Development of utilities to measure effectiveness of tests generated by MP RIS Tool

Major Project Report

Submitted in partial fulfilment of the requirements
for the degree of

Masters of Technology

in

Electronics And Communication Engineering

(VLSI Design)

By

Vivek J. Pandit
(14MECV13)

Electronics and Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad-382481

May 2016

Development of utilities to measure effectiveness of tests generated by MP RIS Tool

Major Project Report

Submitted in partial fulfilment of the requirements
for the degree of

Masters of Technology

in

Electronics And Communication Engineering
(VLSI Design)

By

Vivek J. Pandit
(14MECV13)

Under the Guidance Of

External Guide
Mr. Gunaranjan Kurucheti
(Staff Verification Engineer)
ARM, Bengaluru

Internal Guide
Dr. Usha Mehta
(Professor,EC)
ITNU, Ahmedabad

Electronics and Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad-382481

May 2016

Declaration

This is to certify that

i) The thesis comprises my original work towards the degree of Master of Technology in VLSI Design at Nirma University and has not been submitted elsewhere for a degree.

ii) Due acknowledgement has been made in the text to all other material used.

Vivek J Pandit

14MECV13

Certificate

This is to certify that the Major Project entitled **"Development of utilities to measure effectiveness of tests generated by MP RIS Tool"** submitted by Vivek J. Pandit (14MECV13), towards the partial fulfilment of the requirements for the degree of Master of Technology in VLSI Design Engineering of Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of our knowledge, havent been submitted to any other university or institution for award of any degree or diploma.

Date:	Place: Ahmedabad
Guide	**Program Coordinator**
______________________	______________________
Dr. Usha Mehta	Dr. N. M. Devashrayee
(Professor,EC)	(Professor,EC)
HOD	**Director**
______________________	______________________
Dr. P. N. Tekwani	Dr. P. N. Tekwani

Certificate

This is to certify that the Major Project entitled **”Development of utilities to measure effectiveness of tests generated by MP RIS Tool”**, towards the partial fulfilment of the requirements for the degree of Master of Technology in VLSI Design Engineering of Nirma University, Ahmedabad is the record of work carried out by him at ARM Embedded Technology Pvt. Ltd., under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of our knowledge, havent been submitted to any other university or institution for award of any degree or diploma.

PROJECT MANAGER	**PROJECT GUIDE**
Mr. Kaleshwar Vemuri	Mr.Gunaranjan Kurucheti

Acknowledgments

I take immense pleasure in thanking of my mentor Mr. Gunaranjan Kurucheti, Staff Verification Engineer and Mr. Kaleshwar Vemuri, Engineering manager at ARM Embedded Technologies Pvt. Ltd., Bangalore for having permitted me to carry out this project work.

I wish to express my deep sense of gratitude to my guide Dr.Usha Mehta, Institute of technology, Nirma University for being a source of inspiration and for timely guidance during the project.

I would like to express my gratitude and sincere thanks to Dr.P.N.Tekwani, Head of Electrical Engineering Department for allowing me to undertake this thesis work and for his guidelines during the review process.

Vivek J Pandit

14MECV13

Abstract

In the realm of verification, Random Stimulus Generation or Random Instruction Sequence (RIS) is widely recognized as an effective approach for verifying corner cases that are difficult to anticipate. While most of design bugs are detected by the deterministic formal approach, RIS is highly effective in hitting corner cases. This dissertation report contains the project work of the utilities that were developed for MP RIS tool used for verifying processor coherency. These utilities developed to make an MP RIS Tool capable of measuring the test efficiency and also provides feedback to ensure that verification intent for a given target verification area was met. These utilities have been rolled out along with the RIS tool.

The first utility was targeted towards generating corner-case test sets for floating point operation verification. Here an interface is created that the RIS tool could call to generate test-sets, with which the registers would be initialized before performing the desired FP operation. This approach uses random constrained approach, where the exponent is subject to certain bound taking into consideration the precision which determines the floating point format. The randomness ensures that the generated test-sets cover as many points in testing space.

The second utility created with the aim to help in the analysis of cache-line migration/ intervention events in Multi-Processor verification. One of the key aspects to RIS based MP verification is the ability to generate collision events from different cores to the same region, which can have a cache-line based granularity or page granularity, in a false shared manner. These collision events will ensure that cache lines migrate within the private/local caches of the different processing elements. In this work, a utility called HeatMap is developed, which parses the simulation traces and graphically depicts the collision events to regions that were accessed within a test from the different processing elements. This utility also acts as an additional QA check, to ensure that intent of verification was met.

The third utility helped in the quality assurance (QA) of the tests generated from the RIS tool, with some basic checks like memory page info, exception check and incompatible attribute check.

Abbreviations

AEM	Architecture Envelope Model
AVS	Architecture Validation Suites
DVS	Device Validation Suites
FP	Floating Point
HDL	Hardware Description Language
ISS	Instruction Set Simulator
MMU	Memory Management Unit
MP	Multi Processor or Multi Core
NaN	Not a Number
PA	Physical Address
PE	Processing Elements
QA	Quality Assurance
RIS	Random Instruction Sequence
RTL	Register Transfer Logic
VA	Virtual Address

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Analysis of the generated test is very important thing to ensure that it meet the intent of verifier for the targeted verification area. Floating Point (FP) operations are very important feature of ARM RISC Architecture. So a New MP Tool has to have the capability of verifying it and for that a set of tests (Stimuli's) are required. Deterministic simulation approach will flush out most of the design bugs but some of the corner cases will not covered by it and for that we require random stimulus generation. Already available library based approach isn't enough for all the ARMv8-A FP instructions. So we must have an effective approach for test generation than library, which will be capable of generating interesting stimuli's that will cause overflow/carry after FP Operations. This new approach should also be capable to replace the library approach. There is currently no method to determine the set of addresses that were used by the test. Also, the user of any RIS tool that is focused on MP verification needs to be able to see how many times a particular cache-line migrates or is snooped between the cores. So Heatmap utility is developed to make it easy, Heatmap is to be generated for the addresses used in a test and from a given range of address. Quality Assurance (QA) check utility is also very important for MP RIS Tool generated test analysis. It contains so many checks for it with various parameters.

1.2 Problem Statement

The objective of this project is to develop utilities that will provide an extra capability of analysis to MP RIS Tool. These utilities are important to make analysis and hence to measure effectiveness of the tests generated by the Tool. An effectiveness of the test determines how suitable it is for the verification of targeted area.

1.3 Thesis Organization

This Thesis organized in to five chapters, a brief info about them are discussed below:
Chapter 2, describes an introduction about the Random Instruction Sequence Tool. It also contains how it is an effective than the Deterministic approach to cover corner cases.
Chapter 3, describes about the work related to generation of Interesting stimulis for FP register initialization and various approaches for the test generations.
Chapter 4, describes about the Heatmap Generation utility by block diagram representation, Implementation results and further enhancement in the utility.
Chapter 5, describes about the Quality Assurance (QA) checks developed for the MP RIS Tool generated tests.
Chapter 6, describes Concluding remarks and scope for future work.

Chapter 2

Introduction to ARM s MP RIS Tool

ARM's next-generation MP RIS verification tool focusing on memory sub-system operations and cross-PE coherency transactions in Multi-Processor/Cluster systems.
It's a server-class static RIS generator that achieves high instruction generation rates (greater that 1000 IPS) and designed to allow derived test sequences to quickly achieve their desired intent, while also allowing maximum re-use of generated scenarios for faster overage closure.
It offers full support of ARMv8-A AArch64 execution state. The AArch32 A32 (ARM) ISA is partially supported, and no support is available for AArch32 T32 (Thumb) ISA.
Instruction groupings can be defined to target specific operations and micro-architectural features.
The following are additional high-level features targeted by ARM's new MP RIS Tool:

- Multi-processor memory coherency.
- Barriers.
- Cache and TLB maintenance operations.
- Message passing (Exclusive operations, Load Acquire or Store Release, Atomics).

- Load-store dependencies and hazards.
- Generate traffic to maximize use of load-store pipeline and evictions.

2.1 Random Instruction Sequence (RIS) Generation

Random instruction sequence (RIS) tools are widely used across the industry for processor verification and validation. These tools are often used to find design bugs in a relatively stable but not yet mature RTL design. RIS tools are very effective in generating test scenarios that are hard to envision. However, quite often completely random instruction sequences are of little test value for exposing corner cases in the design, especially if the bug involves a sequence of events happening in a narrow timing window. Macros can help enhance the test quality of the generated instruction sequences by providing controlled randomness around a specific sequence of instructions targeting a specific area in the processor architecture.

2.2 Random Vs Deterministic stimulus generation

Random stimulus generation is widely recognized as an effective approach for verifying corner cases that are hard to anticipate. We found that, while most of design bugs are flushed out by the deterministic approach, random instruction sequences are also highly effective in hitting obscure cases, often finding bugs that may lay undetected for years in real-life applications.
ARM has an internal tool that can generate targeted random code sequences known as RIS. With RIS, we pre-generate self-checking tests using an ISS as the reference design. This technique won't catch design errors that are present in both the ISS and the HDL model, but in practice this situation is rare and these sequences are likely to show design errors in either model when enough sequences have been simulated.
The mainstay methodology that we have used since the early days of the first ARM CPU design is deterministic simulation. This is a common and

well understood methodology that offers a number of advantages, although it's limited by the amount of effort required to generate test cases and the performance of simulation tools. At ARM, we develop test cases as self-checking assembler sequences. We then replay these code sequences on a simple simulation testbench consisting of the ARM CPU, a simple memory model, and some simple memory- mapped peripherals. Our tests fall into two categories, AVS (Architecture Validation Suites) and DVS (Device Validation Suites).
ARM's all AVS class tests check architectural functionality such as the instruction set architecture (32-bit and 16-bit Thumb), the exception model, and the debug architecture. Our DVS tests focus on the behaviour of specific cores and check corner cases arising from the particular implementation. An advantage of this type of test case is that tests are self-contained and portable from ISS (Instruction Set Simulator) environments to Verilog or VHDL test bench environments, or to FPGA prototypes and eventually to silicon. Thus, our customers and we can verify the functional equivalence of all these design views. These suites of tests are effectively the ARM architecture compliance suites.

Chapter 3

Corner-Case Stimuli Generation for FP Operations

Floating point calculations are often used in critical applications, such as numeric simulations in nuclear physics or aeronautics, which need a high level of precision. Formal methods have been used both for hardware level and high-level floating point verification, but this has proved not to be sufficient. In this work, we present a tool that will generate test-sets that can generate test stimuli targeting corner case scenarios in the floating point verification. This approach uses an random constrained approach, where the exponent/fraction is subject to an certain bound taking into consideration the precision which determines the floating point format. The randomness ensures that the generated test-sets cover as many points in testing space.

3.1 Floating Point Formats

In computing, floating point is the formulaic representation which approximates a real number so as to support a trade-off between range and precision. A number is, in general, represented approximately to a fixed number of significant digits (the significand) and scaled using an exponent; the base for the scaling is normally two, ten, or sixteen. A number that can be represented exactly is of the following form:

Mantissa x Base^{Exponent}

Here, Mantissa Z, base N, and exponent Z. ARM Floating Point architecture (VFP) provides hardware support for floating point operations (Data Type) in half-, single-, and double-precision floating point arithmetic. ARMv-8A Architecture Supports Three Floating Point Formats:

- Half Precision(HP)
- Single Precision(SP)
- Double Precision(DP)

3.1.1 Half-precision floating-point formats

ARMv8 supports two half-precision floating-point formats:

- IEEE half-precision, as described in the IEEE 754-2008 standard.

- Alternative half-precision.

The description of IEEE half-precision includes ARM-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For both half-precision floating-point formats, the layout of the 16-bit format is the same.

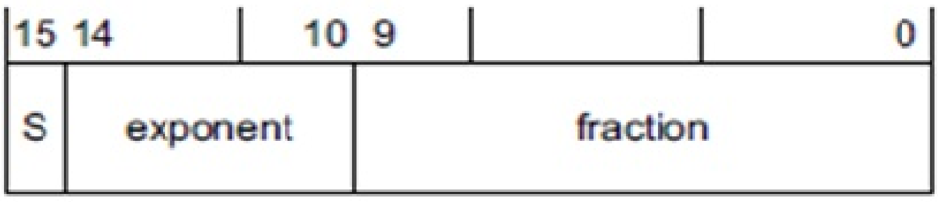


Figure 3.1: Half Precision Floating Point Format

The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

For 0 <exponent <0x1F The value is a normalized number and is equal to:

$$(-1)^S * 2^{(expoent-15)} * (1.fraction) \tag{3.1}$$

The minimum positive normalized number is 2^{-14}, or approximately 6.104 * 10^{-5}.

The maximum positive normalized number is (2 - 2^{-10}) $*$ 2^{15}, or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == 0x1F.

For exponent == 0 The value is either a zero or a denormalized number, depending on the fraction bits: if fraction == 0 The value is a zero. There are two distinct zeros:

+0 when S==0

-0 when S==1.

If fraction != 0 The value is a denormalized number and is equal to:

$$(-1)^S * 2^{-14} * (0.fraction) \tag{3.2}$$

The minimum positive denormalized number is 2^{-24}, or approximately 5.960 * 10^{-8}.

If exponent == 0x1F The value depends on which half-precision format is being used:

- **IEEE half-precision**

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits: if fraction == 0 The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

if fraction != 0 The value is a NaN, and is either a quiet NaN or a signaling NaN. The two types of NaN are distinguished by their most significant fraction bit, bit[9]:
bit[9] == 0 The NaN is a signalling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.
bit[9] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

- **Alternative half-precision**

The value is a normalized number and is equal to:

$$(-1)^S * 2^{16} * (1.fraction) \tag{3.3}$$

The maximum positive normalized number is $(2-2^{-10})*2^{16}$ or 131008.

3.1.2 Single-precision floating-point formats

The single-precision floating-point format is as defined by the IEEE 754 standard. This description includes ARM-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard.

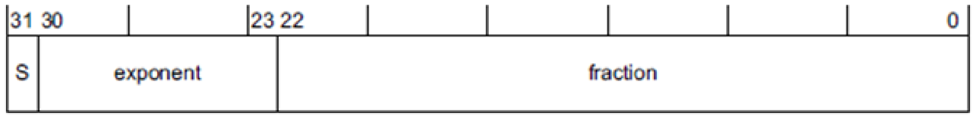


Figure 3.2: Single Precision Floating Point Format

The interpretation of the format depends on the value of the exponent field, bits[30:23]:
For 0 <exponent <0xFF , The value is a normalized number and is equal to:

$$(-1)^S * 2^{(exponent-127)} * (1.fraction) \tag{3.4}$$

The minimum positive normalized number is 2^{-126}, or approximately 1.175 * 10^{-38}. The maximum positive normalized number is (2 - 2^{-23}) $*$ 2^{127}, or approximately 3.403 * 10^{38}. If exponent == 0, The value is either a zero or a denormalized number, depending on the fraction bits:

If fraction == 0, The value is a zero. There are two distinct zeros:

+0 When S==0.

-0 When S==1.

These usually behave identically. In particular, the result is equal if +0 and -0 are compared as floating-point numbers. However, they yield different results in some circumstances.

For example, the sign of the infinity produced as the result of dividing by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words. If fraction != 0, The value is a denormalized number and is equal to:

$$(-1)^S * 2^{-126} * (0.fraction) \tag{3.5}$$

The minimum positive denormalized number is 2^{-149}, or approximately 1.401 * 10^{-45}.

Denormalized numbers are always flushed to zero in AArch32 Advanced SIMD processing. They are optionally flushed to zero in floating-point processing and AArch64 SIMD.

If exponent == 0xFF, The value is either an infinity or a Not a Number (NaN), depending on the fraction bits: If fraction == 0, The value is an infinity. There are two distinct infinities: +infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number. -infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number. If fraction != 0, The value is a NaN, and is either a quiet NaN or a signaling NaN. The two types of NaN are distinguished by their most significant fraction bit, bit[22]:

For bit[22] == 0, The NaN is a signaling NaN. The sign bit can take any value,

and the remaining fraction bits can take any value except all zeros.
For bit[22] == 1, The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

3.1.3 Double-precision floating-point formats

The double-precision floating-point format is as defined by the IEEE 754 standard. Double-precision floating-point is supported by both floating-point and SIMD instructions in AArch64 state, and only by floating-point instructions in AArch32 state. This description includes implementation-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard. A double-precision value is a 64-bit double word, with the format:

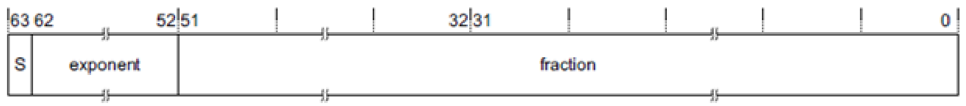


Figure 3.3: Double Precision Floating Point Format

Double-precision values represent numbers, infinities and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent: 0 <exponent <0x7FF , The value is a normalized number and is equal to:

$$(-1)^S * 2(exponent - 1023) * (1.fraction) \tag{3.6}$$

The minimum positive normalized number is 2^{-1022}, or approximately 2.225 * 10^{-308}. The maximum positive normalized number is (2 - 2^{-52}) $*$ 2^{1023}, or approximately 1.798 * 10^{308}. If exponent == 0 , The value is either a zero or a denormalized number, depending on the fraction bits:
If fraction == 0 , The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:
+0 when S==0

-0 when S==1. If fraction != 0 , The value is a denormalized number and is equal to : $(-1)^S * 2^{1022} * (0.fraction)$

The minimum positive denormalized number is 2^{1074}, or approximately 4.941 * 10^{-324}.

For exponent == 0x7FF , The value is either an infinity or a NaN, depending on the fraction bits:

If fraction == 0 , the value is an infinity. As for single-precision, there are two infinities:

+infinity When S==0.

-infinity When S==1.

If fraction != 0 , The value is a NaN, and is either a quiet NaN or a signaling NaN.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

For bit[51] == 0 , The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[51] == 1 , The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

3.2 Condition Codes for FP Corner-Case Tests

Floating Point Architecture is an important part of ARM architecture. ARMv8-A Architecture supports hundreds of Floating Point related Instructions. Each of instruction is used for any of the three floating point format discussed in Literature review. Interesting stimulis generation for FP operations involves generation of the values (tests) within the specified range of respective floating point format to be loaded in operand registers specified in FP Instruction. Now the FP operation is performed on operands with these loaded values, and the final result will be such that it will set condition flags. If any condition occurs out of sixteen possible conditions as shown in Table 1 after any FP operation then those respective operand values are called corner-case Tests for that FP operation.

Condition Code	**Meaning (floating-point)**	**Condition flags**
0000	Equal	Z == 1
0001	Not Equal or Unordered	Z == 0
0010	Greater than, Equal, or unordered	C == 1
0011	Less Than	C == 0
0100	Less Than	N == 1
0101	Greater than, equal, or unordered	N == 0
0110	Unordered	V == 1
0111	Ordered	V == 0
1000	Greater than, or unordered	C == 1 And Z == 0
1001	Less than or equal	!(C == 1 And Z == 0)
1010	Greater than or equal	N == V
1011	Less than, or unordered	N != V
1100	Greater than	Z == 0 And N == V
1101	Less than, equal, or unordered	!(Z == 0 And N == V)
1110	Always	Any
1111	Always	Any

Table 3.1: Possible Conditional Codes with Conditional Flags

3.3 Corner-Case Stimuli Generation Utility Architecture

As shown in the Figure 3.4 its clear that to generate Corner-Case Stimulis arguments required to be pass are described below:
(a) Floating Point Instruction
(b) Mode
(c) Floating Point Precision

(a) First argument is a floating point Instruction supported by ARMv8-A ISA. It can be a ARMv-8A FP instruction in as it is form or it can be followed by some predefined prefixes and suffixes.

(b) Mode can be Library (L) or Random(R).

(c) Floating Point precision can be Half Precision (HP), Single Precision (SP) or Double Precision (DP).

So after getting all above arguments, the utility tool will analyse it, check availability of tests for requested type of tests and returns a Test vector of three elements.

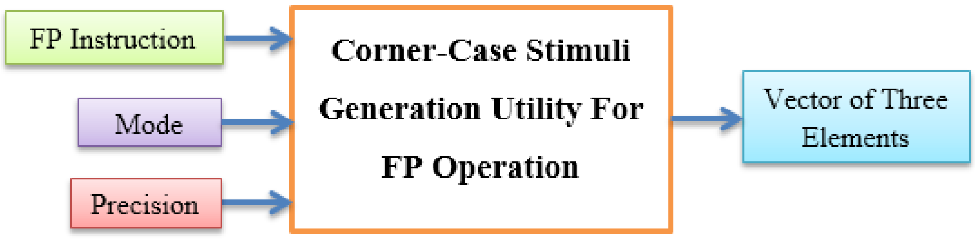


Figure 3.4: Block Diagram Of Corner-Case Stimuli Generator

A complete Corner-Case Stimuli Generation Flowchart is shown in the Figure 3.5. Its clear that to generate Corner-Case Stimulis arguments required to be pass three parameters, Floating Point Instruction, Mode and Floating Point Precision.

So after getting all above arguments, tool will first check whether the mode is Library (L) or Random (R). Then it will take care about FP Instruction and if Library (L) mode is selected then tool will look into whether it exists in to Library or not.

In above case its necessary to check availability of FP Instruction in Library because of Library has limited numbers of FP instruction support. So if an entered instruction will not find tests in to the Library then tool will automatically switch to the Random (R) mode.

Then tool will load the FP Instruction Operands Registers with the generated values according to the entered Floating Point Precision and perform an Operation. This process continues to the specified numbers of time (N time).

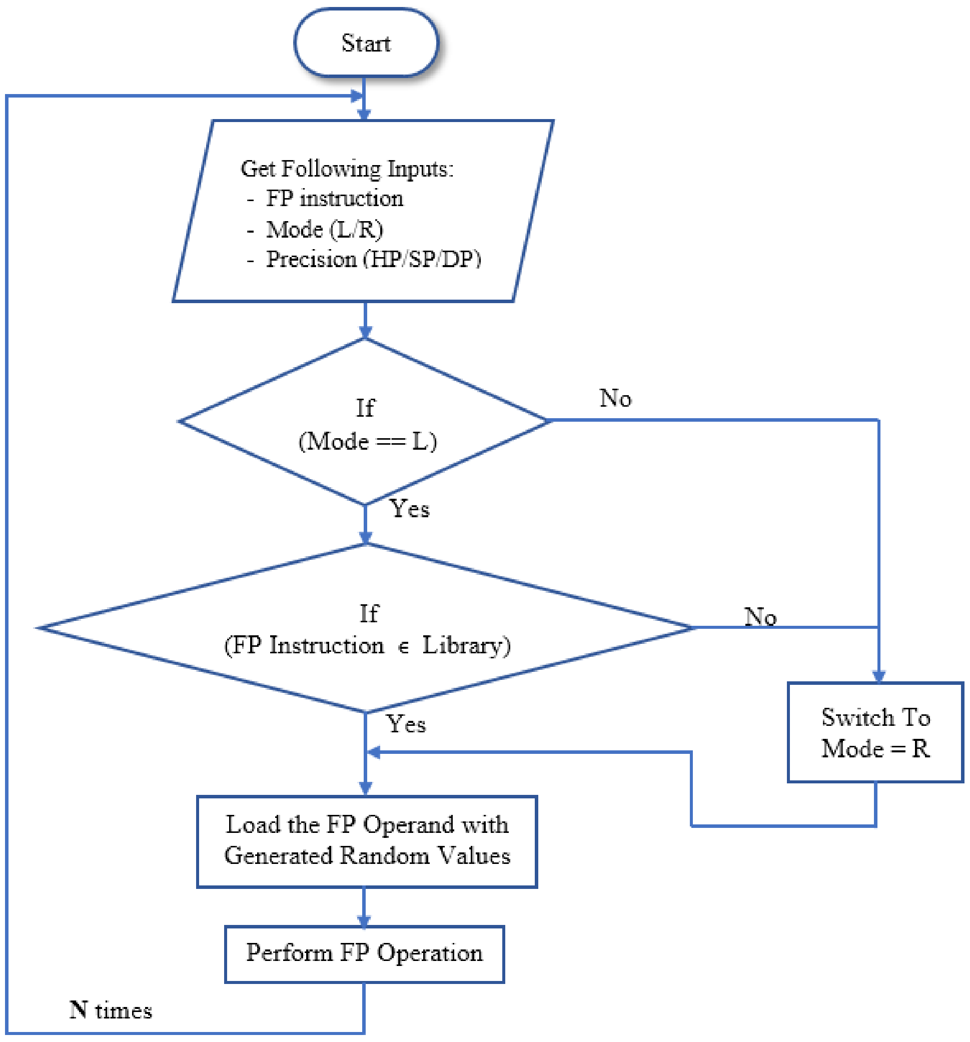


Figure 3.5: Corner-Case Stimuli Generation Flow for FP Operations

3.4 FP Register Initialization Methods

There are two possible methods discussed in this utility for FP register initialization:

- Library
- Randomizing Mantissa and Exponent

3.4.1 Library

In this approach utilizing already available library of corner-case stimulis for Floating Point register Initialization.

But the problem with this approach is that it is useful for very limited numbers of instructions. So for the many other new FP instructions we require a new approach and that can also replace the Library based approach.
Another problem with this approach is that its available for only two Single and Double Precision FP formats.

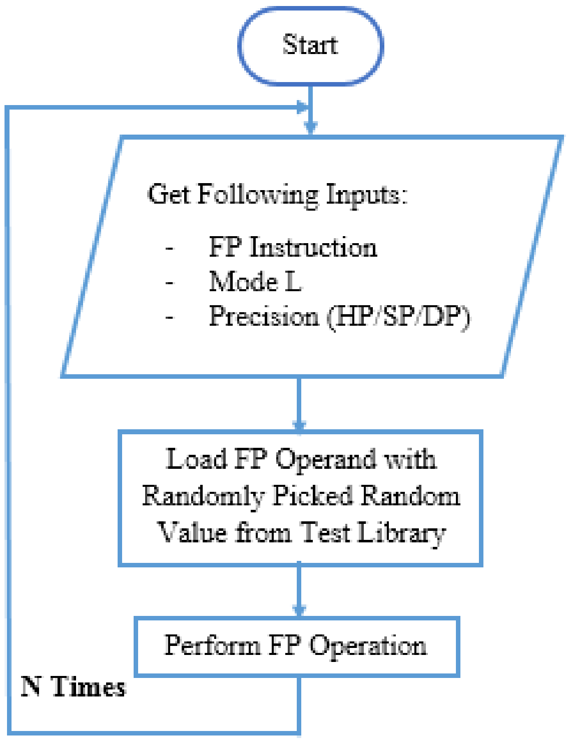


Figure 3.6: Corner-Case Stimuli Generation Flow for Library Mode

3.4.2 Randomize Mantissa and Exponent for FP Register

This is the new and effective approach than available Library. This approach will be useful for all the FP instructions supported by ARMv8-A and all three FP formats. The concept presented here to generate an Interesting stimulis is by randomizing the values of mantissa and exponent values of FP register. Random value that is generated within the specified range varies from each FP formats. As shown in Figure 3.7 only exponent and mantissa part of the FP registers are randomizing. Both are randomized separately and concatenated together in order to get a complete value. The test generation is totally independent to

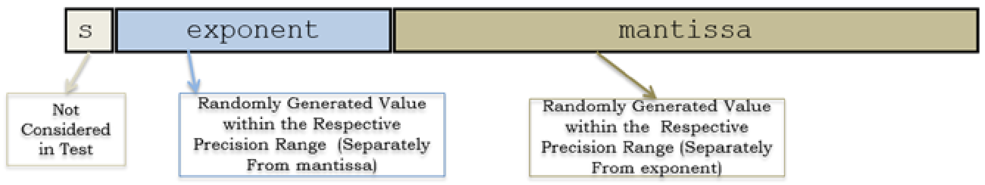

s
exponent
mantissa
Not Considered in Test
Randomly Generated Value within the Respective Precision Range (Separately From mantissa)
Randomly Generated Value within the Respective Precision Range (Separately From exponent)

Figure 3.7: Layout Of Floating Point Number

status of Sign (S) bit. Test = exponent + mantissa

- E.g. Randomize exp and frac and concatenate them as explained below : exponent = 2e mantissa = 32a4de

- Final value to be loaded in the register = 2e32a4de

So this will return a vector of elements that will be used by the MP verification tool for register Initialization. So for example when the FP instruction takes three operands then a vector of three elements (Which are three operands) is returned. Here a detailed working by the flow chart of the Random (R) mode test generator is shown. A test generated by it is according to the range of specified FP Precision and for the given FP Operation it may or may not generate Carry/Overflow.

So the cases for which the test will not generate the Carry/Overflow for them have to set constrains so that for the given FP Instruction and Precision Itll not generate the same test again.

So by following above steps for all supported FP Instructions, Precisions and their combinations and respective constraints to be set so that the Random Test Generator will generate only those tests that will cause Carry/Overflow for respective Precision and FP Operations.

3.5 Corner-Case Stimuli Generation Utility for FP Operation

To use this utility, user has to specify three parameters as discussed before. General form of Parameters or Command line arguments to be passed are shown

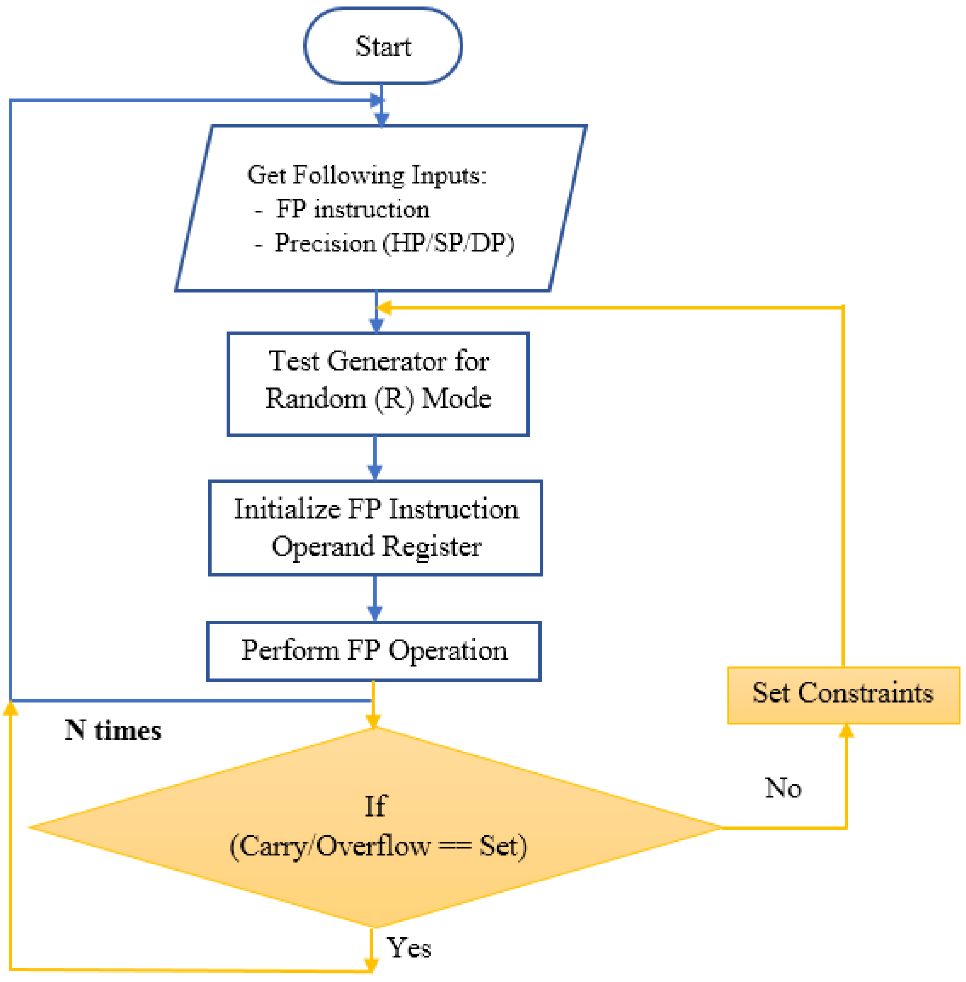


Figure 3.8: Flow Chart for the working Designing of Random(R) Mode

below. >./<Executable >-i <FP_OP_SUFFIX >-m <R/L >-p <HP/SP/DP>
Where,

-i → For FP Instruction

-m → For Mode

-p → For Precision Type

<FP_OP_SUFFIX> → Floating Point Instruction with Prefix FP and an appropriate SUFFIX.

A header file is created in which the FP instructions of enumerated types are defined for both Library and Random mode. So the Tool will generate any number randomly within the respective mode range and return respective

Instruction from enum.

<R/L> → Either Random (R) or Library(L).

R → This mode will return a Test vector by randomizing mantissa and exponent of respective precision Type. It supports all 350 FP Instructions.

L → This mode will return a Test vector from the loaded library for given FP Instruction. Which supports 35 FP Instruction.

<HP/SP/DP> → Precision Type (Either Half/Single/Double Precision FP Format)

3.6 Implementation Result

To use this utility user has to call that function, which contains three parameters and returns a vectors of three elements. Here vector elements are the corner-case tests for respective FP instruction and precision.

Result 1:

Inputs :

FP Instruction : FP_FADD

Mode : L

Precision : DP

Output :

Operand[0] ⇒ 0040000000000000

Operand[1] ⇒ bfb0000000000000

Operand[2] ⇒ 0000000000000000

Result 2:

Inputs :

FP Instruction : FP_FCMP

Mode : R

Precision : DP

Output :

```
Operand[0] ⇒ 0040000000000000
Operand[1] ⇒ bfb0000000000000
Operand[2] ⇒ 0000000000000000
```

Chapter 4

Heatmap Generation Utility

Modern multi-processor designs rely on weak consistency memory models that make it easier to implement performance boosting mechanisms such as caches, out-of-order, and speculative executions. Implementations of these consistency models are highly error-prone and hard to verify due to the vast test space and their distributed nature. One of the key aspects to RIS based MP verification is the ability to generate collision events from different cores/processing threads to the same region, which can have a cache-line based granularity or page granularity, in a false shared manner.

These collision events will ensure that cache lines migrate within the private/local caches of the different processing elements. There is currently no method to determine the set of addresses that were used by the test.

In this chapter, we present a utility called Heatmap, which parses the simulation traces and graphically depicts the collision events to regions that were accessed within a test from the different processing elements. This tool also acts as an additional QA check, to ensure that intent of verification was met. Representation of the accessed addresses in the form of Heatmap will make analysis much easy than the direct representation of the data, and this will also save much time and effort.

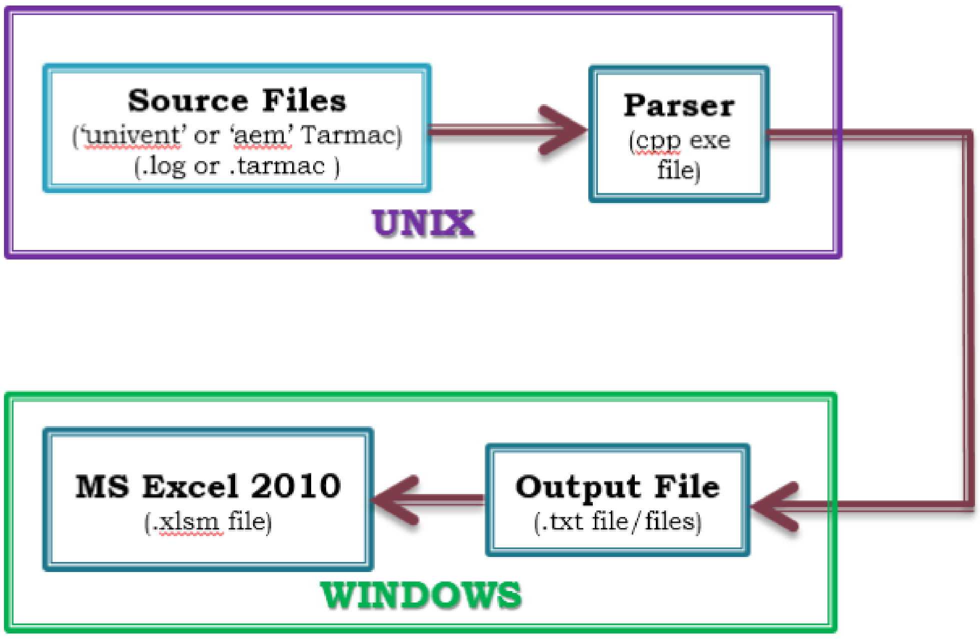


Figure 4.1: Block Diagram of Heatmap Generator Utility

4.1 Heatmap Generation Flow

Heatmap generation utility can be divided in to four parts as shown in Figure 4.1

a) Source Files

b) Parser

c) Output File

d) MS Excel Tool

(a) Source Files :

In order to get Heatmap of addresses used in the test, a source files with the trace of addresses and the respective instruction is required. For now this utility supports only one type source files called Architecture Envelope Model (aem). This source files also called Tarmac files. Here to generate Heatmap, The test addresses respective to Load/Store Instruction are only used.

(b) Parser:

It is a program written in C++ used to parse the tarmac files and extract an information of the addresses related to Load/Store only and generate an output file(.txt) file.

(c) Output File:

This is the .txt file that parser code generate. This file used as a source file for the MS Excel tool.

This file contains the required information in particular format used to generate Heatmap.

This file contains following information :

- page starting addresses
- Nos of time each page addresses are accessed
- Memory locations accessed from each page
- cpu (core) Information (In case of aem only)

(d) MS Excel Tool

This tool is used to represent the data available in the .txt file in the required format in the excel sheet and to generate Heatmap as shown in implementation section.

Excel macros are used to arrange the data and generate the Heatmap.

4.2 Memory Attributes for AEM Tarmac

In the above Heatmap utility part one switch is provided for memory attributes. When it is on then we have info of each memory page with their attributes that re accessed in the Test. Each Page has both Inner and Outer Attributes. Inner or Outer Attributes Can Be Any Of Following:

- Write-Back (WB)
- Write-Through (WT)
- Non-Cacheble (NC)
- Device-Order (DO)

4.2.1 Write-Back(WB)

Write back is a storage method in which data is written into the cache every time a change occurs, but is written into the corresponding location in main memory only at specified intervals or under certain conditions.
When a data location is updated in write back mode, the data available in cache is called fresh data, and the corresponding data in main memory, which no longer matches the data in cache, is called stale. If a request for stale data in main memory arrives from another application program, the cache controller updates the data in main memory before the application accesses it.
Write back optimizes the system speed because it takes less time to write data into cache alone, as compared with writing the same data into both cache and main memory. However, this speed comes with the risk of data loss in case of a crash or other adverse event.
Write back is the preferred method of data storage in applications where occasional data loss events can be tolerated. In more critical applications such as banking and medical device control, an alternative method called write through practically eliminates the risk of data loss because every update gets written into both the main memory and the cache. In write through mode, the main memory data always stays fresh.

4.2.2 Write-Through(WT)

Write through is a storage method in which data is written into the cache and the corresponding main memory location at the same time. The cached data allows for fast retrieval on demand, while the same data in main memory ensures that nothing will get lost if a crash, power failure, or other system disruption occurs.
Although write through minimizes the risk of data loss, every write operation must be done twice, and this redundancy takes time. The active application program must wait until each block of data has been written into both the main memory and the cache before starting the next operation. The "data insurance"

therefore comes at the expense of system speed.
Write through is the preferred method of data storage in applications where data loss cannot be tolerated, such as banking and medical device control. In less critical applications, and especially when data volume is large, an alternative method called write back accelerates system performance because updates are normally written exclusively to the cache, and are backed up in the main memory only at specified intervals or under certain conditions.

4.2.3 Non-Cacheable (NC)

Normal Non-Cacheable memory is the part of main memory that are not looked-up in any cache. Some data that are not required to be used frequently or rarely used data has to store in non-cacheable memory region.

4.2.4 Device-Order (DO)

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for ARMv8 Device memory are: Gathering Identified as G or nG, Reordering Identified as R or nR, Early Write Acknowledgement hint Identified as E or nE, The ARMv8 Device memory types are: Device-nGnRnE Device non-Gathering, non-Reordering, No Early write acknowledgement. Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.
Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement. Equivalent to the Device memory type in earlier versions of the architecture.
Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement. ARMv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.
Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

ARMv8 adds this memory type to the translation table formats found in earlier versions of the architecture.
Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that speculative accesses to Device-GRE memory is forbidden. Collectively these are referred to as any Device memory type. Going down the list, the memory types are described as getting weaker; conversely the going up the list the memory types are described as getting stronger.

4.3 Using Heatmap Utility

4.3.1 UNIX Environment

To use Heatmap utility, user need to enter three required arguments along with the Heatmap executable. General form of required command line argument is explained below:
$>./<Executable.exe >-proj <Project_ Name >-type <Type of File >-arch32/64 -ca-ps <Page_ Size >-dir <Full Tarmac Directory >

- Here, All 0 - 9 arguments are compulsory for HeatMap generation and for Memory attributes only 7th Option (Page_ Size) is not required.

<Executable.exe >$\longrightarrow$ It's an executable(.exe) file to run.
-proj $\longrightarrow$ It's an option indicates that next argument is Project Name.
<Project _ Name >$\longrightarrow$ A project Name, It can be any assigned project name
-type $\longrightarrow$ It's an option indicates that next argument is a type of Tarmac file to parse
<Type of File >$\longrightarrow$ A type of file to parse. For now Its "aem" only.
-ca $\longrightarrow$ Its an option indicates that analysis is for Memory Attributes not for HeatMap.
-ps $\longrightarrow$ It's an option indicates that next argument is Page Size (Its compulsory argument for HeatMap generation)
<Page_ Size >$\longrightarrow$ Page Size can be either "4k" / "4K" or "64k"/"64K"

-arch32/64 $\longrightarrow$ An option for Arch 32 or Arch 64 AEM Tarmac
-dir $\longrightarrow$ An Option Indicates that next argument is Tarmac File directory
<Tarmac_File >$\longrightarrow$ An Absolute Directory of AEM Tarmac File.

4.3.2 Windows Environment

In windows environment an excel file with required macros for reading, arranging and generating HeatMap of the data of generated .txt file is available.
It also generates graph of PAGE ADDRESSES vs PAGE COUNT for all the available CPUs in the test for which the HeatMap was generated. In case of memory attributes option it will generate MEMORY PAGE ATTRIBUTEs Vs MEMORY PAGE ATTRIBUTE COUNT graph.
The excel file is saved with an extension of .xlsm. This will automatically enables all the macros when this excel file is open.

4.4 Enhancing Heatmap Utility

Along with the Heatmap representation extra features are also added for more ease in the analysis to user. Heatmap Utility can be enhanced by including some of the features mentioned below:

- On mouse hover on the Page address, will display all the addresses accessed in that page

- Filter out result based on the Cache attributes

  An Implementation result for the same is shown in the next sub section.

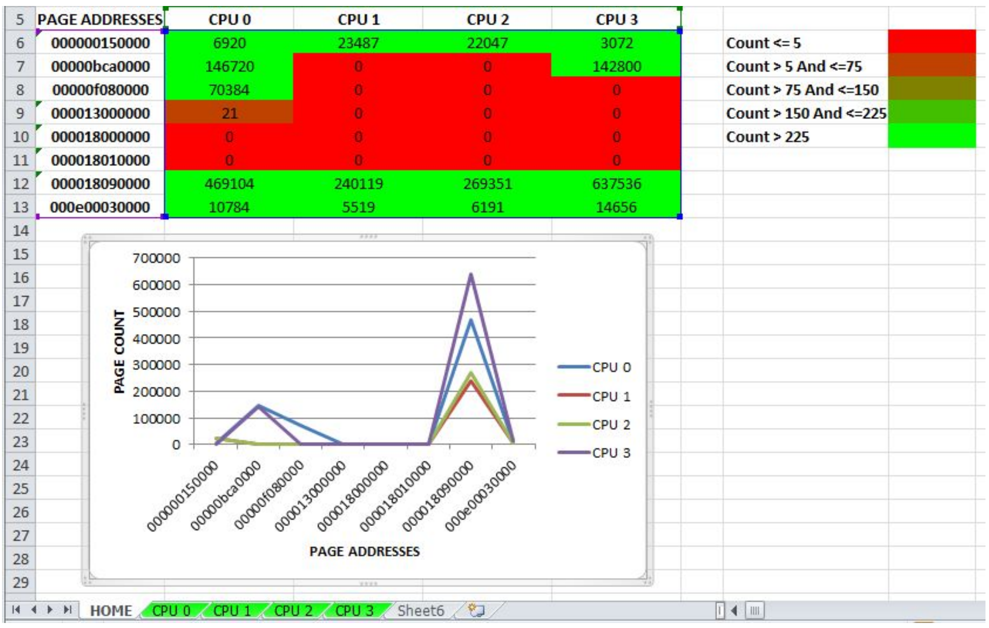
4.5 Implementation results

4.5.1 Heatmap & Graph for the aem tarmac file

Here numbers represented inside the colour matrix shows that many numbers of time respective page address is accessed by respective CPU. And a macro is also written for the graph generation (PAGE COUNT Vs PAGE ADDRESS)

- **For Arch64**

HeatMap for 64-bit test tarmac is shown in Figure 4.2 with respective graph.



	PAGE ADDRESSES	CPU 0	CPU 1	CPU 2	CPU 3
6	000000150000	6920	23487	22047	3072
7	00000bca0000	146720	0	0	142800
8	00000f080000	70384	0	0	0
9	000013000000	21	0	0	0
10	000018000000	0	0	0	0
11	000018010000	0	0	0	0
12	000018090000	469104	240119	269351	637536
13	000e00030000	10784	5519	6191	14656

Count <= 5	🟥
Count > 5 And <=75	🟧
Count > 75 And <=150	🟫
Count > 150 And <=225	🟩
Count > 225	🟩

Figure 4.2: Heatmap of Arch64 aem tarmac file with graph

- **For Arch32**

HeatMap for 32-bit test tarmac is shown in Figure 4.3 with respective graph.

- **With 4K Page Size**

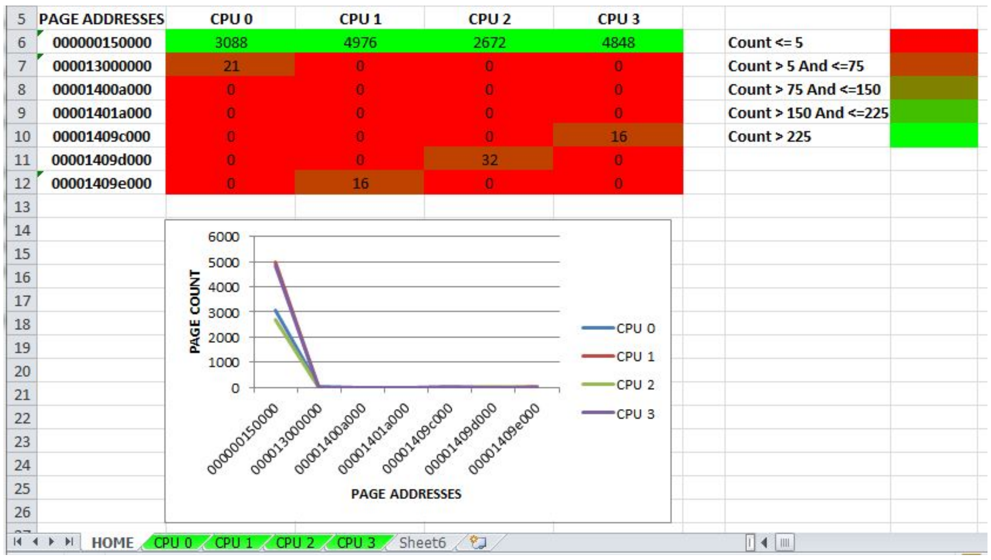
HeatMap for aem tarmac is shown in Figure 4.4 with respective graph for 4K Page size.

- **With 64K Page Size**

HeatMap for aem tarmac is shown in Figure 4.5 with respective graph for 64K Page size.

- **Heatmap with Hover Information Display**

Hover message contains all the memory addresses accessed and numbers of times it accessed from the respective page addresses by the respective CPU. Result for the same is shown in Figure 4.6.

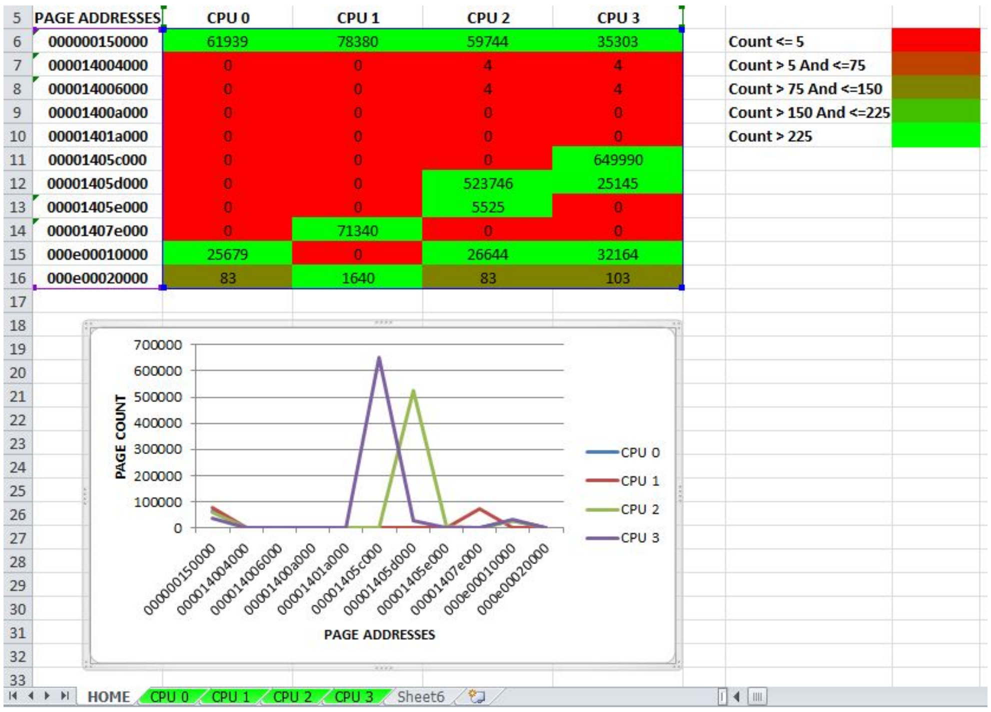


	PAGE ADDRESSES	CPU 0	CPU 1	CPU 2	CPU 3		
6	000000150000	3088	4976	2672	4848	Count <= 5	🟥
7	000013000000	21	0	0	0	Count > 5 And <=75	🟫
8	00001400a000	0	0	0	0	Count > 75 And <=150	🟫
9	00001401a000	0	0	0	0	Count > 150 And <=225	🟩
10	00001409c000	0	0	0	16	Count > 225	🟩
11	00001409d000	0	0	32	0		
12	00001409e000	0	16	0	0		

Figure 4.3: Heatmap of Arch32 aem tarmac file with graph

4.5.2 Memory Attributes for the aem tarmac file

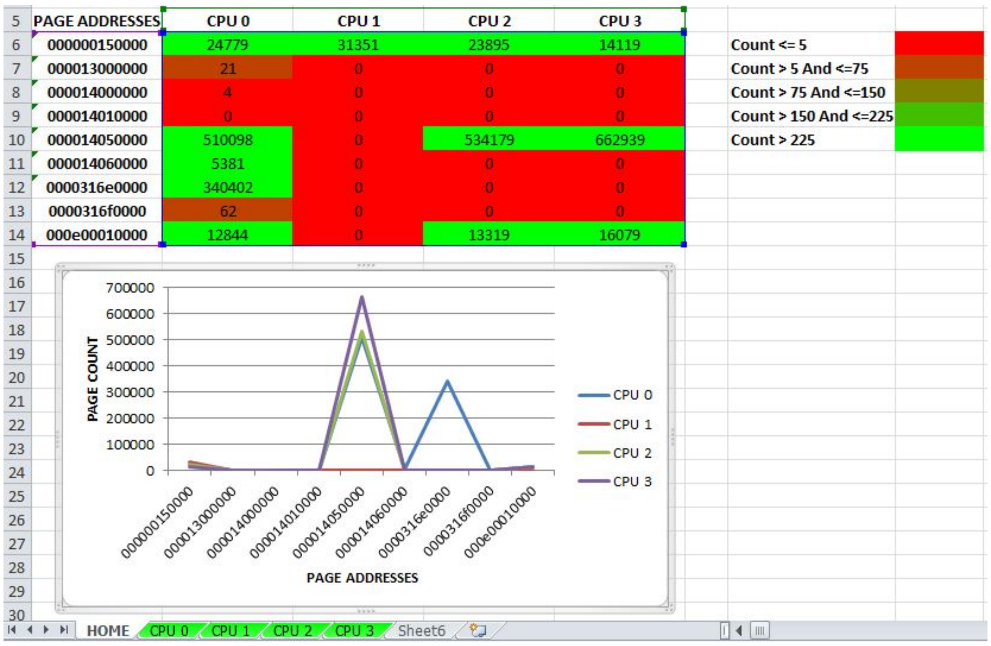
Figure 4.7 shows Memory Attributes of the Pages accessed in the Test by each CPUs. Inner and Outer attributes of each accessed pages for respective CPU are available when mouse pointer hover over the respective cell. And respective MEMORY PAGE ATTRIBUTEs Vs MEMORY PAGE ATTRIBUTE COUNT graph is also shown.



PAGE ADDRESSES	CPU 0	CPU 1	CPU 2	CPU 3
000000150000	61939	78380	59744	35303
000014004000	0	0	4	4
000014006000	0	0	4	4
00001400a000	0	0	0	0
00001401a000	0	0	0	0
00001405c000	0	0	0	649990
00001405d000	0	0	523746	25145
00001405e000	0	0	5525	0
00001407e000	0	71340	0	0
000e00010000	25679	0	26644	32164
000e00020000	83	1640	83	103

Count <= 5	🟥
Count > 5 And <=75	🟧
Count > 75 And <=150	🟫
Count > 150 And <=225	🟩
Count > 225	🟩

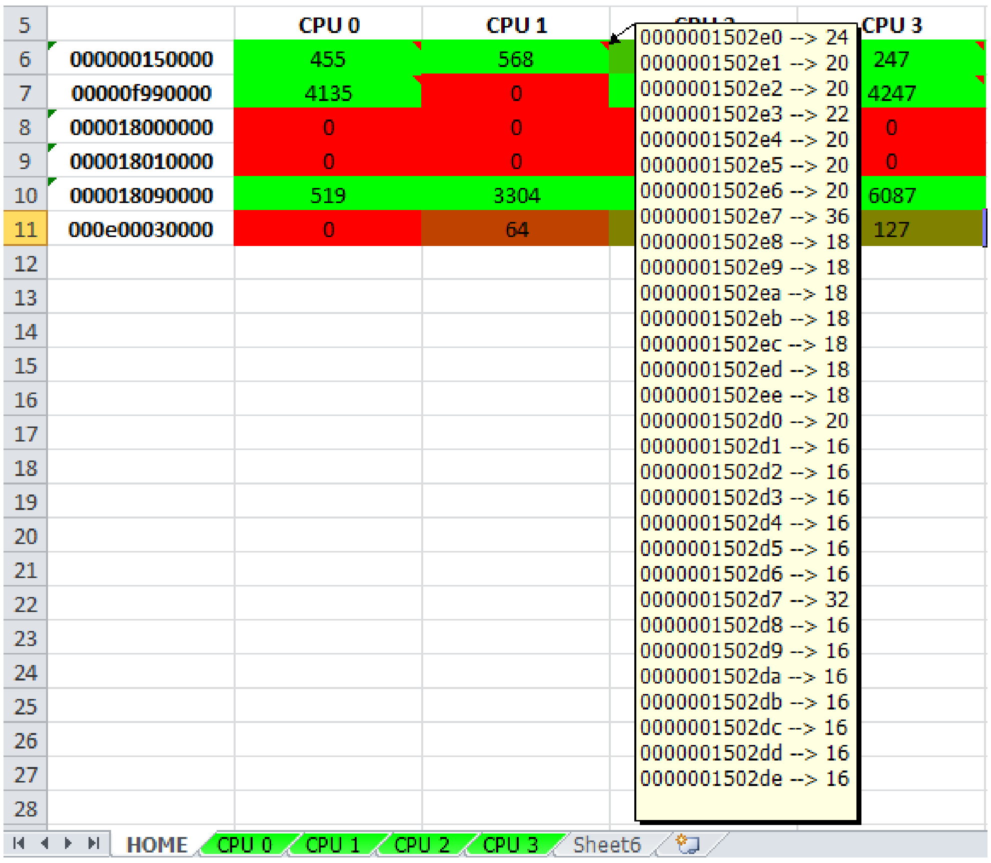
Figure 4.4: Heatmap of 4K Page Size aem tarmac file with graph



PAGE ADDRESSES	CPU 0	CPU 1	CPU 2	CPU 3
000000150000	24779	31351	23895	14119
000013000000	21	0	0	0
000014000000	4	0	0	0
000014010000	0	0	0	0
000014050000	510098	0	534179	662939
000014060000	5381	0	0	0
0000316e0000	340402	0	0	0
0000316f0000	62	0	0	0
000e00010000	12844	0	13319	16079

Count <= 5	🟥
Count > 5 And <=75	🟧
Count > 75 And <=150	🟫
Count > 150 And <=225	🟩
Count > 225	🟩

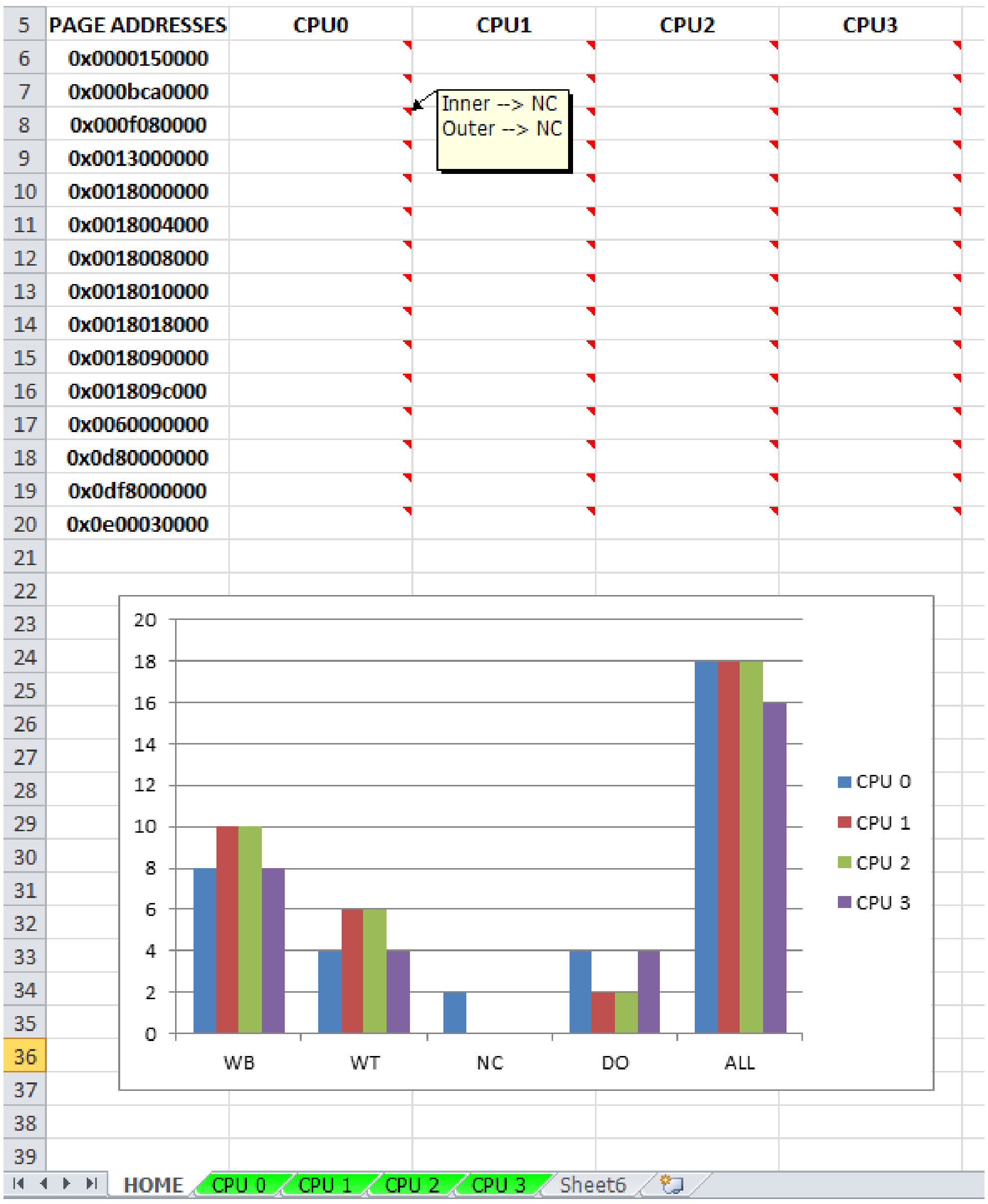
Figure 4.5: Heatmap of 64K Page Size aem tarmac file with graph



		CPU 0	CPU 1	CPU 2	CPU 3
5		CPU 0	CPU 1		CPU 3
6	000000150000	455	568		247
7	00000f990000	4135	0		4247
8	000018000000	0	0		0
9	000018010000	0	0		0
10	000018090000	519	3304		6087
11	000e00030000	0	64		127

```
0000001502e0 --> 24
0000001502e1 --> 20
0000001502e2 --> 20
0000001502e3 --> 22
0000001502e4 --> 20
0000001502e5 --> 20
0000001502e6 --> 20
0000001502e7 --> 36
0000001502e8 --> 18
0000001502e9 --> 18
0000001502ea --> 18
0000001502eb --> 18
0000001502ec --> 18
0000001502ed --> 18
0000001502ee --> 18
0000001502d0 --> 20
0000001502d1 --> 16
0000001502d2 --> 16
0000001502d3 --> 16
0000001502d4 --> 16
0000001502d5 --> 16
0000001502d6 --> 16
0000001502d7 --> 32
0000001502d8 --> 16
0000001502d9 --> 16
0000001502da --> 16
0000001502db --> 16
0000001502dc --> 16
0000001502dd --> 16
0000001502de --> 16
```

Figure 4.6: Heatmap with hover information display



5	PAGE ADDRESSES	CPU0	CPU1	CPU2	CPU3
6	0x0000150000				
7	0x000bca0000				
8	0x000f080000				
9	0x0013000000				
10	0x0018000000				
11	0x0018004000				
12	0x0018008000				
13	0x0018010000				
14	0x0018018000				
15	0x0018090000				
16	0x001809c000				
17	0x0060000000				
18	0x0d80000000				
19	0x0df8000000				
20	0x0e00030000				

Inner --> NC
Outer --> NC

Figure 4.7: Memory Attribute Result for aem tarmac file with graph

Chapter 5

Quality Assurance (QA) Utility

In general for developing products, quality assurance is any systematic process of checking to see whether a product that being developed is meeting specified requirements or not. A quality assurance system is said to increase customer confidence and a company's credibility, to improve work processes and efficiency, and to enable a company to better compete with others.

5.1 QA Checks for Test generated by MP RIS Tool

In this Utility certain checks are developed after running of which a user ensures that the generated test is passing for that check or not and also shows the check results (if any).

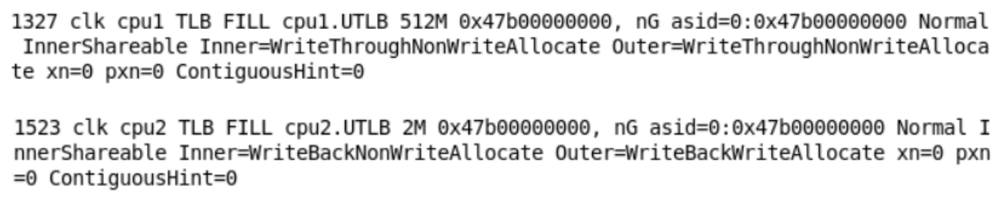
5.1.1 Incompatible Attribute Check (ia_check)

Check for incompatible attributes and display attribute with cpus if incompatibility presents in a Test.
This check is designed such that it will look for incompatibility presents in the test which is given as input to check and collects the cpu and incompatible attribute info and finally displays it.
If an AEM tarmac of respective test having following Trace then ia_ check result

will be as shown below:

Tarmac :

Respective tarmac trace info is shown in Figure 5.1 for ia_check



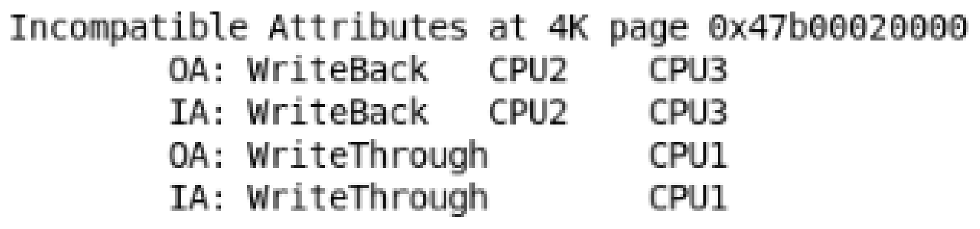
```
1327 clk cpu1 TLB FILL cpu1.UTLB 512M 0x47b00000000, nG asid=0:0x47b00000000 Normal
 InnerShareable Inner=WriteThroughNonWriteAllocate Outer=WriteThroughNonWriteAlloca
te xn=0 pxn=0 ContiguousHint=0

1523 clk cpu2 TLB FILL cpu2.UTLB 2M 0x47b00000000, nG asid=0:0x47b00000000 Normal I
nnerShareable Inner=WriteBackNonWriteAllocate Outer=WriteBackWriteAllocate xn=0 pxn
=0 ContiguousHint=0
```

Figure 5.1: Tarmac trace snapshot shows incompatible attributes

ia_check Result :

Result for the ia_check is shown in Figure 5.2



```
Incompatible Attributes at 4K page 0x47b00020000
        OA: WriteBack   CPU2    CPU3
        IA: WriteBack   CPU2    CPU3
        OA: WriteThrough        CPU1
        IA: WriteThrough        CPU1
```

Figure 5.2: ia_check result snapshot

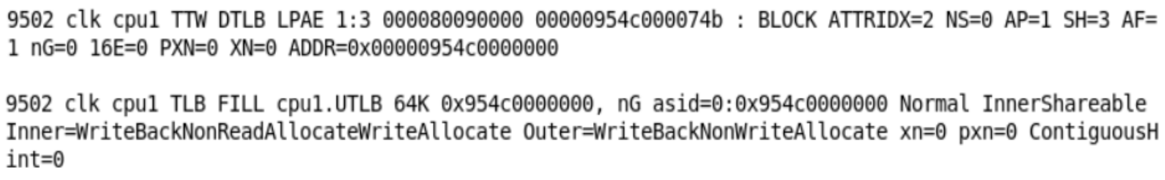
5.1.2 Memory Page information (page_info)

Display page information for each page in AEM tarmac (cpu, security, shareability, memory type, etc.). As shown below the contents under the Tarmac title are present in the Test for which page_ info check is run.

Tarmac :

Respective tarmac trace info is shown in Figure 5.3 for page_info **page_info Check Result :**

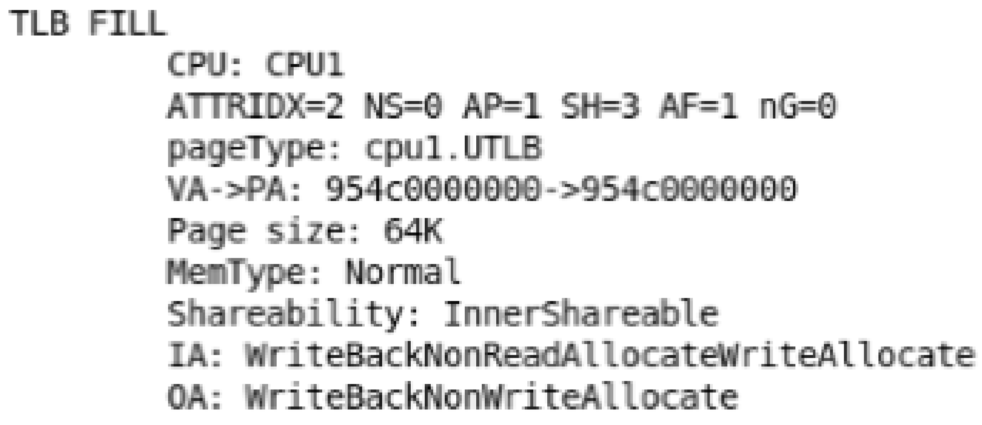
Result for the page_info is shown in Figure 5.4



```
9502 clk cpu1 TTW DTLB LPAE 1:3 000080090000 00000954c000074b : BLOCK ATTRIDX=2 NS=0 AP=1 SH=3 AF=
1 nG=0 16E=0 PXN=0 XN=0 ADDR=0x00000954c0000000

9502 clk cpu1 TLB FILL cpu1.UTLB 64K 0x954c0000000, nG asid=0:0x954c0000000 Normal InnerShareable
Inner=WriteBackNonReadAllocateWriteAllocate Outer=WriteBackNonWriteAllocate xn=0 pxn=0 ContiguousH
int=0
```

Figure 5.3: Tarmac trace memory attributes



```
TLB FILL
        CPU: CPU1
        ATTRIDX=2 NS=0 AP=1 SH=3 AF=1 nG=0
        pageType: cpu1.UTLB
        VA->PA: 954c0000000->954c0000000
        Page size: 64K
        MemType: Normal
        Shareability: InnerShareable
        IA: WriteBackNonReadAllocateWriteAllocate
        OA: WriteBackNonWriteAllocate
```

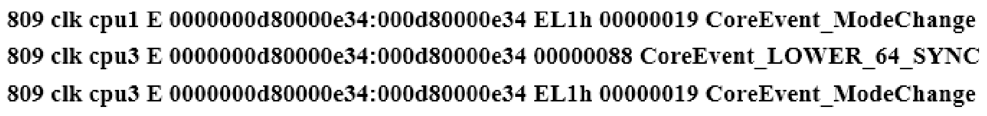
Figure 5.4: page_info check result snapshot

5.1.3 Exception Check (exc_check)

Display exception report with cpu, exception type, and count information. As shown below the contents under the Tarmac title are present in the Test for which exc_ check check is run.

Tarmac :

Respective tarmac trace info is shown in Figure 5.5 for exc_check

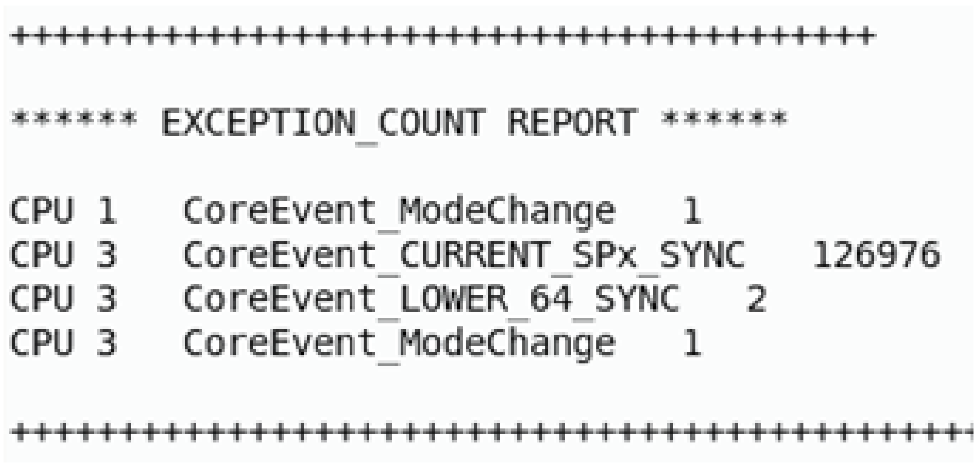


```
809 clk cpu1 E 0000000d80000e34:000d80000e34 EL1h 00000019 CoreEvent_ModeChange
809 clk cpu3 E 0000000d80000e34:000d80000e34 00000088 CoreEvent_LOWER_64_SYNC
809 clk cpu3 E 0000000d80000e34:000d80000e34 EL1h 00000019 CoreEvent_ModeChange
```

Figure 5.5: Tarmac trace for exceptions

exc_check Result :

Result for the exc_check is shown in Figure 5.6



```
++++++++++++++++++++++++++++++++++++++++

****** EXCEPTION_COUNT REPORT ******

CPU 1   CoreEvent_ModeChange   1
CPU 3   CoreEvent_CURRENT_SPx_SYNC   126976
CPU 3   CoreEvent_LOWER_64_SYNC   2
CPU 3   CoreEvent_ModeChange   1

++++++++++++++++++++++++++++++++++++++++++
```

Figure 5.6: exc_check result snapshot

5.2 QA Utility Optimization

5.2.1 Changes did to Improve Execution Speed

- Structure Optimization

  - Sorted Most frequent accessed Variable to Least frequent accessed variables.

  - This will save the time that was wasting due to accessing innermost memories for required variables.

- Pass by value replaced with Pass By reference in function calls.

  - If big objects are passed by value then it takes more time but if it is passed by reference than only pointer to that object is passed and its size is fixed irrespective of object size.

- Some Vectors Replaced with the Map.

  - Map STL's has a facility of obtaining the value by key. So any desired value can be directly accessed using respective key and it saves time. But In vector to access any intermediate element pointer has to iterate over that from start or end to reach desired location.

- Replaced many small and often called functions as Inline.

  - Inline defined functions are expanded in line when it is called. So this will save the time that required to do procedure before calling any function like storing the return address in the stack, jumping to the subroutine, storing the status of registers etc.

- Replaced Post increment assignment with Pre-increment.

  - Post increment assignment will make a copy of value, increment it and then assigns it to variable. But preincrement assignment increment the value without making its copy. So It will save some time.

- Removed And/or Replaced all unused variable(s)

5.2.2 Results

Sr. No.	**Options**	**Execution Time After Optimization**	**Original Execution Time**	**% Improvement**
1	index	29.23 s	438.42 s	1400 %
2	mem_share	1167.25 s	1568.53 s	34.37 %
3	share	24.38 s	61.20 s	151 %
4	pgs_shared	14.17 s	51.35 s	262 %
5	uninit	117.35 s	62.22 s	258 %
6	smc	24.48 s	60.05 s	145.30 %

Table 5.1: Execution Speed Improvement Summary

Chapter 6

Conclusion & Future Scope

6.1 Conclusion

Random Stimuli generator is widely recognized as an effective approach for verifying corner-cases that are hard to anticipate. That is why for todays complex multi- processor design, MP RIS Tool are widely used for its verification, because RIS Tools help in generation of corner-case scenarios for verification of any design. These scenarios might otherwise be difficult to manually hand-code or create using a deterministic approach. Utilities for MP RIS Tool will give an extra capability to analyze the generated test. The RIS generator accepts configuration files, detailing the layout of the target environment, which needs to be verified. So in order to measure an effectiveness of the generated tests for that particular targeted region these analysis utilities are very useful. These utilities also help in weeding out the tests are do not meet the verification intent and the MP RIS generator can be tuned accordingly to ensure the verification is met.

6.2 Future Scope

Currently in the FP corner case test generation utility gives any test randomly from the respective range of floating point format for any FP instruction. So there are many such tests that do not cause corner cases. Therefore constraints must be added so that it will generate only those tests for the respective FP instructions that will cause carry/Overflow after respective FP operation. Enhancing the capabilities of Quality Assurance checks by adding some more checks that will refine the test more than currently we have.

References

[1] Jhon L. Hennessy, David A. Patterson, *"Data-Level Parallelism in Vector, SIMD GPU Architecture"*. in Computer Architecture: A Quantitative Approach, 5th ed., MA: Morgan Kaufmann, 2012.

[2] ARM Architecture Reference Manual (Beta) for ARMv8-A. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset .architecture.reference/index.html

[3] John Harrison, *"Floating-Point Verification"*. International Symposium of Formal Methods Europe (Industry Day), Springer LNCS 3582, pp. 529-532, 2005.

[4] John Harrison, *"Floating-Point Verification using Theorem Proving"*. Proceedings of SFM 2006, the 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems. Springer LNCS 2965, pp. 211-242, 2006.

[5] http://www.cplusplus.com/reference/