

How to Validate Machine Check Architecture complete flow at full chip level

Major Project Report

Submitted in partial fulfillment of the requirements
For the degree of

Master of Technology
In
Electronics & Communication Engineering
(VLSI Design)

By
Ketan Baladaniya
(16MECV01)



**Electronics & Communication Engineering Department
Institute of Technology**

Nirma University, Ahmedabad - 382 481

May, 2018

How to Validate Machine Check Architecture complete flow at full chip level

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

In

Electronics & Communication Engineering

(VLSI Design)

By

Ketan Baladaniya

(16MECV01)

Under the Guidance of

Internal Guide

Dr Usha Mehta

PG Coordinator (VLSI Design)

Nirma University

External Guide

Munigala, Manoj K

Pre-Si Valid/Verif Engineer

Intel Technology India Pvt Ltd.



Electronics & Communication Engineering Department

Institute of Technology

Nirma University, Ahmedabad - 382 481

May, 2018

Declaration

This is to certify that

1. 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.
2. Due acknowledgment has been made in the text to all other material used.

Ketan Baladaniya



Certificate

This is to certify that the Major Project entitled "**How to Validate Machine Check Architecture complete flow at full chip level**" submitted by **Ketan Baladaniya (16MECV01)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in VLSI Design, 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, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Dr. Usha Mehta
Internal Guide

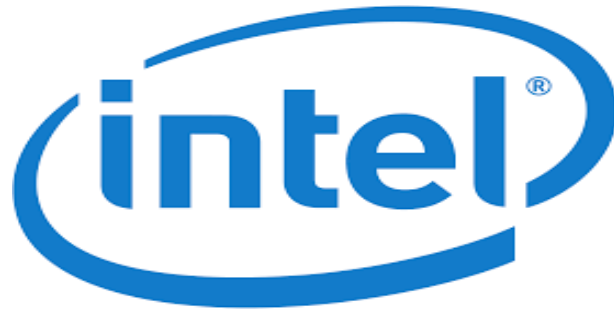
Prof. Dr. N. M. Devashrayee
PG Coordinator (VLSI Design)

Dr D. K. Kothari
Head, EC Dept.

Dr Alka Mahajan
Director, IT - NU

Date :

Place : Ahmedabad



Certificate

This is to certify that the Project entitled "**How to Validate Machine Check Architecture complete flow at full chip level**" submitted by **Ketan Baladaniya (16MECV01)**, towards the submission of the Project for requirements for the degree of Master of Technology in VLSI Design, 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.

(Manager)

Padmanabha, Raghavendra S
Engineering Manager
Intel Technology India
Bangalore

(External Guide)

N, Madhusudhan K
Pre-Si Valid/Verif Engineer

(Mentor)

Munigala, Manoj K
Pre-Si Valid/Verif Engineer

Company Seal

Intel Technology India Pvt. Ltd.(Bangalore)

Date :

Place : Bangalore

Acknowledgment

First and foremost, sincere gratitude to my manager Mr. Padmanabha, Raghavendra S and Madhusudhan K. Also I want to thank Intel Technology India Private Limited, Bangalore for assigning me such project and guide me through. I would like to express my gratitude and sincere thanks to my mentor Munigala, Manoj K, at Intel Technology India Private Limited, Bangalore for their valuable guidance during the project work, they have given me valuable advices and support which I am very lucky to benefit from.

I would also thank to my internal guide, Dr. Usha Mehta, Professor, VLSI Design, Institute of Technology, Nirma University, Ahmedabad for giving valuable the advice and support throughout the semester. I would also like to express my gratitude to all faculty members of Nirma University for providing encouragement, exchanging knowledge during my post-graduate program.

- Ketan Baladaniya (16MECV01)

Abstract

As technology node shrinking number of transistors used in Processor, and faster caches memory and buses has increased the probability of data corruption due to high frequency signal transient and permanent faults. Trend going toward Cloud computing and computing clusters makes each compute node a component of a larger infrastructure. This in turn increases possibility of runtime errors and reduces the effective mean time between machine checks in the system. That why, it is critical to provide rapid identification of error symptoms which trigger corrective actions that can prevent a major entire system or application failure that part is Machine Check Architecture.

Machine Check Architecture (MCA) is a processor internal mechanism that detects error symptoms and logs correctable, uncorrectable and catastrophic errors in the data or bus paths in each CPU core and the Northbridge side. Those errors include parity errors, ECC and TLBs error associated with system bus, caches memory and DRAM.

In this thesis we cover between simulation and emulation which best methodology use for MCA validation and why Full Chip (FC) flow is best way for MCA complete flow validation. How we can write Test cases for MCA so it give maximum coverage and using that test case we can validate MCA complete flow. At emulation level error like catastrophic generation is critical so using existing environment how we can generate that kind of error. Write checker scripts in such a way so using that we can automate MCA validation.

Table of Contents

Declaration	i
Certificate	iii
Acknowledgment	vii
Abstract	ix
List of Figures	
1 Introduction	1
1.1 Motivation	2
1.2 Objective	3
1.3 Overview of thesis	3
2 Literature Serve	5
2.1 Why are they important?	6
2.2 Potential Causes for Machine Check Exceptions	7
3 Machine Check Architecture	9
3.1 Machine Check Architecture	9
3.2 Machine Check Architecture elements	10
3.3 Machine Check MSRs	11
3.4 Error severity	14

4	MCA Validation Strategy	17
4.1	Methods for MCA validation	17
4.2	Tools for MCA validation	18
4.3	Validation scope	18
4.4	MCA flow	18
4.4.1	Tool use for writing Test Cases	20
4.4.2	Test case flow	20
4.5	Error generation:	20
4.5.1	Error generation challenges:	22
4.5.2	Generate Blue screen error:	22
5	Coverage and Self-check	23
5.1	Code coverage	23
5.2	Functional coverage	24
5.3	Assertion coverage	24
5.4	MCA CR coverage flow:	26
5.5	MCA sequence event coverage flow:	27
5.6	Self-check:	27
6	Result and Discussion	29
6.1	Result	29
6.2	Result of after test case run:	30
6.3	Regression report:	30
6.4	Code coverage report:	31
6.5	Assertion coverage report:	31
6.6	Self-check result:	31
6.7	Conclusion:	32
6.8	Future work	32
	References	33

List of Figures

3.3.1 Machine Check Architecture Global and banked registers	11
3.3.2 IA32_MCG_CAP coding [2]	12
3.3.3 IA32_MCG_STATUS coding [2]	12
3.3.4 IA32_MCi_STATUS coding [2]	13
3.3.5 Machine Check Error Classification [2]	14
3.4.1 Error classification [1]	15
4.4.1 MCA flow	19
4.4.2 test case flow chart	21
5.4.1 CR coverage flow	26
5.5.1 sequence event coverage flow	27
5.6.1 MCA self-check coding flow	28
6.2.1 Regression status	30
6.3.1 Regression report	30
6.4.1 Code coverage report	31
6.5.1 Assertion coverage report	31
6.6.1 checker script output	31

Chapter 1

Introduction

The growing number of transistors used in CPUs, and faster caches and buses has increased the probability of data corruption due to transient and permanent faults. Furthermore, the trend toward Cloud Computing and computing clusters makes each compute node a component of a larger infrastructure. This in turn increases the chances of runtime errors and reduces the mean time between machine checks in the system. Therefore, it is critical to provide rapid identification of error symptoms to enable corrective actions that can prevent a major failure of the entire system or application.

IA Machine Check Architecture is an evolving technology adding new features and enhancements with each new processor generation. The common types of errors that are detectable by the CPU include: ECC errors, cache errors, system bus errors, parity errors, etc. As processors become more integrated with new additions of memory and fast I/O, the types of errors the MCA can cover become more diverse and the feature sets get even broader. By using this architecture, the CPU can be configured to generate machine check exceptions (MCEs). Some MCEs are correctable, which means that the hardware can recover from the error and correct them without being reset.

Some MCEs are uncorrectable and the system will need to reset to recover itself. In this situation the CPU has concluded that the system is no longer in a safe or reliable operating

mode, or the cost of trying to recover from the error (either by hardware or software) is prohibitive.

Machine check architecture communicates critical hardware errors to the software as well as possibly recovering from catastrophic system failures. This architecture provides error handling features, which contribute to high processor reliability, reliable error containment and recording, serviceability, and error correction without program interruption.

1.1 Motivation

Systems like server that require high availability as a part of their Service Level Agreement (SLA) often require 24x7 uptime or 99.999% availability (also called 5-9s). This translates to about five minutes of total downtime each year. IT installations require high availability because the loss of business caused by downtime could be very expensive. These installations typically have several servers with fault tolerance, such as backup servers. An example is the Stratus* computer that has been running for 24 years, and hasn't quite yet.

One way to reduce downtime is to improve how the system handles hardware exceptions. The Intel Xeon family of processors notifies the OS about hardware exceptions by signaling Machine Check Exceptions (MCEs). MCEs are broadcast to all CPUs in the Intel processor family. This was designed to limit exposure to uncorrectable errors (UC), such as the case where the data is sent to a requester and the signaling of UC is still stuck in the pipeline. By stopping the processing of all the logical CPUs in the system, the system could be stopped more quickly preventing consumption of the corrupt data. This indicates MCA is an important feature in CPU.

1.2 Objective

The objective is to understand MCA at architecture level and MCA flow at full chip level.

1. Using MCA flow understanding write test case which useful to validate MCA flow at full chip level.
2. Find out different way of generating error at emulation level so using we increase scope of MCA validation
3. Define coverage which include all scenario.
4. Write checker script in such a way it useful for automate MCA test plan.

1.3 Overview of thesis

- In this thesis, Chapter 2 is the Literature Survey which will discuss about what kind of Machine check architecture available now and how MCA play important role in processor and what kind of Potential Causes for Machine Check Exceptions.
- In chapter-3 we cover architecture of MCA, information related MCA bank and using MCA bank how to configure MCA. What kind of error cover in MCA
- In chapter-4 we cover what kinds of methods we can use for MCA validation, which tool is more effective, what is validation scope for MCA, what kinds of flow use for MCA test cases and which tool we use for writing test cases.
- In chapter-5 we cover what kinds of coverage available for MCA validation. How to write self-check for MCA test
- In chapter-6 we cover what kind of result we get after test case compile and run.

Chapter 2

Literature Serve

There are two main kinds of machine check: machine check exceptions (MCEs) and silent machine check. A machine check exception happens when there is an error that the hardware cannot correct. It will cause the CPU to interrupt the current program and call a special exception handler.

With a silent machine check the hardware was able to correct the error, but logged the event to internal registers. There the event can be read by the operating system or the firmware later. Silent machine checks don't need immediate software or administrator action, but it is useful to log and analyze them to get early cues about hardware problems.

Then with the Intel Pentium, basic machine check handling was added to the CPU again. With the Pentium Pro Intel defined a new generic x86 machine architecture [Intel sys]. This architecture is implemented by modern x86 CPUs from Intel and AMD. It consists of a standard exception (interrupt 18) for machine checks and some standardized Machine Specific Registers (MSRs). The common registers allow software to check if a machine check occurred, to enable and disable them, check whether the error was corrected or corrupted the CPU state and some other things. In addition there are some more registers for each bank. A bank is a group of errors generated by a specific subsystem (like CPU, bus unit, cache, North Bridge). The number and meaning of banks is CPU dependent. Each bank has a number of sub-errors that can be enabled

or disabled individually.

Normally a generic machine check handler enables all errors and all banks. A machine check bank also has a register for the address associated with the error. Some CPUs like the Intel Pentium 4 also have extensions over the standard registers. The advantage of this generic architecture is that a single machine check handler can work on many different CPUs. When a machine check is detected, the kernel reads all the generic machine check registers and the registers from any banks that signaled an error.

The actual decoding and interpretation of the different errors is CPU dependent and up to the user. Some generic handling can be done; for example when a bank has a valid error address, the handler can assume that the memory at this address got corrupted. Also the handler can take different action depending on if the error was corrected or not and if the error corrupted the CPU context.

Modern Intel CPUs have special thermal errors that happen when the CPU overheats and gets throttled. This normally only needs to be logged. Some chipsets can be configured to trigger NMIs on various PCI or other bus errors.

2.1 Why are they important?

Modern hardware has internal self-checking, like internal checksums and error detecting and correcting codes for caches and busses. But the number of transistors is growing and feature size is shrinking with each chip generation, which both increases error rates. Clustering Linux machines into clusters for high performance scientific computing becomes more and more popular. In these clusters it is important to gather information about machines failing so that corrective action can be taken by the administrator. With a lot of machines the mean time between failures is significantly decreased, which means error handling becomes more important.

When a hardware error occurs on a node the task that ran on it should fail to prevent silent errors from being introduced into the computation. One way to detect these problems would be self-checks in the software (like checksums over memory buffers or algorithms with internal sanity checks for results). But this is not always possible and requires a lot of effort from the programmer. Another way is to rely on the hardware error detection. When the kernel logs an uncorrected hardware error the cluster software can take corrective action, like rerunning the task on another node and reporting the failure to the administrator.

The same issues apply on servers and high availability clusters. Logging hardware errors makes it possible to predict failures early. Even on a desktop silent errors should be avoided. It is better to tell the user that something went wrong due to a hardware issue instead of silently giving wrong results or crashing randomly. Sources of machine checks can be the CPU, PCI IO1, memory, caches, internal busses. The errors can be corrected errors (only logged to registers, no exception) or uncorrected errors (exception happens, software must react).

When PCI IO errors are enabled machine checks could be also caused by software bugs in drivers

2.2 Potential Causes for Machine Check Exceptions

1. Violations to board design guidelines. For example, routing traces over power and ground planes may cause unwanted noise and inadequate signal spacing may cause signal integrity issues.
2. Electrical supply line fluctuations can be mitigated with a high quality power supply with surge suppression capability
3. Static electricity effects can be lessened by good enclosure design and special static reducing floor covering.
4. Reduced by good thermal design of system enclosure and air-conditioning of the computer room. Hardware may also detect excess temperature and automatically switch to

mode where less power is dissipated (e.g. a lower clock speed and/or voltage, or a reduction in the number of available functional units for retiring instructions).

5. Operating the processor out of specification. Examples include overclocking of the CPU and front-side bus speeds. The behavior of the system cannot be predicted when the processor operates out of specification.
6. Environmental factors, such as: alpha particles or cosmic ray hits, extremely hot, and cold temperatures.
7. Interaction with high energy particles from cosmic ray showers in the earth's atmosphere : Reduce by shielding (locate computer room in the basement) and avoiding high altitude locations for computer systems (intensity in Denver, altitude 5280 feet, is over four times higher than at sea level locations).
8. Improperly fitted heat sinks or fans and incorrect hardware installation.
9. Missing proper microcode updates that could contain fixes for known processor errata.
10. BIOS setup issue or OS issue may cause MCE handling scheme to behave differently.
11. Faulty components, such as: add-in cards, DIMMs, etc., can also cause system errors that may eventually lead to a MCE.

Chapter 3

Machine Check Architecture

In this chapter we cover architecture of MCA, information related MCA bank and using MCA bank how to configure MCA.

3.1 Machine Check Architecture

The Intel Xeon family of processors, starting with the X65xx and X75xx generation, have supported recovery from certain uncorrectable errors (UE), specifically Patrol Scrub (PS) errors and Explicit Write Back (EWB) errors from Last level cache (LLC). These errors are referred to as uncorrected recoverable (UCR) errors; Linux attempts to recover from these UCR errors because they do not require a shutdown, as opposed to UC fatal errors. UCR errors were introduced starting with Intel Xeon processors based on Intel microarchitecture, code named Nehalem. These errors are asynchronous and are not encountered in the execution path. The SDM refers to UCR errors as Software Recoverable Action Optional (SRAO), meaning no action is required at this time, because the corrupt data is not immediately being used. Therefore it is safe for the system to continue to run. Linux added support for recovery from SRAO errors around the v2.6.32 timeframe.

Intel further enhanced MCE handling, starting with the E5-16xx, 26xx, and 46xx series of Intel Xeon family of processors, by adding recovery from certain memory errors in the execution path (also known as memory poisoning support). In cases when the CPU and platform

enable poisoning support, when UC errors are discovered, the data is tagged as poisoned and returned to the requester. When these errors are detected synchronously in the execution path, they are reported by the Instruction Fetch Unit (IFU) or Data Cache Unit (DCU). This class of errors are referred to as Software Recoverable Action Required (SRAR) to indicate that system software must perform some recovery before continuing execution. Linux added support for recovery from SRAR errors in v3.4. Although poison support guaranteed the containment of errors, the Intel family of processors continued to broadcast machine checks at this time.

3.2 Machine Check Architecture elements

Detected errors are split into corrected and uncorrected errors. Corrected errors pose no threat to the system once they have been corrected. Despite their harmless nature they, along with the uncorrected errors, are still made visible to software for tracking purposes. Some errors are harbingers of problems to come so it is important to be able to monitor them. This monitoring allows failure analysis to be done so parts that are likely to fail can be replaced proactively rather than under sudden failure conditions.

Uncorrected errors decompose into three classes: Catastrophic, Fatal and Recoverable. Both Catastrophic and Fatal cause the system to be reset in all cases. Uncorrected Recoverable errors (UCR) have the potential for the system to continue to function but not in all cases. In particular, the Software Recoverable Action Required class requires that software take action to correct the problem. If software fails to correct the problem the system must still be reset.

The machine check errors are reported via MSRs. There are three common global MSRs. The global MSRs enable/disable the error reporting in general, provide capability information (no of available banks) and keep the information if the instruction pointer is related to the error and if the machine check handling is in progress (MCIP). A machine check is the hardware's way to tell you about some internal error. Traditionally when something goes wrong in hardware the machine crashes. With a machine check, software has a chance to do something better.

3.3 Machine Check MSRs

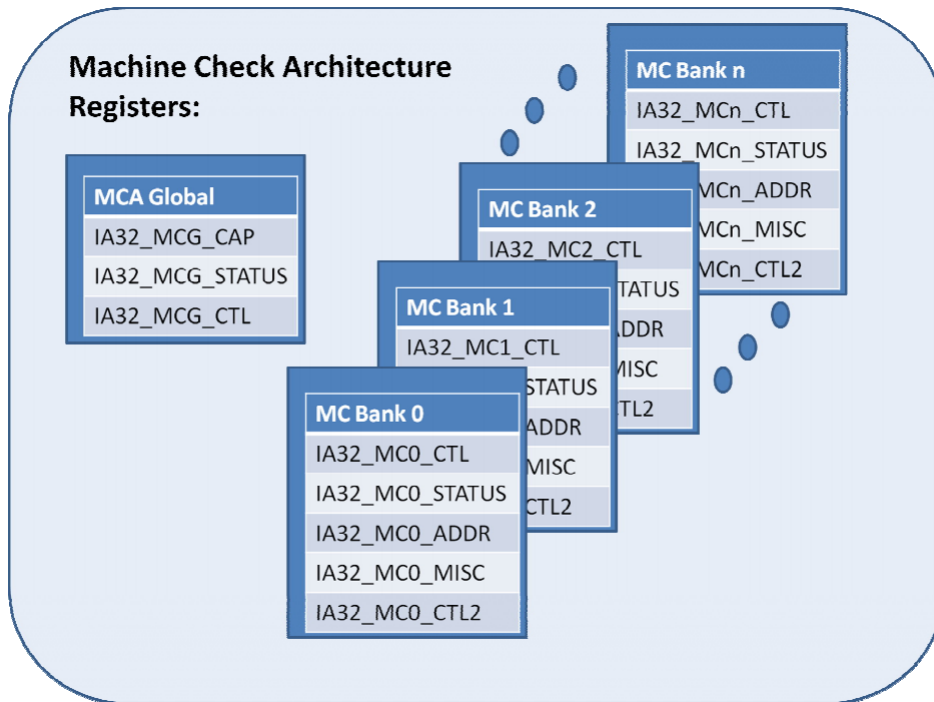


Figure 3.3.1: Machine Check Architecture Global and banked registers

Given these error classes detected by hardware, mechanisms have been defined for notifying software (BIOS, Firmware, Operating System and / or Hypervisors) of the detected errors and the actions needed to be performed. These mechanisms have two flavors, logging and signaling. Logging is done by writing information into the Machine Check (MC) registers (Figure 3.3.1, Machine Check Architecture Global and banked registers) where it can be read by system software and transferred to system logs. The MC 3 registers include a set of global registers and a number of banked sets of registers. Signaling uses a few different signals, CMCI and MCE.

CMCI (Corrected Machine Check Interrupt) is used for signaling Corrected Errors (CE) and Uncorrected Recoverable (UCR) like Uncorrected No Action (UCNA) recoverable errors. MCE (Machine Check Exception) is used for uncorrected recoverable like Software Recover-

able Action Optional (SRAO) and Software Recoverable Action Required (SRAR) as well as fatal errors that require the system be reset. In all these cases information about the error event is written to the MC Banks prior to signaling. CMCI signal is sent to only those hardware threads in the affected socket while MCE signals are sent to all hardware threads in the system.

The MCA Global Capabilities register, IA32_MCG_CAP, defines the capabilities offered by the system. Its interpretation is defined in Figure 3.3.2, IA32_MCG_CAP coding.

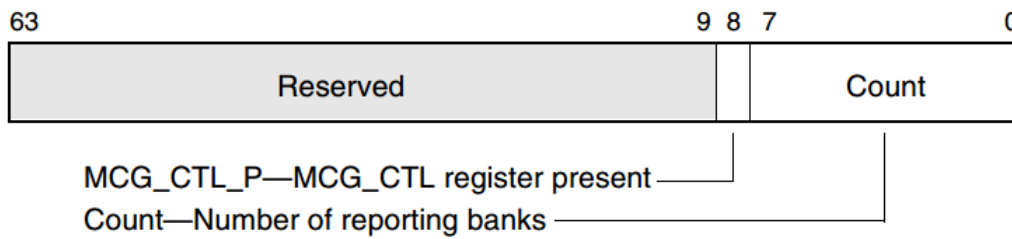


Figure 3.3.2: IA32_MCG_CAP coding [2]

As shown, bits [7:0] hold the number of banks supported by the system. Each of the fields ending with a *_P* (MCG_SER_P, MCG_TES_P, MCG_CMCI_P, MCG_EXT_P and MCG_CTL_P) indicate the existence of a feature or register and should always be checked before utilizing the resource they represent.

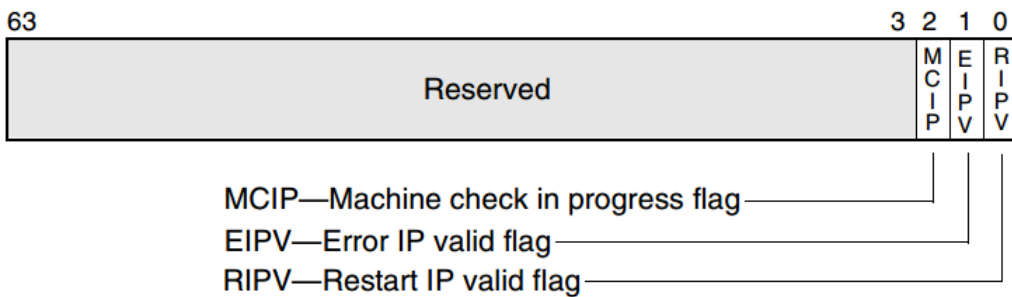


Figure 3.3.3: IA32_MCG_STATUS coding [2]

The coding for information in the global status register, IA32_MCG_STATUS, is given in Figure 3.3.3, IA32_MCG_STATUS coding. The rest of the registers are banked with each bank having its own control register, IA32_MCi_CTL, and status register, IA32_MCi_STATUS.

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

NOTES:

1. VAL=1, EN=1 for UC=1 errors; OVER=0 for UC=1 and PCC=0 errors SRAR, SRAO and UCNA errors are supported by the processor only when IA32_MCG_CAP[24] (MCG_SER_P) is set.

Figure 3.3.5: Machine Check Error Classification [2]

3.4 Error severity

Errors are divided into three categories:

1. Corrected errors are those that are repaired by hardware or by firmware. In either case an interrupt (CMCI2 for processor errors, CPEI3 for platform errors) may be raised for logging purposes. Examples of this type of error are a correctable single-bit ECC error in the processor cache or a correctable single-bit ECC error on the system bus.
2. Recoverable errors involve some loss of state. They require operating system intervention to determine whether it is possible for the system to continue operation. An example of a recoverable error is one where incorrect data is about to be passed to a processor register (e.g. from a load from memory with a multi-bit ECC error).
3. Fatal errors cannot be corrected. A system reboot is required. On this kind of error, the

processor generates a signal that is broadcast on the system bus (called BINIT) that causes the processor to discard all in-flight transactions to prevent error propagation. The error is fatal because there is no way to recover the state of the discarded bus transaction, hence the need for a system reboot. An example of a fatal error is a processor time-out (when the processor has not retired any instructions after a certain time period).

MCA is a hardware based processor centric error detection and reporting mechanism. Figure 3.4.1, Error classification, decomposes system errors into the classes of interest. The two main classes are detected and undetected errors. The undetected errors, while they may be important, are, by their very nature, not handled by the MCA.

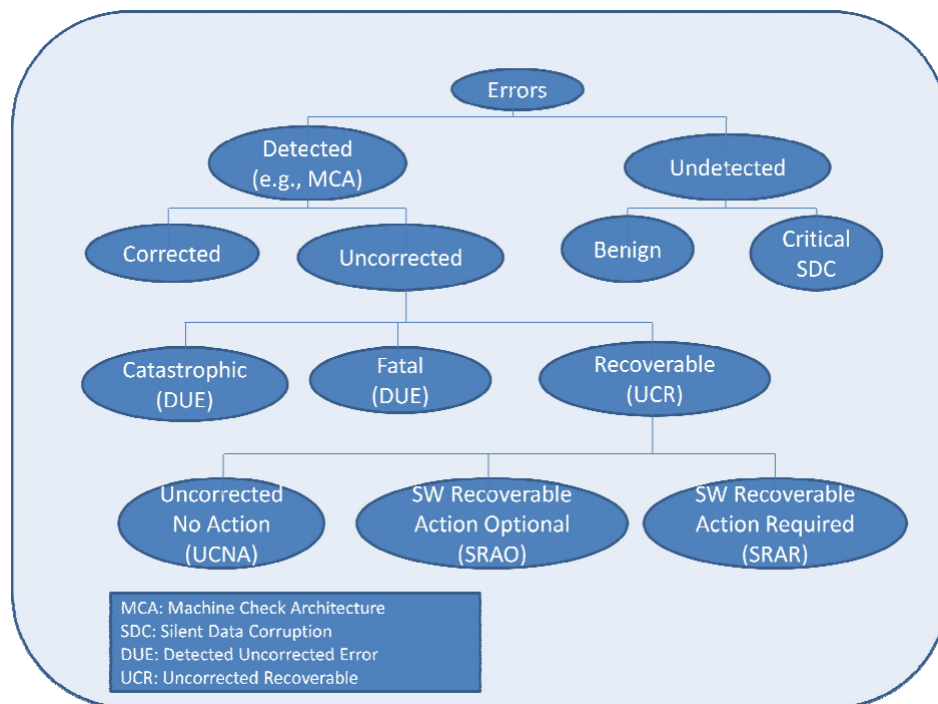


Figure 3.4.1: Error classification [1]

MCA include below types of errors.

1. **System bus errors:** In computing, a system bus error is a fault raised by hardware, notifying an operating system (OS) that a process is trying to access memory that the CPU cannot physically address: an invalid address for the address bus, hence the name. In modern use on most architectures these are much rarer than segmentation faults, which

occur primarily due to memory access violations: problems in the logical address or permissions.

2. **ECC errors:** Error-correcting code is a type of code that can detect and correct the most common kinds of internal data corruption.
3. **Parity errors:** The source of a parity error is a parity bit or check bit.
4. **Cache errors:** Get error on instruction cache read, data cache read write.
5. **TLB errors:** A translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. It is a part of the chips memory-management unit (MMU). The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache. A TLB may reside between the CPU and the CPU cache, between CPU cache and the main memory or between the different levels of the multi-level cache.

Chapter 4

MCA Validation Strategy

In this chapter we cover what kinds of methods we can use for MCA validation, which tool is more effective, what is validation scope for MCA, what kinds of flow use for MCA test cases and which tool we use for writing test cases.

4.1 Methods for MCA validation

1. SoC level MCA validation
2. Full Chip Level MCA validation

MCA we can validate in two ways one is SoC level second is Full Chip level. In SoC level validation core part was not there. Whenever error occurred like ECC, parity error that time machine check architecture update MCA banks resister, that part we validate at SoC level. But after update MCA banks core execute MCA handler or not for uncorrectable error power control unit trigger for reset or not that complete MCA flow we cannot validate at SoC level. In full chip level Core Uncore both are there. So MCA complete flow we can validate at full chip level.

4.2 Tools for MCA validation

1. Simulation
2. Emulation

As long as the DUT's size is manageable if the simulation time is in a day or less HDL software simulators are the best choice for hardware debug. They are easy to use, quick to setup, extremely fast to compile the DUT, and superbly flexible with regard to debugging a hardware design. Furthermore, they are also reasonably priced. However, they become challenging at the system level when the DUT reaches into several tens of million gates at this level it takes simulation times in days so it increase time to market.

When the DUT reaches into several tens of million gates that time for reduce time to market Emulation is good choice. So at FC level emulation is good choice to validate complete MCA flow.

4.3 Validation scope

1. At Core level only the core units will be validated for MCA, which involves Data Cache, Instruction Cache , Bus Unit Load/Store Unit
2. Uncore
3. At core level, the architectural functionality of the MCA MSRs and ucode flows will cover

4.4 MCA flow

1. Whenever uncorrectable error occur in core or uncore part that time MCA update MCA bank register and set sent MCA event to Central Event Control Unit

2. Whenever correctable error occur in core part that time it only generate CMCI for core which inside (like instruction caches) error occurs and not sent any MCA event to Central Event Control Unit
3. Depend upon event Central Event Control Unit broadcast MCA event to Core Punit
4. Core execute MC handler when it get MCA event
5. Punit trigger reset signal when it get MCA event for uncorrectable error

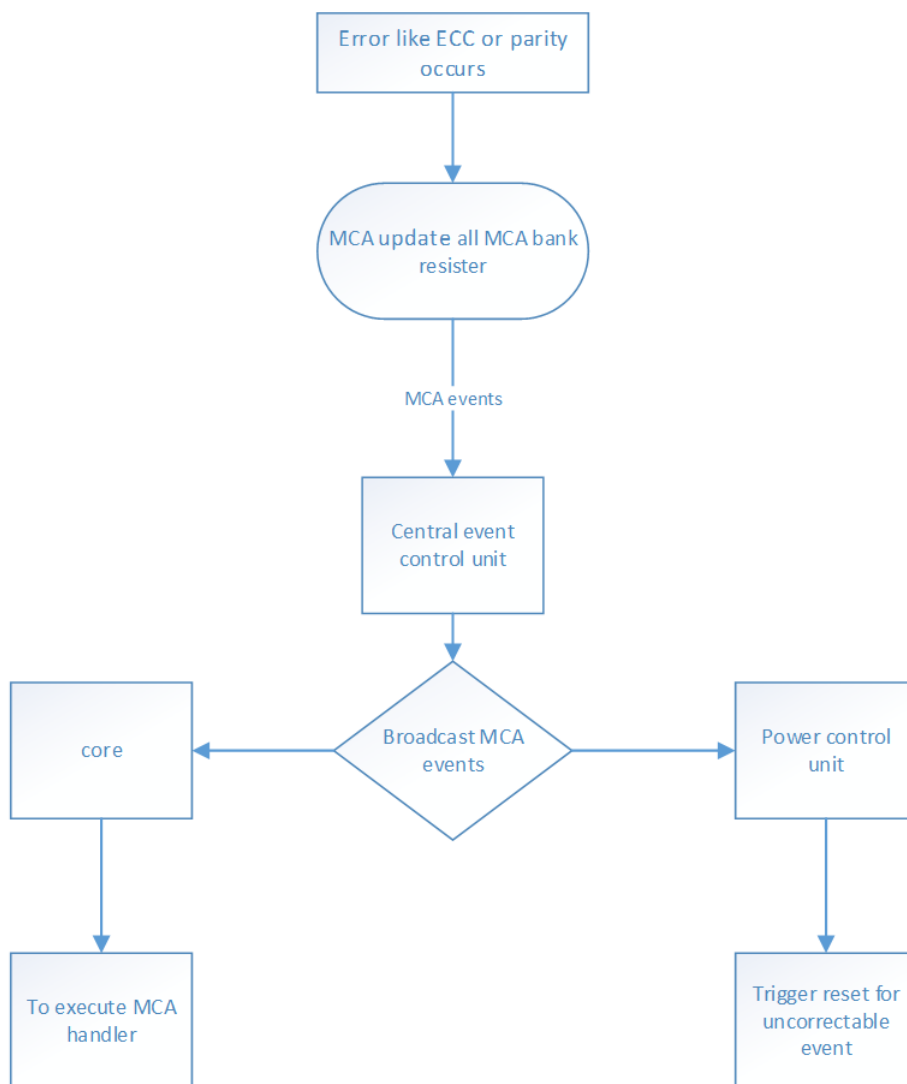


Figure 4.4.1: MCA flow

4.4.1 Tool use for writing Test Cases

We require one compiler which convert any higher level languages such like system Verilog, C, C++, perl, python ect to machine level language which we use in simulation or emulation method. Intel have own tool for write test-case its automatic generate of system tests, architectural and micro-architectural tests. Its targeted to create tests from fully random to much directed flows using a powerful constraints solving technology.

Key features of tool:

- An expressive test specification language for modular, abstract, maintainable test writing. Ability to direct tests towards interesting cases using constraints, biasing, and heuristics.
- A highly-controllable, constraint-based test generator capable of generating a full spectrum of tests, from highly random to highly directed tests.
- A declarative, maintainable, expandable model of IA32 architecture.
- Separation of modular system model from test specifications.

4.4.2 Test case flow

For validate MCA we have to manually generate MCA error in different unit. In simulation we can force signal any signal by test case at any time stamp. But in emulation we cannot force signal using test case. In emulation we can force signal using injector for that we have to write inject script. Here what is correct time to inject error signal that part is critical. So we write our test case in such a way after configure MCA it trigger some RTL signal which we polling by inject script. When RTL signal trigger that we force error signal through inject script.

4.5 Error generation:

For error generation we can use two ways

- Generate real error and due to that update MCA bank error regs

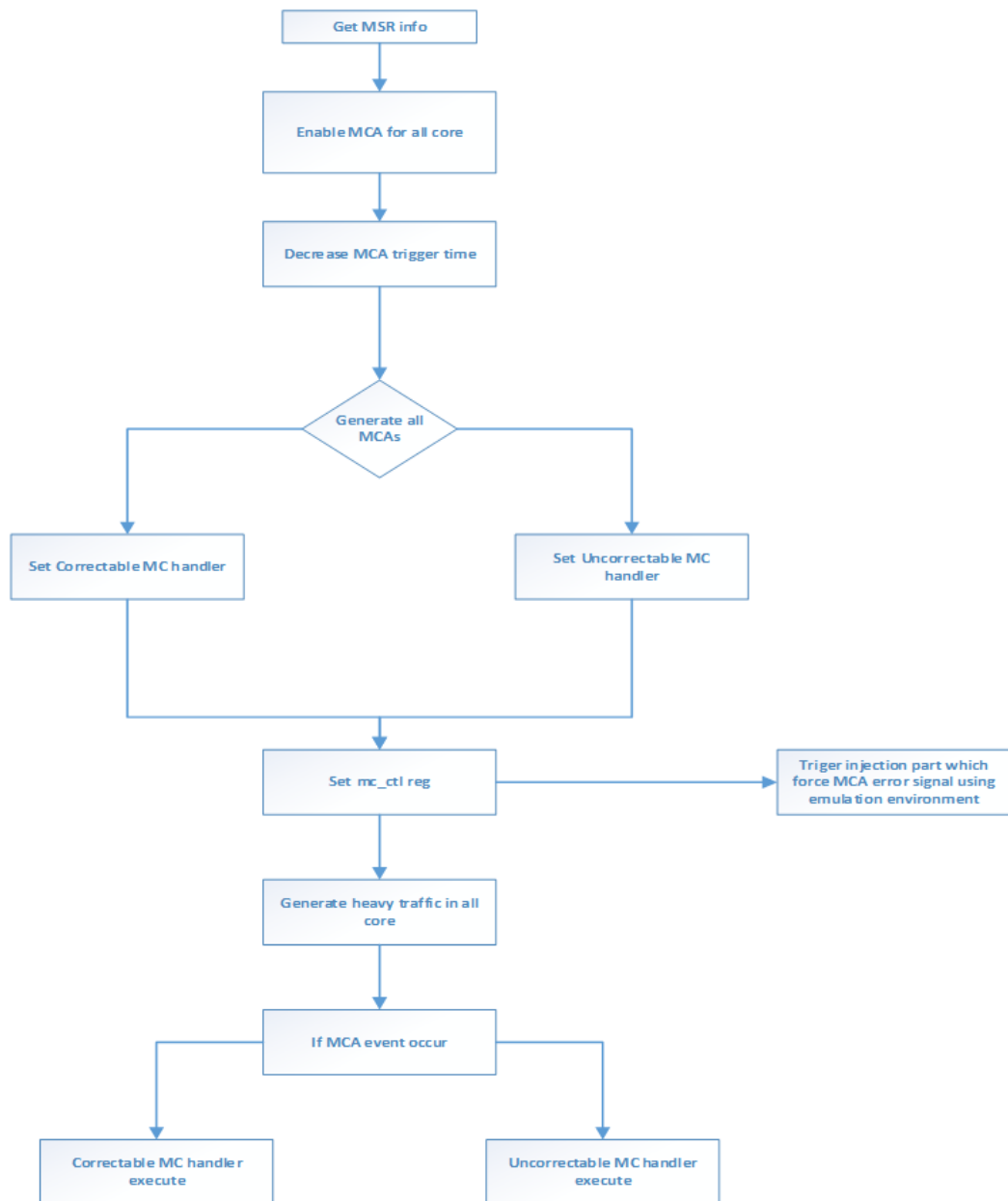


Figure 4.4.2: test case flow chart

- Update error bit in MCA bank regs using inject script

At IP level we have to validate if real error generate it update relevant MCA bank or not. But in full chip level we have to check flow of MCA not MCA logic at IP level. At full chip level we use second way of error generation.

4.5.1 Error generation challenges:

- Flow dependency that creates un-stability to the system: MCA dependent on many flow so here we can take care about generated MCA error its trigger only expected MCA flow not trigger un-intentional flow.
- Time of injection: when I have to inject MCA error signal that is very critical because if we inject before MCA configuration complete its not trigger MCA flow.

4.5.2 Generate Blue screen error:

Blue screen error like catastrophic error generation is crucial. Because this kind of error shunt down whole system before your test complete.

Solution:

Reduced retirement counter value in such a way before instruction complete it will retire. Reverted back the counter value to original value so single instruction gets retire and other instruction complete correctly so test flow continue after error generation.

Chapter 5

Coverage and Self-check

In this chapter we cover what kinds of coverage available for MCA validation. How to write self-check for MCA test

Every design verification technique requires coverage metrics to gauge progress, assess effectiveness, and help determine when the design is robust enough for tape out. At every step of the way and with every bug-finding technology and tool, verification engineers assess coverage results and make critical decisions on what to do next.

In fact, for the verification of large, complex system-on-chip (SoC) designs, coverage metrics and the responses to them guide the entire flow. The term "coverage-driven verification" describes a methodology built around coverage metrics as the primary way to manage verification.

5.1 Code coverage

Coverage-driven verification is made possible by the wide range of structural coverage information available in modern verification tools. The most traditional form, RTL code coverage, has migrated from specialized add-on tools directly into the more advanced simulators, providing much better performance and ease of use.

Once limited to line coverage, today's code coverage metrics may also include toggle, condition, path and finite-state-machine (FSM) coverage. These metrics can be gathered auto-

matically in simulation, under user control to select or exclude specific metrics or portions of the RTL.

Code coverage is very helpful at identifying "holes" in verification: if a section of code has not been exercised then it has not been verified. However, high code coverage metrics do not necessarily mean that a design is bug-free or that the verification effort is complete and thorough.

5.2 Functional coverage

Traditional Verilog and VHDL do not have any built-in notion of functional coverage points. However, both System Verilog and hardware verification languages such as Open Vera have explicit coverage constructs. These allow designers to specify corner cases based on their knowledge of the implementation. Verification engineers can specify additional functional coverage points based on their knowledge of the design requirements, especially on buses and other interfaces.

For example, functional coverage might track whether the verification process has:

- All MCA event occurs at full chip level
- Transmitted all packet types related to MCA event
- Central event control broadcast MCA event to all core or not

Knowing that such coverage points have been exercised builds confidence in the verification thoroughness.

5.3 Assertion coverage

Assertions, an essential part of modern SoC verification, can also provide valuable coverage feedback. VHDL, System Verilog, and python all have assertion constructs that allow design

and verification engineers to capture design intent.

Knowing which assertions pass in simulation and which ones fail is one form of coverage. Any assertions failing indicate that functional bugs have been found. However, successful assertions do not provide any run-time feedback information: they may have succeeded or they may not have had the opportunity to execute at all.

Assertion coverage, a metric similar to code coverage, reports which assertions were successful. Assertion coverage may also report which values within a range of acceptable values have been observed. Some advanced verification tools can extract automatic coverage points from user-specified assertions.

For example : core generate uncorrectable error and it sent MCA event to central event control unit(CEU) than after CEU has to broadcast that event to all core but if CEU doesnt broadcast to all core so this kind of scenario we find out using assert coverage.

5.4 MCA CR coverage flow:

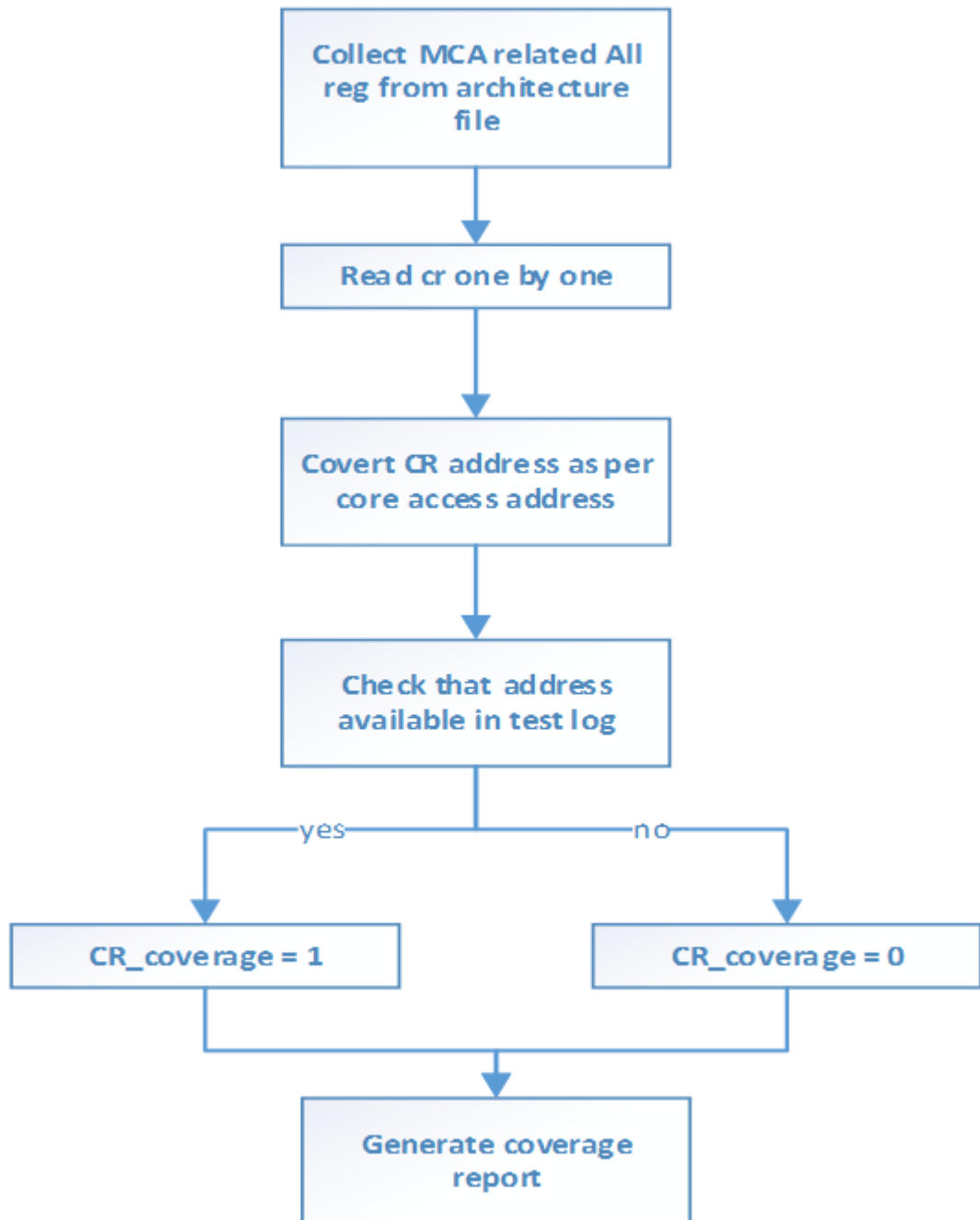


Figure 5.4.1: CR coverage flow

5.5 MCA sequence event coverage flow:

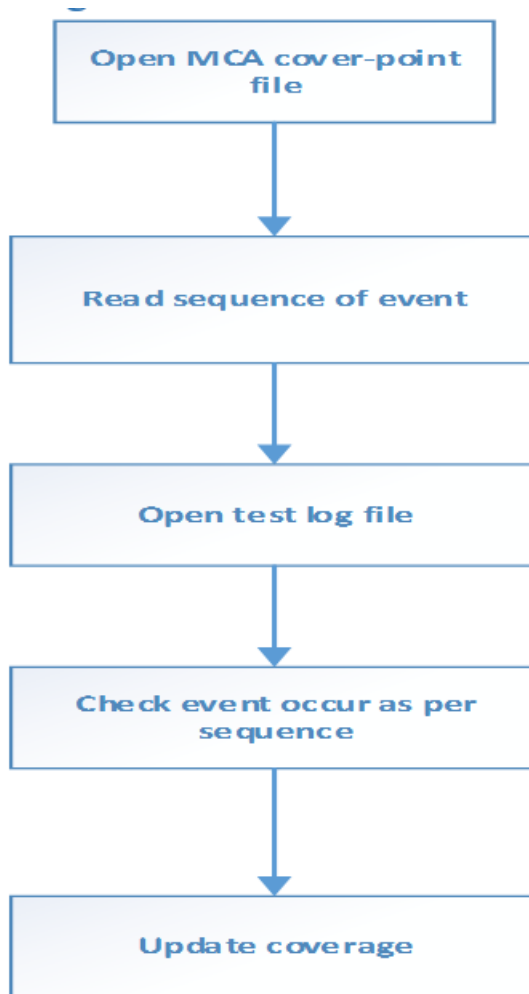


Figure 5.5.1: sequence event coverage flow

5.6 Self-check:

The self-checking mechanisms can report only failures against expectations. The more obvious the symptoms of failures are, the easier it is to verify the response of the design. The self-checking mechanisms should be selected based on the anticipated failures they are designed to catch and the symptoms they present. Using this kind of mechanism we can reduce our manual effort.

MCA self-check coding flow:

- In checker we have to cover architecture flow
- Self-check execute at run time or in post processing step
- For post processing checker we have to do deep analysis of logs
- Checker we can write using any scripting languages

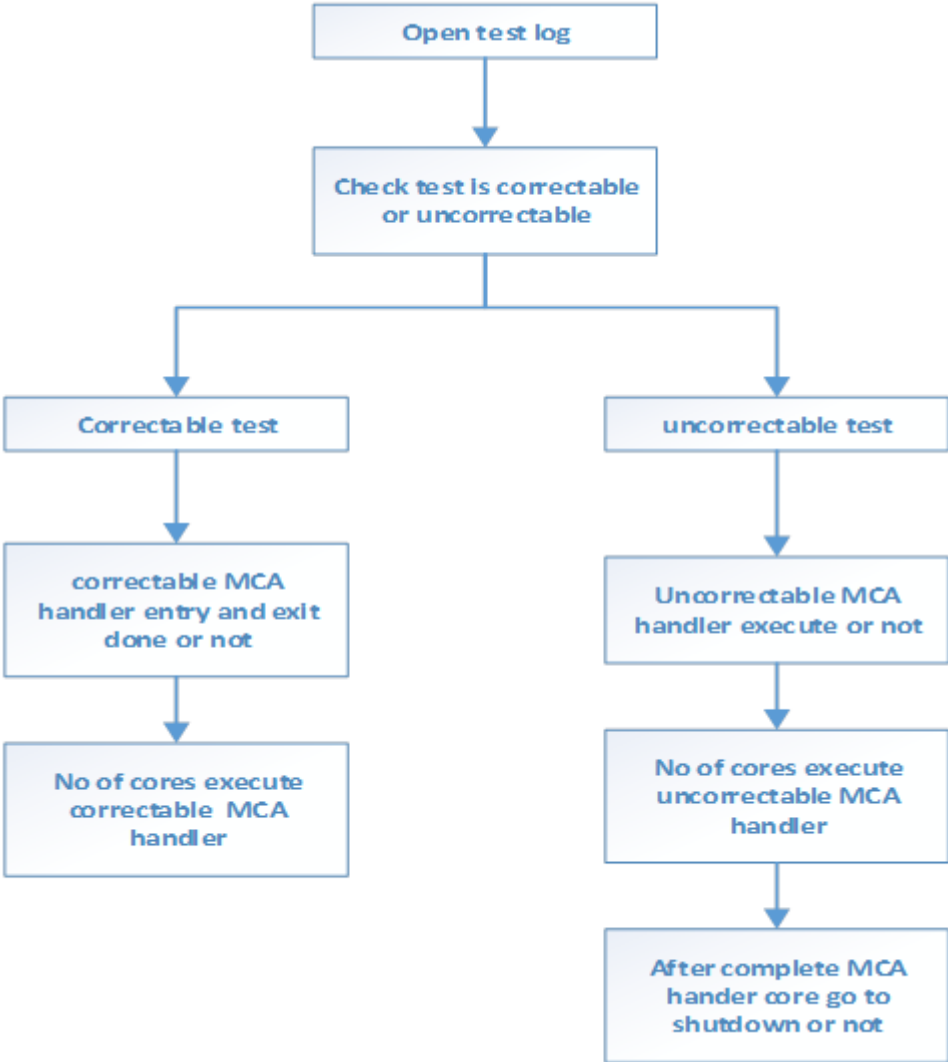


Figure 5.6.1: MCA self-check coding flow

Chapter 6

Result and Discussion

This chapter we cover what kind of result we get after test case compile and run.

6.1 Result

When we run the test it go through below stages.

- **Command line Parsing:** - In this stage we set the fuses using switches, doing that we disable or enable some features and depend upon test plan we can control how much part of test we want to run.
- **Create tests work area:** - In this stage it set environment depend upon project.
- **Test build:** - This stage compiler compile test case and convert higher level language to machine level language.
- **Model run:** - At this stage converted machine level language load in emulation board and run test case on emulation board.
- **Creating RPT:** - In this stage it create all transaction data trackers file which more use full for debug purpose, for coverage and checker.
- **Post processing:** - At this stage all post process script (like coverage and checker script) are run.

6.2 Result of after test case run:

LOGBOOK SUMMARY:

Stage	Elapsed	Errors	Warnings	Status
Init	00:00:00	0	0	PASS
Command line parsing	00:00:01	0	0	PASS
Create test's work area & preprocessing	00:16:38	0	14	PASS
Test build	00:02:17	0	0	PASS
Model run	00:18:17	0	0	PASS
Creating RPT	00:15:28	0	0	PASS
Post processing	00:00:00	0	0	PASS
End of run	00:03:32	0	0	PASS
Final end of run, Copy back	00:04:26	0	0	PASS
Exiting with exit status 0				

Figure 6.2.1: Regression status

6.3 Regression report:



Figure 6.3.1: Regression report

Report Analysis:

- Total 83 test scenario in regression
- 2 test fail due to environment issue
- One test fail in cycle limit this bucket is important because when MCA flow not trigger that time this kind of bucket we get
- 80 out of 83 test are pass

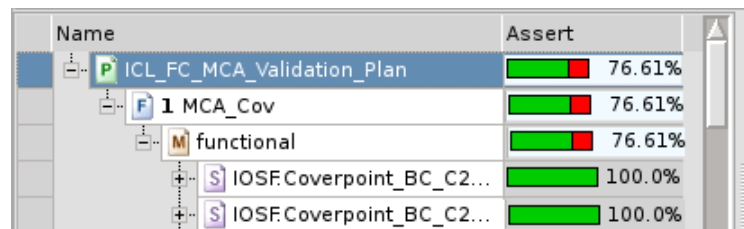
6.4 Code coverage report:

Coverage report: 89%

Module ↓	statements	missing	excluded	coverage
coverage script	91	10	0	89%
Total	91	10	0	89%

Figure 6.4.1: Code coverage report

6.5 Assertion coverage report:



Name	Assert
ICL_FC_MCA_Validation_Plan	76.61%
1 MCA_Cov	76.61%
functional	76.61%
IOSF.Coverpoint_BC_C2...	100.0%
IOSF.Coverpoint_BC_C2...	100.0%

Figure 6.5.1: Assertion coverage report

Report Analysis:

- Code coverage 89% indicate some part is redundant that we can improve
- Functional coverage 76.61% this indicate 24% of scenario not cover in regression

6.6 Self-check result:

```
FCBDC:number_of_entries_to_mca_handler = 32, number_of_exits_from_mca_handler = 309
FCBDC:Good: Number of actual threads which gets MCA (6) greater or equal to expected ones (6)
FCBDC:MCA handler happened
FCBDC:TEST PASS
```

Figure 6.6.1: checker script output

Result Analysis:

- Here actual 6 core have to execute MCA handler and is expected so test pass
- In this way every test have own self check so we cannot need to check every test is real pass or false pass

6.7 Conclusion:

By doing this we can conclude that to validate MCA sub-system is complex thing. To validate complete MCA flow at Full Chip level validation is best platform. Here we use test cases flow in such a way so using that MCA complete flow we can validate and its give maximum coverage. We find out different ways of generating correctable, uncorrectable and catastrophic errors at emulation level. At emulation level error like catastrophic generation is critical so using existing environment how we can generate that kind of error. Write checker scripts in such a way so using that we can automate MCA validation.

6.8 Future work

- To more enhance MCA test case flow and flow of machine check handler part.

References

- [1] <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>.
- [2] Intel. Intel 64 and IA-32 Architectures Software Developers Manual
- [3] Effectiveness of machine checks for error diagnostics ,Technical report ,2009 IEEE/IFIP International Conference on Dependable Systems Networks
- [4] <http://download.intel.com/support/processors/pentiummmx/sb/24318504.pdf>