FORMAL EQUIVALENCE VERIFICATION FOR VLSI DESIGN

Major Project

Submitted in Partial Fulfillment of the Requirement

For the Degree of

Master of Technology (M.Tech)

In

VLSI Design

By

Navni Modi

12MECV36

Department of Electronics and Communication Engineering

Institute of Technology

Nirma University

Ahmedabad

December-2013

Declaration

This is to certify that

1. I, Navni Modi, student of semester IV, Master of Technology in VLSI Design, Nirma University, Ahmedabad hereby declare that the project work “Formal Equivalence Verification methodologies for VLSI designs" has been independently carried out by me under the guidance of Mr Satish Kumar R. and Mr. Akhil Chandran, Intel Technology India Private Limited, Bangalore and Dr. N. M. Devashrayee, Program Coordinator, Department of VLSI Design, Nirma University, Ahmedabad. This Project has been submitted in the partial fulfillment of the requirements for the award of degree Master of Technology (M.Tech.) in VLSI Design, Nirma University, Ahmedabad during the year 2013 - 2014.

2. I have not submitted this work in full or part to any other University or Institution for the award of any other degree.

Navni Modi
12MECV36

Certificate

This is to certify that the Major Project entitled “Formal Equivalence Verification for VLSI Designs” submitted by Navni Modi (12MECV36), 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 her 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 part-I, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Usha Mehta	Mr. Satish Kumar R.,
Internal Project Guide	Mr. Akhil Chandran,
Institute of Technology,	External Project Guide,
Nirma University, Ahmedabad	Intel Technology India Pvt. Ltd.,

Dr. N. M. Devashrayee	Dr. P.N. Tekwani,
Program Cordinator	Head of EE Dept.,
Institute of Technology,	Institute of Technology,
Nirma University, Ahmedabad	Nirma University, Ahmedabad

Date: Place: Ahmedabad

Acknowledgement

First and foremost, sincere thanks to Mr. Satish Kumar R., Manager, Intel Technology India Private Limited, Bangalore for assigning me such project and guide me through.

I would like to thank my mentors, Mr. Akhil Chandran and Mr. Kundan Kumar, Intel Technology India Private Limited, Bangalore for their valuable guidance. Through-out the training, they have given me valuable advice on project work which I am very lucky to benefit from.

I would like to thank all my teammates at Intel Technology India Private Limited Bangalore, for giving valuable help in FEV environment ramp-up and solving my all queries.

I would also like to thank to my Project Coordinator Prof. Usha Mehta, Professor, VLSI Design, Institute of Technology, Nirma University, Ahmedabad for giving me 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.

Navni Modi
12MECV36

Abstract

Verification of any design consumes about 70% of the total turnaround time of design process. Thus different tools and flows are been developed to reduce the time required for verification. All the inputs required for the tool are generated by the flow, a wrapper around the tool. Verification gets easily done if the flow is tool friendly and generates all the required files by tool in proper format. This report details the ways that can be used to reduce the time of verification. As a result of which the total time of design process can be reduced. Verification is done between two different designs either at same abstract level or at different abstract levels. It is observed that the run-time decreased by increasing the auto- mapping in the flow for the constraints picked by tools. All the solutions concluded by this project are applicable for functional unit block (FUB) and sub-section level FEV.

CONTENTS

List of figures

CHAPTER 1

Introduction

1.1 What is Verification of a design?

The logic design of circuits includes steps from specifying the functionality of the circuit to the fabrication of silicon chip. We are seeing a continuous growth in the complexity of the circuit design. The aim is to release a remarkable (find a word for this) product in a specified time-to-market schedule. Thus design productivity, reduction in area, space and power dissipation has become primary importance. The main design process steps include:

1. Specification of the design
2. Behavioral description of circuit design in HDL (hardware descriptive language)
3. Architectural Synthesis: Generates a RTL (register transfer level) model with data path and control units.
4. Logic Synthesis: Takes RTL description to the gate level description using different design styles for example custom design, cell based design, array based design.
5. Physical design: This step converts the gate level design to the physical layout of the chip using geometrical patterns.

These steps are very complex and time consuming. With increasing design complexity the probability of error-prone design also increases. As a result it is preferred to remove all the design errors at earlier stage of design process to avoid any extra procedure to fix these errors. It is important to check that no errors are made during design process. This is called *verification of design.*

1.2 Need for Verification

Simulation based dynamic verification techniques have been used as verification techniques. As the modern day designs are becoming complex, traditional verification techniques are becoming tedious and time taking jobs. Static verification techniques help in reducing the turnaround time required for verification and simplifying overall development cycle for complex design. Today the SoC (System on Chip) designs are more integrated. Around 70% of the overall design time and cost is spent on verification and validation. The only way to reduce the time of verification is to adopt Static Verification techniques.

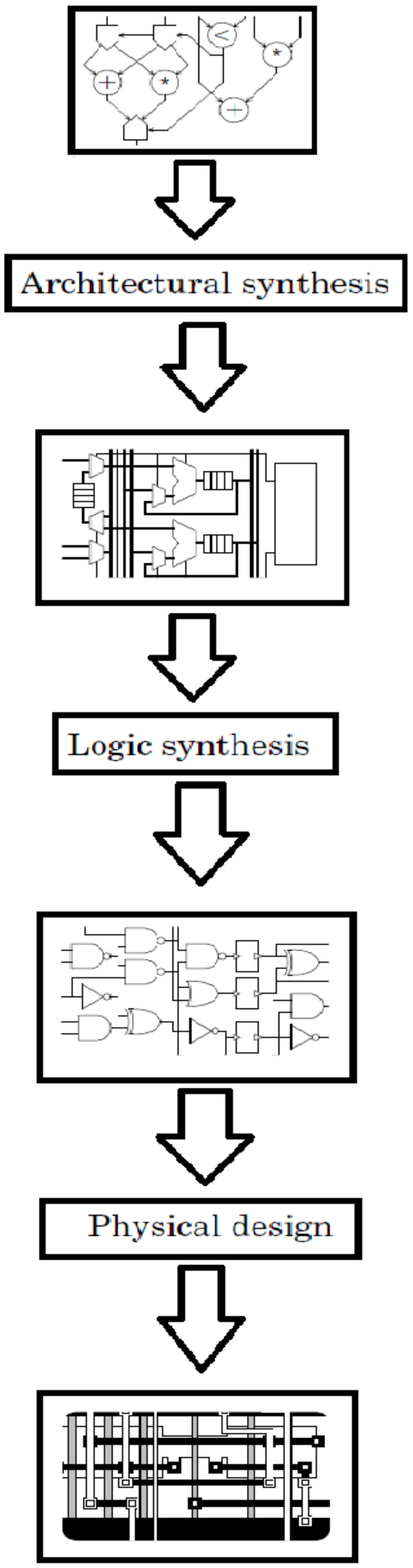

Figure 1 Flow for design process

Synthesis and Formal equivalence verification have helped us move the majority of the design effect to the RTL. One may think that the importance of verification is diminishing with the increasing use of synthesis tools, because the correctness of the underlying techniques has (hopefully) been proved when these techniques were constructed. However, with the widespread use of synthesis tools, it has become clear that this conception is not true. One of the reasons for this is the complexity of the synthesis algorithms. Another reason is, to design for test and design for power, a lot of transformations are still performed on the gate-level netlist. Hence, it is still desirable to verify some specific functionality at the gate-level to ensure the transformations have been performed correctly and the functionality of the design has not been changed. Modern complex designs contain several hundred million transistors. Verification of such a complex system in a shorter span of time becomes a dominating factor before it goes silicon level. Several Formal Verification methods have been proposed and are under research as an alternative to classical simulation techniques, since it can't guarantee sufficient coverage of the design. In addition to this simulation is a slow process. Formal Verification (FV) techniques ensure 100% functional correctness and they are more reliable and cost effective, less time consuming.

1.3 Verification trend in industry

Considering the complexity of the design now-a-days it is extremely necessary to catch and remove bugs in earlier stages of design process. Hence verification plays a very important role. There are two types of verification problems. The first one is related to the questions: Does the initial specification really describe the system we intend to design? This is called *design verification or validation*. The second type concerns the question: Is the result of some design step consistent with previous descriptions of the design? This is called *implementation verification.*

Design verification relates to the correctness of the design specification with respect to the intent of the designer. Design verification often has an experimental character. The designer preforms experiments on the design specification to check that the specified model satisfies certain properties. Implementation verification relates to the correctness of the design steps. It typically involves the comparison of two circuit models to check that there are no inconsistencies. An example is the comparison of the design descriptions before and after logic synthesis to verify the correctness of the manipulations done by synthesis. Static verification differs from the dynamic verification in terms of test vectors. Dynamic verification techniques are generally simulation methods. Simulation has been the most common way for functional verification. However, this approach became complicated as the design increases. Simulation has two major limitations. Firstly, it depends on vector coverage, so it is not complete, and thus, may fail to catch some design errors. Secondly, it is very time-consuming when performed at lower level of abstraction, e.g., the gate-level or the transistor-level.

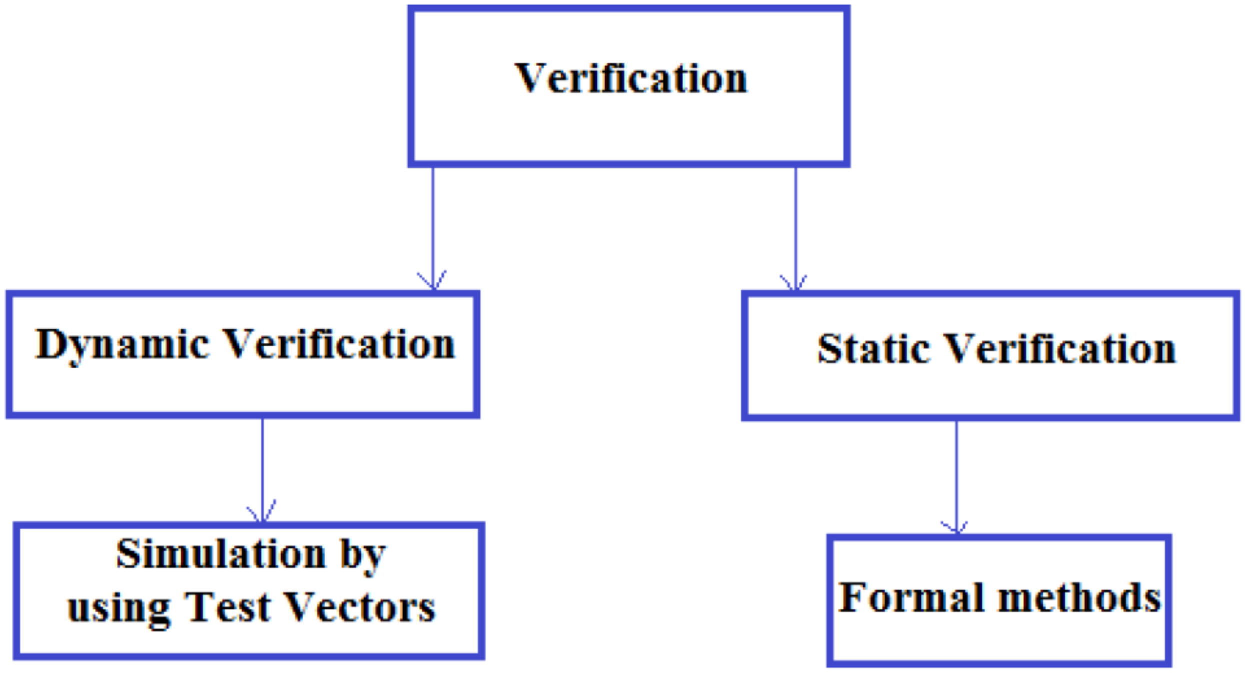

Figure 2 Types of Verification

On the other hand, formal verification has emerged as another solution to prove the correctness of a design. Formal Verification techniques ensure 100% functional correctness and they are more reliable and cost effective, less time consuming.

1.4 Project Overview

Formal verification methods are often divided into the three categories of Theorem proving, Model checking and language containment, and equivalence checking.

Theorem proving is an approach to verification where the verification problem is described as a theorem in a formal theory. A formal theory consists of a language in which the formulas are written, a set of axioms, and a set of inference rules. The inference rules are syntactic transformation rules for the formulas. With these rules and the axioms, theorems can be proved.

Model checking and ***language containment*** are methods to check properties of a design, where the properties are specified respectively as temporal logic formulas. For finite state models, these methods can be fully automated. In practical applications however, the size of the model often constitutes a severe limitation. The selection of a suitable abstraction is typically not automated, because many abstractions only weakly preserve the properties of the designs, i.e., if a property is not valid in the abstract model, it can still hold in the full model.

Equivalence checking is a method to check that two descriptions of a design specify the same behavior, or, if the descriptions are at different abstraction levels, to check that there are no inconsistencies between the specified behaviors. Therefore it is mostly used for implementation verification. Equivalence checking tools typically provide a high degree of automation.

Each verification method has its own strengths and weakness, which are mostly a consequence of the chosen balance between the expressiveness of the underlying logic and the degree of automation. In this project I have worked on the Equivalence Verification methods used in Intel industry and I have tried to automate the flow used here. Also the project is intended to modify the previous codes written for verification. Many different tools for verification are used as per the design requirement. Wrapper around these tools is used to provide environment setup and collaterals of the designs to the tool in required format. Modifications in coding of this wrapper are also the part of this project.

CHAPTER 2

Formal Equivalence Verification

2.1 Definition

Formal verification aims for formally establishing that an implementation satisfies a specification. The term implementation or revised model (IMP or REV) refers to the design description that is to be verified, while the term specification or golden (SPEC or GOL) refers to the design description or the property with respect to which correctness is to be determined. Formal verification is very precise, well-defined, and assures a small probability of bugs slipping unnoticed. Verification is used at various stages of the design flow. Verification is used to check the correctness of the modified RTL model with earlier model. It is used for checking functional equivalence between the RTL and synthesized netlist model. It is used to check the equivalence between the final netlist at the back-end after optimization, placement & routing and synthesized model.

The goal of the formal equivalence verification is to provide a justifiable proof for of the correctness of an implementation of a digital circuit with respect to its specification, based on sound mathematical methods such as Boolean algebra and systems of logic reasoning. Synthesis and logic equivalence checking (LEC) have helped us move the majority of the design effect to the RTL. However, to design for test and design for power, a lot of transformations are still performed on the gate-level netlist. Hence, it is still desirable to verify some specific functionality at the gate-level to ensure the transformations have been performed correctly and the functionality of the design has not been changed.

Both the SPEC and IMP model are reduced to Boolean equations or mathematical theorem or models and these are compared to each other. These two models may be at different level of abstraction i.e. either it can be RTL or gate level model i.e., netlist. SPEC can can be HDL (Verilog or VHDL or system Verilog) model or Netlist model. Similarly IMP model can also be in HDL code or Netlist. Only prerequisite here is SPEC and IMP should have same functionality.

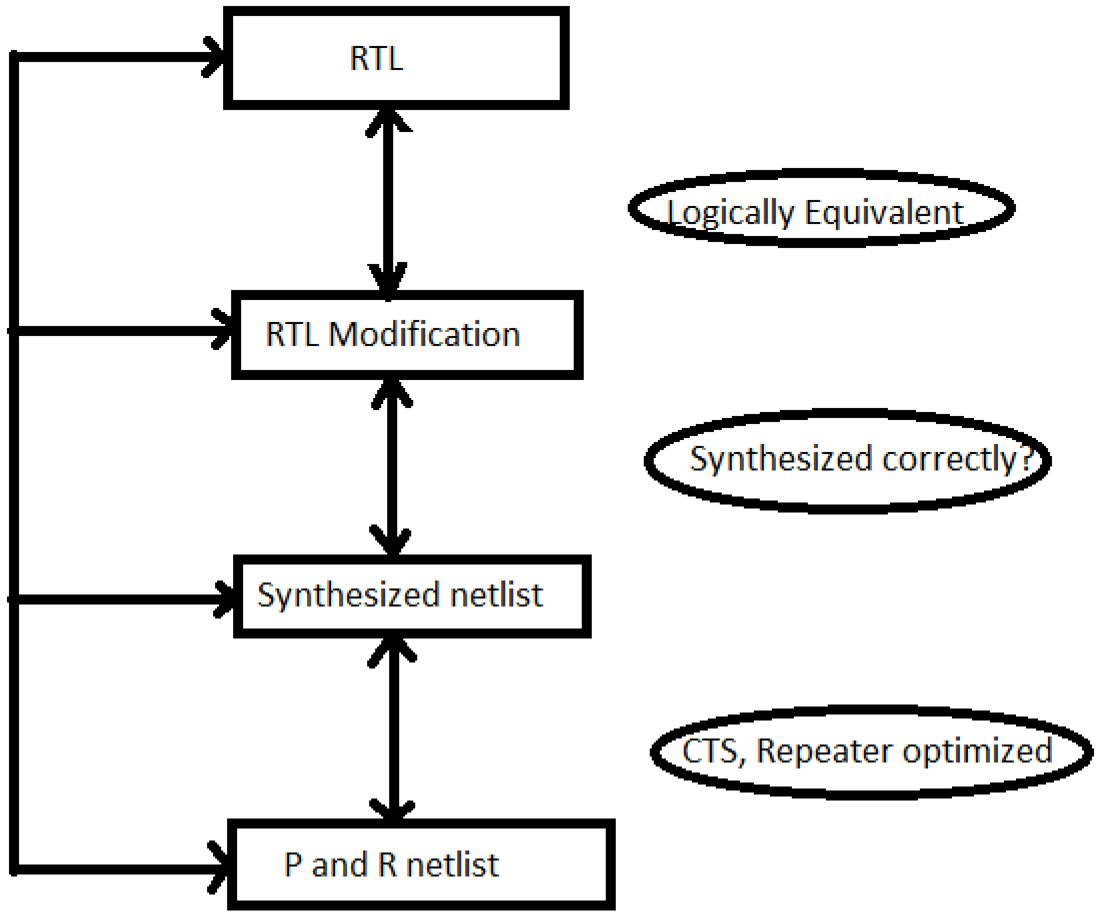

Figure 3 General FEV concept

2.2 Types of FEV

Formal equivalence verification can be divided into 4 categories based on the abstraction levels of SPEC and IMP models. They are:

1. RTL to RTL
2. RTL to netlist
3. Netlist to netlist
4. Engineering Change Order (ECO)

2.2.1 RTL to RTL FEV

In this type of verification the golden design and the revised design will be the RTL models i.e., HDL description. Both the models may be of same design or of different designs. In some cases we may have different RTL models for the synthesis and simulation. Usually RTL to RTL FEV is done to verify the equivalence between RTL model designed for Synthesis and RTL model designed for Simulation (Syn2Sim FEV). RTL to RTL FEV is normally done at the initial stages of the project. Generally RTL to RTL verification is used to verify the following things between the two different RTL models of same design.

1. Gated clock addition
2. Critical Path Optimization
3. Other RTL to Gate constraints

2.2.2 RTL to Netlist FEV

In this the golden design and the revised will be at different level of abstractions. They can be of same design or of different designs. During synthesis the tools optimize the model with the resource sharing, power aware techniques etc.

The tools in the process may implement the design different from the intended one. So, RTL to netlist verification is intended to verify if the RTL model and netlist are equal in functionality. Here the golden model will be the RTL model usually referred as Golden RTL and revised model is synthesized netlist, usually referred as revised netlist.

2.2.3 Netlist to Netlist FEV

The revised design and the golden design will be at the same levels of abstraction. They may be of different designs or of the same design.

This is commonly performed after floor planning, placement & routing. During the process of Floor planning, placement & routing, design changes are added to the synthesized Gate Level Netlist. By performing FEV it is made confirm that no functional changes are occurred due to placement and routing.

2.2.4 ECO

Adding the changes to the design after synthesis is termed as Engineering Change Order (ECO). Making the changes in the RTL model after done with the synthesis, will be a time taking task, as we have to go through all the stages of the flow again. Instead of making the changes in the RTL model, synthesized model (netlist) is modified and it is tested for equivalence with the standard design available.

2.3 Hierarchical FEV

Large VLSI designs are divided into smaller functional units. One example is designing of a CPU. The designing starts from the smallest unit called Functional Unit Block (FUB) which is the discreet part of the chip that achieves functionality. FUBs allow parallel sections of a chip to be designed at the same time. Such FUBS are combined to form a Section. These sections are combined to form Sub Full Chip (SFC), these SFCs in turn combined to form Full chip (FC) or CPU design.

It is highly difficult to verify the design at Full chip level. So FEV start at the FUB level. So The Formal equivalence verification (FEV) in the CPU design at different levels can be categorized as the following:

1. Unit level FEV
2. Section level FEV
3. Sub Full Chip level FEV
4. Full Chip FEV

2.3.1 Unit level (FUB) FEV

Formal Verification can be applied at block (FUB) level to eliminate functional bugs early in the design cycle and match block specifications. At the FUB level the FEV run to check the functionality of the model. This helps in establishing the functional equivalence between the SPEC and IMP FUB.

2.3.2 Section level and Sub full chip level FEV

While performing the section level FEV, blocks or fubs will be black-boxed. The FEV at the section level is performed for connectivity checks. If there are any connectivity problems between the Sections, can be detected in the section level FEV. Also Sections contains some combinational logics with connectivity. The purpose of the section FEV is to ensure that there is no logical difference between the two models after adding this extra logic at the section level.

Sub Full chip level FEV is done to ensure proper connectivity between the Sub Full chips. Even at the Sub Full chip level an additional logic is introduced to meet the timing and power goals. We need to ensure that it didn't introduce any change in the functionality. While performing the Sub Full chip level FEV, all the sections are black-boxed.

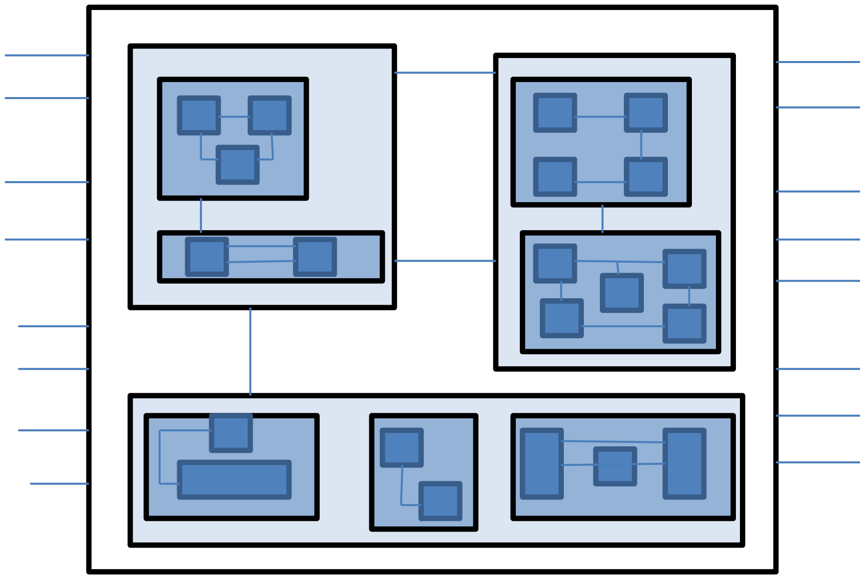
Figure 4 Diagram of full chip

2.3.3 Full Chip Verification

Full chip FEV is preformed to check the connectivity between Sub Full chips. During this the sub full chips are black boxed. During this the connectivity check is performed to ensure that, there are no connectivity issues between the blocks of the design.

The reason to verify this interconnects is to make sure, the additional Virtual repeaters or Flop repeaters present on the path doesn't alter the functionality or logic. These additional Virtual repeaters are buffers or inverters are used to increase signal strength. And Flop repeaters are flip-flops, used for synchronizing the signal between two or more functional blocks.

CHAPTER 3

Verification tool: CONFORMAL

3.1 Overview

Conformal is a Cadence tool is a logic equivalence checking tool. It can verify RTL level, gate level or transistor level designs. Conformal provides equivalence verification for complex SOC designs. It can verify complex logic circuits, data paths, memories and custom logics.

It supports standard library formats like liberty library of Synopsys (.lib files), Verilog libraries etc. This tool is environment independent. No synthesis environment is required to source for this tool.

It is having very good capability of debugging. Interactive windows, user friendly, can be used to debug issues and look at the design schematics. This GUI feature helps user to quickly solve out the design points mis-matches.

3.2 Features

Conformal inculcate many features that helps in authenticating the designs before sending out them for fabrication.

1. Supports Full chip Verification:

   As the conformal is capable of handling complex designs, it reduces the time consumption during the full chip verification.

2. Supports multiple design formats

   It supports Verilog, VHDL, SPICE design formats.

3. Supports standard library formats

   It supports Verilog simulation library format and the Synopsys liberty format.

4. It has a build-in engine to abstract functionality at RTL, gate or transistor level designs.

5. Included Automatic diagnosis

   During every logical mismatch it is very essential to find the exact location of the functional difference. Conformal automatically diagnoses the functional difference, narrows it to a small number of possible location in design.

6. Integrated debugger

   This feature gives the user flexibility to resolve the failures very easily by looking at the schematics and the points of failures.

3.3 Methodology

The below flow graph will explain the conformal flow process.

The steps can be divided into following parts:

1. Reading the designs

   This is the first stage of equivalence checking. Here the conformal reads all the designs (golden design, revised design and the standard libraries).
   When the reading of designs is completed, we can add the specific constraints required for the understanding of the designs.

   Till this stage the conformal is in Setup mode. All the tasks are completed and conformal enters the LEC mode now.

2. Mapping

   In this stage the Conformal maps all the key points automatically. It uses some pre-established algorithms inside it and also uses the constraints file provided by the user.

3. Comparing

   At this point the mapping is done and the conformal is ready to compare all the key points. When comparison is completed reports are dumped out and conformal pinpoints to all the differences found.

4. Diagnosing

   This is the final stage of the formal equivalence checking. From here we can open up the integrated diagnosis tool and resolve the differences. This includes schemetic view and source code manager. After diagnosis a new verification session can be started.

3.4 System modes and mapping

Conformal has two modes: SETUP mode and LEC mode.

In the ***Setup mode***, conformal reads the designs, libraries and mapping files.

Generally the design files are one golden design which is the reference design and another revised design which is post- processed and modified design. Al additional information other than design information is applied in this mode. When all the files are read and constraints are applied the conformal is ready to move into the LEC mode.

Now the conformal is in transition mode from SETUP to LEC mode. Here the rules checking is been done. We can specify how conformal should response to certain types of violations.

In ***transition mode***, conformal also maps the key points. Key points are defined as:

Primary inputs, primary outputs, D- flip flops, D- latches, black boxes etc.

In ***LEC mode,*** mapping is completed on the key points. There are 3 kinds of name-based mapping method that the conformal carries out and after that one no-name based method. No-name mapping method is useful when conformal must map designs with completely different names. By default, conformal maps key point by name-first mapping method. Key points that conformal cannot map are classified as unmapped points.

Unmapped points are classified into 3 categories:

1. Extra unmapped points: These are present in only one side of the design, golden or revised.
2. Unreachable: These points do not have a observable point such as primary output.

3. Not- mapped points: These points are reachable but do not have a corresponding point in the logic fan-in cone of the corresponding design.

3.5 Comparison

After mapping key point next step is to compare key points. This stage determines if the mapped key points are equivalent or non- equivalent. When conformal completes the comparison we can get summary reports showing which points are equivalent and which are non-equivalent.

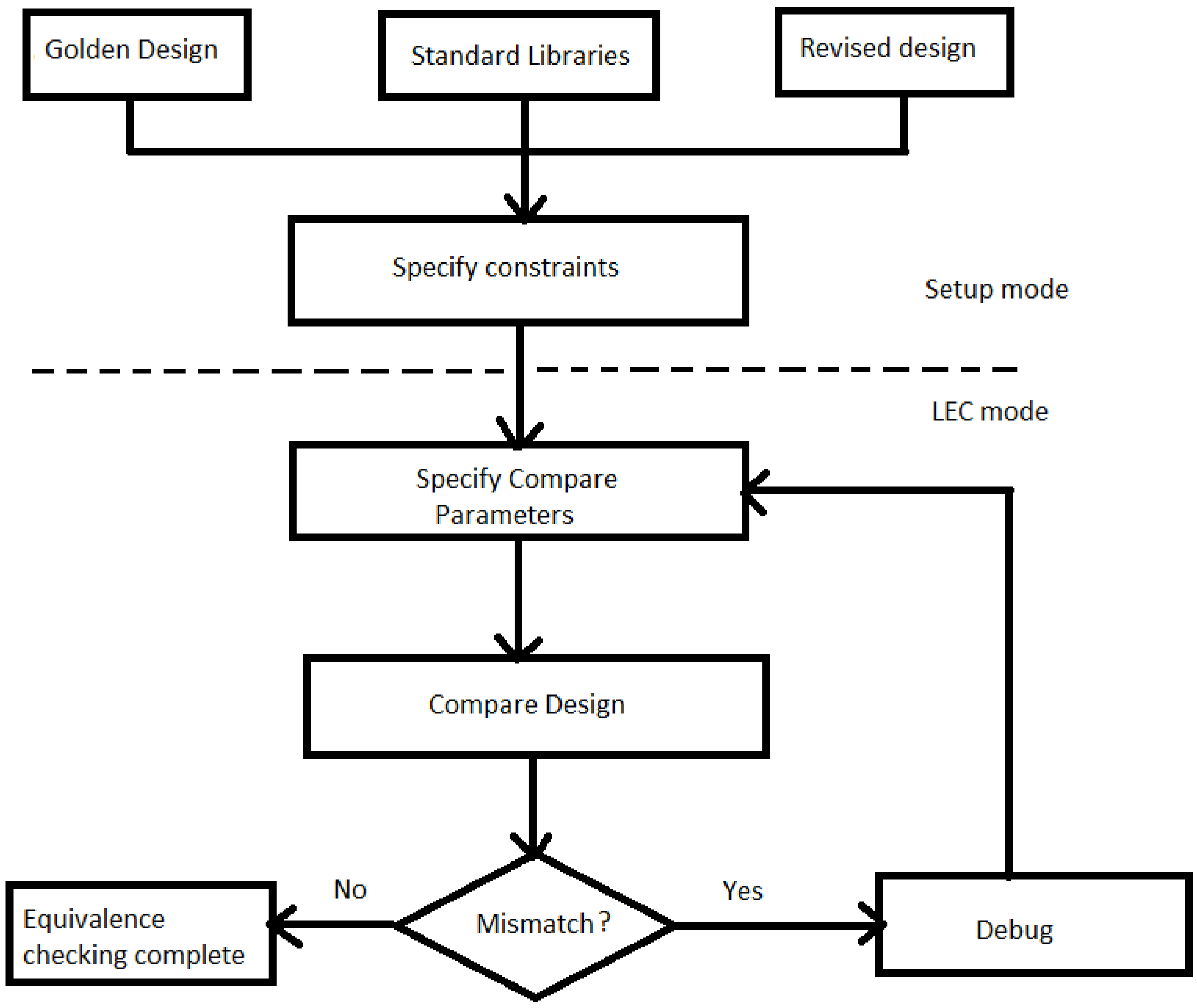

Figure 5 Conformal Equivalence Check flow

3.6 Case study on design De-multiplexer

For testing purpose I have created a test case to understand the flow of the conformal as well as the wrapper flow.

The design under test is a '**de-multiplexer'.** Two different coding styles are used for defining the demux. The golden design is considered as the behavioral design of coding and the implemented design is the dataflow design of coding.

Both the design codes were written in the Verilog format. No library file was required in this case as it is a simple logical function. To provide these files as input to the conformal a dofile was created and directly given from the command line.

Compared results were captured by the wrapper flow script in reports and a log file is generated where all the steps done by conformal are captured.

3.6.1 Code for golden design

The file 'gol_demux.v' is described below which is assumed to be the reference design. The design is in behavioral coding style.

```
module gol_demux (x, s1, s2, o1, o2, o3, o4 );

output o1 ;

output o2 ;

output o3 ;

output o4 ;

input x ;

input s1 ;

input s2 ;
```

```
always @ (x or s1 or s2) begin
  if (s1==0 && s2==0) begin
   o1 = x;
   o2 = 0;
   o3 = 0;
   o4 = 0;
  end else if (s1==0 && s2==1) begin
   o1 = 0;
   o2 = x;
   o3 = 0;
   o4 = 0;
  end else if (s1==1 && s2==0) begin
   o1 = 0;
   o2 = 0;
   o3 = x;
   o4 = 0;
  end else begin
   o1 = 0;
   o2 = 0;
   o3 = 0;
   o4 = x;
end
end
endmodule
```

3.6.2 Code for revised design

The file ‘rev_demux.v’ is described below which is the implemented design and adopts dataflow coding style.

```
module rev_demux ( x ,s1 ,s2 ,o1 ,o2 ,o3 ,o4 );


output o1 ;

output o2 ;

output o3 ;

output o4 ;


input x ;

input s1 ;

input s2 ;


assign o1 = x & (~s1) & (~s2);

assign o2 = x & (~s1) & s2;

assign o3 = x & s1 & (~s2);

assign o4 = (~x) & s1 & s2;


endmodule
```

The input and output ports are of same name in both the designs.

3.6.3 Dofile given as input to Conformal

The dofile is a file where all the commands are written and this single file is given as input to the conformal. For this de-multiplexer ‘dofile.do’ is given below. The file should have extention as ‘.do’

```
set system mode setup

set flatten model -nodff_to_dlat_zero

set flatten model -nodff_to_dlat_feedback

set flatten model -map

set flatten model -gated_clock

//........Reading libray files..........


//........Reading Golden design..........


read design gol_demux.v -systemverilog -noelaborate -golden

elaborate design -golden

set root module gol_demux –golden


//........Reading Revised design..........


read design rev_demux.v -systemverilog -noelaborate -revised

elaborate design -revised

set root module rev_demux -revised


//........Setting up mapping ..........
```

```
uniquify -nolib -all

report black box -detail

set mapping method -nophase -BBOX_NAME_MATCH -name first -nounreach


//........verify..........


set system mode lec

map key points

add compared points -all

compare -threads 1
```

All the commands are Conformal specific to understand the design and constraints given to the tool.

3.6.4 Report generated

```
=====================================================================
Compared points     PO      Total
--------------------------------------------------------------------------------
Equivalent          3       3
--------------------------------------------------------------------------------
Non-equivalent      1       1
======================================================================
Compared points are: Equivalent
  + 4   PO   /o1                    + 4   PO   /o1
```

```
Compared points are: Equivalent
  + 5   PO   /o2                    + 5   PO   /o2
Compared points are: Equivalent
  + 6   PO   /o3                    + 6   PO   /o3
Compared points are: Non-equivalent
  + 7   PO   /o4                    + 7   PO   /o4
```

Here one non-equivalent point is due to the last case of the data flow design in the revised design. This was done intentionally to understand the tool flow.

CHAPTER 4

Project Briefs

4.1 Understanding and adopting the existing flow

FEV and Static verification techniques are easy to deploy. At this stage of the project the existing flows and methodologies were learnt and tested on different models and designs. The methodology and the procedure adopted were studied. Executions of different tools used for verification at different stages of design process were learnt. Ramp up sessions were carried out to get acquainted with external tools used from the vendors for verification.

Following topics were covered during ramp up sessions:

1. Introduction to FEV
2. Working on tools and flows
3. Different types of challenges that we face in large designs
4. What kind of violations can be encountered?
5. How to debug errors?

4.2 FEV flow

FEV is a sequential process. Every stage gets the inputs from the previous stage. Various stages of the FEV flow are as shown below:

1. Extraction and compilation of required command files from input files
2. Mapping and Interface verification
3. Equivalence verification
4. Assumption verification

Extraction and compilation

- In this stage the provided netlist file or .sch file is converted into the HDL format or RTL format (.v or .vs or .vh, fomart), with the help of the tuning files. After this both the specification model and the implementation model are in HDL format.
- Now the tool converts them into the form of mathematical expressions i.e., Boolean expressions.
- The golden and revised models are converted into the binary form.
- Tuning files help in extracting the RTL model from the netlist. So these tuning files help in extracting the information from the netlist file.

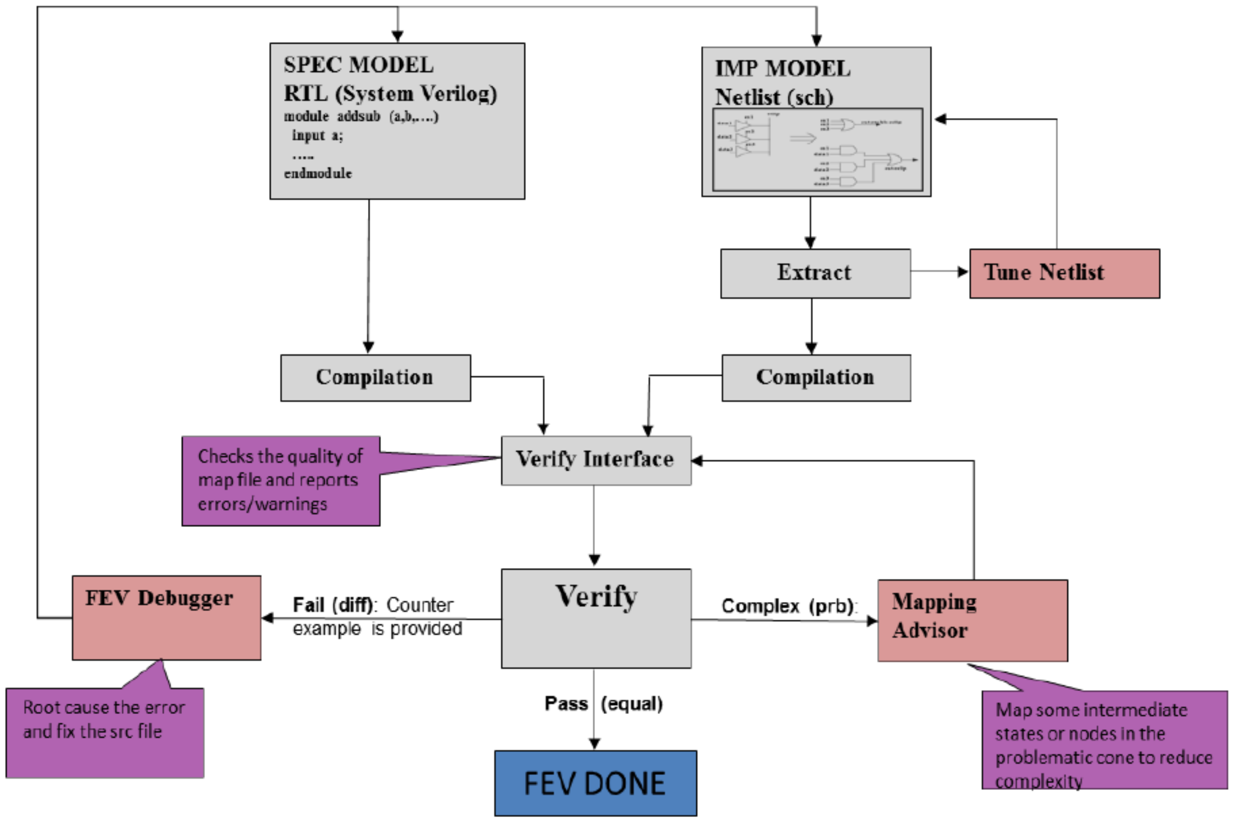

Figure 6 FEV flow

- Additionally some other information is provided, such as standard cell information, blackbox information, etc.
- Outputs of this stage are *gen_spec.log, gen_imp.log, spec.exe, imp.exe, gen.errors, gen.waivers, imp.errors, imp.waivers.*
- *Gen_spec.log, gen_imp.log* gives the detailed of the run. Spec.exe, imp.exe contains mathematical representation of the golden model and reference model in binary form.

- *spec.errors, imp.errors* consists of the errors during the run.
- *spec.waivers, imp.waivers* files consist list of waivers applied on the errors. When a waiver applied on the error, it will be masked and won't be considered as an error during the final audit checks.

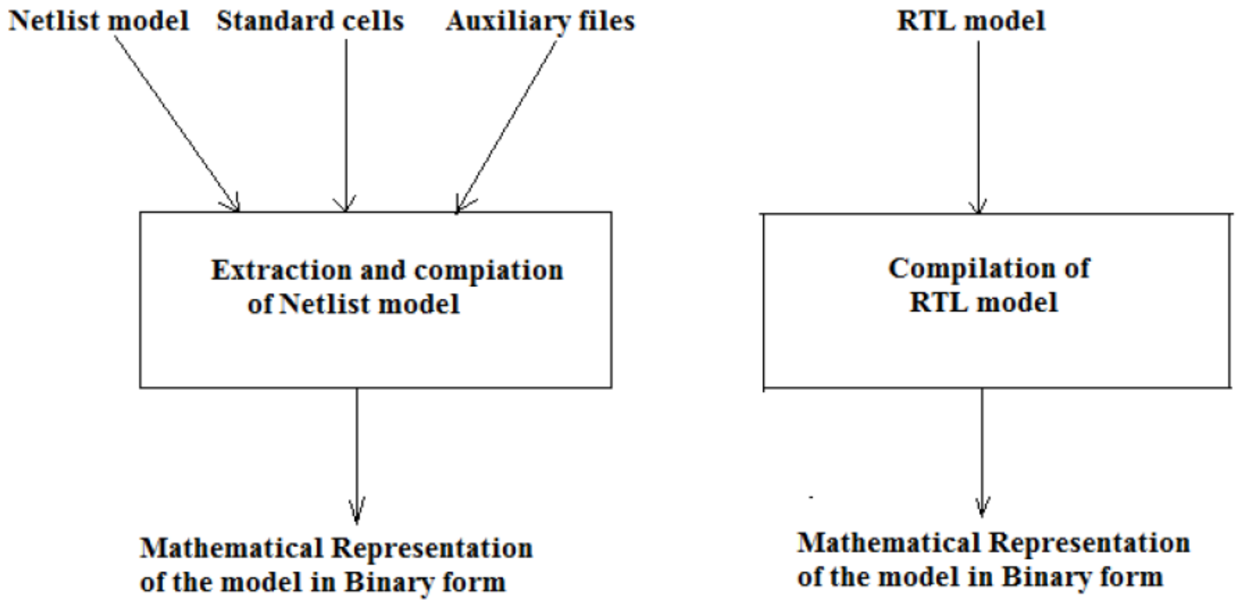
Netlist model
Standard cells
Auxiliary files
RTL model
Extraction and compiation of Netlist model
Compilation of RTL model
Mathematical Representation of the model in Binary form
Mathematical Representation of the model in Binary form

Figure 7 Extraction and compilation

Mapping and interface verification

- In order to compare the functionality of two circuits or models, we need to know the relation between their interfaces.
- In this stage both the models; golden RTL and Implementation netlist are mapped.
- We need to find equivalent nodes or points in these two models, based on which the cones are formed during verification. This process is called interface verification.
- Inputs to this stage are *spec.exe* (golden model), *imp.exe* (revised model), *synthesis flow dump files.*
- Outputs of this stage are *<fub>.map* (consists of the mapping information), *interface.errors, interface.waivers, <fub>.mapped, <fub>.unmapped, <fub>.extra.*
- Various mapping points in these models include:

1. Primary Inputs
2. Primary Outputs
3. Black boxes
4. D flip- flops
5. D Latches

Mapping points can also be called as verification points. The design is divided into two parts : Combinational logic and sequential elements. All the map points other than the Primary Inputs are called as cut points.

- Various methods are used to map the nodes in both the models

    1. Name based mapping
    2. Mapping with renaming rules
    3. Mapping Advisor

Name based mapping

Tool tries to match interface signals that are having same name in Golden RTL and Implementation models.

Mapping with renaming rules

When the naming conventions in both the models are different, tools cannot map the signals. Renaming rules are used to assist the tool to map the interface signals. When you use this command, we identify string patterns and define temporary substitute string patterns, thus enabling mapping tool to automatically map additional map points when names are not the same.

Mapping Advisor

Above two methods help mapping the interface signals in both the models. Even after that if any map points are to be mapped, they should be mapped manually. Mapping advisor helps in mapping the interface signals manually.

Equivalence Verification

- As the tools have limitations, they can't deal with the whole model. The FEV tools follow the divide and conquer approach. They divide the entire design small independent logic cones and these cones are verified for the equivalence.
- Logic cones are formed from probing point, where Probing point is a cut point where the signal output is measured. All Flop-outputs, Primary-outputs and inputs of blackboxes come under this category.
- Verify failures are debugged using Debugger, which opens the failing cone schematic. Debugger with its wide range of options (like Root cause analysis, what if simulator, etc.) helps the user in pin pointing the fault and correcting it.
- Inputs of this stage are *spec.exe, imp.exe, <fub>.map* files. Outputs of this stage are *<fub>.equal, <fub>.diffs*

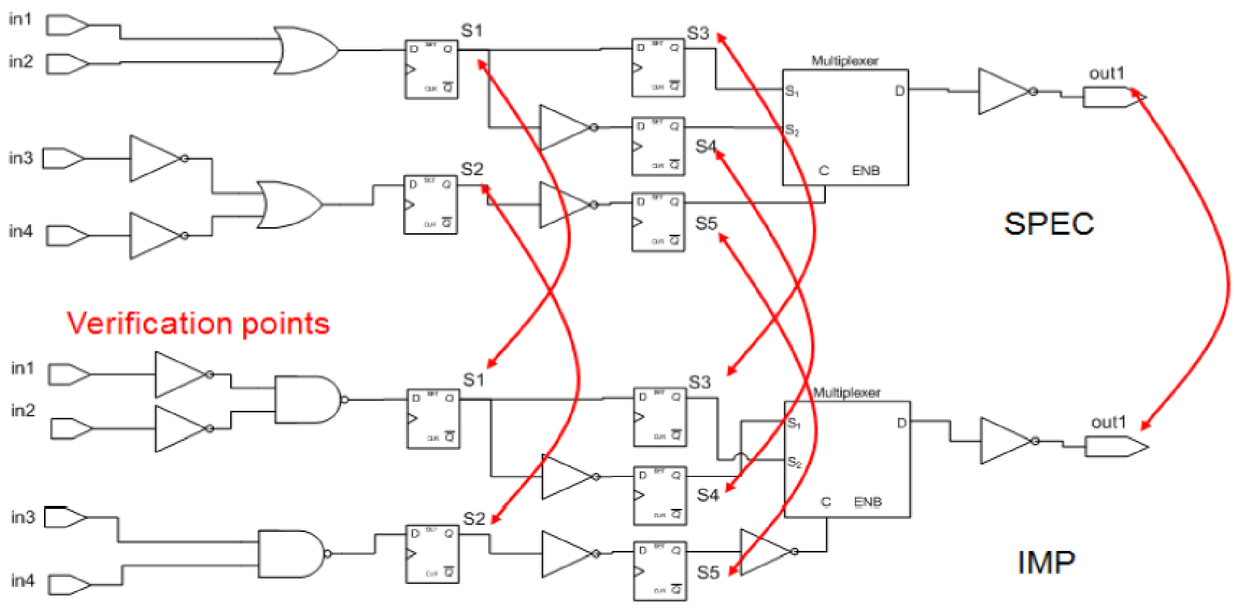

Figure 8 Cone concept

4.3 Flows created for the tool

Any verification tool will have a standard way of taking the inputs and throwing out the output files. Thus it becomes necessary for user to generate the required format of the inputs. We have created wrapper scripts for the tools used here.

When the RTL is designed, there is some information of the design which needs to be passed down to the tool as it is. This information helps the tool to understand the design easily. Such information is known as constraints which are dumped by the designer in ‘constraint file’. In all we can say that there are three main portions which we get from a designer for verification:

1. Design files – RTL file and the netlist file
2. Library files – Definitions of the standard cells and/or macros
3. Constraint files – Files which contains information to be passed to tool as it is.

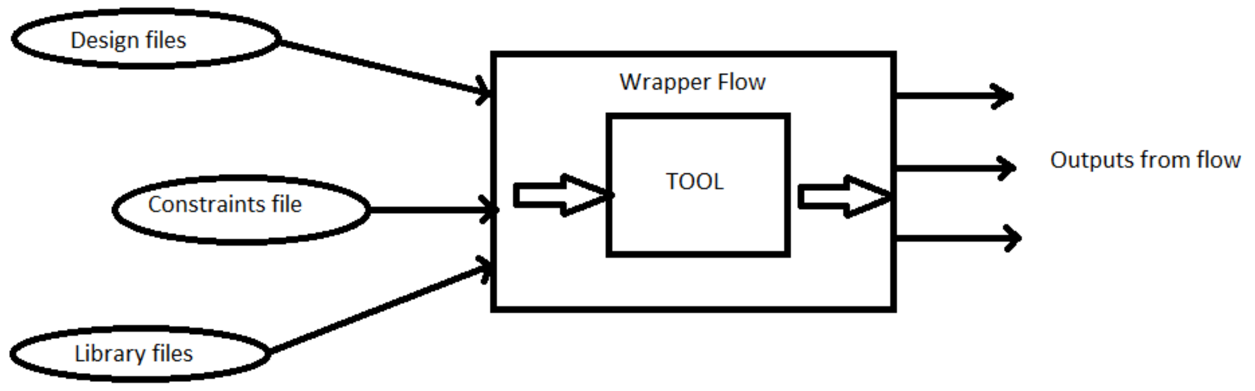

Figure 9 Inputs to wrapper flow

4.3.1 Design files

RTL file is provided in the Verilog, System Verilog or VHDL format (.vs, .sv, .v, .vhd). These files contain the information regarding the modules and their instantiation. Clocking information is also provided in these files.

Netlist file is in Verilog gate-level netlist file format (.vg). This format is to differentiate between the RTL file and the netlist file.

The design file which is taken as reference is golden/reference design and the design against which FEV is done is called the revised/target file.

4.3.2 Library Files

Design can be full customized or it can use standard cells for optimization. Library files are those files which contains the definitions of the standard cells. These cells can be specific to the project or they can be very general circuits like AND, OR, XOR, XNOR gate optimized designs or small functional circuits like ADDER, MULTIPLEXER designs. Library file can also contain the definitions of the macros used in the project.

The tool will also read the associated libraries with the design files. Some of the known library formats used by the Cadence Formal verification tool are Verilog simulation libraries and Synopsys liberty library.

4.3.3 Constraints files

The constraint files are of three types:

1. Defining the constants in the design or pins/ports connected to VSS or ground..
2. Defining the clock routing.
3. Defining the hierarchy in the design.

These files are written by the RTL designers.

No special formats are defined for these files. The file which contains the declaration of the constants and ports connation to VSS or ground is simply written in English language as follows:

port_name A	input	clock_name	0	target
port_name B	output	clock_name	1	reference
pin_name C	input	clock_name	1	reference

The file given for the routing of the clock is provided as a different file. The net of clock has a defined structure. In the clock routing file the pattern is defined and how is clock routed internally in the block is described. It also describes the multi cycle paths for some of the logic sections in the design.

Hierarchy file is a navigation file for the block distribution among the design. It will let anybody know the parent block or the children blocks of a given block.

4.3 Way to give inputs to the tool

All the files (design flies, constraint files and library files) are to be given in a standard format to the tool. Wrapper scripts are created for these. Mostly these scripts are in perl , tcl, python, PHP etc. language.

The wrapper script takes the files of design and converts all the information in the form of dofiles. Dofiles are files that contain the steps to be done by the tool. The directory of the dofiles generated by the wrapper script is as follows:

1. Read_golden_design: In this the RTL or netlist file is read as the reference design for verification.

2. Read_revised_design: In this the RTL or netlist file is read as the implementation design for verification.
3. Read_library_files: In this the library files are read.
4. Map files: All the constraint information is organized in this directory. There can be more than one file in this directory.
5. Verify_design: In this all the directory paths are combined as stages and a single file 'design.verify.file' is given as input to the tool. The verify_design file will direct the tool to all different directories of dofiles to pick the design, library and constraints.

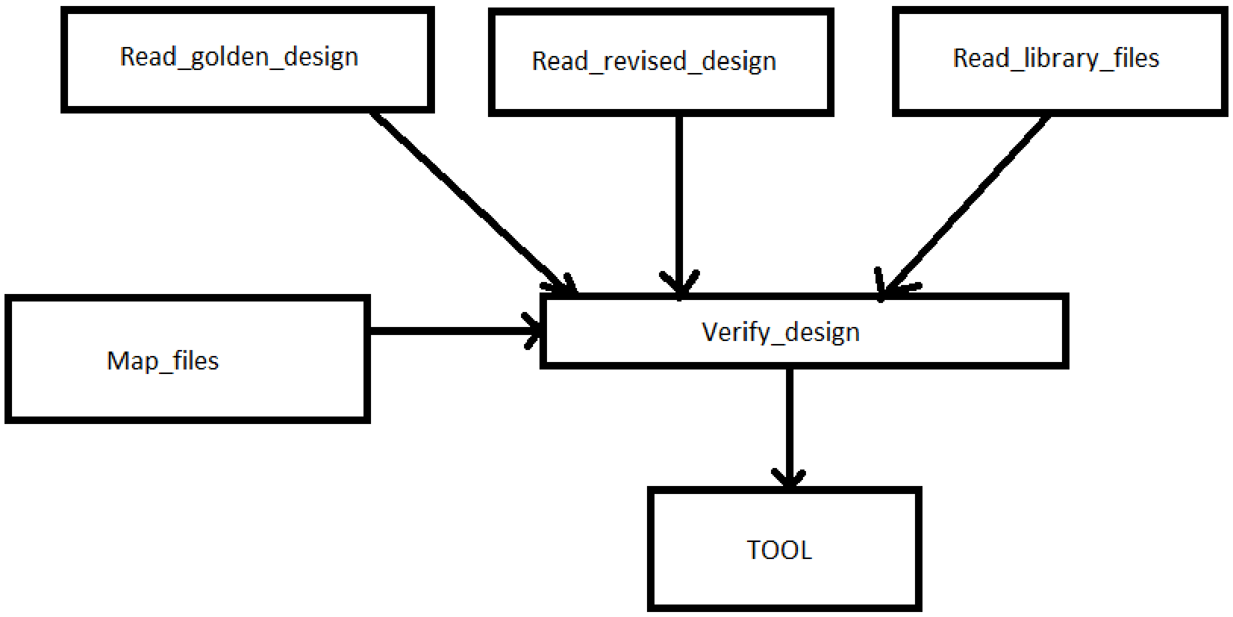

Figure 10 Inputs to the tool

The wrapper flow also guides the tool to create the reports of the verification in a folder that contains information of all the mapped points, compared points, equivalent and non-equivalent points in the design.

CHAPTER 5

Implementation of the Project

5.1 Developments in the flow

With the large CPU designs comes a large number of input information. If the constraint files generated by wrapper scripts are not in correct format it will become very difficult for the tool to understand the design and the verification will not pass. Thus the wrapper scripts should be very strong. Different flows can be associated with different tools according the need of design process or the project requirement. The wrapper script is a package that contains many small modules of scripts used as the utilities.

Such scripts are not developed in a single shot. Now-a-days the designs of SOC have become very complex thus the enhancement in the script is also required to make it compatible with the SOC designs. It becomes very easy and less time is consumed if for all the projects same flow is used. Some constraints are global for a particular project and some constraints are unit/block specific.

The main stage of the verification is the mapping stage. The mapping should be done properly so tool can compare correctly the two designs under verification. For this the flow should produce all the correct constraints required for mapping. The flows used for the larger VLSI designs are very complex due to complexity in the design itself. So many times the tool is not able to map the constraints correctly. Many enhancements are made in the flow during the progress of the design process.

5.1.1 Enhancements in the script of flow

FEV can be done at any level from FUB to full chip. Information is available for every section of the full chip, sub full chip, partition and even for every FUB. Information is processed explicitly for the unit on which we run FEV. All other information is suppressed. The hierarchy file, input to flow script processes all the information regarding the unit and about the hierarchical structure of the design. Many enhancements are made in the script for new and upcoming projects. New set of code is added in the script.

Resolved the problem in script where some designs were actually passing FEV but result showed them failed.

For a certain number of units on which the FEV was performed some internal procedure files were applied. Due to this additional procedure files the mapping was done iteratively by the tool. As a result the flow was broken and wrong reports were generated. Thus the scripted was corrected because the earlier logic did not work for this complex section of the design. Regression was carried out to check that no other units or blocks are affected by the corrections made in the script. It was a success and the script is working fine.

For some of the complex unit runtime was very large than expected. Some rules were added to the unit specific procedure file and the amount of run time was reduced to a great extent.

There can be differences in the representation of the names for some ports/pins/flops in RTL and netlist. We need to create a way so that the tool can understand these changes and map the correct ports/pins/flops. This is done by adding renaming_file in the flow. These rules are global for a specific project but some additional rules are required for particular blocks. Without this renaming file it becomes very difficult for the tool to map the points. It is very easy to map points by names. Another way of mapping is functional mapping. Renaming file helps in name based mapping. Many new rules were added for complex designs and thus run time was reduced by 40 %.

Automated the scripts, to a certain level, used for extracting functional characteristics of a block and create the same result outside the project environment to test and resolve pure tool bugs in it.

FEV for any block (FUB or section) fails either due to some tool bug or due to some mismanagement in the flow.

The failures due to mismanagement of flow can be resolved by some amount of debugging and finding out the solutions or enhancements in the flow. But tool bugs can only be resolved by the vendor. For this purpose same results should be produced outside the project environment without sourcing any setup. All the technology specific information should be suppressed. Definitions of libraries and macros should be removed. This process is called ‘testcase preparation/extraction’. Thus scripts are prepared to reproducing the exact same results which is delivered to the vendor for debug. Human intervention was required to a very great extend for this process.

I have managed to reduce the human involvement to a certain level and have automated the script. All the collaterals are collected and library functionalities are extracted by this script and on giving single tool command the testcase can be run and produce same results with bugs to resolve at the vendor’s end.

5.2 Quality Analysis done for enhanced features in the tools and flows

Quality Analysis or simply QA gives a convincing way to hallmark the quality of the results that are produced by the tools/flows. The process also guarantees that the tool/flow is producing truthful results for the designs that can be sent to the fabrication lab.

5.2.1 Quality Analysis of the wrapper script of the flow

A small test script was generated to carry out QA on the enhanced wrapper script to check that nothing went wrong due to enhancement and the enhancements did get applied at the required places.

The test script generates an excel sheet giving comparison between the results thrown out by the validated wrapper scrip and the wrapper script with the enhancement.

One script is created which runs the flow script on ‘n’ numbers of the blocks present in a given ward. Another script captures the results of the blocks in an excel sheet. This takes about half a day and results are ready for detailed analysis. These scripts are very generic and can be performed on ‘n’ number of versions of the flow as well as on ‘n’ numbers of blocks.

5.2.2 Quality Analysis of the Cadence tool used for verification

It is not necessary to always enhance the wrapper scripts and make it compatible with the tool. Some general features can also be added in the tool so that it can handle some of the complexities of the designs.

As a part of development some very important features were added to the tool and were delivered to the company. A quality analysis was carried out for this purpose. The QA was done between different versions of the tool.

QA was done on two different tools used by the company one of Cadence and another of Synopsys. Different scripts were written for running verification on a lot of blocks (~ 200 in numbers) between 3 versions of the Cadence tool and 2 versions of the Synopsys tool. The test script can be run on ‘n’ numbers of blocks and versions of the tool. The results were captured in an excel sheet containing the parameters like passing points, failing points, aborting points, runtime, memory usage.

One bug was caught of reporting 'unreachable Primary Input' due to some complexity in the blocks. This was reported to the vendor and the bug was removed by them in their latest version of the tool. This was also analyzed during the QA.

5.2.3 Quality Analysis of the Synopsys tool used for verification

A QA was performed for the Synopsys tool similar to that of the Cadence tool. Around 200 units were tested and on 2 different versions of the tool the QA was done.

The results were dumped in an excel sheet. Parameters used for QA were equal points, not-equal points and runtime. New version has same results as the previous version and the runtime was reduced of every unit. Three different scripts were created, one to copy all the units to a local area from the centralized ward, second to run all the units in parallel on dedicated machines and third to capture results from the logs generated by the tool.

5.3 Migration to a generic flow for all kinds of projects

For different projects under execution different wrapper flow scripts are used. The idea is to move to a generic flow which can be used for all kinds of projects. This will save a lot of time and resources consumed in maintaining 2 different flows, updating and validating those flows.

Most of the projects here use the tcl based scripts. But as the SOC designs came into existence some of the sections of the design had to use a different flow written in perl scripting because the constraints needed for such sections were very large in number. We have tried to verify these particular sections from tcl based flow. Design files and library files are provided as it is. The main challenge was to migrate the constraints from perl based flow to the tcl based flow. The tcl flow should take the constraint file as input and produce the map files in same format that the perl flow produces.

Equivalent tcl based scripts were created to get the same constraint files in standard format used by the tool. We managed to pass 5 such units from these sections. We have created standalone runs without sourcing any environmental setup.

This development is successful for handful of units for now. Regression is required to be done on all the units to validate the process and scripts created. The regression will declare more enhancements to be done in those scripts. Thus this is still in progress and is not implemented on on-going execution projects.

5.4 Internal support to the team

Working extensively on the tools I was able to resolve some of the tool issues and flow related issues to bear some load of my team members.

5.4.1 Resolved some of the tool bugs

Macros defined by the library team are specific for the projects and are validated. But some time a small bug may slip from our eyes and can result in wrong logic implementation in designs.

One such bug was caught in macro definition. A signal was un necessarily negated inside macro and was passed to the design. Thus logically opposite results were observed. The design was under observation for a long time and finally the bug in macro was found. This required the re-synthesis of the RTL design. But due to early stage formal verification big loss of time, money and resources were saved.

5.4.2 Reduced the run time for some of the complex units

Tool can map points by looking at the names very quickly. After mapping points by name it maps all other points by looking at their function. Thus functional mapping takes a longer time. One way to reduce the run time is to add the renaming_file to increase mapping by name. Many block level renaming_files were added to different blocks and the run-time was reduced. This is included in the 'Best Known Methods' for generic usage.

Tool aborting at some complex FUBs

Tool can abort at some of the FUBs due to following reasons:

1. Model not built properly
2. Numbers of sequential elements were less compared to combinational logics.
3. Number of inputs exceeded then the capacity of cone.

One of the techniques of overcoming the aborts is to provide additional map points. We need to provide extra cut points, so that more small cones will be formed. It will be easy for the tool to deal with large number of small cones, rather than dealing with small number of large cones. Tool options must be used efficiently to analyze the large cones with extra effort.

Tools were aborting on some of the FUBs. Effective utilization of the options in the tool will eliminate this problem. Extensive work on some of the FUBs with all the available options in the tool has provided required solutions to resolve the aborts. All these “Best known methods” (BKMs) are updated in the flow.

The debugging of a violation issue in some of the FUBs has shown that, some of the coding methods are not good for FEV runs. The map file of the FEV has some issues related to the mapping of the Primary inputs, primary outputs and black box inputs and outputs. The reasons for the violations are traced. The violations are due to the reasons:

1. Direction mismatches of the port
2. Declaring the port as inout
3. Mismatch of the blackbox information

In all, I have explored the tools to a good extent along with the completion of this project.
This has led in better understanding of the company’s flow of releasing a quality product to market and efforts put by every single person involved in the company projects.

CHAPTER 6

RESULTS

6.1 Results of development of flow script

6.1.1 Runtime reduction by adding renaming_file

Runtime reduced for big and complex blocks by 40% by adding renameing_file to the block constraint file. There are many such complex blocks which take hours to complete the verification. Reduction in runtime of such blocks results in overall runtime reduction of the sections in which those blocks are sited. The results are shown in the table below.

S. No	Name	Runtime before adding renaming_file	Runtime after adding renaming_file
1	Block_1	6 hours	3.5 hours
2	Block_2	5 hours	3 hours

Table 1 Adding Rename_file reduces the block runtime

6.1.2 Correction in false reporting of compared results

Due to complexity in blocks the tool iteratively mapped points, compared those points and produced the results. Although the blocks passed the verification but the flow script was broken due to repetitive steps. As a result of which the compared result shown by the flow were false. Below are the results showing the correct results.

S. No	Block	Version	Status	Equal points	Not-equal points
1	Unit_1	Pre_version	Pass	2	0
		Post_version	Pass	249	0
2	Unit_2	Pre_version	Pass	1	0
		Post_version	Pass	556	0

Table 2 Corrected false reporting of compared points

6.2 Results of different Quality Analysis

The following tables give an example of the results of QA done on the flow script and two tools.

6.2.1 QA results of the flow script

Sr.No	UNIT	Version	Status	Fail	Pass	Unmapped	MEM	CPU
							(MB)	(SEC)
1	unitA	Ver_1	pass	0	1322	0	2465	93
		Ver_2	pass	0	1322	0	2450	90
2	unitB	Ver_1	pass	0	4	0	2466	97
		Ver_2	pass	0	385	0	2466	97
3	unitC	Ver_1	fail	48	280	59	3944	468
		Ver_2	fail	48	280	59	2746	300

Table 3 QA results for Wrapper flow script

This result are illustrating that there is reduction in the runtime due to additional rename files. The enhancement is also captured here of false compare results.

6.2.2 QA results of the upgraded version of tool

The results are very clear pictorial of that any generic change in the tool or any enhanced feature to carry out complex verification will not cause any barrier in the implementation of the new version for on-going as well as new projects.

S. No	**Unit**	**Equal**			**Not-equal**			**Status**		
		Ver1	Ver2	Ver3	Ver1	Ver2	Ver3	Ver1	Ver2	Ver3
1	Unit1	296	296	296	0	0	0	pass	pass	pass
2	Unit2	277	277	277	0	0	0	pass	pass	Pass
3	Unit3	294	294	294	0	1	1	fail	fail	fail
4	Unit4	285	285	285	0	1	1	fail	fail	fail
5	Unit5	244	244	244	0	1	1	fail	fail	fail
6	Unit6	304	304	304	0	0	0	pass	pass	pass
7	Unit7	226	226	226	0	0	0	pass	pass	pass

Table 4 QA results for the tool

CHAPTER 7

Conclusion and future scope

7.1 Conclusion

Time-to-market is one of the key factors for any company to release its product and get hold on customers. The designing part should be completed in a short span of time so that verification and fabrication can be completed without missing technology window. Verification of the design takes about 70% of the total turnaround time. Aim of this project was to reduce the time of verification of design so that the total turnaround time can be reduced by some amount. Efforts made in reducing verification run-time of small blocks, units and partitions have cut down the overall verification time of design. Today the flow and FEV methodology is very stable but many more enhancements can be made for better FEV results.

7.2 Future scope

There are different flows to drive the tools which are adopted for different projects. Having a single generic flow for all projects will be a lot easier to maintain and manage all the resources. Different versions can be branched from the generic flow for a specific project if needed. New features can be added in the tool so that it can be made specific for the projects.

REFERENCES

1. Intelpedia Web contents : https://intelpedia.intel.com/

2. Training materials and ppts on FEV

3. Koen van Eijk, Formal Methods for the Verification of Digital Circuits, PhD Thesis, Eindhoven University of Technology