Politecnico di Torino

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING
Computer Systems Engineering

Master Thesis

An Evolutionary Approach for Functional Verification of RISC-V cores

Candidate
Annachiara Ruospo

Thesis Advisors
Edgar Ernesto Sanchez Sanchez
Matteo Sonza Reorda

April 2018

A mio padre, che mi ha trasmesso l'ambizione,
a mia madre, che mi ha insegnato l'umiltà,
ad Ilaria, che mi ha mostrato la bellezza.

Acknowlegments

At the end of this journey, I would like to thank several people responsible for the good success of this thesis. My personal gratitude goes to professor Ernesto Sanchez, the first guide for this project. Thank you for showing me your knowledge in the field of hardware verification and for offering me your precious human side. My sincere gratitude goes also to Professor Luca Benini. Thank you for giving me the opportunity to work with your team in Zurich and for granting me precious advice. I hope I have shown all my gratitude for this. Thank you Davide Schiavone for being the main reference point in all these months. Thanks for your patience and clarity to me, and for the precious suggestions and ideas. A sincere thank also to Florian Zaruba for advising me and helping when needed. Finally, my thankfulness goes to all my colleagues and professors who supported me in this thesis and in general in this master both morally and giving me practical help.

Abstract

Nowadays hardware devices are becoming even more complex and heterogeneous. Many Systems-on-a-Chip are composed by multi-core processors and several IP interfaces, different kind of memories and analog circuitry which communicate via many different interface protocols. On a single Integrated Circuit (IC) there are millions of logic gates shrunk on a smaller area. This means that delivering a bug-free design is more complicated than the past. To address these technical challenges, a new effort and more importance must be invested on verification processes. The proposed methodology describes a new approach on how to perform functional verification on complex pipelined microprocessor cores. Functional verification is considered as the most time consuming tasks in the design cycle since it requires a huge human and computational effort. This thesis presents a simulation-based methodology for the generation of the test set based on an evolutionary algorithm. To take advantage of the outstanding features of an evolutionary algorithm, this methodology merges generation of the test set and verification of the generated programs in a single procedure, so that the correctness of the DUV (Device Under Verification) is checked during the generation of the optimized set of programs. This allows to explore a bigger space of solutions to find hidden bugs and rarely-occurring incorrect behaviour. At first glance, it may seem a time consuming process and to some extent it actually is. However, thanks to a new discovered fitness function, simulation time has been sped up of about 20%. Experimental results are reported and described for RI5CY core, a 32-bits RISC-V ultra low power microprocessor, developed at ETH University in collaboration with University of Bologna. Actually, this master thesis has been developed for 3 months at ETH University in Zurich under the supervision of Professor Luca Benini and 3 months at Politecnico di Torino under the direction of Professor Ernesto Sanchez. Presented results show the efficiency of the methodology as well as the improvements with respect to past approaches.

Contents

List of Figures

List of Tables

Introduction

Functional verification is an important step in the development of today's complex digital systems. It is the process of checking if the implementation of a design matches its specification, by applying stimuli and observing that the results are correct. The growth of the hardware complexity increases the importance of this process even more. Today's devices are made up of millions of logic gates, multi-core processors and a multitude of interface IP, memory and other analog circuitry, integrated in a single SoC. For this reason, delivering a bug free design gets more and more complex. Moreover, this complexity can lead the functional verification to be a bottleneck in the design cycle since it can consume 60% of human and computing resources in a design [1].

This master thesis proposes a new methodology to speed up the process of verification of a hardware component and to guarantee the correctness of the specifications. It suggests a new way of generating the set of programs that are used for verification and a new way of performing verification itself. In the past programs were handwritten by specialized engineers and even if they could appear complete and accurate, they certainly lacked a large and complete space of solutions. To overcome this shortcoming, this methodology makes use of an Evolutionary Algorithm to generate the set of programs used to verify the design. An evolutionary algorithm is quite simple to set up, and requires no human intervention when running. It not only provides an effective methodology to try random modifications, but also it allows merging useful characteristics from different solutions, exploring efficiently the search space. However the main question now is: how to evaluate the performance of these programs? When is the design simulated enough? The quality of a test program is a measure of the achieved coverage of the design. Code coverage metrics belong to a class of high-level metrics used in hardware verification. They act as heuristic measures for quantifying the verification completeness and identifying imperfectly exercised design aspects [2]. These metrics measure how many parts of the code are excited while running software programs. Code coverage metrics are provided by an external evaluator and are typically expressed as a percentage of items covered. Therefore, the key point of this approach is to use an evolutionary

algorithm to generate pseudo-random assembly programs able to increase the code coverage of the design under verification.

Later, to verify the correctness of the device, an Instruction Set Simulator (ISS) has been used as a reference. It is a high level model of our device written in a high level language. Instructions are provided to both the device under verification (DUV) and this golden model, only after results are compared by a hardware module. If the value stored in the destination register is the same, simulation proceeds, otherwise this module informs about mismatches which indicate a bug in our hardware component.

A typical functional verification approach consists in generating first of all the test set and after verifying the correctness of these programs. However, the methodology proposed in this thesis breaks with this approach: it combines these two steps into a single one. Generation and verification phases are joined in a single procedure, so that the correctness of the DUV (Device Under Verification) is checked during the generation of the optimized set of programs. Even though the goal of the evolutionary algorithm is still to increase the code coverage, at the same time every single individual is subjected to verification. Through this approach several bugs have been discovered. Figure 1 shows the architecture of the method used and highlights the various blocks that make up the system.

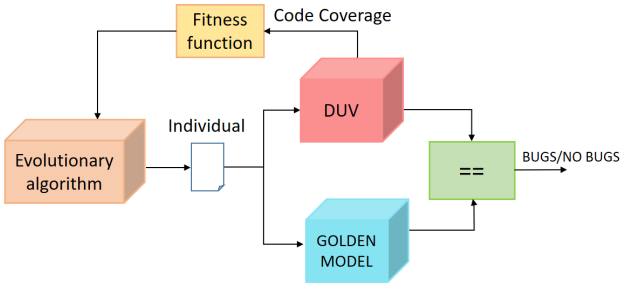

Figure 1: Proposed methodology: Parallel Generation and Verification

At first glance, it may seem a time consuming process and to some extent it actually is. However, thanks to a new discovered fitness function, simulation time has been sped up of about 20%. The fitness function is the feedback of the evolutionary algorithm core, which gives a measure of how fit the solution is.

Implementation Details

For the sake of completeness, the device under verification is RI5CY core, the main processing unit of Parallel Ultra Low Power Platforms (PULP) de-

veloped in collaboration between ETH University and University of Bologna. PULPino is the single-core System-on-a-Chip belonging to the family of PULP platforms. It is built for the RISC-V RI5CY and Zero-riscy core and its architecture is highlighted in figure 2.

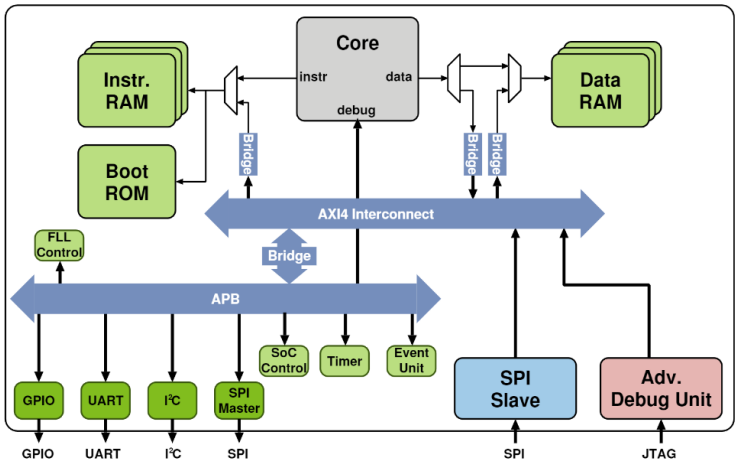

Figure 2: PULPino overview [3]

RI5CY core is a 32-bit 4-stage in-order RISC-V processor core, whose datapath is showed in figure 3. It is a very efficient core for DSP applications that can be also extended with a private Floating Point Unit.

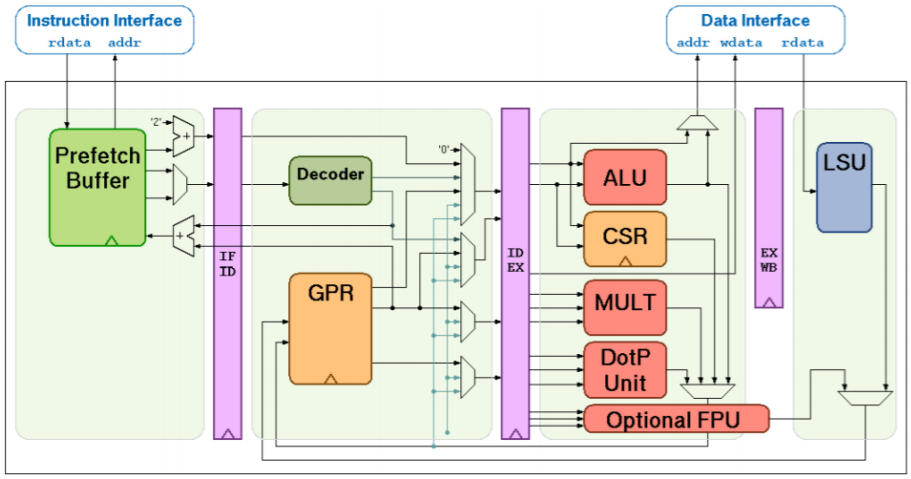

Figure 3: RI5CY core datapath [4]

RI5CY offers two area-efficient children called Zero-riscy core and Micro-riscy core, which main features are highlighted in figure 4. Zero-riscy is a 2-stage in-order 32b RISC-V processor core designed to be small and area efficient. Micro-riscy is even more smaller with only 16 registers and no

hardware multiplication support.

Core Configuration	RI5CY	RI5CY+FPU	Zero-riscy	Micro-riscy
ISA Support	RV32IMCXpulp	RV32IMFCXpulp	RV32IMC	RV32EC
Interrupts	Vectorized + Nested	Vectorized + Nested	Vectorized + Nested	Vectorized + Nested
Debug	Run, Control, Inspect	Run, Control, Inspect	Run, Control, Inspect	Run, Control, Inspect
Enhanced instructions	Hardware Loops, Post-Increment LD/ST, Bit Manipulation, Fixed Point, Packed-SIMD	Hardware Loops, Post-Increment LD/ST, Bit Manipulation, Fixed Point, Packed-SIMD	None	None
Performance [CoreMark/MHz]	3.19[1]	3.19[1]	2.44[1]	0.91[1]

[1] Custom toolchain based on GCC 5.2; Using -O2 -g -falign-functions=16 -funroll-all-loops

Figure 4: PULPino processors features [5]

RI5CY and Zero-riscy cores will be described in *Background* chapter, while Micro-riscy is not a thesis topic. For more details on Micro-riscy core, refer to PULP Platform site [6]. Even though this thesis is mainly focused on RI5CY core, some proof has been made even on Zero-riscy.

Micro Genetic Programming (μGP) is the evolutionary algorithm exploited for this project. It has been devised in 2002 in the CAD Group at Politecnico di Torino and subsequently supported by several people. Starting from an initial set of programs, also called individuals, μGP is capable of iteratively improve and evolve them, according to feedback metrics given to μGP evolutionary core. So its heuristic algorithm uses the result of the evaluations, together with other internal information, to explore the search space and to produce the optimal solutions. Figure 5 shows the architecture of μGP that will be deepened in *Background* chapter.

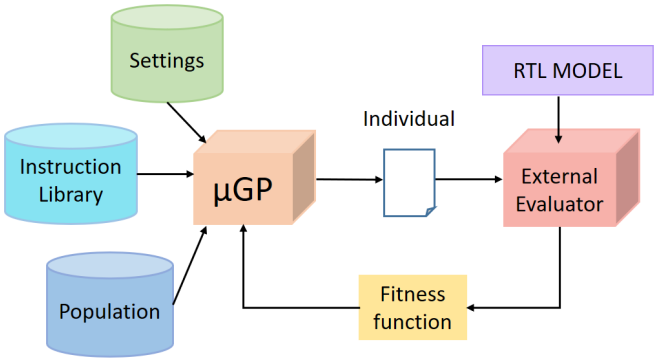

Figure 5: μGP Architecture

This thesis is organized as follows. Chapter 1 gives background notions about RISC-V architectures, focusing on RI5CY core and Zero-riscy. It explains the differences between the RISC-V Base Integer ISA and the RISC-V Extensions. Chapter 1 also offers a section which describes the concept of an evolutionary algorithm and of μGP; afterward it makes clear the importance of functional verification process in hardware design. Chapter 2 describes the proposed approach, as general as possible, in order to widen the deployment to many others hardware components. Then, Chapter 3 deepens the case study, showing the different components and describing the results obtained. Finally, Chapter 4 concludes with suggestions and advice for future work, stressing on the importance of the functional verification process.

Chapter 1

Background

The following chapter aims to explain some fundamental and preliminary concepts concerning the various areas covered by the whole project. Initially the architecture of RISC-V Instruction Set is described, highlighting the excellent characteristics of this Instruction Set Architecture (ISA) and the advantages compared to other solutions. Every RISC-V architecture must include the RISC-V Base Integer ISA, which instructions are listed in the following chapter, but several RISC-V extensions are available to improve performance. The two RISC-V architecture RI5CY core and Zero-riscy core are described. Subsequently is explained what an evolutionary algorithm is, its main properties and the one used for the following thesis, μGP (micro Genetic Programming). Finally, the last part highlights the importance of the functional verification in hardware design.

1.1 RISC-V Architectures

RISC-V is an ISA (Instruction Set Architecture) originally conceived to support computer architecture education and research, but now it is an open standard worldwide distributed. It has been developed at Berkeley into the EECS Department by a research team who was tired to work with proprietaries and complex ISAs. Except for SPARC V8, which is an open IEEE standard, most owners of commercial ISAs carefully guard their intellectual property and do not welcome freely available competitive implementations [7]. RISC-V is a foundation since 2015, made up of more than 100 members. They have access to and participate in the development of the RISC-V ISA specifications and related HW/SW ecosystem [7]. The guiding principle for the creators of RISC-V was, and still is today, to design an ISA suitable for any computer device, for this reason they aimed at separate a small base integer ISA with optional standard extensions, to support general-purpose software development. Moreover, RISC-V ISA offers also the following main features:

- Supports both 32-bit and 64-bit address space.
- Supports variable-length instruction set extensions.
- Facilitate custom ISA extensions.
- Provide efficient hardware support for modern standards, including the IEEE-754 2008 floating-point standard.

1.1.1 RISC-V Base Integer ISA

The base integer ISA is the simplest group of instructions, mandatory for every RISC-V implementation. It is restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers and operating systems [7]. In particular, depending on the width of the integer registers and on the size of the user address space, it is possible to define two primary base integer variants: RV32I for 32-bit architectures and RV64I for 64-bits ones. Of course, hardware implementations and operating systems must provide only one of these two Base Integer ISA at a time.

RV32I is made up of 47 basic instructions, eight of which are dedicated for system calls and performance counters. Being a load and store architecture, it executes operations only on registers, and transfers data to and from memory only through loads and stores. This is a prerogative of RISC architectures, which greatly reduces the complexity of a system.
RISC-V Base Integer ISA is composed by four groups of instructions:

1. **Computational instructions**: They operate on integer registers and optionally take an additional immediate operand, always signed-extended. These instructions include arithmetic, logic and comparisons on both signed and unsigned values. The first set includes additions (ADD, ADDI), subtractions (SUB) and bitwise shifts (SLL, SRL, SRA, SLLI, SRLI, SRAI), while logic instructions performs bitwise Boolean operations (AND, OR, XOR, ANDI, ORI, XORI). Comparison instructions execute arithmetic magnitude comparison (SLT, SLTI, SLTU, SLTUI). Then, there are two more special computational operations in RV32I: the LUI (load upper immediate) and the AUIPC (add upper immediate). Refer to RISC-V manual for more details.

2. **Memory Access Instructions**: They allow transfers to and from memory. There are five instructions that load a value from memory into an integer register (LW, LH, LHU, LB, LBU) and three that store a value in a register to memory (SW, SH, SB). All of these instructions use byte addresses to name memory locations; they form the address by adding the value in register rs1 to the 12-bit sign-extended immediate.

3. **Control Flow Instructions**: They are used to conditionally change the flow of control. In particular, these six instructions perform arithmetic comparisons between two registers and can transfer control to anywhere in a range of about 4KB. To form the new address, the sign-extended 12-bit immediate is added to the current program counter. BEQ, BLT, BLTU, BNE, BGE, BGEU, all of which belong to this category.

4. **System Instructions**: RV32I provides eight system instructions. The ECALL instruction is used to invoke the operating system to perform a system call while the EBREAK is used to invoke the debugger. The remaining six instructions are used to read and write the control and status registers (CSRs). CSRRW is used to overwrite the content of a particular CSR; CSRRC atomically clears bits in a CSR and CSRRS sets bits in the CSR rather than clearing them. The remaining three instructions, CSRRWI, CSRRCI, and CSRRSI, behave like their counterparts without the letter I, using an immediate instead of a register.

On the other hand, RV64I extends RV32I main features supporting other functionalities. It expands the integer register and the user address space to 64 bits. For more details, refer to RISC-V official manual [7].

1.1.2 RISC-V Extensions ISA

The RISC-V Base Integer ISA is suitable for educational purposes and for many embedded processors, but, to improve performace for computational workloads and support multiprocessors, extensions of ISA are needed. Actually RISC-V implementations support extensive customization. It is possibile to identify two kind of extensions named *standard* and *non-standard*. Standard extensions should be generally useful and should not conflict with other standard extensions while non-standard extensions may be highly specialized, or may conflict with other standard or non-standard extensions [7]. Here below standard RISC-V Extensions are listed:

- "M" Standard Extension for Integer Multiplication and Division.
- "A" Standard Extension for Atomic Instructions.
- "F" Standard Extension for Single-Precision Floating-Point.
- "D" Standard Extension for Double-Precision Floating-Point.
- "Q" Standard Extension for Quad-Precision Floating-Point.
- "L" Standard Extension for Decimal Floating-Point.
- "C" Standard Extension for Compressed Instructions.

- "B" Standard Extension for Bit Manipulation.
- "J" Standard Extension for Dynamically Translated Languages.
- "T" Standard Extension for Transactional Memory.
- "P" Standard Extension for Packed-SIMD Instructions.
- "V" Standard Extension for Vector Operations.
- "N" Standard Extension for User-Level Interrupts.

Many extensions have been already completed and delivered, while others are proposals for future work.

1.1.3 RI5CY core

RI5CY is a 32-bit 4-stage in-order RISC-V processor core. It was born from the collaboration between ETH University of Zurich and University of Bologna, in a research project that has lasted many years and continues today, aimed at developing ultra low power platforms suitable for energy-efficient computing. Actually, RI5CY is currently used as the main processing core for PULPino and the brand-new PULPissimo. These platforms are both competitive, state-of-the-art 32-bit processor based on the RISC-V architecture [6], with a rich set of peripherals and full debug support. The difference between the two is that PULPissimo has a more advanced architecture than its more basic brother PULPino [4].

RI5CY core has full support for:

- RV32I Base Integer Instruction Set;
- RV32C Standard Extension for Compressed Instructions;
- RV32M Integer Multiplication and Division Instruction Set Extension;
- RV32F Single Precision Floating Point Extension;

It supports also PULP specific extensions such as:

- Hardware Loops;
- Post-increment Load and Store Instructions;
- ALU and MAC operations;

Figure 1.1 shows the structure and the datapath of the core. The Instruction Fetch (IF) phase is in charge of providing an instruction every cycle to the Instruction Decode (ID) stage. This is helped by the presence

of a Prefetcher which takes the instructions from an Instruction Cache or Instruction Memory. Register file has 31 32-bits wide general purpose registers, from x0 to x31. Register x0 is hardwired to 0 and can only be read.

RI5CY core can be extended with a private FPU, which is capable of performing all RISC-V floating-point operations that are defined in the RV32F ISA extensions. Unlike the instructions that belong to the RV32IMCXpulp ISA, the floating point instructions have a latency (expressed in terms of clock cycles) greater than one. The FPU consists of three parts [4]:

- A simple FPU of ∼10kGE complexity, which computes FP-ADD, FP-SUB and FP-casts.
- An iterative FP-DIV/SQRT unit of ∼7 kGE complexity, which computes FP-DIV/SQRT operations.
- An FP-FMA unit which takes care of all fused operations. This unit is currently only supported through a Synopsys Design Ware instantiation, or a Xilinx block for FPGA targets.

In case the FPU is included, the register file is extended with an additional register bank of 32 32-bits registers, from f0 to f31, needed to store floating point operands or results. These registers are stacked on top of the existing register file and can be accessed concurrently with the limitation that a maximum of three operands per cycle can be read [4].

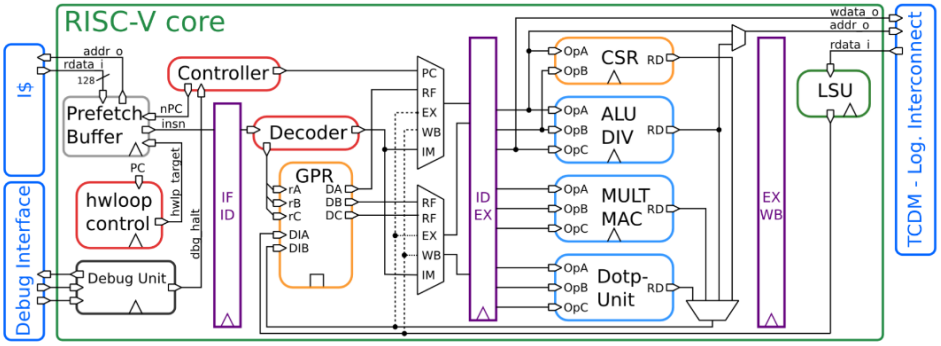

Figure 1.1: Block diagram of RI5CY core [4]

1.1.4 Zero-riscy core

Zero-riscy is a 2-stage in-order 32 bit RISC-V processor core designed to be efficient and smaller than its father RI5CY. It arises from the need to meet the greater demands of low power and low area.

Zero-riscy is the area-efficient core suitable for PULPino platform. It can be enabled by setting the flag USE_ZERO_RISCY and including the

correct compiler in the configuration file.

By setting parameters like ZERO_RV32M, ZERO_RV32E and RCV in addition to RV32I Base Integer Instruction Set, which is mandatory in every RISC-V architecture, is possible to support other three ISA configurations:

- RV32E Base Integer Instruction Set;
- RV32C Standard Extension for Compressed Instructions;
- RV32M Integer Multiplication and Division Instruction Set Extension.

Depending whether RV32E Extension is enabled, Zero-riscy has 32 or 16 32-bits wide registers, where register x0 is hardwired to 0 and can not be written.

Zero-riscy does not implement all control and status registers specified in the RISC-V privileged specifications, but is limited to the registers that were needed for the PULP system [8]. The goal is to keep the footprint of the core as low as possible and avoid any overhead.

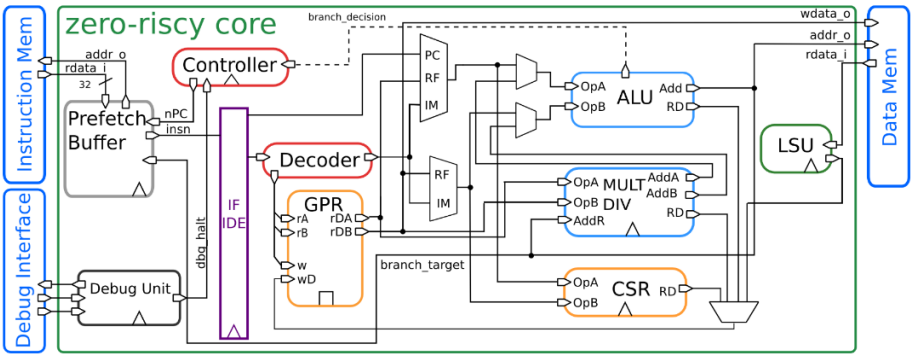

Figure 1.2: Block diagram of Zero-riscy core [8]

1.2 Evolutionary Algorithm

An evolutionary algorithm (EA) is an algorithm inspired by the theory of evolution which claims that "animals and plants have their origin in other types, and that the distinguishable differences are due to modifications in successive generations" [9]. It could lay the foundations in the Darwinian concept of reproduction and the survival of the fittest.

Evolutionary computation is a field of comuputer science in charge of solving and developing these algorithms. In every evolutionary algorithms, a single candidate solution is called *individual* and the set of all candidate solutions existing at a particular time is called *population*. It is also important to highlight that evolution proceeds through discrete steps called *generations*.

Although there are many different variants of evolutionary algorithms, the common idea behind them is the same: given a population of individuals the environmental pressure causes natural selection (survival of the fittest) and this causes a rise in the fitness of the population [10]. Over time, the best individuals will survive and evolve in that environment while the others will be, according to some parameters, discarded. This process is iterative and goes on until a solution is found or a limit is reached. Therefore, the final population will be totally different from the initial random one. It will be composed of the fittest assembly programs. An evolutionary algorithm is strongly inspired by biological mechanisms such as reproduction, mutation, recombination and selection. From this perspective, evolution is often seen as a process of adaptation and fitness function as an expression of environmental requirements and not as an objective function to be optimized. The role of a fitness function in an evolutionary algorithm is to define how good the current solution is, determining the base for selection.

An evolutionary algorithm can evolve applying some operators like *recombination* or *mutation*. The former is applied to two or more randomly selected candidates (also called parents) and produces one or more candidates (offspring); the latter is applied to one single candidate and gets one single child. When the offspring is complete, the evolutionary algorithm selects the candidates for the next generation according to the fitness parameters. Notice that the entire process is ruled by two important forces that put the basis of an evolutionary system: the variation operators and the selection. The cooperation of the two leads to improving fitness values in the next populations.

Typically an evolutionary algorithm follows this pseudocode:

INITIALISE population with random candidate solutions;
EVALUATE each candidate
while *Terminal condition is satisfied* **do**
  SELECT parents;
  RECOMBINE pairs of parents;
  MUTATE the resultig offspring;
  EVALUATE new candidates;
  SELECT individuals for the next generation;
end

Algorithm 1: EA algorithm

The algorithm above described can be also represented by the following flow-chart.

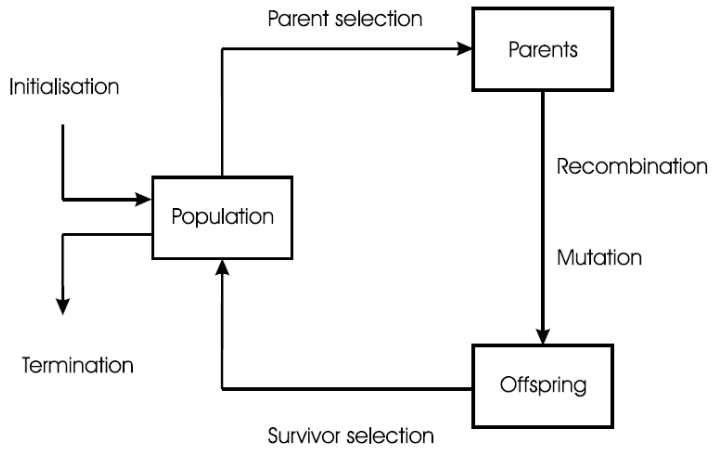

Figure 1.3: The general scheme of an Evolutionary Algorithm as a flow-chart [10]

Currently, the best known approach tackling the evolution of programs is Genetic Programming (GP), a branch of evolutionary algorithms strongly inspired by Darwinian principles and by Mendel's genetics. At the beginning of 1960's, J. H. Holland, an American scholar, discovered the similarity between adaptability of system to environment and biological evolution [11]. Therefore he suggested a computational method called genetic programming, which simulates the biologic evolution. From that time, genetic algorithm has developed very quickly and become the most widely used evolutionary algorithm [11].

1.2.1 Micro Genetic Programming

Micro Genetic Programming (μGP) is an evolutionary algorithm able to find an optimal solution to hard problems. μGP has been conceived in 2002 in the CAD Group at Politecnico di Torino and subsequently supported by several people. Although the initial idea was to generate assembly-language programs for testing different microprocessors, nowadays it has been used for many other fields such as: creation of test programs for pre and post-silicon validation; design of bayesian networks; creation of mathematical functions represented as trees; integer and combinatorial optimization; real-value parameter optimization; and even creation of corewar warriors [9].

Starting from an initial set of programs, also called individuals, μGP is capable of iteratively improve and evolve them, according to feedback metrics given to μGP evolutionary core. So its heuristic algorithm uses the result of the evaluations, together with other internal information, to explore the search space and to produce the optimal solution [12].

μGP algorithm can be thought of as composing of three different blocks: an evolutionary core, an instruction library, and an external evaluator. The *evolutionary core* is where the computation and the selection take place, where the real algorithm lies. The population is the current group of candidate solutions managed by evolutionary core; at the beginning it is made up of random programs. The user can select the initial size of the population which is always bigger or at least equal to the final one. Then, the *instruction library* is used to map individuals to valid assembly language programs [13]. It can be seen as a library including a highly concise description of the assembly syntax or parametric fragments of code. When an inidividual is ready for the evaluation, it is provided to an *external evaluator*, a tool provided by the user which includes the RTL model of the device. It drives the optimization process taking an individual as input and producing a post-elaboration report as output. Finally, this output is used to provide the evolutionary core with the necessary feedback, so closing the loop of the evolution. The post elaboration report is manipulated by a script which computes the parameters for the fitness function. Its role is to define how "fit" and how "good" the solution is with respect to the problem in consideration. The μGP algorithm selects the next generation taking into account only those individuals complying with this fitness function. Figure 1.4 highlights the general architecture of μGP.

Therefore, the algorithm adopted by μGP is comparable to the one described for general evolutionary algorithms. In each generation, starting from an initial population ν, the algorithm generates λ new individuals, the so-called offspring. After creating λ new individuals, the algorithm selects the best μ programs in the new population of $\lambda + \mu$ for survival. This process is also called survivor selection mechanism or replacement and, as opposed to parent selection mechanism which is typically stochastic, it is often de-

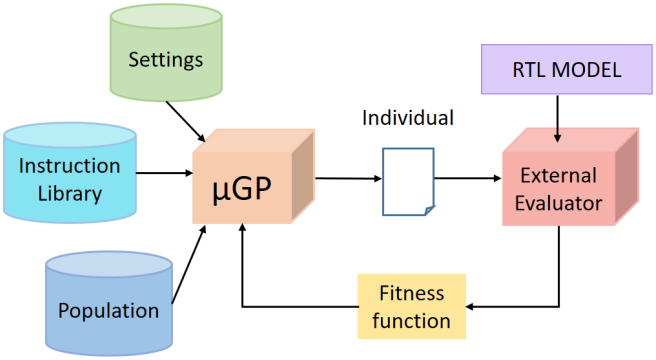

Figure 1.4: μGP Architecture

terministic. The decision is usually based on their fitness values, favouring those with higher quality, although the concept of age is also frequently used. When μGP chooses parents for the new population, it randomly selects a given number of individuals and picks the best ones among them via tournament selection. In each generation, the algorithm generates offspring by applying different operators (mutation and crossover), fully self-adapting the strength of operators.

At the end of the survivor selection mechanism, the old population is discarded and the new one is evaluated by the external evaluator, individual by individual. Process ends when a termination condition occurs. We can distinguish two cases. If the problem has a known optimal fitness level, then reaching this level should be used as a stopping condition. However, evolutionary elgorithms are stochastic and mostly there are no guarantees to reach an optimum, hence this condition might never get satisfied and the algorithm may never stop. Therefore this *terminal condition* is often extended with one of the following, that certainly stops μGP.

1. CPU reaches the maximum allowed time.
2. The total number of fitness evaluations reaches a given limit.
3. For a given period of time, the fitness improvement remains under a threshold value.
4. The population diversity drops under a given threshold.

1.2.2 Internal Individual Representation

A major issue when devising a test program generator for microprocessors is how to represent test programs in a way suitable to allow their efficient manipulation, while guaranteeing syntactical correctness [14].
Each test program, called individual, is internally represented as a Directed

Acyclic Graph (DAG) and must adhere to restricted rules. A DAG is a direct graph where there is not the possibility to create cycles or loop. It consists of many nodes and edges, where each edge is directed from one node to another one. Each node corresponds to a valid assembly instruction, where the syntax of the instruction is defined in the instruction library and the operands are represented by the parameters associated to the node.

μGP allows to organize the DAG as a collection of sub-DAGs of different type (procedures, traps, main program) called frames. Each node can have references to any other node in its subgraph or to the beginning of a different subgraph. A single sub-DAG is build with four kinds of nodes:

1. **Prologue and Epilogue:** These nodes are mandatory and must always be present. They represent required operations, such as function declarations and initializations. The prologue is the first node of the program and has no parent nodes, while epilogue is the last one and has no children. They depend both on the processor and on the operating environment and on the frame type, and they may be empty [14].

2. **Sequential Instruction:** These nodes represent common arithmetical or logical instructions. Due to conditional or unconditional branches, some of them could be unreachable. Unconditional branches are considered sequential instructions nodes.

3. **Conditional Branch:** They include all the conditional branches defined for the target assembly language and provided by the instruction library.

The representation of this DAG structure mimics the syntax of assembly programs with their jumps or sequential instructions and is highlighted in the figure 1.5. The assembly program on the right is divided in many sections, each of them linked to a valid node. For instance, the first gray section is connected to the first node (the Prologue) of the DAG representation and its content depends on the current language adopted. The second node (the red one) points to the XNOR instruction while the green node addresses a call for a subroutine. It is clear that each node can include one or more instructions, depending on its instruction library specifications.

Each DAG node contains a pointer to a specific instruction inside the instruction library. Figure 1.6 shows the internal representation of one assembly instruction, an unsigned sum with three registers (ADDCC reg1, reg2, reg3). This instruction is mapped by μGP to a sequential instruction node there the possible values for its parameters are r25, r18, and r10 and are specified in the instruction library. This bonding is useful to create a kind of evolving space for that instruction. Actually, the role of the instruction library is to constrain the limitless variety of possible individuals, making

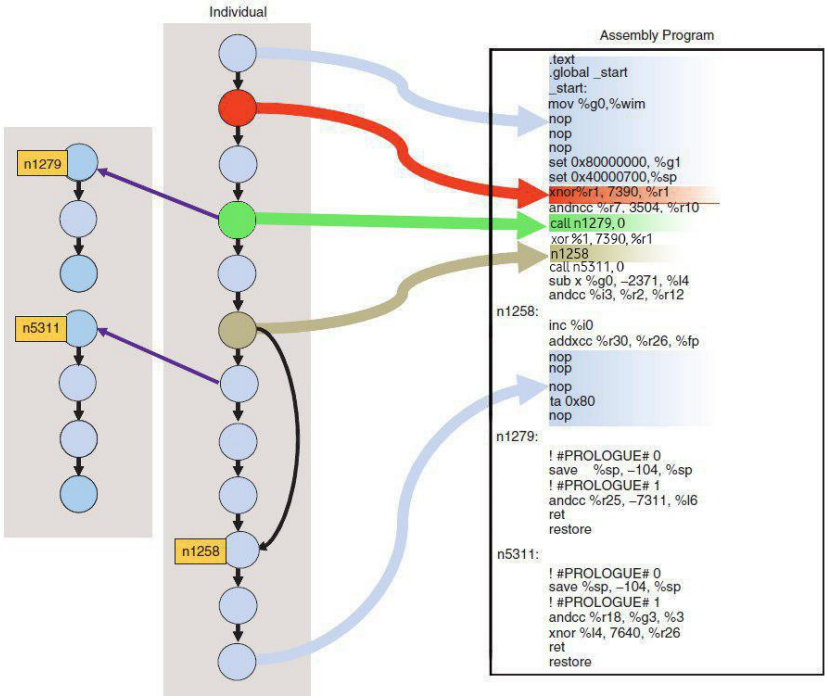

Figure 1.5: Structure of an individual

them suited to the intended application. By drawing on this library, μGP is able to evolve, searching for an individual with the best value of the adopted metrics.

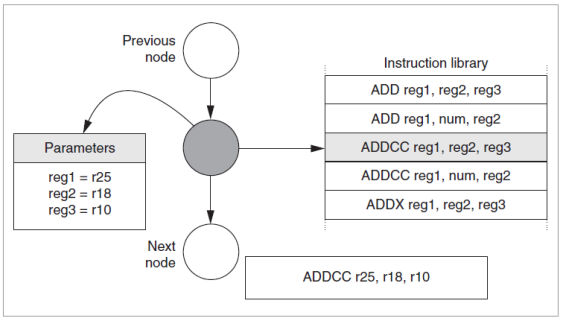

Figure 1.6: Internal representation of one instruction

1.2.3 Macro

μGP internally represents each node as a macro. A macro is a structure associated to each machine level instruction made up of two main sections:

the *expression* part which defines the syntax of the instructions and the *parameter* block, where operand's values are specified. The expression section can define not only a single instruction but also a limited sequence of instructions, where some of all parameters can change. Figure 1.7, represents the building block of the macros. The final assembly program is composed of a proper sequence of macros taken from the instruction library, each activated with proper values for its parameters [15]. The choice of the most suitable parameters value is accomplished by resorting to μGP algorithm.

```
<macro macro_attributes>
    <expression>
        ...
    </expression>
    <parameters>
        <item item_attributes />
        <item item_attributes />
        ...
    </parameters>
</macro>
```

Figure 1.7: Macro basic stucture

Macros describe the possible values that a specific node can assume. The expression part contains the text of the macro: it could be fixed or it can refer to existing parameters through the "ref" attribute. It establishes what is the current instruction and which are the source and destination registers. During the evolution, μGP will create and modify values for these components. On the other hand, the parameters block of the macro defines the type of the values. There can be integer parameters, string parameters, intra-section references and inter-section references.

1.2.4 Genetic Operators

μGP is able to generate new individuals by applying genetic operators to existing solutions. Operators are randomly selected, following the idea that those producing the best individuals should be activated more often, but no operator able to generate valid individuals should ever be completely removed from the process, no matter how bad the fitness of its children is [16]. The genetic operators applied by μGP, belong to the following classes:

- **Standard Mutations:** They are also called *single-parent operators* because work on one single parent. For instance these operators can choose randomly a single parameter inside a macro and mutate it to another possible value, or insert (or remove) a new macro instance inside a randomly selected subsection. They can repeat the mutation described more than once, depending on the current value of sigma, which is defined in population file and determines the strength of the

genetic operators. A mutation operation is always stochastic: its output (the child) depends on the outcomes of a series of random choices [10].

- **Crossovers:** They are also called *multiple-parent operators* because, as the name suggests, it merges informations from two parents into one child or two children. The principle behind is simple: by mating two individuals with different but desirable features, we can produce an offspring which combines both of those features [10].
- **Scan Mutation:** This class of genetic operator selects a variable in a macro and creates a child individual for each possible value that the parameter can assume. It must be used carefully since this operator could produce a huge number of children.

1.3 Functional Verification

Functional Verification is considered one of the most important step in hardware's design cycle. With the advent of even more complex architectures and higher performing devices this process is becoming even hard to perform. In the past three decades, as Moore's law has predicted, transistor density doubled approximately every 18–24 months. Roughly speaking, it has been maintained that the number of bugs in a new design is linearly proportional to the number of lines in its structural Register-Transfer Level (RTL) description [17], and this number is escalating at an increasing pace. This fact indicates that not only delivering a bugs free design is getting even more harder, but also that the importance of functional verification is increasing exponentially. It is considered one of the most critical task in the design cycle and sometimes it can be a bottleneck in design phases. According to Bose and Abraham, design verification represents about 60% of the total cost of the development of microprocessors or microprocessor cores. A research driven by Subhranil Deb, a senior design consultant of Synopsys India highlights that identifying a bug in the earlier phases of design process is less costly, as the figure 1.8 points out. Therefore, the need of functional verification process is strong since the first steps of design.

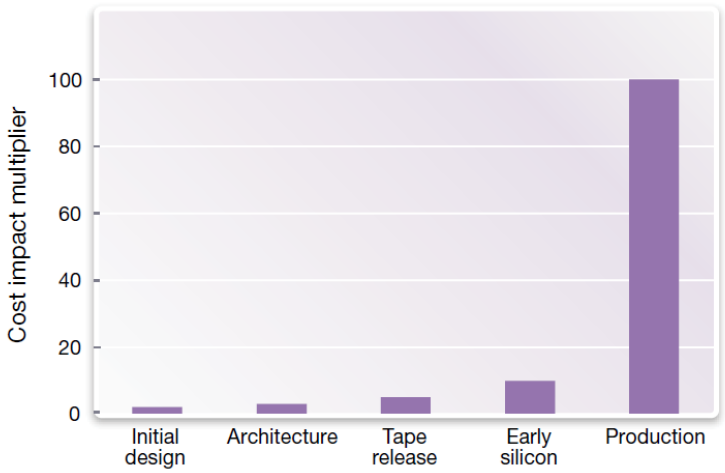

Figure 1.8: Cost impact of bugs at different phases of execution [18]

Functional Verification is the process of determining that the implementation of a design accurately represents the developer's specifications and descriptions. There are many specifications domains in integrated circuits (IC) design, like functional, timing, and electrical. Functional domain is for certain one of the most time consuming tasks in the design cycle. The description of design is simulated applying pseudo-random or deterministic stimuli, and checking the correctness of produced results with a reference

model.

1.3.1 Functional Verification and Formal Verification

Functional Verification is defined a *dynamic methodology*, also called simulation-based. It can be considered complementary to a *static methodology*, also known as formal verification, which tries to verify the correctness of a system by using mathematical proofs.

Formal methods implicitly consider all possible behaviors of the models representing the system and its specification [19]. They include many methods like model checking, equivalence checking and theorem proving. However, the completeness and the accuracy of the system, as well as the required computation resources, are a noticeable limitation. Static methods have the big drawback of requiring a huge computational resources, even for small circuits. For this reason, formal verification is typically adopted for verifying a design's single components. Formal methods for complex microprocessor designs have been targeted by several researchers in the past but, although results were considerable, all proposed techniques suffer from severe drawbacks and need considerable human efforts to handle entire designs of today's processors and advanced microarchitectural features [14].

On the contrary, simulation-based methods do not suffer from the above constraints, but hey can consider only a limited range of behaviours. For this reason they will never reach a 100% of confidence of correctness. The design is verified by applying a pattern set and responses are checked with expected ones to assure correctness. A dynamic verification approach needs stimuli, testbench and responses to be created by hand by verification engineers, or exploiting an automatic test programs generation. This class also can be divided in Intent Verification and Equivalence Verification. More details on these techniques may be found on [1].

It is well known that the real success of a functional verification approach relays on the fairness and accuracy of the inizial *functional verification plan*. Actually, it is composed of three aspects: *coverage measurement*, defining the verification problem, the different metrics to be used and the verification progress; *stimulus generation*, providing the required stimulus to thoroughly exercise the devise obeying to the directives given; and *response checking*, describing how to demonstrate the behavior of the device conform the specifications [2].

1.3.2 Code Coverage Metrics

Code coverage is one of the high level metric used in hardware verification to evaluate program's performance. Code coverage metrics are considered as

heuristic indicators for test program generation, measuring the verification completeness and identifying inadequately exercised design aspect. Roughly speaking, they measure the amount of RTL code that is exercised by running a software program. These metrics are provided by an external evaluator and are typically represented as percentages per units.

There are many available code coverage metrics but it is not possible to indicate one single metric as the most reliable [2]. To achieve a high degree of confidence, a new trend consists in combining them together to have better results.

Some well-known code coverage metrics are listed below:

1. **Statement coverage:** It is the most known coverage metric. It measures the amount of executable statements that are excited during the simulation run. Even though there are multiple statements on a single line, statement coverage considers each line individually. Actually in this example the number of statements is equal to 1.

   if RST = '0' then LD <= '0'; else LD <= '0'; end if;

2. **Branch coverage:** It counts the amount of boolean expressions and case statements that affect the control flow of HDL execution. It is sometimes named decision coverage. In the following example, therefore, to reach a 100% of branch coverage, both the explicit condition (z==0) and the implicit condition (z!=0) must be covered.

   if (z == 0) begin ... end

3. **Condition coverage:** It is considered as an extension of branch coverage. It considers the logic expressions that affect branch decisions and reports the true or false outcome of each Boolean subexpression, separated by logical-and and logical-or if they occur [2]. They measures the sub-expressions independently of each other.

   PROCESS (ina, inb, inc, ind, datin)
   BEGIN
   if ina = '1' or inb = '1' or inc ='1' or ind = '1' then
   datout <= datin;
   ELSE
   datout <= '1';
   END IF;
   END PROCESS;

   In this example, the if condition has 4 inputs. Condition coverage will consider a truth table of 16 inputs and will consider 100% of coverage only in the case of full row's coverage. However, ModelSim

Condition coverage removes unimportant conditions from the display so avoiding wasting time chasing down irrelevant conditions. This is sometimes referred to as "sensitized condition coverage" or "focused condition coverage". In FEC (Focused condition coverage), an input is considered covered only when other inputs are in their quiescent states. This means that the output must be seen in both 0 and 1 state while the target input is controlling it. If these conditions occur, the input is said to be fully covered. Then, the final coverage is given by the ratio between fully covered inputs and total number of inputs. This indicates that verification effort is greatly reduced.

4. **Expression coverage:** It is similar to condition coverage but, instead od covering branch decisions, it considers concurrent signal assignments. It builds a focused truth table based on the number of inputs of a signal assignments and uses the same techniques as condition coverage.

   internal <= a or b or c or d

5. **Toggle coverage:** It reports the number of bits that toggle at least once from 0 to 1 and at least once from 1 to 0 during the execution of a program [2]. Indeed it is a very peculiar metric used in all late stages of the design cycle, and is actually an objective measure of the activity of a design. The basic toggle coverage has been enhanced to include two modes of operation: standard and extended. Standard toggle coverage only counts transitions from low to high and from high to low. On the contrary, extended toggle coverage counts these two transitions plus the following four:

   X or Z –> 1 or H
   X or Z –> 0 or L
   1 or H –> X or Z
   0 or L –> X or Z

   This is particularly useful for examining coverage of tri-state signals and in general to give a more detailed view of verification effectiveness.

6. **FSM State coverage:** It counts the amount of states that are covered during a simulation.

   next_state <= IDLE

7. **FSM Transition coverage:** It is a measure of the amount of transitions that could occur over all possible transitions.

$$RESET <= IDLE$$

Performing a functional verification process guided by these code coverage metrics, allows reaching ample design verification reducing the redundant effort [2]. Code coverage metrics are actively used in industry due to their low set-up costs and are quite easy to manage.

1.3.3 Stimuli Generation

Stimuli generation for microprocessor cores consists essentially in assembly programs. A common goal is to generate a valid sequence of assembly instructions which are able to maximize some high level metrics or to emphasise incorrect behaviour of the microprocessor core. Even though the microprocessor verification problem has been investigated for 20 years, microprocessor verification methodologies are not mature enough to fully automate the generation of stimuli, yet [2].

This issue lies also in the complexity of today devices. Exploiting functional verification of pipelined microprocessors is a challenging task, as it is not sufficient to simply check the functionalities of all possible instructions with all possible operands. Indeed also all possible interactions between instructions must be checked, resulting in not trivial task.

As a first verification phase, it is often useful to apply handwritten programs to check the basic functionality of the core. However, these programs focus on rarely-occurring corner cases and basic specific functionalities, so they can only be exploited as a first defence against issues [14]. Furthermore, they require a deep knowledge of both the design architecture and of the Instruction set.

Nowadays, a solution for these drawbacks is to adopt an *automatic methodology* to generate these programs. Over time many authors have suggested automatic procedures to generate this test set. One simple approach adopted is named VERTIS and consists in generating random verification programs based only on the ISA of the microprocessor core. It takes as input the assembly language instruction set of the processor and the operations performed by the processor in response to each instruction and produces a functional test [20]. The final program will for certain exploit almost all the possible operands for each instructions but the final program will be very large. Random approaches have two main drawbacks: first they are quite slow and second, having no memory in the past, they may result in an explosion of lines.

Another proposed mechanism focuses on verification of the branch prediction mechanism only, for exercising the control behavior of complex microarchitectures. The goal of the technique described in this paper [21] is

to valide PowerPC604 branch prediction mechanism using a branch target address cache (BTAC) and a branch history table (BHT). However, this mechanism requires a deep knowledge in microprocessor architecture and it is extremely difficult to generalize [21].

It has been noticed that by exploiting an *evolutionary algorithm* for generating the test set greatly improves not only simulation time, but also the size of the final programs. The use of such a mechanical approach may help dramatically designers and engineers. Instead of running massive random simulations and checking the enormous mass of output data, seeking for differences with the correct model, experts may let the automatic test-case generator work for a fewdays. At the end, they just need to carefully examine the small test set produced [14]. Moreover an evolutionary algorithm is able not only to try random modification, but also to merge useful characteristics from best programs, exploring better the search space.

Chapter 2

Proposed Approach

The aim of this chapter is to describe the approach used to perform functional verification on a pipelined microprocessor. This methodology describes a procedure of automatic test-program generation based on an evolutionary algorithm, and a substantial improvement from a verification point of view. All functional verification mechanism requires a set of stimuli as input and checks responses with expected ones to assure correctness. The role of the evolutionary algorithm here is to generate these stimuli and test programs useful for verification purposes. A test program is a valid sequence of assembly instructions, that is fed to the processor through its normal execution instruction mechanism: the processor executes it as it would execute any other "normal" program [13]. The programs produced by the evolutionary algorithm are pseudo-random and are able to maximize some defined high-level metrics.

In hardware verification many metrics can be used, but for the current project code coverage metrics have been selected. Performing a verification process guided by code coverage metrics allows achieving ample design verification limiting the redundant efforts [2]. Code coverage metrics act as heuristic indicators to quantify the verification completeness and identify inadequately exercised design aspects. As supported by many authors, while a high coverage figure increases the confidence in the design correctness, it is not possible to select one single code coverage metric as the most reliable [2]. A current trend is to mix these metrics to obtain better results. Therefore, the process of program generations is guided by these code coverage metrics. The purpose of the evolutionary algorithm is to select the best individuals which are able to increase these metrics parameters.

To verify our pipelined microprocessor, in this project two main phases could be identified:

- **Generation**: The evolutionary algorithm generates a set of programs able to maximize code coverage metrics.

- **Verification**: The set of programs generated by the evolutionary algorithm is used to verify the correctness of the model. These programs are executed in parallel by the model under verification and the golden model. In case that results differ, an error is raised and the simulation can stops.

A typical functional verification approach consists in performing these two phases subsequently: generate the test set and then verify the correctness of the instructions belonging to that complete test set. Figure 2.1 sketches this first methodology.

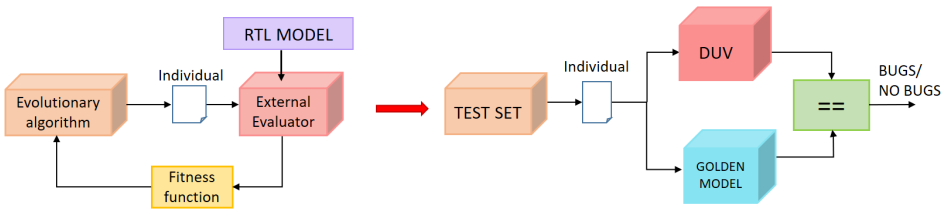

Figure 2.1: Sequential Generation and Verification phases

The methodology proposed in this thesis breaks with this approach, combining these two steps into a single one. Generation and verification phases are joined in a single procedure, so that the correctness of the DUV (Device Under Verification) is checked during the generation of the optimized set of programs. Figure 2.2 shows the architecture of the method used and highlights the various blocks that make up the system.

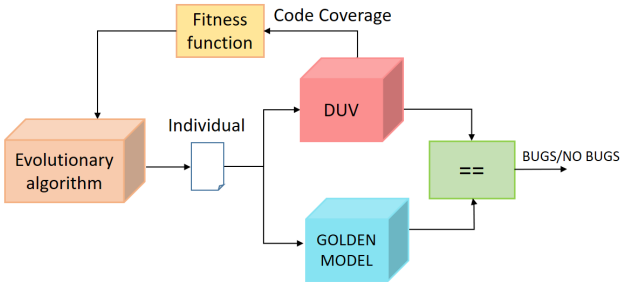

Figure 2.2: Proposed methodology: Parallel Generation and Verification

Even though the goal of the evolutionary algorithm is still to increase the code coverage, at the same time every single individual is subjected to verification. Programs are delivered to both the DUV and the golden model, which execute them and store results in the specific destination registers. At this point a *Simulation Checker* comes into play. It is an hardware module which compares the results of each instruction produced by the evolutionary algorithm with a Golden Model. If these results differ, it issues an error

warning that confirms the presence of a bug. This information is sent to the evolutionary algorithm which can decide to stop the simulation or continue and collect other bugs.

Through this approach it is possible to better explore the research space revealing the presence of hidden bugs inside the DUV. Actually, by exploiting this methodology, several bugs have been discovered in our pipelined microprocessor. Consequently, this technique not only tries to increase the code coverage of the DUV, but, at the same time ensures that the portion of code covered by the set of programs is correct.

2.1 Fitness Function

Typically approaches tackling evolutionary algorithms appear to be rather slow and therefore require a non-negligible computational effort. The more complex the model becomes, the longer the simulation is. The duration depends also on the internal evaluations that the evolutionary algorithm makes, on the number of mutations it performs, and on the amount of offspring it produces. All these decisions are made based on the fitness parameter that it receives as feedback, typically one or more values. Therefore, choosing the appropriate fitness function is essential. For instance, μGP evolutionary core accepts an array of values, but the former takes on more importance than the others: its growth is the primary goal of the algorithm.

The first fitness function was composed by the *arithmetic averages*, *variances* and *sums* of the values of code coverage metrics obtained by running an assembly program. Code coverage metrics are represented as percentages per unit and are provided by an external evaluator. For our purposes, the following ones have been selected:

- Statement Coverage;
- Branch Coverage;
- FEC[1] Condition Coverage;
- FEC Expression Coverage;
- FSM[2] State Coverage;
- FSM Transition Coverage.

[1]FEC is the acronym of Focused Expression Coverage, one metric used by ModelSim to reduce the dimension of the truth table for both condition and expression. It is described in the following chapter.

[2]Finite State Machine.

To compute the average, the variance and the sum for each of these metrics, a script elaborates the code coverage values provided by an external evaluator. By running an assembly program, this evaluator produces a report in txt format with the percentages of these metrics reached by each unit of the model (Controller, Alu, Decoder etc.). Figure 2.3 is a sketch of this report with the values of coverage for the Load and Store unit, the Multiplier and the Prefetch Buffer. For instance, to compute the arithmetic average of the statement metric, the percentage of statement coverage reached by each unit was first added up and then divided by the number of units considered. The variance of each metric has been computed by following the (2.1) equation, where X represents the percentage of coverage for a given unit and E[X] is the arithmetic average of these values. In statistics and probability theory, variance is the expectation of the squared deviation of a variable from its mean. It basically gives informations on how far a set of numbers are spread out from their average value. Finally, the sum is simply the addition of all the percentages for a given metric.

$$\sigma_x^2 = E[(X - E[X])^2] \tag{2.1}$$

The (2.2) equation represents the first fitness function. It is basically an array of 18 values: the first 6 elements are the arithmetic average of the selected metrics, the next 6 the variance and the last 6 elements are the sum of these. Refer to table 2.1. The first value of the array is a and it is the first input for the evolutionary algorithm. It will have higher relevance for the selection process since for μGP its growth has more importance than the growth of the consecutive parameters. However, it has been observed that this approach led to a discrepancy in the growth of the entire code coverage metrics. Statement average (a) grew faster than the others, which were quite stable over time.

$$OldFitnessFunction = \{a, b, c, d, e, f, g, h, i, l, m, n, o, p, q, r, s, t\}\,; \tag{2.2}$$

This methodology proposes a new fitness function with only two feedback values for the evolutionary core: as the (2.3) equation points out, the first one is the result of a weighted average of the various coverage metrics (t), while the second is a value inversely proportional to the length of the programs, so that, with the same code coverage, the program with the lower dimension is chosen. The idea from which this function derives comes from the observation of the behavior of the code coverage values that emerges from the simulation of the programs. In particular, it was observed that

```
445 ================================================================================
446 === File: /home/msc17h5/pulpino/vsim/..//ips/riscv/riscv_load_store_unit.sv
447 ================================================================================
448     Enabled Coverage            Active      Hits    Misses % Covered
449     ----------------            ------      ----    ------ ---------
450     Stmts                           99        94         5      94.9
451     Branches                       120       103        17      85.8
452     FEC Condition Terms              6         5         1      83.3
453     FEC Expression Terms            10         2         8      20.0
454     FSMs                                                      37.5
455         States                       4         2         2      50.0
456         Transitions                  8         2         6      25.0
457
458 ================================================================================
459 === File: /home/msc17h5/pulpino/vsim/..//ips/riscv/riscv_mult.sv
460 ================================================================================
461     Enabled Coverage            Active      Hits    Misses % Covered
462     ----------------            ------      ----    ------ ---------
463     Stmts                           83        83         0     100.0
464     Branches                        43        43         0     100.0
465     FEC Condition Terms              2         2         0     100.0
466     FEC Expression Terms            12         9         3      75.0
467     FSMs                                                      81.2
468         States                       5         5         0     100.0
469         Transitions                  8         5         3      62.5
470
471 ================================================================================
472 === File: /home/msc17h5/pulpino/vsim/..//ips/riscv/riscv_prefetch_buffer.sv
473 ================================================================================
474     Enabled Coverage            Active      Hits    Misses % Covered
475     ----------------            ------      ----    ------ ---------
476     Stmts                           81        79         2      97.5
477     Branches                        70        68         2      97.1
478     FEC Condition Terms             23        15         8      65.2
479     FEC Expression Terms             2         2         0     100.0
480     FSMs                                                      91.1
481         States                       8         8         0     100.0
482         Transitions                 17        14         3      82.3
483
```

Figure 2.3: An example of post-elaboration report

the growth of the metrics used was not the same during the various simulations. Statement and branch coverage are, to some extent, correlated and are quite easy to increase. A complete branch coverage implies complete statement coverage, but the first metric is slightly harder to enhance. On the contrary, condition and expression coverage are the most complex and least understood types of code coverage. Their full coverage is really hard to obtain and their increase is rather slow. The situation is even worse for the FSM state and transition coverage since they require appropriate proceedings and instructions. The goal of the proposed formula is to achieve the

Metrics	Average	Variance	Sum
Statement	a	g	o
Branch	b	h	p
FEC Condition	c	i	q
FEC Expression	d	l	r
FSM State	e	m	s
FSM Transition	f	n	t

Table 2.1: Code Coverage Metrics Manipulation

most uniform growth possible for all metrics taking into account the weight that each metric has during the simulation. Differently from the previous fitness function, only arithmetic averages are considered now to build the equation (2.4).

$$\textit{New Fitness Function}=\left\{t, \frac{1}{\textit{length}}\right\} \tag{2.3}$$

Where:

$$t = \frac{\left(\frac{a+b}{2}\right) + 2c + 2d + 3\left(\frac{e+f}{2}\right)}{4} \tag{2.4}$$

It has been demonstrated that the proposed fitness function can reduce the time required to achieve the same code coverage by approximately 20% with respect to the first one. Details and results are presented in the *Case Study* chapter.

Chapter 3

Case Study

The purpose of this chapter is to deepen the proposed verification methodology by going into detail in the various actors of the system. To evaluate the described approach, RI5CY microprocessor is targeted, a 32-bit processor core based on RISC-V architectures and extended with a private Floating Point Unit (FPU). It has been developed at ETH University of Zurich in collaboration with the University of Bologna. RI5CY core is backbone of PULP platforms but this methodology has been performed exploiting the single core SoC named PULPino. RI5CY core's main features are described in the *Background* Chapter. It is written in SystemVerilog language and is composed of a total of sixty files, twenty five [1] describing the architecture of RI5CY core and thirty five related to the Floating Point Unit. Excluding empty rows, it is made up of a total of 16026 lines of code. Refer to table 3.1.

	RI5CY CORE	FPU	TOTAL
Number of files	25	35	60
Number of lines of code	11310	4716	16026

Table 3.1: Details of the device

As described in *Proposed Approach* Chapter, code coverage metrics are used to evaluate the performace of this DUV. The complexity of the core can be represented by the maximum number of lines, branches, conditions, expressions, states and transitions that must be covered to increase the confidence in the device. Table 3.2 shows the total number of these metrics for the entire device.

[1] The debug unit and the tracer unit are excluded from these experiments.

Metric	Maximum value
Statement	3342
Branch	2082
FEC Expression	610
FEC Condition	176
FSM State	35
FSM Transition	69

Table 3.2: Complexity of the device

The evolutionary algorithm used to generate optimized programs is μGP (Micro Genetic Programming) and it has been developed in the CAD group of the Politecnico di Torino. The Instruction Set Simulator (ISS), also named golden model, is a high level model of the core written in C++. Modelsim v10.6 by Model Technology is used as external simulator and a set of scripts have been developed to allow the correct flow of the operations and also to compute the fitness function for the evolutionary algorithm.

The instruction library for RI5CY core consists of hundreds of macros, targeting instructions that belong to RV32IMCFXpulp ISAs. For sake of completeness, Xpulp indicates a set of PULP specific extensions to the base ISAs which full supports Post-Incrementing load and stores, Multiply-Accumulate extensions, ALU extensions and Hardware Loops.

The aim of the experimental evaluation is two-fold: on the one hand to increase as much as possible the code coverage of our device under verification; on the other hand to detect incorrect behaviour of the microprocessor. However at the end it will be clear that these purposes will be mixed in a single goal.

3.1 Test Set Generation

When addressing the problem of test program generation, the complexity of current microprocessors must be considered: architectural solutions are pipelined, superscalar, hyperthreaded, emulate several virtual processors, rely on several memory caching layers, and new features appear every quarter. Each of these keywords implies a complexity degree in the processor architecture, and test programs should be able to test all these advanced features [13]. It makes no sense to test individual instructions, since the context in which an instruction executes change the processor state and

modify the execution path taken by the instruction. This observation rules out exhaustive test programs, since developing and executing all possible sequences of instructions is combinatorially unpractical [13]. Following results are reported for RI5CY core extended with the FPU.

3.1.1 Handwritten programs

Manual generation of test program could be considered as an initial approach for check some specific behaviour of a microprocessor. These programs are useful for testing basic functionality that is known to be critical but can not issue a complete device check. Due to the complexity of today's microprocessors, it is impossible for the functional verification purposes, to rely completely on a test set made up of these handwritten programs. One class of manually developed test programs is called *systematic* test programs that execute an array of similar operations with small variations (e.g., to test an arithmetic unit with different values of the operands) [13].

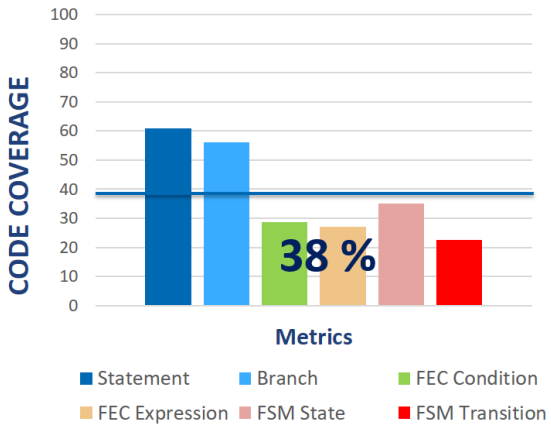

Figure 3.1: Code coverage achieved by running an Handwritten Program

It has been demonstrated that the code coverage achieved by running these programs is quite low. Figure 3.1 shows the value of coverage for each metric, obtained by running testVecArith program. This testVecArith is one of many handwritten programs included in the folder riscv_test, which have been written to test the different features of RI5CY core. In particular it focuses on vectorial ALU instructions.

The low code coverage proves that handwritten programs can be only exploited as a first line of protection against bugs, since they focus on basic functionalities and important but rarely-occurring corner cases. Due to their low code coverage, they can not be considered a good measure of confidence for device correctness. Exhaustive or nearly-exhaustive tests are

often necessary, but the effort required to generate them manually and the time required to simulate them are practically infeasible [14]. Therefore automatic test program generators may help generate random or pseudo-random solution to extend the space of solutions.

3.1.2 Random approach

Subsequently, the best random program yielded by μGP has been selected. Even though μGP is an evolutionary algorithm able to produce an optimal solution, the first set of programs that produces is totally random. As described in [2], the random instruction generator exploits the μGP instruction library to devise syntactically correct fragments of code and the μGP external evaluator to simulate them. Differently from an evolutionary approach, coverage values provided by ModelSim are not used as a feedback to optimize the candidate programs and the loop is not closed. A random program is added to the test set whenever it increases at least one coverage figure. At the end this test set will have a huge size while providing not so optimal results. Moreover, by running the best random program of the test set, it has been observed a slightly increase in the code coverage but results are still unsatisfactory.

Figure 3.2 highlights the increase in the final code coverage of the device, expecially for statement and expression coverage, while FSM transition coverage remains stable. A final 52% of average code coverage obtained by running a random program can not still be considered a significant degree of confidence.

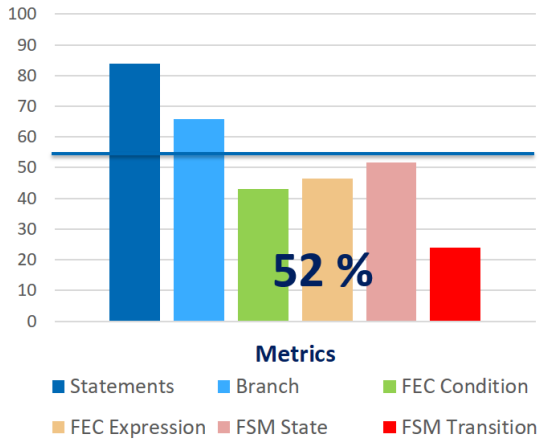

Figure 3.2: Code coverage achieved by running a Random Program

3.1.3 Evolutionary algorithm

A totally random approach is not enought to guarantee a fairly high coverage. An evolutionary algorithm allows to optimize the candidate programs using the feedback information coming from an external evaluator which is able to evaluate them with respect to coverage metrics. μGP, the used evolutionary algorithm, generates a set of correct and valid random assembly programs for use as an initial seed. Then, its goal is to optimize these programs to increase the coverage metrics. Differently from a random approach, the size of the final test set is very small, since at the end of each simulation, μGP delivers one single best optimized program.

Set up phase

To start a μGP simulation, many files and scripts have been written and some parameters have been defined. The evolutionary algorithm selects the assembly instructions from the instruction library, a kind of database which stores all the macros for the supported ISAs. This initial instruction library, named *constraintRI5CYFPU.xml*, was composed by 1613 lines of code [2] and 43 macros. It has been written to support:

1. RV32I Base Integer Instruction Set;
2. RV32C Standard Extension for Compressed Instructions;
3. RV32M Integer Multiplication and Division Instruction Set Extension;
4. RV32F Single Precision Floating Point Extensions;
5. PULP specific extensions: Post-Incrementing load and stores, Multiply-Accumulate extensions, ALU extensions, Hardware Loops.

The average number of assembly instructions for each individual is defined in *constraintRI5CYFPU.xml* file and is fixed to 150, with a maximum of 300 instructions and a minimum of 100.

The *population.settings.xml* file contains details on the current population of programs; by setting few parameters it is also possible to establish the trend of the simulation. For instance, the maximum number of generation is set to 100 and the maximum number of steady state generation is fixed to 50. It means that if the best fitness value does not change for 50 consecutive generations, evolution stops. The size of the inizial random population (ν) and the maximum size of successive populations (μ) is equal to 30, while the numbers of genetic operators applied at every step of the evolution (λ)

[2] Excluding empty rows

is fixed to 20. At the beginning and at the end of each generation, there will be exactly μ individuals and μ must be lower or equal to the initial size of population ν. Besides, note that each genetic operator may create more than one individual, so the offspring size at each generation is usually bigger than λ. Therefore, to limit the computational time, the maximum number of individual that can be evaluated has been fixed to 1200. Note that by setting these constraints, a standard μGP simulation lasts from 9 to 14 hours. By incresing program's size, λ, or the amount of individuals which can be evaluated, simulation time grows rapidly. The other parameters have been left to standard values.

One more important file is the *fitness.sh* script which is in charge of closing the loop of the simulation and creating the fitness function. It receives a post-elaboration report from ModelSim with the code coverage values for each unit. Figure 3.3 is one example of report with code coverage percentages for controller unit. The metrics considered are Statement, Branch, FEC Condition, FEC Expression, FSM State and FSM Transition, and for each of them, some informations are reported alongside (active, hits, missed and % covered). If one of these is not active, the respective code coverage value is 100% but is not considered because that metric actually misses for that unit. FEC is the acronym of Focused Expression Coverage, one metric used by ModelSim to reduce the dimension of the truth table for both condition and expression. FEC is a row based coverage metric which emphasizes the contribution of each expression input to the expression's output value [22]. It consider an input fully covered only when other inputs are in their quiescent states, meaning that the output must be seen in both 0 and 1 state while the target input is controlling it. If these conditions occur, the input is said to be fully covered. Then the final FEC coverage is the ratio between the fully covered inputs and the total number of inputs.

```
================================================================================
=== File: /home/msc17h5/pulpino/vsim/..//ips/riscv/riscv_controller.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                          197       176        21      89.3
    Branches                        96        77        19      80.2
    FEC Condition Terms             27         9        18      33.3
    FEC Expression Terms            22        12        10      54.5
    FSMs                                                        70.8
        States                      12        11         1      91.6
        Transitions                 28        14        14      50.0
```

Figure 3.3: Post-elaboration Report for Controller Unit

These values are manipulated by the script *fitness.sh* to create the feed-

back for the evolutionary algorithm μGP: the fitness function. For each code coverage metric, *fitness.sh* computes the average, the variance and the sum. As described in Chapter 2, the first fitness function was an array of these parameters, where each of them occupied a fixed position.

$$uGPfeedback = \{a, b, c, d, e, f, g, h, i, l, m, n, o, p, q, r, s, t\}\,; \tag{3.1}$$

However, it has been observed that this approach led to a discrepancy in the growth of the all code coverage metrics. Statement average (a) grew faster than the others, which were quite stable over time. Therefore, to achieve a uniform growth of the metrics, the following simulations have been performed with a new fitness function, expressed by the equation (3.2). It is made up of only two parameters: the first one is the result of a weighted average of the various coverage metrics, while the second is a value inversely proportional to the length of the programs, so that, with the same code coverage, the program with the lower size is chosen.

$$t = \frac{\left(\frac{a+b}{2}\right) + 2c + 2d + 3\left(\frac{e+f}{2}\right)}{4} \tag{3.2}$$

$$uGPfeedback = \left\{t, \frac{1}{length}\right\} \tag{3.3}$$

It is necessary to highlight that the final simulation's result does not depend on which fitness has been used. The only thinks that this function influences is simulation time, but at the end, the code coverage reached is the same. As figure 3.4 shows, the proposed fitness function saturates before than the previous one. This graph is the outcome of two different simulations executed on RI5CY core without the FPU. The blue line on the graph represents the arithmetic average of the 6 best parameters (a,b,c,d,e,f) of the first fitness function over time. On the contrary, the red line delineates the trend of the weighted average (t) of the proposed fitness function over time. The first simulation, performed with the first fitness, goes on for 19 hours and 33 minutes, while the second simulation, executed with the second fitness, holds 15 hours and 30 minutes. It is evident that both of them achieve the same final code coverage with a reduction of about 4 hours.

Moreover, it must be noticed that the red line models a growing monotone function. Being represented by the first parameter (t) of the second fitness function it could not decrease over time, at most it can remain stable. On

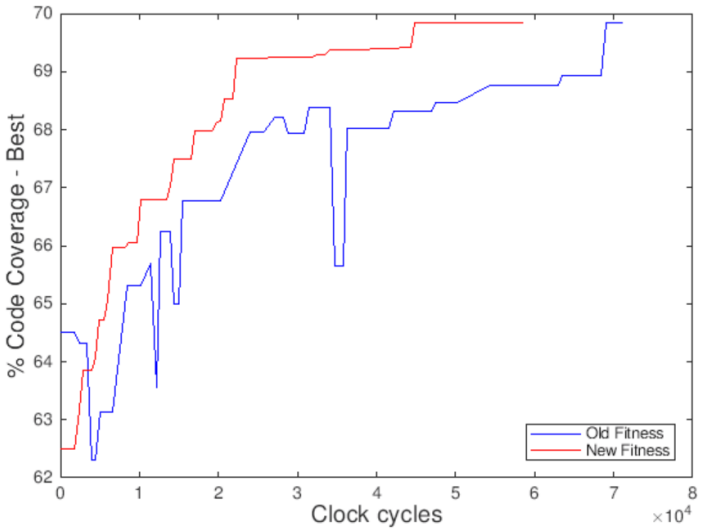

Figure 3.4: Behaviour of Code Coverage over time

the other hand, this could not be said for the blue line. Actually, it does not represent only the first element of the fitness function, which is the statement average (a): it is an arithmetic average of the first 6 elements whose trend is different over time, some values could increase while others could decrease. This does not guarantee that their arithmetic average is always growing.

Finally, as shown in graph 3.5 the new fitness is able to equalize the number of clock cycles required by each generation, flattening the peaks. Every evolutionary algorithm proceeds through steps called generations, and for every simulation, the maximum number of generation has been fixed at 100. Each of them requires a no fixed computational time expressed in clock cycles, since it depends on many factors like the size of the current population, the amount of genetic operators to apply and so on. By using this new fitness, the number of clock cycles required is almost equal for every generation, except for the first 2 and the last 4 ones.

Results

By running a complete μGP simulation, the best optimized program achieves a final average code coverage of 64%, as figure 3.6 shows. Oddly, improvements are not so striking. It has been observed that considering only the ISA of RI5CY core, μGP is not capable of covering specific parts of the core.

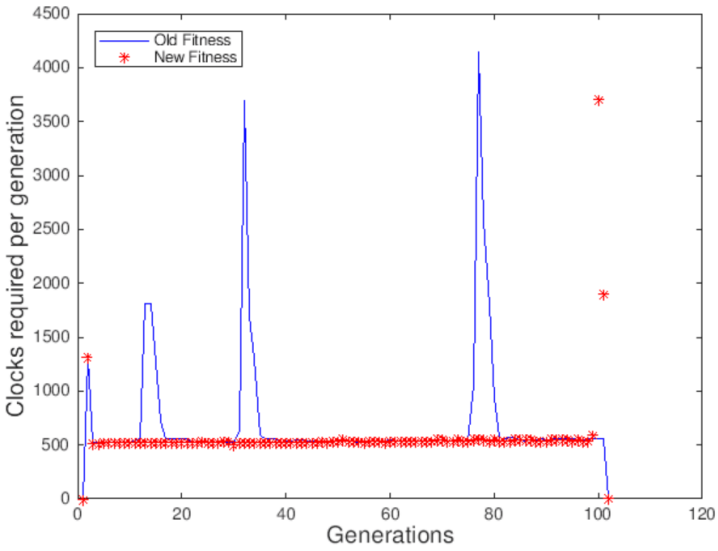

Figure 3.5: Amount of clock cycles required by generations

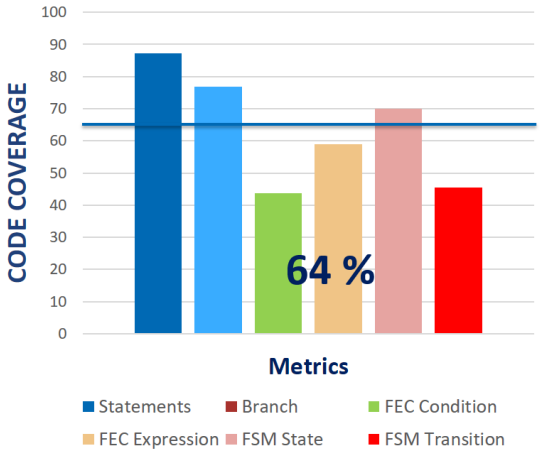

Figure 3.6: Code coverage achieved by running the Best μGP Program

3.1.4 Optimization phase

In order to consider the features and the specifications of the microprocessor under verification, two approaches have been exploited: a first phase called top-down and a second procedure which consists in the optimization of the instruction library.

Top-down approach

A *Top-down* approach consists in splitting the device in smaller and more critical parts in order to improve the coverage on these units and then join the best programs in a single test set. Initially μGP simulations were executed for the entire design: RI5CY extended with the FPU. This means that the instruction library supported all the instructions belonging to RV32IMCFXpulp ISA. However, what turned out was that for example, for every simulations code coverage of the FPU was always under 40%. The goal of this top-down approach is to recursively detect the tricky and harder units and focus only on those ones. Figure 3.7 describes the partition of the device.

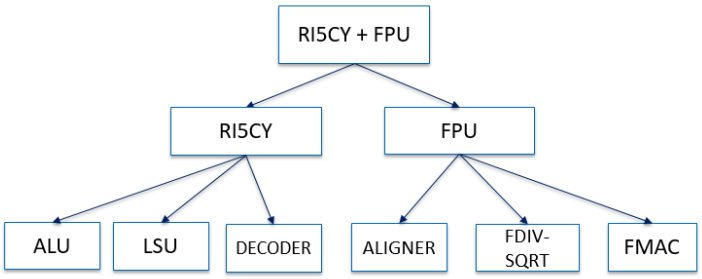

Figure 3.7: Top-down approach

At the beginning RI5CY core and Floating Point Unit were divided; then the crucial units from the code coverage perspective have been selected. The Floating Point Unit had low coverage on the Floating Point Multiply and Accumulate unit (FMAC), on the Aligner and on the unit responsible for division and square root, while the RI5CY's critical units have been identified in the Load and Store unit, in the Decoder and in the Arithmetic Logic Unit (ALU). To address this low coverage issue, the work of μGP has been centered on each of them, one at a time. For instance, by running one single simulation only on the Floating Point Multiply and Accumulate unit (FMAC), code coverage increased from a 7% to a 78% only on that unit.

Instruction Library Optimization

The *Instruction Library optimization* foresees the writing of specific pieces of assembly code to cover some corner cases that are hard to obtain with a random or pseudo-random approach. For instance, there are many if-else statements dealing with illegals instruction. Handling these instructions is crucial and necessary for the automatic completion and refinement of existing test programs. A recent research [13] exploits these illegal instructions

to create a game, called *Core War* whose high-level characteristics resemble clearly the arduous microprocessor test program generation problem. In CoreWar, two or more assembly programs are executed in the same memory area by a timesharing processor, and the goal of each program is to crash the others by having them execute illegal instructions [13].

Therefore, the instruction library has been sustained and reinforced by these assembly illegal instructions. They have been carefully written bit by bit, by looking at their original format on the RISC-V manual. For instance, to write an illegal store instruction, the sequence '111' in position from bit 12th to 14th has been forced to generate an illegal condition, as it is possible to see it in figure 3.8.

31 25	24 20	19 15	14 12	11 8	7 0	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Figure 3.8: Store Instruction Format [7]

Next, pieces of assembly code have been written to generate an infinite number, a Not-a-Number (NaN) and other combinations of instructions to foster hard to cover pieces of code. An example of NaN generation is highlighted in table 3.3. The *fdiv.s* instruction divides the values of two registers which store a logic zero. Then, this number is stored in register *f6* and is used as a souce register for the next two operations.

Generate a NaN
`fcvt.s.w f1, x0;`
`fcvt.s.w f2, x0;`
`fdiv.s f6, f2, f1;`
`fsqrt.s f12, f6;`
`fnmsub.s f27, f28, f15, f6;`

Table 3.3: Sequence of assembly instructions to generate a NaN and force the other instructions to handle it.

Finally, a specific function has been written to cover a huge part of code dealing with the *sleep state* and its transitions. To make the core sleep, the wait for interrupt (WFI) instruction is called and a timer has been programmed to wake up it after 2000 clock cycles.

The idea behind this optimization step is not to replace the activity of

Generate an Infinite Number
fcvt.s.w f1, x0;
li x3, 0x05;
fmv.s.x f5, x3;
fdiv.s f7, f5, f1;
fnmadd.s f2, f3, f4, f7;

Table 3.4: Sequence of assembly instructions to generate an Infinite Number and force the other instructions to handle it.

the evolutionary algorithm, but to support it to increase the coverage of the device, taking into consideration the details of the microprocessor core. By implementing all these optimizations, device's code coverage increased until a 85% of average code coverage. It must be noticed that thanks to the optimization process, condition coverage has almost doubled, since the number of possibilities and combinations of logical expressions has greatly increased. Furthermore, the second major improvement affects the FSM transition coverage: it arose from a 44% up to 69%. The main reason of this increase is due to the sleep function that covers many more states and stimulates many transitions, leading to a considerable increase in both the controller and the interrupt controller's coverage.

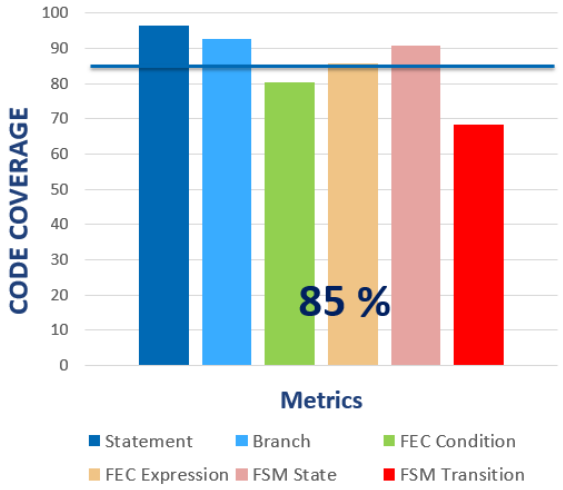

Figure 3.9: Code coverage achieved with Optimizations

3.1.5 Perturbation Module

Typically a microprocessor core is embedded in a SoC and surrounded by many peripherals which warn they want to communicate with the core through interrupts. Indeed it must be considered that peripheral's performance may also affect the speed of the core pipeline, leading to a stalls introduction. Even though a 85% of average code coverage can be still considered a good result, a verification procedure could be considered complete only if also interrupt requests and stalls are simulated. To mimic their behaviour, a hardware module called Perturbation Module [3] has been developed. It is considered as the verification environment for the PULP cores and can be instantiated for both RI5CY core and Zero-riscy core, a 2 stage area-efficient core that implements only RV32-ICM. The perturbation module is able to manage all the hardware-related events and it is directly connected to the core. The main role of this module is to generate interrupts as well as stalls on data and instructions. It is instantiated in the file *tb_riscv_core.sv* and includes three components:

- **Instruction stalls generator:** It can be programmed to introduce stalls on the instruction memory interface;
- **Data stalls generator:** It can be programmed to introduce stalls on the data memory interface;
- **Interrupt generator:** It can be programmed in order to raise interrupts requests on the core.

Both the instructions and data stalls generators have been developed starting from a pre-implemented module called *random_stalls.sv* which was instantiated in the toplevel file of the platform where the core was used. However, this module presented some limitations:

- It *only* introduced a random number of stalls on both instructions and data;
- Even when the number of stalls was equal to 0, it introduced one extra cycle;
- The user had to recompile the entire platform when introducing stalls on the desired feature.

The current stalls generator is able to implement the missing functionalities, by basically working with the same mechanism of its predecessor. To perturb the core's activity, the user can introduce a fixed or a random number of stalls. The user can decide to introduce stalls on data and instructions or only on one of them. It can also choose the maximum number of stalls that

[3] Realized by Francesco Minervini at ETH University during his master thesis project.

can be produced. All these decisions are taken by calling specific functions described in the *tb_riscv.h* library. For instance, the following functions can be used to set the perturbation module using the random interrupt requests generation and inserting random stalls on the data interface:

```
//Functions prototypes
void pert_set_irq_mode ( int mode );
void pert_set_mode (int feature ,int mode );
// Setting modes
pert_set_irq_mode ( PERT_RANDOM );
pert_set_mode ( PERT_DATA , PERT_RANDOM );
```

If the user does not want to use the stall generator, it simply forwards the value on the input lines to the output lines, so the execution is the same as the scenario when the perturbation module is not instantiated. On the other hand, the interrupts generator can be controlled via specific parameters to implement one between the following modes:

- Bypass interrupt requests coming from the external event unit directly to the core, as in the normal execution.
- Introduction of random interrupt requests, randomizing on the number of cycles which separate two consecutive requests.
- Raising an interrupt request when the current value of the program counter is equal to a certain threshold value specified by the user.

Through the use of this module it is possible not only to mimic the communication between core and the surrounding hardware modules, but also to reproduce the presence of bugs. This is a huge benefit for verification environment which can be aided and facilitated by this hardware module. Thanks to its insertion, many bugs have been replicated and solved. Therefore, to increase even more the code coverage of the microprocessor core, these functionalities have been included in μGP test program generations by calling these specific functions of the perturbation module's library. Finally, by running the final set of best programs, the code coverage of RI5CY core increased up to a 90.28%. It is important to highlight that this result has been obtained thanks to these many improvements. The test set includes the best programs obtained running several times μGP on different units as well as the funcions call of the perturbation module.

Table 3.5 highlights some important features of the final test set. It is composed by 15 programs grouped in 5 functions called by the main program. The total number of bytes has been computed by multiplying the number of instructions compressed by 2 and then subtracting this number from the number of total instructions multiplied by 4. It must be considered

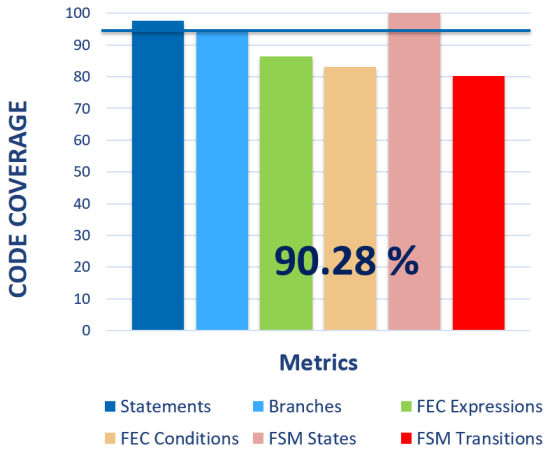

Figure 3.10: Final Code Coverage

TEST SET DETAILS	
Number of programs	15
Number of lines of code	5584
Number of bytes	100214
Number of 32-bit instructions	25721
Number of compressed instructions	1335
Execution time	6 minutes
Generation time for each program	8-14 hours

Table 3.5: Final Test Set details

that the generation time for each and every program goes from 8 to 14 hours. Moreover, the number of lines of code has been computed by removing the empty rows. To conclude, table 3.6 summarizes the results of the average code coverage achieved step by step.

3.1.6 Uncovered Regions

The remaining uncovered 10% depends on multiple conditions that never happen, like FSM transitions related to a RESET state, or special cases of infinite numbers or NaN which affect the execution of instructions. For instance, the Finite State Machine (FSM) of the multiplier is made up of the following states: IDLE, STEP0, STEP1, STEP2, FINISH. As it is possible

TEST SET	CODE COVERAGE (%)
Handwritten Program	38
μGP Best Random Program	52
μGP Best Program	64
μGP Best Program + Optimization Step	85
μGP Best Program + Optimization Step + Perturbation Module	90

Table 3.6: Code Coverage Enhancement

to notice from the figure 3.11, the transitions between each intermediate state of the FSM and the IDLE state are never excited since a reset signal has never been raised to stop the execution of a multiplication.

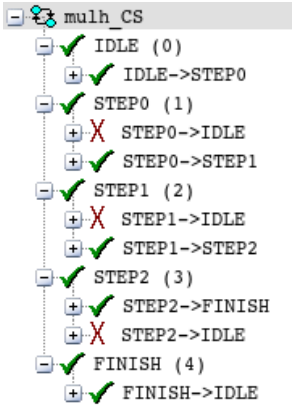

Figure 3.11: Finite State Machine of the Multiplier Unit

By carring out this careful code coverage analysis, it has been also discovered that two FSM states of the Load and Store unit are unachievable because the conditions for which it is possible to enter the states are never realized.

3.1.7 Test Set Generation for Zero-riscy

Preliminary results gathered on Zero-riscy core are provided. First of all, PULPino has been compiled to support Zero-riscy core instead of RI5CY; then, a new instruction library supporting only RV32IM instructions has been provided. Whereas the optimization phase has not been carried out, with the only ISA μGP algorithm has reached a 56% of final average code

coverage of Zero-riscy core.

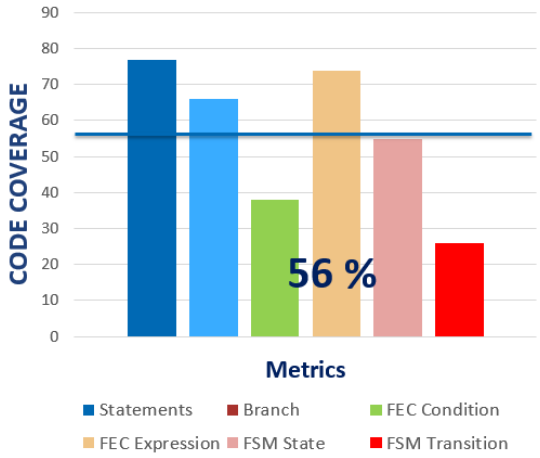

Figure 3.12: Code Coverage achieved for Zero-riscy with the only ISA

As figure 3.12 shows, the two metrics that are under the mean are expression coverage as well as FSM transition coverage. This means that there are transitions and combinations of logic expressions that never occur which need to be excited by ad-hoc sequences of assembly instructions. This 56% can not be considered a satisfactory result, so it must be raised and enhanced by implementing a suitable phase of optimization.

3.2 Verification Phase

During the microprocessor design process, engineers commonly use extensive simulations to increase confidence on device correctness. However, reaching a high code coverage of the device under verification is not a valid measure of correctness but should only be considered as a measure of confidence. Code coverage metrics are used to guide the process of test programs generation but a one more verification step is required to ensure also correctness. For instance, simulations results may be used to compare a hardware model aganst higher-level references or instruction set simulator [19].

This methodology uses an Instruction Set Simulator (ISS) to execute instructions and record results to be checked. An Instruction set simulator is a tool that runs on a workstation called the host machine, to mimic the behavior of a target machine. Typically, instruction set simulation allows the user to examine the internal state of the target machine, such as the values of processor registers during the execution of each instruction [23].

Besides, as showed in figure 3.13, this methodology introduces also a simulation checker, named *Simchecker* [4]. It is written in SystemVerilog and his goal is to verify that the value written in the registers is the same between the core and the golden model, checking the correctness of results. In case of mismatches it informs raising an error warning. Simchecker has been realized to work with the pipelines of both RI5CY core and FPU.

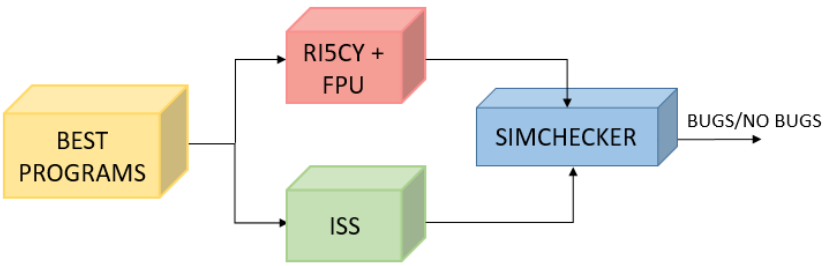

Figure 3.13: A first verification approach

At the beginning, to check the correctness of RI5CY core, the programs with the highest coverage were selected. Roughly speaking, the set of programs with the 90.28% of final average code coverage has been picked and used to identify incorrect DUV behavior. With this methodology, each assembly instruction is executed by both RI5CY core and the ISS and results are stored in the destination registers. Then, the Simchecker compares them and points out possible differences.

By exploiting this methodology, unfortunately not even a bug has emerged.

This fact strengthens the idea that a high code coverage is not a measure

[4]Improved and completed by Francesco Minervini in his master thesis project at ETH University.

of correctness. To reach a 90.28% of device's code coverage, the evolutionary algorithm can reject some low-coverage individuals that could contain a bug. For instance, a program may contain a buggy instruction but it does not increase the coverage with respect to another program, so in the process of generation, it is rejected. μGP is not aware of this because its goal is only to raise the code coverage, without checking the correctness of the instructions. Accordingly, it must be considered that typically the basic features have already been verified with handwritten programs and in most cases they are correct. Problems can arise in very special and difficult to simulate cases. Exhaustive or nearly-exhaustive tests are often necessary, but the effort required to generate them manually and the time required to simulate them are practically infeasible [14]. Shrink computational effort is the main reason why this methodology exploites an evolutionary algorithm to generate the set of programs. It is quite simple to set up, and requires no human intervention when running. An evolutionary algorithm not only provides an effective methodology to try random modifications, but also it allows merging useful characteristics from different solutions, exploring efficiently the search space.

The next experimental idea came out inspired by these assumptions, with the goal of take advantage of these interesting features. To consider the immense test space carried out by the evolutionary algorithm, instead of performing generation of programs and verification of instruction's correctness sequentially, they have been merged in a single phase.

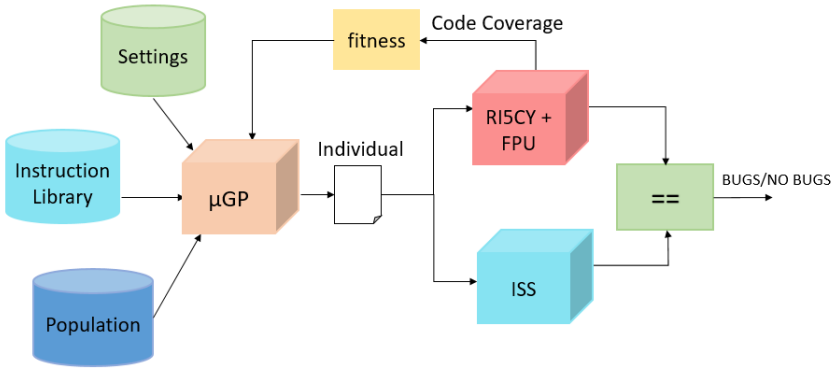

Figure 3.14: Parallel Generation and Verification

While μGP drives programs generation, every single program is executed not only on RI5CY core to produce code coverage feedback, but also on the Golden Model (ISS), as figure 3.14 points out. Then, results are compared by the Simchecker. If these differ, it issues an error warning that confirms the presence of a bug. This information can be sent to the evolutionary algorithm which can decide to stop the simulation or continue and collect

other bugs.

Through this approach it has been possible to better explore the research space revealing the presence of hidden bugs inside the DUV. Consequently, this technique not only tries to increase the code coverage of the processor, but at the same time ensures that that portion of code covered by the set of programs is correct.

RTL	ISS
p.clipr	p.subuRNr
p.extractr	p.addRNr
fclass.s	pv.srl.sc.b
fsqrt.s	p.subRNr
fcvt.s.w	pv.sll.sci.b
fcvt.s.wu	p.macuR
fcvt.w.s	p.machhuR
fcvt.wu.s	p.mulhhuR
	p.macsR
	pv.sra.b
	pv.sll.sci.h

Table 3.7: List of Bugs

Table 3.7 lists all the bugs that have been discovered with this methodology. It is necessary to highlight that normally, beeing the ISS a golden model, it must be perfect. However, the available ISS was under development and therefore had imperfections, especially on corner cases and on cases not specified in the datasheet.

3.2.1 Guilty Detection Process

During a normal execution, the SimChecker makes a comparison between results and exhibits eventual mismatches. If an instruction's result is incongruous, it displays an error message on the external evaluator's interface. How to identify the responsible of the error? As already remarked, this *guilty detection process* should not be introduced in a normal hardware verification mechanism exploiting a golden model since it must not contain

issues. Anyhow, in this methodology it has been a fundamental step which required a not-negligible effort and caution.

Let us consider as example one of the PULP specific faulty instructions which belongs to the set of "General ALU Operations". Its target is to create a kind of range determined by the second source register to clip the value of the first source register and store it into a destination register.

```
p.clipr rD, rs1, rs2
```

Where the value of the destination register (rD) is computed with the following algorithm:

```
if rs1 <= -(rs2+1), rD = -(rs2+1),
else if rs1 >=rs2, rD = rs2,
else rD = rs1
```

Surprisingly, when the Simchecker bumped into this instruction, the following error was raised.

```
p.clipr  x6, x17, x17  x6=9b9bff9b x17:64640064 x17:64640064
SIMCHECKER: reg 6; ISS 64640064; RTL 9B9BFF9B
```

The value of x17 is the result of a random μGP choice and has a hexadecimal format. A careful analysis of the instruction specifications makes clear that the value stored in rs1 is identical to rs2 and therefore rD should be equal to rs2, but the value stored into the RTL destination register is 0x9B9BFF9B that is totally different from 0x64640064. It is evident that the correct result is stored into the destination register of the ISS and the bug is trapped into RI5CY core. After a careful reading of the RTL code, came out that it was derivable from an overflow issue in the RI5CY Arithmetic Logic Unit (ALU). However, in many other cases, it has not been immediate to understand whether issue came from the RTL or ISS. Therefore, a really meticulous investigation on the instruction specifications and on the code has been performed. This process named "guilty-detection" has been repeated for each and every mismatch raised by the SimChecker.

As a conclusion, the power of this methodology actually lies in the ability to find out problems related to cases that are really difficult to imagine and so to excite. It's nice to notice that all these bugs have not been discovered with the first approach (generation and verification in sequence). As already remarked, a 90.25% of final average code coverage of RI5CY core is only a measure of confidence and not a view of correctness. To reach this high percentage μGP could have discarded programs containing faulty instructions or incorrect behaviours to encourage a higher code coverage program. Through this second approach, the correctness of each and every instruction is checked so actually turns out that with this proposed methodology, a high code coverage is also a measure of correctness.

Chapter 4

Conclusions

This thesis describes an efficient methodology on how to perform functional verification on pipelined microprocessors. The approach employs an automatic test program generator based on an evolutionary algorithm to produce the set of assembly programs and a new verification method which enables the full exploitation of the microprocessor functionalities. With the advent of even more higher performing devices, traditional functional approaches are not sufficient to guarantee the correctness of the devices. This methodology exploits μGP, an evolutionary algorithm to optimize and devise automatically programs written in assembly-like languages. While features of μGP stem from standard genetic programming, μGP was designed for generating syntactically correct assembly programs of variable size and fully exploit the available assembly syntax: different addressing modes, instruction set asymmetries, subroutines, interrupt calls [13]. μGP is versatile and suitable for different microprocessors and different goals, as long as their Instruction Set Architecture is described in the form of an instruction library. It has already been used in many different fields such as: creation of test programs for pre- and post-silicon validation; design of bayesian networks; creation of mathematical functions represented as trees; integer and combinatorial optimization; real-value parameter optimization; and even creation of corewar warriors [9]. The need to use an automatic test program generator rises from the necessity to cope with the increasing hardware component's complexity. Actually, the advances in microelectronics technologies allowed semiconductor manufacturers to deliver chips with ever-shrinking form factors and ever-increasing switching frequencies [13]. This requires even more sophisticated verification mechanisms to best exploit the device's functionalities and catch incorrect behaviours. To have a higher degree of confidence, it is anymore possible to rely only on manually written programs. They are useful to check corner cases and specific device functionalities but can not face the increasing complexity of today's pipelined microprocessors. Therefore, the only solution to address these technological changes is to exploit an automatic generation method and μGP is the one adopted for this project. It has been demonstrated that the use of an automatic method can dra-

matically help designers and engineers: Instead of checking massive random simulations for differences with respect to the correct model, validation experts can let the automatic test case generator work for the proper time and even actually examine the test set it produces [19]. However, results demonstrated that μGP algorithm requires several intermediate optimization approaches to improve the quality of test programs, since with the only Instruction Set it is not able to reach specific part of the device. Of course these steps demand expert and skilled engineers whose work is to carefully analyze the code and undertand which part of the design is not excited and why. The process of test-program generation is guided by coverage measurements. Code coverage metrics are useful to have an estimation of the verification completeness, providing a measure of confidence in the design correctness. However, setting a high priority to a single coverage metrics turned out to be inconvenient for the computational effort required. A new fitness function capable of weighing these code coverage metrics in a single formula has been proposed. It has been demonstrated that it can reduce the time required to achieve the same code coverage by around 20% with respect to the standard linear fitness functions adopted, described in Chapter 2. Computational time is reduced and the test program generation process is improved.

This methodology not only proposes an automatic simulation-based test program generation but also improves the verification process by taking advantage of the excellent features of an evolutionary algorithm. In the previous chapters it has been demonstrated that by bonding together the assembly program generator and the instruction set simulator it is possible to perform in parallel the generation of the programs and the checking of their correctness. Exploiting the excellent features of an evolutionary algorithm to generate assembly programs, a wider space of design and specific corner cases are excited and simulated. Therefore, while the goal of μGP is still to increase the code coverage, the correctness of each instruction is checked. Differently from the first approach, a high code coverage now is no more only a degree of confidence, but it becomes a measure of correctness. Consequently, this proposed methodology not only tries to increase the code coverage of the processor, but at the same time ensures that that portion of code covered by the set of programs is correct.

Thanks to this methodology, many bugs have been discovered on both the RTL model of the core and the instruction set simulator (ISS). Many of these are caused by the absence of technical specifications on corner cases. For instance, in the ISS, the behaviour of many instructions was not determined for operands which stored a zero or a maximum number. This led to a divergence in the results. Experiments clearly show that using this method may dramatically help the process of verification of a pipelined microprocessor, since it exploits the exceptional features of an evolutionary algorithm.

4.1 Future work

The major advantage of this project regards its *portability*. The approach proposed can be extended to other RISC-V microprocessors through the use of a proper instruction library to generate the assembly programs; by disposing of an instruction set simulator (ISS) to check instruction's correctness and including an hardware module (i.e the Simchecker) in charge of comparing the results of the microprocessor and the ISS. With some effort it is also possible to extend this approach to memories and caches. The problem of testing embedded memories has become more complex in recent years [24]. A first attempt was made to test the virtual memory of Ariane core, a 64-bits 6-stage RISC-V CPU developed at ETH University. For this purpose, a complete μGP environment has been prepared with a proper instruction library and many scripts. To have a higher visibility of the microprocessor's activity, many SoCs like PULPino include also a Debug unit. An interesting idea could be to extend this methodology to deal with this part of the system or in general expand the features to support some privileged instructions in the RISC-V ISA (uret -sret etc..).

Bibliography

[1] Walter Soto Encinas Jr, César Augusto Dueñas M. Functional verification in 8-bit microcontrollers: A case study. *Brazil Semiconductor Technology Center (BSTC) - SPS - Motorola.*

[2] Ernesto Sánchez, Matteo Sonza Reorda, and Giovanni Squillero. Efficient techniques for automatic verification-oriented test set optimization. volume 34, pages 93–109, Mar 2006.

[3] ETH Zurich and University of Bologna. Pulpino. `https://pulp-platform.org//wp-content/uploads/2017/08/datasheet.pdf`. [Online; accessed 05-Mar-2018].

[4] ETH Zurich and University of Bologna. Riscy datasheet. `https://pulp-platform.org//wp-content/uploads/2017/11/ri5cy_user_manual.pdf`. [Online; accessed 06-Mar-2018].

[5] ETH Zurich and University of Bologna. Pulp platform. `https://www.pulp-platform.org/documentation/`. [Online; accessed 05-Mar-2018].

[6] ETH Zurich and University of Bologna. Pulp platform. `https://www.pulp-platform.org/`. [Online; accessed 05-Mar-2018].

[7] Andrew Waterman1, Krste Asanovic. *The RISC-V Instruction Set Manual. Volume I: User-Level ISA, Document Version 2.2.* Andrew Waterman and Krste Asanovic, 2017.

[8] ETH Zurich and University of Bologna. Zero-riscy datasheet. `https://pulp-platform.org//wp-content/uploads/2017/08/zeroriscy_user_manual.pdf`. [Online; accessed 06-Mar-2018].

[9] A. Tonda G. Squillero. Microgp. `http://ugp3.sourceforge.net/`. [Online; accessed 12-Mar-2018].

[10] Eiben A.E., Smith James E. *Introduction to Evolutionary Computing.* Springer-Verlag Berlin Heidelberg, 2003.

[11] G. Wei. Study on genetic algorithm and evolutionary programming. In *2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*, pages 762–766, Dec 2012.

[12] A. Tonda G. Squillero. Evolutionary algorithms. `https://sourceforge.net/p/ugp3/wiki/EAs/`. [Online; accessed 12-Mar-2018].

[13] F. Corno, E. Sanchez, and G. Squillero. Evolving assembly programs: how games help microprocessor validation. *IEEE Transactions on Evolutionary Computation*, 9(6):695–706, Dec 2005.

[14] F. Corno, G. Squillero, and M. Sonza Reorda. Code generation for functional validation of pipelined microprocessors. In *The Eighth IEEE European Test Workshop, 2003. Proceedings.*, pages 113–118, May 2003.

[15] F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante. On the test of microprocessor ip cores. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 209–213, 2001.

[16] Alberto Tonda. Genetic operators. `https://sourceforge.net/p/ugp3/wiki/Genetic%20operators/`. [Online; accessed 05-Mar-2018].

[17] B. Bentley. High level validation of next-generation microprocessors. In *Seventh IEEE International High-Level Design Validation and Test Workshop, 2002.*, pages 31–35, Oct 2002.

[18] Subhranil Deb. Delivering functional verification engagements: A proven, systematic and assured approach. *Synopsys*, 2015.

[19] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero. Automatic test program generation: a case study. volume 21, pages 102–109, Mar 2004.

[20] Jian Shen and J. A. Abraham. Native mode functional test generation for processors with applications to self test and design validation. In *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, pages 990–999, Oct 1998.

[21] N. Utamaphethai, R. D. Blanton, and J. P. Shen. Superscalar processor validation at the microarchitecture level. In *Proceedings Twelfth International Conference on VLSI Design. (Cat. No.PR00013)*, pages 300–305, Jan 1999.

[22] Gaurav Kumar Verma and Doug Warmke. Supercharge your verification using rapid expression coverage as the basis of a mc/dc-compliant coverage methodology.

[23] Jianwen Zhu and D. D. Gajski. An ultra-fast instruction set simulator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):363–373, June 2002.

[24] S. K. Jain and C. E. Stroud. Built-in self testing of embedded memories. *IEEE Design Test of Computers*, 3(5):27–37, Oct 1986.