distance in this process are the possible short-circuit locations in the actual circuit. For example, in the Figure 3.5, SN0-1,

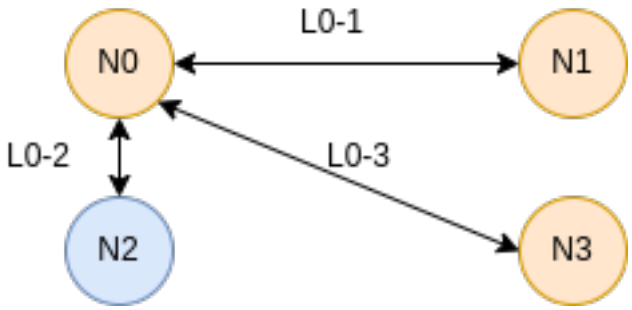

Figure 3.4: relax mode

SN0-2, SN2, SN1 are starting points of the segment on the wire. The distance between SN0-1 to SN1 is less than the distance between SN0-2 to SN2. Therefore, the couple of N0 is N1. In this experiment, we used insane mode to generate the node pair list of IBEX.

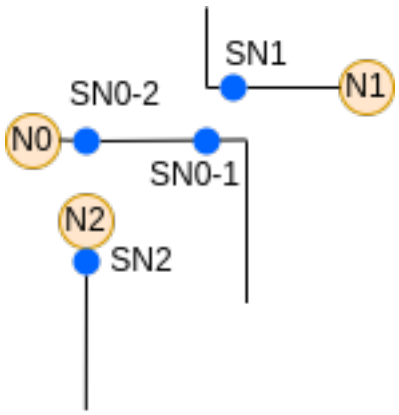

Figure 3.5: insane mode

3.2.2 Evolutionary Algorithm Constraint

As we mentioned in the previous chapter, the constraint defines the format of individuals. In this experiment, we are going to generate stimuli of IBEX adder, compressed decoder, decoder, and load and store unit. Different from RI5CY, IBEX processor only support RV32I, RV32M, and RV32C [13].

Adder

The constraint for the IBEX adder consists of three sections: "load_section", "main", and "label_section", with a fixed relative positioning that cannot be altered by the

operator in MicroGP3. This ensures that initialization errors do not occur after running the constraint. Additionally, the constraint includes a label subsection that handles branch and jump instructions, allowing for the inclusion of more available instructions. This also prevents infinite loops in the stimuli, regardless of the result of the branch instruction.

- load_section
  - load_subsection
- main
  - main_subsection
- label_section
  - label_subsection

Instructions related to adder can be divided into 5 groups according to their format, as shown in the Table 3.5. According to the user manual of IBEX, the operands of instructions in "adder_instructions_g2" are signed number, on the contrary, other instruction groups are unsigned number. Therefore we separated the registers into two groups, one of them contains signed register which means that the operand stored in these registers are regarded as signed number. In the other group, they stored unsigned number. The register's "typeDefinitions" is shown in listing 3.14.

Table 3.5: IBEX adder instruction group

Group	Format
adder_instructions_g1	xxx rd rs1 rs2
adder_instructions_g2	xxx rd rs1 imm12 [-2048, 2047]
adder_instructions_g3	xxx rd rs1 rs2
adder_instructions_g4	xxx rs, label

Listing 3.14: ibex adder register type definition

```
<item type="constant" name="signed_riscv_reg">
    <value>t0</value>
    <value>t1</value>
    <value>t2</value>
    <value>t3</value>
    <value>t4</value>
    <value>t5</value>
    <value>t6</value>
```

```
9       <value>s0</value>
10      <value>s1</value>
11  </item>
12
13  <item type="constant" name="unsigned_riscv_reg">
14      <value>s2</value>
15      <value>s3</value>
16      <value>s4</value>
17      <value>s5</value>
18      <value>s6</value>
19      <value>s7</value>
20      <value>s8</value>
21      <value>s9</value>
22      <value>s10</value>
23      <value>s11</value>
24  </item>
```

To increase the possibility of achieving a stress efficiency of 4, we fixed the number of instructions that appear in the main section to 10, given that this is the maximum length allowed. This was reflected in the XML code using the attributes maxOccurs="10", minOccurs="10", averageOccurs="10", and sigma="0".

Compressed Decoder

Figure 3.6 illustrates the structure of the IF stage, which includes an embedded compressed decoder functional unit, known as RV32C in RISC-V ISA (Instruction Set Architecture). This unit is responsible for handling compressed instructions, expanding them to an uncompressed version, which is then sent directly to the Instruction Decoder (ID) stage. This ensures that the decoder always processes uncompressed instructions. [13]. As the same as adder's constraint, compressed decoder's included 3 sections as well, as shown below:

- load
  - load_subsection
- main
  - main_subsection
- label_section
  - label_subsection

According to this description, instructions that should be included in the constraint are the compressed instructions i.e. RV32C. Instructions related to this functional unit can be divided into 13 groups, which is listed in the Table 3.6

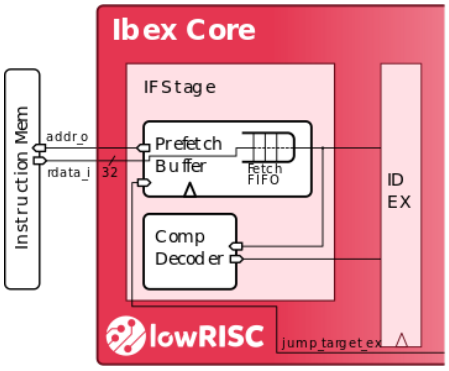

Figure 3.6: instruction fetch stage structure

Table 3.6: IBEX compressed decoder instruction group

Group	Format
compressed_decoder_instructions_g1	xxx rd, uimm5*4(x2)
compressed_decoder_instructions_g2	xxx rd', uimm5*4(rs')
compressed_decoder_instructions_g3	xxx rs', label
compressed_decoder_instructions_g4	xxx rd, imm6[-0x20, -0x1][0x1, 0x1f]
compressed_decoder_instructions_g5	xxx rd, uimm6[0x1, 0x1f]
compressed_decoder_instructions_g6	xxx x2, imm6*16[-0x200, 0x1f0]
compressed_decoder_instructions_g7	xxx rd', x2, imm6*4[0x0, 0x3fc]
compressed_decoder_instructions_g8	xxx rd', uimm6[0,63]
compressed_decoder_instructions_g9	xxx rd', uimm6[-32,31]
compressed_decoder_instructions_g10	xxx rd, rs
compressed_decoder_instructions_g11	xxx rd', rs'
compressed_decoder_instructions_g12	xxx
compressed_decoder_instructions_g13	xxx label
compressed_decoder_instructions_g14	xxx rd,rs,imm12 [-2048, 2047]
compressed_decoder_instructions_g15	xxx rd, rs1, rs2

As for the RISC-V specification, rd' and rs' are commonly used registers for compressed instructions, such as s0, s1, a0, a1, a2, a3, a4, a5. In our experiment, we classified the registers we used into four groups based on their usage for signed/unsigned and normal/popular registers.

In addition to fixing the number of instructions in the main section, we also created three immediate items to handle the compressed store and load instructions based on past experiences. These items were designed to mimic the address and address bias, with "uimm5_4" representing the bias of the load and store address, and "imm6_16" and "imm6_4" serving as biases for c.addi16sp and c.addi4spn,

respectively. IBEX and RI5CY are quite similar, allowing for misalignment, the address in the register are random values but the values of these items were set accordingly to satisfy the format of instruction, as shown in Listing 3.15.Decoder instruction can be classified into 8 groups, which is shown in the Table 3.7.

Listing 3.15: bias and address immediate patterns

```
<item type="constant" name="uimm5_4">
    <value>0x20</value>
    <value>0x34</value>
    <value>0x58</value>
    <value>0x7c</value>
</item>
<item type="constant" name="imm6_16">
    <value>-0x120</value>
    <value>0x1d0</value>
    <value>-0x30</value>
    <value>0xc0</value>
</item>
<item type="constant" name="imm6_4">
    <value>0x120</value>
    <value>0x1d4</value>
    <value>0x38</value>
    <value>0xcc</value>
</item>
```

Table 3.7: IBEX decoder instruction group

Group	Format
decoder_instructions_g1	xxx rd, rs1, rs2
decoder_instructions_g2	xxx rd, rs1, imm12 [-2048, 2047]
decoder_instructions_g3	xxx rd, imm12(rs) [-2048, 2047]
decoder_instructions_g4	xxx rs1, rs2, label
decoder_instructions_g5	xxx rs1, label
decoder_instructions_g6	xxx rd, uimm20
decoder_instructions_g7	xxx rd, csr, rs
decoder_instructions_g8	xxx rd, csr, uimm5
decoder_instructions_g9	xxx

Decoder

The Instruction Decode (ID) controls the overall decode/execution process. It contains the muxes to choose what is sent to the ALU inputs and where the write data for the register file comes from. A small state machine is used to control

multi-cycle instructions (see Ibex Pipeline for more details), which stalls the whole stage whilst a multi-cycle instruction is executing.[13] The decoder is responsible for accepting uncompressed instructions from IF stage and issues appropriate control signals to the other blocks to execute the instruction. Therefore the constraint should contain as much uncompressed instruction as possible i.e. RV32I and RV32M. In addition, the structure of the individual can be the same as the compressed decoder.

The registers that serve the decoder instructions use a mix of representations for signed and unsigned numbers. This is because, in the case of the BI test, the result of the calculation is not relevant; what is important is the activation of the transistors in the processor. Therefore, the register initialization is designed to follow the pattern shown in 3.16 and 3.17. During the running process of the stimuli, the numbers in the registers will be interpreted as either signed or unsigned based on the instruction being executed. Additionally, any errors or calculations that occur provide an opportunity to toggle the transistors that can't be affected by correct operations.

Listing 3.16: decoder register initialization

```
<item type="constant" name="riscv_reg">
    <value>t0</value>
    <value>t1</value>
    ...
    <value>t6</value>
    <value>s0</value>
    <value>s1</value>
    ...
    <value>s11</value>
</item>
```

Listing 3.17: macro of register initialization

```
<macro id="instruction_ld">
    <parameters>
        <item name="load_number1" type='integer' minimum="0" maximum
    ="4294967295"/>
        <item name="load_number2" type='integer' minimum="0" maximum
    ="4294967295"/>
        ...
        <item name="load_number19" type='integer' minimum="0" maximum
    ="4294967295"/>
    </parameters>
    <expression>
    li t0, <param ref="load_number1"/>
    li t1, <param ref="load_number2"/>
    ...
    li t6, <param ref="load_number7"/>
```

```
13      li s0, <param ref="load_number8"/>
14      li s1, <param ref="load_number9"/>
15      ...
16      li s11, <param ref="load_number19"/>
17      </expression>
18  </macro>
```

To handle instructions related to CSRs i.e. "decoder_instructions_g7" and "decoder_instructions_g8", we included CSR read and write instructions in our test. In order to prevent unpredictable behavior, we just defined all read-only CSRs as the operand for these instructions, as shown in listing 3.18.

Listing 3.18: read-only CSRs type definition

```
1  <item type="constant" name="riscv_CSR_reg">
2      <value>mhartid</value>
3      <value>mimpid</value>
4      <value>marchid</value>
5      <value>mvendorid</value>
6      <value>mip</value>
7  </item>
```

Load And Store Unit

The Load-Store Unit (LSU) of the core takes care of accessing the data memory. Loads and stores of words (32 bit), half words (16 bit) and bytes (8 bit) are supported.The LSU is able to handle misaligned memory accesses, meaning accesses that are not aligned on natural word boundaries. However, it does so by performing two separate word-aligned accesses. This means that at least two cycles are needed for misaligned loads and stores.

So, the structure that we designed is as follows:

- load
  - load_subsection
- main
  - main_subsection
- label_section
  - label_subsection

In the type definition segment, we divided the registers into two groups: "riscv_addr_reg" stores the addresses, while "riscv_reg" stores the data. Since the

data memory addresses span from 0xCD080 to 0xFDFFF, we created a look-up-table for the memory addresses rather than a full map. This approach allowed us to achieve high coverage with special address patterns, while also reducing the search space of this subset compared to a full address map. As a result, it saved a significant amount of time in generating the individual. The item definition is listed in listing 3.19

Listing 3.19: memory address subset list

```
<item type="constant" name="mem_addr">
    <value>0xCD880</value>
    <value>0xCFFFF</value>
    <value>0xD0000</value>
    <value>0xE3C3C</value>
    <value>0xEAAAA</value>
    <value>0xE5555</value>
    <value>0xF5A5A</value>
    <value>0xF3333</value>
    <value>0xFCCCC</value>
    <value>0xD3C30</value>
    <value>0xF5A54</value>
    <value>0xDC3C8</value>
    <value>0xC535C</value>
    <value>0xCACAC</value>
    <value>0xFD800</value>
</item>
```

Instructions related to the LSU (Load and Store Unit) can be divided into three groups: the load group (lsu_instructions_g1), store group (lsu_instructions_g2), and load immediate group (lsu_instructions_g3). However, if the data being loaded from an address is empty, certain structures, such as the shift register used to transmit the data from memory, may not be activated. To address this issue, we combined the load and store instructions as a pair, as shown in Listing 3.20.

Listing 3.20: IBEX lsu constraint

```
<macro id="instruction_g1">
    <parameters>
        <item name="rd" type="definedType" ref="riscv_reg"/>
        <item name="rs" type="definedType" ref="riscv_addr_reg"/>
        <item name="imm12" type="integer" minimum="-2048" maximum
    ="2047"/>
        <item name="ins_g1" type='definedType' ref="
    lsu_instructions_g1" />
        <item name="ins_g2" type='definedType' ref="
    lsu_instructions_g2" />
    </parameters>
    <expression>
```

```
10          <param ref="ins_g2"/> <param ref="rd"/>,<param ref="imm12
    "/>(<param ref="rs"/>)
11          <param ref="ins_g1"/> <param ref="rd"/>,<param ref="imm12
    "/>(<param ref="rs"/>)
12      </expression>
13  </macro>
```

Due to this design, if we fix the occurrence of this pattern to 10, the total number of instructions could reach up to 20 in the worst case scenario. Therefore, in this experiment we set the parameters to maxOccurs="5", minOccurs="5", averageOccurs="5", and sigma="0".

3.2.3 Evolutionary Algorithm Setting And Population Setting

As discussed in the previous chapter, the ugp3.settings.xml file is used to set the path for "ugp3.population.settings.xml", statistics, and status logs; hence, the configuration of this file remained unmodified. In addition, the parameters for the evolutionary algorithm, such as μ, ν, and λ, remained intact in the operator. However, we removed some unnecessary operators from the population settings to streamline the process. The operators used in this experiment are shown in Table 3.8, which support all mutation and crossover behaviors for the individual functions we defined.

Table 3.8: IBEX operator list

onePointCrossover
twoPointCrossover
singleParameterAlterationMutation
insertionMutation
removalMutation
replacementMutation
scanMutationCONSTANT
alterationMutation
subGraphInsertionMutation
subGraphRemovalMutation
subGraphReplacementMutation

Chapter 4

Results

4.1 The Result Of Switching Activity

The result of these 2 experiment are extracted from different python script. The first one takes advantage of each individuals that already generated from the MicroGP3, compile them to hex file and simulated in QuestaSim. Finally, extracted the result that from out put of QuestaSim. However, in practice, this method was found to be very time-consuming. Therefore, we changed our extraction strategy for the IBEX experiment. In the IBEX experiment, we extracted result directly from statistics.xml generated by the MicroGP3, in this way, we don't need any overhead on simulation and compilation any more.

The results for the RI5CY processor are presented in Table 4.1, which shows the optimized version of our constraint. However, it should be noted that our initial constraint resulted in an even worse performance. The main issue was that when we designed the constraint for the processor, we only had access to the ISA and limited information regarding the relationship between the ISA and the real architecture. This limited information made it difficult to ensure that the constraint was accurate. To ensure accuracy, we could have investigated the RTL description and opcode definition of the architecture, or we could compare the instructions used in the SBST stimuli to identify any missing instructions.

We experienced a similar challenge in the IBEX experiment where some instructions related to specific functional units weren't completed. To improve the results, we needed to identify which pairs didn't reach an SE of 4. To accomplish this, we first examined the report generated by the statistics to identify all of the node pairs that could be improved. Next, we simulated the SBST stimuli using QuestaSim and configured the node probes using the commands shown in 4.1. Here, the FIRMWARE refers to the hex file of the SBST stimuli, while PROBE_PAIR represents the node pair we want to improve. By viewing the waveform of the node

Table 4.1: stress efficiency of RI5CY stimuli on different functional unit

Functional Unit	Stress efficiency				
	0	1	2	3	4
Adder (269)	2 (0.74%)	0	39 (14.50%)	0	228 (84.76%)
Decoder (308)	3 (0.97%)	1	73 (0.32%)	9 (2.92%)	222 (72.08%)
LSU (502)	8 (1.59%)	0	119 (23.71%)	0	375 (74.70%)

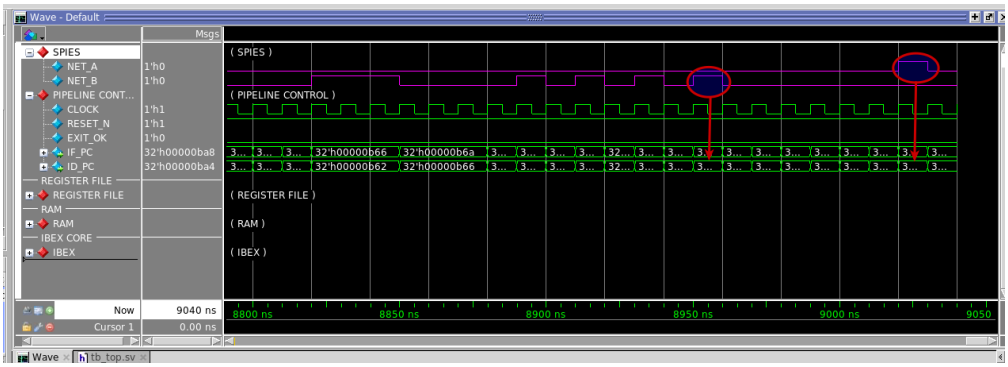

Figure 4.1: QuestaSim GUI simulation interface

pair during "sbst_stuck-at_78pct.S" execution, which is the SBST stimuli that covers 81% stuck-at-fault of the IBEX processor, we could check the address of the instruction on the ID stage and thus determine the address of the instruction that toggled the pair 4 times. The running interface is shown in 4.1, As we can see 4 toggling happens on the location which is labeled in the red circle. And these 2 toggle activity matches with the address pointed by the arrow.

Listing 4.1: SBST QuestaSim simulation

```
1 make lsim/gate/gui-mps-pair FIRMWARE=... PROBE_PAIR=..
```

Next, we disassembled the ".elf" file using the commands shown in Listing 4.2 to identify which instruction caused the toggle activity. However, it should be noted that the disassembly result is not always accurate, as some instructions in the disassembly may not actually appear in the SBST stimuli, for example "c.fld" is at addr=0x3228, but actually they can't be found in the "sbst_stuck-at_78pct.S". The running result is shown in Figure 4.2, as we can see, the number in the red square is the address that we are going to match.

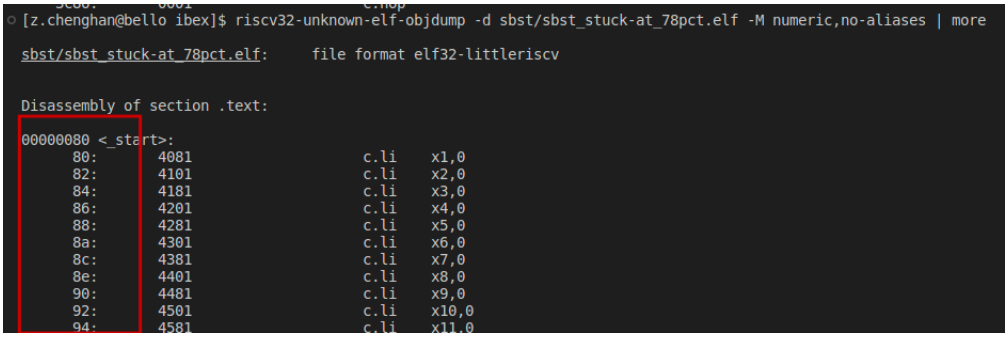

Figure 4.2: dis-assembler interface

Listing 4.2: disassembly command

```
riscv32-unknown-elf-objdump -d sbst/sbst_stuck-at_78pct.elf -M
    numeric,no-aliases
```

Finally, table 4.2 shows the improved result. As we know the "ebreak" instruction is a machine-level instruction in RISC-V architecture used for debugging and software testing purposes. When executed, the instruction causes a debug exception, which triggers a debug handler that stops the program's execution and enters debug mode. This isn't the running state we want our processor to be. therefore this instruction shouldn't appear in the constraint. Due to this reason the switching activity is lower compared to "sbst_stuck-at_78pct.S".

Table 4.2: stress efficiency of IBEX stimuli on different functional unit

Functional Unit	Stress efficiency				
	0	1	2	3	4
Adder (423)	0	0	1 (0.24%)	0	422 (99.76%)
Compressed Decoder (154)	1 (0.65%)	0	27 (15.5%)	3 (1.95%)	123 (79.87%)
Decoder (221)	11 (4.977%)	0	29 (13.12%)	4 (1.81%)	177 (80.09%)
LSU (341)	7 (2.053 %)	0	27 (7.918 %)	3 (0.88%)	304 (89.15%)

4.2 The Result Of Running Time

Table 4.3 presents the running time of the two experiments. As can be seen from the table, the decoder stimuli generation on RI5CY took much longer than on IBEX. This is because we used the emulated address in the item "data_mem_address", which resulted in a huge number of 1025 addresses that needed to be handled by the genetic algorithm, thereby leading to a large search space. However, subsequent experiments have proven that such a large search space is unnecessary. In fact, when bit-reversed patterns are included in the combinations, the algorithm is more efficient than when using random numbers. For example, combinations such as 0x353535 and 0xCACACA were found to be more effective.

The running time is also related to the simulation process, as it involves invoking a command that calls QuestaSim. In terms of the individuals of these two processors, RI5CY lacks a register initialization segment that sets all registers to 0 in its prologue, whereas IBEX has it. Therefore, the individual running duration of RI5CY is generally shorter than that of IBEX, and this may be the reason why the running time of the IBEX processor on adder and LSU is approximately twice as high as that of RI5CY, even though the searching space of RI5CY is larger than that of IBEX.

Table 4.3: Table of result

	RI5CY	IBEX
Adder	6.36h	17.77h
Compressed Decoder	-	1.07h
Decoder	171.25h	1.33h
LSU	16.35h	32.21h

Chapter 5

Discussion And Conclusions

In this thesis, we used the genetic algorithm to generate the stimuli for the BI test. It proves that with the proper definition of instructions to be used and the format of these instructions, we can generate a stimuli with the help of self-defined multi point metrics. And these stimulus are able to toggle the transistor in the processor and aging the node pairs that are most likely to experience short circuits during early use. And the statistics collected from experiment of RI5CY was sorted out and published on a paper of IEEE European Test Symposium.

Even the Evolutionary algorithm is more convenient and faster than manual generation strategy three are still some defect on constraint design:

- The first problem is that it's not able to represent the branch and jump target very well. Because the emulated random label can be duplicated when generating the individual, which may causes error when compiling the individual, or if the position can't be fixed, the process will be stuck in infinite loop. In this thesis we fixed the label at the end of this process as is mentioned before. But when the branch or jump is taken, the instruction between the branch and label will be neglected. So the mutually exclusive emulated variable is necessary for the process, in this way this segment of death code can be automatically removed by the genetic algorithm.

- The second issue with our strategy is that we used a fixed length for individuals in the previous process. Some processes use a range of lengths from a small number to the maximum number. While this strategy can increase the possibility of achieving the highest stress efficiency, we did not have complete control over the length of our individuals. To address this, we can set the length of the individual as the second fitness value.

- The third issue is the instruction we are using target for the specific functional unit. And this is also the largest problem we met in the experiment. Always,

the instruction that we used for the process isn't completed, we need to optimize our constraint comparing to the SBST stimuli. Otherwise the result might not be able to get a good result.

Bibliography

[1] D. Appello et al. «A comprehensive methodology for stress procedures evaluation and comparison for Burn-In of automotive SoC». In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE). Lausanne, Switzerland: IEEE, Mar. 2017, pp. 646–649. ISBN: 978-3-9815370-8-6. DOI: 10.23919/DATE.2017.7927068. URL: http://ieeexplore.ieee.org/document/7927068/ (cit. on pp. 1, 5, 6).

[2] Chen He. «Advanced Burn-In - An Optimized Product Stress and Test Flow for Automotive Microcontrollers». In: *2019 IEEE International Test Conference (ITC)*. 2019 IEEE International Test Conference (ITC). Washington, DC, USA: IEEE, Nov. 2019, pp. 1–6. ISBN: 978-1-72814-823-6. DOI: 10.1109/ITC44170.2019.9000147. URL: https://ieeexplore.ieee.org/document/9000147/ (cit. on p. 1).

[3] *Dynamic Burn-in | Reliability Technology Division | Services | OKI Engineering*. URL: https://www.oeg.co.jp/en/semicon/burn-in.html (cit. on p. 1).

[4] Zhenkai Ji and Wenhu Xie. «Design of FPGA Multi-Program Dynamic Burn-In System». In: *Electronics and Packaging* 22.4 (2022), pp. 72–76. ISSN: 1681-1070. DOI: 10.16257/j.cnki.1681-1070.2022.0401. URL: https://kns.cnki.net/kcms2/article/abstract?v=3uoqIhG8C44YLTlOAiTRKibYlV5Vjs7iJTKGjg9uTdeTsOI_ra5_XUd2qFe5mleAKxzFfuvMlp68Gu5tAhqr1kFHdXfO3DcE&uniplatform=NZKPT (cit. on p. 4).

[5] Ephraim Suhir. «To Burn-In, or Not to Burn-In: That's the Question». In: *Aerospace* 6.3 (Mar. 6, 2019), p. 29. ISSN: 2226-4310. DOI: 10.3390/aerospace6030029. URL: https://www.mdpi.com/2226-4310/6/3/29 (visited on 04/01/2023) (cit. on p. 5).

[6] Jens Lienig and Hans Bruemmer. *Fundamentals of Electronic Systems Design*. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-55839-4 978-3-319-55840-0. DOI: 10.1007/978-3-319-55840-0. URL: http://link.springer.com/10.1007/978-3-319-55840-0 (cit. on p. 4).

[7] Melanie Po-Leen Ooi, Zainal Abu Kassim, and Serge N. Demidenko. «Shortening Burn-In Test: Application of HVST and Weibull Statistical Analysis». In: *IEEE Transactions on Instrumentation and Measurement* 56.3 (2007), pp. 990–999. DOI: 10.1109/TIM.2007.894165 (cit. on pp. 4, 5).

[8] R.-P. Vollertsen. «Burn-In». In: *1999 IEEE International Integrated Reliability Workshop Final Report (Cat. No. 99TH8460)*. 1999 IEEE International Integrated Reliability Workshop Final Report. Lake Tahoe, CA, USA: IEEE, 1999, pp. 167–173. ISBN: 978-0-7803-5649-8. DOI: 10.1109/IRWS.1999.830588. URL: http://ieeexplore.ieee.org/document/830588/ (cit. on p. 4).

[9] T.M. Mak. «Infant Mortality–The Lesser Known Reliability Issue». In: *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*. 2007, pp. 122–122. DOI: 10.1109/IOLTS.2007.40 (cit. on p. 4).

[10] Walter Ruggeri, Paolo Bernardi, Stefano Littardi, Matteo Sonza Reorda, Davide Appello, Claudia Bertani, Giorgio Pollaccia, Vincenzo Tancorre, and Roberto Ugioli. «Innovative methods for Burn-In related Stress Metrics Computation». In: *2021 16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2021 16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS). Montpellier, France: IEEE, June 28, 2021, pp. 1–6. ISBN: 978-1-66543-654-0. DOI: 10.1109/DTIS53253.2021.9505067. URL: https://ieeexplore.ieee.org/document/9505067/ (cit. on pp. 6, 8).

[11] Ernesto Sanchez, Massimiliano Schillaci, and Giovanni Squillero. *Evolutionary Optimization: the μGP toolkit*. Boston, MA: Springer US, 2011. ISBN: 978-0-387-09425-0 978-0-387-09426-7. DOI: 10.1007/978-0-387-09426-7. URL: http://link.springer.com/10.1007/978-0-387-09426-7 (cit. on p. 8).

[12] Seyedali Mirjalili. «Genetic Algorithm». In: *Evolutionary Algorithms and Neural Networks*. Vol. 780. Series Title: Studies in Computational Intelligence. Cham: Springer International Publishing, 2019, pp. 43–55. ISBN: 978-3-319-93024-4 978-3-319-93025-1. DOI: 10.1007/978-3-319-93025-1_4. URL: http://link.springer.com/10.1007/978-3-319-93025-1_4 (cit. on p. 8).

[13] *Ibex: An embedded 32 bit RISC-V CPU core — Ibex Documentation 0.1*. URL: https://ibex-core.readthedocs.io/en/latest/ (cit. on pp. 24, 26, 29).