"Functional Verification of Capture Memory Subsystem at Module Level"

Major Project Report

Submitted in partial fulfillment of the requirements
For the degree of

MASTER OF TECHNOLOGY
IN
ELECTRONICS & COMMUNICATION ENGINEERING
(VLSI DESIGN)

Prepared By
Soria Hiren **[07MEC017]**

Under the Guidance of

Mr.Pranav Joshi
Project Leader –ASIC
eInfochips, Ahmedabad

Prof. Usha Mehta
EC-Department
Institute of Technology

Department of Electronics & Communication Engineering,
Institute of Technology,
Nirma University of Science & Technology
AHMEDABAD-382481

CERTIFICATE

This is to certify that the M. Tech Dissertation report entitled "**Function Verification of Capture Memory Subsystem at Module Level**" submitted by **Soria Hiren (Roll no 07MEC017)** towards the partial fulfillment of the requirements for Semester III and IV of Master Of Technology (Electronics and Communication Engineering) in the field of VLSI Design of Nirma University of Science and Technology , Ahmedabad at Einfochips Pvt Ltd,Ahmedabad is the record of the work carried out by him under our supervision and guidance. The work submitted has in our opinion reached a level required for being accepted for examination. The results embodied in this Dissertation Project Work to the best of our knowledge have not been submitted to any other University or Institute for award of any degree or diploma.

Date: place:

PG (VLSI) co-ordinator
Dr.N.M.Devashrayee
Institute of Technology,
Nirma University

Head of Deaprtment
Prof.A.S.Ranade
Electronics and communication Dept
Institute of Technology,
Nirma University

Internal Project Guide:
Prof. Usha Mehta
Institute of Technology,
Nirma university

Director
Dr K.Kotecha
Institute of Technology
Nirma University.

ACKNOWLEDGEMENTS

I am grateful to **Mr. Nilesh Ranpura**, Manger, ASIC-Division Einfochips, Ahmedabad for giving me an opportunity to work in such a reputed organization.

I especially thanks to **Mr. Pranav Joshi,** Project Leader-ASIC**,** extending his support and providing us with requisite resources for training at Einfochips.

I express my gratitude to **Dr. N.M.Devashrayee**, Head of Department, M.Tech VLSI design, Nirma University, Ahmedabad and **Prof.Usha Mehta** for providing me an opportunity to take up this training and for their constant support and encouragement. After undergoing this project training, I can confidently say that this experience has not only enriched me with technical knowledge, but also with the deep insight into quality concepts that are required to be a successful professional.

I pay my special regards to **Mr. Vilash Jadav** (ASIC Engineer), **Mr. Malkesh Adesra** (ASIC Engineer) and **Mr. Manish Ladani (**ASIC Engineer) for giving me a support on this project and for their esteemed guidance and encouragement for achieving the goal.

Soria Hiren
Roll No: 07MEC017
M.Tech (VLSI Design)
Nirma University, Ahmedabad.

ABSTRACT

As gate counts and system complexity growing exponentially, engineers confront the most perplexing challenge in chip design cycle Verification. Verification of the design RTL is done at various phases of the chip design flow at different abstraction levels. The Major Project traverses through the chip design flow and functionally verifies the module of the chip. This is done at low abstraction level concentrating on the core functionality of the module. The inputs to module are forced through the testbench and its interfaces are not looked upon. This is the verification of chip at module level and is done at the RTL design phase. The designer updates the RTL as per the feedback. After the RTL is been finalized after fixing all the bugs, it is send to the fabrication unit

This project is mainly concern with the verification of the FPGA. This FPGA is used in the High Speed Memory chip tester. For verify this FPGA we used the **SCIF** (System C Interface) method. My area is mainly concern on the Pin-CMEM one of the block of FPGA. Writing the test case in C-side and V-side is providing the timing related information to the DUT. After applying the testcase DUT will generate the response as per the testcase configuration. This response is again passing through v-side and compare with the expected output and check matching the requirement.

Various test cases were developed to cover all the necessary features of the blocks. The test cases were fired and the waveforms were analyzed to debug the RTL as well as test bench issues. All those issues were filed as bugs and resolved. Once all the test cases are passing, code coverage is done using a code coverage tool. The code coverage results are analyzed to uncover any dead code as well as logic which were never exercised. Few test cases are developed to fill the coverage holes to achieve 100% code coverage.

TABLE OF CONTENTS

LIST OF FIGURES

ACRONYMS

ACRONYM	DESCRIPTION
CMEM	Capture Memory
DigCap	Digital Capture
DSSC	Digital Signal Source/Capture
DUT	Device Under Test
HRAM	History Memory.
LFVM	Large Fail Vector Memory.
LMvL	Local Move Link.
MPG	Memory Pattern Generator.
MTC	Memory Test Capture
MvL	Move Link
SRM	Subroutine Memory (a.k.a. SVM)
SR	Synchronous Reject.
STC_LFVM	Store This Cycle into LFVM (CMEM)
STV	Store This Vector.
SUT	Signal Under Test
VM	Vector Memory
SCIF	System C interface
API	Application Programme Interface
SOC	System On Chip
BFM	Bus Function Model
PCM	Program Counter Memory
ATE	Automatic Testing Equipment
VOB	Versioned Object Base
SC	System C
RTL	Register Transfer Language
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array

Chapter 1

INTRODUCTION

With increase in the system complexity, the traditional capture and simulate methodology has changed to design simulate & synthesize. The logic, functionality and gate counts in a chip are increasing tremendously. With gate count and system complexity growing exponentially, engineers confronts the most perplexed challenge in the product design: functional verification. A bulk of time consumed in design of new ICs and systems is now spent on verification. Engineers are compelled to use the best verification and design tools available to shorten design cycle time. The true path to rapid and accurate system verification includes both tool and methodology innovation *[Ref-1]*.

This project is concern with the verification of the FPGA which is used in the ATE (Automatic Testing Equipment).Here in this case ATE is used for the verifying the High Speed Memory chip. ATE contains the many Board. Below figure1.1 show the block diagram of the board. Each board contains the 5 DAGGER chip.

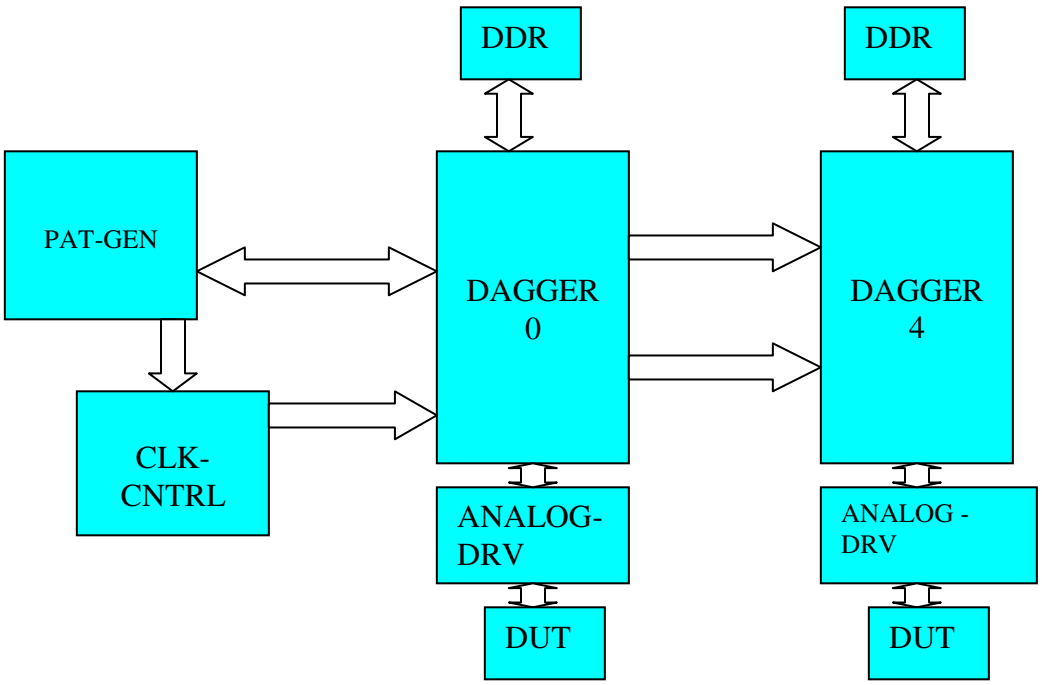

Figure 1.1- Diagram of Board

- **PAT-GEN**

Provides pattern generation control to all Dagger FPGAs through a data chain that goes in and out of each Dagger.

- **CLK-CNTRL**

A token ring provides the Local Move Link physical transport on each channel board.
The PAT-GEN and each Dagger on the channel board (5 totals) reside on the Local Move Link token ring. The PAT_GEN provide Clock control

- **ANALOG DRV**

Each Dagger provides the digital logic to drive and receive data from 2 ANALOG DRV

- **DRAM**

Ram are use to store the pattern and the intermediate results.
In the broad sense this board is lying in the instrument which is use for verifying the DUT (high speed memory chip) so mentioned above the CLK_CNTRL generate the pattern and applying to the dagger as well as PAT-GEN chip and dagger will do the necessary change as per specification of the DUT and giving to DUT via ANALOG_DRV and check the response of the of DUT.

Chapter 2

FUNDAMENTAL OF VERIFICATION

This section presents an evolution from a pure HDL-based verification methodology to a test bench automation system. Figure represents the order of evolution in verification environment.

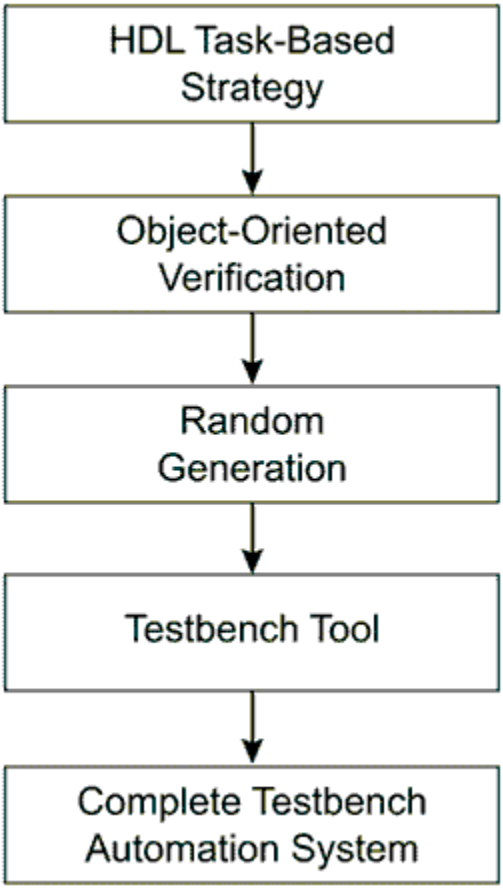

Figure 2.1 Evolution of Verification

2.1 HDL-Based Verification

With the introduction of hardware description languages (HDLs), it became common to describe both the Device Under Test (DUT) and the test environment in VHDL or Verilog. In a typical HDL test environment *[Ref-7]*:

- The Testbench of HDL procedures that wrote data to the DUT or read data from it.
- The tests, which called the testbanch procedures in sequence to apply manually selected input stimuli to the DUT and check the results, were directed towards specific features of the design.

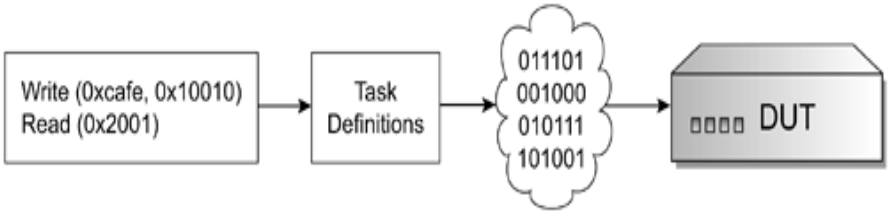

Figure 2.2 HDL-Based Verification

This approach broke down for large, complex designs because:

- The tests were tedious and time consuming to create.
- The tests were difficult to read and maintain.
- There were too many corner cases for the available labor.
- The environment became difficult to create and maintain because it used little shared code. HDLs did not provide robust features necessary to model complex environments.

2.2 Object-Oriented Verification

To make the test environment more reusable and readable, some verification engineers began to write the tests and the test environment code in an Object-Oriented Programming language like C++ *[Ref-6]*.

The test writing effort was reduced, because object-oriented programming facilitated modeling the data input and output of the DUT at a high level of abstraction. The engineer created the abstract data models and let the environment create the bit-level representations of these abstract data models.

Although the amount of time spent creating individual tests was reduced, the time spent on other verification tasks increased. The test environment became more complex because new utilities, such as a simulator interface, were required. The time required to build the test environment was substantial, often overwhelming the time saved in test creation, so the overall productivity gain was not sufficient to handle the increasing design complexity.

2.3 Random Generation

As the efforts for object-oriented verification took root, verification engineers also realized the need to reduce the effort required to create directed tests, which were tedious, time-consuming and difficult to maintain. Therefore, verification engineers began to use random generators to automatically select input stimuli. By writing a single test and running it multiple times with different seeds, an engineer could, in effect, use the environment to create multiple tests.

However, fully random generation created a lot of illegal stimuli. In order to avoid many uninteresting or redundant tests, it was necessary to build a custom generator. Creating and maintaining a custom generator proved to be a difficult challenge.

In addition, random generation introduced new requirements:

- Checking the test results became more difficult — because the input stimuli could be

different each time the test was run, explicit expected results were impossible to define before the run.

- Functional test coverage became a requirement — the engineer could not tell which tests verified which design requirements without analyzing the tests to see which input stimuli were applied.

Thus, although the test writing effort in this environment was greatly reduced, the additional work to maintain the generator meant that the overall productivity gain was not sufficient to handle the increasing design complexity.

2.4 Formal Verification

Formal verification is a precise technique that if used correctly can guarantee the correctness of the design. It mathematically analyzes the hardware design and verifies that it is functioning correctly. Typically, when a formal verification process fails because of design faults, it provides a trace that identifies the failed property. Formal verification is a powerful method for finding the most elusive bugs *[Ref-7]*.

2.4.1 Cycle Based Verification

Cycle Based Verification (CBV) language – is a new chip design methodology and formal specification language. Using the CBV language a verification engineer has the ability to specify and analyze the behavior of the intended design. The verification engineer can build a simulation model of the design and formally verify that the design satisfies the formal specifications.

The CBV language has a special feature that makes it familiar and easy to use:

- The language is structural. It includes constructs such as: “if/else” for conditional placement of statements, “begin/end” for structurally grouping statements together.

- The language is modular. CBV specifications are divided into modules, and then the modules are divided into functions, tasks and a main body. This structure allows the engineer to write CBV specifications in a bottom-up approach.

- The language has a Verilog "feel" to it. Most of the Verilog operations' syntax and semantics are supported in CBV. The CBV language also contains a sufficient number of constructs which make it expressive and capable of describing all types of chip behaviors, that are:
  - Parallel behavior - pipe-lining, multi-threaded, etc.
  - Sequential behavior – starting by reading registers, continuing by performing the calculations and finishing by writing the results.

- Timing constrained behaviors – the calculation will be complete within two clock events as of reading the registers.

A major benefit of using the CBV language for specification is that a large amount of analyzing work is done in the process. It is always the case that when a designer writes down a specification in detail, he constantly discovers new things to specify that were not included in the original specifications. Writing specifications in CBV allows the designer to specify each functional detail individually, thereby eliminating unnecessary details that belong to other functions. The designer can concentrate on the design's behavior without being distracted by implementation details, such as: resource sharing, combinatorial paths, and timing precision.

The CBV language can also be used to write CBV constraints, which specify the legal inputs to the block, and act as the virtual environment of the Verilog block. These constraints can be used both as a means to generate legal test benches and as checkers of the block's interface. The CBV constraints file contains logical Verilog expressions and specifications of state variables (used to specify the behavior of the block's inputs behavior).

2.4.2 CBV Benefits

The major benefit of the CBV methodology is its seamless flow between simulation verification and formal verification. This seamless flow is managed by the CBVM tool graphical user interface. The tool receives as its inputs CBV constraint files, CBV specification files and Verilog design files. The tool can thus interchangeably move from, or between, simulation and formal verification modes using the same inputs. Figure 2.1. Shows traditional simulation versus constrained random simulation with using CBV method.

With the traditional approach transactions are called to achieve certain "directed" scenarios, where as in the constrained random approach "declarative" constraints over state, input and output signals are provided. When running, these constraints are solved on the fly, providing constraint satisfying bit level random stimulus on the DUT inputs. These same constraints can then be used in the regular model checking tool environment as assumptions. The CBV specifications are converted to Verilog monitors in the simulation environment and are converted into the specific specification entry language of the used model checking tool.

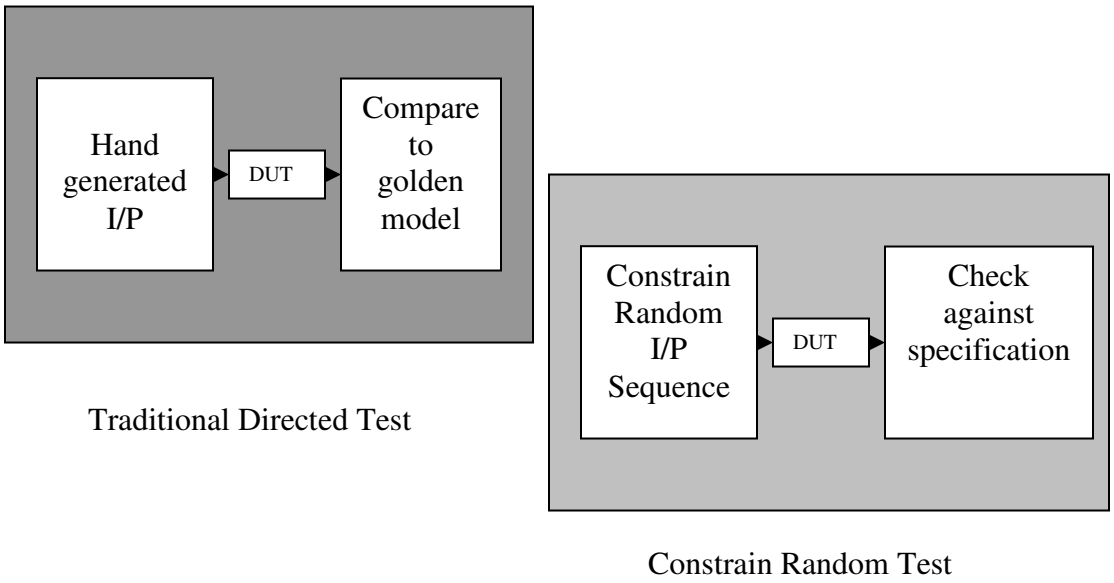

Figure 2.1 Traditional vs. Constrained Random Simulation

2.4.3 CBV Disadvantages

The main CBV disadvantage is that the results are received only at the end of verification work, when a plenty of traditional directed tests have already been written. Although CBV is an efficient method, debugging CBV code is more complicated and longer procedure than for test benches since specifications and input constraints need to be checked too.

2.5 Functional Verification

One of the first steps in design is to write down in a software-like language, typically referred to as a hardware description language (HDL), describing the logical structure and behavior of the circuit. This logical description is later compiled into circuit elements. Functional design verification aims to find problems at this early stage of the design by analyzing the description written in HDL. The behavior and the structure of designs are usually so complex that, in many cases, the design is not entirely correct and will behave differently than expected once implemented as a circuit. Since it is very expensive to fix problems after the design is fabricated, logic functional design verification proves to be indispensable by finding problems early on, before additional work is done on the design *[Ref-7]*.

2.5.1 Testbench

The HDL test bench is a program that describes simulation input using standard HDL language procedures. It is commonly implemented using VHDL or Verilog, but may also include external data files or C routines. Simply speaking, the test bench is a top-level hierarchical model which instantiates the Design Under Test (DUT) and drives it with a set of test vectors and compares the generated results with expected responses. A typical VHDL or Verilog test bench is composed of three main elements:

- Stimulus Generator; driving the DUT with certain signal conditions (correct and incorrect transactions, minimum and maximum delays, fault conditions, etc.)
- Design Under Test (DUT), representing the model undergoing verification.

- Verifier; automatically checking and reporting any errors encountered during the simulation run by comparing model responses with the expected results.

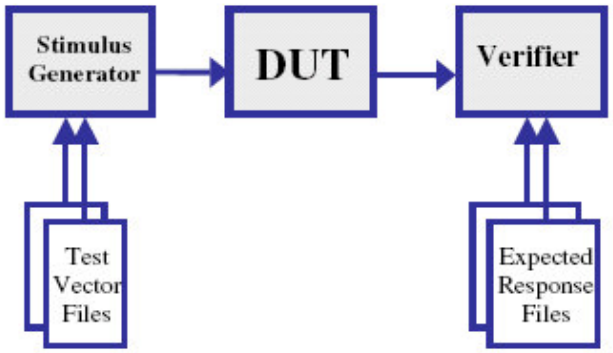

Figure 2.2 Test bench Structure

The test bench provides inputs to the design and monitors any outputs. This is a completely closed system: no inputs or outputs go in or out. The verification challenge is to determine what input pattern to supply to the design and what the expected output of a properly working design is.

2.6 Code Coverage

For a complex design, the length of code goes in thousands. The logic is also very complex and involves much functionality. So, in is impossible to know with 100% certainty that the design being verified is indeed functionally correct. Code Coverage is a tool used to find the coverage of the code by the test cases. The Coverage can be stated in many metrics. Depending on the feedback from this tool, verification engineers modify or write new test cases to cover the remaining statements or logic in code. Different metric of code coverage is calculated for all the test cases and finally the result is accumulated to get the complete picture which is analyzed further. It acts as an measure of the effort placed by verification engineer *[Ref-8]*.

It works by instrumenting the source code. This is done by adding check points at strategic locations of source code to record whether particular construct has been exercised. This method varies from tool to tool. This instrumented code is then simulated

normally using uninstrumented testbench. All results are collected in the database which generates various coverage metrics. Some popular reports are statement, path, expression, branch, toggle, fsm, etc.

2.6.1Statement Coverage

Statement coverage is the most basic form of code coverage. A statement is covered if it is executed. Note that a statement does not necessarily correspond to a line of code. This type of coverage is relatively weak in that even with 100% statement coverage there may still be serious problems in a program which could be discovered through the use of other metrics. Even so, the first time that statement coverage is used in any reasonably sized development effort it is very likely to show up some bugs.It can be quite difficult to achieve 100% statement coverage. There may be sections of code designed to deal with error conditions, or rarely occurring events such as a signal received during a certain section of code. There may also be code that should never be executed:

```
if ($param > 20)
   {
      die "This should never happen!";
   }
```

It can be useful to mark such code in some way and flag an error if it is executed. Statement coverage, or something very similar, can also be called statement execution, line, block, basic block or segment coverage.

2.6.2 Branch Coverage

This type of coverage is relatively weak in that even with 100% statement coverage there may still be serious problems in a program which could be discovered through the use of other metrics. Even so, the first time that statement coverage is used in any reasonably sized development effort it is very likely to show up some bugs. It can be quite difficult to achieve 100% statement coverage. There may be sections of code designed to deal with error conditions, or rarely occurring events such as a signal received during a certain section of code. There may also be code that should never be executed:

```
if ($param > 20)
{
   die "This should never happen!";
}
```

It can be useful to mark such code in some way and flag an error if it is executed.

2.6.3 Path Coverage

There are classes of errors which branch coverage cannot detect, such as:

```
$h = 0;
if ($x)
{--
  $h = { a => 1 };
}
if ($y)
{
  print $h->{a};
}
```

100% branch coverage can be achieved by setting ($x, $y) to (1, 1) and then to (0, 0).

But if we have (0, 1) then things go bang.

The purpose of path coverage is to ensure that all paths through the program are taken. In any reasonably sized program there will be an enormous number of paths through the program and so in practice the paths can be limited to those within a single subroutine, if the subroutine is not too big, or simply to two consecutive branches.

In the above example there are four paths which correspond to the truth table for $x and $y. To achieve 100% path coverage they must all be taken. Note that missing else count as paths.

In some cases it may be impossible to achieve 100% path coverage:

```
a if $x;
b;
c if $x;
```

50% path coverage is the best you can get here. Loops also contribute to paths, and pose their own problems which I'll ignore for now.100% path coverage implies 100% branch coverage. Path coverage and some of its close cousins are also known as predicate, basis path and LCSAJ (Linear Code Sequence And Jump) coverage.

2.6.4 Toggle Coverage

Toggle coverage measures design activity in terms of changes in signal logic values. Toggle coverage reports provide the following information:

1. Whether monitored signals were initialized.
2. Whether monitored signals experienced rising and/or falling edges.
3. The number of rising and falling edges during simulation.

Toggle coverage reports help to verify the quality of the stimulus and locate dead (=unused) structure in the design. Signals which were not initialized during simulation or are not exercised properly by the testbench can be easily identified.

2.6.5 FSM Coverage

Finite state machine (FSM) coverage answers the question, "Did I reach all of the states and traverse all possible paths through a given state machine?"

There are two types of coverage detail for FSMs that Covered can handle:

State coverage- answers the question "Were all states of an FSM hit during simulation?"

State transition coverage - answers the question "Did the FSM transition between all states (that are achievable) in simulation?"

For a design to pass full coverage, it is recommended that the FSM coverage for all finite state machines in the design to receive 100% coverage for the state coverage and 100% for all achievable state transitions. Since Covered will not determine which state transitions are achievable, it is up to the verification engineer to examine the executed state transitions to determine if 100% of possible transitions occurred.

2.6.6 Time Coverage

This isn't really code coverage at all, it's profiling of a sort, but while we're seeing what code gets exercised, why not just see how long it takes for it to be exercised? Maybe it will show up some problems with the algorithm being used, or something.

It's usually a good idea to start with the simplest metrics and move on to the more powerful ones later. It would be nice to be able to achieve 100% coverage for all the criteria, which is probably not a sensible goal for all but the smallest of projects. So what is a sensible goal? Well, it depends. It depends on the goals of project you are working on. It depends on the cost of failure. The code coverage tool will have various limitations in the coverage it performs. One would hope that statement and branch coverage would be almost universally available and consistent, but a number of coverage tools handle only statement coverage. Condition and path coverage, where available, will almost certainly not provide complete information, especially in the case of path coverage. Other coverage criteria may or may not be catered for.

Some Code Coverage tools are:

Verification Navigator: An integrated design verification environment that enables a consistent, easy-to-use and efficient verification methodology with a powerful set of best-in-class tools for managing the HDL verification process. These tools include HDL checking, coverage analysis, test suite analysis and FSM analysis. The environment includes an extensible flow manager for easy incorporation of custom verification flows. Verification Navigator supports Verilog, VHDL and mixed language designs and integrates seamlessly with all leading simulation environments.

SureCov: Engineering teams designing today's chips and semiconductor IP cores need to know, with confidence, how thoroughly the functional test suite is exercising the design. Verisity's SureCov measures FSM and code coverage with the lowest simulation overhead of any tool available, and without requiring changes to the source design. The SureSight graphical user interface shows exactly which parts of the design have been covered and which have not.

Code Coverage Tool: A freeware code coverage tool. Code coverage tool is a Verilog code coverage analysis tool that can be useful for determining how well a test suite is covering the design under test.

Chapter 3

VERIFICATION METHADOLOGY

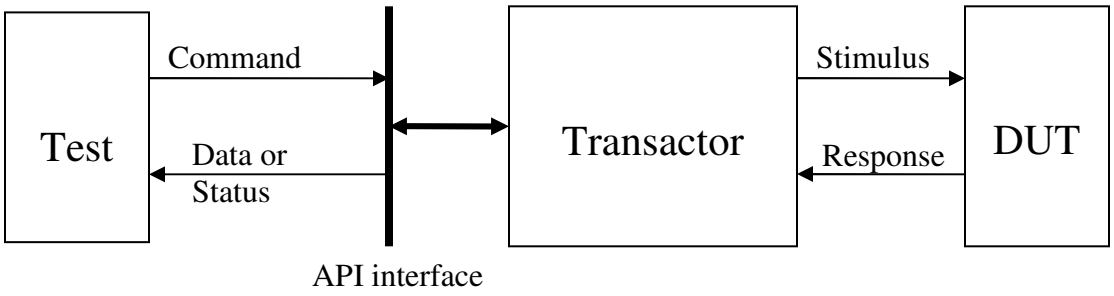

Figure 3.1 Transactor Overview

Transactor *[Ref-3]* is a module which communicates between the test case and the Device Under Test (DUT).Refer Figure-3.1. The transactions between the test case and the DUT are carried out through the Transactors. In general, a transaction is a data transfer such as a Read or a Write operation.

The test cases use the API calls to command the Transactor to do a transaction. The Transactors transform a command from the test cases into stimuli of signal values and these stimuli are sent to the DUT. For a given stimulus the response from the DUT is processed in the Transactor and the status or the data is sent back to the test case.

The Transactors can be of two types

1. **Only with C-side**
2. **With C-side and V-side**

An example for the first type is ‘Expect Models’ used for checkers, to generate expected values of the DUT responses. A typical SCIF Transactor is of second type.

3.1Transactors with Only C-Side

The Transactors with only the C-side *[Ref-3]* act as expect models that are used to calculate the expected values of the responses. These values are used in checkers to compare with the actual values.

Transactors with both the C-side and V-side *[Ref-3]* are explained in detail in the following sections.

3.2 Transactor Detailed Description

A Transactor may be used as 'Receive only Transactor' to do monitoring actions, or as 'transmit only Transactor' to give continuous stimuli. A Transactor may also be used to receive and transmit. The typical SCIF Transactors are used for both reception and transmission.

The Transactor consists of C-side and V-side.

The C-side of the Transactor contains the following

- **A Class** derived from the Base class or the Base class itself
- **A state machine** in Base class or the modified state machine version in the derived class
- **A SystemC interface**

The V-side of the Transactor contains the following

- **C-side interface** which also includes a instance (cside_interface.v)
- **Verilog tasks** which act as counterparts for C-side member functions

The dataflow in a typical SCIF Transactor can be explained using the following figure 3.2. The test cases use the member functions of the derived class or the BaseT class and the API routines. The BaseT instance calls the routine to add commands in the command queue. These commands are read by the state machine thread and these commands are executed.

In a Write command, the necessary data after processing the command are placed in the array tranDataIn. The GetData thread in calls the GetData task to send the data from tranDataIn to DUT. In a Read command, the Getdata is used to convey the read action. The necessary data is placed in the array tranDataOut, by the PutData thread. The PutData thread uses the PutData task to receive the data from DUT. A Verilog module (Transactor V-side) contains the tasks GetData and PutData and it also interacts with the DUT *[Ref-3]*.

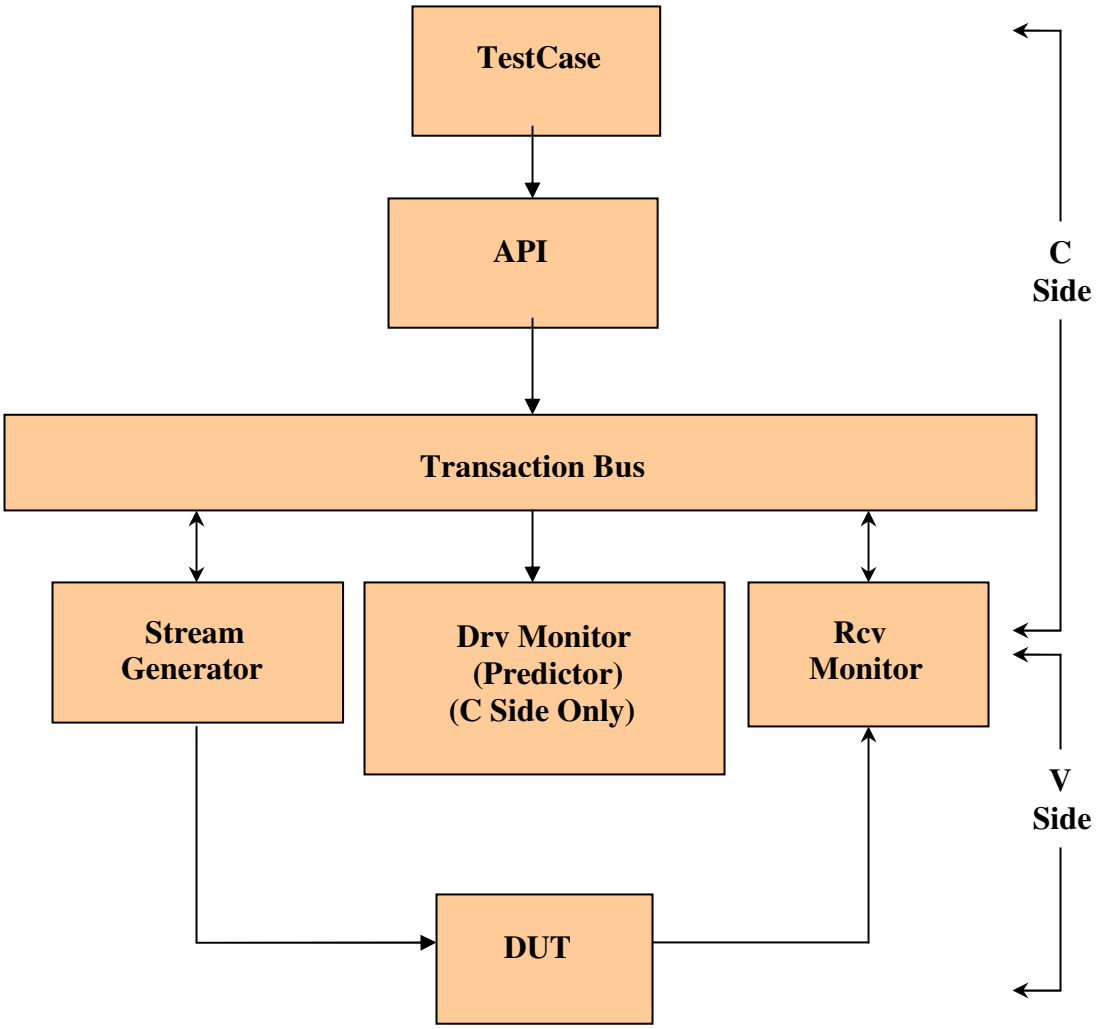

Figure 3.1 Transactor Data Flow

Chapter 4

DAGGER FPGA

Here DAGGER is the FPGA chip name which is mounted on the board of ATE instrument. DAGGER FPGA chip contain the many block like, *[Ref-1]* Drive Timing Generator,Receive Timing Generator , Memory Pattern Generator ,Local Move Link,Program Counter Memory & Source Select, DRAM, Capture Memory (CMEM) ,Fail Processor ,Central Data Bus, Mission Context ,DDR,Vector Memory Client, Download client as shown in Fig 4.1

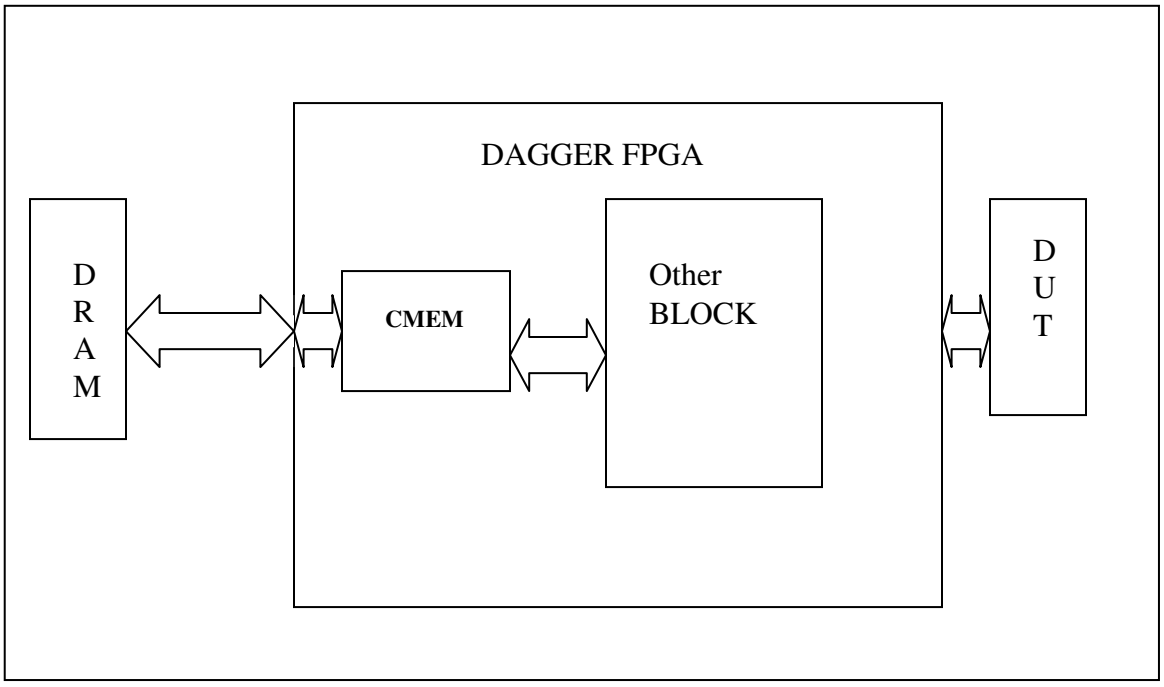

Figure 4.1 Block Diagram of Dagger FPGA

- **MEMORY PATERN GENERATOR**

The Dagger Memory Pattern Generator algorithmically generates address and data used to stimulate and verify memories under test. It supports four unique functions, which can be fanned out to channels.

- **TIMING GENERATOR**

TG is part of the Dagger FPGA. It implements elements of the Timing subsystem for the instrument. The TG consists of a drive path which takes 3-bits of pin data from the PCM (program counter memory) and formats it into Drive and DrvVt data streams to send to the ANALOG_DRV pin electronics. TG also consists of a receive path that latches comparator data from the ANALOG_DRV, samples it and formats it for the Fail Processor.

- **FAILURE PROCESSOR**

Any channel that cares to determine the DUT pin state will use the Fail processor circuits. Timing Generator circuits strobe the DUT comparator per pin and send comparator data to the fail processor. The fail processor will make a per user cycle comparison against what the user expected. If unexpected data is found then an SR is generated. The SR Return can be used to control the pattern and/or data storage. The SR can be latched or counted using CR_EN. The SR or Fail and most of the Fail processor data can be routed for storage in RAM and CMEM.

Pin fail processor is used to process compare data captured at the DUT during a pattern burst. Pin data represents what the user expects to see at the DUT.

- **DOWNLOAD CLIENT**

The Receive TG operates on channel comparator data from two ANALOG_DRVs. Together with the ANALOG_DRVs, the Receive TG provides receive edge placement control, The Rcv TG processes the receive data and forwards it to the Fail Processor block, which compares it to expect data during receive cycles.

- **LOCAL MOVE LINK**

It takes the data stored in DRAM by the CMEM. This data contains information about the test failures that were detected by the ATE. LMVL takes this data and gives to an external DSP host processor for further processing and analysis

.

- **PROGRAM COUNTER MEMORY**

The Dagger PCM supports a sub-routine memory (SRM) and a VM (vector memory) FIFO which is periodically updated external DRAMs through the Client. This 3-bit data can then be serialized and function-stacked to produce scan vectors. The PCM output, also known as vector data, addresses the SS memory holding 3-bits of pin data which represent possible pin states. The SS memory is addressed by several other data sources including alternate data source data from the memory patgen source select information from the pattern generator and scan.

Chapter 5

CAPTURE MEMORY (CMEM)

The CMEM logic will be divided into large functional blocks to implement the requirements *[Ref-1]*. There will be a Common Client Interface block which will handle all the logic which is common to all CMEM clients. There will be a Central CMEM block which will handle the capture of the MPG data. The Pin CMEM block will handle any capture of the channel information which will respond in all CMEM modes.

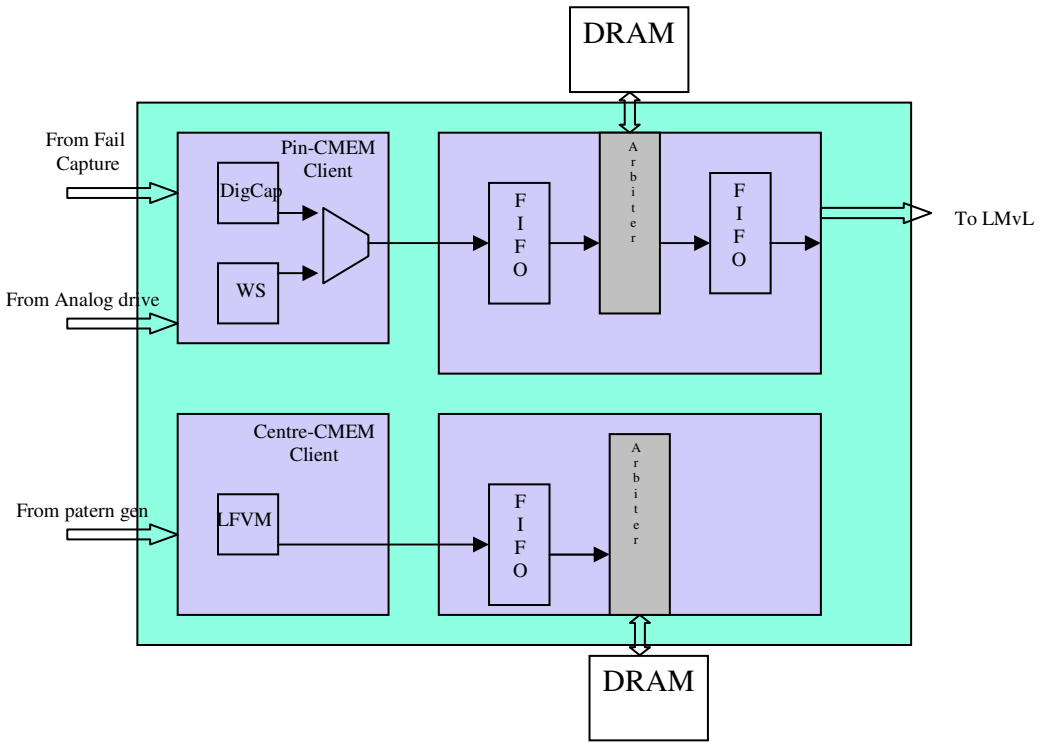

Figure 5.1 Block Diagram of CMEM

The shaded portions in the above diagram are described herewith in. The top half of the diagram is what we have been terming **Pin CMEM** and Central CMEM is the bottom half. The shaded blocks on the left are the clients. The blocks on the right are the common logic blocks to all clients (or the Common Client Interface). The goal was to reuse much of the CCI between the two types of CMEM. It has been designed so it could be used for future designs.

5.1Common Client Interface (CCI) Design

The purpose of this block is to provide a common interface to all CMEM capture clients. The inputs to this block contain all the information that is required to create capture segments for capture. The inputs to this block will contain all necessary information to create move packets. This serves to decouple critical capture from the move activity. The CCI will interface to the DDR2 SDRAM controller/Arbiter, Capture Clients and the Local Move Link Client. This is structured such that the data width is eight times that of the *used* physical memory width.Components of the block will be used for both Pin CMEM and Central CMEM *[Ref-1]*.

5.2 Pin –CMEM

The Pin CMEM is normally involved in all captures. There are distinct differences between the three, but the data source for MTC, LFVM and DigCap is the same. This data source is either 1) any bit of Pin data, 2) DUT data or 3) SR which is selected in the "fail capture block" using the register. The select can be done uniquely "per channel"; however, it is commonly set to the same value across the engine *[Ref-1]*.

The below picture shows the logic on the client side of the Pin CMEM. There are three distinct clients which have subtle differences.

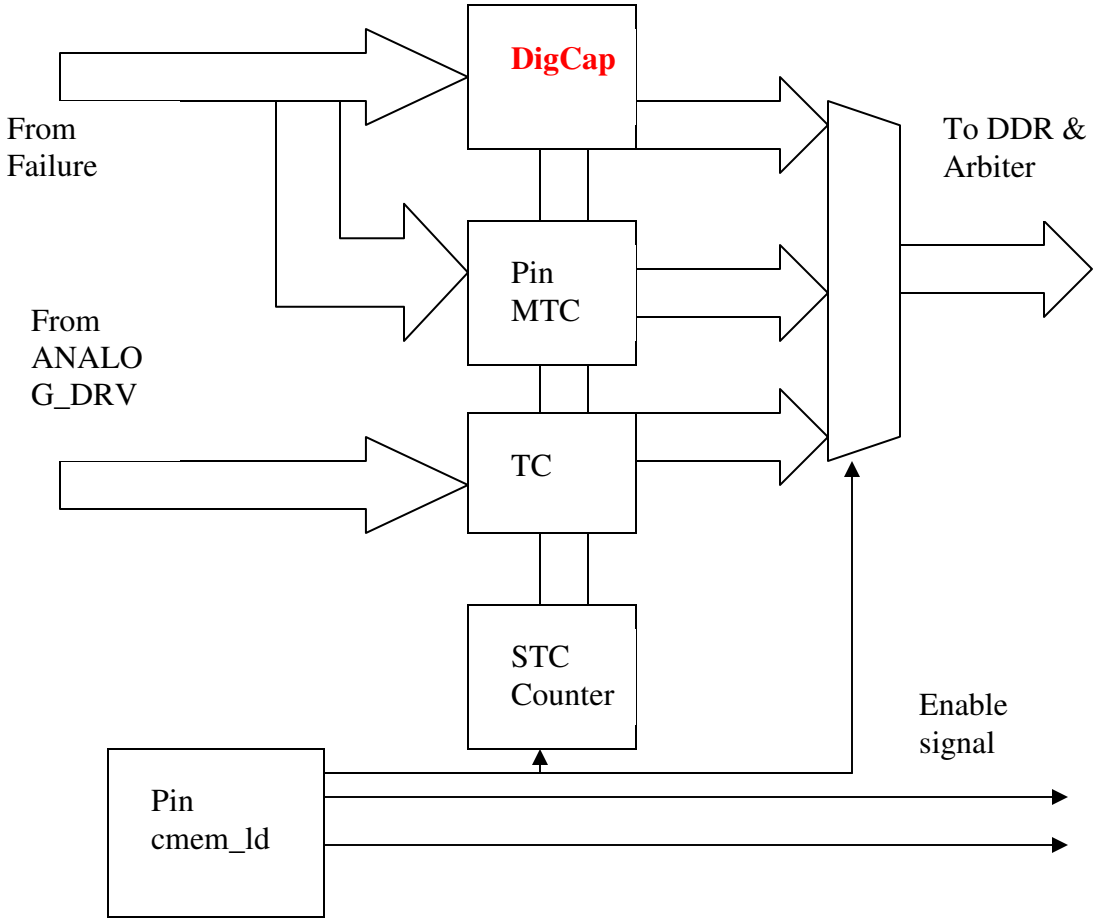

Figure 5.2 Block diagram of Pin CMEM Clients

5.2.1Memory Test Capture (MTC) / LFVM Design

The Memory Test Capture (MTC) Pin Client and the LFVM Pin Client will store data based on the signal which originates from the PatGen. This signal will indicate to the Pin CMEM to store. This is a major cycle opcode, thus data may only be stored on a major vector basis. [*Ref-1*]

5.2.2 Timing capture Capture

Timing capture is used for time measure. Timing capture Capture will only support measurements that are "within" a pin. Jitter, rise time, fall time are examples of such measurments. Timing capture is based on an undersampling technique. This creates a unique problem in that the capture data is now asynchonous to the pattern. The trigger for the timing capture still needs to be done from the pattern. The data for the timing capture is synchronized back to the pattern clock domain for capture. This will be done with an enable and an asynchronous FIFO *[Ref-1].*

The Dagger/ANALOG_DRV interface share the funtional and timing capture path. This means that a functional test and a timing capture capture test may not occur at the same time. To switch between timing capture and functional mode, the register in the ANALOG_DRV must be written. This register may be written at pattern stop or keep alive. The implications of this are that to switch from a functional pattern to a timing capture capture, the user must first go into keep alive to switch Dagger/ANALOG_DRV into timing capture mode. The same holds true for going from timing capture capture to functional test.

The Timing capture Capture is different from MTC/LFVM and DigCap in that it has an additional step of indirection. There is a sampler memory in the WS Cmem client which controls which data should be captured and the explicit close of a segment

5.2.3 DigCap

- Starting a Segment

An MTC or LFVM segment may only be started implicitly. This means that the first STC segment which is received from the CLK will start the segment. When the segment is triggered, the fields used to define the segment are written to the Segment History Memory which is done by asserting the signal to the *CCI [Ref-3].*

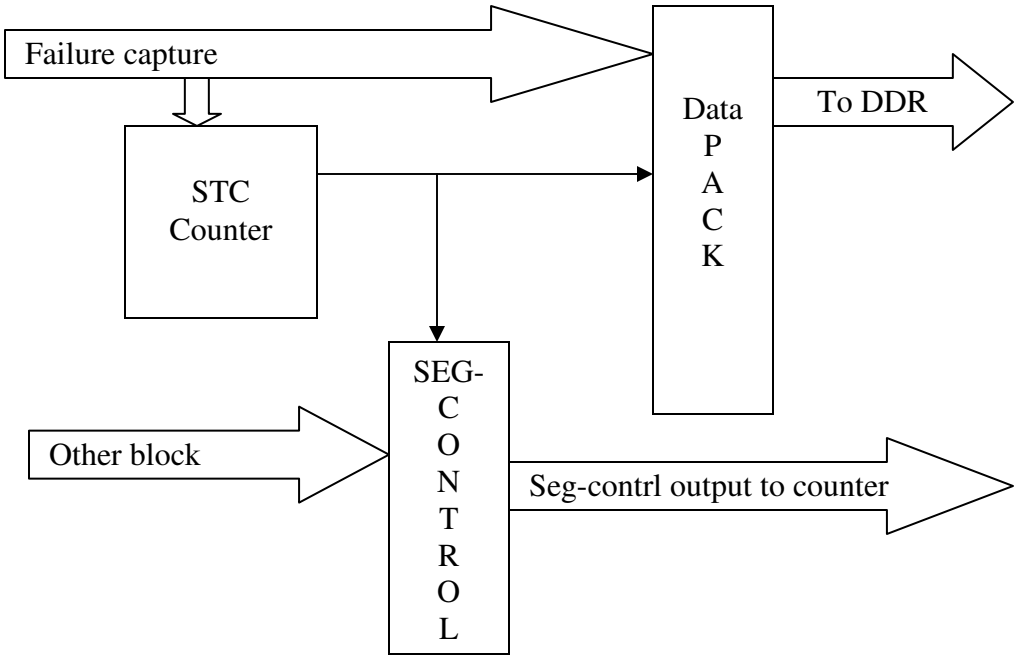

Figure 5.3 Pin CMEM DigCap block diagram

➢ Closing a Segment

The segment must be closed in order for data stored in the local buffer to be flushed to the SDRAM. Only after the data is flushed may it be available to be moved and processed. There are two ways to close a segment *[Ref-1]*.

Implicit Segment Close - When there are enough STV in the pattern to reach the predefined segment size, the segment will be closed. The segment will automatically get closed (if open) on pattern end.

Explicit Segment Close – There is only one way in which a segment may be explicitly closed - Writing to the `capt done` register. This will force the segment to close regardless of how many samples were captured.

Only when the segment is closed will the actual sample size and the last word count be captured to memory. This is done by asserting the close_seg to the Pin CMEM CCI. If the pattern wound up creating no STC and the pattern ended, both Open and Close must

be presented to the interface in order to indicate that the current segment had no data. The standard segment information will be captured; the only difference being zero will be in the seg_size field and the last word field for this segment *[Ref-1]*.

- Segment Sizes

The size of the segment to be captured will be defined via the databus – the register is used to define the sample size of the segment. When this pre-determined size is reached the segment will be closed.

If the segment size is not reached during a pattern burst, and the pattern ends, then an underflow error will be generated that indicates that insufficient number of STOREs were received in the pattern burst. If, after the pre-defined segment size is reached and the segment is closed, additional STOREs are received in the pattern burst, then an overflow error will be generated that indicates that too many STOREs were received in the pattern burst. Both these errors will be maskable and will contribute to bit 0 of the common error register *[Ref-1]*.

- Segment Storage

Once a segment has been started, the client must pack data to the interface. The captured samples will be packed into the 256-bit word. When a full word is created, data and data will be presented to the CCI interface. On each STORE, the amount of data to be captured varies based on mode as defined in the requirements. If there is an incomplete 256-bit word when the segment is closed, the unused bits will be padded with 0s to fill up a complete word.

When an overflow condition is reached, no additional data will be accepted. As mentioned previously this overflow threshold is the allocated size less 128 address. This threshold is set such that we can use the same CCI logic for all clients which may have different supported move modes *[Ref-1]*.

Chapter 6

CAPTURE MEMORY SUB MODULE VERIFICATION

6.1 Introduction

This Chapter discusses the complete verification process for the capture memory Block. As already discussed earlier, this is the first phase where module level verification is done. The complete capture memory module is divided into functional blocks capable of being verified individually. The level of abstraction is very low and interest is to exercise the design from all aspects and verify the transaction at each and every bit transition at every signal and internal node of the design. The design is to be exercised for invalid data as well to check its response to it. The design is to be verified against all the functionality specified in its hardware specification which states the features of the design. The chapter discusses the complete verification process as the flow followed *[Ref-2]*.

6.2 Identification OF Features

This is the first step to the verification. It is important to know what is to be verified. As per the requirement, the module block functionality is decided. Functionality to be implemented by each module block is explained in this document. The addresses and configuration of all registers is explained here. Hence, the module block is to be verified against this desired implementation of functionality. All the register addresses specified and the type of register mentioned must be verified to be correct. It is required to understand the functionality of all the modules as all interact in some or other way.

The important point here is the interpretation of the specification. Designers and Verification Engineers interpret the specifications independently and hence the verification will be for the specification and not for the interpretation. Now, as the functionality desired by the module is understood, the next step is to identify the features of the module. This will help in identifying the number of test cases required to be

written. The features are grouped into categories and test cases are identified for each group *[Ref-2]*.

Branch: segment history

Features:

If the numbers of samples as defined in the sample size register have been captured, the current segment will be closed. The value of sample size count on the first major stc.

Branch: segment history

Features:

The segment history register seg hist done[0] is written to 1 when a segment is closed because the number of samples programmed in the sample size registers have been captured

NOTE: Capture mode capt mde [2:0] is DIGCAP (1)

Branch: segment history

Features:

The segment history register seg last word size is written to the number of valid bits in the last 256 bit word written to the CMEM DRAM when a segment is closed because the numbers of samples defined in sample size have been captured

NOTE: A last word size of zero indicates that the last word stored had 256 valid bits of Packed data.

NOTE: Capture mode is DIGCAP (1)

Branch: segment history

Features:

The 28 bit sample size can be programmed from1 to 256M-1 when the current packing mode generates 32 packed bits per major STC.

NOTE: Capture mode is DIGCAP (1)

Branch: segment history

Features:

The 28 bit sample size can be programmed from 1 to 128M-1 when the current packing mode generates 64 packed bits per major STC.

NOTE: Capture mode is DIGCAP (1)

6.3 Developing The Testbench

As identifying features and development of TestBench are two independent processes so both of these process moves in parallel. The Testbench connects complete verification environment and DUT on same platform for driving stimulus from Testcases and comparing actual response with expected response.. The TestBench is applicable to all test cases of CMEM. The CMEM is instantiated in the Testbench. Other design supported sub modules like *CLK_CNTRL* are also instantiated in the Testbench. This Test Bench Works in conjunction with the generic Testbench that is applicable to all the Functionalities of module *[Ref-2]*.

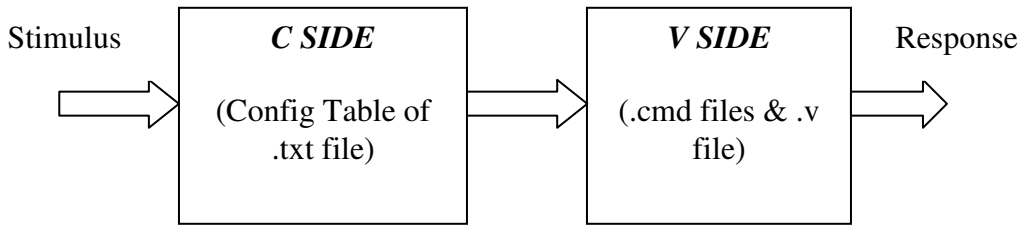

Figure 6.1 CMEM TestBench Configurations

TestBench instantiate complete CMEM along with other modules as well. During compilation only the portion of the Testbench that is applicable to Dig-cap is individual functionality is complied. The Testbench is updated as and when required based on test case requirement and updation in the RTL

6.4 Prioritization of Features

As said earlier for the verification of CMEM functionality, in order to reach to proper verification deadline, RTL and VE stability and coordination are ensured by developing some harness testcases and then depending on decided verification granularity inner functionality of module is started to be verified. Any delay during fixing of bugs, do not results in delay of verification. It is possible to start with test cases which are of some lesser priority.

6.5 Grouping Of The Testcases

As per the complexity of the feature and its implementation in the module, the number of test cases can vary. Hence, as already shown earlier, few test cases group together to and fall into some common category which shares common tasks and require same type of debugging. The grouping of test cases is done with many considerations. Functionality falling into same category requires similar type of verification approach. e.g. Any particular block related functionality is responsible for generating many different combinations of configuration based on occurrence of events. Verification of such interrupts generation requires generation of that type of event. All of them require similar type of top level configuration and hence should be grouped into common category.

6.6 Environment Settings

While starting to use the environment for the first time, it is required to make a working folder where all the environment components, various modules' Test Bench and Test Cases will reside as per specific path. It is required to Check Out the environment from the Clear Tool.

The second requirement is to set the module onto which we are working. This will indicate the module to be verified. For Address Generator, we just have to enter into directory of module. This will use all the CMEM specific code from the Verification Environment. It is possible to have multiple such directories as the project progresses. All

test cases in specific directory will execute in the environment it sees within its project directory. The only requirement for this is to set the variable project accordingly. It is just required to enter into project directory name at the command line and the directory required to be worked with. Code Build and Code Compile will be done within the directory only *[Ref-2]*.

6.7 Developeing and Debugging Testcases

As verification progresses testcases are developed for the features lying into common category. For CMEM block testcase strategy for features are as under

- The segment history register is written to the number of valid bits in the last 256 bit word written to the CMEM DRAM when a segment is closed because the numbers of samples defined in register have been captured
- Databus readback of register is 1 if readback enable is zero, than digcap usr error is one (Sample Count Overflow error occurred).

This feature is implemented in testcase and applies to environment that is already setted for this testcase. *[Ref-Appendix]*

As the verification progresses, issues arises. When any issue arises, first thing is to find the source of issue. This is done by debugging using output generated by simulation. In my project, as the simulation progresses, log files are created. They specify the complete flow of simulation. The debugging is done as under.

- Check the log file to check that whether simulation is completed or is left midway. Log file shows different messages ex. Error, Panic, Milestone, Debug etc. This will show the point of problem in terms of at processing of which stream, does the problem and from where does it arises, whether from testcase or from VE. The RTL can be viewed for this to some extend.

- If the log file is not having any issue, then the next step is Waveform viewer, Simvision is used to see the transactions in the signals. Unwanted signals can be Filtered out and important signals can be zoomed out to identify the problem.

The final conclusion to be made is that whether the problem is because of VE, TestBench, Testcase or RTL. Once it is confirmed that the problem is because of RTL, bug is reported.

6.8 Bug Reporting

As the verification progresses, issues arises. This is a good sign of progress. If no issue arises, that means that something is wrong in the TestBench or VE. So, when any issue arises, first thing is to find the source of issue. This is done by debugging using output generated by simulation. Once it is confirmed that the source of issue is somewhere in the RTL and not in the VE, TestBench or Testcase, a bug is filed for it. Brief description is given so that any one can understand the issue. The test case and the logs generated are sent to the designer who is assigned the bug *[Ref-5]*.

Sr. No.	Bug Id	Date & Time of Reporting	Brief Summary
1	10799	2009-01-03 16:57	Digcap overflow error generates irrespective of capture mode
2	10944	2009-02-20 11:44	Last word size is not latched properly during digcap mode.

6.9 Looping Structure

As the bug get resolved, designer checks in the updated RTL. This RTL is to be checked out and the test case is run on it. The results are again analyzed if the test case fails. The bug id remains same but its status changes. Designer again locates problem in the RTL and this process continues in a loop till finally the test cases passes. It is many times required to update the VE or TestBench to make the test fail pass. Again, designer can add new registers or some extra hardware as a part of DFV for the purpose to aid the verification. Once Testcase behavior comes out as per

expectations and all open issues are resolved testcase is reviewed by designer or other verification experts to make sure testcase able to reach to targeted corner of RTL, only after completion of review testcase status is considered as closed.

6.10 Regression

As and when bugs get fixed and the RTL gets updated, it is required to do regression for all previously passing testcases with the new updated RTL. It is to verify that in the task of fixing one bug, some other bug has not aroused. Hence, a list file is prepared and all test cases undergo regression. At the end of the verification process, once again all test cases are regressed to get the final picture and mark that the process completed is completed.

Regression uses scripts for executing all the test cases. The script contains all the required commands for the compilation and simulation of the test cases. Hence, at the end of regression, it provides results for all the test cases. Regression lasts for many hours depending on number of test cases.

6.11 Coverage

Once all the issues are fixed and all identified test cases passes, coverage tool is used to get the coverage of the code in terms of blocks, expressions and other metrics for
all the test cases. Finally, accumulated result for all the test cases gives the picture of the code which is not covered by existing test cases. Hence, this output serves as an input to the identification of new test cases.
Print of Code coverage tool for CMEM block is put in appendixs. *[Ref-appendixs]*

The range of coverage got for the CMEM test cases is as under:

Block Coverage: 100 %
Expression Coverage: 99 %

Chapter 7

LANGUAGES, S/W & EDA TOOLS

7.1 Platform For Verification

The Platform for Verification is provided by languages. These languages are for coding designs, writing scripts to command processes, etc. The languages used are explained here in detailed.

7.1.1 C/C++

This is the most popular language used since long time. Its flexibility is responsible for its wide use. Most languages have evolved from C/C++ and inherit its features. C++ was the result of dynamic growth of software technology. It gave rise to object oriented approach. Because of this strong feature, C/C++ is always present somewhere in any project. The idea behind VE is to make it generic and applicable to all phases of verification. It is to be made such that it can be updated later to make reusable with such other chip. Hence, C++ was selected as the base for VE.

7.1.2 SystemC & Verilog

SystemC is Hardware Description Language, whose rapidly growing use is because of its compatibility with C/C++. It is also popular for randomization so. SystemC is used to support SCV Randomization for coverage requirements as well as to fill the gap between C++ and Verilog for the project. Verilog is the most popular Hardware Description Language used for Chip Design. Its growth has being like a revolution. Verilog supports designing at many levels of Abstraction importantly Behavioral, RTL and Gate level. It supports any level of hierarchy in the design. It is the most preferred language for verification. Popular language System Verilog has developed from fusion of features of Verilog and C. Basic features of Verilog are as under:

1. Verilog HDL is a general purpose HDL that is easy to learn and easy to use. It is Similar in syntax to the C language.
2 Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL or behavioral code.
3. Verilog HDL is supported by most popular logic synthesis.
4. All fabrication vendors provide Verilog HDL libraries for postlogic synthesis Simulation. Thus, designing a chip in Verilog allows the widest choice of vendors.

7.1.3 Perl

PERL stands for Practical Extraction and Report Language. Perl is a dynamic programming language created by Larry Wall and first released in 1987. Perl borrows features from a variety of other languages including C, shell scripting (sh), AWK, sed and Lisp. The overall structure of Perl derives broadly from C. Perl is procedural in nature, with variables, expressions, assignment statements, brace-delimited code blocks, control structures, and subroutines.

Perl also takes features from shell programming. It is used as a CGI (Common Gateway Interface) language, a programming language for Unix, linux or for Windows. Perl has many and varied applications, compounded by the availability of many standard and third-party modules. Perl is often used as a glue language, tying together systems and interfaces that were not specifically designed to interoperate, and for "data managing", converting or processing large amounts of data for tasks like creating reports. In fact, these strengths are intimately linked. The combination makes perl a popular all-purpose tool for system administrators, particularly as short programs can be entered and run on a single command line.

7.1.4 Makefile

"make" is a utility for automatically building large applications. Files specifying instructions for make are called Makefiles. make is most commonly used in C/C++ projects, but in principle it can be used with almost any compiled language.

The basic tool for building an application from source code is the compiler. make is a separate, higher-level utility which tells the compiler which source code files to process. It tracks which ones have changed since the last time the project was built and invokes the compiler on only the components that depend on those files. Although in principle one could always just write a simple shell script to recompile everything at every build, in large projects this would consume a prohibitive amount of time. Thus, a makefile can be seen as a kind of advanced shell script which tracks dependencies instead of following a fixed sequence of steps.

Today, programmers increasingly rely on Integrated Development Environments and language-specific compiler features to manage the build process for them instead of manually specifying dependencies in makefiles. However, make remains widely used, especially in Unix-based platforms. The makefile just contains a list of file dependencies and commands needed to satisfy them.

7.2 OS: RED HAT LINUX

Linux was originally created by Linus Torvalds during his graduate studies at the University of Helsinki in Finland. Linus wrote Linux as a small PC-based implementation of UNIX. During the summer of 1991 Linus made Linux public on the Internet. In September of that same year, version 0.01 was released. A month later, version 0.02 was released, with version 0.03 following several weeks later. In December, Linux was numbered at 0.10, and by the end of the month, virtual memory (disk paging) was added. Within a year, Linux had a thousand more features and was well on its way to becoming a self-compiling, usable operating system. Linus made the source code freely available and encouraged other programmers to develop it further. They did, and Linux

continues to be developed today by a worldwide team, led by Linus, over the Internet. It supports a wide range of Softwares and Hardwares. Again, it is a secure system and hence is preferred the most when networking is involved in the project. e.g offshore projects. . The Operating System installed in the work station is i386 -RedHat Linux v3.0. All the EDA Tools and the applications to be used through the project are supported by this OS. The secure networking to be used for working at the work station at US, is thoroughly supported by this OS. All the work stations are installed with this OS.

7.3 SUPPORTING APPLICATIONS

For any project, along with the main languages and software, some supporting software applications are required to manage the project. Some commonly used for verification are discussed under.

7.3.1 S/W: ClearQuest

ClearQuest a database for bugs. It lets people report bugs and assigns these bugs to the appropriate developers. Developers can use ClearQuest to keep a to-do list as well as to prioritize, schedule and track dependencies. In verification, all the issues and bugs are filed in ClearQuest for the developer. Each bug can be issued a priority based on urgency for its resolve. Each bug can have interred dependencies. Each bug is assigned a unique id. Any authorized person can login into the ClearQuest and view and update or add some comments the status for any bug. Each bug is composed of many fields. Few of them are mentioned below:

Bug ID, Reported By, Date Found, Root Cause :
This shows unique id of Bug, Registered Reporter Name, Date and time relevant information , Root cause shows reason of bug in terms of VE, RTL coding.

Status, Investigator, Date Last Modified:
This field is used to show status of a bug (open, resolved), Investigator who will work on fixing of bug, last modification date.

Severity, Project, Bug Type, Testcase:

This field targets to show severity of bug ex. urgent, Platform and Project belongs to bug, Testcase name where bug is to be noticed.

Description Log, Resolution Log, Verification Log:

Log of description shows expectation of bug reporter and misbehavior noticed by bug reporter, Log of Resolution shows summary of resolution actions taken by bug investigator. After fixing bug by investigator, summary of verification done by bug reporter.

Dependency:

If a bug can't be fixed until another bug is fixed, that's a dependency. For any bug, you can list the bugs it depends on and bugs that depend on it. ClearQuest can display a dependency graphs which shows which the bugs it depends on and are dependent on it.

Attachment:

Adding an attachment to a bug can be very useful. Test cases, screen shots and editor log can help pinpoint the bug and help the developer reproduce it. If you fix a bug, attach the patch to the bug. This is the preferred way to keep track of patches since it makes it easier for others to find and test.

7.3.2 ClearCase

The ClearCase implements SCM (Software Configuration Management). Its main features or we can say advantages are 1) Version Control: version all types of files and directories. 2) Build Management: ensure integrity of all software elements, accurately reproduce every release. 3) Workspace Management: work in parallel with other developers. 4) Process Control: record and report actions, history and milestones. Its version control system keeps track of all work and all changes in a set of files, typically the implementation of a software project, and allows several (potentially widely separated) developers to collaborate. All the source files are managed in a common platform called as a VOB (Versioned Object Base) *[Ref-5]*.

VOB plays duties like server for one or more projects. ClearCase uses client-server architecture: a server stores the current version(s) of the project and its history, and clients connect to the server in order to check-out a complete copy of the project, work on this copy and then later check-in their changes. Typically, client and server connect over a LAN or over the Internet, but client and server may both run on the same machine if ClearCase has the task of keeping track of the version history of a project with only local developers. The VOB normally runs on Windows XP, Windows 2000, Unix including Red Hat Linux.

Several developers may work on the same project concurrently, each one editing files within his own working copy of the project, and sending (or checking in) his modifications to the server. To avoid the possibility of people stepping on each other's toes, the VOB will only accept changes made to the most recent version of a file. Developers are therefore expected to keep their working copy up-to-date by incorporating other people's changes on a regular basis. If the check-in operation succeeds, then the version numbers of all files involved automatically increment, and the VOB writes a user-supplied description line, the date and the author's name to its log files. ClearCase can also run external, user-specified log processing scripts following each commit. These scripts are installed by an entry in ClearCase's log info file, which can trigger email notification, or convert the log data into a web-based format. Clients can also compare different versions of files, request a complete history of changes, or check-out a historical snapshot of the project as of a given date or as of a revision number. Clients can also use the "gvp uwa –v -merge" command in order to bring their local copies up-to-date with the newest version on the server. This eliminates the need for repeated downloading of the whole project. ClearCase can also maintain different "branches" of a project. For instance, a released version of the software project may form one branch, used for bug fixes, while a version under current development, with major changes and new features, forms a separate branch.

- Few terms related to ClearCase are discussed below:

Main Branch: basically means the code and all its various versions in the repository. Main Branch consists of the main trunk which contains all the main code files, and sometimes may be branch for each separate releasable product.

Release Branch: Branch designated for holding official releases of a product. It facilitates moving and tracking changes among releases. Reduce complexity and length of version tree.

Developer Branch: Branch designated for individual development, facilitates to track individual activity.

Working copy: Your local copy of the source code. You don't work on the server code directly, instead you work on the working copy and changes are merged together.

- Few useful Commands guideline related to ClearCase are discussed below:

prune

Mark your private files as removed. This means that you are updated with latest repository, your local checkin files are no more available. .

merge

Keep you in cope with latest release available at repository, but checkout files are kept unchanged.

mkelem:

Insert a new file into VOB. Will be visible in the repository only after a checkin and a gather command is issued.

checkout

Checks out the source files defined by modules. Note that multiple checkouts can be made in different directories, this can be very confusing.

checkin

Checks in the source files defined by modules to your local workspace, will not be visible in the repository Note that multiple checkins can be made in different directories, this can be very confusing

gather

Commit your changes into the the repository.

diff

Check difference with your private files against the repository.

remove

Remove the file from ClearCase revision control. This command moves the repository file permanently from VOB so no way to recover it.

7.4 EDA TOOLS

EDA Tools plays very important role in this industry. For every task to be accomplished, these EDA Tools tend to reduce the time required and performs the task accurately. Developments of EDA Tools are at the target of every vendor as its demand is increasing tremendously. They have revolutionized every process in the development of any design. In the following topics, I will be discussing the Major EDA Tools use by me for the project. All design tools are by the company Cadence.

7.4.1 Compilation & Simulation
Tool: Affirma NCsim v 06.11-s002
Company: Cadence

For the C++ compilation, freeware g++ compiler from GCC is used. This is free software. It is available with Linux Operating System.

The NC-Verilog Simulator delivers high-performance, high-capacity Verilog simulation with transaction/signal viewing and integrated coverage analysis. It is fully compatible with the Incisive functional verification platform, so design teams can easily upgrade to the Incisive Unified Simulator and Incisive XLD team Verification, with native support for Verilog, VHDL, SystemC Verification Library, PSL/Sugar assertions, and Acceleration-on-Demand. Verilog IEEE 1364-1995 and a majority of IEEE 1364-2001 extensions, SystemVerilog (IEEE-1800). It compiles directly to host processor machine code for maximum performance.

The NC-Verilog Simulator provides the industry's premier simulation performance for Verilog designs using the unique native-compiled architecture of the Incisive Unified

Simulator. It produces efficient native machine code directly from Verilog for high-speed execution. Linked list scheduling of the resulting data structures pre-processes signal actions and maximizes the effectiveness of modern caching algorithms available in today's computing platforms. The NC-Verilog performance profiler identifies bottlenecks. Designers find areas of high activity by viewing how each module contributes to overall performance. Minor changes can greatly improve simulation performance by identifying the areas that consume the most simulation time. NC-Verilog 64-bit capacity simulates designs larger than 100 million gates.

The unified NC-Verilog simulation and debug environment makes it easy to manage multiple simulation runs and analyze the design and testbench. Its transaction/waveform viewer and schematic tracer quickly trace design behavior back to the source. The NC-Verilog source viewer lets designers examine their design, set complex breakpoints to control simulation execution, and access results in both interactive and post processing debug modes. The NC-Verilog Simulator provides access to a wide variety of coverage metrics to help determine how well tests have exercised the design. These include block coverage, path coverage, expression coverage, state variable coverage, state transition coverage, state sequence coverage, and toggle coverage. Integrated coverage analysis and display tools speed the process of determining which additional tests will need to be developed.

7.4.2 Waveform Viewer
Tool: Simvision v 06.11-s002
Company: Cadence

The SimVision analysis environment is a unified graphical debugging environment for Verilog-XL, NC-Verilog, NC-VHDL, and NC-Sim.
You can run SimVision in either of the following modes:

Simulation mode

In simulation mode, you view "live" simulation data. That is, you analyze the data while the simulation is running. You can control the simulation by setting breakpoints and stepping through the design.

SimVision provides several tools to help you track the progress of the simulation:

- Source code window
- Navigator window
- Watch window
- Signal Flow Browser
- Cycle window
- Schematic window
- Waveform window

Many of these windows are linked, so that when you select an object in one window, it is selected in the other windows as well.

Post-processing environment (PPE) mode

In PPE mode, you analyze simulation data after simulation has completed. You have access to all of the SimVision tools, except for the simulator. As in simulation mode, all of these windows are linked, so that when you select an object in one window, it is selected in the other windows as well.

7.4.3 Code Coverage
Tool: Incisive v 06.11-s002
Company: Cadence

Code Coverage is an important metric for Verification Engineers to measure their effort. It gives the view of the scenarios created by the test cases to verify the RTL. The remaining can then be again generated by the Verification engineers.

Incisive by Cadence is a powerful Code Coverage Tool that gives coverage in many metrics as desired by verification engineer. It can provide the graphical view of scenarios created. Desired metrics can be selected as required. It has the capability of extracting FSM from RTL design. It supports code, path, expression, fsm (state, transition and sequence), toggle, variables and gate coverage. Such different metrics of Coverage acts as a feedback loop to improve the input stimulus.

APPENDIX

[1] Test plan and TESTCASE of CMEM segement open on first STV

- Test Plan

TEST Name: mem_seg_open

STATUS: OPEN

REVIEWED: DONE

ASSIGNED TO: soriah

REVIEWED BY:

LAB PORTABLE: YES

OPEN ISSUES:

TEST STRATEGY:

when number of samples defined in a__seg_sample_size_w0/w1 have been captured, check followings:

1. cmem_seg_hist_done[0] is written to 1.
2. the current segment should be closed.
3. cmem_seg_hist_seg_last_word_size[9:0] is written to the number of valid bits in the last cmem word.

REFERENCE DOCS:

CMEM FIS

FEATURES TESTED:
FEAT_ 3001
FEAT_ 3002
FEAT_ 3003

CORNER CASES:

KEY TECHNICAL POINTS:

TEST ASSUMPTIONS:

VERIFICATION AIDS:

RANDOMIZATIONS:

PROCEDURE:

 # FEAT _3001

If the number of samples as defined in the a__seg_sample_size_w0/w1 register have been captured, the current segment will be closed.

 # FEAT_ 3002

The segment history register a_tg_cmem_seg_hist_done[0] is writtento 1 when a segment is closed because the number of samples programmed in the a__seg_sample_size_w0/w1 registers have been captured

 # FEAT _3003

The segment history register a_tg_cmem_seg_hist_seg_last_word_size[9:0] is written to the number of valid bits in the last 256 bit word written to the CMEM DRAM when a segment is closed because the number of samples defined in a__seg_sample_size_w0/w1 have been captured

NOTE: A last word size of zero indicates that the last word stored had 256 valid bits of packed data.

#Scenario 1:

1. Configured the following bit charts
   # Capture Mode A__CAPT_MDE[2:0] =0x1 for Digcap
   # packing mode a_tg_cmem_pck_mde random mode.
   # Flush rank flush_rank_off_en[0] = 0x0 segment open when pattern ends.
   # start address cmem_seg_start_adr_w0/w1[28:0] to random value as per DRAM size and ensure address is modulo-8.
   # end address cmem_end_adr_w0/w1[28:0] Make sure (end_adr - start_adr) >= 256.
   # Seg_size seg_sample_size_w0/w1 = number of stv_lfvm in burst. keep number of stv_lfvm such that last word size should not full with 256 bit.

2. apply cmem init
3. Execute pattern and wait for pattern to finish.
4. checker :
   # Expect a_tg_cmem_seg_hist_done[0] is 0x1.
   # Expect that a__cmem_capt_busy[0] is 0x0.
   # Expect a_tg_cmem_seg_hist_seg_last_word_size[9:0] depend on packing mode and segment size and number of STV_LFVM.
   # Expect a_tg_cmem_seg_hist_seg_size_w0/w1/w2[38:0] according to sample size and number of stv_lfvm.

Check that no errors have occurred

#Scenario 2:

1. Repeat step 1 to 4 of scenario #1 except sample size.
   # Seg_sizeseg_sample_size_w0/w1 = number of stv_lfvm in burst. keep number of stv_lfvm such that last word size should be full with 256 bit.
   # Note given in FEAT_CMEM_SEGHIST_3003 states read back of a_tg_cmem_seg_hist_seg_last_word_size is zero when last word has 256 valid bits.

#Scenario 3:

1. Repeat step 1 to 4 of scenario #1 except sample size is equal to zero.
   # Seg_size seg_sample_size_w0/w1 = number of stv_lfvm = 0.
   # Expect a_tg_cmem_seg_hist_seg_size_w0/w1/w2 and a_tg_cmem_seg_hist_seg_last_word_size[9:0] should be zero

PASS/FAIL CRITERIA:

- Test shouts for any mismatch of a_tg_cmem_seg_hist_done, a__cmem_capt_busy or a_tg_cmem_seg_hist_seg_last_word_size.
- Testcase will report error on mismatches based on comparison of generated Each cycle of Pin CMEM data.

- Test Case

```
//---------File Include section----------------------------------------------------------//

#include "dartTestCommon.h"
#include "apiCmem.h"
#include "mpgCommon.h"
#include "apiMpg.h"
#include "apiMapSupport.h"
#include "bitStructData.h"
//--------Test case start ---------------------------------------------------------------//
extern "C" int cmem_digcap_seg_hist_mem_seg_open(void)
{    //----Local veriable declaration ------------------------------------------------//
    // get DRAM size in Mb and save max address of DRAM
    u_int32 MSB_Address = ((apiMapSupport.getDramSize()) << 17) - 1;
    u_int32 startAddr = Random.getInRange(0x0,(MSB_Address-0xFF)) & (~0x7);
    u_int32 endAddr = startAddr + 0xFF;
    u_int32 stv_count = 15;
    u_int32 sample_size;
    u_int32 cmemDdrRdSize;
    u_int32 pack_mode[9] = {B_TG_CMEM_DATA_PCK_SITE_0,
                        B_TG_CMEM_DATA_PCK_SITE_1,
                        B_TG_CMEM_DATA_PCK_EVEN_SITE_0,
                        B_TG_CMEM_DATA_PCK_EVEN_SITE_1,
                        B_TG_CMEM_DATA_PCK_EVEN_BOTH,
                        B_TG_CMEM_DATA_PCK_ODD_SITE_0,
                        B_TG_CMEM_DATA_PCK_ODD_SITE_1,
                        B_TG_CMEM_DATA_PCK_ODD_BOTH,
                        B_TG_CMEM_DATA_PCK_OR_ADJ
                        };
     u_int32 cmem_feat_en = TT_READ(A_TG_CMEM_FEAT_EN);
    Msg_Printf("== cmem_feat_en=%d \n", cmem_feat_en);

    Scenario 1: Sample size = number of STC and sample size such that last word
                Size should not be full with 256 bit

    //-----CMEM configuration ------------------------------------------------------//
    Msg_Printf("SCENARIO #1\n");
    apiCmemT::ConfigT cmemConfig;
    cmemConfig.startAddr  = startAddr;
    cmemConfig.endAddr    = endAddr;
    cmemConfig.closeSeg   = (cmem_feat_en == 0) ? true : false;
    cmemConfig.captMode   = B__CAPT_MDE_DIGCAP;
    cmemConfig.packMode   = pack_mode[Random.getInRange(0, 8)];
    cmemConfig.lfvcSyncEn     = true;
    cmemConfig.lfvcSyncCyc    = 0;
```

```
apiCmem.checkCmemPackMode(cmemConfig.packMode, cmemDdrRdSize);
cmemDdrRdSize = (cmemDdrRdSize == 1) ? 8 : 4;
do{
      sample_size= Random.getInRange(0, stv_count);
      Msg_Printf("sample_size = %d \n", sample_size);
  }    while((sample_size % cmemDdrRdSize) == 0);

cmemConfig.sampleSize = sample_size;
cmemConfig.lfvcTrigCnt = sample_size;
apiCmem.setupCmemCapture(cmemConfig);
Msg_Printf("start address = %d end address = %d packMode=%d \n", startAddr,
               endAddr, cmemConfig.packMode);

apiCmem.resetCmem();
 PG_ExecATP_name("cmem_digcap_seg_hist_mem_seg_open.atp",
                              "cmem_seg_hist_mem");
apiCmem.execModel(true); //call cmem expect model
//------Expectation----------------------------------------------------------//
TT_EXPECT(A__CMEM_CAPT_BUSY,0x0);
TT_EXPECT(A_TG_CMEM_SEG_HIST_DONE ,0x1);
TT_EXPECT(A__LC_ERR, 0x0);
```

Scenario 2: Sample size = number of STC and sample size such that last word Size should be full with 256 bit

```
Msg_Printf("SCENARIO #2\n");
do{
       sample_size= Random.getInRange(0, stv_count);
        Msg_Printf("sample_size = %d \n", sample_size);
      }while( (sample_size == 0) || (sample_size % cmemDdrRdSize) != 0);

cmemConfig.sampleSize = sample_size;
cmemConfig.lfvcTrigCnt    = sample_size; //keep sample size = number of STV
apiCmem.setupCmemCapture(cmemConfig);
apiCmem.resetCmem();
PG_ExecATP_name("cmem_digcap_seg_hist_mem_seg_open.atp",
                             "cmem_seg_hist_mem");
apiCmem.execModel(true); //call cmem expect model
//------Expectation----------------------------------------------------------//
TT_EXPECT(A__CMEM_CAPT_BUSY,0x0);
TT_EXPECT(A_TG_CMEM_SEG_HIST_DONE ,0x1);
TT_EXPECT(A__LC_ERR, 0x0);
```

#Scenario 3: sample_size = 0x0;

```
Msg_Printf("SCENARIO #3\n");
```

```
sample_size = 0x0;
Msg_Printf("sample_size = %d \n", sample_size);

cmemConfig.sampleSize = sample_size;
cmemConfig.lfvcTrigCnt    = sample_size;
apiCmem.setupCmemCapture(cmemConfig);
apiCmem.resetCmem();
PG_ExecATP_name("cmem_digcap_seg_hist_mem_seg_open.atp",
                        "cmem_seg_hist_mem");
apiCmem.execModel(true); //call cmem expect model

//------Expectation----------------------------------------------------------------//
TT_EXPECT(A_TG_CMEM_SEG_HIST_DONE,0x1);
TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_LAST_WORD_SIZE,0x0);
TT_EXPECT(A__LC_ERR, 0x0);
 return(0);
 }
```

➢ Code Coverage Result

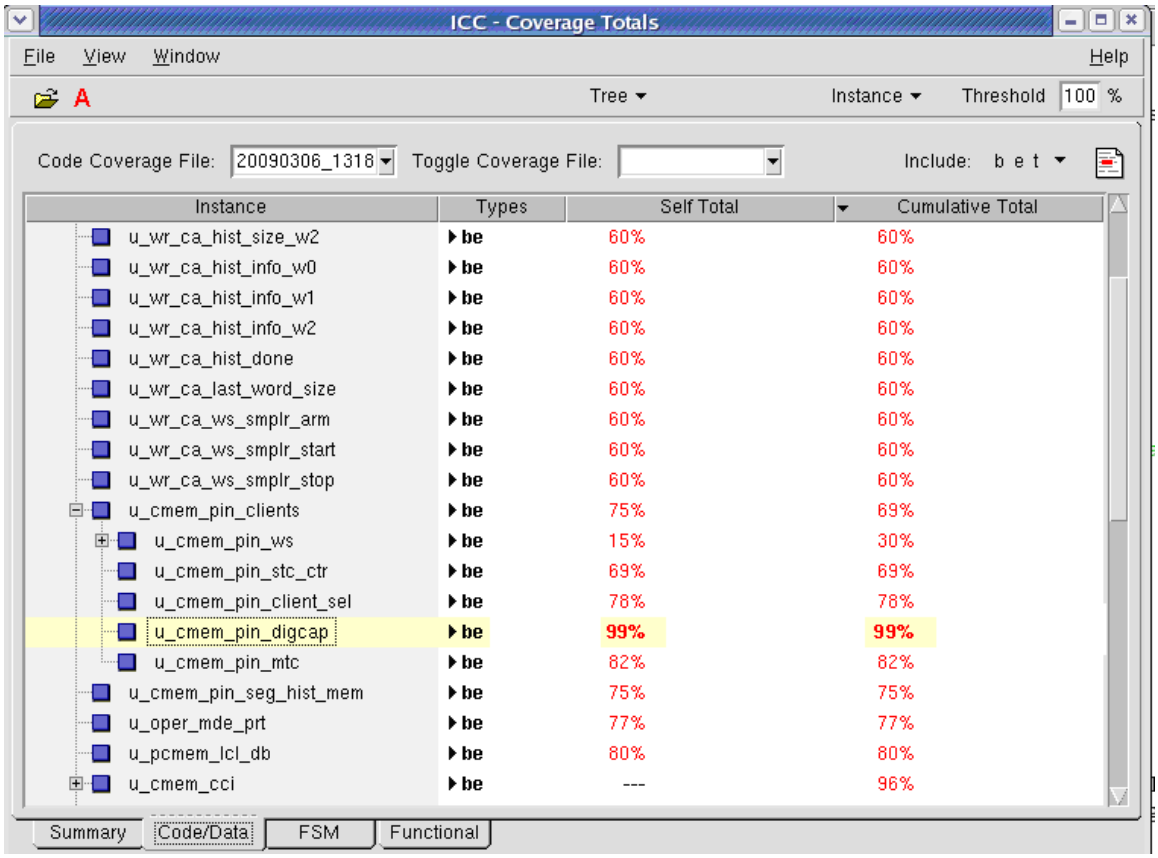

Figure A.1 – code coverage result of digcap block

[2] Test plan and Testcase of CMEM segement close during keepalive

- Test Plan

TEST ID: cmem_digcap_keepalive

STATUS: OPEN

REVIEWED: TESTPLAN

SCHEDULE:

ASSIGNED TO: soriah

REVIEWED BY:

LAB PORTABLE: yes

OPEN ISSUES:

TEST STRATEGY:
To verify that segment open on first stv after cmem init is set using keepalive

REFERENCE DOCS:
DAGGER Cmem.fis

FEATURES TESTED:
FEAT_1100

CORNER CASES:

TEST ASSUMPTIONS:

VERIFICATION AIDS:

RANDOMIZATIONS:
packing mode
cmem start address

PROCEDURE:

Test Prerequisites:
A. run map (patlist)

Scenario pattern part 1 pattern part 2

0	stv < samples	stv > samples
1	stv = samples	stv < samples
2	stv > samples	stv = samples

Repeat Test Sequence for all 3 scenarios listed above
Test Sequence:
- configure DigCap:
  - capture mode = digcap
  - closeSeg = false
  - sample size = N

scenario 0 : N is one more than the number of STV cycles before the vector that sets the user flag
scenario 1 : N is equal to the number of STV cycles before the vector that sets the user flag
scenario 2 : N is one less than the number of STV cycles before the vector that sets the user flag

start address = random value within legal address range - 0x1FF
end address = start address + 0x1FF
difference between end address and start address (0x1FF) must be greater than sample size N

Configure pattern opcode and vector data through atp (see cmem_keepalive.atp file) without running the pattern

- Now overwrite the srm opcodes from the testcase to set user flag and wait on same vector till the flag is cleared.

After loading these opcode over the ones loaded by atp, the opcode sequence might look like below:

. cycle 0 to 10 : with stv_lfvm
. cycle 11 to 300 : without stv_lfvm
. cycle 12 : set user flag zero expect high
. cycle 13 : set user flag zero high
. cycle 14 : stay at this cycle till the user flag zero is low
. cycle 15 to 32 with stv_lfvm

Note that cycles 11 to 300 are intended to push the prior stv cycles through the pipeline to cmem before the user flag is set high. Cycles 11 to 14 do NOT have stv set.

- Clear user flag zero from test
- Start patten execution and wait for user flag zero to be set.
- **Check:**
  if scenario == 0 # we have underflow
    expect A__CMEM_CAPT_BUSY == 1
    expect A_TG_CMEM_SEG_HIST_DONE == 0

```
  if scenario == 1 # segment closed with no error
                expect A__CMEM_CAPT_BUSY  == 0
                expect A_TG_CMEM_SEG_HIST_DONE == 1
  if scenario == 2 # segment closed with overflow
                expect A__CMEM_CAPT_BUSY  == 0
                expect A_TG_CMEM_SEG_HIST_DONE == 1
- Write capt_done to force segment closed
- Check:
  if scenario == 0
                expSTC = sample size - 1
  if scenario == 1
                expSTC = sample size
  if scenario == 2
                expSTC = sample size + 1
                expect A__CMEM_CAPT_BUSY  == 0
                expect A_TG_CMEM_SEG_HIST_DONE == 1
                expect A_TG_CMEM_SEG_HIST_SEG_LAST_WORD_SIZE ==
                        (expSTC * packed bits per cycle) % 256
                expect A_TG_CMEM_SEG_HIST_SEG_SIZE_W0 == (expSTC *
                        packed bits per cycle) / 256
                expect A_TG_CMEM_SEG_HIST_SEG_SIZE_W1 == 0
                expect A_TG_CMEM_SEG_HIST_SEG_SIZE_W2 == 0
                expect A__CMEM_STC_CNT_W0 == STC << 4
                expect A__CMEM_STC_CNT_W1 == 0
                expect A_TG_CMEM_STC_CNT_W2    == 0
  if scenario == 1
                expect A__LC_ERR == 0
  else
                expect A__LC_ERR == 1

- Write cmem_init to initialize for a new segment
- Set sample size = M (where M is the number of STV cycles after the pattern waits on
   the user flag)
- Set a__cmem_flush_rank_off_en to 1
- Clear user flag zero from test
- Wait for pattern to be finished.
- Check:
  if scenario == 0 # segment closed with overflow
                expect STC = sample size + 1
  if scenario == 1 # we have underflow
                expSTC = sample size - 1
  if scenario == 2 # segment closed with no error
                expSTC = sample size
                expect A__CMEM_CAPT_BUSY  == 0
                expect A_TG_CMEM_SEG_HIST_DONE == 1
                expect A_TG_CMEM_SEG_HIST_SEG_LAST_WORD_SIZE ==
```

```
                (expSTC * packed bits per cycle) % 256
            expect A_TG_CMEM_SEG_HIST_SEG_SIZE_W0 == (expSTC *
                packed bits per cycle) / 256
            expect A_TG_CMEM_SEG_HIST_SEG_SIZE_W1 == 0
            expect A_TG_CMEM_SEG_HIST_SEG_SIZE_W2 == 0
            expect A__CMEM_STC_CNT_W0 == STC << 4
            expect A__CMEM_STC_CNT_W1 == 0
            expect A_TG_CMEM_STC_CNT_W2    == 0
if scenario == 2
            expect A__LC_ERR == 0
else
            expect A__LC_ERR == 1
```

PASS/FAIL CRITERIA:

Test generates error for any mismatch occured in expectation as described in procedure.

.

➢ TESTCASE

```
//---------File Include section----------------------------------------------------------//
#include "dartTestCommon.h"
#include "apiCmnSetup.h"
#include "apiTiming.h"
#include "apiFailPro.h"
#include "apiSteering.h"
#include "apiVm.h"
#include "apiSrm.h"
#include "mpgCommon.h"
#include "apiMpg.h"
#include "apiCmem.h"

#define _UNDERFLOW M_TG_CMEM_DIGCAP_USR_ERR_UNDFLW
#define _OVERFLOW  M_TG_CMEM_DIGCAP_USR_ERR_OVRFLW
#define _NONE      M_TG_CMEM_DIGCAP_USR_ERR_NONE
//--------Test case start ----------------------------------------------------------------//
extern "C" int cmem_digcap_keepalive(void)
{  //----Local veriable declaration ------------------------------------------------//
   bool digcapPlus = TT_READ(A_TG_CMEM_FEAT_EN) ? true : false;

   // For now the user err enable is mapped to 0

TT_WRITE(A_TG_CMEM_DIGCAP_USR_ERR_EN,M_TG_CMEM_DIGCAP_USR_
ERR_ALL);

Msg_Milestone("%s:setup SRM Opcode and  operands.\n",__PRETTY_FUNCTION__);
apiSrmT::srmDatT srmPttrn[4];

//There are 69 vectors in the pattern, changing the 29th to 32nd vector
   u_int32 mjrSrmOpcVecIndex = 29;

   //Vector Index: 29 repeat dummy cycles to let cmem flush
   srmPttrn[0].opc  = dartNmSpc::OPC_RPT;
   srmPttrn[0].oper = 384; // the cmem is some 300+ ranks after the PG, so let it catch up
   srmPttrn[0].stvA = 0;
   srmPttrn[0].stvB = 0;
   srmPttrn[0].stvC = 0;
   srmPttrn[0].stvD = 0;
   srmPttrn[0].ign   = 0;
   srmPttrn[0].crEnA = 1;
   srmPttrn[0].crEnB = 1;
   srmPttrn[0].crEnC = 1;
   srmPttrn[0].crEnD = 1;
   srmPttrn[0].tsA = 0;
```

```
   srmPttrn[0].tsE = 1; // The A pipe is major period and E pipe period is zero

 //Vector Index: 30 Set User Flag 0 to Expect Hi (jump while high) and others to
   Ignore
   srmPttrn[1].opc = dartNmSpc::OPC_SETFLGEN;
   srmPttrn[1].oper =

(dartNmSpc::B_SETFLG_EN_OPER_EXP_IGNORE<<dartNmSpc::S_OPER_USER_F
LAG3) |

(dartNmSpc::B_SETFLG_EN_OPER_EXP_IGNORE<<dartNmSpc::S_OPER_USER_F
LAG2) |

(dartNmSpc::B_SETFLG_EN_OPER_EXP_IGNORE<<dartNmSpc::S_OPER_USER_F
LAG1) |
(dartNmSpc::B_SETFLG_EN_OPER_EXP_SET
<<dartNmSpc::S_OPER_USER_FLAG0);
   srmPttrn[1].stvA = 0;
   srmPttrn[1].stvB = 0;
   srmPttrn[1].stvC = 0;
   srmPttrn[1].stvD = 0;
   srmPttrn[1].ign   = 0;
   srmPttrn[1].crEnA = 1;
   srmPttrn[1].crEnB = 1;
   srmPttrn[1].crEnC = 1;
   srmPttrn[1].crEnD = 1;
   srmPttrn[1].tsA = 0;
   srmPttrn[1].tsE = 1; // The A pipe is major period and E pipe period is zero

//Vector Index: 31 Set Usr Flag 0 to 1'b1
   srmPttrn[2].opc  = dartNmSpc::OPC_SETFLG; // "Set" the flag.
   srmPttrn[2].oper =  dartNmSpc::
                B_SETFLG_OPER_SET<<dartNmSpc::S_OPER_USER_FLAG0;
   srmPttrn[2].stvA = 0;
   srmPttrn[2].stvB = 0;
   srmPttrn[2].stvC = 0;
   srmPttrn[2].stvD = 0;
   srmPttrn[2].ign   = 0;
   srmPttrn[2].crEnA = 1;
   srmPttrn[2].crEnB = 1;
   srmPttrn[2].crEnC = 1;
   srmPttrn[2].crEnD = 1;
   srmPttrn[2].tsA = 0;
   srmPttrn[2].tsE = 1; // The A pipe is major period and E pipe period is zero

//Vector Index: 32 Stay at this vector till the flag clears
```

```
    srmPttrn[3].opc  = dartNmSpc::OPC_JMP;
    srmPttrn[3].oper = dartNmSpc::SUBOP_JUMPFLG<<dartNmSpc::S_SUBOP;
    srmPttrn[3].oper|= 32; // subop|oper into oper field.
    srmPttrn[3].stvA = 0;
    srmPttrn[3].stvB = 0;
    srmPttrn[3].stvC = 0;
    srmPttrn[3].stvD = 0;
    srmPttrn[3].ign   = 0;
    srmPttrn[3].crEnA = 1;
    srmPttrn[3].crEnB = 1;
    srmPttrn[3].crEnC = 1;
    srmPttrn[3].crEnD = 1;
    srmPttrn[3].tsA = 0;
    srmPttrn[3].tsE = 1; // The A pipe is major period and E pipe period is zero

//ATP loads with 69 vectors (68 have stv) into srm, it does NOT run the pattern yet
    PG_ExecATP_name("cmem_keepalive.atp","cmem_seg_hist_mem", false);
    //Rewrite the four locations in srm opcode
    Msg_Milestone("%s: Loading SRM Opcode and
                          operands.\n",__PRETTY_FUNCTION__);
    apiSrm.loadSrmOpcode(mjrSrmOpcVecIndex,srmPttrn,4);

    u_int32 expLcErr;
    u_int32 expDigcapErr;
    u_int32 expDone;
    u_int32 expBusy;
// get DRAM size in Mb and save max address of DRAM
    u_int32 MSB_Address = ((apiMapSupport.getDramSize()) << 17) - 1;
    u_int32 cmemDdrRdSize;
    u_int32 expLastWordSize;
    u_int32 expSegSize;
    u_int32 expStcCnt;
    u_int32 pack_mode[9] = {B_TG_CMEM_DATA_PCK_SITE_0,
                        B_TG_CMEM_DATA_PCK_SITE_1,
                        B_TG_CMEM_DATA_PCK_EVEN_SITE_0,
                        B_TG_CMEM_DATA_PCK_EVEN_SITE_1,
                        B_TG_CMEM_DATA_PCK_EVEN_BOTH,
                        B_TG_CMEM_DATA_PCK_ODD_SITE_0,
                        B_TG_CMEM_DATA_PCK_ODD_SITE_1,
                        B_TG_CMEM_DATA_PCK_ODD_BOTH,
                        B_TG_CMEM_DATA_PCK_OR_ADJ
                        };
    apiCmemT::ConfigT cmemConfig;
```

```
// scenario    pattern part 1    pattern part 2
//    0        stv < samples     stv > samples
//    1        stv = samples     stv < samples
//    2        stv > samples     stv = samples

for (u_int32 scenario = 0; scenario < 3; scenario++) {
      // Specify expected samples - first vector has no stv, 3rd vector is a repeat 200
       switch (scenario) {
          case 0:
             // stv < samples (underflow)
             cmemConfig.sampleSize = mjrSrmOpcVecIndex+199-0;
             expStcCnt = cmemConfig.sampleSize - 1;
             if (digcapPlus) {
                expDigcapErr = _UNDERFLOW; expDone = 0; expBusy = 1;
             }
             else {
                expDigcapErr = _NONE; expDone = 0; expBusy = 1;
             }
             break;
          case 1:
             // stv = samples
             cmemConfig.sampleSize = mjrSrmOpcVecIndex+199-1;
             expStcCnt = cmemConfig.sampleSize;
             if (digcapPlus) {
                expDigcapErr = _NONE; expDone = 1; expBusy = 0;
             }
             else {
                expDigcapErr = _NONE; expDone = 0; expBusy = 1;
             }
             break;
          case 2:
             // stv > samples (overflow)
             cmemConfig.sampleSize = mjrSrmOpcVecIndex+199-2;
             expStcCnt = cmemConfig.sampleSize + 1;
             if (digcapPlus) {
                expDigcapErr = _OVERFLOW; expDone = 1; expBusy = 0;
             }
             else {
                expDigcapErr = _NONE; expDone = 0; expBusy = 1;
             }
             break;
          default:
             Msg_Error("Illegal value (0x%0x) in scenario case statement\n",scenario);
       }
```

```
expLcErr = (expDigcapErr) ? 1 : 0;
cmemConfig.captMode   = B__CAPT_MDE_DIGCAP;
cmemConfig.startAddr  = Random.getInRange(0x0,(MSB_Address-0x01FF)) &
                        (~0x7);

cmemConfig.endAddr    = cmemConfig.startAddr + 0x01FF;
cmemConfig.closeSeg   = false;
apiCmem.checkCmemPackMode(cmemConfig.packMode, cmemDdrRdSize);
apiCmem.setupCmemCapture(cmemConfig);

cmemConfig.setupMpg = false; // from now on no more mpg setup
cmemConfig.sampleSize = expStcCnt;

Msg_Printf("Pack mode   = %d bits =
          %d\n",cmemConfig.packMode,cmemDdrRdSize==1?32:64);
Msg_Printf("Sample size = %d\n",cmemConfig.sampleSize);
Msg_Printf("Start addr  = 0x%04x\n",cmemConfig.startAddr);
Msg_Printf("End addr    = 0x%04x\n",cmemConfig.endAddr);
Msg_Printf("DigcapErr   = %d\n",expDigcapErr);

expLastWordSize = (cmemDdrRdSize == 1) ? ((expStcCnt * 32) % 256) :
                  ((expStcCnt * 64) % 256);
expSegSize      = (cmemDdrRdSize == 1) ? ((expStcCnt * 32) / 256) : ((expStcCnt *
                  64) / 256);
if (expLastWordSize) expSegSize++; // account for the last incomplete word

apiPgCtrlT::patgenRunConfigT pgRunCfg;

Msg_Milestone("%s: Setup patgen.\n", __PRETTY_FUNCTION__);
pgRunCfg.srmStartAddr = 0;
pgRunCfg.vmtLoopCnt =0;

pgRunCfg.vmtLoopStartAddr = 0;
pgRunCfg.vmtLoopStopAddr = 0;
pgRunCfg.vmtStartAddr = 0;
pgRunCfg.vmtStopAddr = 0;
pgRunCfg.pgStopEn = M_PG_STOP_EN_OPC;
pgRunCfg.startInLv = 0; //Working with SRM only

apiPgCtrl.loadPatgenRunConfig(pgRunCfg);
apiPgCtrl.setUserFlag(0x0,0xf); /* Clear all four flags. */
dartNmSpc::waitDbFlush();
apiClkCtrl.issueDsync();
Msg_Milestone("%s: Starting Patgen.\n",__PRETTY_FUNCTION__);
apiPgCtrl.startPatgen();
```

```
  //Wait till pattern execution reaches vector 30 where the flag will be set
  if (apiPgCtrl.waitUserFlag(1,1)) {

   Msg_Milestone("%s: FLAG set, Entering
                    KeepAlive.\n",__PRETTY_FUNCTION__);

   Msg_Milestone("Scenario %d keepalive check of data\n",scenario);

     TT_EXPECT(A_TG_CMEM_SEG_HIST_DONE,expDone);
     TT_EXPECT(A__CMEM_CAPT_BUSY,expBusy);
     TT_WRITE(A__CAPT_DONE,0x1); // close the segment
     // Burn some cycles waiting for the segment to close
     for (u_int32 i = 0; i < 40; i++) TT_READ(A__CMEM_CAPT_BUSY);

//------Expectation----------------------------------------------------------//
     TT_EXPECT(A_TG_CMEM_SEG_HIST_DONE, (expDigcapErr ==
                    _UNDERFLOW)?0x0:0x1); // DONE is false if any error occurred
     TT_EXPECT(A__CMEM_CAPT_BUSY,0x0);
     if (expDigcapErr == _NONE) { // don't check segment history if over/underflow
                                       occurred
        TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_LAST_WORD_SIZE,
                    expLastWordSize);
        TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_SIZE_W0,expSegSize);
        TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_SIZE_W1,0x0);
        TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_SIZE_W2,0x0);
     }


     TT_EXPECT(A_TG_CMEM_STC_CNT_W2,0x0);
     TT_EXPECT(A_TG_CMEM_DIGCAP_USR_ERR,expDigcapErr);
     TT_EXPECT(A__LC_ERR,expLcErr);

     switch (scenario) {
        case 0:
          // stv > samples (overflow)
          cmemConfig.sampleSize = 69 /* vectors */ - (mjrSrmOpcVecIndex+4) +
                                       199 - 1;
          if (digcapPlus) {
             expDigcapErr = _OVERFLOW; expDone = 1; expBusy = 0;
          }
          else {
             expDigcapErr = _NONE; expDone = 1; expBusy = 0;
          }
          expStcCnt = cmemConfig.sampleSize + 1;
          break;
```

```
   case 1:
      // stv < samples (underflow)
      cmemConfig.sampleSize = 69 /* vectors */ - (mjrSrmOpcVecIndex+4) +
                                    199 + 1;
      if (digcapPlus) {
         expDigcapErr = _UNDERFLOW; expDone = 0; expBusy = 1;
      }
      else {
         expDigcapErr = _NONE; expDone = 1; expBusy = 0;
      }
      expStcCnt = cmemConfig.sampleSize - 1;
      break;
   case 2:
      // stv = samples
      cmemConfig.sampleSize = 69 /* vectors */ - (mjrSrmOpcVecIndex+4) +
                                    199;
      if (digcapPlus) {
         expDigcapErr = _NONE; expDone = 1; expBusy = 0;
      }
      else {
         expDigcapErr = _NONE; expDone = 1; expBusy = 0;
      }
      expStcCnt = cmemConfig.sampleSize;
      break;
   default:
      Msg_Error("Illegal value (0x%0x) in scenario case statement\n",scenario);
}
expLcErr = (expDigcapErr) ? 1 : 0;

// Reconfigure the cmem storage
cmemConfig.startAddr  = Random.getInRange(0x0,(MSB_Address-0x1FF)) &
                        (~0x7);
cmemConfig.endAddr    = cmemConfig.startAddr + 0x1FF;
cmemConfig.closeSeg   = true;

// Set the starting capture address (lower 3 bits ignored by hw)
TT_WRITE(A__SEG_START_ADR_W0,(cmemConfig.startAddr >>  0)
            & 0xffff);
TT_WRITE(A__SEG_START_ADR_W1,(cmemConfig.startAddr >>
            16) & 0xffff);
// Set the end capture address (lower 3 bits ignored by hw)
TT_WRITE(A__CMEM_END_ADR_W0,(cmemConfig.endAddr >>  0)
            & 0xffff);
TT_WRITE(A__CMEM_END_ADR_W1,(cmemConfig.endAddr >> 16)
            & 0xffff);
```

```
        // Set the number of samples in the segment
        TT_WRITE(A__SEG_SAMPLE_SIZE_W0,(cmemConfig.sampleSize
                >>  0) & 0xffff);
        TT_WRITE(A__SEG_SAMPLE_SIZE_W1,(cmemConfig.sampleSize
                >> 16) & 0xffff);
        // Make sure that a completion of burst Pin CMEM buffer is flushed to dram

TT_WRITE(A__CMEM_FLUSH_RANK_OFF_EN,cmemConfig.closeSeg);
        cmemConfig.sampleSize = expStcCnt;

        Msg_Printf("Pack mode   = %d bits
          %d\n",cmemConfig.packMode,cmemDdrRdSize==1?32:64);
        Msg_Printf("Sample size = %d\n",cmemConfig.sampleSize);
        Msg_Printf("Start addr  = 0x%04x\n",cmemConfig.startAddr);
        Msg_Printf("End addr    = 0x%04x\n",cmemConfig.endAddr);
        Msg_Printf("DigcapErr   = %d\n",expDigcapErr);


        TT_WRITE(A__CMEM_INIT,M__CMEM_INIT_ALL);
for (u_int32 i = 0; i < 40; i++) TT_READ(A__CMEM_CAPT_BUSY);
                // stall to  let cmem_init complete
      } else {
        Msg_Error("%s: Could not retrieved User Flag.\n",__PRETTY_FUNCTION__);
      }

      expLastWordSize = (cmemDdrRdSize == 1) ? ((expStcCnt * 32) % 256) :
                        ((expStcCnt * 64) % 256) ;
      expSegSize      =  (cmemDdrRdSize == 1) ? ((expStcCnt * 32) / 256) :
                        ((expStcCnt * 64) / 256);
      if (expLastWordSize) expSegSize++; // account for the last incomplete word

      // Once observed as clear, issue command to clear the flag.
      Msg_Milestone("%s: Clearing User Flag.\n",__PRETTY_FUNCTION__);
      apiPgCtrl.setUserFlag(0x0,0xf);
      dartNmSpc::waitDbFlush();
      apiClkCtrl.issueDsync();

if (apiPgCtrl.waitPatgenDone()) {
        Msg_Milestone("%s: Successfully completed
burst.\n",__PRETTY_FUNCTION__);
      } else {
        Msg_Error("%s: Did not complete burst properly.\n",__PRETTY_FUNCTION__);
      }

      Msg_Milestone("Scenario %d end of pattern check of data\n",scenario);
```

```
      // Burn some cycles waiting for the segment to close
      for (u_int32 i = 0; i < 20; i++) TT_READ(A__CMEM_CAPT_BUSY);
      // Now check to make sure all is as expected
      TT_EXPECT(A_TG_CMEM_SEG_HIST_DONE, (expDigcapErr ==
_UNDERFLOW)?0x0:0x1); // DONE is false if any error occurred
      TT_EXPECT(A__CMEM_CAPT_BUSY,0x0);
      if (expDigcapErr == _NONE) { // don't check segment history if over/underflow
occurred
         TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_LAST_WORD_SIZE,
expLastWordSize);
         TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_SIZE_W0,expSegSize);
         TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_SIZE_W1,0x0);
         TT_EXPECT(A_TG_CMEM_SEG_HIST_SEG_SIZE_W2,0x0);
      }
      TT_EXPECT(A__CMEM_STC_CNT_W0,expStcCnt << 4);
      TT_EXPECT(A__CMEM_STC_CNT_W1,0x0);
      TT_EXPECT(A_TG_CMEM_STC_CNT_W2,0x0);
      TT_EXPECT(A_TG_CMEM_DIGCAP_USR_ERR,expDigcapErr);
      TT_EXPECT(A__LC_ERR,expLcErr);
   }

   return 0;
   }
```

[3] Test plan and testcase of CMEM segement Condition ERROR

- Test Plan

TEST ID: cmem_digcap_usr_err

STATUS: OPEN

REVIEWED: TESTPLAN

SCHEDULE:

ASSIGNED TO: soriah

REVIEWED BY:

LAB PORTABLE: yes

OPEN ISSUES:

TEST STRATEGY:

- To verify masked/unmasked digcap user error and lc_err/a_cm_err on different combination of invert readback enable and error enable for following scenarios:
- To verify 9th bit of A_CM_RESET or LC_RESET reset the masked/unmasked digcap user error.

1) To run a pattern having same no. of stv as programmed in sample size register to verify that no sample count error is generated.
2) To run a pattern having more no. of stv than programmed in sample size register to verify that sample count overflow error has occured.
3) To run a pattern having less no. of stv than programmed in sample size register to verify that sample count underflow error has occured.

REFERENCE DOCS:
CMEM FIS

FEATURES TESTED:

FEAT _3101
FEAT _3112

CORNER CASES:

KEY TECHNICAL POINTS:

1) If no. of stv programmed in pattern is equal to
SEG_SAMPLE_SIZE_W0/W1
no error will be generated
2) If no. of stv programmed in pattern is more than
SEG_SAMPLE_SIZE_W0/W1
then sample count overflow error will be generated.
3) If no. of stv programmed in pattern is less than
SEG_SAMPLE_SIZE_W0/W1
then sample count underflow error will be generated.
5) when the error enable is zero, that error makes no contribution
- neither a 0 nor a 1 - to a_cm_err or lc_err, so both
of these errors readback 0 regardless of invert readback

TEST ASSUMPTIONS:

VERIFICATION AIDS:

RANDOMIZATIONS:

PROCEDURE:

Test Prerequisites:

A. run map (patlist)

Test Sequence:

- configure DigCap:
. capture mode = digcap
. packing mode = B_TG_CMEM_DATA_PCK_SITE_0
. closeSeg = 1
. start address = random value within legal address range
. end address = start address + 0xFF
(Legal address range will be determind by the DRAM size and make sure end addres
is greater than start address)

- configure pattern opcode and vector data (see cmem_digcap_usr_err.atp file):
. major count = 64
. keep intervening cycles with and without STV_LFVM. Makesure there should be 32
times STV_LFVM on.
- Loop : 0 to 2
0 : configure segment sample size = number of STV_LFVM
1 : configure segment sample size < number of STV_LFVM
2 : configure segment sample size > number of STV_LFVM
- verify that upper 15 bits of a__lc_err and a_cm_err are unchanged

- apply 9th bit of A_CM_RESET or A__LC_RESET to be 0 and check that digcap user error should not be affected
- apply 9th bit of A_CM_RESET or A__LC_RESET to be 1 and check that digcap user error should be zero

- End of Loop(0 to 2)

pseudo_code_1:

// Loop 0-3 to generate different combination of user error enable and invert Readback enable

```
for(u_int32 indx=0; indx<4; indx++){
    switch(indx){
        case 0:
            Loop 0:  a_cm_err[0]          = 0, a__lc_err[0]      = 0
             Loop 1: stv overflow err en   = 0, stv underflow err en   = 0
              Loop 2: stv overflow err en   = 0, stv underflow err en   = 0
                   a_cm_err[0]          = 0, a__lc_err[0]      = 0
        break;

        case 1: inv_rdbk_en = 0, err_en = 1
              Loop 0: stv overflow err en   = 1, stv underflow err en   = 1
              Loop 1: stv overflow err en   = 1, stv underflow err en   = 0
              Loop 2: stv overflow err en   = 0, stv underflow err en   = 1
        break;
        case 2: inv_rdbk_en = 1, err_en = 0
              Loop 0: stv overflow err en   = 0, stv underflow err en   = 0
              Loop 1: stv overflow err en   = 0, stv underflow err en   = 0
              Loop 2: stv overflow err en   = 0, stv underflow err en   = 0
        break;
        case 3: inv_rdbk_en = 1, err_en = 1
              Loop 0: stv overflow err en   = 1, stv underflow err en   = 1
              Loop 1: stv overflow err en   = 1, stv underflow err en   = 0
              Loop 2: stv overflow err en   = 0, stv underflow err en   = 1
        break;
    }
}
```

PASS/FAIL CRITERIA:

Test shouts for any mismatch of digcap user errors or a__lc_err or a_cm_err.

- Test Case

```
//---------File Include section----------------------------------------------------------//
#include "dartTestCommon.h"
#include "apiCmnSetup.h"
#include "apiCmem.h"
#include "mpgCommon.h"
#include "apiMpg.h"
#include "apiMapSupport.h"
#include "bitStructData.h"
//--------Test case start ---------------------------------------------------------------//

extern "C" int cmem_digcap_usr_err(void){
    //----Local veriable declaration ------------------------------------------------//
    u_int32 cmem_feat_en = TT_READ(A_TG_CMEM_FEAT_EN);
    Msg_Milestone("cmem_feat_en = %d \n", cmem_feat_en);

   // get DRAM size in Mb and save max address of DRAM
   // Multiply by 2 because there are 2 drams we store to and then
   // multiple by 1M, but since we store 32 bits at each address,
   // divide by 32. The end result is (getDramSize() << 17) - 1
   u_int32 MSB_Address = (apiMapSupport.getDramSize() << 17) - 1;
   Msg_Printf("MSB_Address = 0x%x DRAM size = 0x%x \n", MSB_Address,
               apiMapSupport.getDramSize());

   //- configure DigCap:
   apiCmemT::ConfigT cmemConfig;
   cmemConfig.captMode     = B__CAPT_MDE_DIGCAP;
   cmemConfig.packMode     = B_TG_CMEM_DATA_PCK_SITE_0;
   cmemConfig.startAddr    = Random.getInRange(0x0,(MSB_Address-0xFF)) &
                                                    (~0x7);
   cmemConfig.endAddr      = cmemConfig.startAddr + 0xFF;
   cmemConfig.closeSeg     = true;

   apiCmem.setupCmemCapture(cmemConfig);

   //Local Variables:
   u_int32 total_stv = 32;// make sure cmem_digcap_usr_err.atp is for total 32 stv on.

   //- Loop : 0 to 2
   // Loop scenario 0 : No error
   // Loop scenario 1 : Overflow error
   // Loop scenario 2 : Underflow error
   for(u_int32 scenario=0; scenario<3; scenario++){
```

```
// Loop lcOrCmLoop 0 : check a_temp_lc_reset
// Loop lcOrCmLoop 1 : check a_cm_reset
for(u_int32 lcOrCmLoop=0; lcOrCmLoop<2; lcOrCmLoop++){
 u_int32 segSampleSize=0;

 switch(scenario){
      case 0: segSampleSize = total_stv; // no error
            break;
      case 1: segSampleSize = Random.getInRange(0, total_stv-1);// overflow error
            break;
      case 2: segSampleSize = Random.getInRange(total_stv+1,
              B__SEG_SAMPLE_SIZE);// underflow error
            break;
 }

 Msg_Printf("== scenario=%d segSampleSize=%d \n", scenario, segSampleSize);


 // set segment sample size
 cmemConfig.sampleSize = segSampleSize;
 apiCmem.setupCmemCapture(cmemConfig);

 // disable all error enables and invert readback enable
 apiMapSupport.disableAllErrEn();
 TT_WRITE(A_GL_INV_RDBK_EN, 0);

 // apply init
 apiCmem.resetCmem();

 // start pattern
 PG_ExecATP_name("cmem_digcap_usr_err.atp","err");

 // run cmem expect model and print to log
 // Do not call expect model under digcap error condtion. As cmem data and
seghist fields are not gauranteed.
 if ((scenario == 0) || (cmem_feat_en == 0x0)) { apiCmem.execModel(true); }

 // local variables for expectation logic:
 u_int32 expDigErr;
 u_int32 expDigErrMsk;
 u_int32 rdbkDigErr;
 u_int32 rdbkDigErrMsk;
 u_int32 rdbkCmErr;
 u_int32 rdbkLcErr;
 u_int32 inv_rdbk_en = 0;
 u_int32 usr_err_en = 0;
```

```
    // Loop 0-3 to generate different combination of user error enable and invert
readback enable
    for(u_int32 indx=0; indx<4; indx++){

      switch(indx){
        case 0:// inv_rdbk_en = 0, usr_err_en = 0
             inv_rdbk_en = 0;
             usr_err_en = 0;
        break;
        case 1:// inv_rdbk_en = 0, usr_err_en = 1
             inv_rdbk_en = 0;
             usr_err_en = (scenario==0)?0x3:
                       (scenario==1)?0x1:0x2;
        break;
        case 2:// inv_rdbk_en = 1, usr_err_en = 0
             inv_rdbk_en = 0xFFFF;
             usr_err_en = 0;
        break;
        case 3:// inv_rdbk_en = 1, usr_err_en = 1
             inv_rdbk_en = 0xFFFF;
             usr_err_en = (scenario==0)?0x3:
                       (scenario==1)?0x1:0x2;
        break;
      }
      TT_WRITE(A_GL_INV_RDBK_EN, inv_rdbk_en);
      TT_WRITE(A_TG_CMEM_DIGCAP_USR_ERR_EN, usr_err_en);

      // expectation logic:
      // Note : when the error enable is zero, that error makes no contribution
      //       - neither a 0 nor a 1 - to a_cm_err or a__lc_err

      if(cmem_feat_en == 0x0){
          if(inv_rdbk_en){
            expDigErr = 0xFFFF;
            expDigErrMsk = 0xFFFF;
            expCmOrLcErr = (usr_err_en) ? 0x1 : 0x0;
          }else{
            expDigErr = 0x0;
            expDigErrMsk = 0x0;
            expCmOrLcErr = 0x0;
          }
      }else{
```

```
if(total_stv == segSampleSize){ // no error
  if(inv_rdbk_en){
    expDigErr = 0xFFFF;
    if(usr_err_en){
      expDigErrMsk = 0xFFFF;
      expCmOrLcErr = 1;
    }else{
      expDigErrMsk = 0xFFFF;
      expCmOrLcErr = 0;
    }
  }else{
    expDigErr = 0;
    if(usr_err_en){
      expDigErrMsk = 0;
      expCmOrLcErr = 0;
    }else{
      expDigErrMsk = 0;
      expCmOrLcErr = 0;
    }
  }
}

if(total_stv > segSampleSize){ // overflow error
  if(inv_rdbk_en){
    expDigErr = 0xFFFE;
    if(usr_err_en){
      expDigErrMsk = 0xFFFE;
      expCmOrLcErr = 0;
    }else{
      expDigErrMsk = 0xFFFF;
      expCmOrLcErr = 0;
    }
  }else{
    expDigErr = 1;
    if(usr_err_en){
      expDigErrMsk = 1;
      expCmOrLcErr = 1;
    }else{
      expDigErrMsk = 0;
      expCmOrLcErr = 0;
    }
  }
}
```

```
  if(total_stv < segSampleSize){ // underflow error
    if(inv_rdbk_en){
      expDigErr = 0xFFFD;
      if(usr_err_en){
        expDigErrMsk = 0xFFFD;
        expCmOrLcErr = 0;
      }else{
        expDigErrMsk = 0xFFFF;
        expCmOrLcErr = 0;
      }
    }else{
      expDigErr = 2;
      if(usr_err_en){
        expDigErrMsk = 2;
        expCmOrLcErr = 1;
      }else{
        expDigErrMsk = 0;
        expCmOrLcErr = 0;
      }
    }
  }

}// end if..else loop

//------- checker-----------------------------------------------------------------//
Msg_Printf(" == scenario=%d indx=%d inv_rdbk_en=%d usr_err_en=%d \n",
                scenario, indx, inv_rdbk_en, usr_err_en);
rdbkDigErr       = TT_READ(A_TG_CMEM_DIGCAP_USR_ERR);
rdbkDigErrMsk = TT_READ(A_TG_CMEM_DIGCAP_USR_ERR_MSKD);

rdbkCmErr       = TT_READ(A_CM_ERR);
rdbkLcErr        = TT_READ(A__LC_ERR);

if(rdbkDigErr != expDigErr){
   Msg_Error("Mismatch of digcap usr error ACT:0x%X EXP:0x%X \n",
                    rdbkDigErr, expDigErr);
}

if(rdbkDigErrMsk != expDigErrMsk){
   Msg_Error("Mismatch of digcap mask usr error ACT:0x%X EXP:0x%X \n",
                    rdbkDigErrMsk, expDigErrMsk);
}

if(rdbkCmErr != expCmOrLcErr){
   Msg_Error("Mismatch of A_CM_ERR ACT:0x%X EXP:0x%X \n", rdbkCmErr,
```

```
                        expCmOrLcErr);
        }
        if(rdbkLcErr != expCmOrLcErr){
           Msg_Error ("Mismatch of A__LC_ERR ACT:0x%X EXP:0x%X \n",
                        rdbkLcErr, expCmOrLcErr);
        }
        // check that writing zero on A__LC_RESET or A_CM_RESET has
          no effect on digcap use errors
        TT_WRITE (A__LC_RESET, 0x0);
        TT_WRITE (A_CM_RESET, 0x0);

        rdbkDigErr = TT_READ(A_TG_CMEM_DIGCAP_USR_ERR);
        rdbkDigErrMsk = TT_READ(A_TG_CMEM_DIGCAP_USR_ERR_MSKD);
        if(rdbkDigErr != expDigErr){
           Msg_Error ("Mismatch of digcap usr error ACT:0x%X EXP:0x%X \n",
                        rdbkDigErr, expDigErr);
        }
        if(rdbkDigErrMsk != expDigErrMsk){
           Msg_Error ("Mismatch of digcap mask usr error ACT:0x%X EXP:0x%X \n",
                        rdbkDigErrMsk, expDigErrMsk);
        }

      }//End indx loop...x
      // set digcap user error enable to 1 and invert readback enable to 0
      TT_WRITE (A_GL_INV_RDBK_EN, 0);
      TT_WRITE (A_TG_CMEM_DIGCAP_USR_ERR_EN, 0x3);

      // set 9th bit of a_temp_lc_reset or a_cm_reset to 1
      if(lcOrCmLoop == 0){
        TT_WRITE (A__LC_RESET, 0x0200);
      } else {
        TT_WRITE (A_CM_RESET, 0x0200);
      }
      //------Expectation----------------------------------------------------------//
      // expect masked/unmasked digcap user error should be zero
      TT_EXPECT (A_TG_CMEM_DIGCAP_USR_ERR, 0x0);
      TT_EXPECT (A_TG_CMEM_DIGCAP_USR_ERR_MSKD, 0x0);

    }//End lcOrCmLoop...
   }//End scenario loop...
   return(0);
}
```

REFERENCES

[1] EInfochips, Capture Memory Functional Interface Specifications Manual.

[2] EInfochips, Capture Memory Verification Environment Reference Manual.

[3] EInfochips, SCIF methodology Functional Interface Specifications Manual.

[4] EInfochips, Memory Pattern Generator Verification Environment Reference Manual.

[5] IBM, SCM271 Essentials of ClearCase v7.0.

[6] Object Oriented Programming with C++, By E balagurusamy .second Edition.

[7] WRITING TESTBENCHES Functional Verification of HDL Models By Janick Bergeron

[8] Andrew Pizali, P. D. (2004), Functional Verification Coverage Measurement and Analysis, Kluwer Academic Publishers (Boston).

[9] Andreas Meyer, P. D.(2003), Principles of Functional Verification, Elsevier Science(USA).