Analysis And Usage Of Formal Equivalence Check On SoC

By

Vaibhav Chavan

Roll No: 10MECV02

Department of Electronics and Communication Engineering

Institute of Technology

Nirma University

Ahmedabad-382481

May 2012

Analysis And Usage Of Formal Equivalence Check On SoC

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

In

Electronics And Communication Engineering

(VLSI Design)

By

Vaibhav Chavan

(10MECV02)

Under the Guidance of

Mr. MadanMohan Gowda

Intel Technology India Pvt. Ltd.

Department of Electronics and Communication Engineering

Institute of Technology

Nirma University

Ahmedabad-382481

May 2012

Declaration

This is to certify that

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

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

Vaibhav Chavan

Certificate

This is to certify that the Major Project entitled **" Analysis And Usage Of Formal Equivalence Check On SoC"** submitted by **Vaibhav Chavan (10MECV02)**, towards the partial fulfillment of the requirements for the degree of **Master of Technology in VLSI Design** of **Nirma University of Science and Technology, Ahmedabad** is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date: ____________________	**Place: Ahmedabad**
External Guide	**HOD**
____________________	____________________
Mr. Madan Mohan K. M.	**Prof. A. S. Ranade**
Intel Technology India Pvt. Ltd.	**Professor, EC, NU**
Director	**Internal Guide**
____________________	____________________
Dr.K.R.Kotecha	**Mrs. Usha Mehta**
IT, NU	**Professor (VLSI Design)**
	Nirma University

Acknowledgements

First and foremost, Sincere thanks to **Mr. Saish Amonkar**, Manager, Intel India Technology PVT LTD. I enjoyed his vast knowledge and owe him lots of gratitude for having a profound impact on this report. I would like to thank my Mentor, **Mr. Kadabagala Mallappa Gowda MadanMohan**, Verification lead, Intel India Technology PVT LTD For valuable guidance. Throughout the training, he gave me valuable advice on project work. Without him, this project work would never have been completed. I would like to thank buddy, **Ms. Khyati Bhatt**, Component Design Engineer, Intel India Technology PVT LTD, for giving valuable help in FEV ramp-up and solving my all FEV related query. I would like to thank buddy, **S. K. Phaniraj**, Verification Engineer, Intel India Technology PVT LTD, for educating me about Open Verification Methodology and solving my OVM related queries which helped me a lot while working on actual project. I would also like to thank **Dr. K. R. Kotecha**, Director, Institute of Technology, Nirma University, for providing me with an opportunity to get an internship at Intel India Technology PVT LTD, Bangalore. I would also Thank to my project coordinator, **Dr. N.M. Devashrayee**, Professor, VLSI Design, Institute of Technology, Nirma University, for giving valuable support for project work. I would like to thank my all faculty members for providing encouragement, exchanging knowledge during my post-graduate program. I also owe my colleagues in the Intel, special thanks for helping me on this path and for making project at Intel more enjoyable.

- Vaibhav Chavan
10MECV02

Abstract

Ever-growing complexity in System on Chip (SOC) is forcing logic design to move above the register transfer level (RTL). For example, functional specifications are being written in software. These specifications are written for clarity, and are not optimized or intended for synthesis. The Logic Synthesis as a process is prone to Bugs. There are too many transformations happening in logic synthesis which can alter the netlist in a wrong manner and make it infer functionality other than what was intended in the original RTL. These bugs are not intentional but happen by accident during synthesis tool development. So Functional Equivalence Verification (FEV) between the software specification and the implementation is needed.

This report introduces the Functional Equivalence Verification approach for SOC. It describes the method for functional equivalence verification. It also introduces the algorithm by which the FEV tool runs. This report imposes upon the importance of the Formal Equivalence Verification in VLSI design flow.

Contents

List of Figures

Chapter 1

Introduction to SoC

A system on a chip or system on chip (SoC or SOC) is an Integrated circuit(IC) that integrates all components of a computer or other electronic system into a single chip. It may contain digital, analog or mixed - signal and often radio-frequency functions-all on a single chip substrate. The VLSI manufacturing technology advances has made possible to put millions of transistors on a single die. This enables designers to put systems on a single chip thus moving everything from board to single chip resulting to growth of system on chip (SOC) technology.SOC design includes efforts to integrate heterogeneous or different types of silicon IPs (intellectual properties) on to the same chip, like memory, microprocessor, random logics, and analog circuitry. SOC often incorporates analog components, and can also include opto/microelectronic mechanical system components in the future. An SOC design provide significant advantages in terms of speed , area ,reliability ,security and power however suffers from high system complexity , fabrication costs and increase verification requirements. However increasing reusability of IP and highly sophisticated tools (hardware/software) are making the SOC designs an extremely favorable and exciting option for modern applications.

Chapter 2

Introduction to Formal Verification

Formal Verification (FV) is a general set of methods that mathematically prove aspects of a design, in contrast to dynamic validation / simulation (which just checks specific values). Thus Formal Verification is a way of creating evidence suggesting that a system either does or does not have some property by comparing a formal model of the system to a formal specification of the property Using logical inference to prove that the model does or doesn't meet the specifications.

Note that formal verification does not guarantee that a real system has an intended property. It only provides suggestive evidence about the real system - the proof is about a formal model. The distinction is important because the evidence may be mistaken for several reasons: the formal model may not accurately describe the real system; the formal spec may not accurately describe the intended property; the logical inference system may not be sound; and the purported proof may not actually follow the rules of inference in the logic. All of these have happened in practice. Thus it is important to also test the real system.

Formal verification is different from dynamic validation (testing by simulation) in that formal verification is complete within some formally defined domain. Dynamic validation may use formal inference, but isn't complete within a well-defined domain.A hybrid verification method combines formal verification and simulation-based methods.

2.1 Contents

- Uses
- Formal Equivalence Verification (FEV)
- Formal Property Verification (FPV) -RTL Level
- Formal Protocol Modeling and Verification
- Methods
- Formal Verification tools and languages

2.2 Uses

Formal verification is used, among other things, to show that different schematics are logically equivalent or not(this is called formal equivalence verification, or FEV). It also shows that the RTL models implement micro architectural or architectural specifications (this is sometimes called formal property verification, or FPV), that microcode implements architecture specifications, that software implements abstract algorithms, and that protocols and other abstract algorithms satisfy even more abstract properties.

2.3 Formal Equivalence Verification

Formal Equivalence Verification (FEV) is the use of formal verification to show that two models are equivalent (under some set of assumptions). Its most common uses are showing equivalence between schematics and RTL models, between different versions of a schematic, and between different versions of an RTL model. The current tools for equivalence verification are Seqver and Clever. Similar tools are available from a variety of external vendors.

2.4 Formal Property Verification

Formal Property Verification (FPV) is the use of formal verification to show that an RTL model satisfies some property or properties, such as implementing a micro architecture specifications or maintaining an invariant. Formal property verification is similar to simulation-based dynamic validation in that both are used to find bugs in RTL models, but FPV is different in that it uses formal verification methods. FPV is distinguished from formal equivalence verification (FEV) in that it verifies more general properties than equivalence. In recent years FPV has become more accessible to non-specialists, through the Assertion Based Verification methodologies. These enable RTL authors to add assertions to their code, and use powerful formal methods to prove them.

2.5 Formal Protocol Modeling and Verification

Another important aspect of formal verification is modeling and verification of protocols above the RTL level. This is done with different languages and tools.

Chapter 3

Formal Equivalence Verification

Formal Equivalence Verification is a formal verification method used for proving that two models are equivalent. This can be used to prove, for example, that a circuit implementation works identically to the RTL it is implementing. It is a method of proving the equivalence of two different view of the same logic design. It uses mathematical technique to verify equivalence of a referenced design and a modified design. These tools can be used to verify the equivalence of RTL-RTL, RTL-gate, and gate-gate implementations. Since equivalence checking tools compare the target design with the reference design, it is critical that the reference design is functionally correct.

3.1 Why Formal Equivalence is needed?

Logic Synthesis as a process is prone to Bugs. There are too many transformations happening in logic synthesis which can alter the netlist in a wrong manner and make it infer functionality other than what was intended in the original RTL. These bugs are not intentional but happen by accident during synthesis tool development. They can happen mostly during the elaboration stage as new verilog/VHDL constructs start showing up from time to time in new language standards. It takes time to build a robust Language analysis/elaboration engine as a front end to synthesis engine. Simulation is not a robust proof of correctness as it was never intended to be exhaustive. However we can run the same functional vectors and check our netlist to see if we are lucky to catch a problem in gates. This is never a full proof verification strategy. Equivalence checking comes to the rescue. And it is fool proof. EC mathematically proves that the RTL and Netlist are equivalent and does not need functional vectors. Advantages of Formal Equivalence Verification:

- Verifies 100
- Faster than performing an exhaustive simulation.

Limitation:

- It does not verify the timing of the design.

3.2 Formal Equivalence Verification at the center

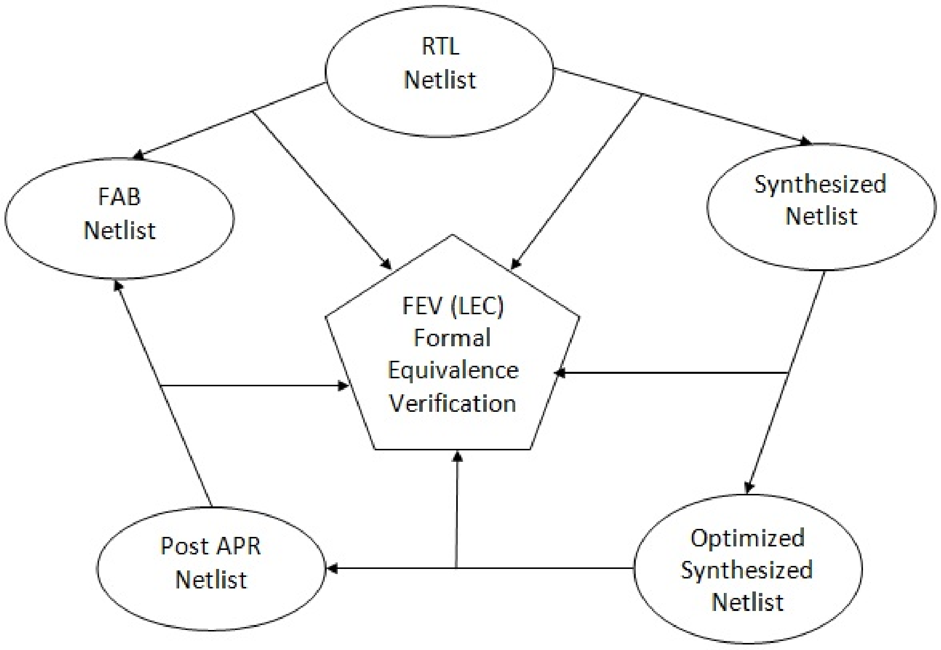


Figure 3.1: Importance of FEV at each stage

FEV is required to compare RTL netlist versus synthesized netlist, Synthesized netlist versus optimized synthesized netlist, optimized synthesized netlist versus Post Auto place and Routed netlist, Post Auto place and Routed netlist versus Fabrication (final) netlist Fabrication (final) netlist versus RTL netlist. Thus it is very important step of VLSI flow.

3.3 Equivalence Checking FLow

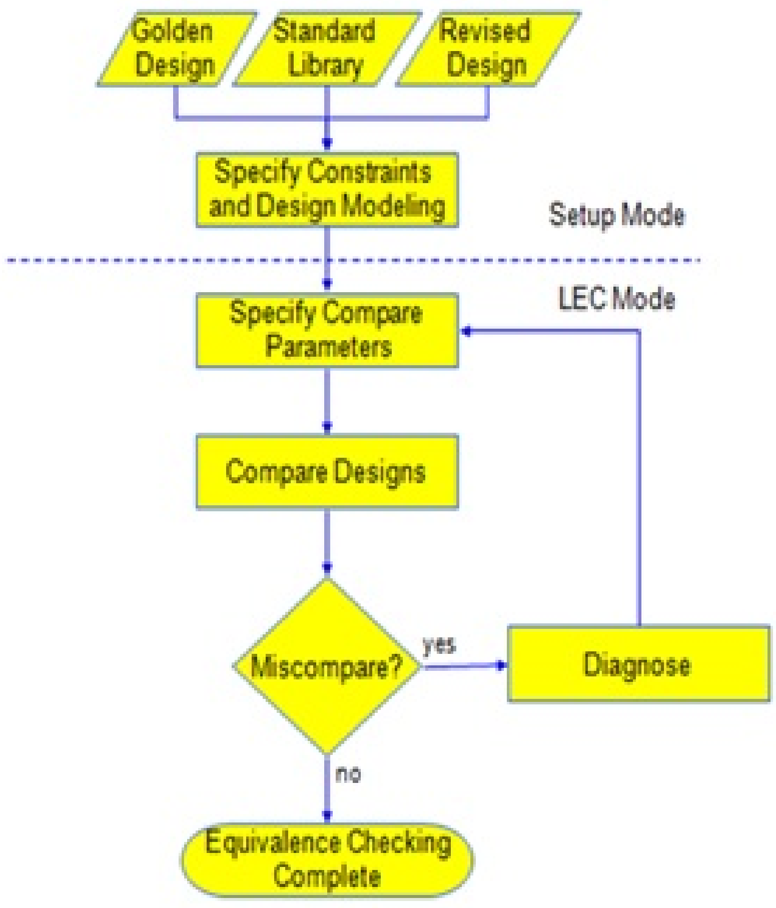


Figure 3.2: The Basic Logic Equivalence Checking Flow

3.4 Main Elements of Equivalence checking tool

The figure shows the simplification of the general equivalence checking problem to logic equivalence. In addition to the inputs, now the state vectors are held to the exact same value at all times. With a second XOR condition, the next-state vectors B1+ and B2+ are checked for equality at all times as well. The checker tool proves combinational Boolean equivalence if E1 and E2 are 0 at all times. Even after this drastic simplification of the original problem, the tool allows the exhaustive comparison of the two state machines with the same set of states and the same initial state.

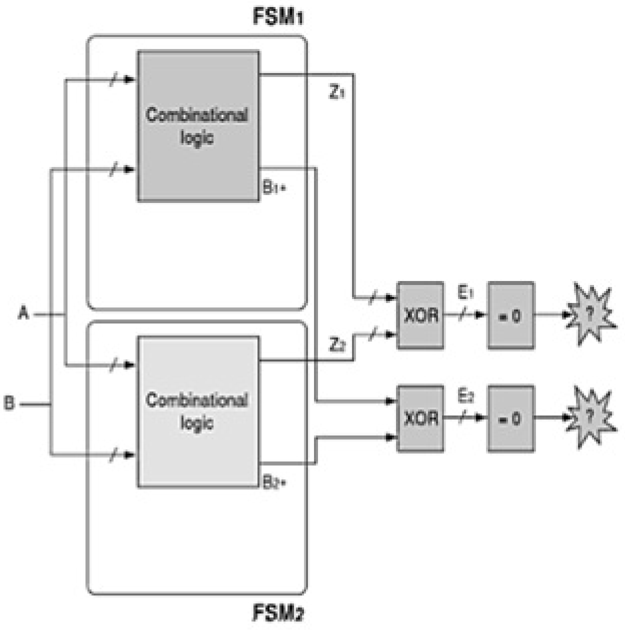


Figure 3.3: Simplification of the general equivalence checking problem to logic equivalence

The application in terms of the methodology flow described above, means that the state vectors and the state encoding of the DUV stay constant after the definition of RTL. This restriction is reasonable for most practical applications because many tool driven design transformation and optimizations change the combinational part of the DUV only.

3.5 DUV partitioning

The DUV partitioning attempts to divide the DUV into smaller pieces for which the EC algorithms prove equivalence individually. The partition can be done vertically, horizontally or combination of both. The equivalence checkers take outputs from either model and compare only the logic that specifically drives these outputs. In this example, the comparison occurs between the pairs of logic partitions A1, A2, B1, B2 and C1, C2. The equivalence checker successively picks a pair of inputs, one per DUV model for check. It iterates over all DUV outputs. For each iteration step, the tool eliminates all the logic that does not directly drive the given outputs

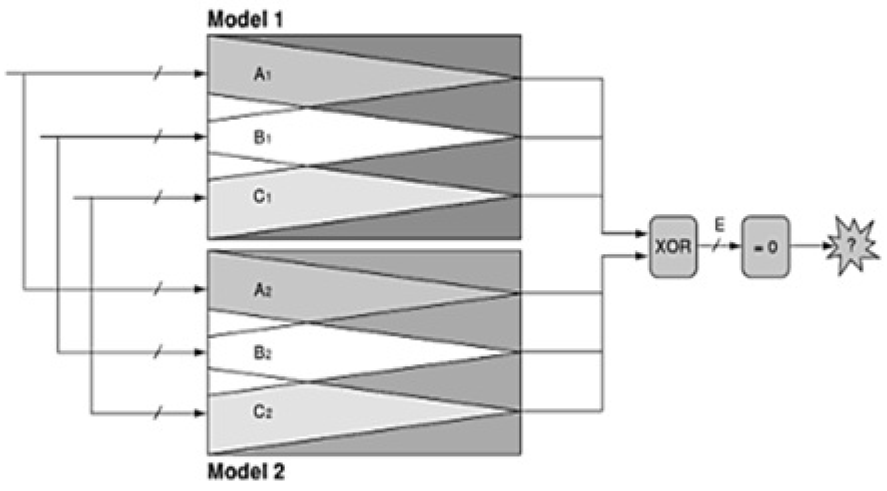


Figure 3.4: The Horizontal Partitioning Approach to equivalence Checking

Horizontal partitioning works only if the equivalence checker indeed prove that the cut points are Boolean equivalent Vertical partitioning selects intermediate points inside the logic of either DUV model. This cuts the depth of the logic cones compared at each step as well as number of inputs and amount of logic covered by cone. The figure shows the illustration of this principle: C1, C2, C3 are the cut points for which the checker must prove equivalence for the partitioning to succeed. Either the user specifies the cut points manually or the EC tool has heuristics to find these automatically.

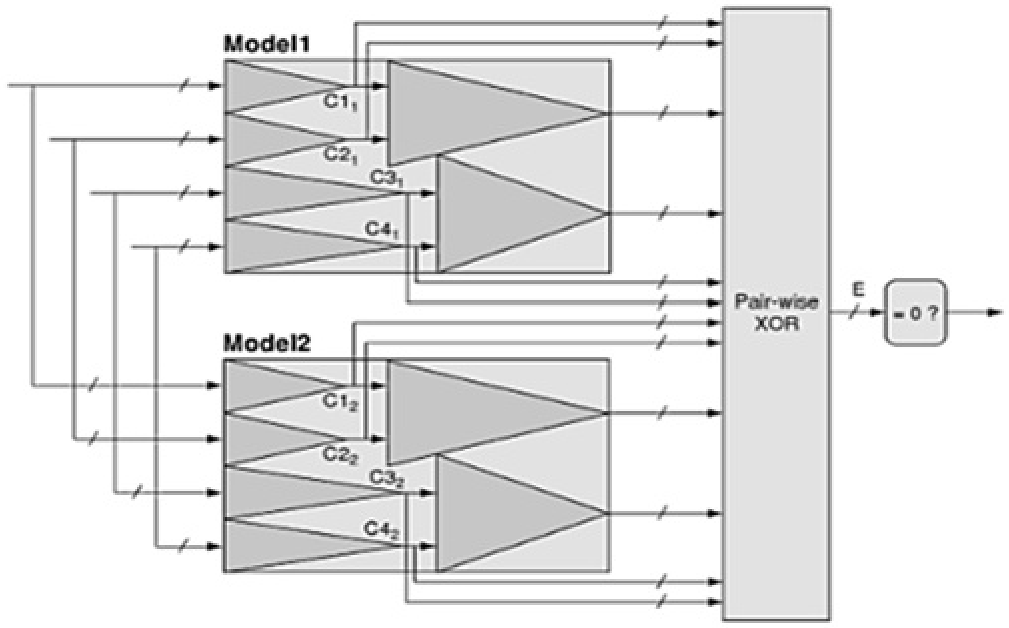


Figure 3.5: The Vertical Partitioning Approach to Equivalence Checking

Vertical partitioning using cut points C1, C2 and C3 allows the comparison process to prove equivalence for smaller subcones with less number of inputs and logic. The XOR check now takes not only the model output to compare pair-wise but also the signals that represent the cut point in each model.

3.6 Mapping

Pairing correspond golden and revised key points. Key points are the points to map. So while doing FEV, the key points of golden design should be mapped with the revised design.

Golden	← →	**Revised**
PI	← →	PI
PO	← →	PO
DFF	← →	DFF
DLAT	← →	DLAT
BBOX	← →	BBOX
CUT	← →	CUT
Z	← →	Z
E	← →	E

Figure 3.6: Mapping points

3.7 State matching FEV

FEV works on state matching algorithm. It does the partition of the whole circuit into many cones. And then compares the key points. Lets us take an example.

Question: Are the following two figures equivalent?

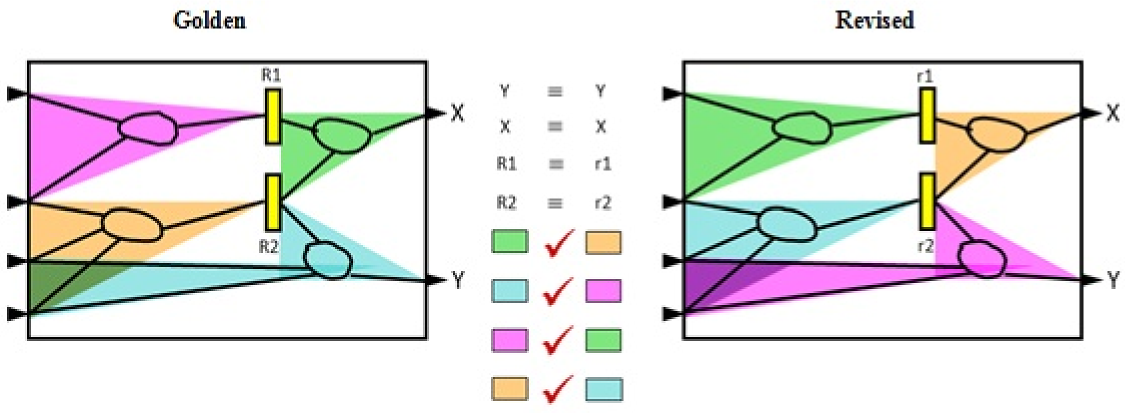


Figure 3.7: Partitioning and Mapping for equivalence checking

Answer: yes, but again the question is how the tool will understand that both the figures are equivalent. So the tool does this comparison by using state matching algorithm. First of all it maps the key point. Once the key points are mapped then the comparison can be done. The design is converted into groups of cones. And the cones inputs and outputs are compared by mathematical equation.

The steps are given below.

Step 1: Map the key Points.

Check whether inputs of both the circuits are mapped? Check whether outputs of both the circuits are mapped? Check whether states of both the circuits are mapped? i.e. f1–f3, f2–f4

Step 2: build equations. What if there is an error? Now compare the equations.

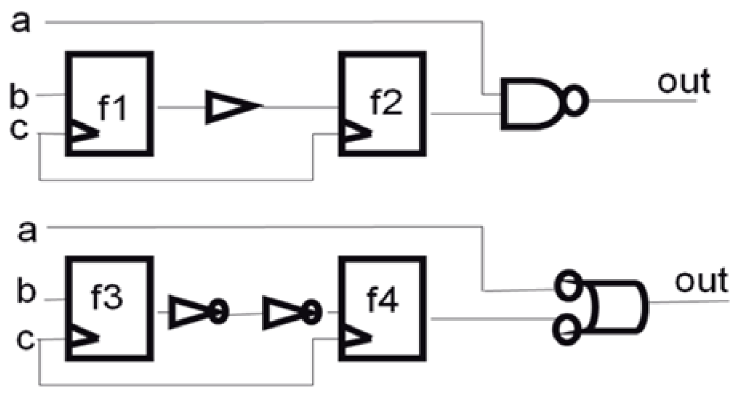


Figure 3.8: Example of Equivalent Design

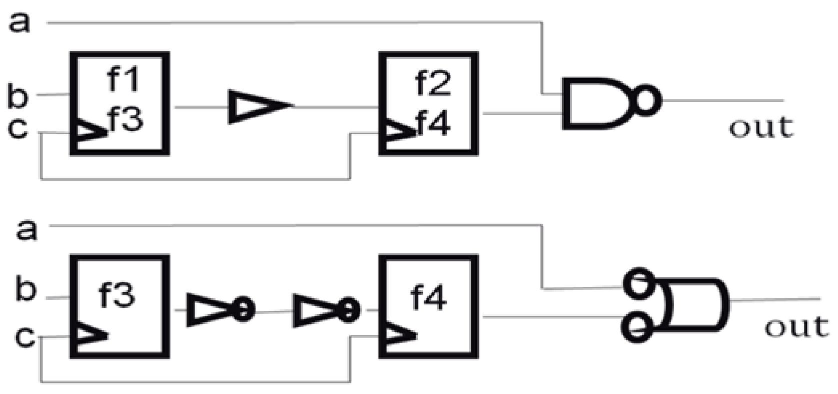


Figure 3.9: Mapping Key Points

The Example of equivalent design is shown in figure 3.8. It can be seen that the two designs have identical output equations. Tool does this comparison by the steps described above which is shown in the figures. figure 3.9 shows the mapping of key points in both of the designs. then the equations are build and compared, which is shown in figure 3.10.

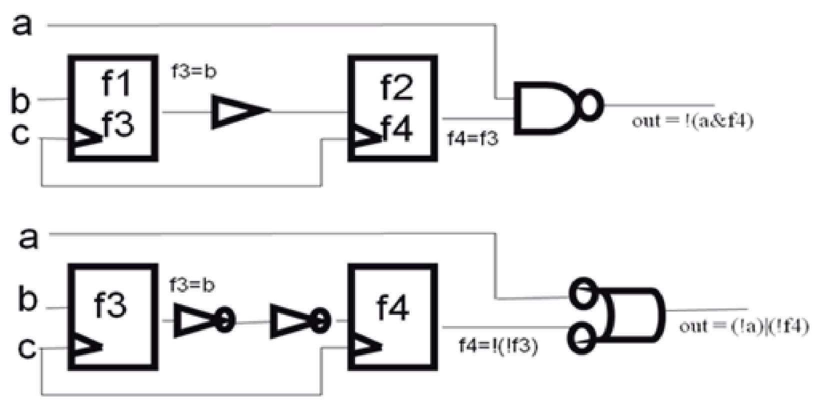


Figure 3.10: Building Equations for the design

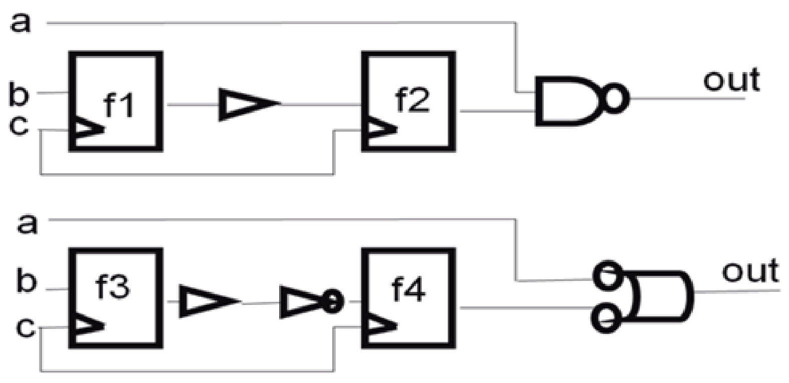


Figure 3.11: Example of Non-Equivalent Design

The Example of Non-equivalent design is shown in figure 3.11. It can be seen that the output equations of these two designs does not matches, so the mathematical equation or the logic in both the design is not same. and this is how the tool does the comparison by mathematical equations.

Chapter 4

Cadence Conformal LEC

What is cadence conformal LEC?

Formerly known as Verplex LEC, this is a vendor FEV tool in use by many chipset and miscellaneous ASIC teams, as well as for occasional specialized tasks by processor teams. It is much more user-friendly and its engine is optimized for combinational cones with large amounts of logic designs.

4.1 Conformal usage model

- Based on command console

  - Capable of taking general tcl scripts

- ”help” available for any command

  - Example: ”help read design”

  - Full manuals in /pkgs/cadence6/CONFRML71/doc

- ”set gui on” / ”set gui off” can be done any time

- ”dofile filename.do” to execute script

4.2 Mapping Key Points

- LEC has good automapper

  - Can guess many mappings

  - But sometimes fails to map properly

- View mapping as ”renaming”

  - Temporarily rename RTL sig to match netlist

  - ”add renaming rule” to specify mappings

4.3 LEC Do file

- set log file lec.log -replace
- read library - file path
- read design -systemverilog/verilog -golden -f myrtl.filelist
- read design -systemverilog/verilog -revised -f mynetlist.filelist
- add renaming rule r1 foo bar -golden
- add blackbox -module name -cell name - golden/revised/both
- add pin direction - in/out/IO - pin name - module name
- abstract logic -module name - all -golden/revised
- add notranslate filepaths filepath names -library/design -both/golden/revised
- add pin constraints 0/1 primary pin -module module name
- set sys mode lec
- report unmapped points add compare points -all compare
- report compare data

4.4 Steps for FEV run

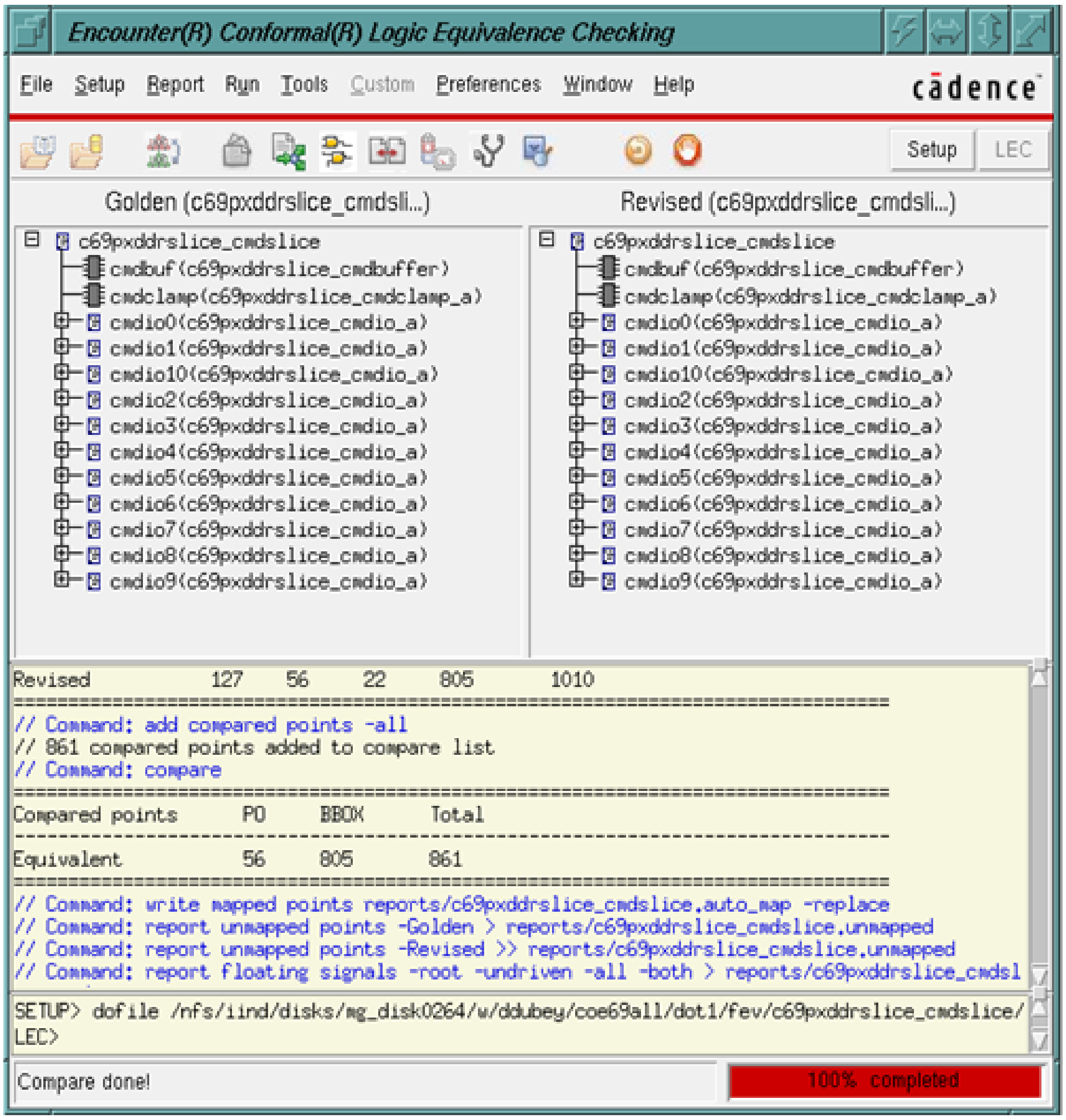
- Create work area for the design on which FEV is to be performed.
- Copy the necessary files i.e. golden and revised netlist and libraries into their corresponding path in the work area.
- Run command for FEV. This command will give the result for key points and compared points
- Debug the key points and diagnose the not mapped key points.
- Make local copy of the input files and make changes in that file in order to solve not mapped points
- Report the key point
- Debug the compared points. Diagnose the non-equivalent points
- Make changes in local copy in order to solve the non-equivalent points i. Report the compared points, if they are the major issues
- Prepare Lec.do file which will automatically run the script to make FEV clean

4.5 Inputs for FEV run

- RTL/BMOD
- Schematic netlist (With correct tagging info)
- Standard Cell Library Files

4.6 Conformal GUI

Conformal Graphical User Interface can be useful for analysing the FEV key points and compared points. It helps for debugging the errors. The Conformal GUI is shown here.


Encounter(R) Conformal(R) Logic Equivalence Checking
File Setup Report Run Tools Custom Preferences Window Help
cadence
Setup LEC
Golden (c69pxddrslice_cmdsli...)
Revised (c69pxddrslice_cmdsli...)
Compare done!
100% completed

```
c69pxddrslice_cmdslice
  cmdbuf(c69pxddrslice_cmdbuffer)
  cmdclamp(c69pxddrslice_cmdclamp_a)
  cmdio0(c69pxddrslice_cmdio_a)
  cmdio1(c69pxddrslice_cmdio_a)
  cmdio10(c69pxddrslice_cmdio_a)
  cmdio2(c69pxddrslice_cmdio_a)
  cmdio3(c69pxddrslice_cmdio_a)
  cmdio4(c69pxddrslice_cmdio_a)
  cmdio5(c69pxddrslice_cmdio_a)
  cmdio6(c69pxddrslice_cmdio_a)
  cmdio7(c69pxddrslice_cmdio_a)
  cmdio8(c69pxddrslice_cmdio_a)
  cmdio9(c69pxddrslice_cmdio_a)
```

```
c69pxddrslice_cmdslice
  cmdbuf(c69pxddrslice_cmdbuffer)
  cmdclamp(c69pxddrslice_cmdclamp_a)
  cmdio0(c69pxddrslice_cmdio_a)
  cmdio1(c69pxddrslice_cmdio_a)
  cmdio10(c69pxddrslice_cmdio_a)
  cmdio2(c69pxddrslice_cmdio_a)
  cmdio3(c69pxddrslice_cmdio_a)
  cmdio4(c69pxddrslice_cmdio_a)
  cmdio5(c69pxddrslice_cmdio_a)
  cmdio6(c69pxddrslice_cmdio_a)
  cmdio7(c69pxddrslice_cmdio_a)
  cmdio8(c69pxddrslice_cmdio_a)
  cmdio9(c69pxddrslice_cmdio_a)
```

```
Revised             127    56     22     805      1010
================================================================================
// Command: add compared points -all
// 861 compared points added to compare list
// Command: compare
================================================================================
Compared points       PO      BBOX       Total
--------------------------------------------------------------------------------
Equivalent            56      805        861
================================================================================
// Command: write mapped points reports/c69pxddrslice_cmdslice.auto_map -replace
// Command: report unmapped points -Golden > reports/c69pxddrslice_cmdslice.unmapped
// Command: report unmapped points -Revised >> reports/c69pxddrslice_cmdslice.unmapped
// Command: report floating signals -root -undriven -all -both > reports/c69pxddrslice_cmdsl
SETUP> dofile /nfs/iind/disks/mg_disk0264/w/ddubey/coe69all/dot1/fev/c69pxddrslice_cmdslice/
LEC>
```

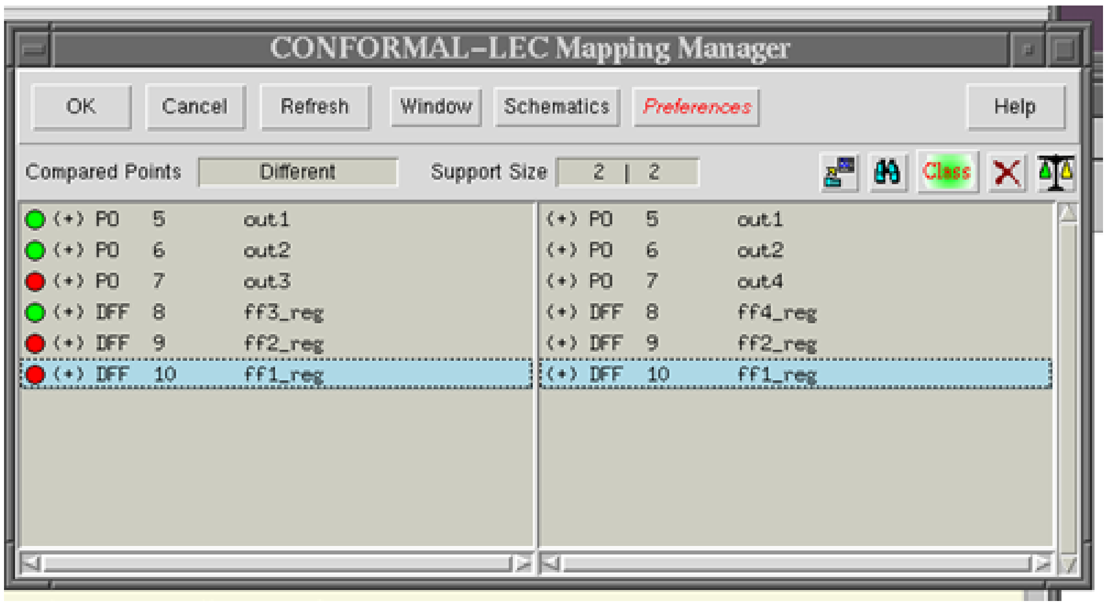
Figure 4.1: Conformal GUI showing compared points

4.7 Debugging mapping points and compare points

- Extra : this indicates that key point exists only in the golden design or only in the revised design, but does not affect the circuit functionality.
- Unreachable: this indicates that key point that is not propagated to any observable point
- Not mapped: this indicates that key point that has no correspondence on either side, may be resolved with the renaming rules.

sectionDebugging Mismatches

The GUI shows the different compare points by different colors. The Compared points can be categorized as shown below



Compared Points: Different   Support Size: 2 | 2

🟢	(+)	PO	5	out1	(+)	PO	5	out1
🟢	(+)	PO	6	out2	(+)	PO	6	out2
🔴	(+)	PO	7	out3	(+)	PO	7	out4
🟢	(+)	DFF	8	ff3_reg	(+)	DFF	8	ff4_reg
🔴	(+)	DFF	9	ff2_reg	(+)	DFF	9	ff2_reg
🔴	(+)	DFF	10	ff1_reg	(+)	DFF	10	ff1_reg

Figure 4.2: Identifying and debugging mismatches by conformal -lec mapping manager

- Equivalent: refers to the key points proven to be equivalent. (green filled circle)
- Inverted Equivalent: refers to the key points proven to be complementary equivalent to the key point. (divide green filled circle)

- Non Equivalent: refers to the key points proven to be different. (red filled circle)
- Abort: refers to the key points not yet proven equivalent or non equivalent due to time-out or other system parameter (yellow filled circle)
- Not-compared: refers to the key points not yet compared.

4.8 Analyzing non-equivalent compare points

There are different ways to analyze and debug the non-equivalent points

- Diagnosis manager: it will tell the corresponding point.
- Schematic viewer: viewing the schematic of both Golden and Revised design.
- Source code: by viewing the source code of Golden and Revised design.

4.9 Frequently seen issues

- Top level and black box pin name mismatches.
- Extra components in either schematic or RTL.
- Incorrect logic gates used in feeding compare points.
- Lots of cells not listed in initial black box lists can't be extract and need to be manually black boxed during the run.

4.10 Debug tips

- Black box any known analog components provided by the circuit and RTL owner
- Extract the remaining components using the 'abstract logic' command.
- Any components that fail to extract need manual transistor direction assignment

- Check the schematic and assign all bi-directional transistors the correct direction using the 'add mos direction' command.
- Check the schematic and assign the direction to the some of the pins of the revised module (if applicable) using 'ass pin direction'.
- Check the spelling non-equivalent points; sometimes the points are not equal because of the mismatch in spelling or incorrectly written.
- Visually examine using the conformal schematic viewer. If the logic is still not extracted after assigning the transistor direction the component must be black boxed.
- Once the simulation runs with no errors, check the primary inputs and outputs for the HDL and schematics to ensure they all match.
- Look for any unmapped points. All these points must be debugged or ignored before proceeding.
- Debug all the failing compare points.

Chapter 5

Results of FEV

The Cadence Conformal Tool shows the results in terms of mapped points and compared points. The following is the results obtained while performing FEV on different modules of our project.

Table I: FEV Results

No	Module name	Error	Fix	Fix Working?
1	SPI	pin direction mismatch	Assign pin direction	yes
2	TAP	Pins missing	Pins spelled correctly	yes
3	ADC	Floating pin in golden	Mapped that Pin in RTL	yes
4	DMA	Unmapped DFFs	Remodel -seq merge	yes
5	GPIO	Unmapped DLATS	Remodel-seq merge	yes

Chapter 6

Introduction to Open Verification Methodology

OVM provides the best framework to achieve coverage-driven verification (CDV). CDV combines automatic test generation, self-checking testbenches, and coverage metrics to significantly reduce the time spent verifying a design. The purpose of CDV is to:

- Eliminate the effort and time spent creating hundreds of tests.
- Ensure thorough verification using up-front goal setting.
- Receive early error notifications and deploy run-time checking and error analysis.

6.1 Contents

This Chapter includes the following contents which can be helpful to understand the basic concepts of OVM.

- OVM Test bench and Environments
- OVC Overview

- Syestem Verilog Class Library

6.2 OVM Test bench and Environments

An OVM testbench is composed of reusable verification environments called OVM verification components (OVCs). An OVC is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, a design sub module, or a full system. Each OVC follows a consistent architecture and consists of a complete set of elements for stimulating, checking, and collecting coverage information for a specific protocol or design. The OVC is applied to the device under test (DUT) to verify your implementation of the protocol or design architecture. OVCs expedite creation of efficient testbenches for your DUT and are structured to work with any hardware description language (HDL) and high-level verification language (HVL) including Verilog, VHDL, e, SystemVerilog, and SystemC.

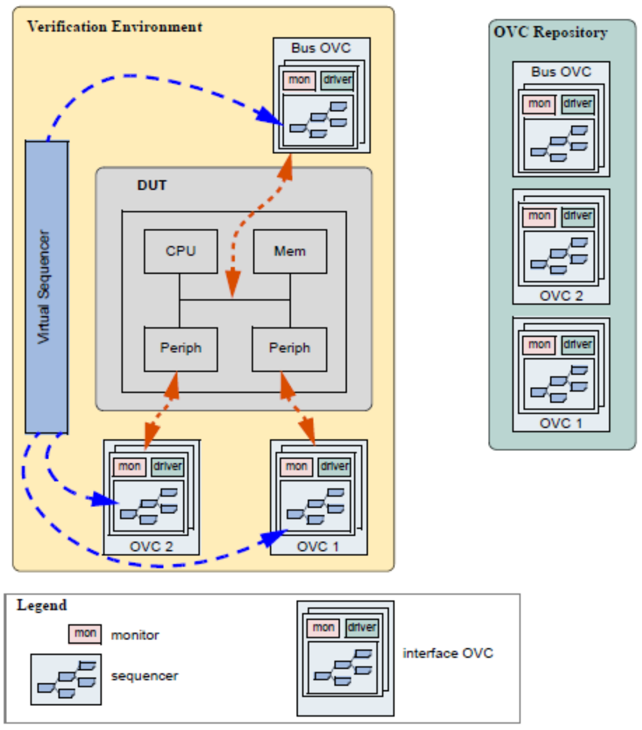


Figure 6.1: Verification Environment Example

Figure 6.1 shows an example of a verification environment with three interface OVCs. These OVCs might be stored in a company repository and reused for multiple verification environments. The interface OVC is instantiated and configured for a desired operational mode. The verification environment also contains a multi-channel sequence mechanism (that is, virtual sequencer) which synchronizes the timing and the data between the different interfaces and allows fine control of the test environment for a particular test.

6.3 OVC Overview

The following subsections describe the components of an OVC.

- Data Item (Transaction)
- Driver
- Sequencer
- Monitor
- Agent
- Environment

6.3.1 Data Item (Transaction)

Data items represent the input to the DUT. Examples include networking packets, bus transactions, and instructions. The fields and attributes of a data item are derived from the data item's specification. For example, the Ethernet protocol specification defines valid values and attributes for an Ethernet data packet. In a typical test, many data items are generated and sent to the DUT. By intelligently randomizing data item fields using SystemVerilog constraints, you can create a large number of meaningful tests and maximize coverage.

6.3.2 Driver (BFM)

A driver is an active entity that emulates logic that drives the DUT. A typical driver repeatedly receives a data item and drives it to the DUT by sampling and driving the DUT signals. For example, a driver controls the read/write signal, address bus, and data bus for a number of clocks cycles to perform a write transfer.

6.3.3 Sequencer

A sequencer is an advanced stimulus generator that controls the items that are provided to the driver for execution. By default, a sequencer behaves similarly to a simple stimulus generator and returns a random data item upon request from the driver. This default behavior allows you to add constraints to the data item class in order to control the distribution of randomized values. Unlike generators that randomize arrays of transactions or one transaction at a time, a sequencer captures important randomization requirements out-of-the box. A partial list of the sequencer's built-in capabilities includes:

- Ability to react to the current state of the DUT for every data item generated.
- Captures the order between data items in user-defined sequences, which forms a more structured and meaningful stimulus pattern.
- Enables time modeling in reusable scenarios.
- Supports declarative and procedural constraints for the same scenario.
- Allows system-level synchronization and control of multiple interfaces.

6.3.4 Monitor

A monitor is a passive entity that samples DUT signals but does not drive them. Monitors collect coverage information and perform checking. Even though reusable drivers and sequencers drive bus traffic, they are not used for coverage and checking. Monitors are used instead. A monitor:

- Collects transactions (data items). A monitor extracts signal information from a bus and translates the information into a transaction that can be made available to other components and to the test writer.

- Extracts events. The monitor detects the availability of information (such as a transaction), structures the data, and emits an event to notify other components of the availability of the transaction. A monitor also captures status information so it is available to other components and to the test writer.

- Performs checking. Checking typically consists of protocol and data checkers to verify that the DUT output meets the protocol specification.

- Perform Coverage. Coverage also is collected in the monitor..

6.3.5 Agent

Sequencers, drivers, and monitors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities. To reduce the amount of work and knowledge required by the test writer, OVM recommends that environment developers create a more abstract container called an agent. Agents can emulate and verify DUT devices. They encapsulate a driver, Sequencer, and monitor. OVCs can contain more than one agent. Some agents (for example, master or transmit agents) initiate transactions to the DUT, while other agents (slave or receive agents) react to transaction requests. Agents should be configurable so that they can be either active or passive. Active agents emulate devices and drive transactions according to test directives. Passive agents only monitor DUT activity.

6.3.6 Environment

The environment (env) is the top-level component of the OVC. It contains one or more agents, as well as other components such as a bus monitor. The env contains configuration properties that enable you to customize the topology and behavior and make it reusable. For example, active agents can be changed into passive agents when the verification environment is reused in system verification. Figure 6.2 illustrates the structure of a reusable verification environment. Notice that an OVC may contain an environment-level monitor. This bus-level monitor performs checking and coverage for activities that are not necessarily related to a single agent. An agent's monitors can leverage data and events collected by the global monitor.

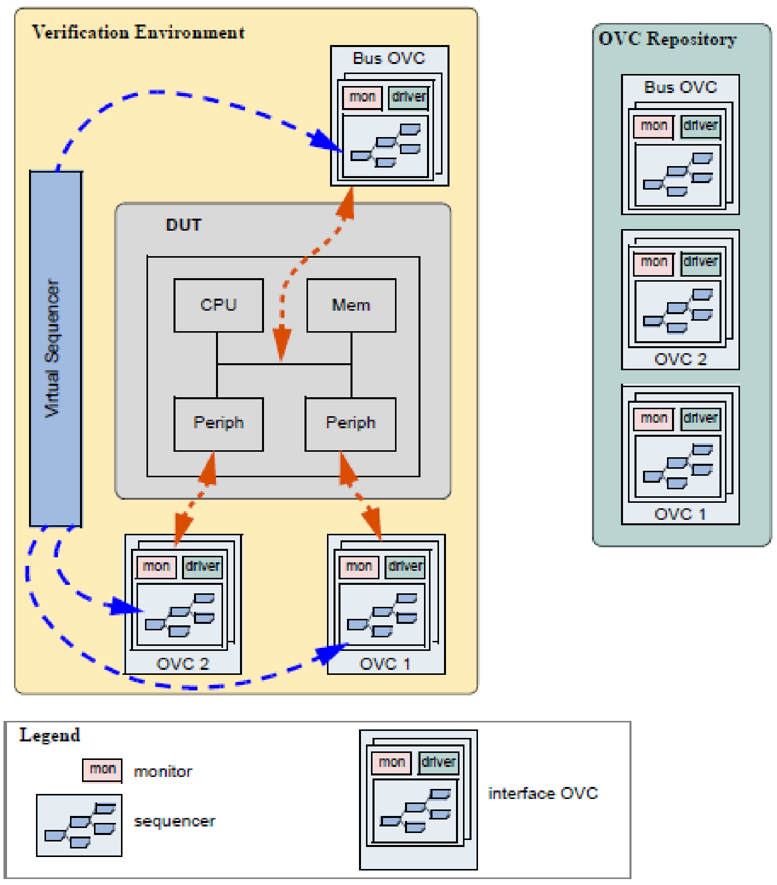


Figure 6.2: Typical OVM Environment

The environment class (ovm_env) is architected to provide a flexible, reusable, and extendable verification component. The main function of the environment class is to model behavior by generating constrained-random traffic, monitoring DUT responses, checking the validity of the protocol activity, and collecting coverage.

6.4 The System Verilog OVM Class Library

The SystemVerilog OVM Class Library provides all the building blocks you need to quickly develop well-constructed, reusable, verification components and test environments. The library consists of base classes, utilities, and macros. Components may be encapsulated and instantiated hierarchically and are controlled through an extendable set of phases to initialize, run, and complete each test. These phases are defined in the base class library but can be extended to meet specific project needs.

The advantages of using the SystemVerilog OVM Class Library include:

- A robust set of built-in features-The SystemVerilog OVM Class Library provides many features that are required for verification, including complete implementation of printing, copying, test phases, factory methods, and more.

- Correctly-implemented OVM concepts-Each component in the block diagram in Figure 6.2 is derived from a corresponding SystemVerilog OVM Class Library component.

6.5 Other OVM Facilities

The SystemVerilog OVM Class Library also provides various utilities to simplify the development and use of verification environments. These utilities support debugging by providing a user-controllable messaging utility. They support development by providing a standard communication infrastructure between verification components (TLM) and flexible verification environment construction (OVM factory).

The SystemVerilog OVM Class Library provides global messaging facilities that can be used for failure reporting and general reporting purposes. Both messages and reporting are important aspects of ease of use. This section includes the following:

6.5.1 OVM Factory

The factory method is a classic software design pattern that is used to create generic code, deferring to run time the exact specification of the object that will be created. In functional verification, introducing class variations is frequently needed. For example, in many tests you might want to derive from the generic data item definition and add more constraints or fields to it; or you might want to use the new derived class in the entire environment or only in a single interface; or perhaps you must modify the way data is sent to the DUT by deriving a new driver. The factory allows you to substitute the verification component without having to provide a derived version of the parent component as well.

The SystemVerilog OVM Class Library provides a built-in central factory that allows:

- Controlling object allocation in the entire environment or for specific objects.
- Modifying stimulus data items as well as infrastructure components (for example, a driver).

Use of the OVM built-in factory reduces the effort of creating an advanced factory or implementing factory methods in class definitions. It facilitates reuse and adjustment of predefined verification IP in the end-user's environment. One of the biggest advantages of the factory is that it is transparent to the test writer and reduces the object-oriented expertise required from both developers and users.

6.5.2 Transaction Level Modelling

OVM components communicate via standard TLM interfaces, which improves reuse. Using a SystemVerilog implementation of TLM in OVM, a component may communicate via its interface to any other component that implements that Interface. Each TLM interface consists of one or more methods used to transport data. TLM specifies the required behavior (semantic) of each method but does not define their implementation. Classes inheriting a TLM interface must provide an implementation that meets the specified semantic. Thus, one component may be connected at the transaction level to others that are implemented at multiple levels of abstraction. The common semantics of TLM communication permit components to be swapped in and out without affecting the rest of the environment.

Chapter 7

Contributions and Conclusion

7.1 Contributions

Addressing the formal equivalence Verification of specification vs. Implementations of system on chip, I have contributed for making FEV clean for some of the modules in our project. I also checked the top level connectivity for the SOC and made FEV clean for Full Chip.

I contributed to my team members by helping them to integrate OVM Compliant JTAG Bus Functional Model in our project.

7.2 Conclusion

Formal Equivalence Verification is very useful and important for VLSI design flow. It is also helpful for verification of ECO (Engineering Change Order) design with exhaustive verification. It is the easiest way to check the top level connectivity by FCFEV (Full Chip FEV).

From the Work done on Open Verification Methodology, I can conclude that, The OVM provides a library of base classes that allow users to create modular, reusable verification environments in which components talk to each other via standard transaction-level modeling (TLM) interfaces. This methodology enables an unprecedented level of flexibility, customization, and reuse.

References

[1] http://verificationacademy.com/verification-methodology

[2] http://www.testbench.in/

[3] http://systemverilog.in/

[4] http://www.cerc.utexas.edu/ jaa/ee382m-verif/lectures/5-2.pdf