# “Verification Of A Chip Block At Module And Board Bringup Level”

A Major Project Report

**Submitted in Partial Fulfillment of the Requirements for the Degree of**

***Master of Technology***

**IN**

**ELECTRONICS & COMMUNICATION ENGG.**
**(VLSI Design)**

**By**

**Ms. Heena Shah**

**Department of Electronics & Communication Engineering**
**Institute of Technology**
**Nirma University of Science & Technology**
**Ahmedabad-382481**

# “Verification Of A Chip Block At Module And Board Bringup Level”

A Major Project Report

**Submitted in Partial Fulfillment of the Requirements for the Degree of**

*Master of Technology*

**IN**

**ELECTRONICS & COMMUNICATION ENGG.**
**(VLSI Design)**

**By**

**Ms. Heena Shah**

Under the Guidance of


| | |
|---|---|
| Mr. Darshan Sheth | Mrs. Usha Mehta |
| Project Leader - ASIC | Asst. Professor – EC |
| eInfochips, | NIT, Nirma University, |
| Ahmedabad. | Ahmedabad. |

**Department of Electronics & Communication Engineering**
**Institute of Technology**
**Nirma University of Science & Technology**
**Ahmedabad-382481**

## *Certificate*

This is to certify that the Major Project entitled **"Verification Of A Chip Block At Module And Board Bringup Level"** submitted by **Ms. Heena Shah (05MEC014)**, towards the partial fulfillment of the requirements for the Semester-III-IV of the degree of **Master Of Technology** in **Electronics & Communication (VLSI Design)** of **Nirma University of Science & Technology** is the record of work carried out by her under our supervision and guidance.

The work submitted has in our opinion reached a level required for being accepted for the examination. The results embodied in the major 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.

**Prof. Usha Mehta**
**E.C. Department**
**Project Guide**

**Prof. A.S. Ranade**
**H.O.D.**
**E.C. Department**
**Nirma University**
**Ahmedabad**

**Prof. A.B. Patel**
**Director**
**E.C. Department**
**Nirma University**
**Ahmedabad**

# ACKNOWLEDGEMENT


***Sometimes our light goes out,***
***but is blown into flame by another human being.***
***I owe deepest thanks to those***
***who have rekindled this light.***

*“**Verification Of A Chip Block At Module And Board Bringup Level**” is my Major Project for M.Tech (VLSI Design). I take this opportunity to express my gratitude to all those people who have been instrumental in making my project successful.*

*I am grateful to eInfochips, Ahmedabad for providing me the opportunity to do my M.Tech Project in its premises.*

*I extend my sincere thanks to Mr. Darshan Sheth, Project Leader – ASIC Division, eInfochips, Ahmedabad, for his constant guidance and encouragement in all possible ways.*

*I am grateful to Mr. Nilesh Ranpura, Project Manager - ASIC Division, eInfochips, Ahmedabad, for providing all necessary resources and his constant support.*

*I am thankful to Prof. Usha Mehta, Asst. Professor, EC Dept, NIT, Nirma University, Ahmedabad, for her guidance.*

*I extend my thanks to Prof. N.M. Devashrayee, Head and Course Coordinator – M.Tech (VLSI Design), NIT, Nirma University, Ahmedabad for his always unconditional support and encouragement.*

*Last but not the least, I am thankful to the almighty who blessed me with the zeal to work hard.*



*Heena Shah*
*05MEC014*
*M.Tech (VLSI Design)-!V*
*NIT, Nirma University,*
*Ahmedabad.*

# ABSTRACT

*With 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 "The Verification Of A Chip Block At Module and Board Bringup Level" concentrates on two types of Functional Verification at different stages in the Chip Design Flow. During Design phase, verification is done at low abstraction level, concentrating on the core functionality of the module. The inputs to the module are forced through the testbench and its interfaces are not looked upon. This is the functional verification of chip at module level and is done at the RTL design phase. After the RTL is been finalized after fixing all the bugs, it is send to the fabrication unit. The first chip that will tap out from the fabrication unit has to be tested on the operational board for it. The verification engineers work on the development of simulation platform for verifying each module of the chip for its functionality and dataflow within and external to the chip. The test cases developed for this, once pass on the bringup simulation platform, will be executed by the first chip on the board. The abstraction level is high and concentration is on the interfaces.*

*The modules called MI (Mate Ingress) and ME (Mate Egress) are verified for functional verification at module level. The Verification Environment Serving as Platform for simulation is developed with C++ and Verilog. The Test Bench is in Verilog and instantiates two modules Mate Ingress and Mate Egress that are interfaced internally. The module is thoroughly verified which different combinations of input streams generated by Standard SONET frame generator. The same Standard SONET frame analyzer is used to verify the output from the Design under Test. The test cases required to exercise the module for its core functionality are written and simulated as per the test plan. As the verification progressed, different issues related to Verification Environment, Test Bench, Test Case and RTL arouses. The RTL issues are filed as bugs and get resolved by the designers. This results in updation of all the verification components as per requirement. After passing all the test cases, the code coverage is done and based on it new test scenarios are added.*

*The module called EC (Enhanced Co-processor) is verified for functional verification at board bringup level. This module is a co-processor to a Video Processor and performs complex computational tasks for it. The flow starts with preparation of Test Plan for the bringup verification. The next step is to develop the bringup verification environment. This platform for simulation considers the complete data path for the module under test. It mimics the actual scenario the module will face when the chip is operational on the board. The development of test cases as per the test plan follows. Test cases are written to test each instruction in each operational mode along the complete data path. The chip on board exercises the pass test cases.*

# LIST OF FIGURES


| Sr. No. | Fig. No. | Figure Title | Page No. |
|---|---|---|---|
| 1 | 1.1 | Functional Verification Reconvergent Model | 2 |
| 2 | 1.2 | The Chip On Board | 15 |
| 3 | 2.1 | Verification Flow | 22 |
| 4 | 3.1 | Optical Network | 27 |
| 5 | 3.2 | SONET End To End Connection | 28 |
| 6 | 3.3 | Basic SONET frame | 28 |
| 7 | 3.4 | SONET Multiplexing Structure | 30 |
| 8 | 3.5 | SDH Multiplexing Structure | 31 |
| 9 | 4.1 | The C++ Environment | 33 |
| 10 | 4.2 | Complete VE For MIME Module | 37 |
| 11 | 4.3 | The Simulation Flow | 39 |
| 12 | 5.1 | MIME TestBench Configuration | 44 |
| 13 | 5.2 | Coverage Log's Snap Shot | 52 |
| 14 | 6.1 | Block Diagram Of Chip On Board | 54 |
| 15 | 6.2 | H.264/AVC Technologies & Applications | 56 |
| 16 | 7.1 | Bringup Code Generation For EC | 63 |
| 17 | 7.2 | Bringup Env Comp/Data/Control Flow | 66 |
| 18 | 7.3 | Test Case Snap Shot | 73 |
| 19 | 7.4 | Waveform Snap Shot | 77 |

# LIST OF TABLES


| Sr. No. | Fig. No. | Figure Title | Page No. |
|---|---|---|---|
| 1 | 3.1 | SONET/SDH Digital Hierarchy | 30 |
| 2 | 5.1 | Features Of MIME Module With Identified Test Cases | 46 |
| 3 | 5.2 | Bugs Filed With Brief Summary | 48 |
| 4 | 5.3 | New Test Cases identified after Coverage | 49 |
| 5 | 8.1 | Test plan for the EC module | 69 |
| 6 | 8.2 | The EC Test Cases Executed | 71 |
| 7 | 9.1 | EVCD signal dump format | 91 |

# CONTENTS


| Ch No. | Topic | Pg No. |
|---|---|---|
| **1** | **INTRODUCTION TO VERIFICATION** | 1 |
| 1.1 | Introduction | 1 |
| 1.2 | What is Verification? | 1 |
| 1.3 | Types Of Verification | 2 |
| 1.4 | Levels Of Verification | 4 |
| 1.5 | Verification Tools | 5 |
| 1.6 | Functional Verification | 12 |
| 1.7 | BUGS | 14 |
| 1.8 | The Chip Bringup Verification | 14 |
| 1.9 | The Verification Methodology | 16 |
| 1.10 | High Level Verification Languages | 17 |
| | | |
| **2** | **THE VERIFICATION METHODOLOGY & FLOW** | 19 |
| 2.1 | Introduction | 19 |
| 2.2 | The Verification Methodology | 19 |
| 2.3 | The Verification Flow | 20 |
| 2.4 | Designer – Verification Engineer Interaction | 25 |
| | | |
| **3** | **THE MIME MODULE FOR FUNCTIONAL VERIFICATION** | 26 |
| 3.1 | Introduction | 26 |
| 3.2 | The Protocols/Standards | 26 |
| 3.3 | MIME Module | 31 |
| | | |
| **4** | **THE MODULE LEVEL VERIFICATION ENVIRONMENT** | 33 |
| 4.1 | Introduction | 33 |
| 4.2 | The C++ Environment | 33 |
| 4.3 | The Verilog Environment | 34 |
| 4.4 | The TestBench | 37 |
| 4.5 | The Simulation Flow | 38 |
| | | |
| **5** | **MODULE LEVEL VERIFICATION FLOW** | 41 |
| 5.1 | Introduction | 41 |
| 5.2 | Languages Used | 41 |
| 5.3 | OS & S/W | 42 |
| 5.4 | EDA Tools Used | 43 |
| 5.5 | Identification of features | 43 |
| 5.6 | Developing the test bench | 44 |
| 5.7 | Prioritization of features | 45 |
| 5.8 | Grouping of test cases | 46 |
| 5.9 | Environment settings | 46 |
| 5.10 | Debugging the test cases | 47 |

| | | |
|---|---|---|
| 5.11 | Bug filing | 47 |
| 5.12 | Looping Structure | 48 |
| 5.13 | Regression | 48 |
| 5.14 | Coverage | 49 |
| 5.15 | Documentation | 52 |
| | | |
| **6** | **THE EC MODULE FOR BOARD BRINGUP VERIFICATION** | 53 |
| 6.1 | Introduction | 53 |
| 6.2 | The Chip On Board | 53 |
| 6.3 | The Protocols/Standards | 55 |
| 6.4 | The Enhanced Co-processor Module | 60 |
| | | |
| **7** | **THE BRINGUP VERIFICATION ENVIRONMENT** | 62 |
| 7.1 | Introduction | 62 |
| 7.2 | Characteristics Of Bringup VE | 62 |
| 7.3 | The Bringup Code Generation for EC | 63 |
| 7.4 | The Data/Control Flow in Bringup VE | 65 |
| 7.5 | Critical Stages in Bringup VE development | 67 |
| 7.6 | Verilog Assertions | 68 |
| | | |
| **8** | **THE BRINGUP VERIFICATION FLOW** | 69 |
| 8.1 | Introduction | 69 |
| 8.2 | The Test Plan | 69 |
| 8.3 | Bringup Verification Environment | 71 |
| 8.4 | Simulating The Test Cases | 71 |
| 8.5 | Instruction Simulation | 75 |
| 8.6 | Debugging Test Cases | 77 |
| 8.7 | Executing Test Cases On Board | 78 |
| 8.8 | Executing Test Cases On Tester Board | 79 |
| | | |
| **9** | **EDA TOOLS, S/W AND LANGUAGES** | 80 |
| 9.1 | Introduction | 80 |
| 9.2 | EDA Tools | 80 |
| 9.3 | Application S/W | 83 |
| 9.4 | Languages | 86 |
| 9.5 | The Verilog Dump File | 89 |
| | | |
| | **Conclusion** | 93 |
| | **Future Scope** | 93 |
| | **References** | 94 |
| | **Abbreviations & Acronyms** | 95 |

# 1

# INTRODUCTION TO VERIFICATION

## 1.1 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.

Today, in the era of multi-million gate ASICs, reusable Intellectual Property (IP), and System On Chip (SOC) designs, verification consumes about 70% of the design effort. The number of verification engineers is usually twice the number of design engineers. When design projects are completed, the code that implements the testbenches makes up to 80% of the total code volume. Verification of design is done at various levels of design phase. Different type of verification is done on the design. The foremost of them is the functional verification that verifies the functionality implemented by the design with respect to the specifications.

## 1.2 What is Verification?

I will start with the IEEE definition of Verification. IEEE defines Verification as ***"Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled."*** Lets first understand the definition. The design which is to be verified implements some specific functionality as per the requirements. Now the aim of verification is to examine that design implements these specified functionality. This is to be done by examination and not mere observation. It has to be supported by some objective evidences.

Verification can be done at various granularity levels. Depending on the features of the design to be verified, the abstraction level is decided. It follows methodology to accomplish it accurately and quickly. It is a parallel process with the design. Verification engineers and design engineers have to interact a lot to get a verified design at the end of the design cycle. Various EDA tools and languages are available for verification. I shall be going through the complete methodology in detail in the later chapters.
Verification Process is conceptually represented using a reconvergence model. It also illustrates what exactly is being verified. The purpose of verification is to ensure that the

result of some transformation is as intended or as expected. This is analogous to what we do daily in our life. e.g. confirming bank transactions with the available balance.

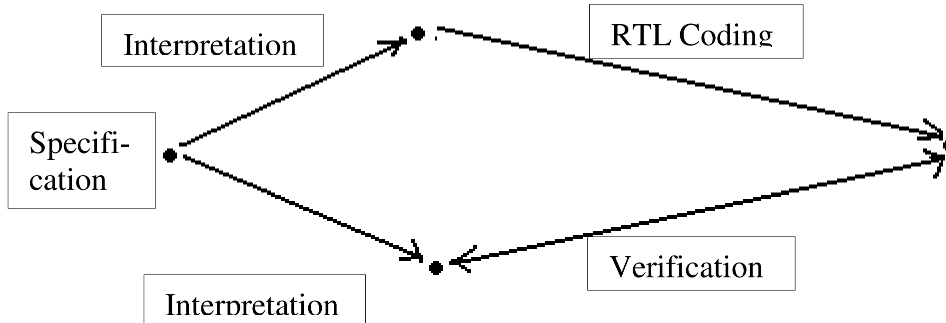



Fig1.1 Functional Verification Reconvergent Model

A design team interprets a written specification document and produces what they believe to be functionally correct synthesizable HDL code. If the same individual performs the verification of the RTL coding that initially required interpretation of a specification, then the common origin is that interpretation and not the specification. In this case, verification verifies designer's interpretation and not the specification. If that interpretation is wrong, then verification will not be able to highlight it. Hence, the process of verification also starts from the specification and verifies the RTL coding against the specifications. Figure is for functional verification. Choosing the common origin and reconvergenence points determines what is being verified. For functional verification, it is the RTL coding verification against specifications.

Currently, verification is on critical path. It is on target of new tools and methodologies. These tools and methodologies attempt to reduce the overall verification time by enabling parallelism of effort, higher level of abstraction and automation. If effort can be parallelized, additional resources can be applied effectively to reduce the total verification time. For this, it is necessary to be able to write and debug testcases in parallel with each other as well as in parallel with the implementation of the design. Higher level of abstraction enables to work fast but this reduces control and hence should be used wisely. The verification process can be at higher level of abstraction by working at the transactions or bus cycle levels instead of dealing with lower level zeros and ones.

### 1.3 Types Of Verification

As explained earlier, in a reconvergence model, the points and path decides the type of verification. Different tools are used for different types of verification. Broadly, verification can be functional verification, formal verification, model checking and testbench generators.

### 1.3.1 Functional verification

Functional verification is to verify the functionality implemented by the design against the specification. This can be done at various granularity level. This depends on how much depth in verification is required based on the deadline. It can show that the design meets the intended specifications but cannot prove that design is free from any discrepancy. More explanation on this is in the later chapters.

### 1.3.2 Formal verification

Establishing properties of hardware or software designs using logic, rather than just testing or informal arguments is formal verification. This involves formal specification of the requirement, formal modeling of the implementation, and precise rules of inference to prove, say, that the implementation satisfies the specification.

**Equivalence Checking** is a type of formal verification. This process mathematically proves that the origin and output are logically equivalent and that the transformation preserves its functionality. It compares two netlist to ensure that some netlist post processing such as scan chain insertion, clock tree synthesis or any manual modification, did not changed the functionality of the circuit. It is also used to verify that the netlist correctly implements the original RTL code. It can be used to see that the synthesizer tool is honest. It can also be used to verify that two RTL descriptions are logically equivalent. Equivalence checking is interested in comparing Boolean and sequential logic functions and not mapping these functions to a specific technology.

**Model Checking** is a recent application of formal verification. It is a method to algorithmically verify formal systems. This is achieved by verifying if the model satisfies a formal specification. The specification is often written as temporal logic formulas. A model-checking tool accepts system requirements or design (called models) and a property (called specification) that the final system is expected to satisfy. The tool then outputs yes if the given model satisfies given specifications and generates a counterexample otherwise. The decision process often uses some form of binary decision diagram (BDD). Here, assertions or characteristics of a design are formally proven or disproven. For example, all state machines in a design could be checked for unreachable or isolated states. Even deadlocks can be detected. The greatest obstacle to model checking is identifying, through interpretation of the design specification, which assertions to prove. Only a subset of identified assertions are feasible to prove.

### 1.3.3 Testbench Generation

Here, there is no reconvergence point. The RTL code is the common origin. With the help of code coverage metrics and the source code under analysis, testbench generators generate testbenches to either increase code coverage or to exercise the design to violate a property.

## 1.4 Levels Of Verification

As per the Rule Of Ten for chip design, the cost of detecting a bug gets multiplied by $10 as the design progresses through increasing abstraction level. It means that it is better to detect a bug at the initial stage of chip design. As the chip design progresses from component level to module level to system level and so on, the cost of detecting that bug increases with a multiple of $10. Hence, Verification becomes most important to keep the cost of complete chip within the budget.

Chip Design is a long, complex and a precise process. The chip design goes through number of different stages where it is looked upon from different abstraction levels. It starts from specifications developed from market serve and ends when the chip is fully tested and reaches its final destination: market. At each step of design, verification follows. At each abstraction level, verification is performed rigorously on the design to confirm its corrective operation.

The verification of the functionality chip being design is normally done from bottom to top. The abstraction level goes on increasing. The level of granularity required for the verification effort is determined by the level of verification. The design is potentially composed of several levels. Verification proceeds with these partitions.

### 1.4.1 Unit Level Verification

This is the lowest level of verification. It works on logical partitions of the design units. They could be relatively small components like FIFO and state machines. They usually do not have an independent specification document to verify against either. This is basically an adhoc verification and designer can perform this himself. If the features to be verified at the unit level require interaction with other units, they have to be reverified at a higher level where the features are fully contained, to ensure that the integration correctly implements them.

### 1.4.2 Module Level Verification

Each chip is designed in terms of various functional modules. These modules are responsible for dedicated functionality of the chip and they interact internally to accomplish the ultimate functionality of the chip. Verification engineers starts with verification of chip RTL at module level. The test patterns applied are both random and deterministic. The module under test is verified from each angle of its functionality as per the specifications. Much of the work is done on the corner cases to identify the bugs in the RTL. This process is simultaneous to the design process and the designer keeps on fixing the bugs as the verification engineer identifies them. The input patterns applied are fired directly on the module only. This level does not verify the interfaces which the module has with other modules.

### 1.4.3 Full Chip Level Verification

Studied reports say that 25% to 40% of bugs are identified at full chip integration level verification. The chip is verified with all its modules integrated. The basic aim is to identify bugs related to the interactions of the modules inside the chip and system level interactions. The chip is verified against its complete specification which was used by the designer. Much of the effort deals with transactions, sequence and dependency of execution within the modules.

### 1.4.4 Board Level Verification

The designed chip is intended for use with other peripheral blocks. It interacts with these blocks to accomplish the application. At board level, the main aim is to concentrate on such system level interactions. This requires, obtaining suitable models for all such components. Third party models are used for this. The third party model is basically a non-synthesizable code, which has the functionality of the component. Hence, the chip is verified as if it is working in a real environment with all its peripheral components.

### 1.4.5 Chip Bringup Verification

This is done to test that the chip was fabricated correctly. Tests are written and simulated to run on the actual chip when it taps out of the fab. These tests are basically to check the core functionality of the chip, its interactions with the peripheral blocks, dataflow within and external to chip and the application of the chip. The development and simulation of these test cases starts with the RTL tap out and deadline is governed by the data of first chip tap out.

## 1.5 Verification Tools

To improve the reliability and efficiency of a process, automation is required. I will be explaining the tools used in state-of-art functional verification environment. These tools aid the verification process and helps to achieve the deadline.

### 1.5.1 Lint Tools

Lint tool identifies common mistakes programmers make. It allows finding the mistakes quickly and easily. It identifies real problems such as mismatched data types between arguments and function calls or mismatch in number of arguments. The source code that is syntactically correct and compiles without any error, may result in error at runtime. Lint tool is helpful here in identifying such problems prior to runtime. Diagnosing the problems at runtime would require a runtime debugger and would take several minutes. Lint tool is a static tool i.e. it does not require and stimulus or any description about the output expected. They perform checks that are entirely static in nature, with expectations built into the lint tool itself.

**Lint tool for Verilog & VHDL**

Verilog is a typeless language. Any value can be assigned to a reg or subprogram argument. This creates problem many times. Linting Verilog code ensures that all data is properly handled without accidentally dropping or adding to it. VHDL is a strong typing language. It does not need linting as much as Verilog. Still common problems in VHDL, like that created using std_logic type can be resolved faster.
Some Lint Tools are:

**Leda** : Leda is a code purification tool for designers using the Verilog and VHDL Hardware Description Language (HDL). Leda is uniquely qualified to analyze HDL code pre-synthesis and pre-simulation and is totally compatible with all popular synthesis and simulation tools and flows. By automating more than 500 design checks for language syntax, semantics and questionable synthesis/simulation constructs, Leda detects common as well as subtle and hard-to-find code defects, thus freeing designers to focus on the art of design.
**HDLint** : A power full linting tool for VHDL and Verilog.
**nLint** : nLint is a comprehensive HDL design rule checker fully integrated with the Debussy debugging system.
**SureLint** : Designers need tools to analyze and debug their designs before integrating with the rest of the project. SureLint offers finite state machine (FSM) analysis, race detection, and many additional checks the most complete lint tool on the market.

### 1.5.2 Simulators

Simulators are most common and familiar verification tools. They are named such because their role is limited to approximating reality. They attempt to create an artificial universe that mimics the future real designs. This lets the designers interact with the design before it is manufactured and correct flaws and problems earlier. Many physical characteristics are simplified to ease the simulation task. e.g. A digital simulator assumes that the only possible values for a signal are 0,1 unknown and high impedance. However, in reality, value of a signal is a continuous function of voltage and current across copper wire and can have infinite number of possible values. Within that simplified universe, simulator executes the description of the design.

Simulators are not static tools. They require a facsimile of the environment in which the design will find itself. This is a testbench and it needs to provide inputs, so that design can emulate the design's response based on its description. Simulators can be event driven or cycle driven. Event driven simulators executes the model only when there is any transition in the input provided. A cycle based simulator works on the simulator clock. Here, all timing and delay information is lost. They assume the design meets all the setup and hold requirements.

**Co-Simulators & Single Kernel Simulators**

Co-Simulators can provide simulation based on cycle and event, VHDL and Verilog, HDL and C or digital and analog. During co simulation, all simulators involved progress along the time axis in lock step. But now co-simulators which progress in time wrap synchronization have evolved where some simulators are allowed to move ahead of the others.

The biggest hurdle here is the communication overhead between simulators. Whenever a signal generated within a simulator is required as an input by another, the current value of that signal, as well as the timing information of any change in that value, must be communicated. This communication involves translation of event from one simulator into an equivalent event in another simulator.

**Modelsim** was the first to put the award winning Single Kernel Simulator (SKS) technology in the hands of design engineers, enabling transparent mixing of VHDL, Verilog and now SystemC in one design, with a common intuitive graphical interface for development and debug at any level, regardless of the language. Different languages are compiled into a single internal representation or machine code and the simulation is performed using a single simulation engine.
Some Simulators are:

**Verilog-XL** : This is the most standard simulator in the market, as this is the sign off simulator.
**NCVerilog** : This is the compiled simulator which works as fast as VCS, and still maintains the sign off capabilities of Verilog-XL. This simulator is good when it comes to gate level simulations.
**VCS** : This is worlds fastest simulator, this is also a compiled simulator like NCverilog. This simulator is faster when it comes to RTL simulation. Few more things about this simulator are direct C kernel interface, Covermeter code coverage embedded, better integration with VERA and other Synopsys tools.
**Finsim** : This is 100% compatible simulator with Verilog-XL, runs on Linux, Windows and Solaris. This is compiled simulator like VCS and NCVerilog, but slower then VCS and NCVerilog.
**Modelsim** : This is most popular simulator from Mentor Graphics. It has got very good debugger, it supports SystemC, Verilog, VHDL and SystemVerilog.

### 1.5.3 Third Party Models

In a board level design, along with your design, there will be some devices purchased from a third party. These can be PLDs, FPGAs, memories, etc. To ensure that the design will work functionally correct with these devices, it is necessary to include models for all the parts in the simulation. Models for these devices can be supplied from the vendor from whom you purchased the device or it can be from some third party. There are several service providers of models for standard SSI and LSI components. These models are non-synthesizable codes. Such models are generic and hence are verified to a greater degree of confidence. Hence, such models can be trusted well.

### Hardware Modelers

When any design is very new or complex, it is difficult to find a model for it as no provider had time to develop it. For such conditions, Hardware Modeler is a solution. This is a small box that connects to your network. A real physical chip that needs to be simulated is plugged in. During simulation, the hardware modeler communicates with your simulators through special interface to supply inputs from the simulator to the device, and then sends the sampled output values from the device back to the simulation. This does not perform timing checks nor does it accurately reflect the output delays. But they greatly speed up the board and system level simulation as the execution occurs on real hardware that is most fastest.

### 1.5.4 Waveform Viewers

These are the most common verification tools used in conjunction with simulators. They let you visualize the transitions of multiple signals over time, and their relations with other transitions. Zooming of any particular time interval allows to measure time between two transitions. It can display data in various formats. They provide filters to filter out unwanted signals and see only the necessary signals in case of any complex design. You can locate time of any particular transition to locate any bug. They can be used interactively during the simulation, but more importantly offline, after the simulation has completed. They can playback the events that occurred during the simulation that were recorded in same trace file. Viewing waveforms as a post-processing step lets you quickly browse through a simulation that can take hours to run.

Some waveform viewers can compare two sets of waveforms. One set is presumed to be a golden reference, while other is verified for any discrepancy. The comparator visually flags or highlights any discrepancy found. Waveform viewers are advisable for debugging purpose. Do not use it to determine whether a test case passes or fails.
Some Waveform Viewers are:

**nAnalyzer** : This is from Novas Software which provides a single environment for analysis and debugging critical design implementation issues.
**Dinotrace** : This is a free Verilog VCD waveform viewer for an X -11. It understands Verilog value change jump dumps, ASCII and other trace formats. It allows placing cursors, highlighting and searching signals.
**SimVision** : The SimVision analysis environment is a unified graphical debugging environment for Verilog-XL, NC-Verilog, NC-VHDL, and NC-Sim from Cadence. It can run in Simulation mode or Post processing mode.

### 1.5.5 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 covered 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.

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.

**Statement 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.

**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.

**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 elses 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. Ideally, the code coverage tool you are using will recognise this and not complain about it, but unfortunately we do not live in an ideal world. And anyway, solving this problem in the general case requires a solution to the halting problem, and I couldn't find a module on CPAN for that.

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.

**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.

**FSM Coverage**

inite 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.

**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 most simple 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, that 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.

## 1.6 Functional Verification

I will be concentrating more on functional verification as my project is functional verification of an OPTICAL CHIP design. The main purpose of functional verification is to ensure that a design implements intended functionality. The starting point for functional verification is specifications. The verification engineer interprets the specification and verifies whether the design coincides with the specifications or not. Functional verification, as a process can show that a design meets the intent of its specifications, but cannot prove it. One can easily prove that a design does not implement a desired functionality by identifying just one discrepancy. But, the converse is not. No one can prove that there are no discrepancies. Functional verification can be accomplished using three complementary but different approaches: black box, white box and grey box.

### 1.6.1 Black Box Functional Verification

As the name suggests, the design to be verified is looked upon as a black box. i.e. only the interface is known for the design. The internal information of the design is not know. The intention is to verify that the design generates required output for a specific input applied. With a black box approach, the functional verification must be performed without any knowledge of the actual implementation of the design. All verification must be accomplished through the available interfaces, without direct access to the internal state of the design, with knowledge of its structure and implementation. This method suffers from an obvious lack of visibility and controllability. It is difficult to set up interesting state combination or to isolate some functionality. It is equally difficult to observe the response from the input and locate the source of the problem.

The advantage of black box approach is that it does not depend on any specific implementation. Whether the design is implemented in a single ASIC, multiple FPGAs or board, is irrelevant. It forms a true conformance verification that can be used to show that a particular design implements the intent of a specification regardless of its implementation. Kin very ;large or complex design, black box approach requires some non functional modifications to provide additional visibility and controllability.

Additional software accessible registers to control some internal states can be provided. In complex design, some module is taken from third party. This IP (Intellectual Property) is fully verified, but to verify its working within our design, it is verified using black box

approach. In my project, the design using an IP of 32 bit embedded processor from Tensilica. For this module, black box approach was used.

### 1.6.2 White Box Functional Verification

As the name suggests, white box approach has full visibility and controllability of the internal structure and implementation of the design being verified. This approach has the advantage of being able to quickly setup an interesting combination of states and inputs or isolate any functionality. Results can be observed as verification progresses and the source of any problem can be located.

This approach is tightly integrated with a particular implementation and cannot be used on alternative implementations or future redesigns. It also require detailed knowledge of the design implementation to know which significant conditions to create and which results to observe. In my project, for module level functional verification, all the modules were verified using white box approach. This approach ensures that design behave properly with respect to any functionality. All FIFOs, counters or datapaths are appropriately steered and sequenced.

This approach is the main verification used to verify any design. It is the foremost important approach to verify any design that is done for the first time. It verifies the design from all respects. The designer modifies and updates the design as per the feedback from the functional verification team. The final outcome is the completely verified and almost correct design which can be synthesized and fabricated further.

### 1.6.3 Grey Box Functional Verification

Grey box approach is a compromise between the aloofness of a black box approach and the dependence on the implementation of white box approach. As in black box approach, a grey box approach controls and observes a design through its top level interfaces, but it is aware of the internal controls and can use them. This approach is used based on the priority of the features to be verified. If a functionality is to be verified is not prime one, then to attain the deadline, grey box approach can be used.

## 1.7 BUGS

Neither any human being nor any process is perfect. Hence, there is always a possibility of having a bug in the design. Bug can be in the small functionality of some micro module, timing mismatch, short circuit, open circuit, current/voltage tolerances, etc. Endless cycles are done to identify and locate the bug. The aim of bringup functional verification is to develop test cases to find bugs, which have occurred due to the

fabrication process, and the bugs that might have been missed during its functional verification at earlier stage. They aim to test the functionality of the design being manufactured.

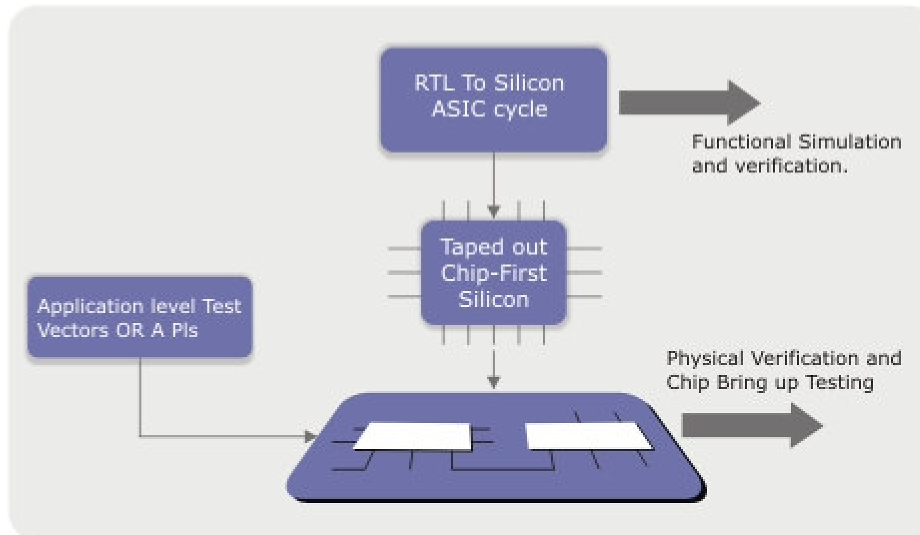
Bugs usually come in three categories:
1) Bugs that have software work around.
2) Bugs that have minor impact on performance or behavior and may ultimately be tolerable or require a layer of respin just to be safe.
3) Bugs that are showshoppers and which actually requires a respin.

The bugs detected in the design after silicon are to be taken care by the software team as nothing can be done to the RTL now. Sometimes in case of some critical failure, extra external peripheral block can be added to nullify the effect of the bug. Other bugs are taken care of in the next version of the chip.

## 1.8 Chip Bringup Verification

This is a part of the complete chip verification process that consists of both verification and testing. It is done on both RTL and the first chip that has arrived from the fab. It is a sort of bringup functional test. It takes into account, the chip with all its peripheral devices and is a board level testing. The aim is not to verify every small sub module in the design but to verify the data paths within and outside to the chip.

After the chip RTL is designed and redesigned from the feedback from the verification side, and after completion of all timing analysis of the chip design, the RTL gets ready for being getting fabricated. Hence, the chip RTL is ready for tap out. This is a very critical period and decision that the RTL is stable affects the cost of the final chip. Hence, this is done only after RTL is considered as being free from bugs. This RTL is given to the fab where the silicon cycle begins. The time between the chip RTL tap out and the arrival of first chip on silicon is used to develop the bringup environment, test cases and the operational board of the chip. The fig 1.3 depicts this phase of chip design.




Courtesy: eInfochips

Fig 1.2 The Chip On Board

The first task done is the finalization of the test plan for the bringup activity. This will include the tests to be written for the each module of the chip to check its functionality and datapath. More emphasize is given on following verification.
To verify core functionality
To verify interactions with peripheral blocks
To verify dataflow within and external to the chip
To simulate for the chip application.

The verification engineers starts with the development of the environment for the chip bringup. This is a verification done at a quite high level of abstraction. This is again done in terms of chip module. Each module is to be verified with its interactions with all its onchip modules and external devices. The environment is made reusable and generic so that it is applicable to all the modules of the chip. No signal is forced as in functional verification of the design. Every signal arrives from its source and passing thought all the hops in its path. This is to ensure the complete data flow.

As the bringup environment gets stable, test cases are written as per the test plan. References are taken from the test cases of functional verification. These test cases are written for the module of interest and the input to the module comes from its actual peripheral sources and is not forced like done in functional verification at module level.

The other task is the development of the operational board for the chip. This board is to be used to run all the test cases developed for bringup after they are simulated. The board as all peripheral devices required by the chip to accomplish its application. All the input source devices are present on board. This is board is used as a demo board later for chip marketing.

The software team starts working on the development of the firmware for the chip. They develop the application code for the chip. Various algorithms required are also developed. The bugs which will be detected by the process of bringup verification are to be handled later by the software. Hence, the software team starts concentrating on such issues.

Other testing done on the chip is using the tester board. It consists of only the chip and no other peripheral devices. It aims to exercise chip will different combinations of test vectors. All the signals are forced on the chip in the test.

## 1.9 The Verification Methodology

Methodology is the step by step procedure to be followed for successful accomplishment of any project. As digital logic designs grow larger and more complex, functional verification has become the number one bottleneck in the design process. Reducing verification time is crucial to project success. The only way to address this problem is to adopt a reuse-oriented, coverage-driven verification methodology built on the rich semantic support of a standard language. For Verification, the Methodology used contributes great to the final conclusion for the process. Design Methodology covers from plan to closure and it includes the Verification Methodology midway. Some Approaches are stated under:

Constrained-random Stimulus Generation

Traditional verification relies on directed tests, in which the testbench contains code to explicitly create scenarios, provide stimulus to the design, and check (manually or with self-checks) results at the end of simulation. Directed testbenches may also use a limited amount of randomization, often by creating random data values rather than simply filling in each data element with a predetermined value.By building randomization into the types of scenarios that are created, not just in the data values that get generated, additional tests are much more likely to hit corner cases and thereby find more design bugs.

Coverage-driven Verification

Coverage metrics serve two critical purposes throughout the verification process. The first is to identify holes in the process by pointing to areas of the design that have not yet been sufficiently verified. This helps to direct the verification effort by answering the key question of what to do next — for example, which directed test to write or how to vary the parameters for constrained-random testing.

Assertions based Verification

The capabilities of any verification environment can be enhanced by the addition of assertions, which are statements of design intent. Ideally, as the designer writes the RTL, he or she documents with assertions the requirements on how the design is expected to behave and the assumptions on interfaces with adjoining blocks. Assertions can range from low-level statements about how specific design elements should behave to high-

level, end-to-end rules about how information should flow through a design. Assertions can be specified in many ways, including with general RTL expressions, special statements within hardware verification languages, and the built-in assertion constructs

## 1.10 High Level Verification Languages

Verilog and VHDL were widely used for verification languages. Due to increase in complexity of functionality within designs, the need for developing language that would aid verification grew. As a part of it, many new verification languages developed. These languages are very powerful in creating conditions which verification engineers require to verify the design from all aspects. Many features of powerful languages are blended together in these languages to support all new approaches. I will be discussing some high level verification languages. These languages are now evolving fr both design and verification. Today's system-on-a-chip designs require multi-discipline engineering teams with a range of skills covering embedded software, system architecture, RTL design and verification. Traditionally these teams use a variety of C modeling styles for architecture design and a variety of hardware description languages (HDLs) and hardware verification languages (HVLs) for RTL design and verification. These traditional methods have led to very complex design flows, prohibited reuse, and have increased the total time to market and development costs for today’s chip designs.

Two industry standards have emerged to allow convergence of the different C-based and HDL and HVL-based approaches. These are SystemC, for C-based system-level modeling and SystemVerilog, providing a unified language for RTL design and verification. Both SystemC and SystemVerilog span multiple levels of abstraction. These languages can support verification at transaction level of abstraction. They enable ease call to functions of other languages and hence provide good interface. Assertion based approach is well supported. Object oriented approach and reusability is taken care of in these languages.

### System Verilog

IEEE 1800TM SystemVerilog is the industry's first unified hardware description and verification language (HDVL) standard. SystemVerilog is a major extension of the established IEEE 1364TM Verilog language. It was developed originally by Accellera to dramatically improve productivity in the design of large gate-count, IP-based, bus-intensive chips. SystemVerilog is targeted primarily at the chip implementation and verification flow, with powerful links to the system-level design flow. SystemVerilog has been adopted by 100's of semiconductor design companies and supported by more than 75 EDA, IP and training solutions worldwide.

### System C

SystemC provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++. Its use spans design and verification from concept to implementation in hardware and software. SystemC provides an interoperable modeling platform which enables the development and exchange of very fast system-level C++ models. It also provides a stable platform for development of system-level tools. The Open SystemC Initiative (OSCI) is an independent not-for-profit organization composed of a broad range of companies, universities and individuals dedicated to supporting and advancing SystemC as an open source standard for system-level design.

**Vera**

Vera® is an industry-leading testbench automation product that increases design quality by finding simple as well as corner-case bugs, quickly. Vera allows engineers to create coverage-driven tests using advanced testbench concepts like constrained-random stimulus generation, real-time data and temporal checking and extensive analysis of functional coverage. Vera combines next-generation constraint solving and coverage analysis engines with a proven reference verification methodology and interfaces to leading Verilog and VHDL simulators. Vera supports the OpenVera® hardware verification language, including OpenVera Assertions, and is an integral part of the Synopsys Discovery™ Verification Platform.

**e**

It is the most powerful HVL. Specman is the compiler/debugger/simulator is for e language. Specman Elite offers a comprehensive verification environment that is based on the e hardware verification language (HVL). The Verisity's Specman Elite is acquired now by Cadence. It is playing an important part in developing reusable verification components.

# 2

# THE VERIFICATION METHODOLOGY & FLOW

## 2.1 Introduction

Methodology is defined as (1) "a body of methods, rules, and postulates employed by a discipline", (2) "a particular procedure or set of procedures", or (3) "the analysis of the principles or procedures of inquiry in a particular field". The common idea here is the collection, the comparative study, and the critique of the individual methods that are used in a given discipline or field of inquiry. Methodology refers to more than a simple set of methods; rather it refers to the rationale and the philosophical assumptions that underlie a particular task.

Verification is a critical part in the specification to silicon path. Hence, it should be done with proper planning and proper methods to make the process effective and quicker. The ultimate aim is to attain the most critical challenges while maximizing overall speed and efficiency. Following an appropriate path is very important for this. Hence, for any verification process, first the approaches, methods, algorithms, sequences, etc is decided upon from all aspects.

## 2.2 The Verification Methodology

As already mentioned earlier, the methodology adopted plays a vital role int the progress and accomplishment of the process. Verification remains the single biggest challenge in the design of system-on-chip (SoC) devices and reusable IP blocks. As designs continue to grow in size and complexity, new techniques emerge that must be linked by an effective methodology for significant adoption and deployment. The SoC industry needs a reuse-oriented, coverage-driven verification methodology built on the rich semantic support of a standard language. Different approaches are possible for targeting verification of a design.

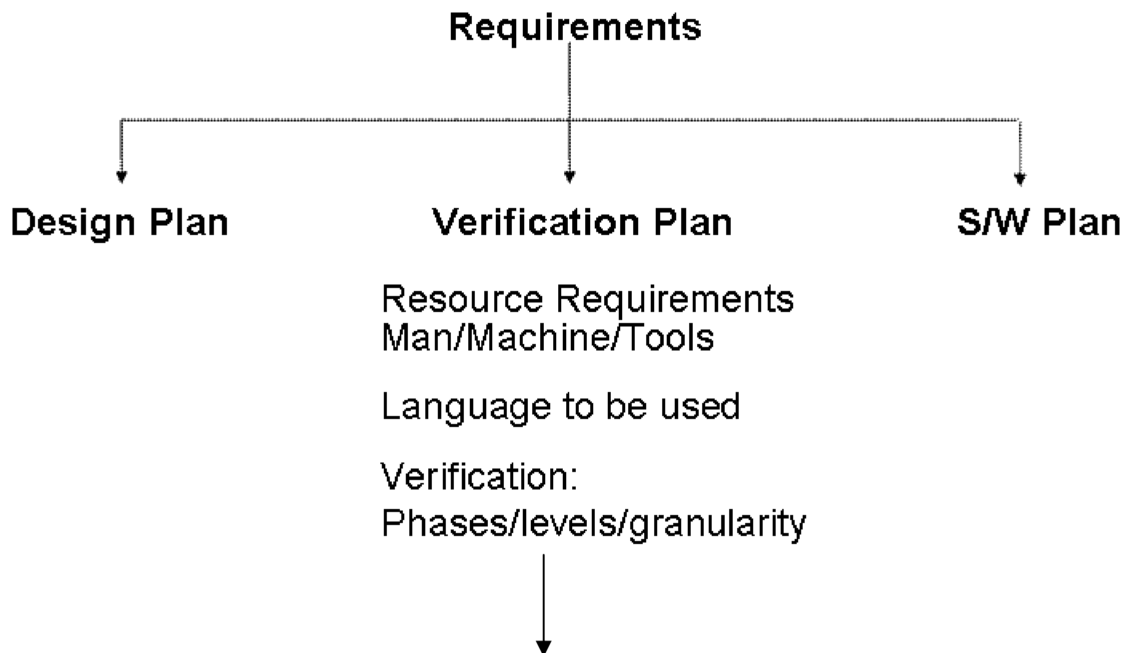
### 2.2.1 The Bottom Top Methodology

In the project, Bottom Top Methodology is followed. The designer have written the RTL Code for the design. But the design is not complete yet. It will be modified and new features will be added based on the feedback from verification process. In Bottom Top Methodology, the verification starts with from the base of the design. In the project, the complete design is divided into functional modules which can be individually verified. These modules are to be verified using white box approach with full visibility. The designer and verification engineer interact to find bugs with the design and modify the RTL to get a functionally correct design. The level of abstraction at module level is very low. Not many assumptions are made. Interest is to verify the design with transaction in

each signal. The modules are to be verified for all possible inputs and even the invalid inputs. Corner cases are to be identified to verify the design from all aspects. After the completion of this phase, the level of abstraction rises. Now all the modules are to be integrated to and verification is done for correctness of the interactions between the modules. At this time, the modules are assumed to be functionally correct and only there mutual interactions is verified. After this phase system level verification is done. The design is verified for its functionality with the all peripherals and system components. The level of abstraction is highest here. The approach will be clearer as I start with the Verification flow.

**2.3 The Verification Flow**

I will be discussing the complete flow followed by the company to achieve the verification. I will start with common approach for any designing and will proceed with concentration on the verification. Any project is the outcome of some requirement. As without any requirement, there is no profit in going for any project. Hence, depending on the market requirement and market availability for the project product, a project is finalized. Figure below shows the Verification Flow followed for the project. It shows the complete flow from Plan To Closure. Different Tools are used at each step. The Flow goes in loop as the project progresses. Depending on the feedback from tool or updation of RTL due to addition of new feature or fixing of some bug, the flow undergoes in a loop.



**Development of VE. Understanding of Tools, VE, Spec.**
BFM, Checkers, Analyzers, Assertions,
Functions

**Identifying the features of module to be verified at component level.**
This will be white box verification approach. The abstraction level is very low

Test Case Req
New features
RTL updation

**Developing the test bench for the module.**
Module and other tasks will be instantiated here.

**Prioritizing the features to define first time success**
Smoke test (makes sure that VE and RTL is stable).
Reg Reset & R/W Values verification.
Prime functionality.

**Grouping the features into test cases. Identifying no. of test cases**
Common features (functionality/resource reqd) form same category of test case.

New features
RTL Updation
Coverage f/b

**Writing & Execution of test cases with diff i/p combinations**
Valid & Invalid Inputs

**Regression**
This step is done several times as RTL undergoes updation when bugs get fixed.

New Issues

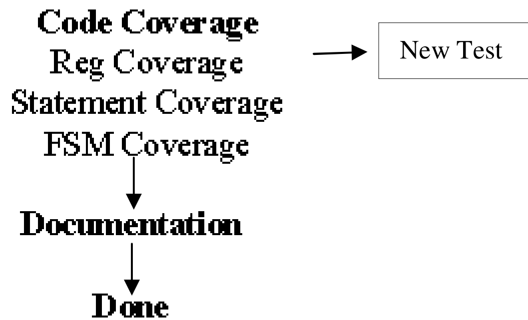



Fig2.1 Verification Flow

### 2.3.1 Design Plan, Software Plan & Verification Plan

The Project Plan decides other plans for the project. These are the Design Plan, Software Plan and the Verification Plan. These three teams play part in the project. These are the most important teams responsible for successful completion of the project. Design Plan and Software Plan states all the requirements from there point of view.

**Verification Plan**

The Verification plan acts as a specification for the verification effort. It is used to define what is first time success, how the design is verified and which testbenches are written. The verification plan states everything related to Man/Machine & Tool requirements. It gives the approach to verify every module. The verification plan includes:

1. Requirements (Man/Machine) Verification.
2. The languages to be used for verification.
3. The tools to be used for all tasks in verification.
4. The engineers to be worked on the project and the duties to be assigned to them.
5. The requirements in terms of System OS, S/W Drivers, Application S/W, etc.
6. The CHIP configuration.
7. Level of granularity/abstraction for all modules to be verified.
8. Division of the complete verification process into different phases with deadlines defined for each of them.
9. All the verification specific things.

The verification process is divided into phases.
**First Phase** : This is the Block level verification. Each block is assigned to individual and is verified with interactions with the designer.
**Second Phase** : This is Cluster level verification. Few blocks with high mutual interactions are combined and are verified.
**Third Phase** : This is Full Chip level verification. All blocks are combined and the functionality of the complete design is verified.

**Fourth Phase** : This is System level verification. The chip is verified with the system in which it will be used. Third Party Models are included here.

### 2.3.2 First Step To Project

As the complete plan for all three major team is ready, now they start working on the project. Design team starts the designing of individual blocks depending on the architecture finalized. Software team starts with the implementation of algorithm to be used to make sure that it will work in the design. If the algorithm works well, then RTL coding for it is done. The software team then starts with the necessary software required by the project.

### Development of VE

The verification team's task starts with complete understanding of the specification. IN our project, the chip works on a protocol. So, we need to completely understand that the protocol first and then the specifications stated for each block. While verifying, it is required to also check that designer stick to the protocol. Then all the tools to be used is studied. I have described all the tools used in our project in detail in later chapters. Then starts the development of Verification Environment. All the requirements required from VE is mentioned in verification plan. The VE is developed accordingly. The VE in our project is in C++ and Verilog. The BFMs (Bus Functional Models), Checkers, Analyzers, Assertions, etc are written as a part of VE.

### 2.3.2 Identifying The Features

The development of VE is completed at this point. It would be modified or updated later depending on the requirements as verification proceeds. Now, each verification engineer is given individual module and it is his/her responsibility to verify it. Now, starts the identifications of the features to be verified. This is a white box approach and hence all the features have to be verified. There would be some features which are to be verified at full chip level and not at module level. e.g. interrupt propagation structure.

### 2.3.4 Developing The TestBench

When the design will be verified, it is required that the design should be given the same environment which it will find when it will be actually used in the system. The testbench is like a universe for the design. Testbench is a closed system. It generates all the inputs and compares the outputs generated by the design. Testbench will configure the design as per the requirements by the test case which verify design functionality. It is important to be able to develop generic testbench which is applicable for verification of all functionality of the design. The testbench is updated as per the feedback from verification process later.

### 2.3.5 Priotizing Features

For any design, few features are of prime importance. Bugs in such functionality implementing design, leads to more bugs in the functionalities depending on them. Hence, such functionality must be considered first. In any design, you will find many

configurable registers. The configuration, operation and output from the design depend on these registers. So, it is of prime importance to see that all registers are functionally correct.

### 2.3.6 Grouping Into test Cases

The ultimate aim is to verify all the features of the design within the deadline. So, time taken is very important. Hence, the prime objective is to keep the number of test cases less and still be able to verify all the features. Hence, the features are grouped into same category of test cases depending on the functionality they implement. some features require similar configuration, granularity or verification strategies. Hence, to increase the productivity, these features should be grouped into common test case. Again here the priority among the features is always taken care of. Now, all the test cases required is known. Number of test cases required for each functionality verification is known.

### 2.3.7 Writing & Execution Of Test Cases

The test cases are written and are executed with valid and invalid inputs. All possible corner cases are applied on the design. For the failing test cases, debugging is done. The source of problem is found. The issue could be related to VE, Test Case or RTL. Once it is confirmed that the problem is related to the RTL, bug is filled for that. Designer debugs the design and locates the problem. The RTL is modified accordingly to fix the bug. The test case is executed again. This process repeats till the test cases passes.

### 2.3.8 Regression

As and when bugs are filed, the designer debugs the RTL and modifies it. Hence, the RTL goes on updating as the verification progresses. Hence, there can be a chance that a test case passing previously might fail with this new updated RTL. Hence, after every updating of RTL, regression of all test cases is done. Again, at the end of the verification, all test cases are executed for the same.

### 2.3.9 Code Coverage

This is very important task to which defines the effort placed by verification engineers and the completion of the verification task. From the feedback from the coverage tool, new test cases required are written and executed. It is desirable to get 100% coverage. But this is a difficult task. Depending on the deadline of the project, the coverage required can be loosen.

### 2.3.10 Documentation

This is the final step in the process and declares the closure of the verification process. It includes documenting all the effort done and the results generated. This documentation can be used by some other person or team in case the design is updated later.

## 2.4 Designer – Verification Engineer Interaction

At all the steps discussed earlier, interaction between designers and verification engineers is required. Any open bug is discussed by both to come to a conclusion. The partition of the design into modules is done by the both teams together. Many DFV (Design For Verification) features are added to the RTL to aid the verification process. Some more register for configuration or some mux to bypass some functionality can be added.

### Design For Verification

There are two major reasons for the presence of design errors (bugs) in a design. First, the sheer complexity of a module, often including multiple-state machines, makes it virtually impossible to anticipate all possible conditions to which the module can be subjected in the context of an application. Typically the state space is very large and bugs can be buried very deep into the logic. Hence, some corner cases may simply not have been anticipated in the implementation. Second, designing a module often requires the designer to assume a particular behavior on the interface, that is, make assumptions on the behavior of modules physically connected to the module under design. These assumptions are needed to assure minimum area/maximum speed micro architectures to be designed. Analyzing the bugs indeed shows that incorrect or imprecise assumptions are a major cause of design flaws.

To improve the quality of the design process we clearly have to address specification, design, and verification in a concerted manner. Similar to other seemingly independent tasks in the past such as manufacturing test, design quality needs to become the whole team's concern, and the methodology employed must support this notion. That is exactly what the DFV methodology offers coherent methodology to find design errors through constrained-random stimulus generation and advanced tools for formal state-space analysis, powerful means to eliminate ambiguity in specifications, and improved conformance checking capabilities to ensure that the imported design IP complies with required standards.

# 3

# THE MIME MODULE FOR FUNCTIONAL VERIFICATION

## 3.1 Introduction

The revolutionary growth of communication is further revolutionized by the evolution of optical networks and standard for it namely SONET & SDH. The need for chips to work in optical network is increasing and the market is increasing. Hence, an optical networking chip finds a good market in today's world. Depending on the customer's requirements and need, the company came up with a SONET/SDH Processor & Cross Connect chip. This chip is called OTII. It works on the SONET/SDH standard.

## 3.2 The Protocols/Standards

### 3.2.1 Historical Background

When data is transmitted over a communications medium, a number of things must be provided on the link, including framing of the data, error checking, and the ability to manage the link (to name a few). For optical communications these functions have been standardized by the ANSI T1X1.5 committee as Synchronous Optical Networking (SONET) and by the ITU as Synchronous Digital Hierarchy (SDH).

In those days, the telephone companies looked at optical communications as simply a replacement for the older wire or microwave communications they had been using for years. But then they encountered a practical problem. Vendors of optical communications equipment had used their own framing techniques on the optical fiber. Once you selected a vendor, you were stuck with that vendor for all the equipment in that optical network. Thus was born the concept of standards in optical communications. It's extremely important to recognize that the first standards for optical communications were focused on handling voice circuits, and especially legacy plesiochronous channels like DS-1s and DS-3s. If you keep this fact in mind, many of the odd things about SONET and SDH will make more sense. At the time that these standards were developed, the tremendous volumes of data traffic had not appeared and most people did not foresee it.

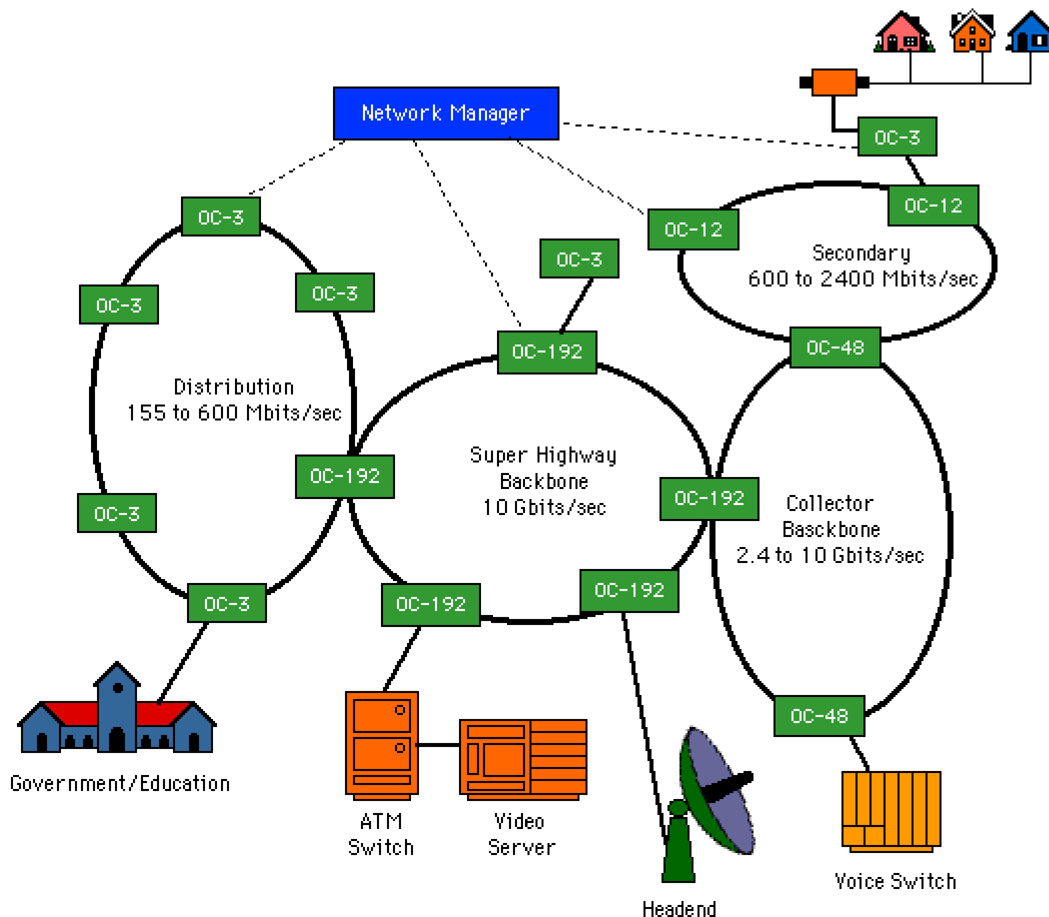



Fig3.1 Optical Network

### 3.2.2 Introduction to SONET/SDH

SONET/SDH defines the low level framing protocol used on these optical links. By “framing”, we mean a block of bits (or octets) which have a structure, and which utilize some technique which allows us to find the boundaries of that frame structure. Parts of the block may be devoted to overhead for the network provider to use to manage the network. Other parts will be dedicated to carrying payload, or information we want to communicate. The end-to-end connection through a SONET/SDH network is always called the “path.” The connection between major nodes, such as between add/drop multiplexers is called a “line.” And the link between an add/drop multiplexer and a regenerator, or between two regenerators, is called a “section.” The SONET is a subset of SDH. So, i will be discussing the SONET first. It is equally applicable to SDH as well.

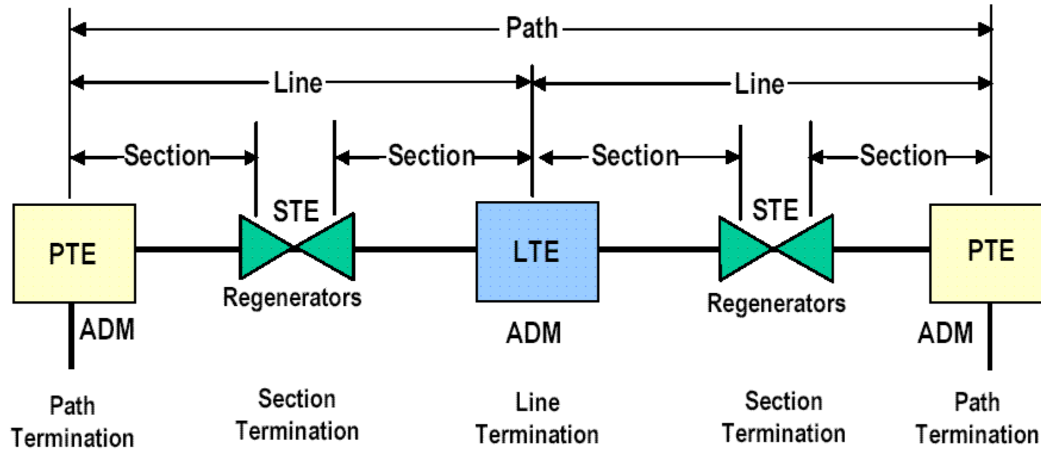



Fig 3.2 SONET End To End Connection

### 3.2.3 SONET Basic Frame Structure

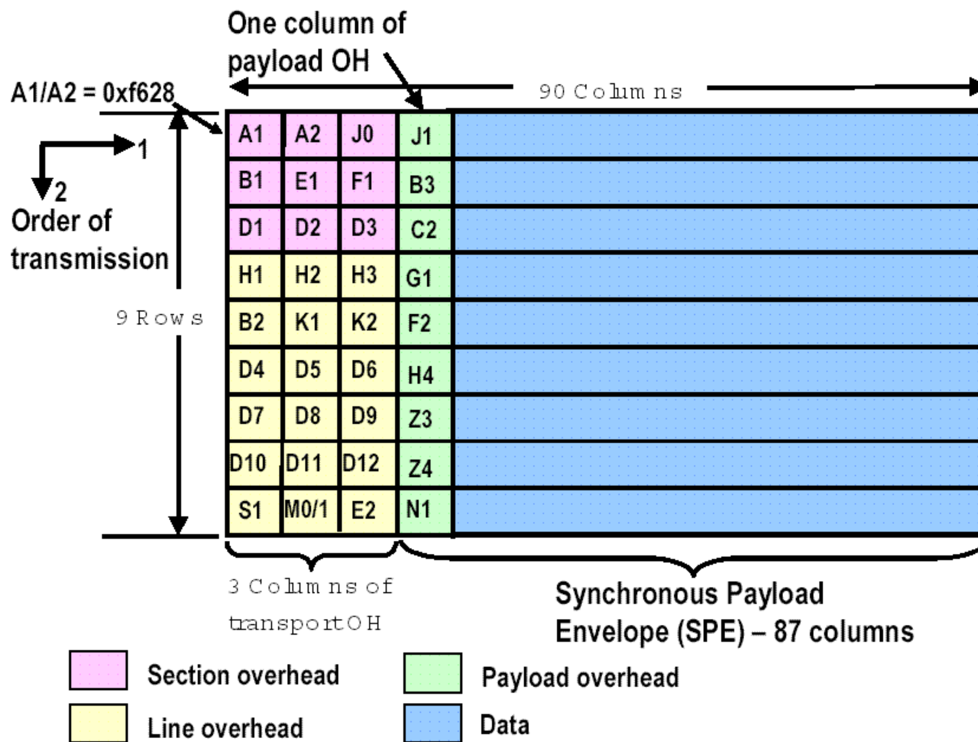



Fig3.3 Basic SONET Frame

The basic SONET frame is set up as shown in Figure,9 rows of 90 octets. It is transmitted from left to right and top to bottom. That is, the octet in the upper left corner is transmitted first followed by the second octet, first row, etc. When we get to octet 90, we come back and start with the first octet of the second row. It's important to realize that this two-dimensional representation is just for convenience. The bits are simply transmitted one after another in a serial stream. We could also represent this SONET frame as a linear sequence of 810 octets. Every 90 octets we'd have three overhead octets.

Framing is accomplished by the first two octets, called the A1 and A2 octets. When the frame is transmitted, all octets except A1, A2, and J0 are scrambled to avoid the possibility that octets in the frame might duplicate the A1/A2 octets and cause an error in framing. The bit pattern in the A1/A2 octets is 1111 0110 0010 1000 (0xf628). The receiver searches for this pattern in multiple consecutive frames, allowing the receiver to gain bit and octet synchronization. Once bit synchronization is gained, everything is done, from there on, on octet boundaries – SONET/SDH is octet synchronous, not bit synchronous. The first three columns of a SONET frame are called the Transport Overhead (TOH). The 87 columns following the TOH are called the Synchronous Payload Envelope (SPE). Within the SPE there is another column of overhead, called the Payload Overhead (POH), whose location varies because of timing differences between networks. This leaves 86 columns by 9 rows for usable payload in this basic frame.

SONET designers were interested in carrying their legacy plesiochronous traffic. Because of this, they tied everything to the existing voice traffic. And in the digital telephone network, voice is digitized according to ITU specification G.711. That is, the granularity of the voice samples is one octet, and samples are taken 8,000 times per second, or every 125 micro seconds. So this basic SONET frame is designed to carry one octet for each voice conversation in each frame. This means that a SONET frame has a period of 125 micro seconds. Hence, it turns out that every SONET frame repeats every 125 micro seconds, no matter how fast the line speed gets. As the line rate goes up, the SONET frame gets bigger by some number of octets, just sufficient to keep the frame rate at 8,000 frames per second. For this first level basic SONET frame, this gives a data rate of 51.84 Mbps (90 columns times 9 rows, times 8,000 times per second, times 8 bits per octet). This signal is known as a Synchronous Transport Signal - Level 1 (STS-1). Once the scrambler is applied to the signal, it is known as an Optical Carrier – Level 1 signal or OC-1. Since there are 86 non-overhead columns of 9 rows, a SONET STS-1 frame has a usable payload rate of 49.536 Mbps, sufficient bandwidth to carry 774 simultaneous voice conversations4. This is in excess of the 672 simultaneous voice conversations carried in a DS-3, allowing one DS-3 to be easily mapped into a SONET STS-1 channel. Plesiochronous digital hierarchy (PDH) traffic (DS-1, E1, DS-1C, DS-2, or DS-3) is encapsulated with additional framing octets, designed to allow the PDH traffic to be carried within a SONET/SDH channel. This is known as a virtual tributary (VT) in SONET and a virtual container (VC) in SDH. Multiple DS-1 circuits, for example, may be combined into a single SONET channel, up to 28 DS-1s in an STS-1.

### 3.2.3 SONET/SDH Multiplexing Structure

| SONET name | SDH name | Line rate (Mbps) | Synchronous Payload Envelope rate (Mbps) | Transport Overhead rate[5] (Mbps) |
|---|---|---|---|---|
| STS-1 | None | 51.84 | 50.112 | 1.728 |
| STS-3 | STM-1 | 155.52 | 150.336 | 5.184 |
| STS-12 | STM-4 | 622.08 | 601.344 | 20.736 |
| STS-48 | STM-16 | 2,488.32 | 2,405.376 | 84.672 |
| STS-192 | STM-64 | 9,953.28 | 9,621.504 | 331.776 |
| STS-768 | STM-256 | 39,813.12 | 38,486.016 | 1,327.104 |

Table3.1 SONET/SDH Digital Hierarchy

An STS-3 can be thought of as three STS-1 bit streams transmitted in the same channel so that the resulting channel rate is three times the rate of an STS-1. And when multiple streams of STS-1 are transmitted in the same channel, the data is octet multiplexed. For example, an STS-3 signal will transmit octet A1 of stream 1, then octet A1 of stream 2, then octet A1 of stream 3, then octet A2 of stream 1, octet A2 of stream 2, etc. This multiplexing is carried out for all levels of SONET and SDH, including STS-192 and STS-768. Because of this, SONET/SDH maintains a frame time of 125 micro seconds. Overhead associated with the transport overhead columns only. Excludes overhead contributed by the POH. SONET/SDH utilizes octet multiplexing in order to reduce delay. Octet multiplexing is used instead of bit multiplexing because, in SONET/SDH, everything is done in octets instead of bits.

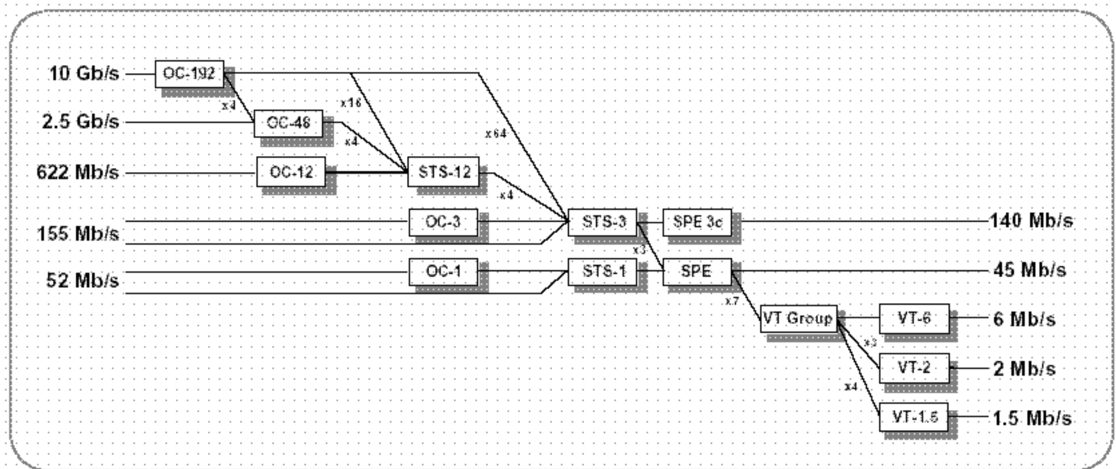



Fig3.4 SONET Multiplexing Structure

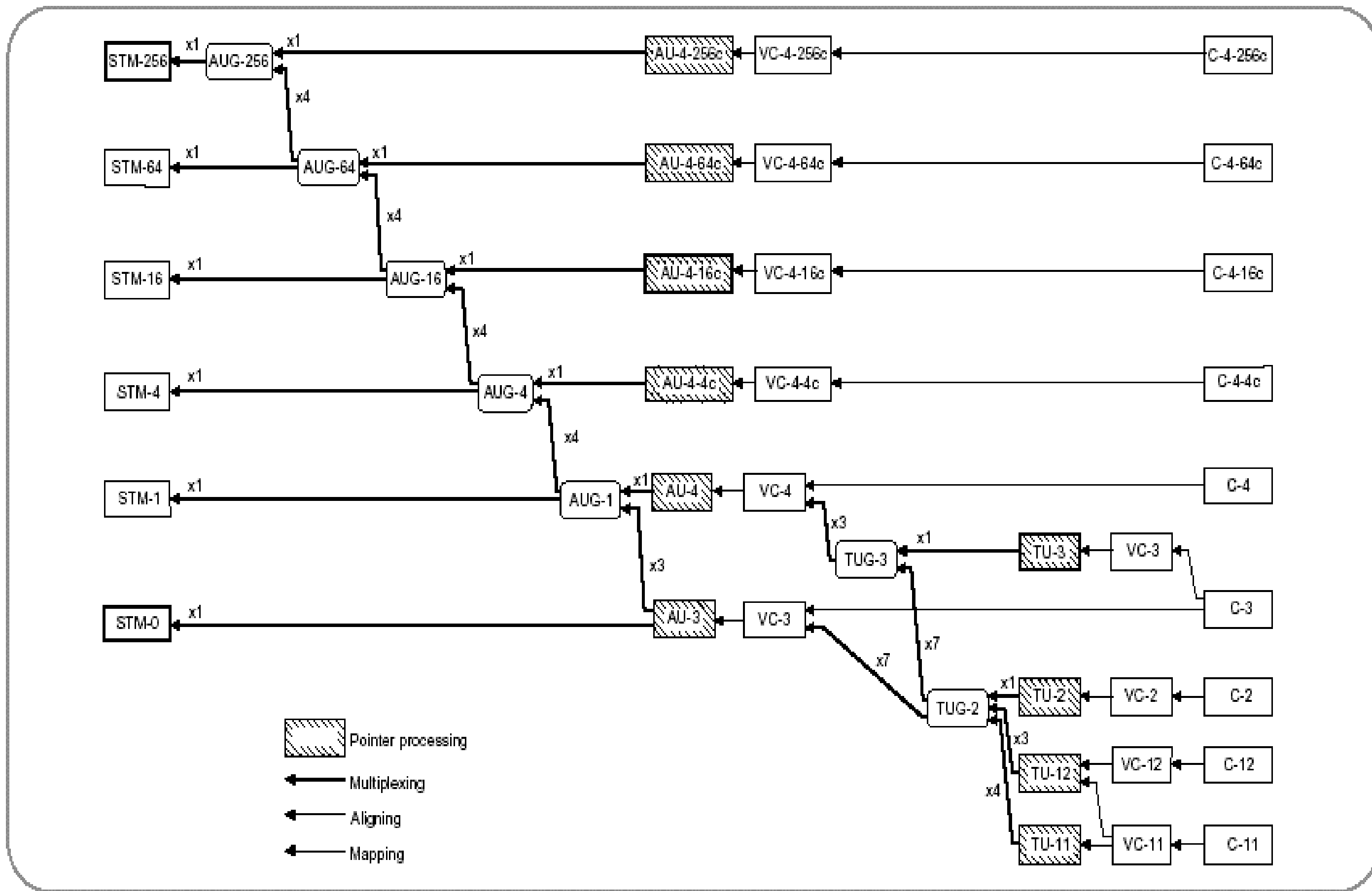



Fig3.5 SDH Multiplexing Structure

## 3.3 MIME Module

Due to company's confidentiality, the complete detailed block diagram can not be shown. On the chip, there are many independent modules that performs specific functionality. Each verification engineer is assigned an individual module. The module, I worked on is MIME (Mate Ingress & Mate Egress). The features of this module are described below.

**Features Of Mate Ingress module:**

**One port for MI with OC-48 capacity :**

The MI module has one input port. Port has 16 channels and the capacity of each channel is OC-48. i.e. each channel can carry traffic of maximum OC-48 rate. SerDes Interface is

provided. SerDes or serializer/deserializer that converts parallel data to serial data and vice-versa. The transmitter section is a serial-to-parallel converter, and the receiver section is a parallel-to-serial converter. Any channel can be configured as OC-12 or OC-48. Any channel can be optionally disabled.

**Deskew and Frame Alignment :**
The data coming from different channels can be at offset from each other. To process the frame further, it is required to deskew i.e. align all the frames coming from channels. MI module has the ability to deskew channel offsets upto 64 bytes. This adjustment is configurable in terms of byte.

**Higher Order Pointer Interpretation and Path Extraction :**
After the channels are deskewed, the frames are processed. Depending on the Higher Order Pointer, the SPE can be extracted.

**Generation Of Various Alarm Indication Signals :**
The frame is processed and different alarms are generated based on the data in the frame. Different alarms like loss of frame, loss of pointer, B1 B2 parity error, deskew error, etc are generated. This generation can be optionally disabled.

**Optional Use For Line Side :**
Optionally it can be used for Line Side traffic.

**Features Of Mate Egress module:**
**One port for ME with OC-48 capacity :**
The ME module has one input port. Port has 16 channels and the capacity of each channel is OC-48. i.e. each channel can carry traffic of maximum OC-48 rate. SerDes Interface is provided. Any channel can be configured as OC-12 or OC-48. Any channel can be optionally disabled.

**Overhead bytes in pass through mode :**
All the TOH and POH bytes are in pass through mode.

**B1 B2 Insertion :**
It computes Bit Interleaved Parity B1 and B2 and inserts in the outgoing stream. Optionally insertion can be disabled and B1 B2 can be made in pass through mode.

**A1 A2 byte Insertion :**
It inserts correct value of A1A2 i.e. 0xF628 in the outgoing stream. The A1 or A2 can be inverted by various configurations possible.

# 4

# *THE MODULE LEVEL VERIFICATION ENVIRONMENT*

## 4.1 Introduction

To simplify the task of Verification at both module and full chip level, the verification environment is made as generic as possible. Most of the portion is based on MACROS. The VE comprises of C++ and Verilog components. They interact with each other through files. The C++ Components forms the basic platform for the VE. The Verilog components deal with the test bench portion. Generation of clocks, source and sink drivers are part of Verilog components. It has a common test bench file, which is used by all modules. It is complied selectively based on the working module. All top level headers are defined in a separate file. It includes many tasks to govern the complete verification process.

## 4.2 The C++ Environment

The C++ components deal with initial generation of various command files for TWB, Register Map files related functions and other useful functions like randomization. The C++ VE generated all the required commands files for the TWB generator and analyzer. Its main block is the test command file generation engine, which reads the test case and generates all command files for TWB to generate and analyze the frames accordingly. It contains different functions for the configuration of frames in any SONET/SDH standard compatible manner. It generates files, which Verilog VE reads to execute the test cases. It contains functions to generate CRC, DCC frames, Randomization, etc. Diagram shown below depicts the generation of files generated by the C++ environment.

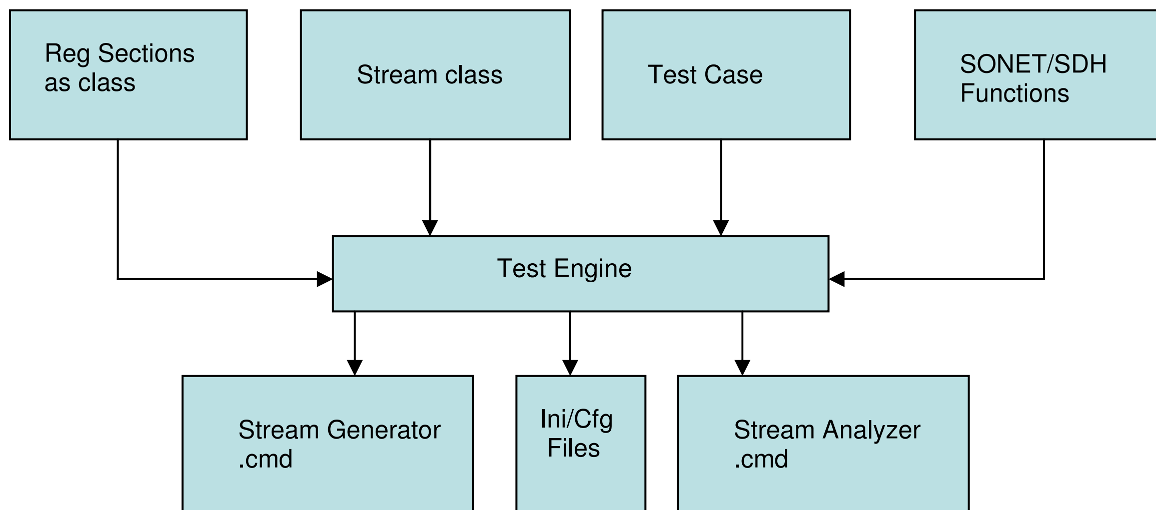



Fig 4.1 The C++ Environment

**Inputs to Test Engine:**

As seen from the Fig4.1, the Test Engine Section is the heart of the C++ environment. It generates all the files required by the Stream generators, analyzers and the test bench. The Test case for verifying particular feature of the design is in pseudo C code which is converted into C and acts as input to the Test Engine. From it, Test Engine gets to know the various configurations required to be done to the test bench to accomplish the process. All the configurable registers of the design are encapsulated in form of individual class for each section. Each of the registers in OTII defined as a class object. There are class functions that provide access to the elements of the class to set and clear various objects. The reason for implementing it this way is that the test files do not change if the register does move from one address to another during the RTL implementation phases. The register call remains the same. A set of classes is called as constructor functions in the main {} so that these classes are present as objects in memory for the period of the test engine run.Stream Class gives configuration for various different types of input streams from simple to mixed type. Test Engine also calls other functions useful for stream configuration.

**Outputs from Test Engine:**

The Test Engine generates command file for the input stream generator. Hence, the generator gets configure accordingly and generates the stream which is input to the DUT. Similar type of command file is generated to configure the Stream Analyzer. This generates stream with which the generated output form the DUT will be compared. The analyzer output is first used to check all the overhead bytes in the stream. It does not perform data integrity check. This is performed later.Test Engine also generates various configuration files to configure the test bench as per the test case requirement. Various memory programming files and register files are generated. This files are used to actually configure the DUT registers.

## 4.3 The Verilog Environment

The Verilog components deal with the test bench portion. Generation of clocks, source and sink drivers are part of Verilog components. It has a common test bench file which is used by all modules. It is complied selectively based on the working module. All top level headers are defined in a separate file. It includes many tasks to govern the complete verification process. The following sections explain in detail, the overall contribution of all these files. The module specific test bench is in its folder of module. This file is having the instantiation of the module and the other required verilog modules.
One of the main features are that the test bench is self checking and self configuring. It has hooks to report errors which can be overridden. In order to implement these features, the testbench is constructed using a modular approach. There are three main components to the testbench as indicated below. These sections are merely broken up for clarity, maintainability and portability across the various modules.

**The Global Definitions Section**
This section defines the global constants that are common to all the simulations. It does not contain any module specific information in it.

**The Common Test Bench Section**
This section has the following components in it
- Clock Instantiations for various frequencies. ( common ones only)
- Reset
- Configuration memory
- Tasks to start the sources and sinks,
- Tasks to clean up simulation after ending
- Variables to control Behavior of the testbench itself.
- Microprocessor handler
- Initialization sequence task
- Register readback sequence task
- Dump tasks
- Frame counters

As can be seen, There are no source or sink instantiations for each of the modules or the module instances themselves in this section.

**The DUT Section**
This section has the following components in it
- The actual sources and sinks required by the specific module under test
- Any additional clocks not provided by the main section.
- The Dut itself
- Any specific hardwiring of signals.

To further divide the complete stuff, i have divided the files as per their contribution to

the verification process.

**Clock Generation**
These files contribute to the generation and distribution of clock. The main file is clock.v. It generates clock as per the period defined. The timescale is kept as per MACRO TB_THARAS. clock_select.v, clock_new, clock_mon are for the distribution and monitoring of the clock. check_clock.v checks whether the period of the clock is as required. It can be enabled or disabled by assigning MACRO DISABLECHECK to DISABLE 1'b0 or 1'b1. Four D Flipflops are kept of synchronization.

**Frame offset and synchronization**
The frame_offset.v is generates the frame offsets for all IoConfigs and channels. The enabling is asserted only after FrameOffsetDone. For frame synchronization four files each for oc3, oc12, oc48 and oc192 rate is defined. These files frame_synchronizer.v

inputs frame, clock and reset and outputs frame, clock and start of frame. This is done based on the detection of specific number of A1A2 pairs as per the oc rate.

**Source Drivers**
This section includes all the required files for driving the source stimulus. It also includes driver for TFI source. It also includes source drivers for specific module. e.g. OH port for SE. Top level source driver takes care for the actual injection of the data through the ports. These modules outputs data from the memory.

**Sink Drivers**
This section includes all the required files for dumping the signals. It also includes sink driver for TFI source. It also includes sink drivers for specific module. e.g. OH port for LI. Top level sink driver takes care for the actual ejection of the data through the ports. These modules inputs data.

**Parallel To Serial Converters**
Four files by the name Source_p2s.v are to convert data from parallel to serial for all oc rates. Input for oc192 is 128 bits in parallel which is serialized into 16 bits serial links. Input for oc48 is 32 bits in parallel which is serialized into 4 bits serial links. For rest rates, input is 8 bits in parallel and it is serialized into a single serial link.

**The Top Header Files**
Two top header files are defined. top.vh and tb_Ruby.vh. These are the top System level Verilog Header files. tb_Ruby.vh is included in the main test bench only in case MACRO SIM_CORE_PU defined which is solely for top level PU verification. top.vh includes uses complier directives to declare shared macros and constants. It defines macros for clock tree delays, watchdog timer, flush delay, different data widths, clock periods and half periods, signal propagation delay, clock and clock recovery after reset delay, configurations, cpu related stuff, common global regs and defined reserved reg space.

**Common Test Bench**
The common test bench is tb_template.v. This file is the basic test bench common to all modules and defines all the common tasks. It is based on MACROS and hence each module complies it depending on the working module. As already explained earlier, it defines all the major features of the test bench. It contains the top module in which module specific file is included. It defines the IoConfig words and configures test bench accordingly. The microprocessor interface is used for reading and writing the regs. All required tasks are defined here. Any new reg defined in reserved space has to be declared in top.vh and its bit fields are to declared in this file. These bit fields can now be accessed using their reg instant. Include these bit fields in the up read and write tasks to enable read and write to these new regs.

**Module Test Bench:**
This is the module specific test bench named tb_<module>_mod.v e.g. tb_mime_mod.v, and can be found in tb folder of that module. e.g. sim/mime/tb. As explained previously,

the DUT is instantiated here along with other required sub modules. It declares all the input and output signals of the module. Additional clock required can be also declared here. Various drivers and are enabled or disabled as per the configuration of the DUT. Any additional logic can be added to the test bench for realizing new scenarios required for verification.

Hence, the test bench gets configured as per the test case. As explained earlier, tb_template.v file is the main common test bench for all modules. This file contains top module which will include the module specific test bench as per the scope selected. All the required sections of test bench are used based on working module. Input stream is generated by the Telecom Workbench Generator program. The DUT instantiated in the test bench is simulated with the input applied and generates output streams which goes ahead to the checker and finally test passes or fails. For debugging of the test case, the signals can be monitors in the waveform viewer.

### 4.4 The Test Bench

As already explained earlier, the Test Bench is in Verilog. Two most Important files for Test Bench are the generic test bench tb_template.v and the module specific test bench tb_mime_mod.v. Figure explains the interaction between this files and the data flow for simulation of the MIME module.

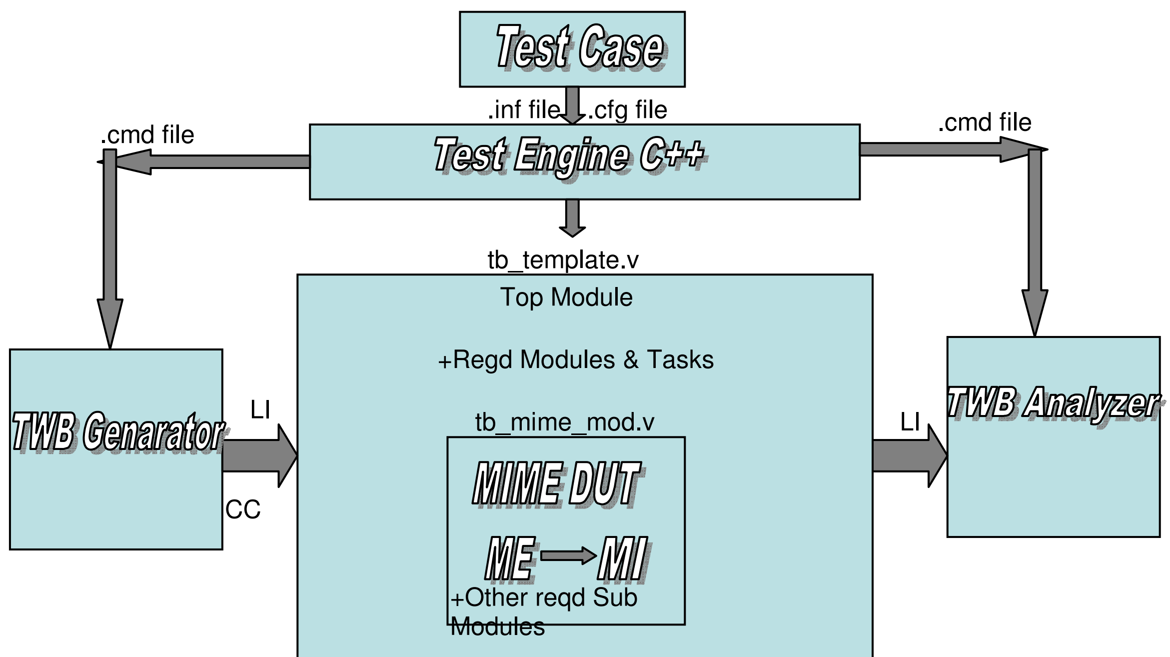


Fig4.2 Complete VE For MIME Module

The Test Case is written in pseudo C++ language which is converted into C++ using a PERL script. The C++ Component test engine contains all the required VE components and generates command files which are read by TWB. Other files are read by Testbench.

### 4.5 The Simulation Flow

To execute simulation we first need to compile the environment. The process to compile the environment called environment bring-up. There are two separate environment that we need to compile before to go further action. First is “C++” environment while second is VERILOG environment.

We use different kind of scripts to accomplish the purpose. The parent script is written in PERL language. The PERL script has name “cmsim”. The cmsim accept different switches to perform different kind of execution. Following are the general functions which cmsim will perform regardless of command line arguments.

- Check the configuration of environment variables
- If particular environment variables are not set that set it to appropriate value
- Call RealMakefile with different input command option – make file

Fig explains the complete simulation flow.

1. The test writer obtains a template from the library and customizes this template to reflect the changes.
2. The test fed into a test command generator engine
3. The test generator engine compiles an executable and then runs the executable
4. The test generator engine generates the following files:
    a. Generator command file
    b. Analyzer command file
    c. Memory programming file
    d. PLI initialization file
    e. Testbench configuration file
    f. Write register file
    g. Read register file
5. The test writer obtains a template from the library and customizes this template to reflect the test conditions.
6. The test fed into a test command generator engine
7. The test generator engine compiles an executable and then runs the executable

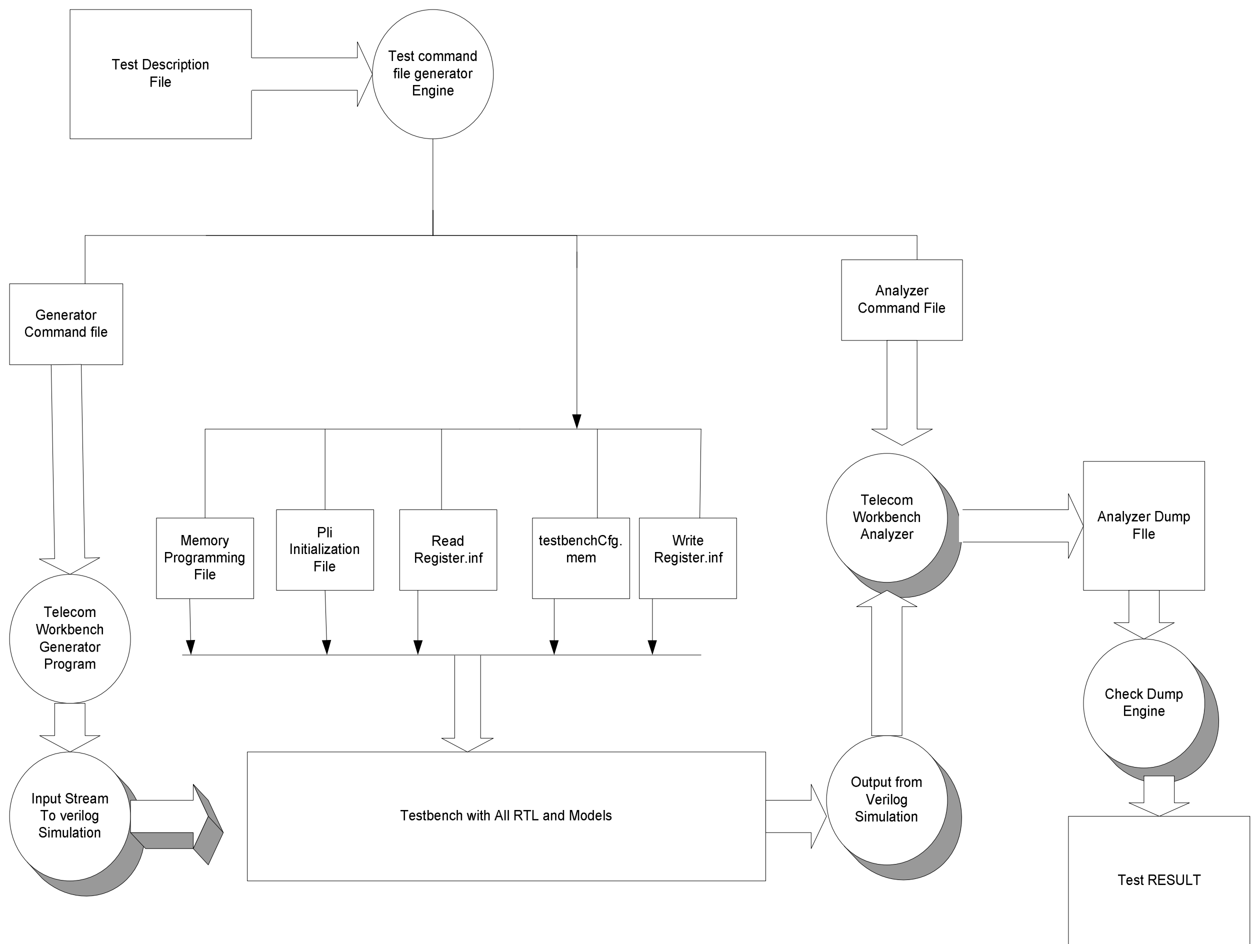



Fig4.3 The Simulation Flow

8. The test generator engine generates the following files:
   a. Generator command file
   b. Analyzer command file
   c. Memory programming file
   d. PLI initialization file
   e. Testbench configuration file
   f. Write register file
   g. Read register file
9. The telecom workbench program then takes the generator command file and creates a stream.
10. The stream from step #4 and the other files (excepting the analyzer command file) are read into a verilog model that has the DUT and other components in it. The verilog simulation then runs till it runs out of input stream. As it is running, it produces a output stream file and a log file.

11. The telecom workbench Analyzer is then run on the output stream produced in Step #5 along with the Analyzer command file produced in Step #3. This verifes that the stream is telecom standard compliant. It does NOT check for data integrity. It extracts each payload for each VC11/VC12/(VC2/VT6)/STS/M1 and produces a dump file called Analyzer.dump.

12. The check stream is run on the analyzer.dump produced in Step #6. It verifies that no data corruption happened in the data from the analyzer.dump.

13. If the following

    a. Register read/write miscompares

    b. Analyzer log file

    c. Check dump

Do NOT report any errors, then the test is deemed as passing. Please note that sections (a),(b),(c) are to be ANDED and all are required to declare the test a pass.

# 5

# *THE MODULE LEVEL VERIFICATION FLOW*

## 5.1 Introduction

This Chapter discusses the complete verification process for the module MIME. As already discussed earlier, this is the first phase where module level verification is done. The complete OTII chip 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 verufied against two things here. First is the SONET/SDH standard that is the design has to process the incoming streams as per the SONET/SDH standard. And second is to verify that the design does 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.

## 5.2 Languages Used

As mentioned earlier, the VE is based on C/C++ and Verilog. RTL design is in verilog. The supporting languages are Perl, Makefile and TCL Tool Command Language.

### 5.2.1 C/C++ and Verilog

C/C++ traditional language is used due to its flexibility. 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/C++ was selected as the base for VE. Verilog is used in all Testbench related portion. The Telecom Workbench used, works with Verilog with inbuilt interface and hence requires no transformation of any kind. Again the testbench was required to be generic and applicable to all modules and full chip level verification. Verilog with its powerful interaction with C/C++ through PLI and file management syntaxes was selected. Hence, the VE works smoothly with C/C++ and Verilog.

### 5.2.2 Perl,Makefile & TCL

PERL stands for Practical Extraction and Report Language. In larger and complex projects, there is a requirement of automation of the commands to be given for compilation, simulation, etc. All high-level language code must be converted into a form the computer understands. For example, C language source code is converted into a lower-level language called assembly language. The assembly language code made by the previous stage is then converted into object code which are fragments of code which

the computer understands directly. The final stage in compiling a program involves linking the object code to code libraries which contain certain built-in functions. This final stage produces an executable program. To do all these steps by hand is complicated and beyond the capability of the ordinary user. A number of utilities and tools have been developed for programmers and end-users to simplify these steps. Hence, Makefile is used. It contains dependencies in terms of commands and sequences of these commands. Hence, to complete the process, only one command is given with the name of the test case. All the required files are complied and executed as and when required by getting commands from the Makefile.

Lot many EDA Tools are used in the projects. There is a requirements of a Commanding these tools. Tcl is an interpreted language and Tcl programs are referred to as scripts. Tcl scripts have to be executed by a TCL shell, called tclsh.

## 5.3 OS & S/W

Software and Networking applications serve as a backbone for the complete project. It tends to make the project manageable. EDA Tools are the vital part of the project and are needed at each and every phase in the project. The applications and EDA Tools used and their importance are discussed below. Detailed discussion on each of them is given in the later chapter.

### 5.3.1 OS: Red Hat Linux Ver9.0

Operating System is basic need for any application. It is the base of the project. The Operating System installed in the work station is Red Hat Linux Ver9.0. 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.

### 5.3.2 S/W: Bugzilla, CVS

For the verification project, as the process progresses, many issues with the RTL arises. These issues are to assigned to the designers for modifying the RTL to resolve them. The status of the bugs changes as designer works on them. Many bugs have dependencies within. Hence, a bug management system is required. This system must be secure and must manage all the bugs requirements. Hence, a software application called BUGZILLA is used. Each engineer is having a unique id to file, modify are see the status of the bugs. All the reference of the bugs are managed here for future reference.

Like managing the bugs, it is also require to manage all the test cases, designs, environment, and all the needful files. Many engineers work on common files and hence anyone can change it. Such changes should not affect work of other engineer. Hence, this is to be managed perfectly. For this, CVS (Concurrent Versioning System) is used. This is a source code management software. All the source files are managed in a common

server called as a CVS server. All the engineers working on the project checks out required files from this server. This is using there unique id. They work on this files and modify them. Now, when they want to check in this modified file into the CVS, a record is maintained. A log is generated which keeps track of all modifications done. Hence, when a modified file is checked in, it is logged with a brief comment on the modification made. The file is then checked in with a new version. Hence, when ever someone checks out the file, he/she can get any version of the file required without being affected by the modifications made by someone else.

### 5.4 EDA Tools: Affirma NCsim ver5.3, Incisive, Simvision: Cadence

EDA Tools are the heart of the project. Tools used for major processes through out the project are discussed here.

#### 5.4.1 Compilation & Simulation: Affirma NCsim ver5.3 & g++ compiler<br>Waveform viewer: Simvision<br>Code Coverage: Incisive

The VE is based on C/C++ and verilog. For C/C++ codes, freeware g++ compiler from GCC is used. This For the verilog codes, Affirma NCsim ver5.3 from Cadence is used. The simulation by the simulator is done remotely and the transactions in the signals generated is captured and viewed by Simvision from Cadence. This tool aids a lot to the task of debugging. For the code coverage by the test cases, Incisive from Cadence is used. As much of the design is synchronous and based on many interacting FSMs, the FSM coverage is important. This tool extracts FSM from the design and finds the coverage for it. This serves as input for modifying and adding new test cases to verify the design.

### 5.5 Identification of features

This is the first step to the verification. It is important to know what is to be verified. I my project, i had to refer to two specifications. They are as below:
**SONET/SDH Standard** :
As the chip is based of processing of SONET/SDH standard, the first specification to be understood and interpreted was the SONET/SDH standard. This standard is little bit complicated as it involves much of overhead bytes, multiplexing and interleaving. The location of payload with the help of pointers is to be understood. Various mapping of Digital signals onto SONET/SDH payload is to be understood. Other important thing is APS (Automatic Protection Switching). The functioning of the chip is influenced by the data in the overhead bytes. Hence, to create new scenarios for verifying the functionality from all aspects, these specifications are to well understood and interpreted.

**CHIP Hardware Document** :

As per the requirement, the chip functionality is decided. Functionality to be implemented by each module is explained in this document. The addresses and configuration of all registers is explained here. Hence, the chip 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.

## 5.6 Developing the TestBench

The TestBench is written in Verilog HDL. The Testbench connects two sub modules MI and ME into one MIME for verification. The TestBench is applicable to all test cases of MIME module. The MI and ME module are instantiated in the Testbench. Other required sub modules like BFM are also instantiated in the Testbench. This Test Bench Works in conjunction with the generic Testbench that is applicable to all the modules. This Testbench is responsible for generating all the required clocks. The generic Testbench is based on Macros and complier directives. Hence, during compilation only the portion of the Testbench that is applicable to MIME module is complied. The Testbench is updated as and when required based on test case requirement and updation in the RTL.

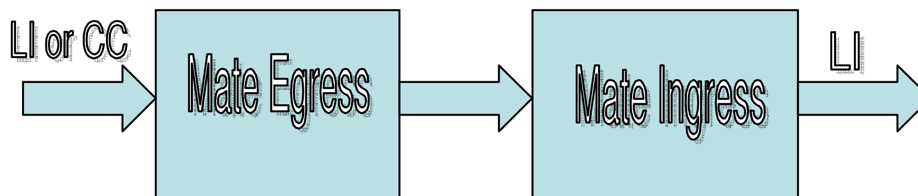



Fig5.1 MIME TestBench Configuration

One important thing is that it is possible that for verifying particular functionality, register is not present in the RTL. In such cases, software accessible registers are made in the reserved space of the addresses. This are then written in the Testbench as per the change in the signal and can then be accessed for verification.
**The Status byte case**: In case of Status byte, there was no register internal to the RTL to monitor the status of change in the signals with respect to the Status byte decoded in the incoming stream. In order the verify the Status byte decoding functionality, four software accessible registers were configured in the reserved space. This registers are to be read through the microprocessor interface. They reflect the status of the signals and hence are used to verify the functionality.

### 5.7 Prioritization of features

This is an important task as the aim of completing the verification at proper deadline is also equally important. Below are few points which are taken care of while verification.

1. The chip design consists of various configurable registers. These registers are used to configure functionality or for reflecting status. So, whenever any functionality is verified, this registers are to be verified first. The registers are of different types. Following are the type of registers:

- **R/W**: Regs can be read as well as written. They are normally use for configuration of the chip.
- **RO**: Regs can only be read. They normally reflect status of some alarm or interrupt.
- **RO/WT**: Regs can only be read but can be written for testing purpose. This is done by configuring some other global register.
- **COR**: Clear on read regs. They are normally interrupt registers which automatically gets cleared once they are read.

Again, some registers are global which are applicable to all modules. Hence, they should also be verified. These global registers will not be instantiated in the TestBench as they are not part of the MIME module. Hence, they are to be handled externally. All the registers are to checked for:

- Default/Reset value
- Dependencies
- Register type

The normal problem with registers is that of tristated. Due to lack of access in the RTL, they might remain unassigned and get tristated. Either some bit fields are some times complete register can be tristated. So, the first step to verification is to make sure that all registers are working perfectly.

2. The second important task is to verify that the RTL and VE is stable and ensure that both of them works fine in co-ordination with each other. This will ensure that there is no major issue in the VE or TestBench or RTL, because of which the test case might fail. Normally, this takes some time. This test is called smoke test. It can be thought of a top level test.

3. Now, some smoke test cases are written to verify that the RTL is stable for the major functionality. In this test cases, interest is to verify only the that RTL gets configure according to the top level configuration. No internal generation of any signal is looked upon.

4. Now, starts the verification of the inner functionality of the module. The level of

granularity is quite low. But here the functionality checked is sorted out as per the

importance. The first aim is that all functionality is to be verified. Then, depending on time and requirement, the granularity is reduced.

5. Other important point affecting the test cases is the fixing of bugs. Due to delay in fixing of some bugs, it is possible to start with test cases which are of some lesser priority.

### 5.8 Grouping of Test Cases

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 interrupt related functionality is responsible for generating many different interrupts 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. The cases are now written and executed with the aim of verifying functionalities implemented by the design. Following is the list of test cases identified for each feature.

| Sr. No. | Features Categorization | No. Of Test Cases identified |
|---|---|---|
| 1 | Reg At Reset & Reg R/W | 1 |
| 2 | Framer Sub module | 10 |
| 3 | Bit Shifting Through Streams Of Channels | 2 |
| 4 | Pointer Processing Sub Module | 12 |
| 5 | Interrupt Structure & propagation | 6 |
| 6 | Status byte decoding | 2 |
| 7 | Channel deskewing & frame Alignment | 2 |
| 8 | ME Framer Sub Module | 2 |

Table5.1 Features Of MIME Module With Identified Test Cases

### 5.9 Environment Settings

While starting to use the environment for the first time, it is required to make a new sandbox. This is the working folder where all the environment components, Test Bench and Test Cases will reside as per specific path. It is required to Check Out the environment from the CVS. It is possible to have multiple sandboxes as the project progresses. All test cases in specific sandbox will execute in the environment it sees within its sandbox. The only requirement for this is to set the variable proj accordingly. It is just required to type proj at the command line at the sandbox required to be worked with. Code Build and Code Compile will be done within the sandbox.

The second requirement is to set the SCOPE of working. This will indicate the module to be verified. For MIME module, it is required to execute the command “setenv SCOPE mime”. This will use all the MIME specific code from the Verification Environment.

### 5.10 Debugging The Test Cases

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.

1. Check the log file to check that whether simulation is completed or is left midway. This will also show the point of problem in terms of at processing of which stream, does the problem aroused. Normal problem is in the register mismatch and the aim is to find the problem for this mismatch. The RTL can be viewed for this to some extend.

2. If the log file is not having any issue, then the next step is to see if any thing is wrong in the overhead bytes compared by TWB. TWB will create a log for this.

3. If problem is not in the overhead bytes than it is in data integrity. So, it gives SPE check fail.

4. 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 filed.

### 5.11 Bugs Filing

As the verification progresses, issues arises. This is a good sign of progress. If no issue arise, 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 send to the designer who is assigned the bug.

| Sr. No. | Bug Id | Date & Time of Filing | Brief Summary |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 | 160 | 2006-10-10 04:57 | Pointer Processing Status Not Reflected. |
| 2 | 174 | 2006-10-16 08:44 | Reg Access Issues.<br>--Tristated Reg<br>--COR Not Working<br>--Reg r/w mismatch |
| 3 | 197 | 2006-10-26 20:43 | Malfunctioning Of Framer Alarm Interrupts. |
| 4 | 206 | 2006-10-31 23:29 | Pointer Processing Status Always returns LOP Indication. |
| 5 | 218 | 2006-11-08 06:03 | Status byte Signal Monitoring Mismatch. |

Table5.2 Bugs Filed With Brief Summary

## 5.12 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.

## 5.13 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 around 1 day depending on number of test cases.

## 5.14 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 code statements, fsm, 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.

Incisive from cadence is used as coverage tool. There are total 37 test cases and hence, coverage for the RTL will be the accumulation of coverage due to all these test cases. RTL consists of many modules. Hence, the coverage tool will show the coverage in graphical manner. For FSM Coverage, tool extracts FSM from each modules and shows the result for them. The range of coverage got for the test cases is as under:

**Statement Coverage: 70 – 75%**
**FSM Coverage: 90 – 97%**
**Toggle Coverage: 60 -70%**
**Branch Coverage: 70 – 80%**

The FSM Coverage is highest as mostly FSMs are involved in the initialization part of the design. Hence, for all test cases, FSM extracted are covered. For other coverage, value shows the range got. It was not possible to get high coverage here as few things can not be verified due to company's legacy and few features are for full chip level and are not to be verified at module level. Feedback was taken from this analysis, and 5 new test cases where written. Most selection was based on random selection and hence, coverage is affected but it ensures good coverage of corner cases. The identified new Test cases to cover remaining features are as under:

| Sr. No. | Features | No. Of Test Cases Identified |
|---|---|---|
| 1 | SS Bits interpretation | 2 |
| 2 | Channel disabling | 1 |
| 3 | P-DIP patterns | 2 |

Table5.3 New Test Cases identified after Coverage

A Snap Shot of the Coverage report is shown below.

```
========================================================================
=======

    HDL Analysis Successful:
       0 Errors,
       0 Warnings

========================================================================
=======

########################################################################
#########
[hdli] set_default_case_scoring -on

########################################################################
#########
[hdli] set_implicit_block_scoring -on -both
  47 instances affected

########################################################################
#########
[hdli] set_scoring_style -vector
  Done
  47 instances affected
```

```
########################################################################
#########
[hdli] set_coverable_operators -all

########################################################################
#########
[hdli] set_coverable_statements -all

########################################################################
#########
[hdli] select_coverage -instance -block -path -expr -nlevel *...
  338 instances selected

########################################################################
#########
[hdli] # Enabling fsm

########################################################################
#########
[hdli] extract_fsm -auto
  Extracting automatically recognized and tagged FSMs in all modules
...

==> USX Compile <==

  Compiling Module: <mich>
  Compiling Module: <li_lichx_rowcol>
  Compiling Module: <mi_rsohsi>
  Compiling Module: <mi_msohsi>
  Compiling Module: <micg_upif>
  Compiling Module: <ptrfifo_sw_sr_24x17>
  Compiling Module: <mihocgagg>
  Compiling Module: <mitoh_chtocg_buf_sw_sr_8x52>
  Compiling Module: <micsidec>
  Compiling Module: <hopi_ais>
  Compiling Module: <hopi_mast_slave_ais_intord>
  Compiling Module: <int_ext_ord>
  Compiling Module: <ext_int_ord>
  Compiling Module: <mi_upif>
  Compiling Module: <li_cgfa_shiftreg_clb>
  Compiling Module: <li_cgfa_bshift_clb>
  Compiling Module: <li_cgfa_fdet_clb>
  Compiling Module: <li_cgfa_b1calc>
  Compiling Module: <li_cgfa_b1agg_clb>
  Compiling Module: <descr_filter>
  Compiling Module: <descrambler>
  Compiling Module: <si_dskw_chwr>
  Compiling Module: <si_dskw_chrd>
  Compiling Module: <li_cgfa_dist_clb>
  Compiling Module: <li_cgfa_b2cconv>
  Compiling Module: <mifr_upif>
  Compiling Module: <si_dskw_cntrl>
  Compiling Module: <mifr_dmx>

========================================================================
======
```

```
    HDL Analysis Successful:
       0 Errors,
       0 Warnings

========================================================================
=======

Processing FSM: <TCA_FSM_r_chsel_micg_upif>

  Verifying hardware model...
  Creating internal representation...
  Analyzing design...

========================================================================
=======
  Error/Warning Report

========================================================================
=======

  File: /regression/dsheth/ruby2_mime_1/rtl/mi/micg_upif.v

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++
  Line Severity MsgID Description

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++
   368 USXWarn   126  Symbolic state <USX_S0> created for constant
value <0>
   452 USXWarn   126  Symbolic state <USX_S1> created for constant
value <1>
   505 USXWarn   126  Symbolic state <USX_S2> created for constant
value <2>
   429 USXWarn   126  Symbolic state <USX_S3> created for constant
value <3>

========================================================================
=======

    Controller Analysis Successful:
       0 Errors,
       4 Warnings

========================================================================
=======
Verifying hardware model...
  Creating internal representation...
   Line         578|   2193
```

Fig 5.2 Coverage Log’s Snap Shot

### 5.15 Documentation

The project ends with documenting all the work done. This will serve as reference for future and also it is to be submitted to the client to show the final status of the verification process in terms of various metrics of coverage. Major documents prepared are as under:

1. Documenting the Verification Environment: In this document, complete Verification Environment developed is explained with respect to each block and full chip level. The VE is generic and can be reused again with updated design.

2. Test Plan: The Test Plan is prepared at the start of the project which serves as specification for the verification task. Test Plan is updated as and when new features are added to the design and the new test cases are identified because of this and feedback from coverage.

3. Register and Pin coverage: This document shows all the registers and pins for the module and full chip level with name of test cases covering them and combinations of inputs applied. Inputs not applied are also to be specified. This can be because of leaving some combinations unverified due to deadline.

4. Traceability Matrix: This document corresponds to the final Test Plan. It maps each feature specified in test plan with test case executed. Features not covered due to shortage of time is also mentioned.

5. Bug Report: This document specifies all the bugs filed during the complete verification process with brief description of the issue, date of filing, test cases affected and its final status.

**Important Consideration**

**If a test case passes in earlier stage without any discrepancies, then it should be debugged to verify that it passed because of correctness of RTL and Testcase and not because it verified nothing.**

# 6

# *THE EC MODULE FOR BOARD BRINGUP VERIFICATION*

## 6.1 Introduction

In the year 1969, Gorden H. Moore gave an empirical relation that the count of number of transistors on the chip doubles every 18 months. This has been followed and the density and complexity of chip has gone on increased every year. The chip has become complex in from of SOC implementing the system onto a single chip. The chip has gone on shrinking as the gate length of transistors within has gone on reducing. Recently, Advanced Micro Devices have developed transistor with a gate length of 15nm. It can switch more than 3 trillion times in one second. Such advancement in transistor size reduction will further go on reducing the size of the chip and increasing the complexity of the logic implemented on the chip.

The chapter discusses the complete architecture and features of the chip. It discusses the protocols, the chip implements and the interfaces it support. The block diagram and the operation of the chip is described. The module EC which is verified for bringup is explained in detail. The actual chip on board is described at the end.

## 6.2 The Chip On Board

We are in the midst of a revolution sparked by rapid progress in digital image processing technology. Image Processing is considered to be one of the most rapidly evolving areas of information technology today, with growing applications in all areas of business. As such, it forms the basis for all kinds of future visual automation. Image Processing deals with images which are two-dimensional entities captured electronically through a scanner or camera system that digitizes the spatially continuous coordinates to a sequence of 0's and 1's. A digital image is a mapping from the real three-dimensional world to a set of two-dimensional discrete points. Each of these spatially distinct points, holds a number that denotes grey level or colour for it, and can be conveniently fed to a digital computer for processing. Here, processing essentially means decoding/encoding of image standards, algorithmic enhancement, manipulation, or analysis (also understanding or recognition) of the digital image data. Every image processing technique or algorithm takes an input, an image or a sequence of images and produces an output, which may be a modified image and/or a description of the input image contents.

At each stage of digital video signal transmission and reception, various standards are implemented. The intermediate devices need to decode/encode various compression standards and communication protocols. This task requires processing of large number of data. Hence, the much of parallel processing is required. Again, the input and the output source can vary depending on final application of the data. Fig 6.1 shows a hypothetical chip required for any system implementing application for video processing.

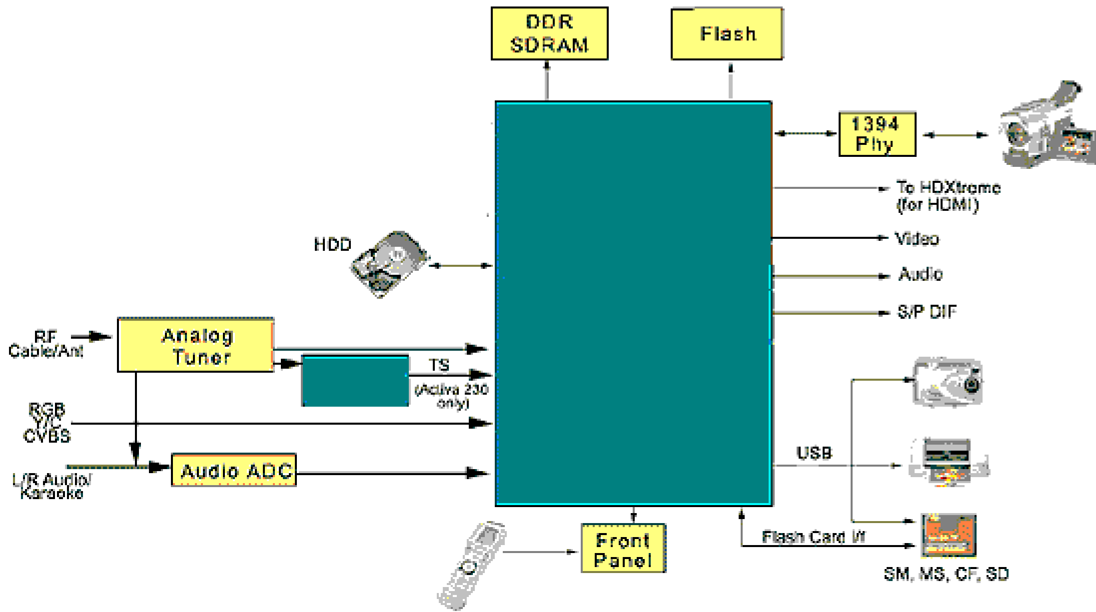



Fig 6.1 Block Diagram Of Media Chip With its Interfaces

The first chip that taps out from the fab is tested to ensure that the operation of the chip is same as that implemented at the time of chip RTL tap out. This is to ensure that the fabrication process has not introduced any wrong or missing logic in the chip due to some fault in the fabrication process. The COB is a widely used term to describe such first chip on board. The physical environment in which chip is introduced is precisely duplicated in the bringup verification environment. The chip bringup test cases are simulated with exactly same conditions as faced by the real chip on board. Hence, after ensuring that test case passes for simulation, it is executed on board.

The board for the chip is developed in the phase when chip is being fabricated. As and when the first chip is out from the fab, it is inserted into the board for testing. The board is a platform for the chip similar to one it will find when used by the companies for its application. This can be said to be an application board with all peripheral devices like input/output sources, memory, controllers, crystals, etc. Peripheral input and output ports are also configured with proper logic for testing purpose. The devices interfaced with the chip are same as used in simulation. For, simulation, IP core of these devices is used. Common global reset switch is provided to reset complete system along with the chip.

The PVP chip on board is interfaced with devices through its interfaces as discussed previously.

## 6.3 Protocols/Standards

The Protocols and Standards supported by the chip are described below.

### 6.3.1 MPEG-2

MPEG-2 is a standard for "the generic coding of moving pictures and associated audio information [1]." It describes a combination of lossy video compression and lossy audio compression (audio data compression) methods which permit storage and transmission of movies using currently available storage media and transmission bandwidth.

It is widely used around the world to specify the format of the digital television signals that are broadcast by terrestrial (over-the-air), cable, and direct broadcast satellite TV systems. It also specifies the format of movies and other programs that are distributed on DVD and similar disks. The standard allows text and other data, e.g., a program guide for TV viewers, to be added to the video and audio data streams. TV stations, TV receivers, DVD players, and other equipment are all designed to this standard. MPEG-2 includes a Systems part (part 1) that defines two distinct (but related) container formats. One is Transport Stream, which is designed to carry digital video and audio over somewhat-unreliable media. MPEG-2 Transport Stream is commonly used in broadcast applications, such as ATSC and DVB. MPEG-2 Systems also defines Program Stream, a container format that is designed for reasonably reliable media such as disks. MPEG-2 Program Stream is used in the DVD and SVCD standards. MPEG-2/System is formally known as ISO/IEC 13818-1 and as ITU-T Rec. H.222.0.

The Video part (part 2) of MPEG-2 is similar to MPEG-1, but also provides support for interlaced video (the format used by analog broadcast TV systems). MPEG-2 video is not optimized for low bit-rates (less than 1 Mbit/s), but outperforms MPEG-1 at 3 Mbit/s and above. All standards-conforming MPEG-2 Video decoders are fully capable of playing back MPEG-1 Video streams. MPEG-2/Video is formally known as ISO/IEC 13818-2 and as ITU-T Rec. H.262. With some enhancements, MPEG-2 Video and Systems are also used in most HDTV transmission systems.

The MPEG-2 Audio part (defined in Part 3 of the standard) enhances MPEG-1's audio by allowing the coding of audio programs with more than two channels. Part 3 of the standard allows this to be done in a backwards compatible way, allowing MPEG-1 audio decoders to decode the two main stereo components of the presentation. Part 7 of the MPEG-2 standard specifies a rather different, non-backwards-compatible audio format. Part 7 is referred to as MPEG-2 AAC. While AAC is more efficient than the previous MPEG audio standards, it is much more complex to implement, and somewhat more powerful hardware is needed for encoding and decoding.

### 6.3.2 H.264

The H.264/AVC (Advanced Video Coding) is a standard that is capable of providing good video quality at substantially lower bit rates (e.g., half or less) than previous standards (e.g., relative to MPEG-2, H.263, or MPEG-4 Part 2), and to do so without increasing the complexity of design so much that it would be impractical (excessively

expensive) to implement. It also provides enough flexiblity to allow the standard to be applied to a wide variety of applications (e.g., for both low and high bit rates, and for low and high resolution video) and to make the design work effectively on a wide variety of networks and systems (e.g., for broadcast, DVD storage, RTP/IP packet networks, and ITU-T multimedia telephony systems).

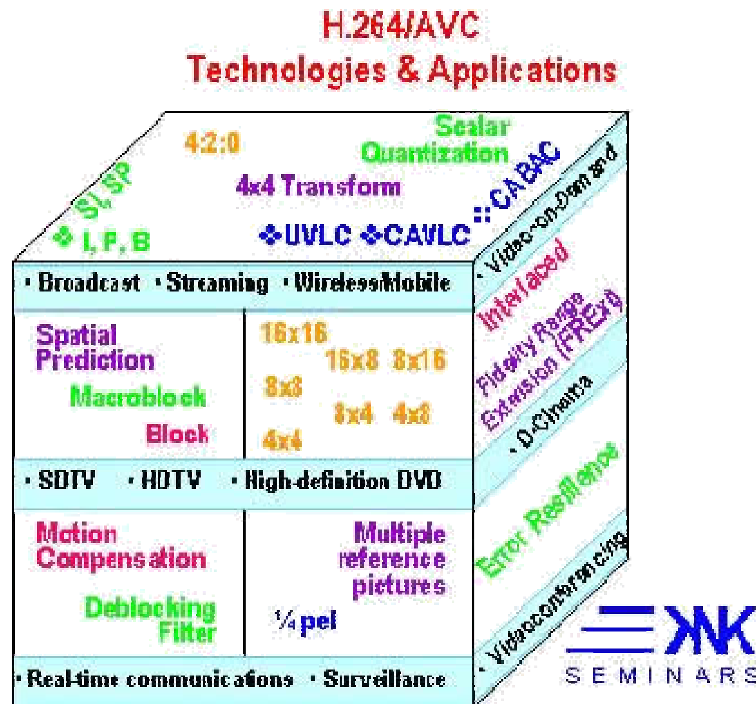



Fig 6.2 H.264/AVC Technologies & Applications

H.264 is the next-generation video compression technology in the MPEG-4 standard, also known as MPEG-4 Part 10. H.264 can match the best possible MPEG-2 quality at up to half the data rate. H.264 also delivers excellent video quality across the entire bandwidth spectrum — from 3G to HD and everything in between (from 40 Kbps to upwards of 10 Mbps). H.264, MPEG-4 Part 10, or AVC , is a digital CODEC (Coder/Decoder) standard that is noted for achieving very high data compression.

### 6.3.3 CABAC

CABAC stands for Context Adaptive Binary Arithmetic Coding. CABAC is a coding scheme for H.264. It achieves good compression performance through (a) selecting probability models for each syntax element according to element's context. (b) adapting probability estimates based on local statistics. (c) using arithmetic coding.

It makes only binary decisions. Any non-binary valued symbol (eg transform co-efficient or motion vector) is binarized prior to arithmetic coding. A context model is used for this. This is a probability model for one or more bins of binary symbol.

### 6.3.4 VC1

VC-1 is the informal name of the SMPTE 421M video codec standard initially developed by Microsoft. WMV3, better known as Windows Media Video 9 codec, served as the basis for development of the VC-1 codec specification. VC-1 is an evolution of the conventional DCT-based video codec design also found in H.261, H.263, MPEG-1, MPEG-2, and MPEG-4. It is widely characterized as an alternative to the latest ITU-T and MPEG video codec standard known as H.264/MPEG-4 AVC. VC-1 contains coding tools for interlaced video sequences as well as progressive encoding. The main goal of VC-1 development and standardization is to support the compression of interlaced content without first converting it to progressive, making it more attractive to broadcast and video industry professionals.VC-1 minimizes the complexity of decoding high definition (HD) content through improved intermediate stage processing and more robust transforms. As a result, VC-1 decodes HD video twice as fast as H.264, while offering two to three times better compression than MPEG-2
Since VC-1 is optimized for decoding performance, it ensures a superior playback experience across the widest possible array of systems regardless of bit rate or resolution. These systems range from the PC (where VC-1 playback at 1080p is possible), to set-top-boxes, gaming systems, and even wireless handsets.

### 6.3.5 ITU-656

The ITU656, describes a simple digital video protocol for streaming uncompressed PAL or NTSC Standard Definition TV (525 or 625 lines) signals. The protocol builds upon the 4:2:2 digital video encoding parameters defined in ITU-R Recommendation BT.601, which provides interlaced video data, streaming each field separately, and uses the YCbCr color space and a 13.5 MHz sampling frequency for pixels.

The standard can be implemented to transmit either 8-bit values (the standard in consumer electronics) or 10-bit values (sometimes used in studio environments). Both a parallel and a serial transmission format are defined. . Horizontal scan lines of video pixel data are delimited in the stream by 4-byte long SAV (Start of Active Video) and EAV (End of Active Video) code sequences. SAV codes also contain status bits indicating line position in a video field or frame. Line position in a full frame can be determined by tracking SAV status bits, allowing receivers to 'synchronize' with an incoming stream.

Individual pixels in a line are coded in YCbCr format. After a SAV code (4 bytes) is sent, the first 8 bits of Y (luma) data are sent then 8 bits of Cb (chroma U), followed by 8 bits of Y for the next pixel and then 8 bits of Cr (chroma V).

**The Chip Interfaces**

### 6.3.6 Video Interfaces

The chip supports 2 Video Input streams and 2 Video Output Streams simultaneously.

**MPEG I/P and O/P Ports**

The chip supports synchronous serial (SSI) or synchronous Parallel (SPI) interfaces of MPEG data as per EN 500083-9:1998. SPI is an interface for a system for parallel

transmission of variable data rates. The data transfer is synchronous to the 13.5 MHz byte clock of data stream which is the MPEG Transport Stream. Data transmitted are MPEG Transport Packets with 188 or 204 bytes.

**Video I/P and O/P Ports**

The Video DMA accepts YCbCr (multiplexed or separated) or RGB (Separated) digital video data. The format of each input is software selectable. The Video DMA generates YCbCr (multiplexed or separated) or RGB (Separated) digital video data. The format of each input is software selectable. 4 different modes are provided which can be selected through software for selecting appropriate input and output.

### 6.3.7 Audio Interfaces

The chip supports 2 Audio input stream and 6 Audio output streams simultaneously.

**I2S I/P and O/P Port**

Input ports A and B consist of one I2S receiver each. Audio DMA receiver accepts their input in I2S format. I2S itself consists of 3 1-bit serial input signals (Bit clock BClk, word select WS, and serial data SD) consisting on 2 time multiplexed data channels. It acts as a slave with regards to external devices. sclk is input from external devices.

4 Output Ports are indirectly sent to DAC to produce associated 8 channels of analog audio stream. It always operate as a master with regard to the external device. sclk is always given out for the external device.

**Digital Audio Interface**

An Audio stream of uncompressed IEC 60958 Type I format or compressed IEC 61937 format is intended for an SPDIF transmitter which will produce the electrical or optical output conforming for SPDIF.

### 6.3.8 Flash Interface

The chip interfaces to a NOR flash, a single device of 32Mbits with an 8 bit data bus. The byte wide data appears on the lower 8-bit data bus. All read, program and erase are performed using a single power supply. The device can be programmed using simple EPROM programmer. Specific commands are issued in order to read, erase, or program the external flash memory. Bits [31:28] of the Flash Timing Register (FTR) will specify the command. The Flash interface will use the parameters written into the Flash Timing Register (FTR) to generate the appropriate sequence of signaling for the external flash memory. The initial settings will be determined during the chip bring-up.

### 6.3.9 DRAM Interface

The chip interfaces to 64 bit DDR SDRAM. Multiple memory devices are used to form the 256Mbytes of DDR memory. The DRAM Controller presents a synchronous memory mapped profile of the external DRAM to the QAHB bus. The DRAM controller has

36x32 bit control registers which are configured by the device driver. The DRAM controller has several programmable features managed by the device driver.

### 6.3.10 I2C Interface

I2C is an acronym for Inter Integrated Circuit bus. I2C is a 2-wire serial interface standard defined by Philips Semiconductor in the early 1980's. It's purpose was to provide an easy way to connect a CPU to peripheral chips in a TV-set. The BUS physically consists of 2 active wires and a ground connection. The active wires, SDA and SCL, are both bidirectional. Where SDA is the Serial DAta line and SCL is the Serial CLock line. The key advantage of this interface is that only two lines (clock and data) are required for full duplexed communication between multiple devices. The interface typically runs at a fairly low speed (100kHz to 400kHz). With I2C, each IC on the bus has a unique address. Chips can act as a receiver and/or transmitter depending on it's functionality.

The I2C interface is used to configure the off-chip peripherals. The I2C interface is therefore a master slave port. This means that more than one device capable of controlling the bus can be connected to it. It should be noted that the master-slave relationships are not permanent, but only depend on the direction of data transfer at that time.

### 6.3.11 PCI

The Peripheral Component Interconnect, PCI, specifies a computer bus for attaching peripheral devices to a computer or any intelligent device. These devices can take any one of following forms. An IC or an expansion slot.

The chip PCI interface is a 32-bit, 66MHz, PCI REV 2.2 compliant interface. This PCI interface is functionally verified using PCI REV 2.3 card edge interface, device and a PCI Analyzer.

### 6.3.12 GPIO

The General Purpose IO interface can be used for many low bandwidth functions such as bidirectional serial control of peripherals, interrupts, leds, etc. They can be used to display different patterns during the testing process indicating different stages the test case proceeds to. This is total 16 in number and can be made to use in different ways by adding any external logic to them.

### 6.3.13 EJTAG/BSCAN

The CA1024-PVP includes a single IEEE Std 1149.1-1990 interface that supports both the boundary scan and the MIPS EJTAG chains. EJTAG provides a standard debug I/O interface, enabling the use of traditional MIPS debug facilities on system-on-a-chip

components. In addition, allows a MIPS processor. Both instructions and data can be accessed in EJTAG memory, which allows debugging of systems without requiring the presence of a ROM monitor or debugger scratchpad RAM. It also provides a communications channel between debug software executing on the processor and an external debugging agent.

### 6.4 The Enhanced Co-processor Module

The EC module is the Enhanced Co-processor. This is the module on which I have worked for the bringup verification. The EC module is a co-processor to the Video Processor. Its registers and program/data memories are mapped in the VR range. This locations and registers can be accessed only by VR. The internal local registers for each processing element can be accessed only by that PE. The EC module is also called CABAC as its main application is for decode/encode of the CABAC. It is a MIMD (Multiple Instruction Multiple Data) architecture processor. It has 8 processing elements which can work independently and together to accomplish a task. Each PE can work in different modes depending on the configuration. It is a very fast device and operates to complete the complex task for the VR.
The major features of the EC module are stated below:

(1) MIMD architecture of 8 tightly coupled processing elements.
(2) Powerful Instruction set to develop any complex logic.
(3) Explores instruction level parallelism.
(4) H.264 CABAC decoding at residual coefficient level.
(5) H.264 CABAC encoding at residual coefficient level.
(6) Two modes of operation for ALU and MU within each PE.
(7) Special tree mode for Variable length decoding.

The EC is a MIMD array of 8 tightly coupled processing elements (PEs). EC is able to exploit instruction-level parallelism, even for highly branched code, using speculative instruction execution. Each PE is like a microprocessor having ALU, MU and BU. They have their own program and data memory. They have 8 16-bit local registers which can be accessed only by them. Apart from this, 8 16-bit global registers are present for each PE to interact with each other. Any PE can access any of these global registers. A 16-bit global predicate register is present which can be accessed by any PE and VR. Hence, this can be used as a common resource between EC and VR.

EC performs computational intensive tasks under the control of VRISC:

- EC program memory is mapped in VRISC address space and will be loaded by VRISC.

- EC configuration registers are mapped in VRISC address space.

EC can perform applications that contain both a very short loop which needs to be accelerated as much as possible (ideally 1 cycle per loop) and a control program which feeds the high speed loop with data and modifies the parameters of the loop. The control program has different processing requirements than the loop - it has to be able to efficiently execute complex control functions with highly branched code.

When the processing loop is accelerated to 1 cycle/loop, EC can be seen as a programmable circuit, where:

- instruction registers determine the function of the circuit
- ALUs are used to implement the logic
- the operand registers store the state of the circuit
- special function registers provide IO connections

# 7

# THE BRINGUP VERIFICATION ENVIRONMENT

## 7.1 Introduction

The word bringup deals with processing some application on the first version of any thing. Hence, it must be precise and complete. The phrase bringup verification deals with ensuring that the first version operates as it was intended to do. The phrase bringup verification environment deals with the conditions that truly resemble a real world situation faced by the first version of a chip. Hence, a generic environment, which correctly resembles the chip on board environment, is said to be bringup verification environment.

This chapter discusses the Bringup Verification Environment developed for the EC module. It starts with all the characteristics possessed by the VE to make the process of verification complete. The code generation and the data/control flow through the VE is discussed. The challenges faced during the development of VE and the learning from them are stated. The chapter ends with a brief note on assertions and the directory structure maintained.

## 7.2 Characteristics Of Bringup Verification Environment

The Bringup VE needs to be precise, complete and generic. The characteristics of the Bringup VE developed for the EA module of PVP chip is stated below.

(1) The environment is generic for all modules on the chip. Same environment is applicable for bringup verification of all the modules of the chip. This is done by accommodating proper switches and defines.

(2) The Bringup VE realizes complete dataflow through the board. The dataflow from external input sources to external output sources through the chip is implemented. All EC to MIPS interaction is realized in VE.

(3) Precise assertions are kept so that the test case is simulated as per proper behavior required. An abnormal behavior ends the simulation with proper note.

(4) Initial Configuration of all the Registers is done at reset. Booting code is kept for this.

(5) A common approach is kept to develop any test case. Hence, little glue logic is required when the test differs in terms of requirements. The VE provides common procedure for testing of all instructions of the EC module.

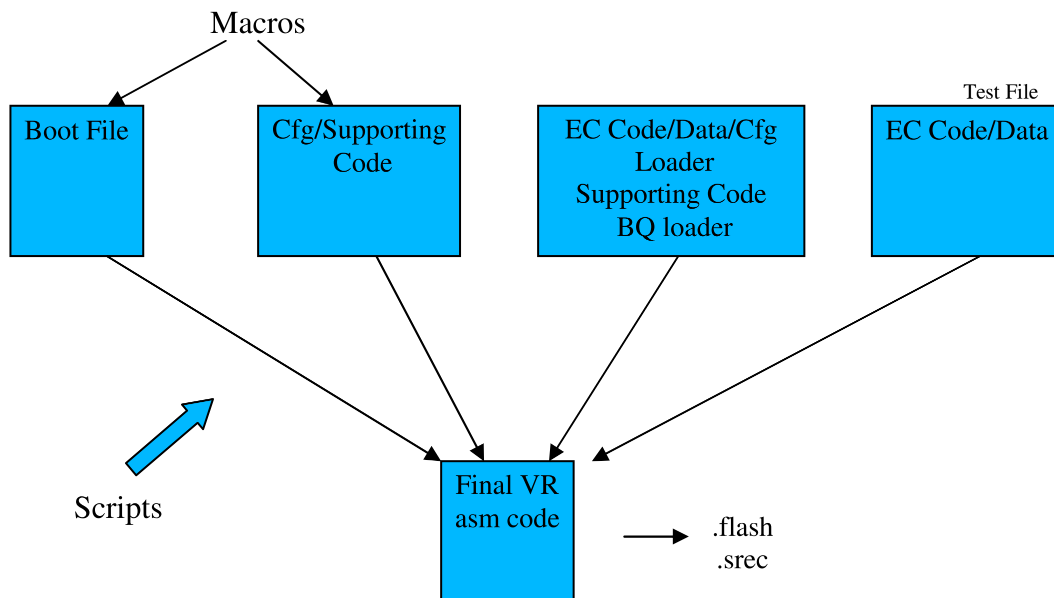
(6) Proper Regression suits are maintained for the test cases simulations.

(7) The execution flow as the test case simulates is shown at the output as some specific messages to depict the path which test cases follows. This helps to easily debug the test logic on its failure.

(8) All required log files and waveform files (wlf and evcd) are generated for test case debug.

### 7.3 The Bringup Code Generation for EC

The EC module is a processor with 8 processing elements executing simultaneously. The execution of the test case goes through the Video Processor and the EC module. The first step is the generation of the code to be executed. The code is in MIPS assembly language.




Courtesy: eInfochips

Fig 7.1 Bringup Code Generation For EC

The final code for MIPS execution is generated by the VE using different codes from various files.

(1) **Boot File**:
The boot file contains the boot code for all the processors of the chip and the initial configuration. The first step is the enabling of the CPU required. For EC module, only

Video CPU is enabled as EC is a co-processor to it. All other CPU and modules are kept in reset state. The next step is to do all the initial configurations. This includes the flash

timing configuration, DRAM Controller configuration and other register initialization. The next part follows the enabling and initialization of the interrupts. Depending on requirements, the interrupts are enabled and the vector addresses are configured. The execution is transferred to other file for more configurations.

The Boot file also contains the Interrupt Service Routine and the Routine for handling any exception. On any interrupt or system exception generation, the CPU jumps to a defined vector location based of its configuration. Routines to execute them are organized at those locations.

(2) **Cfg/Supporting Codes**:
This file contains all other required configuration of the registers. Depending on the test case type of the EC module, some specific configuration of its registers and some memory locations are required. Hence, depending on the type, the code selectively configures as needed. The enabling, masking and status clearing is done in this file. The program execution transfers to required file depending on the test type.

(3) **Macros**:
A separate dedicated file is maintained with all the macros. This file contains all commonly used logic like delay, memory access with defined offset, 32-bit register initialization, selective branch, displaying messages on output port, common configuration, etc. These macros are used in all the files for better code. This file is updated with new macros as and when required.

(4) **EC code/data/cfg loader code**:
The EC module is a co-processor to the Video RISC. Hence, all the program/data memory and the configuration registers are to be loaded by the VR. A configuration file is maintained for this. It contains paths to the program and data files of each processing element of EC. These code and data is loaded at respective program and data memories of the processing elements by the VR. The file also gives the path for the configuration of the registers of the all PE. It gives the binary data file to load the data in the bit buffer of EC module specifying its starting address and size. The paths specified in this file are used by the loader code.

Four loader codes are kept to for loading EC. As EC is a slave to VR, the VR executes the loader code. The loader codes are for loading program memories of all PE, data memories of all PE, configuration registers of all PE and the bit buffer of EC. EC takes this code and data for EC, as data and loads them to appropriate locations. The loader codes are in C language and sde-gcc cross compiler is used to cross compiled it into MIPS code.

(5) **EA Code/Data**:
As explained before, the paths of these files are specified in the configuration file. Total 8 files having code and data for each PE of EC module are written. It contains code in the

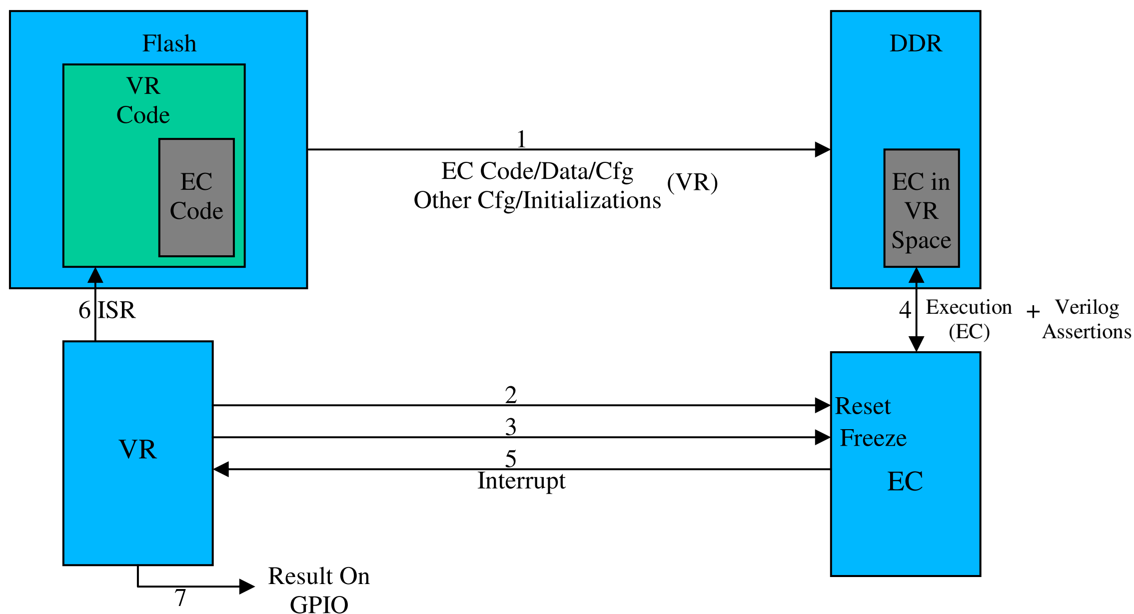
assembly language of EC module. EC assembler is used to get the hex codes of it and these hex codes are then loaded into respective memory locations by VR as if they are data. These file is the actual test case that verifies the functionality of the EC module.

(6) **Scripts**:
Scripts are used at all stages in the complete bringup code generation. This scripts automates the complete process of compiling, assembling, linking and optimizing. The files used are in three languages namely C, MIPS assembly and EC assembly. Hence, the script merges all of them and generates a final hex file for the VR to execute. Makefiles are used for this. The scripts selects appropriate segments based on the defines. These scripts used are in perl and shell.

(7) **Final VR hex code files**:
The final files generated are the hex code for the VR. Files with different formats are generated. Text file is generated which has the final assembly code. It specifies each mnemonic with the location where is it is stored. This file is used for debugging. The flash file generated is the file which is loaded in the flash and executed by the VR then after. The .srec file generated is to be used by the flash programmer for programming the flash on the board.

### 7.4 The Data/Control Flow in Bringup VE

The VE being for Chip bringup, ought to incorporate precise and complete flow along the datapath as seen by the EC module. The fig below describes the complete flow of execution through the environment.




Courtesy: eInfochips

Fig 7.2 Bringup Env Comp/Data/Control Flow

The execution sequence followed is described below.
(1) The final code for VR to execute is stored inside the flash. It contains the code, data and configuration for EC as well. As VR executes the boot code, it gets configured. The next step is to configure the EC. VR executes the loader code and loads EC code and data at its memories. The configurations are loaded in the registers. The data from the binary file is loaded in the DDR which will be further loaded into its buffer by its DMA.

(2) VR unresets EC. A common boot register is used for enabling/disabling of CPUs and modules of the chip. EC accesses it to get EC out of the reset state and make it ready for test code execution.

(3) The EC module has a added enabling called a freeze signal. This signal is basically for freezing/stalling the PC of all PEs of the EC module. It is an active low signal. VR deactivates it and get EC out of the freeze state. All PEs of EC start executing the code in their program memories.

(4) All PEs of EC executes their test code. VR also executes further. This is done under the control of assertions. They halt simulation if any improper behavior of EC or VR is detected.

(5) The EC executes the test code and generates results. The test cases are made self checking. At the end of the test code execution, EC interrupts VR.

(6) As VR gets interrupt from EC, it gets jump to the vector location and starts executing the ISR. Further interrupts is disabled and the status register is cleared. EC is freeze and the logic for detecting the test case result is executed. Before EC interrupts VR, it writes appropriate code in its respective memory to indicate the stages of code it executed and the final result of it. These locations are read by the VR and test case result is decided.

(7) The result of the test case and the reason of its failure along with the PE number which failed is displayed on the GPIO output port. Simulation is halted using assertion.

### 7.5 Critical Stages in Bringup VE development

The development of VE for the EC module faced many critical halts in between. The VE has to take care of all the limitations of VR and EC. Hence, proper sequence of configuration is to be followed. Some configuration is to be done only in a proper sequence else it has no effect. The common resources are to accessed with due care. In the following points, I discuss some critical problems faced during the VE development.

(1) **EC Freeze signal**

This is an active low signal configured through the boot register of the chip. This signal has to be activated and deactivated after the reset signal. If freeze is already deactivated and we try to get EC out of reset, the PC gets unknown value and gets XX.

Again, there are some configurations which have effect only when freeze is active.
(2) **Code Optimization**
For the RISC processor, a cross compiler is used for the C code. It optimizes the code by removing unnecessary NOPs and branches. This optimization creates problem when code is for interaction between EC and MIPS. Hence, the code is to be written is a manner which will take care of this.

(3) **ALU configuration**
For the ALU configuration of each PE of EC, particular pattern is written in the register. This register is to be written under different conditions for different configuration of other registers else it do have any effect.

(4) **DRAM Initialization**
The DRAM controller is to be initialized before accessing any DDR location. This is done by configuring 32 32-bit registers. Any improper configuration makes particular section of DDR inaccessible.

(5) **Synchronization between EC and VR**
The operating frequency for EC and VR is different. EC is must faster as compared to VR. Hence, while interaction between EC and VR, proper synchronization is to be done. Again, proper synchronization between the 8 PEs of EC is to be taken care of. All PEs have common reset and freeze signal. Hence, if any one PE is required to execute some logical code, other PEs must synchronize with it by executing some sort of NOP loop.

(6) **The unknown XX**
When ever, some memory location is read before initializing it, the content read is XX. This leads to RTL getting XX and the simulation halts. Same is a case, if comparison with some XX data is made or there is no NOP after any branch code. This is not a situation with test executing on board as physically there is nothing like unknown value XX.

(7) **Global Condition bits**
The Global condition bits are not initialized to 0x0 on reset. Hence, VR has to explicitly make it 0x0 else it will be XX when selected and will ultimately result in RTL getting XX and simulation getting halted.

### 7.6 Verilog Assertions

The execution of all tests in the bringup VE occurs in presence of the Verilog assertions. These are assertions for RISC. Assertions basically take care of any requirement or assumptions we want to adhere to. It fundamentally takes care of how a particular design

should behave or should not behave. They can be said as “expression of design intends” or “designer’s assumption”. They are to prevent any improper behavior of VR or EC to be carried forward. If the simulation is not halted at this point, it might result into unpredictable results at the later stage. As an example, whenever VR does comparison with any unknown data, the PC gets XX and the Verilog assertion halts the simulation by printing appropriate message. If this would not have been halted, the comparison might generate some unpredictable results which may ultimately make test case to execute in some wrong direction.

The final stage of halting simulation after displaying proper message for pass and fail, is handled using Verilog assertion. Two pins of GPIO are used for this namely x and y. If test case passes, x is made 1 else it is made 0. y is made 1 at the end. As y does a transition, the Verilog assertion checks x and depending on its value, displays appropriate message and halts the simulation.

# 8

# *THE BRINGUP VERIFICATION FLOW*

## 8.1 Introduction

This chapter describes the complete bringup verification of the EA module of the PVP chip. The complete chip is broadly divided into various modules. These modules are verified with their complete interactions with other modules and devices on the board. The chapter goes through the flow of the complete process of the bringup verification of the EA module. It starts with the test plan and ends with the actual test case being executed by the chip on board. It discusses the tester board in brief.

## 8.2 The Test Plan

After the chip RTL tap out, the development of its bringup gets started. The first step to it is the identification and categorization of the features to be tested. For this, the reference is taken from the verification plan at the functional verification stage. The abstraction level is quite high now and more of the interest is to exercise the module's data paths. Hence, according changes are made in the plan and a test plan describing the tests to be performed on the chip is laid out. The major input is taken from the original verification plan, final updation to the RTL and application of the module.

| | | | | | |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | [-] -- | Vid | EC | KillEC | Check that EC can be stopped. |
| 3 | | | | | |
| 4 | [-] -- | Vid | EC | FreezeEC | Check EC freeze. PC resumes on unfreeze |
| 5 | | | | | |
| 6 | [-] -- | Vid | EC | VRRdWr | Video RISC read/write PE data memory. |
| 7 | | | | | |
| 8 | [-] -- | Vid | EC | VRBQ | Use VR to activate BQ, then consume BQ bits. |
| 9 | | | | | |
| 10 | [-] -- | Vid | EC | ECInstr | Test basic EC instructions, memory and registers |
| 11 | | | | | |
| 12 | [-] -- | Vid | EC | ECTreeMd | Test tree mode instruction and operation |
| 13 | | | | | |
| 14 | [-] -- | Vid | EC | ECCktMd | Test circuit mode operation. |
| 15 | | | | | |
| 16 | [-] -- | Vid | EC | ECBQ | Test EC consuming bit queue (through EC instruction). |
| 17 | | | | | |

Courtesy: eInfochips

Table 8.1 Test plan for the EC module

Each test name in the test plan describes a group of test cases to be simulated. All these test cases have some thing in common and hence are taken into same category. The points considered while finalizing the test plan are
Functionality and data path followed
Common features to be tested
Common test scenarios required to be generated
Based on application

(1) **KillEC**: This is the test to check that EA is reset and unreset through VR.

(2) **FreezeEC**: This is the test to check that EC is made to freeze on activating the freeze signal through VR. The EC PC gets freeze and on unfreezing, it resumes from the point it stopped.

(3) **VRRdWr**: This test case is to access the data memories of all PEs of the EC module. The Video RISC can access the data memories of EC. Hence, the test case makes VR load data into the PE data memory and read it back and compares to verify it. This is a basic smoke test. This is the first test to be simulated. All the PEs data memories are verified for different combinations of data. The data path here is the loading of data for EC from flash by VR and then storing it to EC data memory after required configuration and under proper conditions. For, example, VR should write to EC data memory only when EC is in reset state else it will not have any effect.

(4) **VRBQ**: This is also a basic smoke test to check that VR is able to load data at proper DDR locations from flash and then these data is transferred correctly inside the BQ buffer by its DMA. The BQ data is specified as a binary file. Starting location of the DDR and the length of data to be transferred is specified through the configuration file. This is a basic test for any BQ related and tree related tests.

(5) **ECInstr**: Module EC is a MIMD processor which 8 processing elements. It works in two modes namely Full Processor and Circuit mode. This test case includes tests for testing all the instructions in FP or normal mode. Each test is to be tested with different combinations of data. The test is to be done on all PEs or on some selected PEs.

(6) **ECTreeMd**: Tree mode is an extension of FP mode. This mode is entered from FP mode. It has its own instruction set. These tests are for variable length decoding like Huffman coding. The test is using different tree structures and binary data. This tree structure and data are given by the software team.

(7) **ECCktMd**: Circuit mode is a special mode of EC. In this mode, the PE configured in circuit mode executes a single configured instruction at each clock cycle. The task of providing data and command bits is done by other PEs. All PEs work in coordination for circuit mode task.

(8) **ECBQ**: This test includes all the tests for consuming the bits from the BQ buffer. It includes various configurations of the BQ registers and various interrupts generated for VR.

## 8.3 Bringup VE development

The next step after deciding the test plan is to create an environment for the bringup simulation of the tests to be written. As already discussed in the previous chapter, the VE must be generic enough for all the test categories and the modules. The ultimate aim is to develop a stable bringup duplication of the environment faced by the chip on board. The VE must be made as reusable so that it can be used again with minimum updation for second version of the chip. The VE is made to follow all the data paths and at each stage it is made to display proper messages onto the GPIO. This is to aid the debugging of the test and the test execution on board.

The VE is developed using different high and low level languages. It includes scripts in different languages. The VE is a combination of C, RISC assembly, EC assembly, perl, shell and TCL. The process of compiling every thing in sequence is automated by the scripts. Scripts are again used for the simulation of the test cases.

## 8.4 Simulating The Test Cases

After the VE is ready, test cases are developed. EC being a processor, each test basically tests one instruction. The different ways in which the instruction is tested is more important here. The table below shows the count of test cases written under each test name.

| Sr. No. | Test Name | No. of Test cases |
|---|---|---|
| 01 | KillEC | 01 |
| 02 | FreezeEC | 01 |
| 03 | VRRdWr | 01 |
| 04 | VRBQ | 01 |
| 05 | ECInstr | 97 |
| 06 | ECTreeMd | 3 |
| 07 | ECCktMd | 47 |
| 08 | ECBQ | 33 |

Courtesy: eInfochips

Table 8.2 The EC Test Cases Executed

A common sequence is maintained while writing all the test cases. The format preserved by the test is as described below. The first four tests are basically smoke tests for ensuring the basic operation.

(1) Test starts with initialization of the locations and counter. The counter keeps a count of the tests performed as the test proceeds. It indicates the test count of the test if it fails and the total number of tests if it passes. First two memory locations of each PE are initialized to 0x0. They are to be used at the end of the test for indicating the pass or fail code of the test and the test count.

(2) Initial selection of the global condition bits or global flags is done. Each PE can select its own combination or can have common global bits selected. These bits are reflected based on the result generated by the test and hence are used to indicate pass, fail or proceed.

(3) Initialization of the registers to be used for the test. These are registers, local or global, which will be used as operands or source or destination address. For global registers, the data is written after one cycle from accessing it.

(4) The instruction under test is executed. This instruction will be tested many times in the test with different combinations of data, sources and destinations.

(5) Compare the result with the expected value. This takes one more clock cycle due to pipelined architecture. The register is compared with expected value. The memory locations are compared with the expected values.

(6) Depending on the result, if the value matches the expected one, the test count is incremented and the test proceeds further for next test with different register data. If the value is not as expected, the test count and the subroutine code is stored in at the location and VR is given an interrupt. The subroutine code specifies whether it is register data mismatch or memory data mismatch.

(7) Finally, after all tests are completed and if everything goes fine, pass code pattern and test count is stored in the memory and VR is interrupted.

(8) EC now goes into an infinite loop. This is to ensure that EC do execute something until it is freeze by VR in the ISR.

A sample test case is shown below.

```
sample.asm + (~/Connex_new/Etna1.0/dv/bringup/diags/satests_FP) - GVIM
File  Edit  Tools  Syntax  Buffers  Window  Help

 1 .text
 2                  mv16i %l0, 0x0          //portingAddOn
 3                  st16i %l0, 0x0          //portingAddOn
 4                  st16i %l0, 0x100        //portingAddOn
 5                  jpt    0xF,0x5, 0x0     //portingAddOn: Jump To TestLabel0
 6
 7 SubRoutineTestInc: ld16i %l0, 0x100      //portingAddOn
 8                  adv5  %l0, %l0, 1       //portingAddOn
 9                  st16i %l0, 0x100        //portingAddOn
10                  rett  0xF, 0x0          //portingAddOn
11                  nop                     //portingAddOn
12
13
14 TestLabel0:
15    //[0]Begin of Test.
16            clrcnd 0xFFFF
17            cndsrc 0x0630 //Enabling EQ on PE1
18    //[0]Initializing  Resources.
19            mv16i %g4, 0x5960
20            mv16i %g4, 0x5960
21            mv16i %g3, 0xF9D7
22    //[0]Instruction Under Test.
23            nop
24            sub %g3,%g4,%g4
25            nop
26    //[0]Checking Resources.
27            ucpv16 %g3, 0x0
28            nop
29            jpt 0xC, 0x2, 0x0
30            callt 0xF, SubRoutineSAr2, 0x0
31            ucpv16 %g4, 0x5960
32            nop
33            jpt 0xC, 0x2, 0x0
34            callt 0xF, SubRoutineSAr0, 0x0
35            ucpv16 %g4, 0x5960
36            nop
37            jpt 0xC, 0x2, 0x0
38            callt 0xF, SubRoutineSAr1, 0x0
39    //[0]Disable Global Flags.
40            cndsrc 0x3F3F
41            clrcnd 0x8000
42    //[0]End of Test.
43            callt 0xF, SubRoutineTestInc, 0x0   //portingAddOn
44
45              //[3]Disable Global Flags.
46            cndsrc 0x3F3F
-- VISUAL --                                                    24,28        Top
```

Fig 8.1 Test Case Snap Shot

As the test cases are written depending on the priority, they are simulated. The normal sequence for simulation starts with VR, proceeds through EC and ends with VR again. The basic aim of these tests is that they should pass. These tests are already valid for the RTL which was tapped out. Hence, the these test cases must be made to pass in the bringup VE and then executed on the chip on board to check if some thing goes wrong. The basic steps are as under:

(1) Boot VR
(2) Register initialization and configuration
(3) Load EC program memory
(4) Load EC data memory
(5) Transfer BQ data from flash to DDR
(6) Configure BQ Registers
(7) Start EC
(8) VR ISR execution
(9) Display the final result

The later portion changes depending on the test type and its requirements. EC is restarted and reconfigured based on requirement of the test. EC – VR interactions is done through some memory locations or global condition register. The brief simulation flow for each Test is described below:

(1) **KillEC**: The EC test code maintains a counter and goes on incrementing it with each clock pulse. This counter is in form of a memory location. The VR reset EC after some time and reads this memory location. It compares EC registers and expects their reset value. EC is again restarted and reset again. The value read from the location this time is expected to be lower than the previous one as EC started executing the test code again from the starting and this time VR had not wasted any time before resetting it. Depending on the result, appropriate display message is given out on the GPIO.

(2) **FreezeEC**: The test code execution is same as the previous one but with the difference that EC is freeze and not reset. On reading the location again after unfreezing the EC, the expected count is greater as the EC restarted from where it stopped.

(3) **VRRdWr**: The simulation flow is basically in the VR only. EC is reset for this test. This test is basically a part of the VE itself. It tests that data path for EC data memory access through VR is working properly. In case of any mismatch between the expected and actual value, test case is resulted into fail. The test is debugged for making it pass.

(4) **VRBQ**: The simulation flow is basically in the VR only. EC is reset for this test. This test is basically a part of the VE itself. It tests that VR is able to read the flash memory and store the read data at the DDR. In case of any mismatch between the expected and actual value, test case is resulted into fail. The test is debugged for making it pass.

(5) **ECInstr**: The simulation flow of these test cases is exactly same as described in the previous topic. The ISR executed by VR goes through all the PE sequentially. Before EC interrupts VR, it stores the pass/fail code and the test count at its proper locations. ISR sequentially goes through PEs from 0 to 7 to detect for any failing code. If any failing code is detected, it displays the code, test count and the PE number on the GPIO. Hence, from the message on GPIO, complete status about the test is interpreted. Test case is debugged for failing conditions and is made to pass.

(6) **ECTreeMd**: The simulation flow of these tests is different at the later stage. It consumes the BQ data from its buffer. The test code is basically a tree structure used to decode Huffman coding. The BQ data are coded as per a particular compression pattern. Tree structure is defined to decode it. EC code has this tree table as a routine. The EC consumes the BQ data while executing tree routine and terminates when decodes a character. This character is stored in local memory and its content is compared with the expected one. This process goes on in a loop. At the end the EC interrupts MIPS with all the required codes and count.

(7) **ECCktMd**: The simulation flow for these tests requires interaction between EC and VR. PE which is configured in circuit mode executes same instruction every clock cycle. This PE requires data and command bits to generate and reflect the result. Some instructions require transition from FP to circuit mode and visa versa. The register content and the code are to be matched properly for this. As an example, one 32 bit

memory load instruction, loads data from memory only in its local register pair. These local register pair can be only accessed by the same PE. Hence, that PE is reconfigured into FP mode to check the result generated by it in the circuit mode. Reverse is true for memory store operation.

(8) **ECBQ**: The simulation flow for this test is similar to that of tree test. The simulation flow of VR shifts to altogether a new file for each test of BQ. This is because of the requirement of different initial configuration and reconfiguration. EC-VR runtime interaction is required. Some tests are EC self checking while others are checked by the VR. Some tests are for multiple generation of interrupt. This tests normally consume data in terms of KB. Hence, it lasts much long for simulation. Hence, many such tests are directly tested on board.

## 8.5 Instruction simulation

The test case is simulated rigorously with various possible combinations of data, registers and conditions. I shall explain this with an example of ADD and JUMP instruction.

### 8.5.1 ADD

The syntax for the instruction is ADD R1,R2,R3.
Operation: The content of 16-bit registers R2 and R3 are added and the result is stored in destination register R1. The carry is affected.

Possible combinations of test scenarios:
(1) ADD is executed with some data in local registers. Different combinations of registers are selected.
(2) ADD is executed with some data in global registers. Different combinations of registers are selected. The global registers depict the result after one clock cycle.
(3) ADD is executed with data generating carry and not generating carry.
(4) All PEs executes the ADD instruction simultaneously.
(5) Only one PE executes the ADD instruction and others only executes NOPs for synchronization.
(6) The ADD can be executed with ALU configured to reflect the final result on the destination register only at the time the command bit is generated by master-slave PE. The value is to be compared with correct value when command bits are generated. For other condition, the result is remains same as the previous one.
(7) ADD instruction is executed in circuit mode by configuring the ALU configuration registers. Again different combinations of registers can be selected.

### 8.5.2 JUMP

The syntax for the instruction is JUMP tab, offset, command bits

Operation: tab is nothing but a truth table with 2-bit input. Hence, it is of four bits. It specifies whether the JUMP is true or false based on the 2 global condition bits selected by the previous instruction. If both bits are 0, then JUMP is true if least significant bit of tab is 1, else JUMP is false. In case of JUMP being true, the PC is incremented by the offset specified and the execution shifts there. At the same time, 16 command bits are generated which governs different tasks of other PEs.

Possible combinations of test scenarios:
(1) JUMP is executed with different combinations of tab and different global condition bits selected.
(2) Jump is executed without selecting any global conditions bits.
(3) The range of offset for JUMP is +-128 bytes. The JUMP is made true to jump to different offsets.
(4) Different command bits are generated using JUMP instruction. JUMP generating command bits by getting executed in different PEs has different effect on other PEs. JUMP executed by master PE affects PE0 and PE1 partially. JUMP executed by slave PE affects PE3, PE2 and PE1 partially.
(5) JUMP is executed by different PEs with different PEs combinations.

A sample waveform file is shown below.

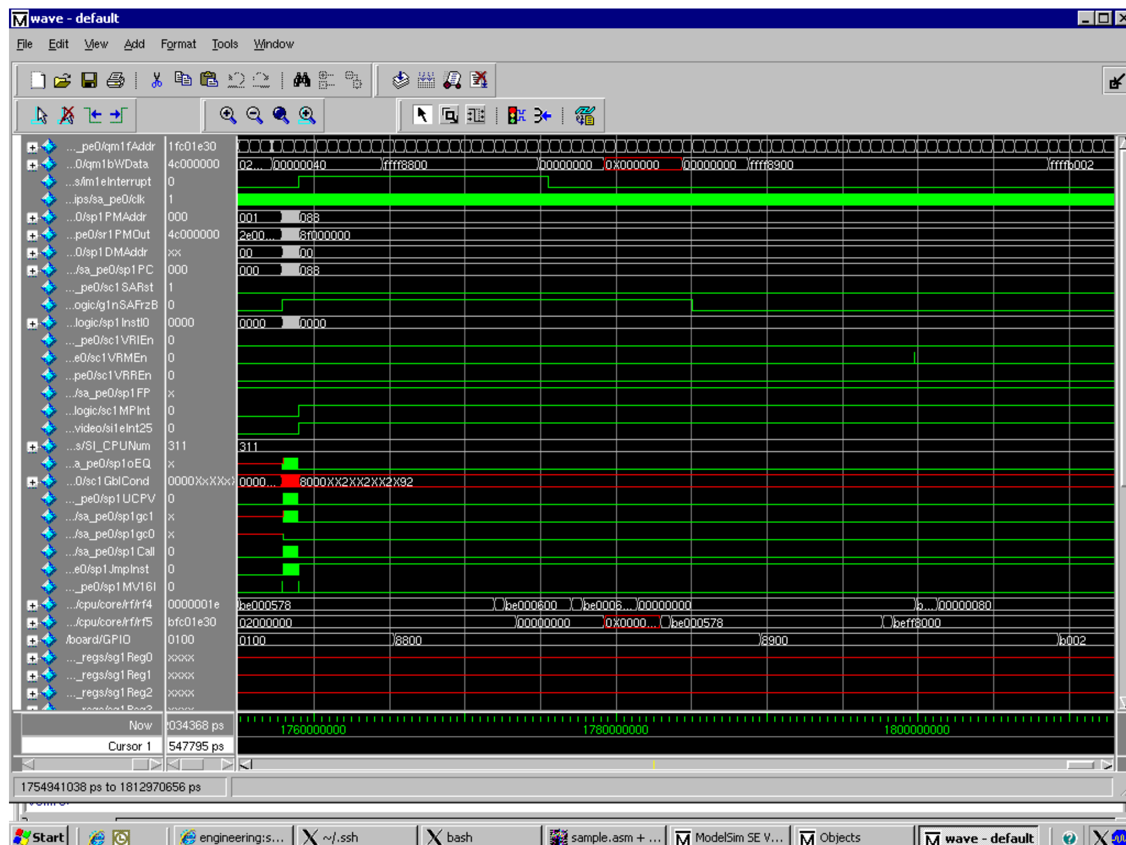



Fig 8.2 Waveform Snap Shot

### 8.6 Debugging the test cases

The ultimate aim for bringup verification is to pass all the written test cases as the RTL is already proved. The testcases are simulated in an environment similar to one on the board. Hence, once it is ensured that these tests pass in simulation, they are executed on the board. When ever any test fails, a proper procedure is adapted to debug it. The log files, waveform files and the display messages on the GPIO are traced for debugging the failed tests.

#### 8.6.1 GPIO and display messages

For failed tests due to assertions, appropriate message is displayed. Hence, the point where test went wrong is known. From GPIO messages, the PE number which failed is known along with the failing test count and test code. Hence, that section of the code can be rechecked to find any error. The intermediate GPIO messages can be traced to check the data path followed by VR. If the path followed is wrong, the VR code is to be rechecked.

#### 8.6.2 Dump and log files

The dump file contains the actual code executed by VR in its assembly language. It again states the PC content for each mnemonic. This file is used to trace the execution of the VR. If test has failed due to VR getting PC=XX, the reason can be as VR read some un-initialized memory location or made some comparison with XX data. Hence, the code needs to initialize that location or register before accessing it.

#### 8.6.3 Waveform files

This is the ultimate tool to debug. For any failed test cases, waveform files are used to see whether required signals are generated and required data is coming onto the data bus or not. For an example, the problem can be with the generation of enable signal. Address is proper on the address line and data is proper on the data bus, but the data in not getting written on to the memory location. The reason can be as the write enable signal is not generated. This signal gets generated due to some specific configuration. Hence, configuration needs to be changed. The waveform files are use together with the RTL to trace the signals.

#### 8.6.4 Design RTL

The design RTL is the actual stuff to be verified. It is the most important thing used for debugging the failed tests. Any data path can be traced through the RTL and the signals can be observed in the waveform. The point where the signal does not get the expected value is searched for drivers. The drivers are next to be traced.

## 8.7 Executing test cases on board

After the test cases are ensured to pass in the simulation, they are executed on board. The .srec file generated after compilation contains the code in .srec format so that it can be loaded into the flash by the programmer. This is stored using the universal scan software. The board is rest using the rest switch provided. Hence, the chip gets boot. The VR gets activated and executes the code. As it proceeds, the EC is activated. EC executes its code and generates interrupt. As the process goes further, messages of the data path followed by the test gets displayed on the GPIO. The GPIO is interfaced to the PC through USB and message gets displayed onto the PC screen. At the end, EC interrupts VR and final pattern of test pass or fail is displayed on GPIO. Test end code is displayed and the VR goes into an infinite loop. For debugging any failed test case, the pad pins of the chips are analyzed using logic analyzer. This board is used at later stage as a demo board for demonstrating the chip application.

## 8.8 Executing test cases on the tester board

Tester board is for rigorous testing of the chip with different combinations of data. Basically tester is big machine which gives different inputs to the driving pins of the chip as per our functional vector generation provided using EVCD files. So based on the inputs from EVCD file and output received by chip and outputs expected from EVCD file after simulation are compared and if there is any timing mismatch, you need to adjust it by changing the clock freq. Sometimes you might need to change the setup/hold time of diff. signals if your clock freq. is correct then also you have some timing mismatches, all of this can be done in tester house with software for tester board. Mismatch in the timing suggests that some signals must be delayed so that we sample them when they have the correct value. The tester board is too costly. On the tester board, only chip is present. All peripherals are not present.

Due to clock tree insertion at the time of synthesis, extra timing delay is introduced in the chip. Hence, the chip is not able to work at the same maximum frequency as specified on the board. Hence, testing it on the tester board with frequency adjustments tests the chip for finding the maximum frequency of operation.

# 9

# EDA TOOLS, S/W AND LANGUAGES

## 9.1 Introduction

This chapter discusses the EDA tools and the Languages used for bringup verification environment. It also discusses the application software used for maintaining the bug history. EDA tools are the must to do precise work through automation and reach the deadline. They aid the task of analyzing and debugging the flow of signals. The VE uses many scripts to automized the compilation, simulation and generation of files. These scripts are written in different languages. The chapter discusses the perl and shell used for scripting. The VE and the test code is a combination of different languages.

## 9.2 EDA Tools

EDA Tools plays very important role in this industry. For every task to be accomplished, these EDA Tools tend to reduce trhe time required and performs the task accurately. Development of EDA Tools are at the target of every vendors 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 dby me for the project. All design tools are by the company Cadence.

### 9.2.1 Compilation & Simulation
**Tool: Affirma NCsim Ver5.8**
**Company: Cadence**

For the C++ compilation, freeware g++ compiler from GCC is used. This is a 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, SystemCÂ®, 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

### 9.2.2 Waveform Viewer

**Tool: Simvision**

**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.

### 9.2.3 Code Coverage

**Tool: Incisive**

**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 verified 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, transistion and sequence), toggle, variables and gate coverage. Such different metrics of Coverage acts as a feedback loop to improve the input stimulus.

### 9.2.4 SDE-GCC

The VE and the code are written in different languages. The test code to be executed by the EA is written in EA assembly language. EAASM is a proprietary of the company which assembles the code. The major portion of VE is in RISC assembly language. Assembler specific for it is used for its assembling. Some portion of VE is in C. This code is cross compiled into RISC assembly code using sde-gcc cross compiler specific to RISC.

The sde-gcc is a cross compiler for the RISC core. The code for RISC processor can be written in C or C++ language. This being a higher level language, makes the logic development easy and fast. The logic is cross compiled into RISC assembly code using sde-gcc. It contains all the libraries as source code in pre-compiled form Different versions of sde are available and should be selected depending on the requirements. These versions are for different versions of RISC processor IP used. It provides various optimization levels for the code. The optimization includes removing unnecessary NOPs from the code, removing unneeded branch statements from the code. Different options can be used along with it to get the required code.

It also supports source level debugging of an embedded application. The host debugger sde-gdb has the access to your source and object files and understand the structure of your program and data. It also provides a standard procedural interface by which host software can be connected to the EJTAG probe or software simulator via a dynamically loaded library conforming to the microprocessor debugging interface specification.

### 9.2.5 Chip Specific assembler

The test case contains test codes written in assembly language of EA and VR. Hence, assembler specific to this is used for getting the hex codes corresponding to them. VR being a IP core, the assembler for it available with its packet. The assembler and simulator for EA instruction set is developed by the vendor. The simulation helps to simulate the EA module test code and take a reference from them. This helps to understand the basic flow of how the test code will be executed when it will be actually ran by the RTL and the chip.

### 9.2.6 ModelSim 6.2e SE

ModelSim 6.2e SE provides a comprehensive simulation and debug environment for complex ASIC and FPGA designs. Support is provided for multiple languages including Verilog, SystemVerilog, VHDL and SystemC. It is an UNIX, Linux, and Windows-based simulation and debug environment, combining high performance with the most powerful and intuitive GUI in the industry.

**Features**

Unified Coverage Database (UCDB) which is a central point for managing, merging, viewing, analyzing and reporting all coverage information.
Source Annotation. The source window can be enabled to display the values of objects during simulation or when reviewing simulation results logged to WLF.
Finite State Machine Coverage for both VHDL and Verilog is now supported.
Code Coverage results can now be reviewed post-simulation using the graphical user environment.
Simulation messages are now logged in the WLF file and new capabilities for managing message viewing are provided in the message viewer.
The GUI debug and analysis environment continues to evolve to provide greater user-customization and better performance.
Message logging and viewing. Simulation messages are now logged in the WLF and new capabilities for managing message viewing are provided. Messages are organized by their severity and type.

### 9.2.7 ScriptSim

In complex simulations, the verilog language does not have the power to meet all the verification requirements. As a result, vendors have developed proprietary languages, with expensive licensing, to extend verilog capabilities.

ScriptSim extends the verilog capabilities by using powerful, open source languages. Perl and Python are easy to learn, and the many perl and python modules make the languages incredibly powerful.

ScriptSim, like perl and python, is open source. No license server problems! No more waiting months for a simple fix because the vendor won't let you see the source code --- with open source you have the option of modifying the code yourself. Don't get locked into a proprietary product and language!

Almost all modern verilog simulators are compatible with ScriptSim. The simulator needs to support PLI version 1.0.

To use ScriptSim, add the verilog user function $scriptsim() to create script (perl or python) processes, and to communicate with existing processes.

## 9.3 Applications Softwares

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.

### 9.3.1 S/W: Bugzilla

Bugzilla is a database for bugs. It lets people report bugs and assigns these bugs to the appropriate developers. Developers can use Bugzilla 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 bugzilla for the developer. Each bug can be issued a priority based on urgency for its resolve. Each bug can have inter dependencies. Each bug is assigned a unique id. Any authorized person can login into the buzilla 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:

**Status Whiteboard :**
The Status Whiteboard is used for writing short notes about the bug.
**Keywords:**
This field is used to store various keywords. For instance, when bugs have test cases associated, name of test cases affected are used.
**Target Milestone:**
This field targets to the bug development process. It shows the status of bug. The bug is assigned to the concerned developer.

**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. Bugzilla 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 logs all 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.

**Bug Life Cycle:**
What happens to a bug when it is first reported depends on who reported it. When a bug becomes NEW, the developer will probably look at the bug and either accept it or give it to someone else. If the bug remains new and inactive for more than a week, Bugzilla nags the bug's owner with email until action is taken. Whenever a bug is reassigned or has its component changed, its status is set back to NEW. The NEW status means that the bug is newly added. When a bug is closed it's marked RESOLVED and given one of the following resolutions.

### 9.3.2 CVS (Concurrent Versioning System)<br>Source Code management software

The Concurrent Versions System (CVS), also known as the Concurrent Versioning System, implements a version control system: it 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.

CVS 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 CVS has the task of keeping track of the version history of a project with only local developers. The server software normally runs on Unix (although at least CVSNT server supports various flavors of Windows and Unix), while CVS clients may run on any major operating-system platform.

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 server 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. This task is mostly handled automatically by the CVS client, requiring manual intervention only when a conflict arises between a checked-in modification and the yet-unchecked local version of a file.
If the check-in operation succeeds, then the version numbers of all files involved automatically increment, and the CVS server writes a user-supplied description line, the date and the author's name to its log files. CVS can also run external, user-specified log processing scripts following each commit. These scripts are installed by an entry in CVS's log info file, which can trigger email notification, or convert the log data into a web-based format.

**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.
**Checkin**: When a developer checks in their code, they merge their changes with the remote repository. Similar to commit.
**Commit:** When all changes to a local working copy of a repository are confirmed and uploaded. Normally accompanied by a log message.
**Update**: CVS command that synchronizes your working copy with the copy on the repository.

Few useful Commands related to CVS are discussed below:
**add**
Insert a new file into the repository. Will not be visible in the repository before a commit 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.
**commit**
Commit your changes into the the repository.
**diff**
Check difference with your private files against the repository.
**import**
Hands over revision control to CVS of an existing project.

**log**
Display revision log information.
**release**
Mark your private files as removed. This means that the repository is updated (if necessary) and the file is mark as NOT checked out.
**status**
Display information.
**update**
Updates your local files, i.e. if the repository files have been changed this command brings your local files up to date.

## 9.4 Languages

Languages provide the Platform for Verification. These languages are for coding designs, writing scripts to command processes, etc. The languages used are explained here in detailed.

### 9.4.1 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. Following is the list of vital features of C/C++, which makes it versatile.

Object Oriented Approach: Data Encapsulation
Polymorphism
Inheritance
Virtual Functions
File Managements

### 9.4.2 Verilog

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.
5. PLI (Programming Language Interface) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

### 9.4.3 PERL

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 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.

### 9.4.4 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.

### 9.4.5 TCL

TCL (Tool Command Language) scripts are called tclsh.. Tcl scripts are made up of commands separated by newlines or semicolons. Each Tcl command consists of one or more words separated by spaces. All Tcl commands return results. If a command has no meaningful result then it returns an empty string as its result. Tcl allows you to store values in variables and use the values later in commands. You can also use the result of one command in an argument to another command. This is called command substitution. Tcl provides a complete set of control structures including commands for conditional execution, looping, and procedures. Tcl control structures are just commands that take Tcl scripts as arguments.

Tcl commands are created in three ways. One group of commands is provided by the Tcl interpreter itself. These commands are called builtin commands. They include all of the commands you have seen so far and many more (see below). The builtin commands are present in all Tcl applications. The second group of commands is created using the Tcl extension mechanism. Tcl provides APIs that allow you to create a new command by writing a command procedure in C or C++ that implements the command. You then register the command procedure with the Tcl interpreter by telling Tcl the name of the command that the procedure implements. In the future, whenever that particular name is used for a Tcl command, Tcl will call your command procedure to execute the command. The builtin commands are also implemented using this same extension mechanism; their command procedures are simply part of the Tcl library.

When Tcl is used inside an application, the application incorporates its key features into Tcl using the extension mechanism. Thus the set of available Tcl commands varies from application to application. There are also numerous extension packages that can be incorporated into any Tcl application. One of the best known extensions is Tk, which provides powerful facilities for building graphical user interfaces. Other extensions provide object-oriented programming, database access, more graphical capabilities, and a variety of other features. One of Tcl's greatest advantages for building integration applications is the ease with which it can be extended to incorporate new features or communicate with other resources.

The third group of commands consists of procedures created with the proc command, such as the power command created above. Typically, extensions are used for lower-level functions where C programming is convenient, and procedures are used for higher-level functions where it is easier to write in Tcl.
Tcl contains many other commands besides the ones used in the preceding examples

### 9.5 The Verilog Dump File

In Verilog, Dump files can be created after simulation. The standard Verilog VCD/EVCD files contains information for signals dumped into it in form of their change in the value at any instant. This file is used by the tester machine to apply the same sequence of signals to the chip and under and compare it against the test results generated and specified in the VCD/EVCD file. VCD stands for Value Change Dump and EVCD stands for Extended VCD.

In Verilog, the way that you create the VCD dumpfile is by using two types of Verilog system calls (1) $dumpfile and (2) $dumpvars. The VCD/EVCD file generated is a ASCII file and is compatible with different waveform viewers.

```
initial
begin
$dumpfile("test.vcd"                                    );
$dumpvars(1,test.top);
end
```

The following syntax creates a dump file named test.vcd with all signals of module top dumped into it. he depth of level is given as 1. Hence, only signals of modules at depth 1 will be dumped.

**$dumpfile**

The $dumpfile system call takes in one parameter that is a string of the name of the dumpfile to create, in this case the dumpfile we want to create is called "test.top". The purpose of this function to create the file (open it for writing) and outputs some initialization information to the file.

**$dumpvars**

The $dumpvars system call takes in two parameters. The first is the number of levels of hierarchy that you want to dump. To dump a module and all submodules beneath it, set the dump level value to 0 (this means the level specified and all levels below it). The second parameter is a Verilog hierarchical reference to the top-level module instance that you want to dump.

The $dumpfile system call may only be called once within a Verilog design. Typically, it is called in the top-most level of the design (or testbench as it is commonly referred to as); however, the language allows you to call it from anywhere in your design as long as it precedes any calls to $dumpvars.

The $dumpvars system call may be called as many times as necessary to dump the Verilog that you need. For example, if you want to get coverage results for several modules scattered around the design, you may make several $dumpvars calls to dump exactly those modules that you want to see coverage for.

$dumpports is the function used to turn on and dump signals in EVCD format.

Since the EVCD and VCD formats are similar, only the key EVCD format differences will be discussed in this section:

- Signal Strength Levels
- Value Change Data Syntax
- Port Direction and Value Mapping

**Signal Strength Levels**

Verilog HDL allows scalar net signal values to have a full range of unknown values, and different levels of strength or combinations of levels of strength. For logic operation, multiple-level logic strength modeling resolves combinations of signals into known or unknown values, allowing the behavior of hardware to be represented with improved accuracy.

EVCD signal strength can be defined by eight values, ranging from `0` (weakest) to `7` (strongest). The most commonly used values are `0` and `6`.

For example, for logic value 0

<0_strength_component> =6 , <1_strength_component> =0

and for logic value 1

<0_strength_component> =0 , <1_strength_component> =6

**Note:** Logic strength levels are not defined in VCD files because only four states are supported.

**Value Change Data Syntax**

The EVCD `data` command is different from the one used with VCD because the EVCD version can provide strength information and additional signal states.

**Data**

p[port_value] [0_strength_component] [1_strength_component] [identifier_code]

**Description**

The value change section shows the actual value changes at each simulation time increment. Only variables that change value during a time increment are listed. In the EVCD file, strength information and a larger number of value states with port direction are presented in the value change section. The arguments for the EVCD data command are listed below.

**Arguments**

| | |
|---|---|
| p | Key character that indicates a port.<br>**Note:** There is no space between p and port_value. |
| port_value | State character which contains information about driving direction and the value of the port. |

| 0_strength_component | One of the eight Verilog strength values indicating the `strength0` component of the value. |
|---|---|
| 1_strength_component | One of the eight Verilog strength values indicating the `strength1` component of the value. |
| identifier_code | The identifier code for the port, which is defined in the $var construct for the port. |

Table 9.1 EVCD signal dump format

**Examples**

The following example

pU 0 7 <0

tells the Virtuoso UltraSim simulator the one bit bus with identifier <0 (defined by $var) has a binary value of U, and the strength of 0 component is 0 and the strength of 1 component is 7.

In the next example

pCCC 667 667 !

tells the simulator the bus with identifier ! (defined by $var) has a binary value of CCC, and the strength of 0 component is 667 and the strength of 1 component is 667. There is more than one driver on this port and the resolved value is CCC.

# CONCLUSION

**Module Level Functional Verification aims to exercise module to its core functionality before RTL tap out. The Module Level Bringup Verification aims to exercise first tap out chip's module to its core functionality along with its internal and external data paths. This tests the fabrication process.**

## FUTURE SCOPE

1. Reusing the same Generic Environment with updation for the next version of the chip.

2. Executing the chip application on the board with all modules working together.

## REFERENCES


1. Chip Hardware Reference Manual
2. Chip Specific Operational Board Document
3. Writing Testbenches – Janick Bergeron
4. Essentials Of Electronic Testing For Digital, Memory and Mixed Signal VLSI Circuits – Michael L. Bushnell, Vishwani D. Agrawal

## ABBREVIATIONS & ACRONYMS

| | |
|---|---|
| **ASIC** | **Application Specific Integrated Circuit** |
| **ASSP** | **Application Specific Standard Product** |
| **ATPG** | **Automatic Test Pattern Generation** |
| **ATSC** | **Advanced Television Systems Committee** |
| **AVC** | **Advanced Video Coding** |
| **AAC** | **Advanced Audio Coding** |
| **AMBA-AHB** | **Advanced Microcontroller Bus Architecture-Advanced High- Performance Bus** |
| **ABV** | **Assertion based Verification** |
| | |
| **BDD** | **Binary Decision Diagram** |
| **BFM** | **Bus Functional Module** |
| | |
| **CODEC** | **Coder/Decoder** |
| **CABAC** | **Context Adaptive Binary Arithmetic Coding** |
| **CVS** | **Concurrent Versioning System** |
| **CRC** | **Cyclic Redundancy Check** |
| | |
| **DDR** | **Dual Dynamic Random Access memory** |
| **DVD** | **Digital Video Disc** |
| **DVB** | **Digital Video Broadcasting** |
| **DCT** | **Discrete Cosine Transform** |
| **DCI** | **Dynamic Control Interface** |
| | |
| **EJTAG** | **Enhanced Joint Test Action Group** |
| **EPROM** | **Electrically Programmable Read Only Memory** |
| **EVCD** | **Extended Value Change Dump** |
| **EDA** | **Electronic Design Automation** |
| | |
| **FPGA** | **Field Programmable Gate Array** |
| **FSM** | **Finite State Machine** |
| | |
| **GPIO** | **General Purpose Input Output** |
| | |
| **HDL** | **Hardware Description Language** |
| **HVL** | **Hard Verification Language** |
| **HDVL** | **Hardware Description & Verification Language** |
| **HDTV** | **High Definition TeleVision** |

| | |
|---|---|
| **ITU** | **International Telecommunication Union** |
| **ISO/IEC** | **International Organization for Standardization/International Electrotechnical Commission** |
| **I2C** | **Inter Integrated Circuit** |
| **IP** | **Intellectual Property** |
| **MPEG** | **Moving Pictures Experts Group** |
| **MIMD** | **Multiple Instruction Multiple Data** |
| **MIME** | **Mate Ingress Mate Egress** |
| **NTSC** | **National Television Standards Committee** |
| **OS** | **Operating System** |
| **OSCI** | **Open SystemC Initiative** |
| **PAL** | **Phase Alternating Line** |
| **PIM** | **Processor in Memory** |
| **PERL** | **Practical Extraction and Report Language** |
| **PLD** | **Programmable Logic Devices** |
| **PLI** | **Programming Language Interface** |
| **SOC** | **System On Chip** |
| **SIMD** | **Single Instruction Multiple Data** |
| **SVCD** | **Super Video Compact Disc** |
| **SMPTE** | **Society Of Motion Picture & Television Engineers** |
| **SSI** | **Synchronous Serial Interface** |
| **SPI** | **Synchronous Parallel Interface** |
| **SPDIF** | **Sony/Philips Digital Interface Format** |
| **SVN** | **Sub Versioning System** |
| **SCM** | **Single Chip Multiprocessor** |
| **SONET** | **Synchronous Optical NETwork** |
| **SDH** | **Synchronous Digital Hierarchy** |
| **SOC** | **System On Chip** |
| **SKS** | **Single Kernel Simulator** |
| **STS** | **Synchronous Transport Signal** |
| **STM** | **Synchronous Transport Module** |
| **SRS** | **System Requirement Study** |
| **TU** | **Tributary Unit** |
| **TWB** | **Telecom WorkBench** |
| **TCL** | **Tool Command Language** |

| | |
|---|---|
| **VE** | **Verification Environment** |
| **VCD** | **Value Changed Dump** |
| **VPN** | **Virtual Private Network** |
| **VT** | **Virtual Tributary** |
| **VC** | **Virtual Container** |
| | |
| **WMV** | **Windows Media Video** |