Validation of Core RAS features for Intel XEON Server Processor using Random Instruction Generation Tool

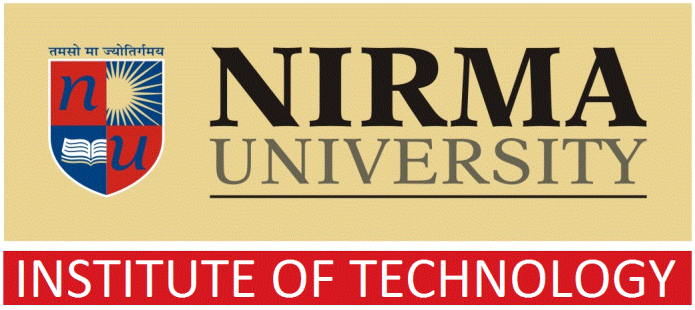
Major Project Report

Submitted in fulfillment of the requirements
for the degree of

Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)

by

Shival Singh
(13MECE16)



Electronics & Communication Engineering branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481

May, 2015

Validation of Core RAS features for Intel XEON Server Processor using Random Instruction Generation Tool

Major Project Report

Submitted in fulfillment of the requirements
for the degree of

Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)
by

Shival Singh
(13MECE16)

Under the guidance of

External Project Guide:

Mr. Raghava M Rao
Design Automation Engineer, SSV
Intel Technologies India Pvt. Ltd.,
Bangalore.

Internal Project Guide:

Mr. Vijay Savani
Assistant Professor, EC Department,
EE Branch,Institute of Technology,
Nirma University, Ahmedabad.

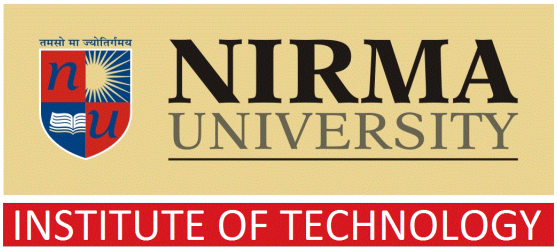
Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2015

Declaration

This is to certify that, the thesis comprises my original work towards the degree of **Master of Technology** in **Embedded Systems** at **Nirma University** and has not been submitted elsewhere for a degree.

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

SHIVAL SINGH



Certificate

This is to certify that the Major Project entitled **"Validation of Core RAS features for Intel XEON Server Processor using Random Instruction Generation Tool"** submitted by Shival Singh (13MECE16), towards the fulfillment of the requirements for the degree of "Master of Technology" in "Embedded Systems" at "Nirma University, Ahmedabad" is the record of work carried out by him 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, have not been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Mr. Vijay Savani
Project Guide

Dr. N.P. Gajjar
Program Coordinator

Dr. D.K.Kothari
Section Head, EC

Dr. P.N.Tekwani
Head of EE Dept.

Dr. K. Kotecha
Director, IT

Certificate

This is to certify that the Major Project "Validation of Core RAS features for Intel XEON Server Processor using Random Instruction Generation Tool" submitted by Shival Singh (13MECE16),towards the fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Date:

Place:

Mr. Raghava M Rao
Design Automation Engineer, SSV

Mr. Ananda P Chandranna
Engineering Manager, SSV

Acknowledgements

With immense pleasure, I would like to take this opportunity to thank all those who helped me for the successful completion of the first phase of the dissertation and for providing valuable guidance throughout the project work.

I would first of all like to acknowledge Dr. N. P. Gajjar, whose keen interest and farsightedness helped everyone in Embedded Systems carry out their project meticulously. I would offer thanks to Mr. Vijay Savani for reviewing this report and giving valuable suggestions to make it presentable, his keen interest in this dissertation work right from the beginning has been a great motivating factor in outlining the flow of my work.

Special thanks to Mr. Ananda P Chandranna, for motivating me throughout the internship, Mr. Raghava M V Rao, for giving me a chance to work on this project, providing valuable inputs, reading material and mentoring me throughout this project. I would also like to thank Sandeep, Poornima and Supreeth for providing valuable reading material, clearing doubts and guiding me through the course of completion of this project.

Last but not the least would like to thank my father, mother and sister for constantly supporting me throughout the course of post graduation, all of this would not have been possible without them.

The duration of my internship has been a fruitful experience and a great learning curve.

SHIVAL SINGH
13MECE16

Abstract

Geography is history, human technology is shrinking geographical boundaries, now people can collaborate, access resources present in a remote corner of the world from their comforting bedrooms, offices and on the move, due to the revolution called "Internet of things", cloud computing is gaining popularity day by day. Cloud computing needs lot of storage space, which requires maintaining servers, for storing and manipulating critical data (for example online retail stores, stock markets, investment banking firms, social networking sites, search engines), hence it is imperative that the servers are up and running all the time, and keep the data reliable, and are easily serviceable even after a catastrophe.

Intel uses a very through validation methodology, making use of architectural validation technique to spot bugs missed during pre silicon validation, so that the validation cycle can be reduced to achieve product readiness amidst the growing competition. The advanced automation infrastructure maintained in Server System Validation in Intel means that the target under test is continuously bombarded with random test cases to get a 24/7 coverage.

The approach in designing validation plan, targeting the areas which were left untouched/ loose ends is explained. Intel architecture uses MCA, to detect, log and recover from hardware errors, which may lead to compromising server integrity and even bringing down the system.

RMCA, an advanced version of MCA is discussed at length in this report, the advantages of this approach is that without increasing the cost of hardware, an uncorrectable error can be handled (isolated or resolved) by using the operating system or system software.

Pre requisites of loader include for running test cases targeting RMCA are discussed, which includes, setting up the platform, where the Silicon to be tested is mounted, checking the health of Silicon, enabling memory poisoning, and giving the control to the test case, which is self-driven and doesnt need any OS to operate.

Pre requisites of loader include setting up the Machine Check registers to disable error overflow, enabling poisoning of memory, generating complex instructions which will access the poisoned memory location, synchronizing the machine check exceptions across all cores, setting up the handlers for servicing the uncorrectable errors.

A case study is discussed in the end to describe the step by step approach followed to debug a failure, from loading the seed to arriving at the cause/ workaround.

Abbreviations

SSV	Server System Validation
SVE	System validation and Enabling
DIMM	Dual Inline Memory Module
RIG	Random Instruction Generator
RAS	Reliability, Availability and Serviceability
QPI	Quick Path Interconnect
SPI	System peripheral Interconnect
CMCI	Corrected machine-check error interrupt
SMI	System Managed Interrupt, Scalable Memory Interconnect
MCIP	Machine check in progress
ECC	Error Correction Code
SECDED	Single Error Correction Double Error Detection
UCR	Uncorrected Recoverable Error
MCERR	Machine Check Error
RMCA	Recoverable Machine Check Architecture
SRAO	Software Recoverable action optional
UCNA	Uncorrected no action required
MSR	Model Specific Registers
MC	Machine Check Exception
SRAR	Software Recoverable Action Required
MP	Multi Processor
ITP	In Target Probes
TAP	Test Access Ports
DFT	Design For Test
DCE	Detected but Correctable Error
DUE	Detected but Uncorrectable Error
TLB	Translational Look aside Buffer

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The nature of doing business is changing very rapidly across the world. Leading companies in computation intensive industries, such as financial trading, retail, and social networking sites are moving towards a real-time business model in which information sharing and data transfer takes place at a pretty high rate. This sharply decreasing product readiness timeline is resulting in increased demand on the performance, capacity, availability of the underlying infrastructure supporting it. In the wake of increasing competition and requirements, the penalty for infrastructure going down will result in serious financial loss and will hamper the brand value. Hence, business-critical operations must be up and running at all times.

For years, Intel has been able to meet the growing challenge of providing high performing, reliable and always running infrastructure. Advanced reliability, availability, and serviceability (RAS) and security enhancement for safeguarding the privacy of data are deep rooted across all of the Intel server products.

These attributes are made possible by exhaustive methodology for testing and validating platform performance by running focussed test cases (generated by a RIG tool) across multiple hardware and software environments.

The biggest challenge is in preparing the validation plan (used for generating test cases which will hit corner cases missed while designing the server processor, due to limitations of the verification tools to simulate the real time signals) in the least possible time, which makes the server processors fool proof before shipping it to the customers at the earliest.

1.2 Aim of the project

1. Studying and understanding Machine Check Architecture, a mechanism of detection and correction of hardware errors, primary contributors for uncorrectable errors, compromising the integrity and reliability of the servers.

2. Learning how the validation plan is created, what are the parameters to be considered before generating test scenarios to hit corner cases.

3. Learning the debugging approach used in Intel to root cause different kinds of failures (hardware, tool specific, test environment specific) .

4. In the process, developing an understanding on the validation flow for hardware error detection and correction mechanism used by Intel Xeon Server Processor.

1.3 Thesis Organization

Chapter 2 Literature Survey of the project.
Chapter 3 Description of MCA(Machine Check Architecture) and RAS its implementation in XEON server processor family.
Chapter 4 Description of RIG Tool.
Chapter 5 Post Silicon Methodology followed in Server Validation and Enabling Group.
Chapter 6 Debug techniques followed to spot a bug.
Chapter 7 RMCA and flow for Debugging RAS failure using RIG Tool.
Chapter 8 Result.
Chapter 9 Conclusion and Future Scope.

Chapter 2

Literature Survey

Various research papers related to Post Silicon Validation were studied.

Hemant Rotithor,Postsilicon Validation Methodology for Microprocessors, IEEE Design & Test of Computers, Technical Journal, was referred to understand the validation strategy while designing the test cases. The validation cycle at Intel was studied and is highlighted in the following sections.

2.0.1 Validation History

This chapter describes how improvement in validation techniques are made and requirements are met at Intel. CPU post-silicon platform-level validation cycle at Intel, consists of simultaneous development and testing of system validation (SV), compatibility validation (CV), and OEMs programs. Figure 2.1

Validation support for server processors requires more complex test development efforts to support requirements that are quite different from mainstream desktop, mobile and enterprise product lines. Products also requires high level of integration to meet customer requirements, posting new challenges to validation. The technologies used in post-silicon to verify functionality are:

- Analog validation (AV),
- Compatibility validation (CV) and
- System validation (SV)

SV represents controlled, virtual implementation at every interface in a platform environment. AV verifies signal integrity, and CV runs in a real-life operating system with peripherals and applications.

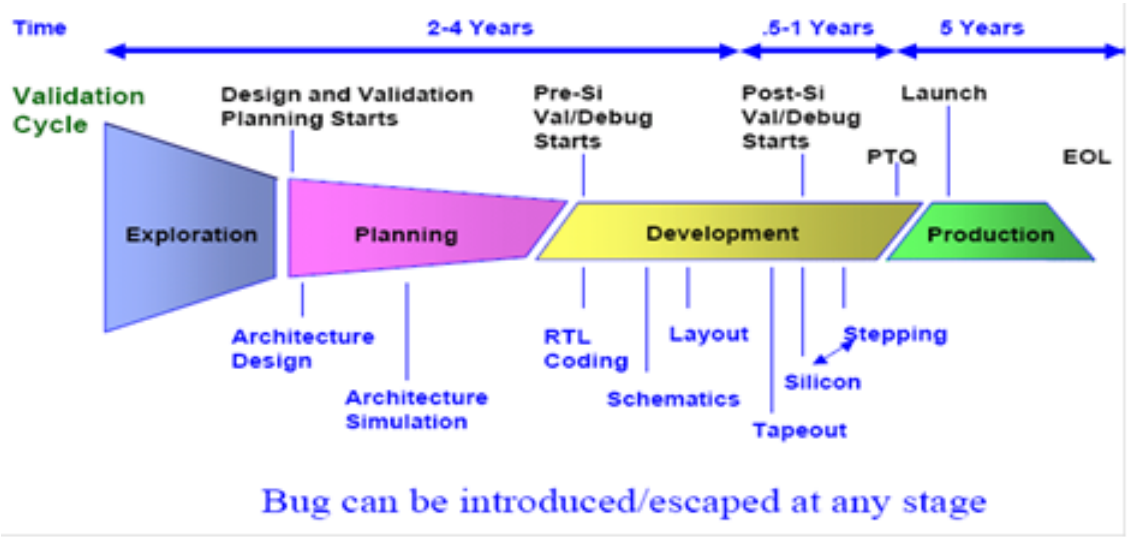


Figure 2.1: Validation cycle in Intel
[11]

Server processor support, however, may demand validating a rich set of peripherals with an Intel architecture processor on a single die. This brings the customer a wealth of inexpensive tool chains, programming language support and scalability across diverse product offerings. It also brings testing challenges that cannot be met by reuse alone.

In presilicon validation, the challenges of supporting a large-gate-count CPU along with large gate count standard I/O functionality are magnified by the sheer size of the overall model. Currently, the entire silicon model cannot fit into one tool chain and must be processed in chunks across various tools.

In post-silicon validation, the biggest issue is likely to be the lack of visibility into the internal states of the silicon for debug during validation runs. Access to the deep internal states of the silicon, is directly related to reducing debug and verification times and thereby reducing back-end validation costs.

Three strategies are needed to enable server processor support.

- The first is more exhaustive presilicon validation. The benefits are fewer tape outs in the back end, resulting in faster time-to-market and reduction in project cost.

- The second (post silicon validation) is to enhance the built-in-self-test functionality of the processor and core logic at a platform level.

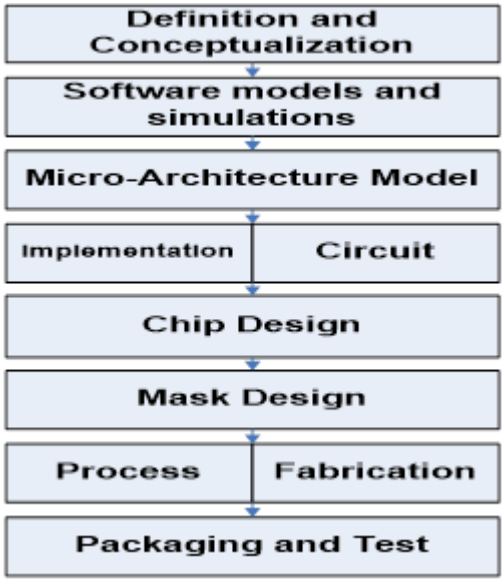


Figure 2.2: Processor design flow

- The third strategy is to use a high ratio of known-good building blocks to newly designed blocks of silicon for internal units, connecting them with an internal communication bus, enabling isolated testing of each block. By using this method, stress on front-end tools is reduced and higher-level formal verification methodologies are made use of. Refer Figure 2.2.

2.0.2 Comprehensive validation process Followed in Intel

As shown in Figure 2.3. There are eight stages in Intels comprehensive validation program [13]:

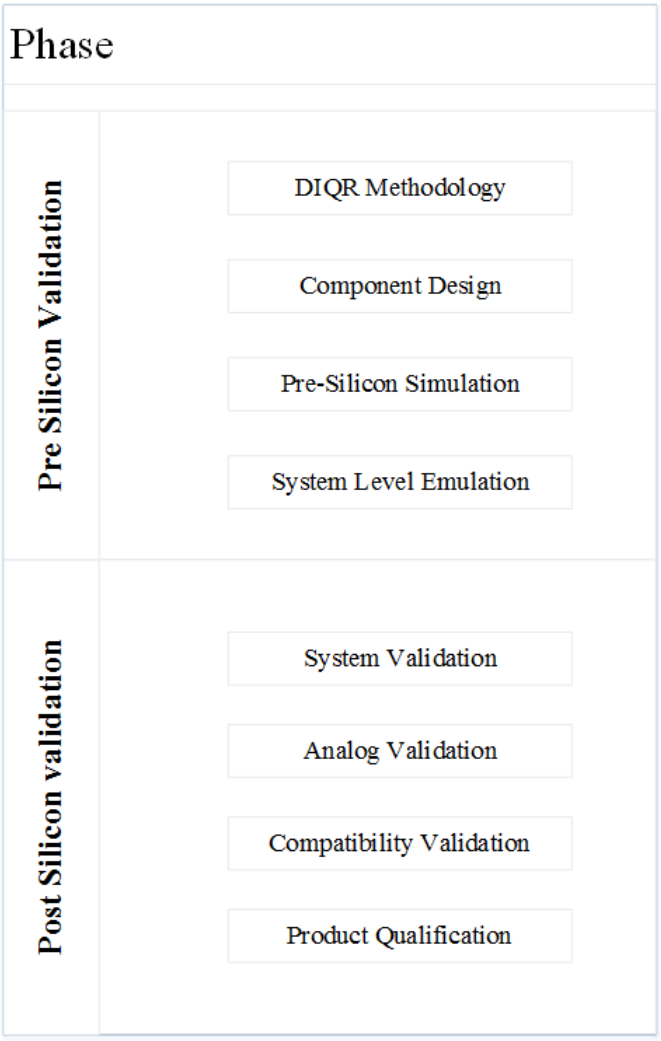


Figure 2.3: Stages in Comprehensive Validation Cycle followed in Intel [13]

Chapter 3

RAS and MCA(Machine Check Architecture)

Various white papers and Intel IA 32 software developers manual pertaining to RAS and MCA were studied. The details in this chapter describe the understanding made from them.

Core RAS means RAS features related to Core components only, for example memory controller, Cache, on chip registers. Project is limited to validation of Core RAS features only, and specifically, validating L1 cache by corrupting the Main Memory (DRAM) when accessed by Cache will be flagged as a Machine Check Error and then the MCA comes into picture.

Figure 3.1 describes the areas where RAS is implemented in Intel Xeon E7 processor.

3.1 Introduction to RAS

RAS stands for Reliability, Availability and Serviceability.

Traditionally, server RAS was thought as errors that were to be handled by the hardware only, and unrecoverable hardware error would result in entire system going down, causing hindrance to the organization and end user. Since the online transactions are increasing exponentially, hence the infrastructure supporting it should be up and running all the time, hence they should be able to handle unrecoverable hardware errors for delivering continuous service to the end user. Modern approach is designed to handle unrecoverable errors, right from the infrastructure hardware to the underlying software Figure 3.2.

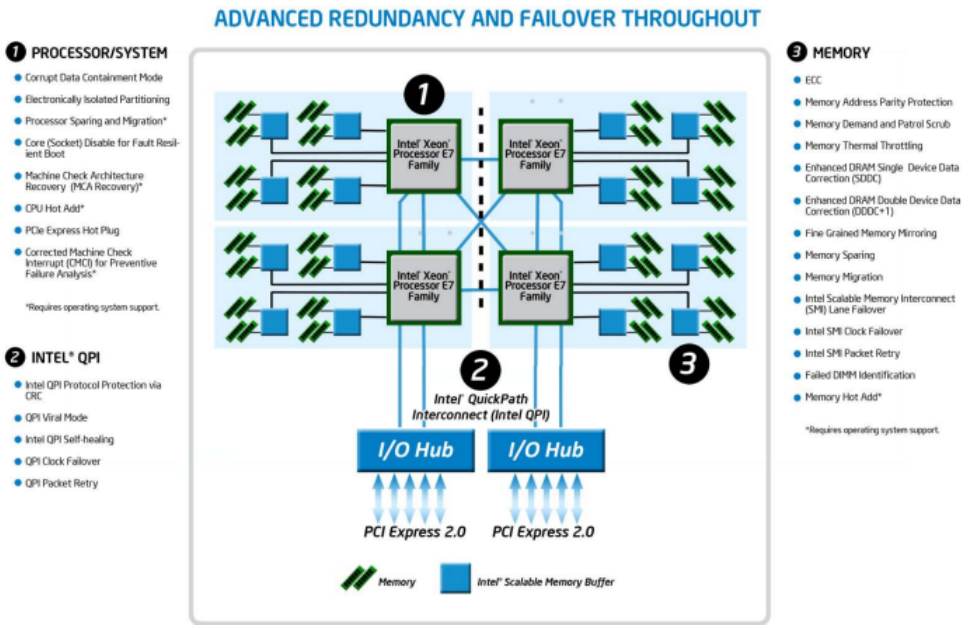


Figure 3.1: Advanced RAS features on a Xeon E7 Server
[1]

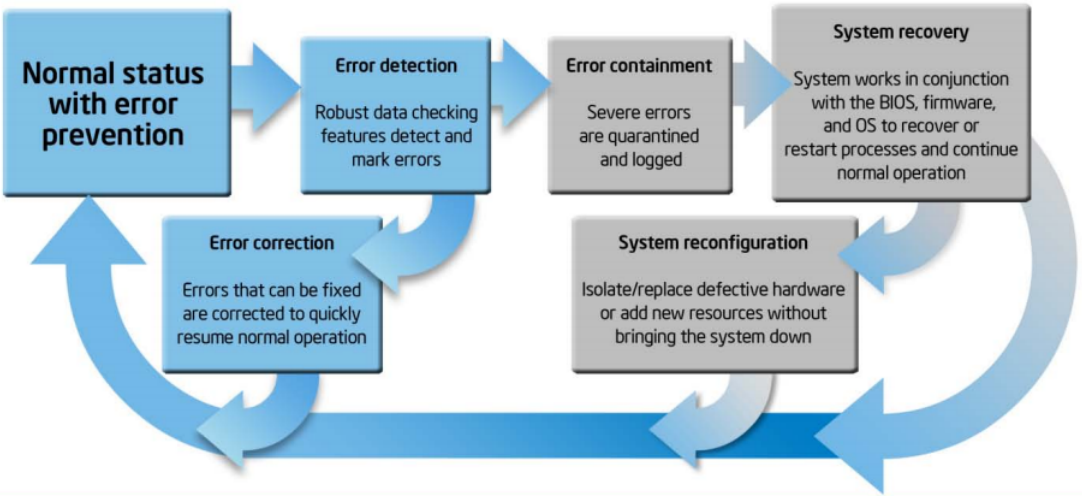


Figure 3.2: RAS Flow
[1]

The modern approach is designed keeping in mind three aspects:

1. Reliability, how the infrastructure preserves data sanctity.

2. Availability, guaranteeing unhindered operation with minimal impact on performance.

3. Serviceability, how quickly can the infrastructure can be made to function to its maximum again.

1. **Reliability** Data sanctity, concerns the protection of data through detecting, correcting errors and containment of errors in the worst case scenario.

   Error detection identifies the errors at the instruction and data level. Error Correction is a mechanism by which the hardware detects hardware error and corrects the faulty bits in a memory location.

   Error containment segregates the corrupted data and broadcasts it across all major components and data buses so that the other subsystems can take appropriate action on encountering such errors. [1]

2. **Availability** Modern design approach provide mechanism to enable continuous operation even on occurrence of uncorrectable errors. The approach makes use of spare processors, DIMMs and I/O resources in case of a catastrophe. Making use of BIOS/OS to log and recover from an uncorrectable hardware error if possible.

3. **Serviceability** This approach makes use of predictive failure analysis to predict the components that may lead to uncorrectable errors in the future by polling the current state of the system thereby reducing the downtime.

   System isolation and partitioning is used to segregate the components affected by uncorrectable errors from components running on server infrastructure to aid in maintenance.

3.1.1 Hardware Errors and Self-healing

Hardware errors affect computed data stored in memory, data in transit between components. Such errors can affect the precision and sanctity of data used in computations. Hardware errors fall into two categories: soft (transient errors) and hard (permanent errors).

Soft errors occur mostly due to surrounding radiations in the form of random signal affecting electronic circuits at the molecular level, such as alpha particles or cosmic rays dislodging electrons and therefore transferring charge, thereby changing

the logic behavior of one or more gates. Soft errors seriously affect dynamic random access memory (DRAM). Soft errors, can be corrected by circuitry that can change the logic state of a failing bit.

Hard errors are permanent physical failures at the hardware level, e.g., a stuck bit in a data bus, a bad bit in a dual inline memory module (DIMM), or a faulty internal circuit in a processor.

Hard errors can be corrected only by physical replacement with a spare component.

Self-healing systems can diagnose and recover themselves automatically from component level hardware failures. Self-healing requires a robust failure detection, interpretation, handling and failure isolation capabilities.

The method to ensure reliability is to detect and correct error wherever possible, recovering from errors by replacement of failing component or memory interconnects, and preventing errors that may spring up in the future.

Error correcting codes (ECCs) were devised to detect and correct multiple bit hardware errors. One ECC in common use is SECDED (single error correct double error detect), which allows the correction of one bit in an error or detection of a double-bit error in a memory block. [5]

Currently the Xeon D 1520 and 1540, launched in April, 2015 can correct up to 4 bits in a memory block. Hardware errors can be classified under two categories

1. Correctable
2. Uncorrectable
    (a) Recoverable uncorrected
    (b) Non recoverable uncorrected
        i. Fatal Error Blue screen error, this error is not supported by Random Instruction Generation Tool
        ii. Non Fatal Error

Correctable Errors are handled in Silicon using ECCs and doesnt result in slowing down the systems performance.
If the error is not correctable, processor signals the OS to take over. OS checks whether the uncorrectable error is recoverable or not, if it is recoverable, OS finds out whether any task is using the poisoned data, if not, then it unmaps it form the main memory and flags it as suspicious, if an application is using the poisoned data

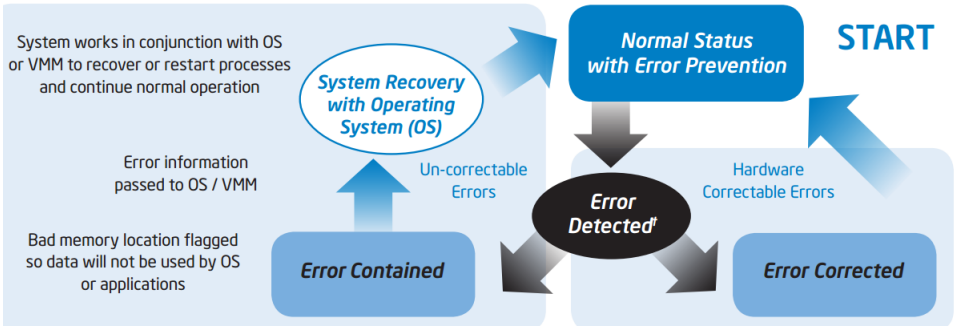


Figure 3.3: MCA recovery
[1]

, then it transfers the control to the application software to perform the recovery action, if recovery is not possible, the task is aborted, else the poisoned memory is reconstructed. If the uncorrectable error is fatal in nature, then the OS signals system shutdown (the error was fatal uncorrectable error). Figure 3.3

3.1.2 Memory Errors

The most common cause of crashes in a large scale systems are memory errors, to recover from a memory failure, replacement of the faulty module is the commonly used practice. Hence, memory modules are amongst the most frequently replaced server components. But, memory replacement is quite costly. ECC was developed in the 1990s to reduce the frequent replacement of memory modules by detecting hardware errors in the form of bit flips in memory modules and recovering from them.

ECC reduces fatal memory errors responsible for system crashes, helping the systems maintain an illusion of operating error-free when the error rate is limited to a particular range.

There have been a lot of advancements in ECC technology, for example IBM Chipkill (trademark), which can detect and correct up to eight-bit flips in memory. Chipkill is used with memory sparing method, keeping a spare memory module on a channel, so that it can be used in case a memory module gets corrupted above the correcting capability of chipkill technology. Chikill technology is helpful in improving reliability of a server. [7, 8, 6] Intel also provides an improved ECC feature along with the hardware called Enhanced Double Device Data Correction (DDDC), allowing

recovery from two back to back DIMM failure due to bit flips. MCA Recovery is an Intel Xeon processor family E7 family feature allowing the Silicon layer to lay down the rules for recovery of uncorrectable hardware errors making use of Operating System, Virtual Machine Management and even application software.

As shown in Figure 3.4, in case the hardware error is a correctable error, the Silicon using the intrinsic ECC technique corrects it and notifies the OS to keep track of the number of times correctable error was hit, so as to improve the predictive error analysis . If the error cannot be resolved by the hardware, it signals a machine check to the OS. The OS then checks whether the faulty page causing the uncorrectable error is currently in use by the memory or not, if it is not being used, then it is disassociated from the memory and marked for repairing. If the page is currently mapped to the memory, the application using it is notified that it has cached a poisonous data, and the application checks whether or not the corrupted data can be marked for repairing. If the data cannot be repaired, then the OS signals for termination of the application currently holding the poisonous data.

The striking feature of this approach is the co-ordination between hardware and software while detecting whether the hardware error is correctable or uncorrectable. This collaboration can save a lot of resources, since the cost of including all of this in the hardware would result in huge pressure on the designers and would prove costly.[1]

3.2 Hardware error

Examples of hardware error sources include the following:[9]

- Processor machine check exception (for example, MC no.)
- Chipset error signals (for example, SCI, SMI, SERR#, MCERR#)
- I/O bus error reporting (for example, PCI Express root port error interrupt)
- I/O device error

3.3 Scope of the project

RAS validation is a very important concept for maintaining quality and efficiency of a server.

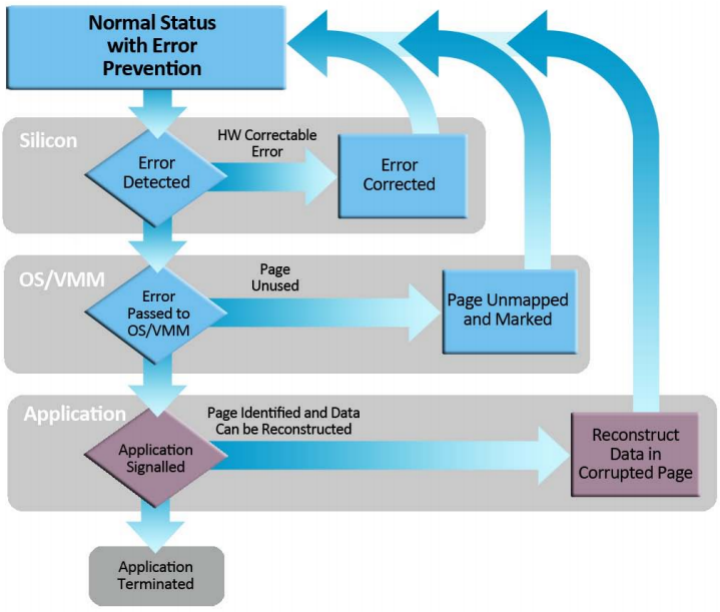


Figure 3.4: Software-assisted MCA recovery process
[1]

Machine Check Architecture is a mechanism for validating RAS for a number of core and uncore components.
In SSV, the scope is limited only to validate the machine check raised by core components. For example, cache, Memory controller etc. after main memory is poisoned. Refer Figure 3.5
In this project, the main memory is poisoned by RIG using the DFX while generating the random code.
Now, as soon as a component tries to access the poisoned memory location, a machine check is signalled, RIG also generates the code for the machine check handler (for servicing the hardware error).
The topic of discussion in this report is limited to poisoned memory access by the cache, and the steps to validate whether the cache is able to recover from the different severity of errors while being exercising other instructions related to security, power management etc.

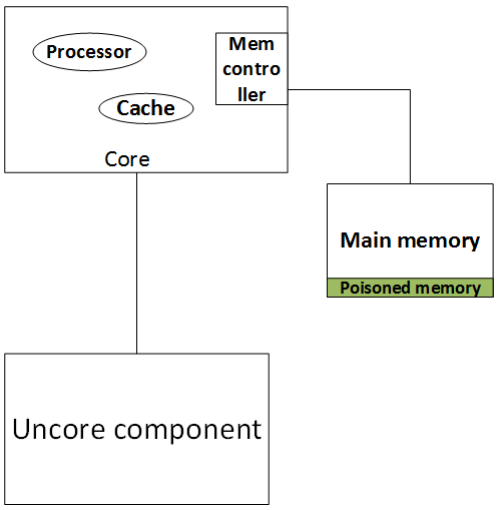


Figure 3.5: Diagram depicting the scope of RAS validation in this project

3.3.1 Real world scenario of a hardware error and how it gets handled

Suppose Flipkart is using an Intel Xeon server, after a finite time duration, on a certain instant, the main memory gets corrupted because of frequent reading and writing into it, and a customer logs in to its website to purchase an expensive mobile, things go smoothly, but as soon as he tries to make payment, the application handling this transaction access a poisoned memory location, at this instant a machine check will be signalled, the processor will detect whether it is correctable or uncorrectable, suppose hardware is not able to correct it, its poison tag is enabled and the core signals a Machine Check Error across all other cores with poison tag enabled, so that other cores do not try to access the memory location with a poison tag

3.4 Machine-Check Architecture

Machine-check architecture provides a mechanism for detecting and reporting hardware (machine) errors, such as cache errors, ECC errors, system bus errors, TLB errors and parity errors.

There are two kinds of Machine Check Registers, one for setting up the machine check banking registers and the other ones are used to record the errors detected

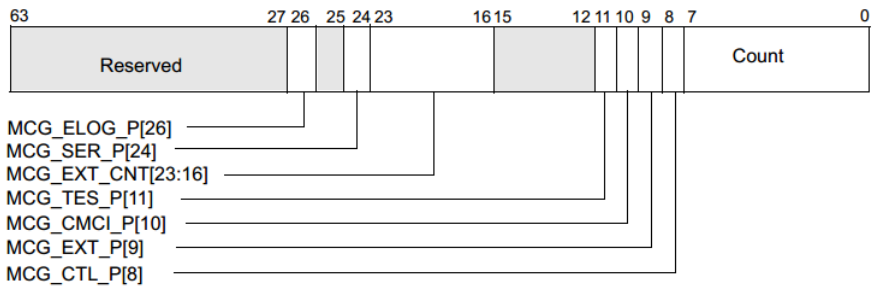


Figure 3.6: IA32_MCG_CAP Register
[10]

during machine check. Machine Check Exception (MC) is signaled when the processor encounters an uncorrectable machine check error. The machine check exception is a mechanism by which the processor begins to thoroughly analyze an exception after it crosses a certain threshold (that is when the hardware is no longer to resolve the hardware error). The machine check exception handler collects the details of the processors state from the machine check MSRs.

The processor on being able to recover from a hardware error, signals a CMCI (Corrected Machine Check Interrupt). To keep track of the number of times a particular hardware unit encountered a hardware error for predictive failure analysis by the OS. [10]

3.5 Machine-Check Global Control MSRs

The machine check global control MSRs include the IA32 MCG CAP, IA32 MCG STATUS and IA32 MCG CTL.

IA32_MCG_CAP MSR

The IA32_MCG_CAP MSR is a read-only register providing information about the machine-check architecture of the processor. Figure 3.6 shows the structure of the register in Intel Xeon family processors.[10]

IA32_MCG_STATUS MSR

The IA32_MCG_STATUS MSR describes the current state of the processor after a machine-check exception has occurred (see Figure 3.7).[10] Where:

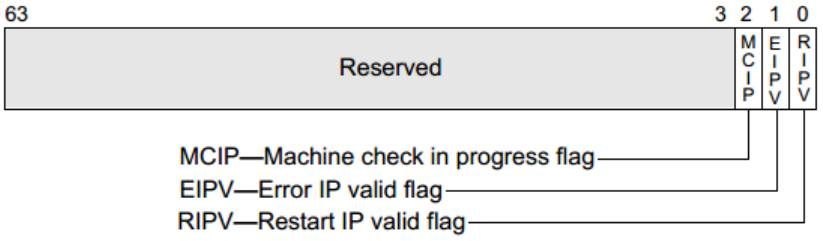


Figure 3.7: IA32_MCG_STATUS Register
[10]

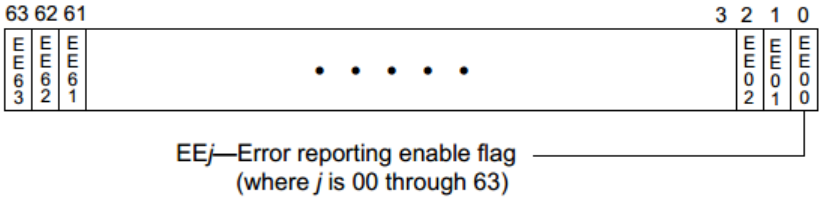


Figure 3.8: IA32_MCi_CTL Register
[10]

IA32_MCG_CTL MSR

IA32_MCG_CTL controls the reporting of machine-check exceptions. If present, writing 1s to this register enables machine-check features and writing all 0s disables machine-check features.[10]

3.6 Error-Reporting Register Banks

Each error-reporting register bank can contain the IA32 MCi CTL, IA32 MCi STATUS, IA32 MCi ADDR, and IA32 MCi MISC MSRs. The number of reporting banks is indicated by bits [7:0] of IA32_MCG_CAP MSR (address 0179H).[10]

3.6.1 Corrected Machine Check Error Interrupt

CMCI improves the efficiency of processor in the sense that the processor no longer has to continuously poll the correctable error count, to find whether it has exceeded a particular threshold. The software can program the threshold value in IA32_MCi_CTL2 MSRs register to deliver a local interrupt.

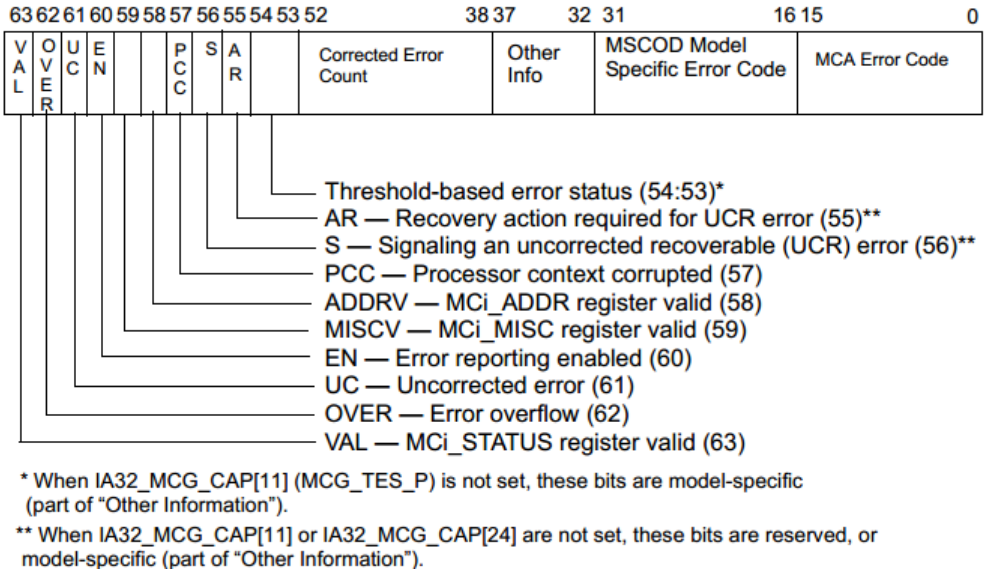


Figure 3.9: IA32_MCi_STATUS Register
[10]

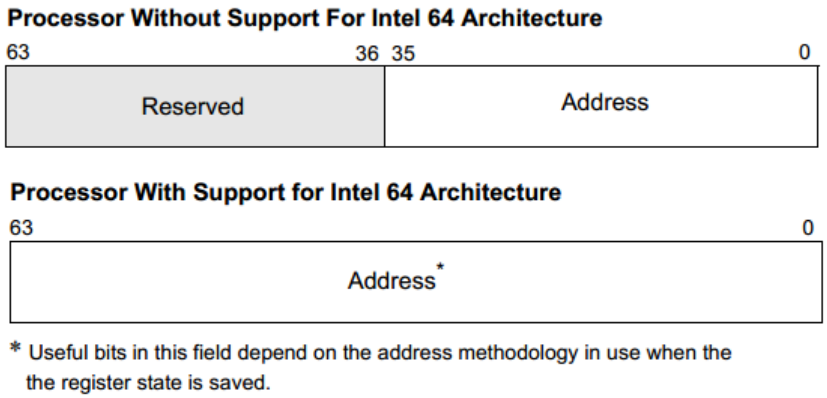


Figure 3.10: IA32_MCi_ADDR MSR Register
[10]

System software is required to enable CMCI for each IA32_MCi bank that support the reporting of hardware corrected errors if IA32_MCG_CAP[10] = 1.

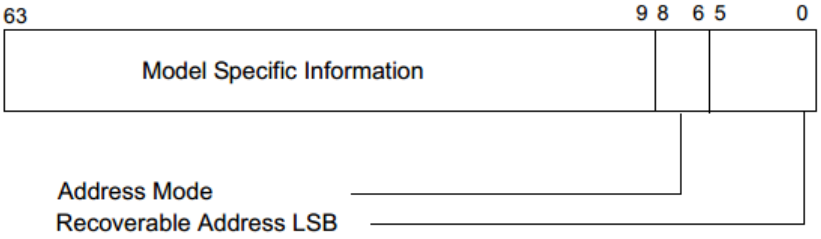


Figure 3.11: UCR Support in IA32_MCi_MISC Register
[10]

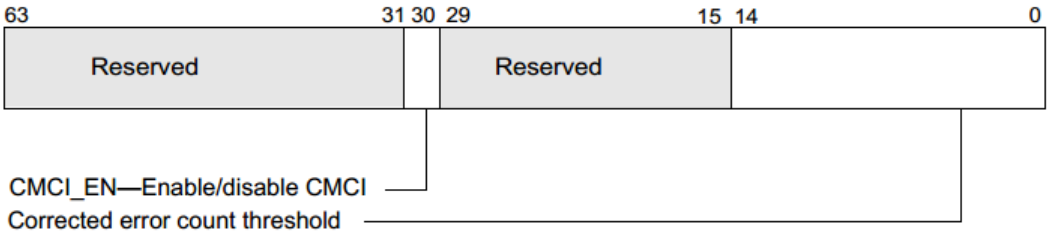


Figure 3.12: IA32_MCi_CTL2 Register
[10]

3.6.2 Detection of Software Error Recovery Support

Bit 24 of IA32_MCG_CAP (MCG_SER_P) is used to detect whether software error recovery support is enabled or not. When 24^{th} bit is set, indicates that software error recovery is supported by the processor. When this bit is cleared, it indicates that software error recovery is not supported by the processor. Hence in case bit number 24 is set, would mean that if the hardware is not able to correct the hardware error by itself, it will have to issue shutdown.

The type of error which can be recovered by use of a system software are called Uncorrected Recoverable Errors (UCR). UCR error indicates that it has been detected as an uncorrectable error, but has not corrupted the processor to an extent such that the state of the processor can be restored by taking use of system software. The processor can resume its normal operation once recovery has been done by system software. The machine check handler by using error logging technique by making use of error reporting MSRs to analyze and implement actions for recovering from UCR error.

Type of Error[1]	UC	PCC	S	AR	Signaling	Software Action	Example
Uncorrected Error (UC)	1	1	x	x	MCE	Reset the system	
SRAR	1	0	1	1	MCE	For known MCACOD, take specific recovery action; For unknown MCACOD, must bugcheck	Cache to processor load error
SRAO	1	0	1	0	MCE	For known MCACOD, take specific recovery action; For unknown MCACOD, OK to keep the system running	Patrol scrub and explicit writeback poison errors
UCNA	1	0	0	0	CMC	Log the error and Ok to keep the system running	Poison detection error
Corrected Error (CE)	0	0	x	x	CMC	Log the error and no corrective action required	ECC in caches and memory

Figure 3.13: Machine Check Error Classification
[10]

3.6.3 UCR Error Reporting and Logging

IA32_MCi_STATUS MSR is used to report UCR errors and existing corrected or uncorrected errors.

When IA32_MCG_CAP[24] is set, a UCR error is indicated by the following bit settings in the IA32_MCi_STATUS register:

- PCC (bit 57) = 0
- UC (bit 61) = 1
- Valid (bit 63) = 1

3.6.4 UCR Error Classification

With the S and AR flag encoding in the IA32_MCi_STATUS register, UCR errors can be classified as, refer Figure 3.13:

Chapter 4

RIG Tool

4.1 RIG at Intel

As shown in Figure 4.1. RIG used at Intel is a template-driven pseudo-random instruction stream generator. It is used during both pre and post silicon to validate the RTL and silicon. The test results are validated against reference CPU model. Templates, written in C++ and assembly language (IA-32 and/or IA- 64), describe where and how randomness will be introduced into the test cases. An architectural reference CPU model is used to generate the test cases and serves as a basis of comparison against the device under test. As each random instruction is generated, the instruction is simulated. After each test case, template-specified memory ranges are compared after running the test case on simulator with that on the hardware under test. Hardware abstraction is used so the templates can run in many different environments. RIG at Intel is used in System Validation (SV), Compatibility Validation (CV), Architectural Validation (AV), and Micro-Architectural validation (uAV). RIG templates are designed to generate directed-random assembly code. A copy of generated assembly code is given to the golden reference model and the hardware under test. A checker tool compares the architectural states during simulation and flags the differences. RIG at Intel currently supports the IA-32 and IA-64 ISAs.

4.2 RIG Architecture

There are seven main components to the RIT architecture. Each component is described in the following sections.

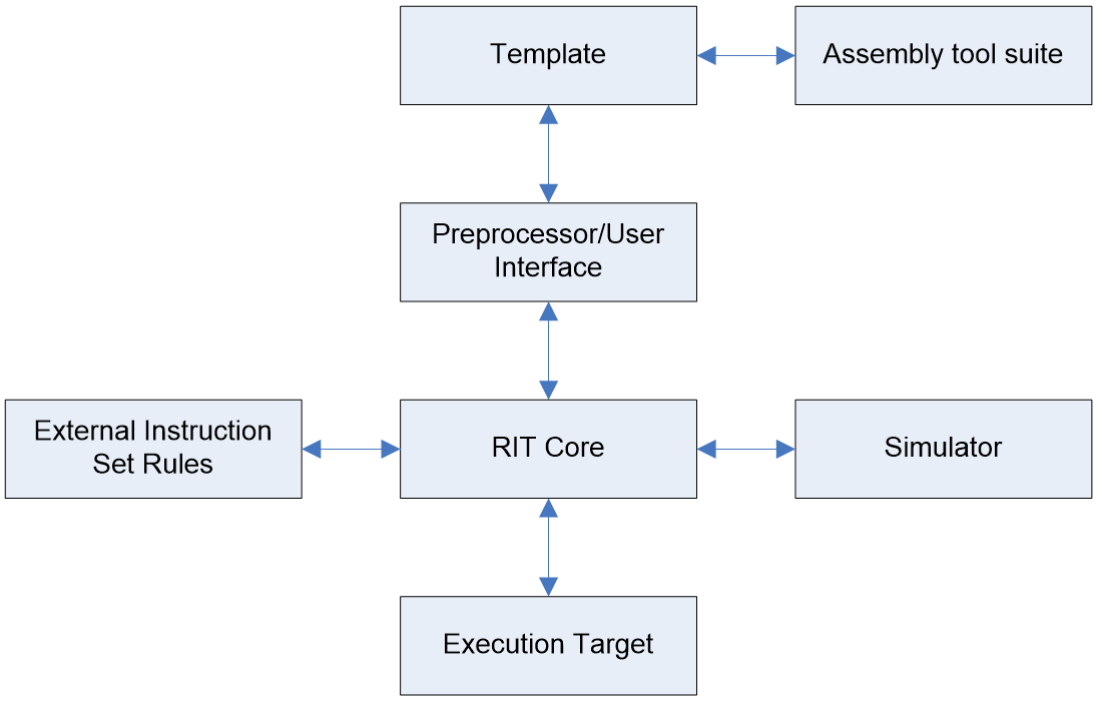


Figure 4.1: RIG architecture

4.2.1 Template

The template is one of the users primary interfaces into the RIG architecture. Via the template, the user provides the framework in which the random data and instructions will be placed. This defines the structure of the test that will run on the DUT. In a template, you do the following:

- Configure the target test environment
- Customize the distribution of random code generation
- Control the test execution.

Everything necessary to ensure correct execution of the template is contained within the template. The template concept in RIG gives this control to the user, who can change the format of the test, with no impact on the other elements of RIG.

4.2.2 External instruction set rules

The RIG architecture provides built-in code generation capability (rules) for the IA-32 and IA-64 instruction sets. This portion of the RIG architecture allows plug-and-play extensions to the built-in code generation mechanisms without adversely affecting the core of RIG and without having to create a new version of RIG.

4.2.3 Simulator

The simulator in the RIG architecture is responsible for simulating the test case defined by the template. This includes both fixed and randomly generated code. Fixed code refers to the hand-written assembly code present in the template. Random code refers to the instructions generated randomly by RIG.

4.2.4 Assembly tool suite

The assembly tools suite is responsible for assembling, linking and binding the assembly language portion of the template. The output of the assembly tool suite, an executable module, is then loaded onto the simulator.

4.2.5 Execution target

The execution target represents the final destination of the test case defined by the template. Examples of execution targets would be SV test vehicles or an RTL model.

4.2.6 RIG Core

The RIG core is the heart of the RIG architecture and is resposible to binding all the attributes into a single entity.Test Architectural changes shouldnt impact the RIG core to a large extent, it should be designed to provide a common architecture and interface.
The RIG core is designed to be as adaptive to the targeted microprocessor architecture as possible, allowing RIGs for both future and existing microprocessors to be developed off the same foundation.

4.2.7 Preprocessor/User Interface

The use of RIG can be typically divided into two phases: template development and validation. During template development, templates are being written and debugged. This phase requires heavy use of the template preprocessor. The template preprocessor essentially prepares the template for use by RIG.

Template Processing Template processing flow is described in the Figure 4.2. After pre-processing template source code, two output files are generated by the micro pre-processor.

- C++ file
- Assembly code file

The C++ file contains all of the C++ code present in the template, as well as the auto-generated code. The assembly code file contains all of the assembly code present in the template. Space will be reserved in the assembly code for the random data and random code blocks that is filled in later by RIG. Everything that occurs during the pre processing stage is automatic and should not concern the template writer.

The template writer may add intelligence several ways: Biasing in Templates: A critical part of template writing. Biasing is one of the easiest methods of generating random code. Almost every random value the rules module generates can be controlled by biasing. For preventing access to a particular register, or lowering the instances of such an access, biasing is a fairly simple solution. Bias files are used for controlling the generation of a random code stream (output) without modifying and recompiling existing templates. Inside the template, biasing is divided into two phases:

- initial biasing
- dynamic biasing

Initial biasing occurs only in the Init Region of a template. Any code in the Init Region is executed once. It is the first piece of code RIG executes inside the template. Biasing that occurs within the Init Region will occur only once. Initial Biasing allows global adjustments to be made on code generation. For example, if a template should not generate any random floating-point instructions, then the

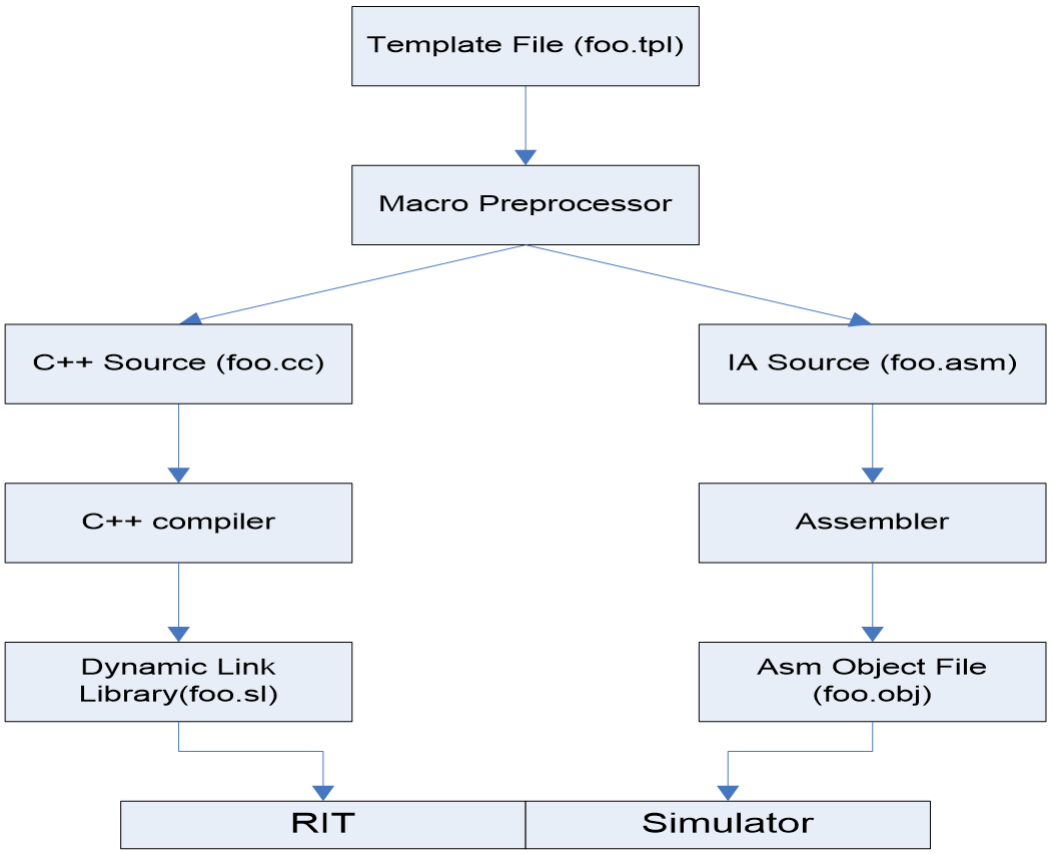


Figure 4.2: Template processing flow

floating-point branch of the instruction tree should be biased to zero. Dynamic biasing can occur throughout template execution. Bias files are useful for eliminating certain instructions from occurring and Increasing the probability of an instruction to occur randomly Libraries: Template writers can create libraries that will randomly be placed into the code stream. These libraries allow the user to exercise more control over a block of code without having to specify exactly where the code goes. Post-emulation checks via the Rules Module: After each random instruction is executed on the simulator, there is a check to make sure the results of the execution are desirable. If the instruction places the processor in an undefined state, the rules module will discard the instruction. After the rules module completes its check, it will call any other functions that have been specified by the template writer.

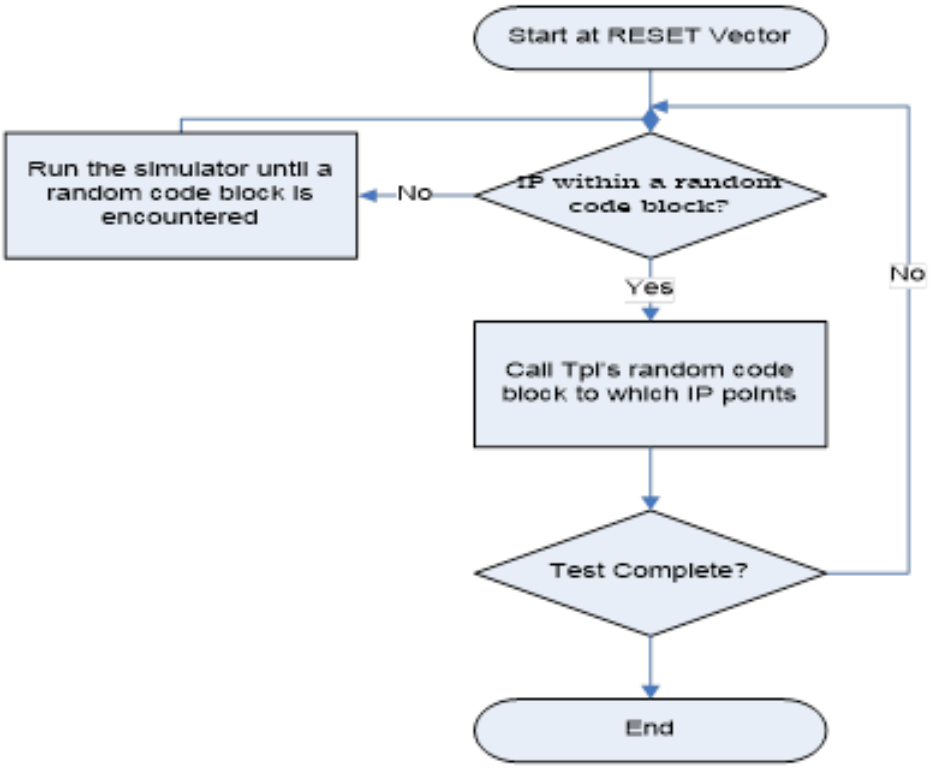


Figure 4.3: Random Code Generation Flowchart

4.2.8 Generation of Random Code

RIG generates random code only as it is needed, in a track laying style, analogous to railroad workers laying down track only right in front of a train as it needs it. RIG is linked to a behavioral simulator that simulates the test as it is being generated.

The Code Generation diagram above, illustrates the flow of random code generation. The simulator is reset before the generation of each test. RIG examines the simulators Instruction Pointer (IP) to determine whether it lies within one of the templates random code blocks. If it does, then the templates C++ code corresponding to the IP is called to direct random code generation. Again, since C++ is used to describe how to generate test code, the user has great flexibility. The user may explicitly specify instructions to introduce, read instructions from a file, or ask RIG to generate instructions randomly (subject to the current biasing state). If the IP does not point to a random code block, then RIG directs the simulator to simulate the invariant template assembly instructions, without RIG intervention, until a random code block is encountered. This process continues until a template

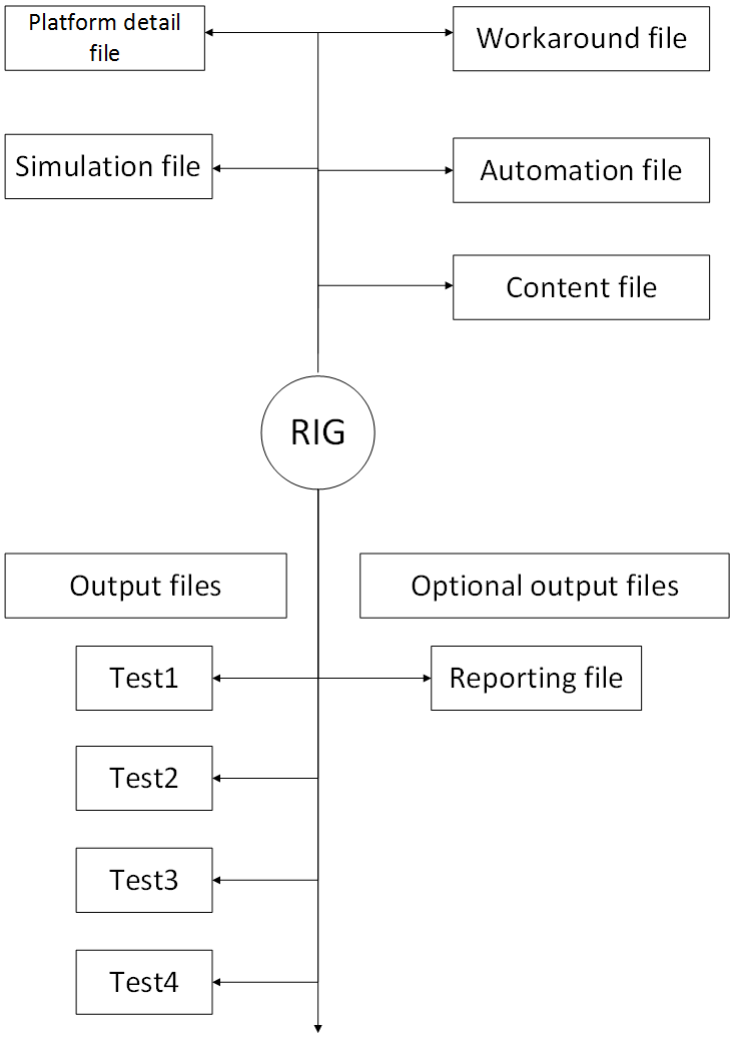


Figure 4.4: RIG Diagram

directive is encountered signaling that test generation has completed.

RIG templates are developed and run to produce directed-random assembly code. The assembly code is run on both the golden architectural reference model and the RTL model. A checker tool compares the architectural states during simulation and flags any mismatches. RIG at Intel currently supports the IA-32 and IA-64 ISAs. A typical structure of Random Test Tool is shown in Figure 4.4

Random Test Tool takes the following input files to generate a Test. Platform Specific File: It is a large file describing the number of available processors, their CPUID information, the available memory, APIC IDs, Non-Uniform Memory Access configuration, and specific Machine Status Register data. The Figure 4.4 describes

the input and output files pertaining to an RIG.
Description of the files :

1. **Working File**: The working file allows enabling various workarounds within RTT. Generally, workarounds are added to bypass certain processor fuse, architectural simulator, environment or Random Test Tool bugs. This file allows the flexibility to supersede certain changes which the validation engineer wants to disable/ enable while generating test cases.

2. **Automation File**: This file is used when Random Test Tool is working under automation Infra structure.

3. **Content File**: This option allows us to do things like specialized cache setup, power management setup, etc.

4. **Simulation File**: These files are used to simulate the processor model of current working processor.

5. **Test1**: The Test1 file basically contains the command line used to generate the Test, and when the generator is generating the Tests for the loader, it indicates start of execution by signaling Test generation completed message as a marker to loader for starting execution.

6. **Test2**: When the loader starts the execution it reads Test2 file and sets target configuration accordingly.

7. **Test3**: Code and data segments are present in this file and it contains the op codes of the instructions which are executed by the loader on the target under test. It also contains the description of:
   Location of System Data Structures and Support Code:Shows where the system control blocks (page table structures, GDT, IDT, and LDT) and the setup, completion, and other non-random code modules were placed in memory.

8. **Test4**: Test4 file contains information generated by simulator which is to be compared with expected result (on the target) in order to make a decision whether the test case passed or failed.

9. **Reporting file**: The reporting file is generated by RIG to aid in debug. A great deal of information is contained in this file(s). Some of the most often used information is the assembly code run on each thread, and the error

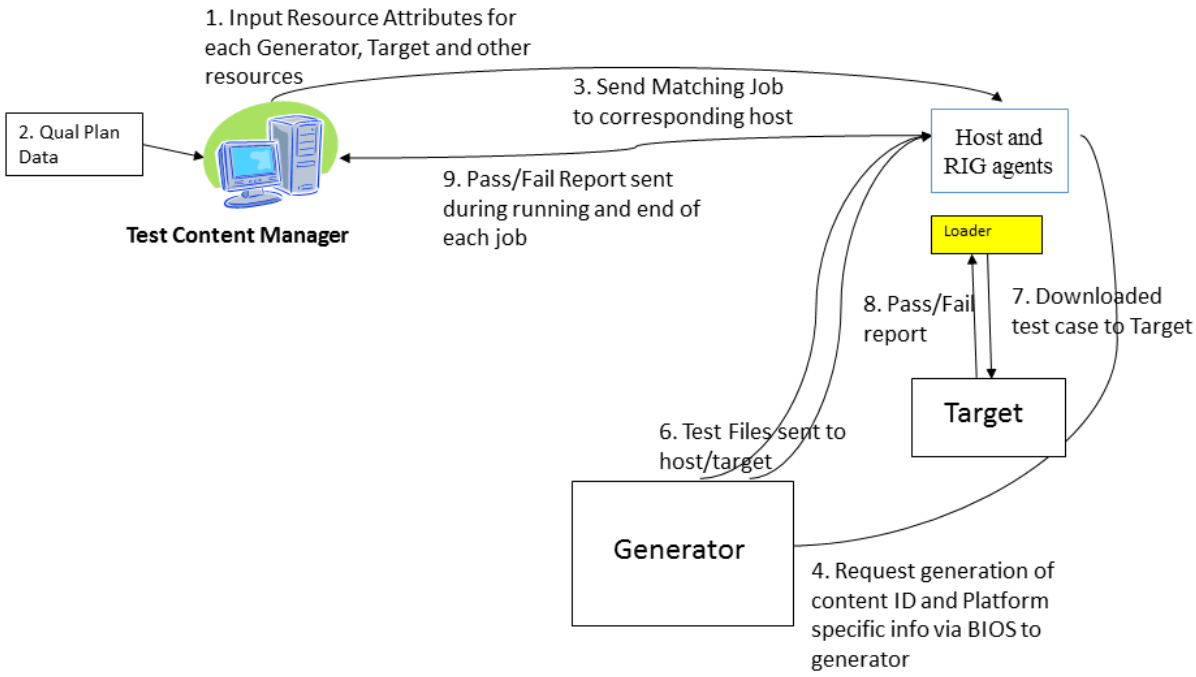


Figure 4.5: RIG and infrastructure tools flow

message block decode information. Whenever user is debugging a failure, user will want to generate the reporting file to gain extra information.

4.3 Infrastructure for validation set up using RIG

Random test instruction generation at a high level consists of following modules (Figure 4.5)

4.4 Steps involved in test case generation (to target RAS features) using RIG

Random test case generation in RIG is divided into 3 sections, launcher, generator and executor.

1. Launcher
2. Test Generator
3. Test Executor

1. **Launcher module** It parses the command line and checks for its validity and splits one platform capabilities file into desired number of files, by considering all memory and other resource constraints. Random Test Tool takes these spitted Platform capabilities file and other input parameters, and produces more number of standalone Tests. Random Test Tool is already developed module which generates the test cases.
2. **Test Generator** In this phase, the tool collects the parameters from the input file and selects the weightage for each parameter and generates the test instructions, in case of input files, designed to target RMCA, the test case is divided into 3 major sections; the set up section, which is used to set up the processor registers before poisoning the main memory; the random section, which contains the instructions which will try to access the poisoned memory locations; completion section, which will do the cleaning up of poisoned memory locations :
   (a) **Set up code**
      i. Encode memory Poisoning Non Repeated Setup code
      It allows poisoned status to be synchronously transferred with data to the receiver.
      ii. Apply processor Workarounds
         A. Disable MC Exception Overflow Broadcast
         B. Disable LLC Writeback Machine Check Exception Signalling
         C. Disable Speculative Data Access and Prefetch
      iii. Initialize per package registers

A. Elect a non random cafe thread from each package for memory poisoning.
B. Unlock the memory poisoning DFX (Design For Testing) mechanism on all sockets.
C. Memory poisoning DFX must be unlocked on all sockets.
D. Initialize PCIe registers per package.

iv. Inject machine check errors using ITP based DFT.
v. Enabling RMCA feature
vi. Enabling memory poisoning feature
vii. **Completion Code**

A. Disable RMCA
B. Disable Memory Poisoning
C. Enable Machine Check Exception Overflow broadcast (disabled in non repeated setup code)

3. **Test Executor** Executor is used to run the Test onto the target and check whether the results are matching with the golden reference values obtained from running the Test on the simulator.

4.5 Test case generation

Content file contains a set of text blocks called switch. These switch form a large control panel that lets you select processor operating modes and other operational characteristics, aspects of the architecture to be tested, and instructions to be focused. When RIG is run, it reads these settings in the content file and generates the test code accordingly.

4.6 Significance of content file in system validation

In case of RAS validation through RIG, content file plays a vital role in shaping the effectiveness of test instructions generated for introducing uncorrectable errors by poisoning memory locations (e.g introducing bit flips purposely in some cache lines) to check whether a system is able to recover from an uncorrectable machine check

exception or not. We can even specify how many cache lines to poison (by choosing a random number in switch). The content file contains certain switch which can be used to mold the generation of test cases in a desired manner so as to target a particular feature of a processor design (in this case the hardware error handling capability of a processor in a cache, QPI, SPI).

For making the content file, the validation engineer has to keep in mind a lot of factors e.g

- Whether it will be compatible with set up.
- Whether it will be able to cover all the corner cases.
- Whether it will take the least time to hit corner cases, since validation has to be done in a specified amount of time.
- Since generation of a test case according to content file takes a lot of parameters e.g no. of active processors, no. of active threads, no. of test instructions, memory of the host, the types of operating modes(since each operating mode has a limited amount of physical memory associated with it) it is targeting. Before actively testing a processor with huge amount of test cases, firstly the health of Silicon (processor arriving from fab yet to be released in the market) is to be tested by running a specific amount of Tests within a given set up, this stage is the Silicon bring up stage. Once enough confidence is build, the validation engineers set up the test plan to extensively fire random test cases generated by RIG and monitor the response of processor.

4.7 Biasing in RIG and importance of content file

Biasing is generally used to control the degree of randomness. Consider an example, if MUL has an absolute weight of 35% and MOV has an absolute weight of 90%, then, on average, MUL occurs 90 % of the time. Absolute weights should sum to 100%. Biasing fall into broadly two categories:

- Absolute,
- Relative.

Relative biasing signifies the priority of a particular instruction compared to others, and absolute biasing signifies the amount of contribution a particular instruction will

have. Absolute biasing if being mentioned in percentage should add up to 100%. Hard coding the Bias values to particular instruction results in uniform distribution of instruction by a Random Instruction Generator, resulting in poor quality of test cases generated, and it will take a lot of time to hit corner case scenarios.

By maintaining flexibility in selecting test instructions, the quality of test case also increases, as different selections will result in different scenarios.

In case of RIG Tool used in Intel for validation, every mnemonic has a bias associated with it. Mnemonics might be combined into sub-groups, for example branching instructions, looping instructions or floating-point instructions. The mnemonics weights within a sub-group should sum to 100%.

Say the sub-group floating-point is set to 80%, and ADD within floating-point, was set to 10%, then ADD occurs, on average, 8% of the time. If we are interested in directing our test case at floating-point, then set its group weight to 100% and all others groups to 0%. ADD now occurs 10% of the time. The more flexible a tool is towards biasing, weights and groups, the easier it is to focus towards a particular area for validation.

This gives an added advantage to the validation engineer to control the RIG to generate test case scenarios according to the requirement.

The pseudo random nature of RIG brings in the advantage of generating test cases to target the different sections of platform such as control unit, cache, RAM, memory management unit, high speed interconnects etc.

The content file is used to do just that, by using random selection of instructions, different aspects of a processor are targeted.

e.g Exercising on generating Load/store instructions validate interconnects/bus to handle data traffic.

Exercising on generating a lot of exceptions validates the ability of handling interrupts.

Exercising on generating a lot of critical sections validates the ability of synchronization capability.

4.8 Validation flow

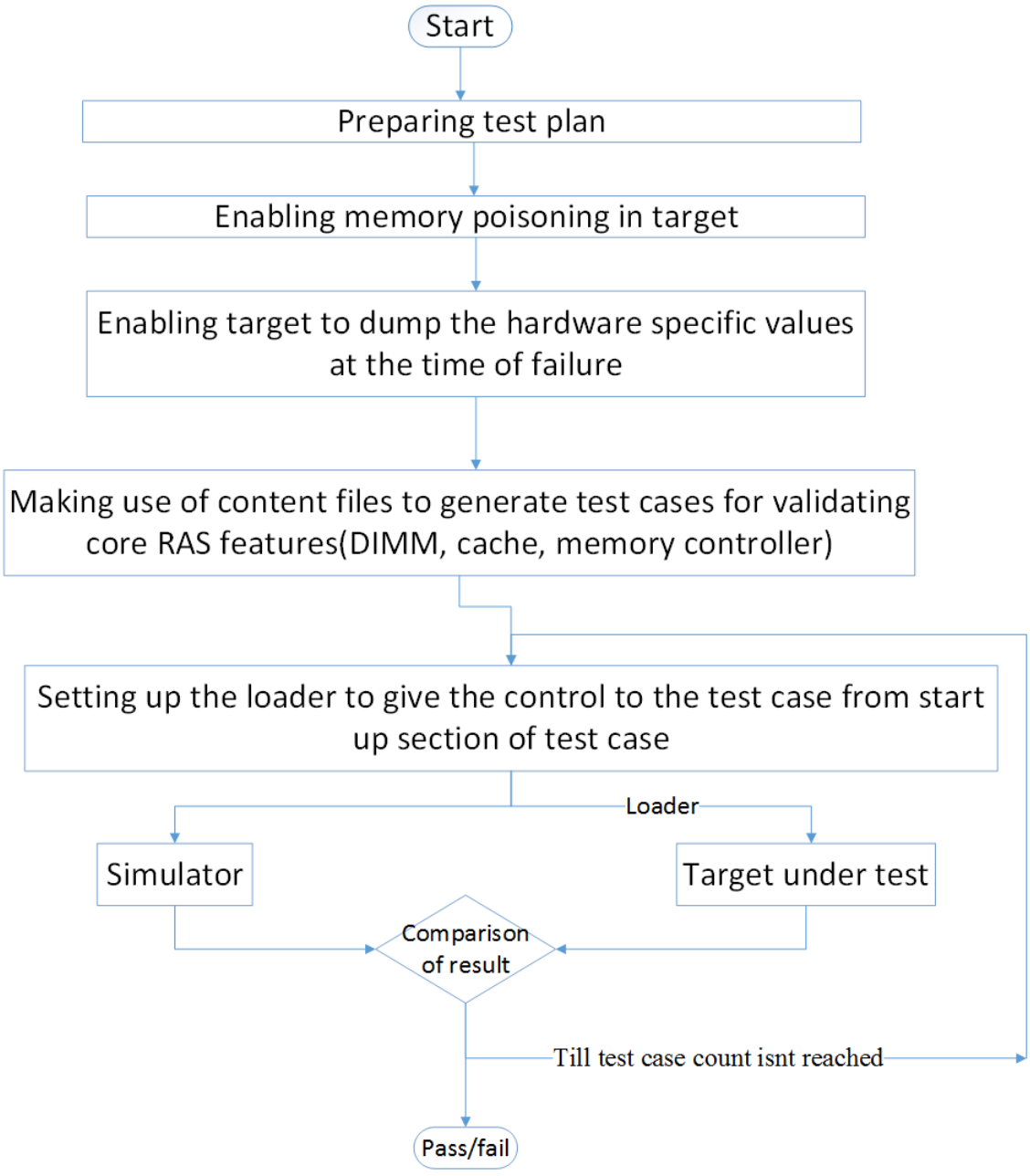


Figure 4.6: Events taking place in Validation flow

Chapter 5

SV POST SILICON VALIDATION METHODOLOGY

5.1 Introduction

Post Silicon validation approach consists of a host machine running on top of a preferred operating system (windows/linux) to interact with the target under test, programming the BIOS to initialize and check whether the core and uncore components on the platform are functional or not and facilitate running of seeds by jumping to the start of the test code to validate the MCA, once all the pre requisites have been done by the loader (i.e running BIOS, initializing the memory poisoning instructions and checking the stability of the hardware). The failures are then diagnosed, reproduced and analyzed, and a workaround is provided to remove hindrance from validation activity.

In system Validation, the prime aim is to validate the CPU and chip set (platform) in an embedded environment (using which we can control the test case and reproduce a failure) using a loader (software and hardware interface) to download the random test cases generated using RIG tool into the platform for running it to validates Silicon. Multiple hooks are provided to maintain flexibility while testing, PCI (Peripheral Component Interconnect) and external graphic cards are provided to generate test instructions to test computational capability of a processor. The test used for SV are intended to find the complications arising from the interaction of processor with the different types of peripherals, be it related to graphics, network, audio and video input devices etc.

Typical microarchitecture based specifications such as cache coherence, memory

ordering, multi-threading capabilities are pretty difficult to hit, the need of the hour is a random test tool generator which can generates random test scenarios for even the most rarest of rare complex scenarios not possible to imagine by the validation engineers.

5.2 Validation strategy for Microarchitecture focused Post Silicon Validation

Microarchitecture based validation strategy involves generation of test cases to hit critical/corner cases. This approach consists of step by step calculated and accumulated test strategy to generate precise validation vectors which are bound to hit scenarios rarely possible in a real world scenario, so that the customer doesnt face reliability issue, even in the most challenging scenario.

The validation strategy for these scenarios has evolved over a period of time, by conducting research on the type of bugs faced during pre silicon , post silicon validation and root causing them and finding the reason why they were hit, and coming up with a better validation approach each time. Identification of the correct attributes plays an important role in product readiness. The following are the best known methods for designing microarchitecture-focused tests:

5.3 Test Template Construction

With the help of validation strategy, validation test plan is laid out, which describes the different functional areas of the processor to be targeted and hence begins the construction of test templates which are used to create different test scenarios to exercise the validation plan on the chipset. Figure 8 shows the distinct phases of a test template construction. In the validation plan is, different properties under each partition are validated separately by a unique test template. The properties under each test template are used to generate a test algorithm to validate it. The biasing is varied depending upon the different stages of validation and the condition of Silicon being validated. For example, for the first stepping, the instructions causing some gating issues are dropped to continue the validation cycle. The test template is used an input for RIG tool to generate instructions to exercise the different improbable conditions. Test template fall into two categories:

1. Independent parameters

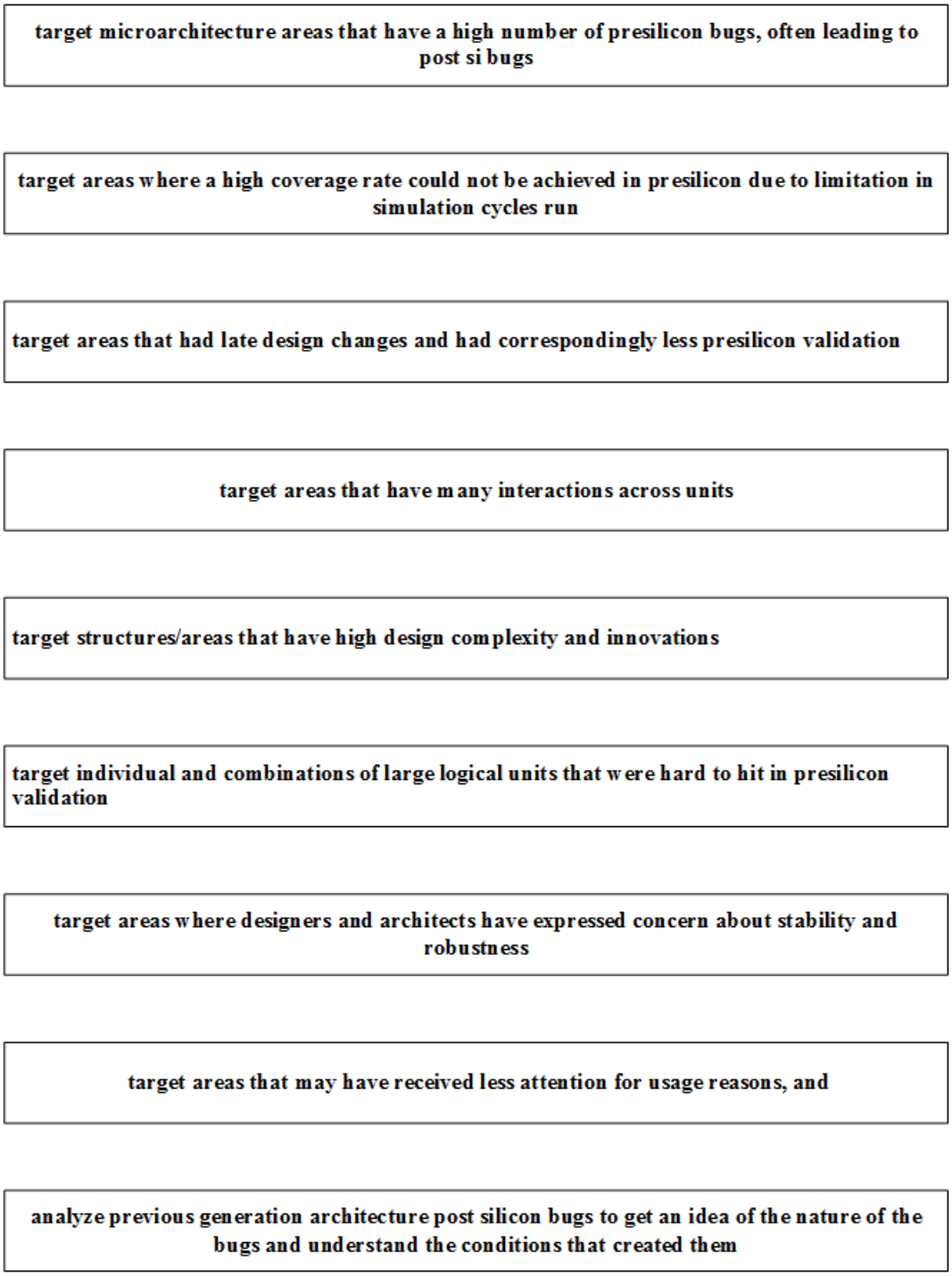


Figure 5.1: validation strategy for Micro architecture focused Post Silicon Validation

2. Dependent parameters

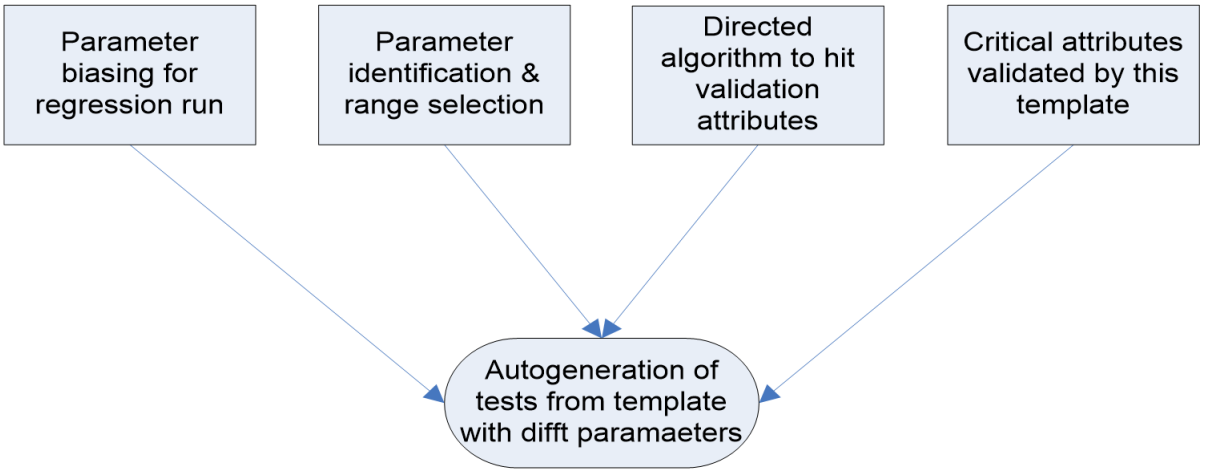


Figure 5.2: Test Template Construction

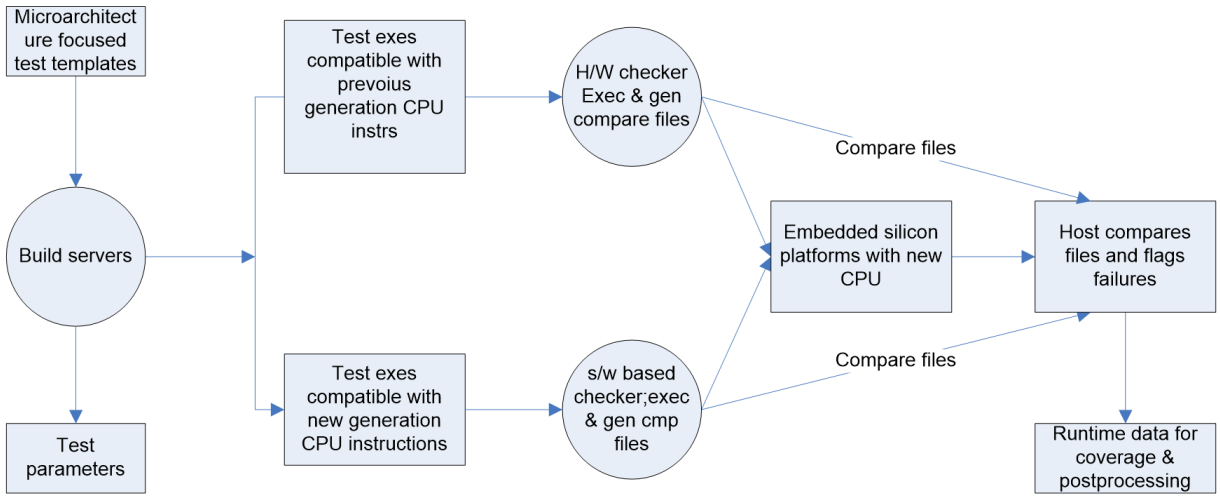


Figure 5.3: Post Silicon Test flow

5.4 Post Silicon Test Flow

Postsilicon test flow is shown in Figure 5.3. The test templates and respective parameters are used by a collection of host machines running on Intel Xeon Server Processors to generate test executable.

The executables generated by the RIG are separated into two sections. The first section contains executables compatible with previous generation processor. The second section contains executables compatible with upcoming and previous

generation processors. The first set of executables are sent to hardware reference checker, the second of the two executables are sent to a server executing a reference model of the processor to be validated. Both the ends generate compare file at the end of the test case. The compare file generated from the above set up are compared with the result generated after running the test case on the target under test. The results are generated depending upon the comparison; a seed may even fail to load, it is marked as load error; it may get stuck at a particular instruction/region, it is marked as a hung. The hardware checker provides a high throughput. The failures are then debugged using the ITP tool, used to tap into the architectural states of the processor at the time of failure. The test flow goes on continuously for a particular length of time, till the validation goals are met.

5.5 Debugging methodology used in Intel

The bugs which are hardest to find are those that result from extremely improbable events, often the product of corner cases in several different components of a complex design. This type of bug is hard to find using hand-written tests because test writers cannot guarantee that every possible interaction is exercised. Only Random testing might find this case, but each of the conditions is so improbable that finding an error that occurs at the conjunction of these cases requires a prohibitively large number of simulation/execution cycles.

Finding real bugs are very difficult in multiprocessor environment that needs debug expertise apart from integration and automation tools. We used logic analysers, In Target Probes and trace files for debugging the failure in Postsilicon environment.

For this the target platform should provide necessary debug hooks and integration tools should not interfere with the regular transaction in the bus.

All XEON family processors support an In-Target Probe (ITP) for program execution control, register/ memory/IO access and breakpoint control. This tool provides functionality commonly associated with debuggers and emulators. An ITP uses on-chip debug features of the processor to provide program execution control. Use of an ITP will not affect the high speed operations of the processor signals, ensuring the system can operate at full speed with an ITP attached.

In-Target Probe (ITP) is a debug tool which allows access to on-chip debug features via a small port on the system board called the debug port. The ITP communicates to the processor through the debug port using a combination of hardware and software. The software is an application running on a host PC. The hardware

consists of a PCI board in the host PC connected to the signals which make up the processors debug interface. Due to the nature of the ITP, the processor may be controlled without affecting any high speed signals. This ensures that the system can operate at full speed with the ITP attached. Intel uses the debug port for internal debug and system validation and recommends that all Xeon family processor-based system designs include a debug port.

Chapter 6

Debug techniques

6.1 Introduction

The silicon debug process starts after the arrival of initial silicon and continues even after the product has been given a nod for volume production. The duration of debug process lasts from months to even years, depending upon the feedback from the customers . Once the chip design is complete, the design is sent to the fabrication unit for production. The first lot of chips after being manufactured are sent back to the design team and post-silicon validation for initiating extensive debugging. Silicon debug phase is very crucial and mistakes made during this phase can be hazardous to Intel's reputation and profit. If bugs are found during Silicon debug, subsequent refinement in chip design (steppings), delay in product shipment to customers, and in worst case product recall is the only choice left. Refer Figure 6.1

During the introduction, importance of Silicon debug was highlighted so as to hit the corner cases in the least possible time. For accomplishing this task, the debug team must outline an exhaustive plan to hit critical cases in the design as early as possible to meet the organisational goal, which will benefit the higher authorities to plan out the product readiness on time, thereby holding an edge on the rival companies. The strategy used to outline the bug prone areas and devise an effective validation approach is called validation plan.

The functional validation plan is used to validate whether all the features of the product work according to the specifications given by the customer. For example, to validate an ALU would mean to test whether all the operations being performed by it are precise to the extent desired by the customer. Lets take an example of

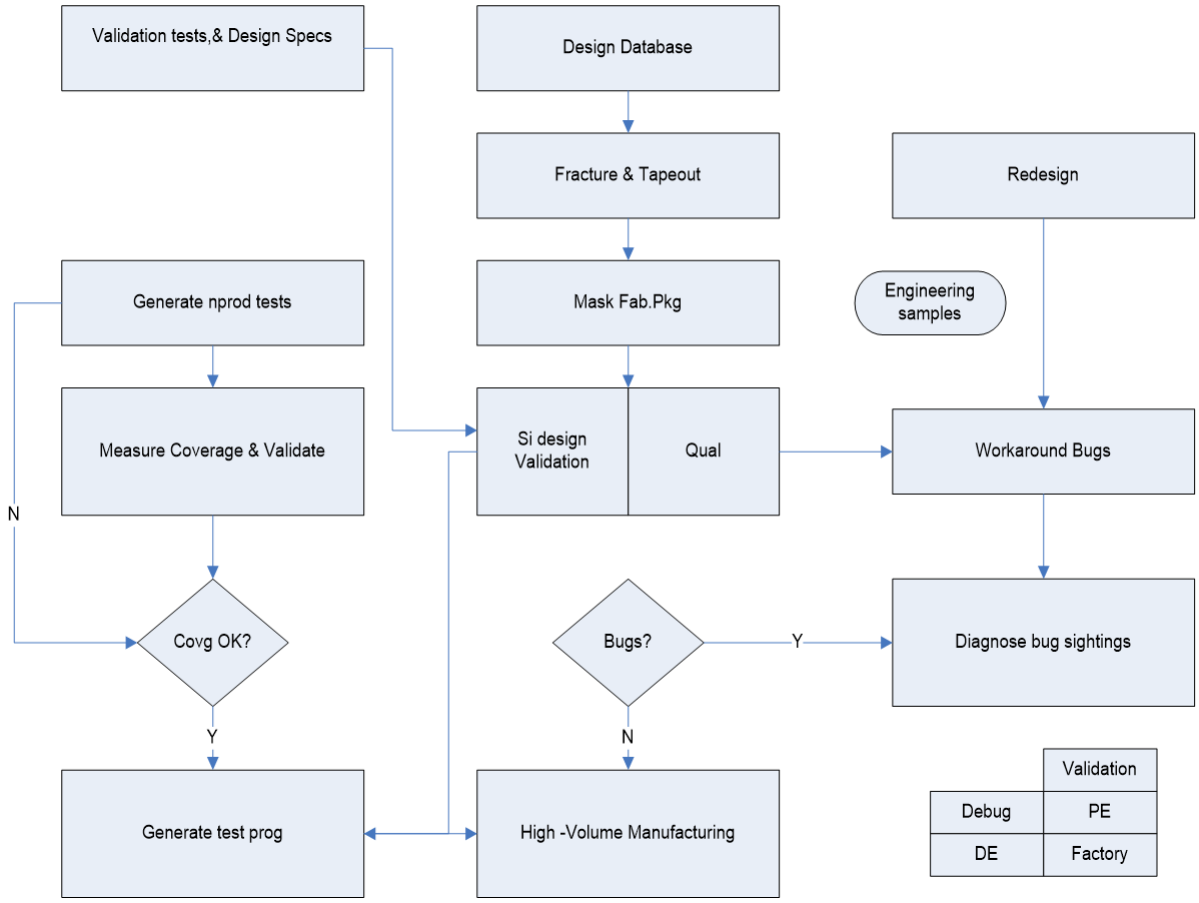


Figure 6.1: Detailed debug flow

subtracting two 32-bit numbers, it requires 2^{64} different input combinations, an impossibly large test cases to cover sequentially. Hence, a lot of planning is needed to define the instructions and data to be exercised to confirm the behaviour of a design exhaustively. Functional validation is usually done in systems where the product will be used.

6.2 TAP (Test Access Port)

TAP used at Intel is called ITP (In Target Probe), it is a debug tool allowing access to the on chip features using a debug port. ITP is used to tap into the architecture specific details of a Silicon using a combination of hardware on top of a software. In SV, the target under test is controlled by a host PC running on windows operating

system, the software part of ITP runs on the host machine (using a Xeon server processor). The hardware part of the ITP consists of a PCI board on the host PC connected to the debug port, which make up the processors debug interface. With the help of ITP, the processor can be controlled without affecting high speed signals, hence ITP does not affect a systems performance. Intel uses the debug port for internal debug and system validation and recommends that all Intel Xeon family processors include a debug port.

The primary function of an ITP is to provide a control and query interface for multiple processors. With the help of ITP, program execution is controlled and architecture specific registers of a processor along with system memory and I/O can be accessed. Thus one has the flexibility of starting and stopping the program execution using a variety of breakpoints, step through the assembly code instructions running on the target, as well as read and write registers, memory and I/O. The on-chip debug features are controlled by an application running on an Intel processor-based PC with a PCI card slot.

6.2.1 Steps followed while debugging

The steps of the debug flow are as follows [12], refer Figure 6.2:

1. Controlling the Failure
2. Isolating the Faulting Circuit
3. Root Causing the Failure
4. Expanding the Problem

1. Controlling the Failure

   This is the most important step in the debug process. For functional failures, controlling the failure involves spotting the segment of code which is the reason why the test case is hitting failure.

2. Isolating the Faulting Circuit

   Once the segment of code and conditions that caused the failure has been spotted, the next step is to spot the circuit failing on the chip. It requires extracting the internal details on the chip at the time of failure. Design team facilitates a higher level of debugging by increasing the visibility of internal state of the chip while minimally affecting the actual device operation.

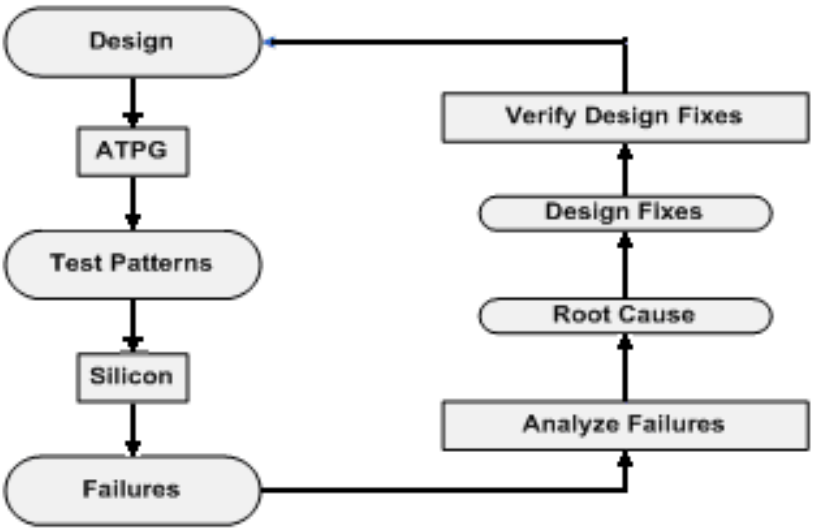


Figure 6.2: Debug Flow

3. Root Causing the Failure

   After isolating the failing circuit as far as possible using the techniques described above, the next step in the debug process is to identify what was causing the identified failing circuit, and whether anything was missed during the design process resulting in missing the bug during design phase. For some failures, isolating the failing circuit (logic) is enough to understand the cause of the failure.

4. Expanding the Problem

   Once the root cause of the problem has been found, the design team must understand why they missed this bug and is their a chance of hitting other bugs related to this faulty design and accordingly refine the pre silicon verification tools and test it further. Once all of these steps are complete, the debug engineer can consider the issue closed, and can move on to the next issue.

   So in the complete SV system, starting from RIG tool generation, seed execution, debugging the failures and disposing them in the shortest possible time is the key challenge. Efforts need to put so as to improve the methodology for capturing the critical bugs in the shortest possible time. Hence, there is need for better throughput and cross product coverage to cover all corner cases.

Chapter 7

RMCA flow and debugging RAS failure

7.1 RMCA

(Recoverable Machine Check Architecture) is a mechanism of extending the machine check architecture by making use of system software to recover from an uncorrectable error. This feature is enabled in a Random Instruction Generation Tool by following a step by step y approach as described below:

The test code is divided into 3 main sections, the set up section, the random section and the completion section. The RIG tool sets up the target, and each section is used to perform a specific task before the running the test cases, the following describes the functions of each section:

1. **Non Repeated Set up code**
   (a) Encode memory Poisoning Non Repeated Setup code
   It allows poisoned status to be synchronously transferred with data to the receiver.
   (b) Apply processor Workarounds
      i. Disable MC Exception Overflow Broadcast
      ii. Disable LLC Writeback Machine Check Exception Signalling
      iii. Disable Speculative Data Access and Prefetch
   (c) Initialize per package registers
      i. Elect a non random cafe thread from each package for memory poisoning.

ii. Unlock the memory poisoning DFX (Design For Testing) mechanism on all sockets.

iii. Memory poisoning DFX must be unlocked on all sockets.

iv. Initialize PCIe registers per package.

2. **Repeated Set up Code**

   (a) Inject machine check errors using ITP based DFT.

   (b) Enabling RMCA feature

   (c) Enabling memory poisoning feature

3. **Repeated Completion Code**

   (a) Disable RMCA

   (b) Disable Memory Poisoning

   (c) Enable Machine Check Exception Overflow broadcast (disabled in non repeated setup code)

4. **Non Repeated Completion Code**

   (a) Poison status is cleared from the memory

   (b) It will be notified to all other threads by syncing them before exiting

   (c) Only one thread is needed to clear poisoned memory

7.2 Flow for Debugging RAS failure using RIG Tool

Describe the setup needed before running tests What are the debugging files generated by the tool How the test is loaded How is a seed deemed pass/fail by satellite Snapshot of satellite How to narrow down whether a failure is due to tool, environment, part issue ? Once we have identified the failure, confirm it by reproducing, to check the frequency

Setup required before validating a test case intended to validate RAS features

- The BIOS should be programmed to enabling poisoning of memory which has to be corrupted in order to validate whether the processor is able to successfully recover form it as mentioned in the product specifications.

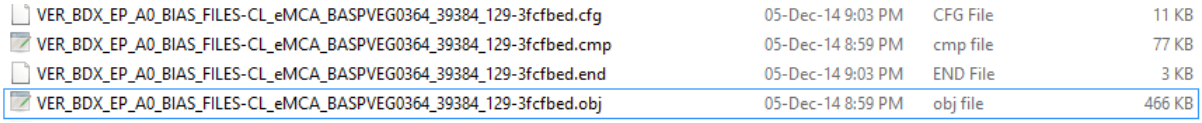


Figure 7.1: Test structure -the different input files dumped on Target Under Test

- The test cases are generated by the allotted generators which contain the platform, target, attribute, automation setup files.
- The test cases are dumped in a shared folder accessible to the loader. Figure 7.1 describes the types of files generated by the RIG.
- Once the loader is restarted, it restarts the platforms and BIOS gets loaded, BIOS is used to initialize the peripherals on the platform and checks whether each of them is functioning or not, once the BIOS performs this check it signals to the loader that it can start running scripts built on python framework to enable poisoning of main memory by using special instructions to read and write to CR after halting the threads using ITP.
- The loader recreates target initialization and platform configuration file after resetting. The loader after running the python scripts signals the test case to take control, and loads the address of start of test.
- The test code then takes control of the target and begins executing the setup code, which contains instructions for poisoning the cache memory. The test code then starts executing the random section which contains instructions generated by exercising the attributes from the input files.
  Loader GUI is shown in Figure 7.2
- On accessing a poisoned memory location, a machine check exception is signalled by the processor.
- The error is logged in machine check bank registers. Machine check handler 18 is called, the control jumps to the handler section which analyses the machine check banks, whether it is correctable or not. Register states are described in
- If the error is uncorrectable, it signals an MCERR (Machine Check Error) with poisoned tag to all other threads.

```
Test - (05/04/15 15:03:49) Found test file  C:\svshare\Run\VER_BDX_DE_V0_QA_BIAS_FILES-CL_AVX_content_BASPVEG0326_39972_126-77438a5f
Test - Load time: 176.50 milliseconds
Test - Run time: 0.07 milliseconds
Test - WaitForDone time: 3991.26 milliseconds
Test - Compare time: 0.03 milliseconds
Test - Rep   1 Cfg  1 Time 4166 ms Seed VER_BDX_DE_V0_QA_BIAS_FILES-CL_AVX_content_BASPVEG0326_39972_126-77438a5f --> Pass
Test - (05/04/15 15:03:53) Found test file  C:\svshare\Run\VER_BDX_DE_ES2_BIAS_FILES-CL_core_arch_hle_noabort_BASPVEG0326_39969_118-52ab
Test - Load time: 97.52 milliseconds
Test - Run time: 0.07 milliseconds
Test - WaitForDone time: 4257.73 milliseconds
Test - Compare time: 0.03 milliseconds
Test - Rep   1 Cfg  1 Time 4368 ms Seed VER_BDX_DE_ES2_BIAS_FILES-CL_core_arch_hle_noabort_BASPVEG0326_39969_118-52abd664 --> Pass
Test - (05/04/15 15:03:57) Found test file  C:\svshare\Run\VER_BDX_DE_ES2_BIAS_FILES-CL_vmx_pml_BASPVEG0326_39988_77-f6ecc0d
Test - Load time: 81.03 milliseconds
Test - Run time: 0.06 milliseconds
Test - WaitForDone time: 618.90 milliseconds
Test - Compare time: 0.03 milliseconds
Test - Rep   1 Cfg  1 Time 702 ms Seed VER_BDX_DE_ES2_BIAS_FILES-CL_vmx_pml_BASPVEG0326_39988_77-f6ecc0d --> Pass
Test - (05/04/15 15:03:58) Found test file  C:\svshare\Run\VER_BDX_DE_ES2_BIAS_FILES-CL_core_smep_BASPVEG0326_39974_140-41123949
Test - Load time: 66.66 milliseconds
Test - Run time: 0.06 milliseconds
Test - WaitForDone time: 3826.85 milliseconds
Test - Compare time: 0.03 milliseconds
```

Figure 7.2: Loader GUI while running a seed, the different stages of execution are shown, viz. Loading, Running, Waiting for completion and Comparing results,at the end the seed either passes or fails

- If any other thread tries to access this particular memory location, it will signal an uncorrectable error.
- After the end of test is signalled by the loader, it matches the memory location, exceptions of the simulators results with that on the hardware.
  In case of a mismatch, the relevant error code along with the debug details grepped by the error info script is dumped in a file by the loader.
- Depending on the type of error code, the debug engineers try to go behind a failure.
- Let us discuss the debug steps to be followed for an exception mismatch failure on an Intel Xeon Server Processor and how it was root caused. The parsed error information file is shown in Figure
- In case of an exception mismatch failure, we root cause why the exception was missed/ not taken by mapping the CS:EIP of the instruction causing the exception.
- The exception may have been missed because it hit another exception preceding it or the environment would not have been set up properly.
- The next step involves introspecting the environment by setting up the same scenario for checking whether the input files have the correct attributes or

not. Once it is made sure, then the hardware configuration is examined by replacing the memory modules and then recreating the failure again.

- If we are still hitting the failure, then it means the memory modules were not the culprit for the failure.
- As a next debug step, the Silicon is replaced. If the failure is reproducible again, then we are sure it has nothing to do with RTL
- Next step is to check whether we were hitting similar issue with earlier microarchitectures. This would give us an idea whether there is something wrong in the tool. If the failure is unique to this microarchitecture, then we need to check whether we hit this fail because of missing the setup before loading the test seed. This way we narrow down to the cause of a failure viz. Environment, RTL or Tool issue.
- By now it s clear that it is not a hardware failure, then the debug engineers go after the collateral set up, by verifying whether something was missed in setting up the test scenario.
- As part of the next debug step, the test case is tweaked to generate all the instructions (including poison code/data fetch) from a single thread where memory is being poisoned, to check whether we missed programming the DFT on other sockets, since it is a pre-requisite to unlock memory poisoning DFT in the repeated setup code for each socket, may be it was missed leading to the processor not pulling a machine check.
- On performing the above experiment, we found that the seed passed since one of the package (where DFX was not unlocked), on accessing a poisonous code/data instruction was not able to pull a machine check, and since the golden reference model was anticipating a machine check on accessing poisonous memory location, their was an exception mismatch.
- Workaround for this failure, while generating the test case, unlock DFX for each socket.

Chapter 8

Result

This section describes the advantages gained after enabling and validating RAS features. It describes the understanding made from reading Intel documents describing how the validation engineers design the test plan to create complex scenarios by making use of instructions targeting different operational parameters, and how the RIG tool creates a test case by setting the relevant environment, for example the number of memory blocks to be poisoned, the steps to enable memory poisoning across all the cores, setting up the interrupt handlers to service the machine check interrupts, broadcasting or receiving machine check errors across all the cores.

It describes how the debug efficiency increases by using the automation setup to manage test cases, run them on the target mounted on a platform and resolve them by using efficient failure prediction/ triaging tools.

Learned the debugging approach followed in Intel, to root cause a failure by carrying out the relevant experiments.

It describes the debugging approach using the RIG automation setup, to gain an understanding of Intel Recoverable/Enhanced Machine Check Architecture.

The major advantage of using a microarchitecture based Random Instruction Generator is that it is more efficient in spotting corner cases by running test cases on the actual hardware, which helps in catching the real time failures taking place due to timing constraints, parallel execution of instructions across multiple processors under stressful conditions.

The advantages of RAS on Intel Xeon E7 processor are discussed in the Figure 8.1.

Benefits for IT	RAS Silicon Features of the Intel Xeon Processor E7 Family
Protects Data	
• Reduces circuit-level errors • Detects data errors across the system • Limits the impact of errors	Error Correction Code (ECC)
	Memory Address Parity Protection
	Intel® QuickPath Interconnect (Intel® QPI) protocol protection via Cyclic Redundancy
	Memory Demand and Patrol Scrub
	QPI Viral Mode
	Corrupt Data Containment Mode
	Electronically Isolated Partitioning
Increases Availability	
• Heals failing connections • Supports redundancy and failover for key system components • Recovers from uncorrected data errors	Memory Thermal Throttling
	Single Device Data Correction and Enhanced DRAM Double Device Data Correction (DDDC)
	Fine Grained Memory Mirroring
	Memory Sparing
	Memory Migration
	Intel Scalable Memory Interconnect (SMI) Lane Failover
	Intel SMI Clock Failover
	Intel SMI Packet Retry
	Processor Sparing and Migration
	Socket Disable for Fault Resilient Boot
	Intel QPI Self-healing
	Intel QPI Clock Failover
	Intel QPI Packet Retry
	Machine Check Architecture (MCA) recovery
Minimizes Planned Downtime	
Helps IT • Predict failures before they happen • Maintain partitions instead of systems • Proactively replace failing components	Failed DIMM Identification
	CPU Hot Add
	Memory Hot Add
	PCIe Express Hot Plug
	Electronically Isolated Partitioning
	Corrected Machine Check Interrupt (CMCI) for Preventive Failure Analysis

Figure 8.1: Benefit of RAS on Xeon E7 server processor [1]

8.1 Bugs found while validating RAS features during this project

1. **Failure Title**: Exception Mismatch
   Failure Description: Exception Machine Check was not taken, even after accessing a poisonous memory location by an instruction.
   Failure Reason: The thread trying to access poisonous memory location was present in a different socket, where the Design For Test was not unlocked (pre requisite before running a RAS test case), due to which that socket could not interpret the machine check.
   Failure workaround: It was caused due to missing of DFT by tool, hence classified as RIG issue.

2. **Failure Reason** : Encountering Poison Data while Memory Mirroring is Enabled May Cause an Invalid Machine Check [14]
   Problem: Memory mirroring is a RAS feature which may allow the memory subsystem to survive an uncorrectable memory error. Due to this erratum, under a complex set of conditions, when mirroring and poisoning are both enabled and poison is encountered on one mirror channel, an invalid uncorrectable machine check may occur along with the expected corrected error. They will be reported as an uncorrectable Cache Hierarchy Error, IA32_MCi_Status with MCACOD = 0000_0001_0011_0100 and MSCOD = 0000_0000_0001_0000, along with a corrected Memory Controller Error, IA32_MCi_Status with MCACOD = 0000_0000_1010_cccc and MSCOD[2] = 1 in combination with IA32_MCi_Misc[32] = 1.
   Implication: Due to this erratum, a spurious uncorrectable machine check may occur.
   Workaround:A BIOS code change has been identified and may be implemented as a workaround for this erratum

Chapter 9

Conclusion and Future Scope

9.1 Conclusion

This project talked about the importance of RAS amidst the changing business needs and the important features underlining RAS. It talked about how RAS is implemented on an Intel Xeon Processor, thus detailing Machine Check Architecture of the Intel Xeon Series processors. It talked about the validation flow which is in use in Intel Server Validation Group, describing how RIG works, what are the different input files and output files of an RIG.

It listed the requirements of a platform on which the target processor is to be mounted before firing test cases to check its validity as well as the parameters which the validation engineer has to keep in mind before generating content for testing the platforms.

Cause of hardware errors, how does Intel architecture handles reporting/ correcting or flagging of a correctable/uncorrectable error were discussed.

Implementation of Intel MCA and its improvised versions to develop more efficient, robust and self healing systems.

Development of test content for validating the core components of an Intel XEON server processor, and debugging different types of machine check errors using RIG tool.

9.2 Future Scope

With the advancement in process technology and the growing business needs of clients for servers, will put lot of pressure on the time to market a particular product. Since post silicon validation cycle is an important step in a products life cycle, so improvement to cut short the debug process would save a lot of time.
The automation set up and the Random Instruction generator Tool being used today can be used more efficiently to hit corner cases earlier than what is the rate right now.
Validation engineers are always in pursuit of improving the validation cycle, so as to catch as much bugs as possible in the functioning and complex interactions of different modules with each other through the processor. The goal is to make Intel Xeon server processors 100% reliable and available.
The current technique used to define the coverage of test case (test plan), failure collection and failure analysis are mostly manual in nature, automating them to work collectively would save a lot of quality time needed for debugging failure and yielding a higher validation throughput.
The introduction of Enhanced Machine Check Architecture (EMCA) has further raised the bar for Machine Check Architecture's functioning, this technique can be further improved to make MCA more robust in handling the failures which cannot be corrected even by EMCA right now.

References

[1] "Intel Xeon Processor E7 Family:Reliability, Availability, and Serviceability" White Paper on The Intel Xeon Processor E7 Family RAS Features, 2011

[2] Intel Corporation; Kumar, Mohan; Demshki, Michael; and Shiveley, Robert. "Advanced Reliability for Intel Xeon Processor-based Servers." March 2010.

[3] "RAS features of the Mission-Critical Converged Infrastructure" Reliability, Availability, and Serviceability (RAS) features of HP Integrity Systems: Superdome 2, BL8x0c, and rx2800 i2 Converged Infrastructure, June 2010

[4] "Reliability, Availability, and Serviceability for the Always-on Enterprise" Intel Solutions White Paper/ RAS Technologies for the Enterprise, August 2005

[5] Dao, T.T, Fairchild Camera and Instrument Corporation. "SEC-DED Nonbinary Code for Fault-Tolerant Byte-Organized Memory Implemented with Quaternary Logic." IEEE Transactions on Computers v. C-30 Issue 9 (1981). Accessed May 12, 2011, http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1675864. doi:10.1109/TC.1981.1675864.

[6] Hewlett-Packard Development Company, L.P. "RAS Features of the Mission-Critical Converged Infrastructure." June 2010.

[7] IBM Corporation; Mitchell, Jim; Henderson, Daniel; Ahrens, George; and Villarreal, Julissa. "IBM Power Platform Reliability, Availability, and Serviceability (RAS)", accessed June 5, 2009.

[8] IBM Corporation; Neaga,Gregor; Buratti, Pierluigi; Kellermann, Helmut ; linkert, Peter; Labauve, Christian ; Raffel, Gordon. Continuous Availability Systems Design uide. December 1998. Accessed on May 12, 2011, http://www.redbooks.ibm.com/redbooks/pdfs/sg242085.pdf.

[9] "http://msdn.microsoft.com/en-us/library/windows/hardware/ff559382 (v=vs.85).aspx", accessed 17 Nov, 2014

[10] Intel 64 and IA-32 Architectures Software Developers Manual

[11] Hemant Rotithor,Postsilicon Validation Methodology for Microprocessors, IEEE Design & Test of Computers, Technical Journal, Oct-Dec 2000

[12] B. Gottlieb. " The crazy mixed up world of silicon debug [IC validation]", Proceedings of the IEEE 2004 Custom Integrated Circuits Conference (IEEE Cat No 04CH37571) CI CC-04, 2004

[13] Intel Internet Source, http://download.intel.com

[14] Intel Xeon Processor E5 Product Family Specification, Jan 2015