CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Microprocessor Verification Framework For SPARC-LEON Microprocessor

Master's thesis in Embedded Electronic System Design

Kenneth Peter

Prathamesh Moralwar

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Microprocessor Verification Framework For SPARC-LEON Microprocessor

Kenneth Peter
Prathamesh Moralwar

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Microprocessor Verification Framework For SPARC-LEON Microprocessor
Kenneth Peter
Prathamesh Moralwar

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Advisor: Martin Rönnbäck , Cobham Gaisler
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Courtesy of Cobham Gaisler. GR712RC is a dual-core processor from Cobham Gaisler

Typeset in LaTeX
Gothenburg, Sweden 2019

Microprocessor Verification Framework For SPARC-LEON Microprocessor
Kenneth Peter
Prathamesh Moralwar
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Verification is a crucial part of an engineering project to assure quality of the product. In digital design, verification is a heavily researched topic as companies desperately look for ways to make the process faster and more reliable. The challenge only grows by the day as designs ambitiously swell in size and complexity. The language used, the methodology followed, the goal of verification and the evaluation of the verification process itself are some important aspects to be decided upon before commencement of this imperative exercise.

Directed testcases can overlook peculiar cases which can easily be encountered by allowing a certain degree of randomness in test vectors. In this thesis it is shown how verification can be improved by randomizing test inputs. The usefulness of functional coverage is also elucidated. It is also shown how randomizing test vectors and implementing functional coverage compliment each other. While randomization help achieve coverage goals faster, the results from functional coverage can be analysed to update test vector generation. The design at hand is the SPARC based LEON-3 microprocessor by Cobham Gaisler. The above tasks were performed by maintaining the spirit of software-based testing.

Keywords: Verification, Code Coverage, Functional Coverage, Methodology, SPARC, LEON-3, Randomization, Coverage Metrics.

Acknowledgements

We would like to express our sincere gratitude to our supervisors Pedro Petersen Moura Trancoso, Martin Rönnbäck and Alen Bardizbanyan for guiding us during our master thesis work.

We would also like to thank all the employees working at Cobham Gaisler specially to Fabio Malatesta and Marcel Ramos Lopes for their constant support, advice, feedback and suggestions.

We also thank Per Larsson-Edefors and Lars Svensson for providing necessary resources to execute this thesis.

Lastly, we would like to extend our gratitude to our family and friends for their love and support.

Kenneth Peter & Prathamesh Moralwar , Gothenburg, June 2019

Contents

List of Abbreviations

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application Specific Integrated Circuits
ASCII	American Standard Code for Information Interchange
BA	Branch Always
CC	Code Coverage
CP	Co-Processor
EDA	Electronic Design and Automation
FC	Functional Coverage
FPU	Floating-Point Unit
GRLIB	Gaisler Research Library
HDL	Hardware Description Language
HVL	Hardware Verification Language
IP	Intellectual Property
IU	Integer Unit
MMU	Memory Management Unit
NOP	No Operation
OVM	Open Verification Methodology
PC	Program Counter
PROM	Program Read Only Memory
RAW	Read After Write
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
ROM	Read Only Memory
SoC	System On Chip
SV	SystemVerilog
SPARC	Scalable Processor Architecture
SVA	SystemVerilog Assertions
SREC	S-Record
TCL	Tool Command Language
URM	Universal Reuse Methodology
UVM	Universal Verification Methodology
VMM	Verification Methodology Manual
VHDL	VHSIC Hardware Description Language
WAW	Write After Write
WAR	Write After Read

List of Figures

List of Tables

1
Introduction

Microprocessors are widely used in a variety of applications ranging from small embedded systems to large scale data centres and supercomputers. Design complexities of microprocessors have increased over the years due to tremendous research which has resulted in architectural advancements such as more pipeline stages, branch prediction and speculative execution. The boost in research is also due to an increasing demand on computational capabilities, efficiency in power and stricter time constraints and energy consumption requirements.

Since processors are becoming more powerful than ever and support more and more features, they are also getting increasingly complex to design, verify and debug. Complex designs demand detailed verification and debugging. Therefore, Electronic Design and Automation (EDA) tools and verification methods have become more relevant for and researched by verification teams over the years.

1.1 Motivation

The entire workforce on any digital design can be broadly classified into two categories, the design team and the verification team. A design engineer constructs a design based on the specifications. To verify the design implementation by the design team, a verification team is required. A verification engineer must verify the design under test (DUT) covering all possible cases. But complex designs can have an infinite number of cases to verify. The approach of a verification engineer should be different from that of a design engineer [1]. If both of them think along the same lines then both of them are likely to commit the same errors.

An estimated 70% of the total effort in a digital design goes into verification and this number continues to increase with the design complexities [2]. Therefore, a significant portion of project cost (time, effort and money) is dedicated to verification teams. Microprocessors are functionally verified by writing functional behavioural testbenches that verify the functional correctness of the design under test (DUT). Designing a dynamic and robust verification framework to verify the correctness of a processor is important. A microprocessor with a large number of features has a large number of dependencies and corner cases, all of which need to be verified or 'covered' by the testbenches. Such designs require fast, robust and trustworthy verification methodologies. The faster the verification, the quicker the time-to-market.

Verification can only be performed up to a certain level of satisfaction. This depends largely on the verification methodology used for verification. A metric to measure the satisfaction of verification can be helpful to gain confidence on verification conducted on a design.

Many methodologies for verification exist. They are discussed in later sections. What is required is a methodology that is robust, easy to use and extendable to later versions of the design. Many ways of measuring the level of verification also exist each with their pros and cons. It is required to use such metrics in a thoughtful way so as keep the verification process headed towards the right direction.

1.2 Problem Statement

A testbench is said to be of good quality if it tests or 'covers' every aspect of a design code, for example, the number of design code lines it has hit, the number of input vectors it provides, the functionalities it tests and the quality (validity, legality and relevance) of the input vectors it provides. Coverages offer a good way of measuring the quality of testbench. As important it is to verify the design, it is vital to achieve clean coverage values (100%), or no part of the code or aspect of the design should be left uncovered during verification process. These two tasks, verification and coverage measurement, go hand-in-hand such that one is not complete without the other.

The current method of verification for the LEON family of processors in Cobham Gaisler is software-based testing, i.e, software programs are executed on RTL models of the processors. Since these codes are handwritten for a specific purpose, this method could be classified as directed testing. Code coverage reports are generated for the same. Directed testing, with its many pros, is not ideal when clean coverage values are of top priority. When one person writes an RTL for a complex design, such as a processor, and another is given the task of writing a software program for testing the same design, it is difficult for the latter to know what kind of code would test every part of the RTL satisfactorily.

A general way of achieving coverage goals, as well as finding hard to catch bugs, is to test the DUT with random input vectors. For the LEON processor, the random input vectors would be given as a test program with random instructions. A script that generates this random program was to be written. Another kind of coverage metric, namely functional coverage, was also to be introduced. The purpose of this was two-fold, to broaden the coverage perspective of the processor verification and to aid the development of the random program generator. In this case, implementing functional coverage is less straightforward as the design is tested with an executable program rather than a stream of bits. It was also a necessary to make these implementations adoptable to other flavours of the LEON processor or even processors of different architectures. An environment or a verification framework was to be

developed with all the above features and flexible enough to accommodate more functions in the future.

1.3 Thesis Objective

The objectives of this thesis work are:

- Measure and improve verification metrics viz improvement in code coverage with less instructions, runtime and efforts.
- Develop a new verification framework capable of generating constrained random test vectors and perform functional coverage using SystemVerilog.
- Evaluate newly created verification framework with the existing framework.

1.4 Related Work

Coverages (code & functional) were originally intended to be used as a means for gaining satisfaction on the verification performed on a digital design. The usefulness of these metrics have made them so popular that entire verification flows are often centred around coverages. What was once used as a complimentary tool is now a deciding factor on not just how verification is concluded, but also how it is planned and performed. Languages such as SystemVerilog offer inbuilt constructs to implement functional coverage. In [3], an approach to perform functional verification has been proposed in SystemC, a verification language that does not have inbuilt constructs for functional verification. The purpose of this is to be able to perform coverage-driven verification for those familiar with SystemC.

A little more advanced approach is described in [4]. At the initial stages of verification, random testing (discussed in later paragraphs) is performed. Directed test cases are introduced as bugs are detected. When the number of bugs starts reducing, the focus is shifted on coverage goals. This helped in detecting hard to catch bugs and writing creative test cases. A method of performing verification driven by coverage along with randomizing test inputs was proposed in [5] for data and control hazards in a 5-stage processor. An automated iterative system is developed where verification is performed until coverage goal is reached. With coverage goals in place, time for verification was brought down from an average of 30 days to an average of 7.5 days.

Random test vector generation has been widely used by processor manufacturers. The characteristics of these input test vectors can be used to randomize them meaningfully for specific units of the processors. Functional coverage driven test generation for validation of pipelined processors is presented in [6]. A graph based model approach is presented in [7]. A graph model of the pipelined processor is derived

from the functional abstraction and then later functional test programs are generated using coverage of the pipeline behavior. In this model based approach, the test generation time is drastically reduced due to the use of module level property checking.

A random program generator based on SPARC V9 is presented by Hegde *et al.* [8]. This paper presents a random program generator that takes a user input with a seed value. The sequence of assembly instructions is later run through the RTL model to check the correctness of the design. Another program generator example is presented by Wagner *et al.* [9] using Markov-model-driven random instruction generator. A model is generated from user template and then this model template is used to generate random programs.

Jian Shen *et al.* [10] proposed an abstract based technique for verification of processor microarchitectures in which an abstract finite-state machine model is derived directly from the processor HDL description. This paper also presents a way to translate abstract FSM transition into assembly instructions by way of system level operations. Their model is used for validation of coverage analysis and also applied on two general purpose microprocessors successfully.

The work developed in this thesis work is targeted for verification of LEON family microprocessor and other processor architectures such as RISC-V. The framework developed in this thesis work not only generates constrained random programs but also verifies and measures functional and code coverages. The framework is built using SystemVerilog UVM base classes which also makes it reusable for other processor architectures such as RISC-V. The microprocessor verification framework is fully automated with user controls which makes verification process faster and more robust. A part of framework i.e the random program generator has already been applied successfully to catch RTL bugs from processors which are under development at Cobham Gaisler. In a nutshell, this thesis work is unique in nature in that it combines a random program generator along with verification metrics to verify microprocessors.

1.5 Report Outline

The report is organized into different chapters as follows. Chapter 2 gives some background on the LEON-3 processor, verification methodologies and coverage metrics. Chapter 3 focuses on methods used to achieve the thesis objectives. Chapter 4 details on results obtained from the thesis work and discussion. Chapter 5 concludes this report with conclusion and future work.

2
Background

Some background is required to understand the thesis work presented in this report. This chapter will walk reader through verification techniques, SystemVerilog, UVM, coverage metrics, SPARC architecture and LEON-3 processor. All relevant and necessary information is added in this chapter which will help to understand subsequent chapters.

2.1 Verification Techniques

Verification had and continues to take a lion's share of the entire design cycle of any digital design. For microprocessors many methodologies have emerged to tackle this complicated challenge. Dill presented spectrum of verification techniques in [11] shown in Figure 2.1.

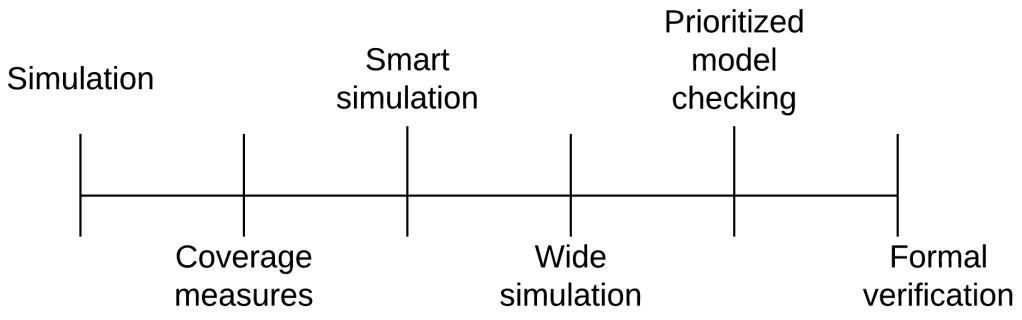

Figure 2.1: A spectrum of verification techniques [11]

The spectrum of verification techniques range from conventional simulation to full formal verification. In the beginning of the spectrum, conventional simulation is coupled with coverage analysis which provides metrics of how thoroughly a design under test is verified. Coverage metrics can also be used to guide and create the test vectors. A bit further on the spectrum is a technique which the author defines as smart simulation. Smart simulation generates functional tests based on coverage metrics. Wide simulation on the spectrum is defined as symbolically representing large sets of states in relatively few simulations. Wang *et al.* [12] presented that we could improve the efficiency of functional verification based on test prioritization

thus saving simulation time. Model checking and formal verification are out of scope of the thesis work.

The existing verification framework at Cobham Gaisler for the LEON-3 processor only supports directed testing. Directed testing fits well within conventional simulation and coverage measures. While directed testing is a straight forward approach with good control of test vector generation, a lot of unexpected bugs can stay hidden. Such bugs tend to appear with special input vectors often overlooked by the engineer. To tackle this, a mechanism to auto-generate constrained random test cases is needed. There are many verification languages that can be used for this purpose. One such example is presented by Behm *et al.* [13] paper discusses about industrial experience with test generation languages for processor verification. The Genesys and Genesys-Pro verification languages developed at IBM are a result of more than twenty years of test generation for processor verification within IBM. Genesys provides the ability to auto-generate constrained random test cases. This is also implementable in other general verification languages like e [14], Vera [15], System-C [16] and SystemVerilog [17]. However, SystemVerilog has been used in many processor vendors due to capabilities and features offered by the language (please refer section 2.2).

The extent of verification performed on a design is measured using coverages. The level of priority given for coverages is different in different methodologies. For example, Intel's coverage driven verification methodology takes feedback from coverage analysis and used it to enhance and modify the simulation environment and testbench. The coverage driven methodology was adopted for one of Intel's mobile processor [4]. An upgraded version is the coverage oriented methodology used in the verification of Merom processor can be found in [18].

Currently, for the LEON-3 processor only code coverage is measured. While this is a necessary parameter of the verification process, it is not adequate, especially for large designs. Absence of other coverage metrics (functional coverage) does not necessarily indicate that there is a presence of bugs nor does it provide any absolute guarantee. Functional coverage is a well known and a more trusted way of measuring verification progress. This is also a feature of SystemVerilog. Custom coverage measure can also be tailored for the design at hand to get a better understanding of input vectors generated and functionalities tested. For example, in microprocessor verification data hazards can be tested that arise due to program dependencies, i.e, when instructions that exhibit data dependency modify data in different stages of a pipeline. There are three different types of data hazards mainly classified as RAW (read after write), WAW (write after write), WAR (write after read). Therefore, for microprocessor verification, verification engineer should include test scenarios to cover these hazards. Due to its importance RAW hazard should also be included in coverage metrics, one such example is shown in [19].

Formal verification as shown in the Figure 2.1 can also be used instead of traditional approach. The most fundamental difference between simulation and formal

verification is that the latter is implemented using mathematical techniques. This approach does not require a vector set and therefore does not require a testbench to be written to exercise the design. This saves time right away. Seger *et al.* [20] proposed a practical formal verification in microprocessor design methodology to tackle the formal verification problems. This methodology involves circuit assessment, scalar verification, model checking and theorem proving. Methodology has been effective in large scale industrial projects at Intel and can be applied to complex design projects. One such example from the paper is IEEE compliant floating-point adder.

SystemVerilog assertions can be used to perform formal verification. The static formal verification tool mathematically derives a model for the RTL logic under test. SystemVerilog assertion properties can test the design in all possible modes of operation and therefore can isolate bugs and undesired behavior that a designer might not have thought to test. However, currently static formal verification is limited to small design blocks because when the static formal algorithm evaluates the logic under test and creates a model, its ability to explore all possible corners of logic hits the so-called state-space explosion problem. The number of automatically created input stimuli combinations reaches a prohibitive limit and it would take forever to prove that the SystemVerilog assertion properties for a given logic block never fail.

Any verification techniques that don't fall under the formal category are called Informal techniques and are shown in Figure 2.2.

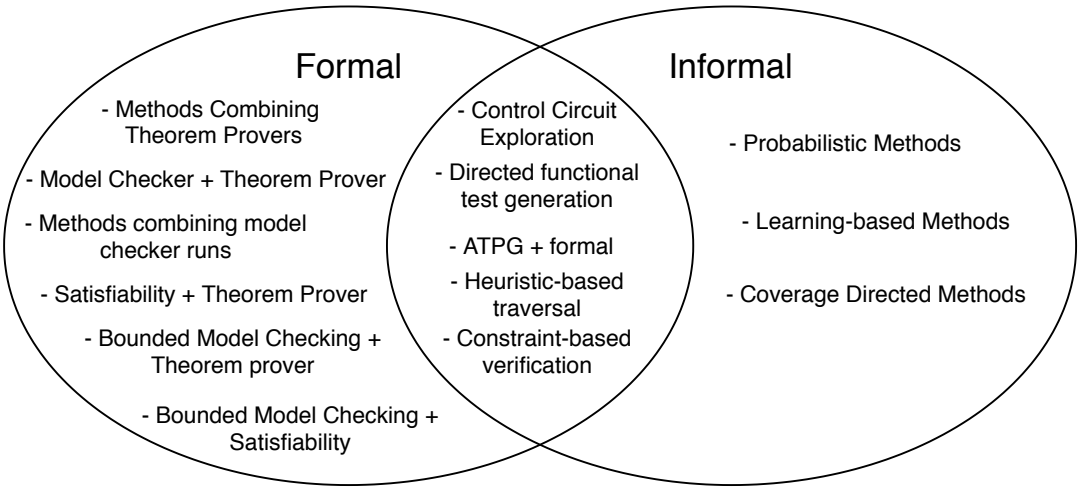

Figure 2.2: Hybrid verification techniques [21]

One possible solution to bridge the gap between formal verification and simulation is by using hybrid techniques. A survey on hybrid techniques by Bhadra *et al.* [21] explains how hybrid verification techniques that are a mix of formal and informal verification techniques can be used to compliment each other to improve functional verification. Hybrid techniques, shown in Figure 2.2, have been around for some years and continue to be used for processor verification. Benjamin *et al.* conducted

a study in coverage-driven test generation [22]. The results showed 50% improvement in transition coverage with less than a third the number of test instructions, demonstrating that hybrid techniques can significantly improve functional verification. As discussed earlier, this thesis work will focus on informal ways of verification specially coverage directed method.

2.2 SystemVerilog and the Universal Verification Methodology

VHDL and Verilog became a popular choice for digital hardware design. As designs grew larger and more complex, the testing features offered by these languages became insufficient to perform fast and reliable verification. Hence, they are still referred to as hardware description languages (HDLs). Another family of languages called hardware verification languages, or HVLs, were created to tackle this problem. OpenVera by Synopsys and e [23] were some widely used HVLs. However, OpenVera was not supported by all EDA vendors and the e language required extra effort to learn which engineers tend to avoid. While these languages are still in use, they are no longer the "cool kids in the block".

EDA companies grouped together to form Accellera, which formulated a new language called SystemVerilog. With the HVL features of OpenVera and HDL features of Verilog, SystemVerilog was widely accepted as a verification language as it was easy to learn and provided all features required for a verification environment such as constraint randomization, assertions and functional coverage. Object oriented programming features help write SystemVerilog testbenches that are reusable and scalable. Most of the popular verification methodologies are based on SystemVerilog which also makes this language an extremely useful skill for a verification engineer.

It is said a numerous times and with good reason, verification takes a significant amount of design time [2]. A verification methodology, or a way to perform verification, should be formulated to reduce the time-to-market and efforts to perform quick and reliable verification. Companies formulated their own methodologies, the top ones being from the top three EDA companies: open verification methodology (OVM) by Mentor Graphics and Cadence, universal reuse methodology (URM) from Cadence, and verification methodology manual (VMM) from Synopsys. Each of these came with their own set of libraries, offered everything needed to write reusable testbenches and were all based on SystemVerilog. Verification engineers therefore could not choose one methodology to stick to. Again, Accellera introduced to the world a new methodology that was industry standard and universally accepted using OVM as foundation and called it the universal verification methodology (UVM).

The UVM is a collection of base classes written in SystemVerilog which can be used to write testbenches which are reconfigurable and reusable [24]. UVM libraries are offered by all vendors. Framing a universal convention enabled engineers to write

testbenches with generic components usable by different designs and portable into different EDA tools [25].

2.3 Coverage Metrics

As important it is to test an RTL design as it is to test the quality of the testbench. Terms such as reusability and scalability describe physical characteristics of the testbench or the verification environment. The quality of test inputs used to verify the design is measured by a class of metrics called coverages. Each coverage metric gives a different view of the test performed and any one metric on its own seldom gives the whole picture on the completeness of verification. Three kinds of coverage metrics are described in the following paragraphs.

One way of knowing if the testbench has done its work is to check how many lines of RTL code it has activated. For example, an RTL code for ALU could have a 'case' statement to perform different operations depending on an opcode. A testbench could be said to have done its job if it generates all values of legal opcodes to the RTL during testing because then all legal operations of the ALU would have been tested by the testbench. All cases under the case statement would have been visited or covered by the testbench. Such a way of measuring the percentage of lines covered by a testbench is called code coverage. It is expected for a testbench, or a set of testbenches, to have 100% code coverage, and anything less should have solid justification, such as a feature not supported by a particular configuration of the design and therefore certain parts of the code can never be activated. Code coverage is measured automatically by EDA tools.

While code coverage is necessary for verification, it is not sufficient to conclude the process. Consider the ALU example again and let one of the operations be addition. Code coverage can detect if the testbench ever tested the addition operation on the ALU. However, it can give no information on the values the testbench passed on for addition. It would be useful to know if the testbench, especially one which randomizes its test vectors, has generated values in all ranges, in all combinations and of all corner cases. It is also helpful to know if the testbench generated any illegal inputs. This can be achieved with functional coverage [24]. Functional coverage is a user-defined and specification-based approach to measure completeness of verification. It helps in keeping track of the test cases and combinations of test cases that the DUT has been subjected to. It is very useful in detecting holes in verification and is more helpful in catching bugs. Functional coverage is, however, prone to human errors. It is also supported by only a handful of languages, SystemVerilog being one.

SystemVerilog offer another language construct called assertions, or more commonly known as SystemVerilog assertions (SVA). Assertions are basically used to track the behaviour of the RTL design. They can be used to check if two signals are equal or not equal or if any signal encounters certain values at some point of time and take necessary action. The actions could be a warning message or terminating the simulation entirely. The set of SVA could be added in our testbench and checked if

all assertions have been tested. This is assertion coverage. In this thesis only code and functional coverages are considered.

2.4 SPARC Architecture

Scalable processor architecture or SPARC is an RISC type instructions set architecture. The design goal of SPARC architecture was to ease hardware implementations of pipelined processors. SPARC has a register window support which enables high performance by reduction of memory load and store instructions over other RISC architecture.

The SPARC architecture version 8 includes the following principal features. These features are taken into consideration for processor verification.

1. All instructions are 32 bit wide and the bit fields are formatted or encoded in three major ways. Instruction formats are used to determine type of instruction and are exploited in measuring instructions functional coverage. The three major formats are:
   - Format 1: For CALL instruction.
   - Format 2: For SETHI and Branch instructions.
   - Format 3: For all other instructions.

   For more details on instruction formats, please refer the SPARC v8 manual [26].

2. Fewer addressing modes, only load and store can access memory. A memory address can only be accessed by registers or register + immediate offset.

3. Triadic register addresses: Almost all SPARC instructions (exception NOP) operate on two operands and place the result in third register.
   Example: ADD rs1,rs2, rd
   rs1: Source 1 register
   rs2: Source 2 could be a register or a constant immediate
   rd: Destination register.

4. Delayed control transfer: An instruction after the branch or jump is always fetched by the processor and execution of that delayed instruction depends on the control-transfer instruction's "annul" bit. It is also known as delay slot instructions
   Example: BA Label 1
   ADD %G1, %G2, %G3
   In the above instruction even after branch always (BA) instruction execution, Integer Unit (IU) will execute addition and then program counter (PC) will jump to address of Label 1.

5. Trap handlers: All traps are vectored through a table with allocation of a fresh register window. Trap/an interrupt is used for I/O subroutines or graceful exit

of programs.

A SPARC compliant processor contains an integer unit, floating-point unit and an optional coprocessor unit with separate registers. Separate integer, floating-point and coprocessor unit implementation allows maximum concurrency thereby improving performance. A SPARC processor has two modes, a user and supervisor mode. When software is in supervisor mode, it can access certain instructions, registers which are also known as privileged instructions. Privileged instructions cannot be executed in user mode. User mode software attempting to execute privileged instruction will result in a trap.

The integer unit (IU) executes integer arithmetic instructions and calculates load and store address computation. In addition to this, IU maintains program counter and controls instruction execution of floating-point and coprocessor. Floating-point unit (FPU) performs floating point arithmetic operations. FPU has 32 floating-point registers (or 32 f registers) each of size 32-bit. SPARC does not require all aspects such as overflow and underflow for instruction set conforming to the IEEE standard for binary floating-point arithmetic, ANSI/IEEE standard 754-1985. If FPU is absent in the processor, an attempt to execute floating point instruction generates an fp_disabled trap. The coprocessor (CP) is an optional unit in SPARC architecture. Coprocessor like other units has its own set of registers. Coprocessor load/store instructions are used to move data between the coprocessor registers and memory. If CP is not available then attempt to run coprocessor instructions generates a cp_disabled trap just like FPU trap. Upon trap, trap handlers will take the control of the execution. SPARC processor comprises of integer, floating and an optional coprocessor units each with own separate registers. SPARC instructions set fall into six different categories of three different instruction set formats.

1. Load/store
2. Arithmetic/logical/shift
3. Control transfer
4. Read/write control register
5. Floating-point instructions set
6. Coprocessor instructions set

2.5 LEON Processor

LEON-3 is a 32 bit processor core designed and distributed by Cobham Gaisler that conforms to the IEEE-1754 (SPARC V8) architecture. Following are the main features of the processor: Harvard architecture with 7-stage pipeline [27], separate cache lines for instruction and data, memory management unit (MMU), hardware multiplier and divider, on-chip debug support and multi-processor extensions. LEON-3FT is another flavour of the processor which is a fault-tolerant version for protection of on-chip memory from radiation in the space. The block diagram of LEON-3 processor is shown in Figure 2.3.

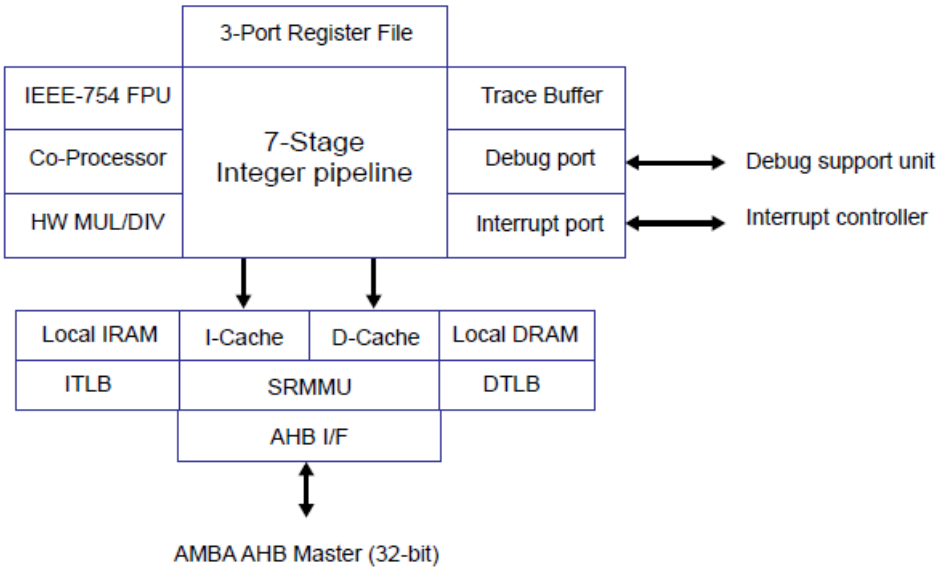

Figure 2.3: LEON-3 processor block diagram [28]

2.5.1 7-Stage Instruction Pipeline

Following are the 7 stages of the LEON-3 integer pipeline which uses a single instruction issue pipeline with 7 stages which are shown in Figure 2.4.

1. Instruction fetch: Instructions are fetched in this stage of the pipeline from the cache or main memory through AHB bus. After instruction is fetched, it is being passed onto decode stage of the pipeline.
2. Instruction decode: Instructions are decoded additionally, target address for the CALL and branch instruction are generated
3. Register Access: After decoding, operands are determined. In register access operands are read from register file.
4. Execute: Execute stage executes arithmetic, logical, shift operations. Execute stage is also responsible for the address generation of memory instructions (Load/store).
5. Memory: There are 2 different cache in LEON-3. Data cache and instruction cache. Instruction cache holds the instruction whereas data cache holds the program data and results. In memory stage of the pipeline data cache is read or written.
6. Exception: Traps and interrupts are of utmost importance for processor to work flawlessly. Therefore, upon receiving them they need to be answered/resolved. This is taken care in the exception stage of the LEON-3.
7. Write-back: Instruction results are written back to the register file.

2.5.2 Forwarding

LEON-3 processor is an advanced 7 stage integer pipeline processor, for that reason, we choose a block of the processor which was small yet important for Cobham Gaisler to start with the verification and coverage. A data hazard in the processor

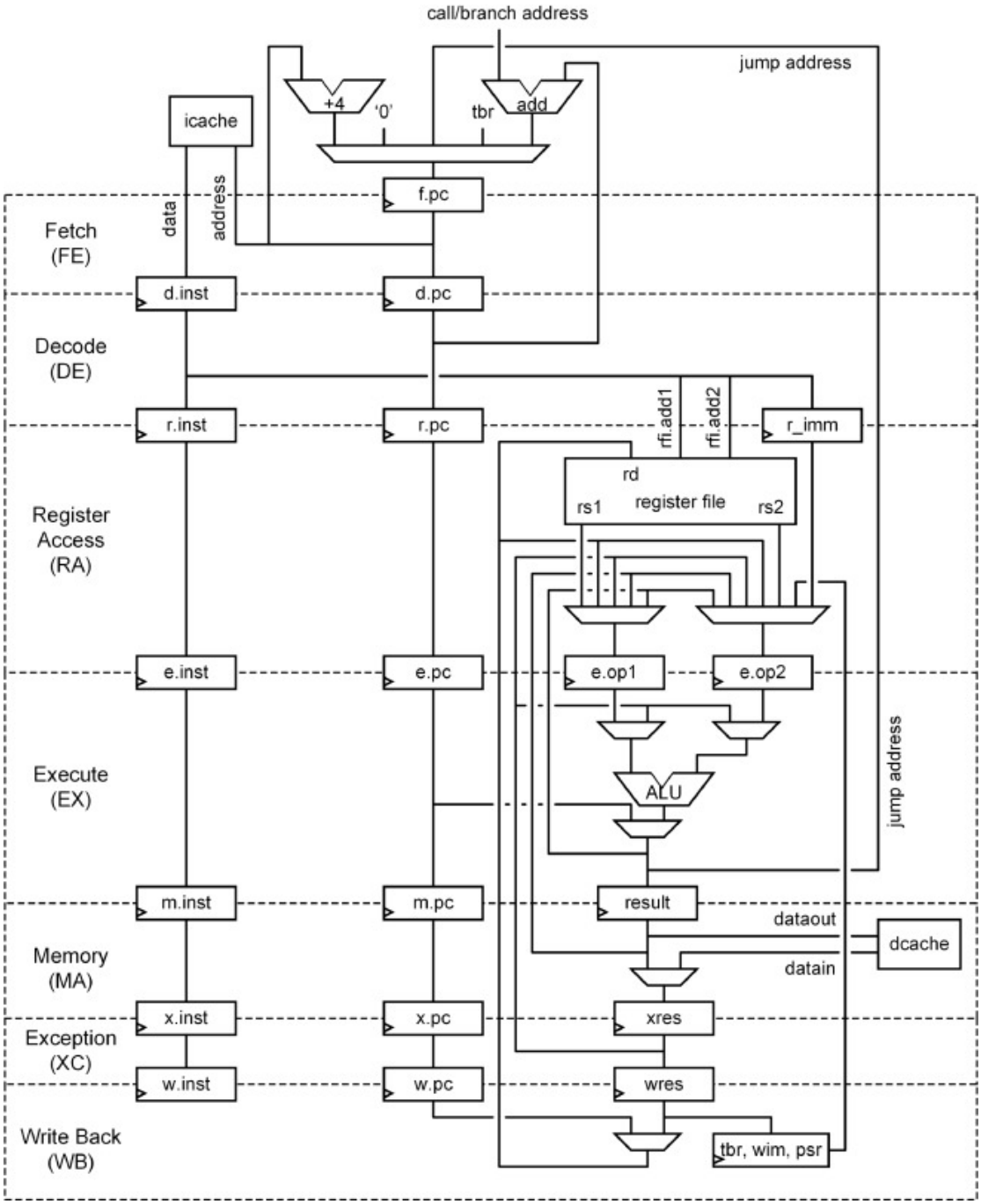

Figure 2.4: LEON-3 processor 7 stage integer pipeline [29]

arise due to program dependencies, i.e, when instructions that exhibit data dependency modify data in different stages of a pipeline. This data hazard later lead to pipeline stall yielding pipeline operation to wait for the results of an earlier operation due to operand dependencies. To avoid this, operand forwarding is adopted in pipelined CPUs. Forwarding is an optimization technique to limit pipeline stall and thereby improve performance. Forwarding is explained with 3 simple example instructions [30].

Example:
Instruction, destination register, source register 1, source register 2
ADD R1, R2, R3
SUB R4, R5, R1
AND R6, R1, R7

Table 2.1: Instruction pipeline execution without forwarding

		1	2	3	4	5	6	7	8	9	10	11	12
ADD	R1,R2,R3	FE	DE	RA	EX	MA	XC	WB					
SUB	R4,R5,R1		FE	Stall	Stall	Stall	DE	RA	EX	MA	XC	WB	
AND	R6,R1,R7			FE	Stall	Stall	Stall	DE	RA	EX	MA	XC	WB

Table 2.2: Instruction pipeline execution with forwarding

		1	2	3	4	5	6	7	8	9
ADD	R1,R2,R3	FE	DE	RA	EX	MA	XC	WB		
SUB	R4,R5,R1		FE	DE	RA	EX	MA	XC	WB	
AND	R6,R1,R7			FE	DE	RA	EX	MA	XC	WB

The first ADD instruction performs addition of values in registers R2 & R3 and result is stored in register R1. SUB instruction needs the result of addition in R1 register in order to perform subtraction. This condition will add additional latency in the processor pipeline if R1 was to fetched from memory. This is shown in Table 2.1 using example instructions. Stall indicates that due to data hazard processor cannot execute instruction until data from previous instruction is available in memory. To avoid these stalls forwarding technique is useful in which the results of the previous instructions can easily be forwarded back from pipeline to ALU/IU, thereby, limit pipelines stall yielding improvement in performance. Similarly, next processor instruction AND is required to have R1 which is forwarded from the different pipeline stage. As shown in Table 2.1 and Table 2.2 that without forwarding 3 instructions will take 12 processor cycles to execute, whereas with forwarding instructions will be executed in 9 processor cycles. Table 2.2 shows that first forwarding is for value of R1 from EXadd (Execute) to EXsub shown in green color. The second forwarding is also for value of R1 from MAadd (Memory) to EXand (Execute) shown using blue color. It is important to verify these forwarding paths from different stages of pipeline, therefore, we had focused on forwarding coverage in functional coverage. Details about forwarding functional coverage is explained in section 3.8. Functional coverage for forwarding path test forwarding paths from execute stage (shown in brown color), memory stage (blue), exception (green) and write-back stage (red). in LEON-3 processors from different pipeline stages. Figure 2.5 shows LEON-3 forwarding paths which were tested using different instructions and later functional coverage for forwarding paths were measured.

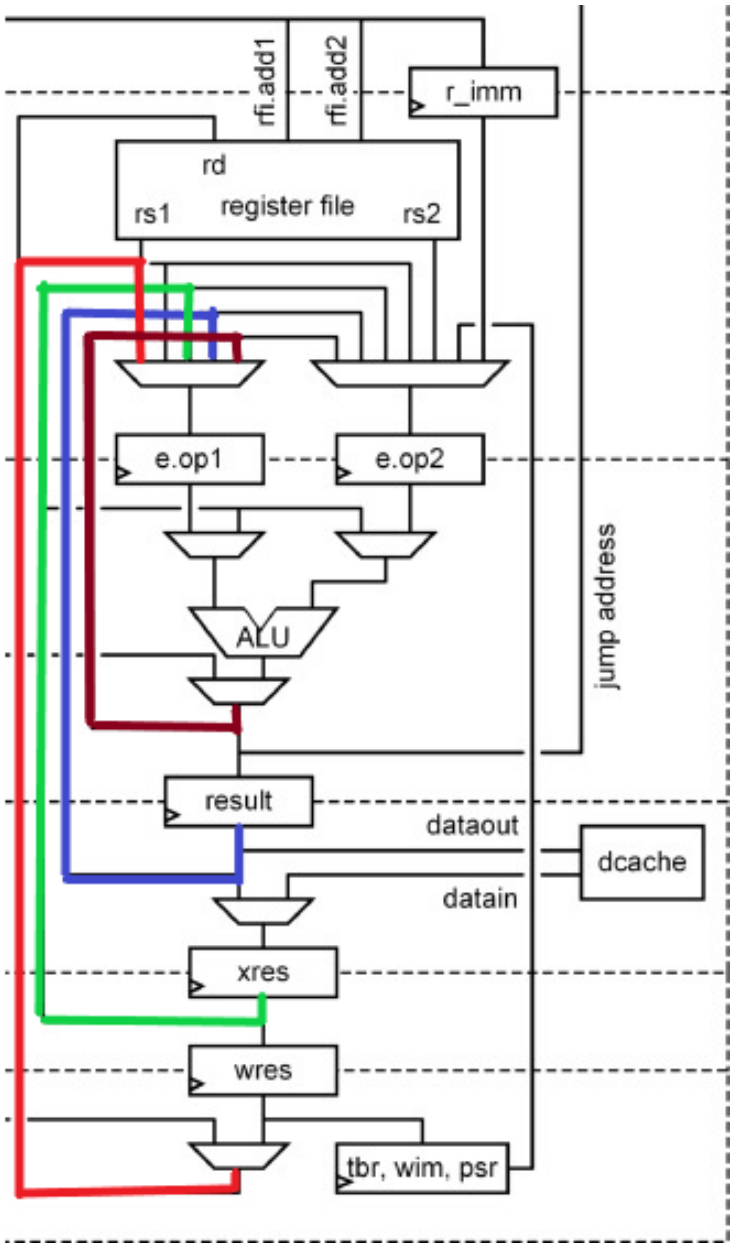

Figure 2.5: LEON-3 processor forwarding paths from different pipeline stages

2.5.3 Branching

When an instruction makes a decision to perform one of two or more choices it is known as branching, and such instructions are called as branch instructions. SPARC architecture instructions set have branching instructions to control the flow of the program. After branch, program counter (PC) jumps to an address specified if branch is taken otherwise it continues the serial execution of the program. Branch instructions fall under control and transfer category.

It was important to include branching instructions to this thesis work. Certain rules must be taken into consideration when adding branch instructions in a random

way. Adding branch instruction to a random program generator must not create an infinite loop and program must always exit gracefully.

2.6 Verification Methodologies for LEON-3

As discussed in the previous sections, there are several verification techniques. For LEON-3, testing is performed by running test programs. A C program is loaded into a RAM model in the design and the RTL is simulated by executing the loaded program.

Directed testing is adopted by Cobham Gaisler for LEON-3. Simulation based directed verification approach in software verification terminology is also known as smoke testing. In this technique, prior deterministic test cases are generated to test the design. At Cobham Gaisler, this is done using hand-written C programs, one of them being a self checking program called Systest that tests different features of the processor. Different functions are called from the Systest's main.c top level to verify different units such as adder, multiplier, divider, timer, interrupt etc. The C program file is compiled and then assembly instructions are simulated on RTL model of the processor. This self checking directed testing program generates input stimuli, calculates expected result and compares the actual and expected results. Directed testing is fairly straight forward, easy and simple, however directed testing is not suitable for exhaustive verification. With directed approach corner case bugs cannot always be discovered. We have shown code and functional coverage of Systest in the results section and compared it with random program generator.

As the name suggests, in random verification technique, the executed assembly program is random in nature. In software verification terminology it is also called as Gorilla testing. Purpose of this technique is to generate assembly programs with valid random instructions and data. By doing this, we put verification in a state which is likely to discover hidden scenarios which can escaped one's notice in directed testing. One such dynamic random program generator is proposed by Corno *et al.*[31] and Hudec [32] to improve fault coverage using random program generator.

3
Methods

As described in section 1.3, the objective is to achieve the same (or if possible better) measure of code coverage with lesser effort than software driven directed test cases and with a satisfactory level of functional verification. Instructions can be generated with a certain level of randomness that can reach corner cases and achieve code coverage faster.

The satisfaction of functional verification can be achieved using functional coverage, which is another objective of this thesis. SystemVerilog Functional Coverage is used to check if certain functionality or test case has been covered by the random program generator.

3.1 LEON-3 Verification Framework

LEON-3 verification framework is designed and developed as part of master thesis work is presented in the Figure 3.1. The proposed framework is controlled though a SystemVerilog tester class. Through this class, certain parameters and coverage goal can be set. These arguments control the random program generator and framework flow. LEON-3 verification framework consists of generating random programs, generating SRECORD software binary of these random programs. RTL simulation with the random program and verification of these random program using TSIM tool [33]. TSIM tool is capable of emulating SPARC architecture based processors. Following TSIM verification, functional coverage goal is measured and checked. If functional coverage goal measured is less than the coverage goal set in the SystemVerilog tester class then entire flow is rerun but with more instructions generated from random program generator. This process is iterative until the functional coverage goal is satisfied. The process is terminated immediately if any simulation error is encountered. Due to limited instructions support in random program generator it is not possible to achieve more than 47% of functional coverage. When coverage goals are reached with no simulation errors, code and functional coverage results are generated using simulators in HTML format. In the following sections, LEON-3 verification framework is discussed in detail.

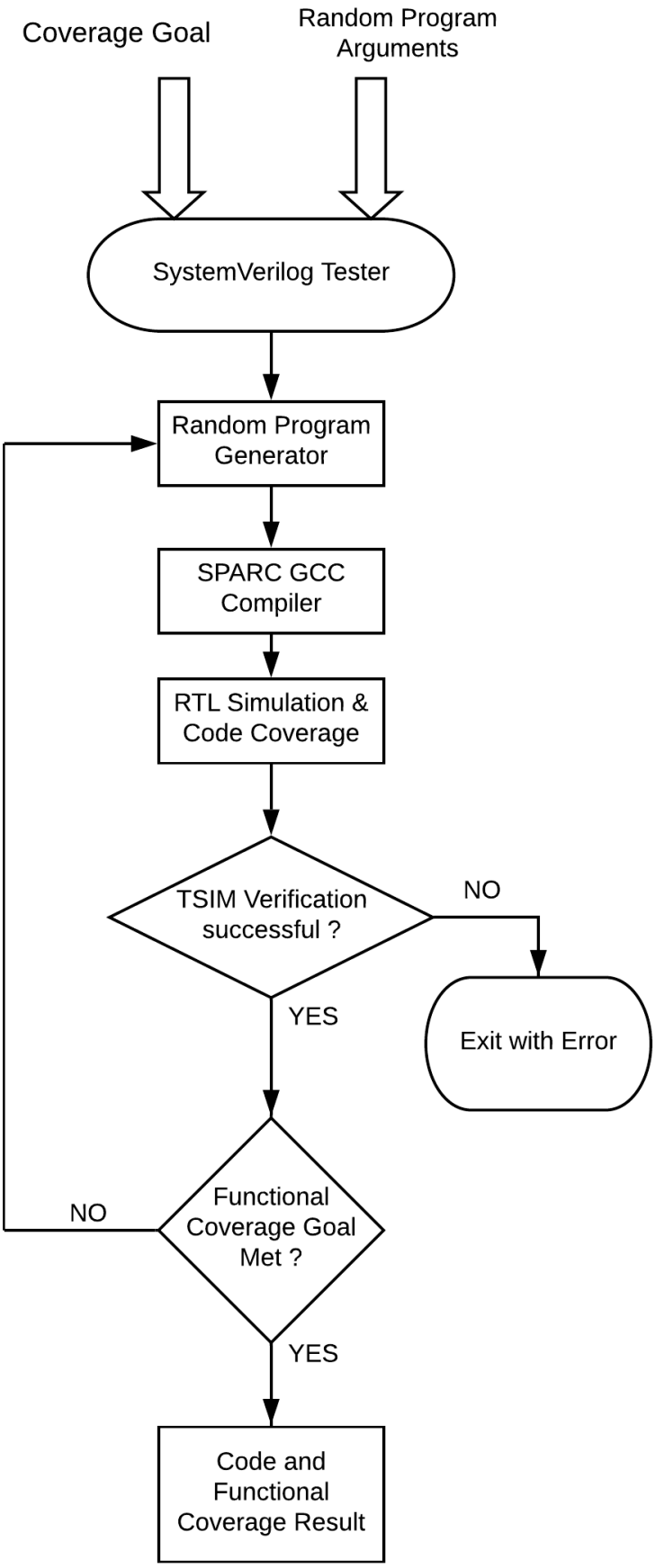

Figure 3.1: LEON-3 verification framework flow

3.2 SystemVerilog UVM

Since functional coverage was implemented using SystemVerilog, it was decided to implement the universal verification methodology or UVM as an added step to this thesis. A block diagram of the verification environment is shown in Figure 3.2. Each block represents a component or a class which is an extension of a UVM base class. The environment has several components each serving a different purpose.

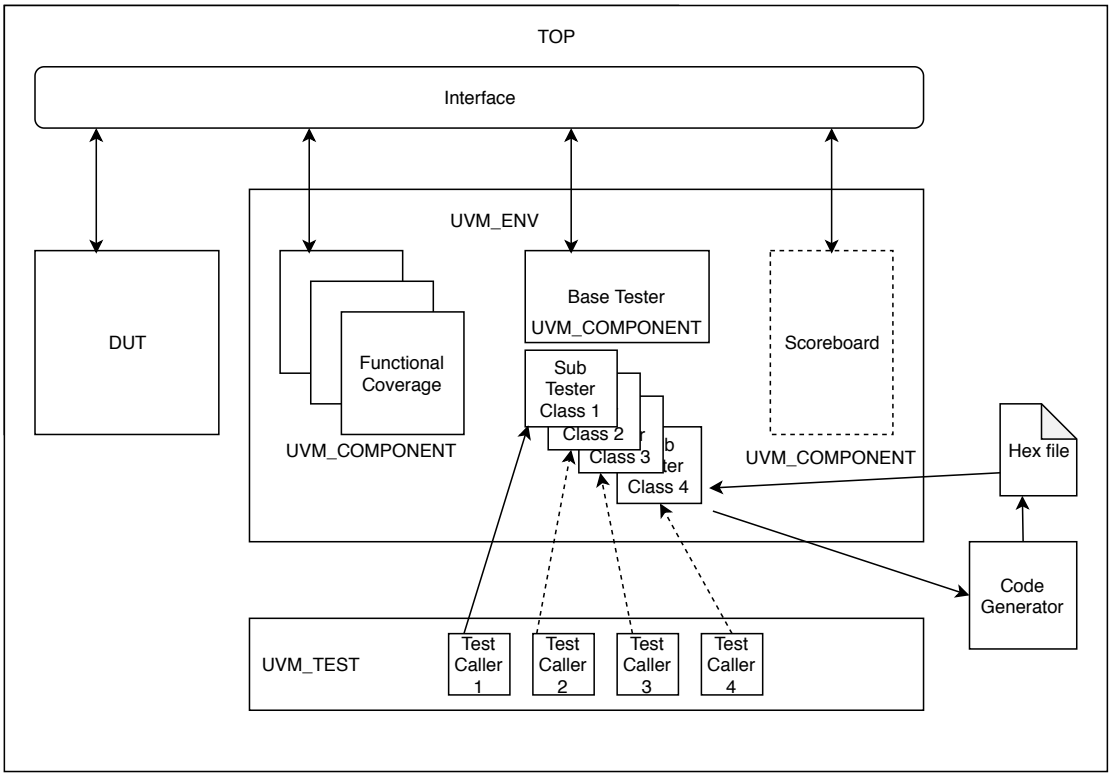

Figure 3.2: The Verification Environment

The interface consists of signals that connect the DUT and the testbench. Interface is a SystemVerilog construct used to encapsulate signals in a module. These signals can be shared by other modules by instantiating the interface ensuring better connectivity between components. Interfaces can also contain constants and functions useful for the testbench. The interface of the LEON-3 testbench has a function to splits opcodes into different fields usable by Functional coverage modules.

The UVM_ENV encapsulates the base tester and functional coverage modules, which are extensions of the UVM_COMPONENT base class. The base tester generates an assembly program with random instructions and performs all functions explained in section 3.1. The number of instructions and type of instructions to be included in the assembly file can be set by the sub tester module which is an extended class of the base tester. The required sub tester can be called creating a class that extends UVM_TEST base class, connecting them together and specifying the name of the sub tester class as the value for +UVM_TESTNAME option in

the compile command. Three modules for functional coverages have been defined to cover functionality in three different way. These are defined in section 3.8.

The scoreboard has been shown as a dotted box because its function is taken care of by the base tester class. The duty of the scoreboard is to verify the functionality of the RTL, which is done using TSIM here. The scoreboard, if made into a separate module, will also be a child class of the UVM_COMPONENT.

3.3 Random Program Generator

Due to increase in complexity of the microprocessor design it is impossible to cover all the corner cases and it requires a lot of time to perform directed testing. The longer time it takes to verify processor affects time to market, it is very time consuming to write all tests manually. This results in the necessity of a random program generator. Random program generator is not a new method in verification domain.

In our thesis work, random program generator is a tool command language (TCL) [34] based program that generates random test programs. These programs are generated in SPARC-compliant assembly instructions. Generated random program is then later converted in binary using sparc-elf-gcc compiler. TCL based program generator is based on full randomness of instruction, registers and immediate data, however, random program generator must follow several rules to generate error free compatible code.

1. Instructions generated must be valid with legal syntax, operands and data.
2. A number of branches or subroutine calls must not create an infinite loop
3. Program generated by random generator must not have instructions that would cause processor to trap such as divide by zero and other cases must be taken care to avoid traps.

Random program generator is a user controlled program with an ability to generate instructions. Random generator is a standalone implementation meaning that we can work on improving it without having to worry about test setup. Following instructions can be generated using random program generator

1. Arithmetic Instructions (ADD, SUB, MUL, DIV)
2. Shift Instructions (SLL SRL)
3. Logical Instructions (AND NOT XOR XNOR OR)
4. Multiply-Accumulate Instructions (UMAC SMAC)
5. Branch Instructions
6. Program mixed of all the above instructions

Random program generator picks up any random instructions from the instructions library with random source as well as destination register. There are 4 different sets of registers available in SPARC architecture. Global registers (%G0-%G7), input registers (%I0-%I7), output registers (%O0-%O7) and local registers (%L0-%L7).

Instruction	Source register 1	Source register 2 or Immediate data	Destination register
ADD	%G1	%G2	%G3
ADD	%G4	144	%L3
ADD	%O0	%G1	%I1

Table 3.1: Example instruction generation using random program generator

Example: Above variations of the ADD instruction is possible using random program generator. Similarly, all possible combinations of instructions including branch instruction are generated using random program generator, refer Appendix 1.

3.4 SPARC GCC Compiler

SPARC GCC compiler is a bcc cross compiler for LEON-3 processor. It is based on the GNU compiler tools (GNU stands for GNU's Not Unix, a recursive acronym) and newlib standalone. The compiler is used to generate object file from the generated assembly program. Object file is then later used to generate an SRECORD(SREC) file. SREC file conveys binary information in ASCII hex text form of executable code and data. SREC file is also used to run the LEON-3 SoC RTL model simulation and emulate the program behavior on TSIM. In the next section, how this SREC is being used for RTL simulation and TSIM will be discussed.

3.5 RTL Simulation

LEON-3 verification environment consists of a full model of a System-on-chip (SoC) device with simulation models of peripheral components such as memories.

These RTL models are simulated using EDA tools and application software. Processor simulation model then read application SREC and execute random program instruction one after another on processor. Every random program generated has a trap at the end of the program for a graceful exit. Upon trap, simulator stops simulation and exit with meaningful message on the simulator console. At the end of the simulation, simulator tool is instructed to measure code coverage using simulator's code coverage feature. Simulator tool then generate code coverage report with different coverage types which are branch coverage, statements coverage and toggle coverage. Code coverage measures the number of lines of code that has been hit/exercised while running a suit of tests.

Simulation setup is driven from the processor side by software application which, here, is an assembly program created using random program generator. Current verification system at Gaisler is driven by C application software. This creates a possible hole in the verification coverage since the generated applications depend on compiler behavior. Compiler behavior is not static over time because of updates in toolchain the generated machine code also changes and there are also large changes

in characteristics of code complied from, for example, C and Assembly. Because of this it is desirable to break the dependency between the processor verification environment and compilers. The proposed verification framework in this thesis work breaks this dependency since application software is generated directly in assembly with random in nature.

3.6 Code Coverage

As discussed earlier, code coverage measures the number of lines of code that have been hit or exercised while running a suit of tests. There are 4 different types of coverage results generated which are statement, branch, toggle and focused expression coverage (FEC). Statement coverage is the count of how many times a given statement is executed during simulation. Example in Figure 3.4 shows that when the result is positive only 5 lines out of 7 will be exercised, resulting 71% of statement coverage.

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Figure 3.3: Statement coverage

Branch coverage is related to branching constructs such as “if” and “case” statements. True branch and “AllFalse” branch execution are measured. In order to achieve 100% branch coverage, each branching statement in the source code must have taken its true path, and every AllFalse branch must have been taken. As shown in Table 3.2 that depending on the value of input A branch outcome is decided resulting branch coverage of 33% in case of value of A is 2 and 67% in the other case.

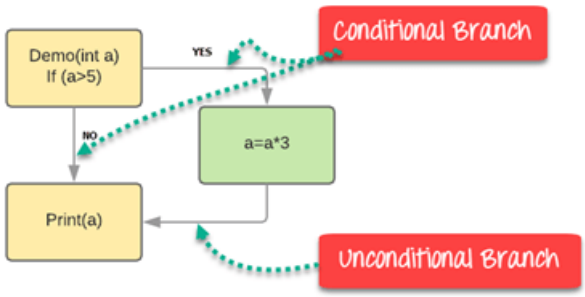

Figure 3.4: Branch coverage

Toggle coverage is the ability to count and collect changes of state on specified nodes. Toggle coverage reports the number of times each bit of a register or signal

Value of A	Output	Branch Coverage
2	2	33%
6	18	67%

Table 3.2: Example of branch coverage percentage

has toggled its value. An example of toggle coverage is shown in Table 3.3 [35]. Toggle coverage of 12 bit register state is shown in the table, 0->1 transition shows that particular bit has changed its value from 0 to 1 and vice versa.
Example: reg [10:0] state

Bit	0->1	1->0
reg[10]	12	11
reg[9]	5	5
reg[8]	9	10
reg[7]	31	30
reg[6]	31	30
reg[5]	31	30
reg[4]	31	30
reg[3]	31	30
reg[2]	31	30
reg[1]	31	30
reg[0]	31	30

Table 3.3: Example toggle coverage of state register

Focused expression coverage(FEC) is a row based coverage metric which evaluates the contribution of each expression input to the expression's output value. FEC measures coverage for each input of an expression. If all inputs are fully covered, the expression has reached 100% FEC coverage. In FEC, an input is considered covered only when other inputs are in a state that allow it to control the output of the expression. Further, the output must be seen in both 0 and 1 states while the target input is controlling it. If these conditions occur, the input is said to be fully covered. The final code coverage condition and expression coverage FEC coverage number is the number of fully covered inputs divided by the total number of inputs.

3.7 TSIM Verification

TSIM verification is the most important step of the verification framework. TSIM is an instruction-level simulator capable of emulating LEON processor based systems. with TSIM we could emulate LEON-3 application software generated by random program generator. TSIM is used as a golden reference for verification and it was developed by an independent team thus there is some assurance of its results. We used TSIM to verify the correctness of the simulation results obtained from the RTL simulation. Verification is carried out by comparing result of the instructions from

RTL simulation with TSIM result of the same instructions. Simply put, comparison of RTL simulation results with TSIM results. If the comparison is successful that means simulated RTL behavior is as intended and produces valid results. After successful TSIM verification, only then we proceed to measure functional coverage. If there is a mismatch between the result of simulation with TSIM result, appropriate error is flagged and framework flow stops at that point.

3.8 Functional Coverage

While code coverage is essential, it is not enough for knowing verification completeness. For example, a line of code implementing addition can get hit when a test case involves an addition operation. This gives no information regarding the operands of the addition. For the LEON-3 processor, the operands can be two registers or one register and an immediate value. Code coverage cannot detail if all these combinations have been encountered. There are also illegal inputs or combinations of inputs which cannot be detected with just code coverage. All these needs can be catered by using functional coverage.

Unlike code coverage, functional coverage is engineer-defined. This means that the verification engineer decides the functionalities that should be tested. In SystemVerilog, this is done by creating 'covergroups' and 'bins'. Three separate files have been created for covering different aspects of functionalities. They are listed below:

- **Instructions:** Here it is checked if all allowable instructions are executed at least once. One illegal bin is defined for Unimplemented instructions.

- **Instruction-Operand combinations:** All allowable instruction - operand combinations are checked. Different instructions take three, two, one or no operands. All these combinations are defined using 'cross coverages'.

- **Forwarding paths:** Bins can be used to track transitioning of values. Such bins are called transition bins. Transition bins are used to check if all forwarding paths are activated with all registers as operands.

As discussed earlier, user argument from SystemVerilog tester determines the outcome of the random program generator. For example, if user is interested to generate and test only a particular set of instructions then appropriate user argument needs to be provided to the generator. Currently, random program generator can generate instructions discussed in the section 3.3. Additionally, user can also determine the number of random instructions. After input program is generated using random generator it is fed to SPARC-elf-gcc compiler to generate two different output files. .SREC and object dump files are generated from the compiler. Out of two, one file .SREC is used to run the simulation by instantiating ram memory with SREC file. After simulation run, questa-sim is instructed to measure the code coverage.

Format (3):

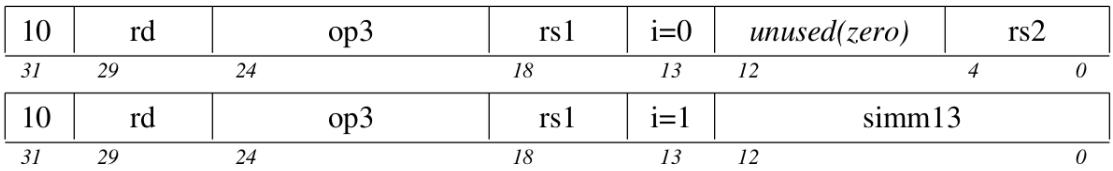

Figure 3.5: Format 3 instruction

bit [31:30]	Instruction format
bit [29:25]	Destination register
bit [24:19]	Opcode of format 3 instructions
bit [18:14]	Source 1 register
bit [13]	i field when i==1 : immediate operand
bit [12:0]	immediate operand
bit [13]	i field when i==0 : address space identifier
bit [12:5]	Address space identifier supplied by load/store alternate instruction, else unused
bit [4:0]	when i==0 : source 2 register

Table 3.4: Instruction opcode

SPARC assembly instructions are divided into different formats. Format 1, format 2, format 3. Description of format 3 instruction is as shown in Figure 3.5. Every bit of the opcode is explained in Table 3.4. Second generated file .object dump is a disassembly of the random program. This disassembly contains instructions with their opcodes. A TCL script is used to convert this file into a file with instructions in their 32 bit binary format. This file is read by the base tester class in the UVM framework (figure 3.2) one instruction at a time. Each instruction is passed on to the interface where is it separated into fields as per its format. These fields are monitored by the functional coverage classes. Each field or a combination of fields is a coverage point. Bins are declared for different values under each coverpoint and are hit when the respective values are detected from the instructions read by base tester. Functional coverage is measured by counting the number of bins hit at least once.

In the verification framework, functional coverage is measured and compared with the goal set in the SystemVerilog interface. If the functional coverage is less than the coverage goal then entire verification framework is re-run but this time with additional instructions generated from random program generator. The framework continues until functional coverage goal is met thereby making verification process robust and self automated.

4

Results and Discussion

This section discusses and compares the coverage results obtained from running a benchmark software (Systest) and a program generated with the random program generator. Both Code and Functional coverages were extracted. A brief information about Systest is discussed in section 2.6.

4.1 Code Coverage

Code coverage was first measured for Systest program. Figure 4.1 is a cut-out of code coverage report in HTML format. The colour convention in the report is red for coverage less than 50%, yellow for between 50 - 90% and green indicates >90% coverage. It should be noted that LEON-3 SoC is a complex processor with a lot of peripheral, therefore, we restricted ourselves to the processor only in this thesis work. The code coverage for the processor (p0) with Systest is 67.5%.

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Condition	Toggle	Assertion
TOTAL	61.36	73.92	60.05	35.80	37.03	100.00
p0	67.56	100.00	--	--	35.12	--
iu	42.34	59.62	56.44	28.29	25.00	--
mgen/mul0	75.24	95.12	68.60	28.57	83.93	100.00
mgen/div0	78.04	91.89	78.57	42.85	98.85	--
c0mmu	64.13	80.54	62.02	40.26	37.85	100.00

Figure 4.1: Code coverage of systest

A set of 1000 instructions was generated and code coverage was measured for the same. The snapshot of the HTML coverage report is shown in Figure 4.2. The code coverage for the processor is 63.9%. It can be seen how with a lot less effort, i.e, nearly 2.5% of the number of instructions in Systest, the code coverage obtained was only 3.6% less than that obtained with Systest.

Another set of 3000 random instructions was generated and tested for code coverage. Figure 4.3 shows that coverage report and the coverage obtained in 64.8%.

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Condition	Toggle	Assertion
TOTAL	46.80	58.65	39.26	17.60	18.52	100.00
p0	63.90	100.00	--	--	27.81	--
iu	31.72	50.39	40.48	15.88	20.13	--
mgen/mul0	77.10	95.60	74.72	31.25	83.93	100.00
mgen/div0	27.79	48.64	42.85	7.14	12.54	--
c0mmu	46.96	62.27	36.84	18.51	17.16	100.00

Figure 4.2: Code coverage of 1000 random instructions

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Condition	Toggle	Assertion
TOTAL	47.96	59.74	41.38	18.17	20.50	100.00
p0	64.86	100.00	--	--	29.73	--
iu	33.84	52.07	44.49	17.13	21.66	--
mgen/mul0	77.10	95.60	74.72	31.25	83.93	100.00
mgen/div0	74.14	94.59	88.09	14.28	99.61	--
c0mmu	47.01	62.27	36.84	18.51	17.42	100.00

Figure 4.3: Code coverage of 3000 random instructions

Number Of Instructions	Code Coverage %	Runtime in Seconds
Systest 30000 Instructions	67.86	67
1000 Random Instructions	63.90	23
3000 Random Instructions	64.86	33
5000 Random Instructions	65.32	37
10000 Random Instructions	65.71	42
15000 Random Instructions	65.72	49
25000 Random Instructions	65.81	55

Table 4.1: Code coverage results for different sets of instructions

The above observations from Table 4.1 show that code coverage can be easily achieved by randomizing instructions. Code coverage was less than Systest because of the fact that random program generator does not generate all instructions set. While increasing the number of instructions help achieve better code coverage, smarter test vector generation can help achieve good coverage values also. This could be done with constrained randomization, defining rules for instruction combinations, arranging instructions with registers and immediate values and many other ways.

The results also highlight the drawback of code coverage. It was seen how easily good code coverage values, which were on par with a benchmark software, could be achieved with so less an effort. A set of mere 1000 instructions is, of course, nowhere near sufficient to verify a complicated system. Functional coverage was therefore measured for the test vector sets.

4.2 Functional Coverage

Functional coverage was first measured for Systest, the HTML report of which is shown in Figure 4.4. The colour convention in the report is red for coverage less than 50%, yellow for between 50 - 90% and green indicates >90% coverage. The report lists three ways in which functional coverage was measured, as described in section 3.8. They are

- **fc_basic_LEON-3:** For instructions.
- **fc_forwarding_path:** For forwarding paths.
- **fc_inst_operands:** For instruction - operand combinations.

The number for instructions and forwarding path coverage are nearly 80%. For instruction-operands combination, the coverage is understandably low. This is because the number of all possible combinations of instructions and operands, registers and immediate, is huge.

Coverage Summary By Instance:

Scope ◂	TOTAL ◂	Cvg ◂
TOTAL	56.32	56.32
leon3_tb_pkg	56.32	56.32
fc_basic_leon3	80.09	80.09
fc_forwarding_path	81.18	81.18
fc_inst_operands	18.65	18.65

Figure 4.4: Functional coverage of Systest

The functional coverage report for 1000 random instructions is shown in Figure 4.5. The instruction coverage is only half of that for Systest, which is quite good considering the reduction in the number of input instructions. The coverage for forwarding paths is lower by 30% as the instruction generator distributes registers

as operands randomly. Instruction-operand combination coverage is very low due to lower number of instructions.

Coverage Summary By Instance:

Scope ◂	TOTAL ◂	Cvg ◂
TOTAL	37.55	37.55
leon3_tb_pkg	37.55	37.55
fc_basic_leon3	58.95	58.95
fc_forwarding_path	34.94	34.94
fc_inst_operands	10.77	10.77

Figure 4.5: Functional coverage of 1000 random instructions

With 3000 instructions, functional coverage is better for forwarding paths. Instruction-operand combinations coverage has a marginal increase. The coverage for instructions is the same.

Coverage Summary By Instance:

Scope ◂	TOTAL ◂	Cvg ◂
TOTAL	40.43	40.43
leon3_tb_pkg	40.43	40.43
fc_basic_leon3	58.95	58.95
fc_forwarding_path	60.75	60.75
fc_inst_operands	10.82	10.82

Figure 4.6: Functional coverage of 3000 random instructions

Another test was conducted with 5000 instructions. 100% functional coverage could be achieved for forwarding paths. Instruction-operand combinations coverage has increased by 1.62%. The coverage for instructions is again the same.

The above results are shown graphically in Figure 4.8. The red dotted lines are the values obtained for Systest and are shown for reference. Three points can be inferred from the Figure 4.8:

Coverage Summary By Instance:

Scope ◂	TOTAL ◂	Cvg ◂
TOTAL	45.40	45.40
leon3_tb_pkg	45.40	45.40
fc_basic_leon3	58.95	58.95
fc_forwarding_path	100.00	100.00
fc_inst_operands	12.39	12.39

Figure 4.7: Functional coverage of 5000 random instructions

1. 100% functional coverage could be achieved for forwarding paths with only 5000 randomly instructions, as compared with 80% coverage with Systest with nearly 30000 instructions.

2. Functional coverage for instructions has remained constant no matter the number of instructions. This points to the limitation in our verification strategy which is that the random program generator does not support all kinds of instructions.

3. For instruction-operand combination, the functional coverage is very low, increasing by a little with increase in number of instructions. This highlights the fact that functional coverage is prone to human errors as it is user-defined. Over enthusiasm should be avoided while creating 'bins' to prevent unrealistic verification goals.

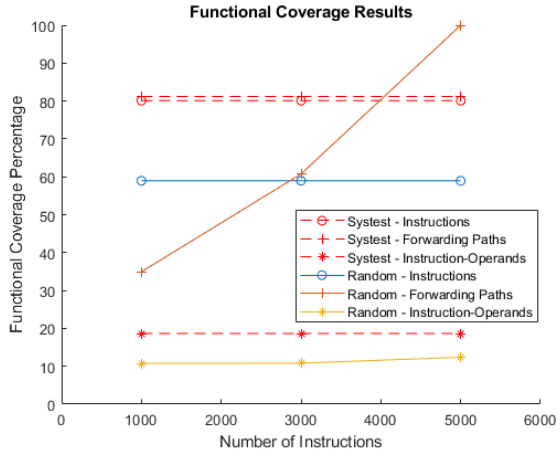

Figure 4.8: Graphical Representation of Functional Coverage Results

5

Conclusion and Future Work

The aim of this master thesis was to build a microprocessor verification framework capable of generating intelligent random programs to verify the LEON-3 processor. Another objective of this thesis work was to improve verification metrics viz code coverage in an efficient way and introduce relevant microprocessor metric and evaluation of the same metric.

The goals of the thesis were achieved using microprocessor verification framework proposed in section 3.1. Code coverage improvement was achieved with less instructions, less runtime and less efforts using random program generator. Verification of these random program generator was carried out using emulation tool TSIM. Another microprocessor metric was also introduced which is functional coverage of the LEON-3. We evaluated existing verification framework of LEON-3 with a new metric (functional coverage).

The coverage results discussed in section 4 shows that better code and functional coverage was successfully achieved with the microprocessor verification framework. Forwarding path functional coverage was introduced and 100 % achieved using new microprocessor verification framework. A key achievement of this framework was 100% forwarding path functional coverage with mere 5000 intelligent random instructions.

The work carried out in this thesis work can easily be adopted to other LEON processors or processors with different architecture. We have also successfully demonstrated in the Appendix 1 that random program generator can also be used to generate RISC-V architecture program in addition to LEON-3 (SPARC) architecture.

5.1 Future Work

Result artifacts show that the microprocessor verification framework is very useful for LEON-3 microprocessor verification, however, there are a few things that needs to be introduced, improved and should be taken into consideration. In this section, we discuss a few realistic future extension to the work.

Code coverage presented in Table 4.1 results section points out that code coverage can easily be improved with less efforts, however, code coverage improvement was

capped at around nearly 65% due to limited instructions support from the random program generator. Not all SPARC instructions set is generated by random program generator resulting in a limit in code coverage achievement. Functional coverage results also points out that fc_basic_leon3 functional coverage was at 58.95% for N number of instructions. This points out to the fact that more instruction support should be added to improve code and functional coverage.

The framework can also be extended to cover more test scenarios such as register windowing. Register window is important feature of SPARC architecture, therefore, coverage for register window and register window instructions (Save and Restore) support could be added in the framework to test register window feature.

The current implementation of the microprocessor verification framework presented in section 3.1 can be redesigned to reuse old random programs. Only new instructions can be added to the old one to improve coverage statistic after every simulation run and also lower simulation runtime. This will enable verification framework to fill the coverage limitation and improve performance.

If the functional coverage goal specified by user is more than 47% in the current implementation, framework will go into infinite loop and will never be able to come outside the verification loop. The reason for this behavior is already discussed which needs support for all the instructions from random program generator. However, a small modifications can easily be made in the code to solve this infinite loop problem.

If the TSIM verification is unsuccessful, at this stage framework will stop the flow with an error. But at this stage user has to manually analyze the simulation logs to check which instruction has caused discrepancies between RTL simulation and TSIM. This method is laborious and needs careful examination of the logs which could be improved.

In order to use this thesis work for other architectures such as RISC-V architecture, one needs to replace emulation tool TSIM with appropriate emulation tool of the architecture. Some modifications are also required to be done in SystemVerilog tester.

Bibliography

[1] W. K. Lam. Hardware design verification: Simulation and formal method-based approaches. *Prentice Hall PTR*, 2008.

[2] L. Wang, Y. Chang, and K. Cheng. Electronic design automation: Synthesis, verification, and test. *Morgan Kaufmann Publishers Inc.*, 2009.

[3] C. Kuznik and W. Müller. Functional coverage-driven verification with systemc on multiple level of abstraction. *Proceedings of DVCON*, March 2011.

[4] A. Gluska. Coverage-oriented verification of banias. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 280–285, New York, NY, USA, 2003. ACM.

[5] Y. Wu, L. Yu, W. Zhuang, and J. Wang. A coverage-driven constraint random-based functional verification method of pipeline unit. In *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, pages 1049–1054, June 2009.

[6] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. In *Design, Automation and Test in Europe*, pages 678–683 Vol. 2, March 2005.

[7] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 182–187 Vol.1, Feb 2004.

[8] B. K., N. Hegde, and L. Pankaj. Design defect diagnosis in a buggy model of sparc t1 processor using random test program generator. In *2015 Fifth International Conference on Advances in Computing and Communications (ICACC)*, pages 3–6, Sep. 2015.

[9] I. Wagner, V. Bertacco, and T. Austin. Microprocessor verification via feedback-adjusted markov models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(6):1126–1138, June 2007.

[10] J. Shen and J. A. Abraham. Verification of processor microarchitectures. In *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)*, pages 189–194, April 1999.

[11] D. L. Dill. What's between simulation and formal verification? In *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, pages 328–329, June 1998.

[12] S. Wang and K. Huang. Improving the efficiency of functional verification based on test prioritization. *Microprocessors and Microsystems*, 41:1 – 11, 2016.

[13] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, and M. Vinov. Industrial experience with test generation languages gar processor verification. In *Proceedings. 41st Design Automation Conference, 2004.*, pages 36–40, July 2004.

[14] S. Palnitkar. *Design Verification with e*. Prentice Hall Professional Technical Reference, 2003.

[15] F. Haque and J. Michelson. *Art of Verification with VERA*. Verification Central, 2001.

[16] Ieee standard for standard systemc language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012.

[17] Ieee standard for systemverilog–unified hardware design, specification, and verification language - redline. *IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline*, pages 1–1346, Dec 2009.

[18] A. Gluska. Practical methods in coverage-oriented verification of the merom microprocessor. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 332–337, July 2006.

[19] J. Ralph. https://jerralph.github.io/riscv-vip/doc/index.html, Accessed Online: 01.01.2019.

[20] R. B. Jones, J. W. O'Leary, C. . H. Seger, M. D. Aagaard, and T. F. Melham. Practical formal verification in microprocessor design. *IEEE Design Test of Computers*, 18(4):16–25, July 2001.

[21] J. Bhadra, M. S. Abadir, L. Wang, and S. Ray. A survey of hybrid techniques for functional verification. *IEEE Design Test of Computers*, 24(2):112–122, March 2007.

[22] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 970–975, June 1999.

[23] S. Iman and S. Joshi. *The e hardware verification language*. Springer Science & Business Media, 2007.

[24] A. B. Mehta. *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[25] Ieee standard for universal verification methodology language reference manual. *IEEE Std 1800.2-2017*, pages 1–472, May 2017.

[26] CORPORATE SPARC International, Inc. The sparc architecture manual: Version 8. 1992.

[27] Cobham Gaisler. https://www.gaisler.com/index.php/products/processors/, Accessed Online: 01.04.2019.

[28] Cobham Gaisler. https://www.gaisler.com/products/grlib/grip.pdf, Accessed Online: 01.04.2019.

[29] R. Bansal and A. Karmakar. Efficient integration of coprocessor in leon3 processor pipeline for system-on-chip design. *Microprocessors and Microsystems*, 51:56 – 75, 2017.

[30] G. Prabhu. http://web.cs.iastate.edu/ prabhu/tutorial/pipeline/forward.html, Accessed Online: 07.01.2019.

[31] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. Fully automatic test program generation for microprocessor cores. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 1006–1011, March 2003.

[32] J. Hudec. An efficient technique for processor automatic functional test generation based on evolutionary strategies. In *Proceedings of the ITI 2011, 33rd*

International Conference on Information Technology Interfaces, pages 527–532, June 2011.

[33] Cobham Gaisler. https://www.gaisler.com/doc/tsim2.pdf, Accessed Online: 07.04.2019.

[34] M. Harrison and M. McLennan. *Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[35] A. Piziali. *Functional Verification Coverage Measurement and Analysis*. Springer Publishing Company, Incorporated, 1st edition, 2007.

A

Appendix 1

Random program generator was developed as part of the thesis work which is a property of Cobham Gaisler. Random Program Generator Examples are listed in the Appendix. Random Program generator can be used to generate different types of instructions, determine number of instructions to be generated. Additionally it is also possible to generate program for other microprocessor architectures SPARC & RISC-V.

Example:
tclsh generator.tcl -help
Description:
Script to Generate Random Instructions for LEON3 and RISC-V Processors

Syntax:
tclsh generator.tcl [-tni <arg>] [-it <arg>] [-processor <arg>] [-help]

Usage:
tclsh: TCL shell <name of the source tcl file>
-tni : Total Number of Instructions to be generated
-it : Type of instructions to be generated
-processor : Processor RISC-V or LEON-3

Example:
The following example will generate total 100 all random instructions for SPARC architecture
tclsh generator.tcl -tni 100 -it all_instructions -processor sparc

Code generated using a random program generator is shown in Figure A.1. Following command was used to generate the program. The program generated is mix of all instructions such as logical, branching, nop and sethi and other instructions. Generated instructions are more than 20 its because of the fact some instructions are required to disable the trap and to make sure exit from the random program. This will make sure that random program generated does not run forever and exit in a graceful way.
tclsh generator.tcl -tni 20 -it all_instructions -processor sparc

Following program generation, .S assembly program is then passed onto SPARC GCC compiler to build the SREC binary.

```
A1:
be  label2
nop
sdivcc %o0, 955, %o6
label2:
bvc,a  A3

A3:
umul %o3, %o6, %l3
be,a  label4
nop
udivcc %o2, 636, %o2
label4:
bvc,a  A5

A5:
be,a  label6
nop
sdiv %i6, 574, %l7
label6:
be,a  label7
nop
sdivcc %l1, 256, %i3
label7:
xnor %l4, 919, %g1
subcc %o7, %g3, %o3
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
set 0x10c0, %g1         ! disable traps
mov %g1, %psr
nop
ta 0
```

Figure A.1: LEON-3 assembly program generated using random program generator

Verification framework developed in this thesis work can also be extended easily to other architectures such as RISC-V. Random program generator example of generating RISC-V program is given in Figure A.2.
tclsh generator.tcl -tni 10 -it all_instructions -processor risc

```
#Test program to generate instructions for RISC-V

# Data Initialization of RISC-V registers

lui x0,  129
lui x1,  63
lui x2,  65
lui x3,  733
lui x4,  279
lui x5,  112
lui x6,  576
lui x7,  112
lui x8,  632
lui x9,  577
lui x10, 206
lui x11, 153
lui x12, 962
lui x13, 888
lui x14, 923
lui x15, 758
lui x16, 718
lui x17, 819
lui x18, 479
lui x19, 103
lui x20, 298
lui x21, 635
lui x22, 168
lui x23, 371
lui x24, 746
lui x25, 731
lui x26, 933
lui x27, 485
lui x28, 182
lui x29, 406
lui x30, 341
lui x31, 921

AND x19,x7,x7
SLL x28,x28,x12
OR x20,x11,x18
SRL x7,x8,x13
XORI x30,x0,1849
SRL x22,x28,x3
ADDI x19,x25,1249
ADDI x6,x11,1044
SUB x29,x12,x10
SLL x21,x8,x17
```

Figure A.2: RISC-V assembly program generated using random program generator