BACK END MEMORY COMPILER VALIDATION

Thesis submitted in partial fulfilment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By
Manpreet Kaur
(Roll No. 801632024)

Under the supervision of:
Mr. Sumit Miglani
Assistant Professor, CSE Department

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY
PATIALA – 147004

June 2018

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Back End Memory Compiler Validation"**, in partial fulfilment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar Institute of Engineering and Technology, Patiala, is an authentic work carried at STMicroelectronics Greater Noida under the supervision of industrial mentor **Mr. Rohit Pandey** and faculty mentor **Mr. Sumit Miglani** and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Signature:

Manpreet Kaur

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Mr. Sumit Miglani
Assistant Professor
Computer Science and Engineering Department
Thapar Institute of Engineering and Technology
Patiala

ACKNOWLEDGEMENT

First of all I wish to acknowledge the benevolence of God who gave me courage and strength to face the challenges and to overcome the obstacles that occurred while working on this task.

It gives me immense pleasure in expressing thanks and profound gratitude to ***Mr. Sumit Miglani***, Assistant Professor, Computer Science and Engineering Department, Thapar Institute of Engineering and Technology, Patiala for his valuable guidance and continual encouragement throughout this work. The appreciation and continual support he has imparted has been a great motivation to me in reaching a higher goal. His guidance has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come.

It is also worth thanking ***Mr. Ashu Talwar***, Project Manager, STMicroelectronics Pvt Ltd and ***Mr. Rohit Pandey***, Tech Lead, STMicroelectronics Pvt Ltd for their invaluable guidance and support during the course of the project.

It gives me an immense pleasure to thank ***Dr. Maninder Singh***, Hon'ble Head of Computer Science and Engineering Department, Thapar Institute of Engineering and Technology, Patiala for his kind support and providing basic infrastructure and healthy research environment.

I would also thank the Institution, all faculty members of Computer Science and Engineering Department, Thapar Institute of Engineering and Technology, Patiala for their special attention and suggestions towards the project work.

- Manpreet Kaur

801632024

ABSTRACT

We use various electronic gadgets in our daily life. Everything from cooking to music uses electronics or electronic components in some way. These electronic gadgets like cell phones, touch sensors, desktops contain silicon chip which is intended to perform a specific task. Inside each silicon chip is a **semiconductor intellectual property core**, **IP core**, or **IP block** which is a reusable unit of logic, cell, or integrated circuit (commonly called a "chip") layout design that is the intellectual property.

This intellectual property also called as **cut library** at initial level is generated using Back End Memory Compiler product. This cut library or intellectual property is generated after cut generation. An Intellectual Property contains various views like CDL (Circuit Design Language), GDS (Graphical Design System), LEF (Library Exchange Format), etc. To ensure correct working of device, Intellectual Property must be correct technically and in terms of functionality. This is possible if its various views are validated.

Back End Memory Compiler Validation is an automated tool to validate the various views of a cut library (Intellectual Property at its initial stage). It saves much of time involved in validation of cut library and also validates the BE Compiler product used to generate the cut library. This tool reduces the risk involved in case errors are found at final stage of IP delivery since all errors are detected at initial stage itself.

Thus in order to provide client with correct and dynamic memory, **Back End Memory Compiler Validation** product is assisting the needful.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Overview

Back End Memory Compiler is a software which is used to design memory compilers of different technology and different flavour that differs in terms of various parameters like dual or single port, size in nano meters and memory capacity. The Output of the Back End Memory Compiler is termed as 'Library'. Library is collection of IPs (Intellectual Property) and an IP contains various cells and each cell has views which are the representation of the cell. All the cells have : Layout, Abstract, Schematic, Symbolic and Timing view. This library is often referred to as **Cut Library** where each library may contain multiple cuts. Each cut in a library has different configuration. To generate a library, Back End Compiler is used which is a software to design memory compilers of different technology and different flavours.

BE Compiler is automation to generate cuts of various capacity (collectively called cut Library) for a particular techno and particular flavour. Previously, this work was done manually. For even a slightly different configuration, whole procedure of building compiler product had to be repeated which is very time taking. BE compiler reduces much of man time memory compiler products with different configuration can be made in least time. Each of these BE Compiler is further used to generate library according to the technology and configuration.

Once BE compiler product is ready, it needs to be tested in order to be used further for designing chip. In order to insure precise delivery of product, several validation steps are performed to check the functioning of generated BE product. Cuts are generated with varying configuration so that all test cases are covered. For cuts generated using the BE product, all outputs are checked manually for correctness. Same procedure is repeated for validating all BE products which is very time taking and consumes long hours sometimes leading to delayed delivery of product.

For this validation, **BE Product Validation Tool** is created. It is capable of performing full testing the BE product's library. Testing of single or multiple cuts can be performed.

1.2 Motivation

Successful use of an IP (Intellectual Property can be made) when it is functionally corrected, validated and is on time. Validation is plays major role in timely and successful delivery of IP and its use in SoC. A multinational firm – Arteris Inc. that builds up the on-chip interconnect technology (utilized as a part of SoC semiconductor designs for different types of electronic gadgets) conducted a survey [1] related to top design challenges that contributed to delayed delivery of SoC. According to the survey reports, quality problems and bugs contributed to 8% and debugging designs contributed to alarming 11%. These results clearly highlight the importance of validation. The cost involved is much more in case issues are found at later stages since fixing these issues needs time and money both. Jean-Pierre HELIOT in his paper [2] on SoC found that In the semiconductor industry, System On Chip (SOC) is extremely complex and success of SoC requires best in class knowledge and proper level of validation on silicon of each block of library/IP's used within new chip. Hence, a proper tool or some automated technique is needed to find the issues at initial level.

Following chart shows top design challenges faced in designing SoC as per the survey conducted by Arteris Inc.

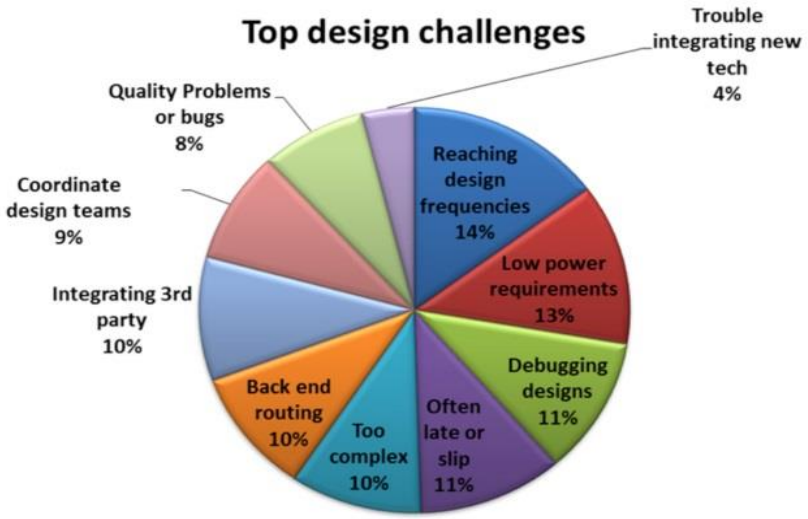


Figure 1.1: Top Design challenges (Source: Arteris Inc.)

1.3 Need for New Validation Approach

Manual validation of BE product takes very long time. If some errors are discovered at a later stage, it may lead to delay in release of the product. To test this automated approach of creating BE product, outputs i.e. gds and cdl and other output views of the product had to be checked manually. Gds being a binary file, manual checking was a very tedious process. To save this validation time, a tool was needed so as validation time is reduced. Since, same checks are required to validate any BE product, idea of automating the process came into picture. As the procedure of creating a BE product was automated, a tool for validation of the product could also be built using automation.

1.4 Objectives of New Validation Approach

The major objective behind this thesis are as follows:

a) To propose and deploy time saving validation technique for a Back End Compiler.

b) To validate all views of a BE Compiler using automation.

c) To provide the functionality to check or validate whole cut library, even if number of cuts are very large.

d) To get proper validation report for each version of Back End Compiler for its submission, so that the cause of issues occurring at later stages can be easily identified.

1.4.1 Expectations From The New Solution

Expectations from the new Back End Compiler Validation solution are as listed below:

Functional (User):

a) Validation of a single view and automated enabling of checks with respect to the cut library provided as input.

b) Automatic setup creation and enablement of check if an entire library or multiple views are provided as input for validation.

c) A unified solution with a complete coverage of checks. Encapsulating all required BE checks inside a single framework.

d) More flexibility to user in terms of configuration of checks and the level of validation required.

e) Option to run checks on default cut in case of very large number of cuts in cut library.

f) The solution should be able to handle multiple cuts simultaneously.

g) Flexibility to the user to add any new check if required.

Technical:

a) Improved check algorithms in terms of run time.

b) More use of data structure than intermediate files.

c) New architecture should be flexible enough to use any other In-house or Third party tool.

d) Functional coding, where in all common functions are in a separate script that is invoked by main script.

1.5 Thesis Outline

The organization of this thesis is as follows:

Chapter 2 (Literature Survey) describes the fundamentals required for this thesis. It describes the Significance of Validation Phase in the Library development Flow. It describes the review of the existing approaches for the Validation process. Pros and cons of the Validation Approaches has been mentioned as well.

Chapter 3 (Proposed Solution: Back End Memory Compiler Validation Tool) describes the layered architecture of the framework and its significance.

Chapter 4 (Design and Implementation) describes the whole development of the tool including the prerequisites, implementation, user interface and the output directory structure.

Chapter 5 (Results and Analysis) describes the various experiments done on Back End Memory Compiler Validation Tool and the analysis of results is also outlined.

Chapter 6 (Conclusion and Future Scope) describes the conclusion of the thesis. Future directions related to the topic are also outlined in this particular chapter.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction to Back End Memory Compiler

For an IP (intellectual Property) or library development, a Back End product is necessary. For a back end product's development, user gives its specification and the whole product is developed which takes 10-15 months of time to get fully mature. All the parameters are fixed according to given specification. If there is any change in parameters then, again we need to start from scratch to design product. If same product is needed by some another customer with different configuration then again it will take 10-15 months of time. So, an automated approach for the development of product was needed. According to Ender Yılmaz and Gunhan Dundar [3], automation technique should include layout generator and netlist generator. Thus, BE Compilers came into picture. BE Compiler is a software to design memory compiler with variable capacity. It is automation of the whole manual process. Making use of BE Compiler, we can develop whole BE product with required specifications. There are different technologies like SRAM, ROM etc [5] of compilers and different flavours for each techno. Thus, a wide variety of compilers can be created using similar coding. A lot of time is thus saved using BE Compiler automation. Ender Yilmaz also proposed [4] that layout automation reduces the design time spent by qualified engineers, thus reducing the cost and the time to market.

2.2 Types of Memory Compilers

There are three types of memory compilers:

1. Mono Architecture
2. Split Architecture
3. Bank Architecture

Each of the above listed compilers have certain things in common viz. CORE, ROWDEC or DECODER, CONTROL, IO, DCOL etc which are explained below:

- **Core**

Core is the ultimate heart of the memory as it holds the data. Core is made of small memory cell each capable of holding 1 bit of data. Now these basic memory cells are multiplied n times to give shape to the core that is capable of holding n words. These n words are distributed among the rows and column, say M rows and N columns, thus memory is identified as M*N memory.

- **Decoder or Rowdec**

Rowdec is a vertical block that is used to identify the rows in a core. To identify any particular memcell (logic for single core memory cell) in a core, rowdec identifies the row in which that particular memcell resides.

- **IO**

Like rowdec is used to identify the rows of the core, IO is used to identify the columns of a core. Thus by the combination of rowdec and IO, any particular bit within a core is identified. The block that transforms the necessary requirement of row number to the rowdec and column number to the IO is Control.

- **Control**

Control block controls the working of rowdec and IO by passing them the required information regarding the row number or column number in the form of electronic signals. Thus, by the combination of above 4 blocks various architecture of compiler came to being.

- **Basic block existence**

Each of the above blocks are made up of basic cells that give them the shape and functionality after which they are capable of offering the work that is expected from them. The point here worth mentioning is that each of these basic cells is made up of circuitry derived from the interconnection of various transistors and logical gates and during the signal processing due to flux or electronic and magnetic inferences there is a danger that these cells may give wrong result (Finally what they are holding is just a bit and noise can easily invert the bit from 0 to 1 or vice-versa). Thus because of current fluxes or inferences or to meet the required voltage differences, new

architecture was proposed. Thus by the combination of above 4 blocks various architecture of compiler came to being.

2.2.1 Mono Architecture

Mono Architecture is the basic architecture of all three. This Architecture works fine when the memory design is of small bits(i.e columns) but in case of large number of bits, voltage difference between the 0th bit and nth bit is very large that may affect the performance of the memory.

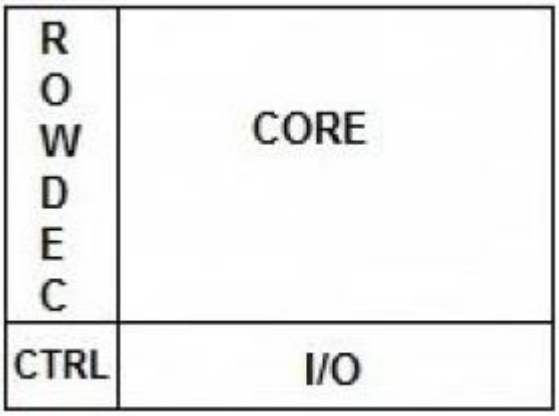


Figure 2.1: Mono Architecture

To overcome the above problem, Split architecture came into existence.

2.2.2 Split Architecture

In split Architecture, as can be seen from the figure, core block is divided into two parts i.e left core and right core. As a result the voltage difference between the first and last bit has reduced to a significant level. Due to core split, voltage difference is now identified between 0th bit and middle bit and (middleBit +1) and last bit. Also IO block is split into left IO and right IO as the purpose of IO was to identify the bit lines so if the core block was split and to identify all the bit lines, IO block too was split. Though the voltage difference between of the bit was taken care but in case of large number of rows similar high voltage difference may encounter.

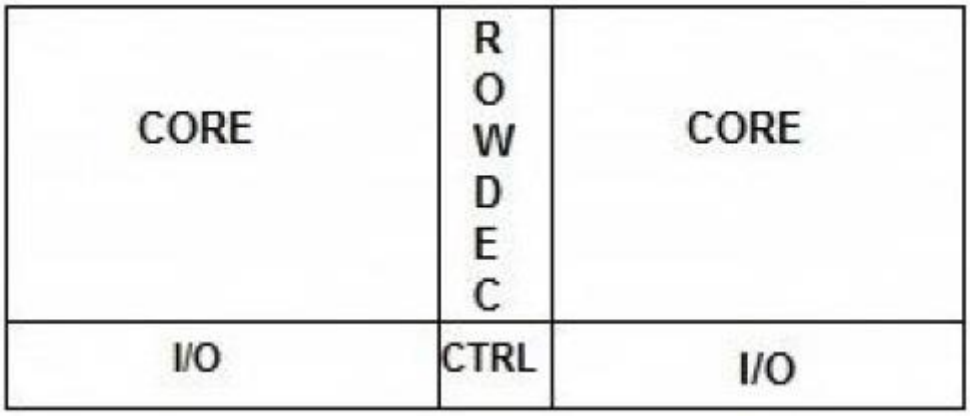


Figure 2.2: Split Architecture

To overcome these issues, bank architecture was proposed.

2.2.3 Bank Architecture

As it can be identified from the figure, each of the left and right core is further partitioned horizontally to reduce the word-line voltage difference. Like split architecture in which IO block too was split to identify all the bit line, bank architecture along with core, Rowdec block too is split to identify all the word-lines. Bank architecture has a global control that controls local control and split global IOs to control split local IOs.

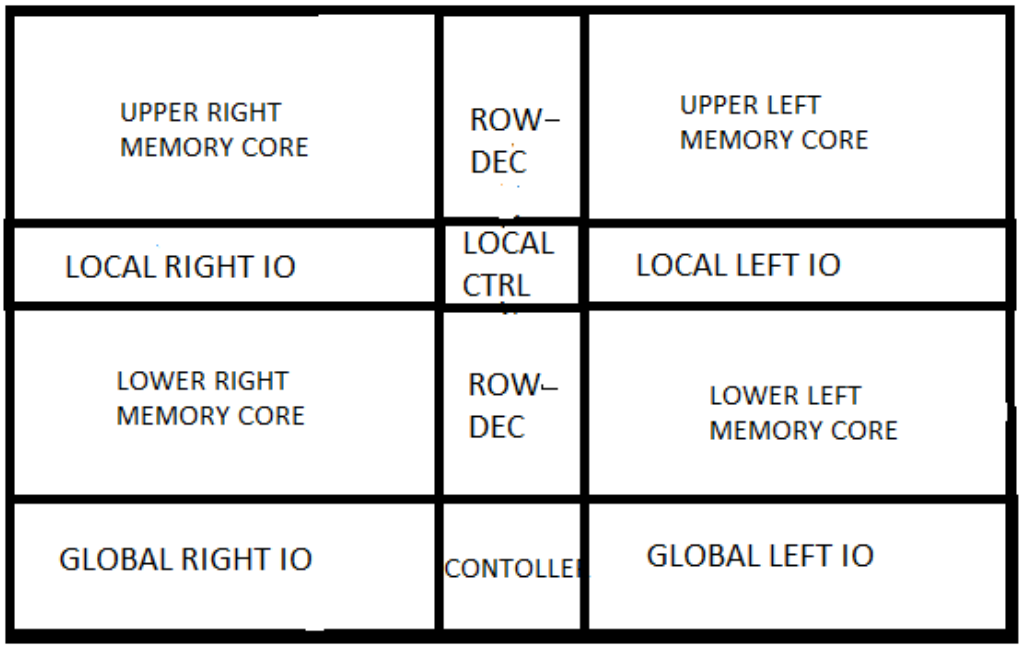


Figure 2.3: Bank Architecture

2.3 Important Terminology

Memory is constantly given as far as words*bits. In any case, when we create memory compiler for library generation we change it over into rows and columns.

Rows= words/mux;

cols=bits*mux;

2.3.1 Importance of Mux

Assuming present structure of memory is not suitable for SoC, for instance if the aspect ratio is not desirable. We can diminish the structure of memory and expand the width by keeping up the same size as far as words*bits factor is concerned. Memory =rows*cols = ((words/mux)*(bits*mux)) = words*bits. In spite of the fact that we have changed over words*bits into rows*col still we are looking after same memory size. With mux = 2 Rows = 512/2 = 256 Cols =16*2 =32. So now width is twice and height is half. Now in the event that we utilize mux = 4 height will be diminished to 1/4 and width will be increased by variable of 4. Along these lines we can change the perspective proportion with the utilization of mux [8].

2.3.2 Importance of Row-decoder

Rowdec is used for selecting a specific row in core. Presently, single rowdec can select from predefined set of rows for example: 4. In this way, rowdec can choose 4 rows only. The arrangement of columns from which a rowdec can choose a line is known as Rowdec unit. In this way we require numerous rowdec units to get to all columns of memory. The reason we can't choose all rows with single rowdec is because the quantity of columns are not fixed. It relies upon words and mux so taking the chance that we make single rowdec it won't work when configuration changes. In this way number of rowdecs is computed on premise of columns.

NoRowdec = (lines / rowdec unit) + (columns % rowdec unit == 0?0:1); [9]

2.3.3 Importance of IO Cells

Number of IO cells depends upon number of bits. One IO will be put corresponding to each bit. In mono structural planning, bits are even or odd so placement of IO cells is not much complex. On the other side, in case of split and bank architecture it is to be decided that on which side we need to put one additional IO [8] as here our left and right part will not be identical.

2.3.4 Importance of Wrapper Cells

As the name suggest, it is wrapper to a cell and is used to provide additional functionality to that particular cell.

2.3.5 Importance of Environment Cells

Environment cells are used in between core and IO cells. There are two sorts of environment cells: Introductory one is *Vertical Environment strap* which is used in core and IO while other is *Horizontal Environment strap* which is used in core and Rowdec. Environment Cells are used to absorb electromagnetic impressiveness made by the group of cells. So after each predefined number of cells, strap cells are used so that effect of electromagnetic impressiveness [9] is diminished as far as possible.

2.4 Process of Designing Memory

The basic stage in IC organizing is the making of the design using software called virtuoso. This configuration is checked using DRC to affirm that the outline does not violate any standards chosen in the DRM which is required for the best working of the chip. By then, once the design is DRC clean it is separated and the flair model of plan is checked to guarantee that the course of action addresses the same circuit as it is in the flair model. Following is data about the checks :

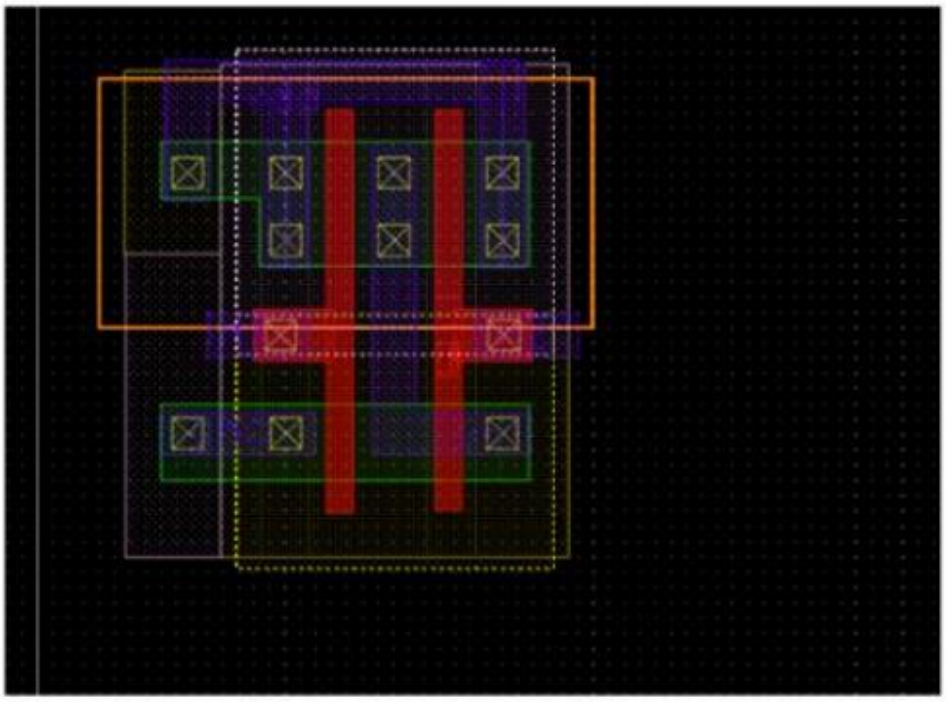


Figure 2.4: Memory Layout

- A diagram block set focuses on certain geometric and structure impediments to ensure attractive edges to record for variability in semiconductor passing on star cesses, with a specific choice of target to ensure that a liberal bit of the parts work precisely. While diagram statute checks don't reinforce that the organization will work precisely, They are made to watch that the structure meets the framework obligations regarding a given layout just like brick layout handles building. DRC programming by and large takes as enter a course of action in the GDSII standard connection, and produces a report of plan to decide encroachment that the maker may change.Purposely "extending" or waiving certain blueprint benchmarks is routinely used to widen execution and bit thickness to the disservice of yields.
- LVS checks whether the outline and the schematic are undefined or not. Affiliation showed in both outline and schematic must be same. Inputs for doing LVS may be GDS file or CDL file. The GDS file contains the configuration information for circuit. CDL file contains the schematic information required for circuit. Stream of LVS is showed in the below figure.

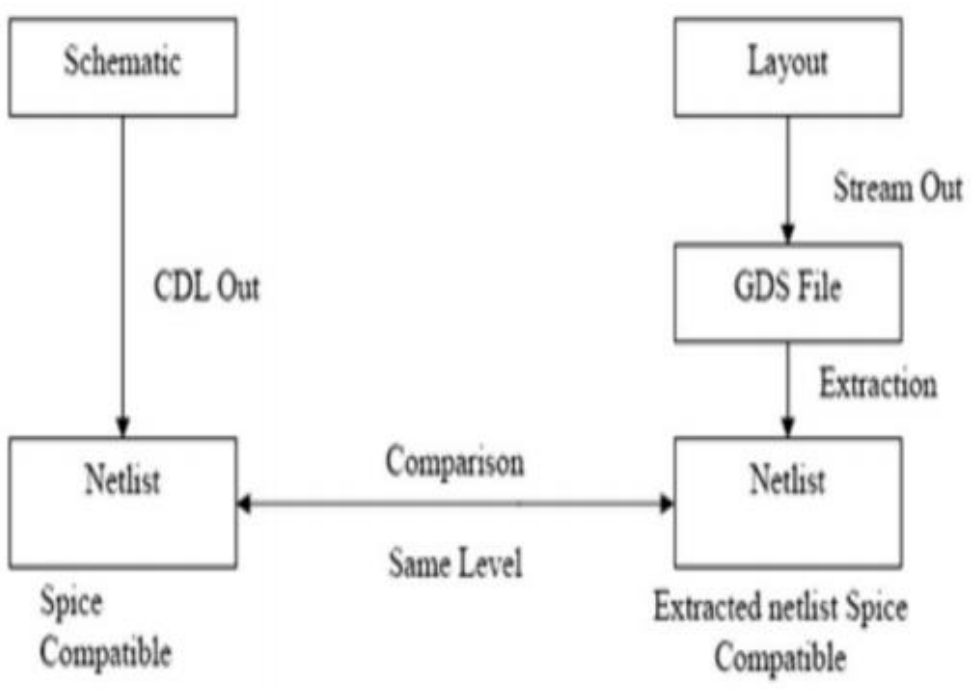


Figure 2.5: Flow of LVS

2.5 Introduction to BE Compiler Output

Output of Back End Memory compiler is a library containing various views. As discussed previously, Library is a collection of IPs and an IP contains various cells and each cell has views [6] which are the representation of the cell. All the cells have : Layout, Abstract, Schematic, Symbolic and Timing view. A cell is delivered as a set of all views and each view is utilized by different EDA tools in a given electronic design [7]. Main library views are explained below :

2.5.1 Schematic View

Schematic view gives a simple picture of an electronic circuit. These views represent the different components of the circuit as simple standard symbols, and various signal and power connections between the devices. It shows the representation of the cell at transistor level. It gives a schematic view of component instances, wires and pins.

Cirtcuit Description Language (CDL) or Netlist

In a SoC IP, a **netlist** is a description of the connectivity of the entire electronic circuit. A simple netlist consists of a systematic list of all the electronic components

in a circuit and also a list of the nodes that the electronic components are connected to. A network (net) is a collection that consists of two or more interconnected electronic components. Netlists may vary in terms of their structure, complexity and representation, but the fundamental purpose of every netlist is to give connectivity information of the whole electronic circuit. Netlists usually consist of instances, nodes, and some attributes of the components involved. If a Netlist contains more information than this, it is considered to be a hardware description language (such as Verilog or VHDL). Following are the possible categorizations of Netlist:

- *physical* or *logical*
- *instance-based* or *net-based*
- *flat* or *hierarchical*. The latter can be either folded or unfolded.

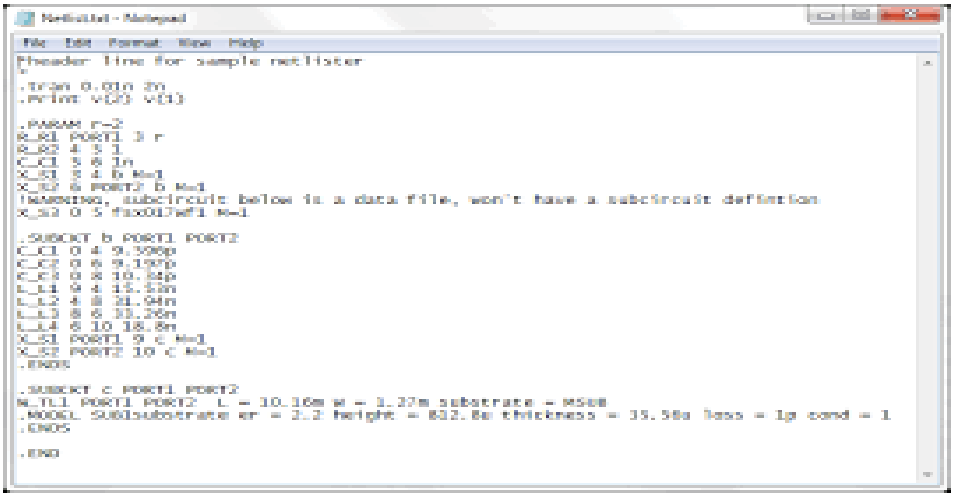


Figure 2.6: An example of netlist or cdl view file

2.5.2 Layout View

Layout view is the proper physical portrayal of a cell`s electronic circuits that goes on the silicon for fabrication. It is the representation an IC, showing its various geometric shapes which correspond to patterns of MOS (Metal oxide Semiconductor). Layout view must pass a sequence of checks during Verification process. These verification tests are done at a stage after the whole library generation package is available. In this verification process, most important checks are Layout vs Schematic (LVS) and Design Rule Check (DRC). The parameters for such checks are provided by chip manufacturers. Another name of Layout View mask design is IC mask layout.

Graphical Design System (GDS)

GDSII stream format, commonly referred to as **GDSII**, is a database file format which is the de facto industry standard for data exchange of integrated circuit or IC layout artwork. GDSII is a binary file format representing planar geometric shapes, text labels, and other information about the layout in hierarchical form. The data can be used to reconstruct all or part of the artwork to be used in sharing layouts, transferring artwork between different tools, or creating photomasks.

2.5.3 Abstract View

Abstract view, gives the information about signal and power pin layers present in the layout view. Abstract view also gives the information about the obstruction area (i.e. non-routing area) between the various metal layers used.

Library Extension Format (LEF)

LEF file in its simple form is ASCII portrayal of the Abstract view. LEF file comprises of data like Name of Cell, No. of I/O Pins in a Cell, Pin definitions, Pin directions, Cell size, Obstruction layer definition, Pin locations

2.6 Validation of Library Views

The correctness of IPs and libraries is straightforwardly identified not only with their reuse but also with reconciliation as well as the design effectiveness of SoC [10]. Therefore, a library should be validated properly at initial stages itself so that there is no problem at later stages and customer can be provided with the product delivery on time. For the same reason, Lin [11] discussed about the prerequisites of a high quality standard cell library, for instance, the appropriate functionality of a cell, its design in which there are no rule violations as compared to an ideal design and its precise timing of execution. Then, all the general errors were classified in five different by him. The categories are: inaccuracy, incompleteness, functional errors, inconsistency and design rule violation. According to him, the errors are very frequently possible during the library evolution flow. This is the reason why library validation stage is really needed for the development of a highly reliable and usable library. Helena Zheng highlights the main challenges that are faced in the process of successful IP integration [12] if validation is not done at initial stage, issue multiplies its effect at later stages. There is a difference between Verification, Validation and testing of SoC

designs [13]. Larry Cooke, explains the reason why we still do not have the quality of IP that we expect [14].

2.6.1 Important Validation Parameters

The validation process is to ensure that the library under validation is precise and complete in terms of its output views. Following points are considered in the validation flow:

A. Checking whether library is complete

a) Main part of validating a library is checking whether it is complete. It is checked if all the views are present in the library. For any number of cuts in the library, presence of all view files for each cut and also the presence of packed view files is checked.

b) Therefore, the completeness of a IP/Library can be ensured by checking whether all the required views are present on the specified paths.

B. Checking whether library is correct

a) Correctness of the views present in a Library is checked based on the characteristics of each library view as mentioned in the specifications provided [17].

b) Correctness of library is ensured by checking the vital characteristic values of a view. After assuring the existence of a library view, the characteristic values are cross verified to ensure that the IP/Library has been designed as per required rules.

C. Checking whether library views are compatible with various tools

a) The IP/library package must provide the chip designer with a complete list of views of library available inside the package to design a complete flow. While utilizing these views in a design flow there should be no compatibility problems with the EDA tools. This simply implies that the syntax and other parameters of the views i.e output of BE Compiler must be accurate.

b) To ensure the view compatibility, syntax of each view is checked by parsing the view with specific EDA tools. For instance, cdl view is parsed through cdl parser to ensure the correctness of view in terms of syntax and compatibility.

D. Checking whether library is consistent

a) During the validation process, the consistency among the various library views is ensured.

b) It is ensured that data is reliable between different views. For example, the cell names used in a particular view are same as the cell names present in another view. Cell names consistency between main and supporting views is checked.

2.7 Previous Validation Approach

Previously, library was validated manually based on specifications considering all the parameters [15]. Manual procedure was followed to validate each view. Some shell commands were used to check cell names and consistency between different views. Same steps were repeated for each validation flow. For each compiler, all the validation steps were repeated. However for validation of bitmap view, a tool was available which saved some time in validation of this view. All the other checks were manual which consumed a lot of time. Moreover, one may miss some points in manual validation which is a big risk at later stages.

Benefits of previous solution:

a) Previous solution was eyeball testing which involves human instinct and inductive thinking.

c) Previous validation approach is more dependable than the automated one (as in many situations it won't cover all the cases).

d) It requires less cost to start profitable validation [16] using this approach.

e) Computerized Testing can't change course in case one wants to analyze something that had not been already considered.

Problems with previous solution

a) Manual validation is very time consuming since repetition of tests is not easy. Validating a single library may take 4-5 days.

CHAPTER 3

PROPOSED SOLUTION : BE MEMORY COMPILER VALIDATION TOOL (BEMCVT)

Essential use of BE compiler is to generate Layout and netlist as per data specification. For different technology, particular BE compiler comes into picture. Previously, validation of BE Compiler was done by performing different checks which had to be done mostly manually. For multiple cuts, it is a very tedious task.

"BE Memory Compiler Validation Tool" is a validation tool that encapsulates all checks for an efficient and optimized BE Compiler Validation. It validates BE Compiler by running various checks on library generated using BE Compiler product. Certain products need to be sourced pre-hand which are required by this validation product. However, no extra effort is required to source the required products. When we source BE MEMORY COMPILER VALIDATION Product, all the required products are sourced automatically as user dependency is reduced to a possible extent.

Following includes the motivation for this project

- Manual checks took long time.
- Similar checks were needed for each cut of every compiler.
- Library level testing could be handled by automating this testing.

BE MEMORY COMPILER VALIDATION Tool contains a significant number of checks which are enough to validate a BE compiler by running these checks on output views of BE Compiler. There is a dynamic list maintained in the source code so that new checks may be added in future. There is provision for user to run only the checks that are required. The dynamic nature of this tool also allows to limit the output views that need to be validated. Test cases for all the checks were analysed and all checks have been duly tested.

The need for this validation product was realised and work on it was started immediately taking into account the benefits of such a validation tool. Project flow for building this validation product is as shown in following figure :

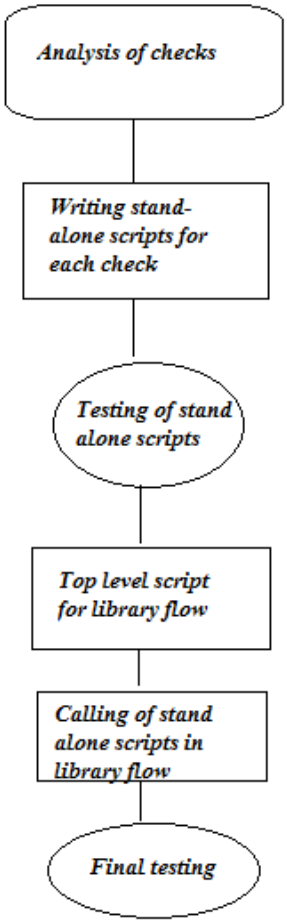


Figure 3.1: Project Flow Chart

3.1 Checks Supported In BEMCVT

Listed below are various checks currently supported by BE Compiler Validation Tool:

3.1.1 Uniquification

Uniquification refers to appending a certain string (called serial) with cell names in so that the cells can be uniquely identified. Uniquification is done in the compiler code itself. If skipped, it can create issues in future. If uniquification is not done in the code, when two different versions of same compiler are used on SOC (Sysytem On

Chip), there will be ambiguity, cell names being same cannot be uniquely identified. To avoid such ambiguity uniquification is done in compiler code itself.

So validating whether uniquification is performed is an important step. BE Compiler Validation tool does the needful in checking whether uniquification is being done. Gds and Cdl are separately checked for uniquification. It involves following two checks which have separate standalone execution capability :-

- **Gds uniquification check**
  This check tests if cell names contain cut name and end with a certain serial value.

  Check fails if atleast one cell is found that is not uniquified.

- **Cdl uniquification check**
  This check tests if cell names end with a certain serial value. Check fails if atleast one cell is found that is not uniquified.

3.1.2 Hcell Check

Hcell is one of the output files created after cut generation of a compiler. It contains common in a column wise manner common entries from gds and cdl. One entry is for cut name cut name.

Correctness of hcell file is checked in this check. Checks for gds and hcell are performed on the hcell file.

- **Gds hcell check**
  Gds entries in hcell are verified. It is checked whether cdl entries corresponding to each gds entry exist

- **Cdl hcell check**
  Cdl entries in hcell are verified. It is checked whether gds entries corresponding to each gds entry exist.

3.1.3 Check Port Order

It is a check for cdl where ports and their order is verified. All connections are checked. This check is explicitly for cdl i.e. netlist. Port order is checked only for top

level sub circuit. According to convention, [1] port should be in alphabetical order and in terms of indices, the order should be decreasing.

Example of correctly ordered ports:

A[7] A[6] A[5] A[4] A[3] A[2] A[1] A[0] gndm WL[4] WL[3] WL[2] WL[1] WL[0] vddm

3.1.4 Memcell Replacement

Memcell Replacement is done in compiler code itself. Belib product provided by memory team has reference to memcells from memcell product. After coding part, memcells in belib are replaced by memcells in memcell product. Reason for memcell replacement is to incorporate any changes in the memcell product without the need to update the belib product. So memcell replacement is an important step that has to be insured. This check has two different stand alone checks :-

- **Gds memcell replacement check**

  Local belib after updating (a check string is added) memcells is used to generate cut with only gds view. Memcells are checked for the updation made.If the check string is found, it indicates that memcell replacement is not done and mmcell replacement check fails in this case.

- **Cdl memcell replacement check**

  Local belib after updating (a check string is added) memcells is used to generate cut with only cdl view. Memcells are checked for the updation made.If the check string is found, it indicates that memcell replacement is not done and mmcell replacement check fails in this case.

3.1.5 pmake log check

After code completion, BE product is created from source code by compiling using pmake command. This command's log files are created after compilation separately for gds and cdl and one top level log file is maintained. This check reads these log files and checks for any error or warning. If any error or warning is present in any of the log files, message is displayed and the check fails. This check is not particular to a cut.

3.1.6 Bitmap check

Bitmap is a file that contains coordinate related information. It is a csv file that has a header containing various parameter names and following lines contain their values. This check validates the bitmap file by verifying the correctness of header, sorted order of values, negative check etc. Bitmap check is performed cut wise.

3.1.7 Tag check

Tag is like a label that represents various information like product name, version, vendor name etc. BE Compiler Code places top level tag. This tag check tests the validity of tag i.e whether the tag follows tag specifications, whether number of fields are correct, whether values are correct.

3.1.8 Cell id value check

Cell id refers to the name of cut, computed by BE compiler code. It contains reference to parameters like words, bits, mux, redundancy etc. Cell name is any random cut name provided by user during cut generation. It may or may not contain any reference to capacity of memory compiler. In bitmap generated after cut generation, cell name and cell id both are mentioned in the header. Also, top level tag for generated gds contains a tag field for cell id. Cell id should be same in bitmap header and top level tag. This check is to validate that the cell id value is correct in bitmap header as well as top level tag and contains reference to capacity of compiler.

3.1.9 Gds Cdl Diff:

This check is used in rare cases where some changes are made to optimize the code. To insure that the changes made do not have any unwanted impact on the compiler output, difference in output files i.e gds and cdl is checked. For comparison a reference compiler is taken to compare with the one in which changes are made. This check can also be used to compare outputs of previous submitted version of particular BE compiler and the which is ongoing.

Gds being a binary file, checking difference is very tedious job. Commands from another product (DK) are used to check the diff.

Three types of diff checks are supported:

- Abstract Diff
- Flat diff

- Hierarchical Diff

There are three possible scenarios, when this check is called in standalone mode :

- Checking diff after generating cut for both the compilers
- Checking diff after generating cut for one of the compilers while cut for other compiler is already generated and its cut library path is available
- Checking diff when cuts libraries for both the compilers are already generated

3.2 Importance of BEMCVT

BMCVT provides an efficient validation technique for a BE compiler. Any BE compiler can be validated using this tool. We can restrict the number of checks if we don't want to run all checks for the compiler. This tool saves a lot of time as validation is completely automated. We only need to provide the cut library generated using particular BE compiler as input, all checks are executed on this library and final reports are stored in a separate folder. We can repeat the validation process after each update since it is not very time taking.

A csv file called "Summary File" is also generated by this tool which displays the status of all checks executed on the library, path of log files corresponding to the checks and time taken by each check.

CHAPTER 4

DESIGN AND IMPLEMENTATION

As the entire implementation work was done at ST Microelectronics, so due to the company policies some of the commands and check names have been rephrased.

4.1 Prerequisite

4.1.1 Technical Requirements

Platform	Unix
RAM required	4 GB+
Scripting language used	Shell, tcl
Input formats supported	Binary, txt, csv (comma separated values)
Report files	Csv, txt

Table 4.1 Technical Requirements

4.2 Execution

To run the checks on whole cut library, following command can be used.

1) runChecks <cut library path>

   To use the gui mode, use -gui option.
   cmd1 <cut library path> -gui

Following are the commands for standalone checks:

1) gdsUniquification -gui

   In the gui window, provide path of gds and value of serial

2) cdlUniquification -gui

   In the gui window, provide path of csl file and value of serial

3) gdsHcell –gui

   In the gui window, provide path of gds file, path of hcell file and value of serial.

4) cdlHcell –gui

   In the gui window, provide path of cdl file, path of hcell file and value of serial.

5) bitmapCheck –gui

   In the gui window, provide path of gds file and bitmap file.

6) gdsMemcellReplacement –gui

   In the gui window, provide path of .ucdprod file, ucdprod file

7) cdlMemcellReplacement –gui

   In the gui window, provide path of .ucdprod file, ucdprod file

8) cellIdValueCheck –gui

   In the gui window, provide path of .ucdprod file, ucdprod file and default cut name

9) checkPmakelog –gui

   In the gui window, provide BE product path.

10) checkPortorder –gui

    In the gui window, provide path of cdl file.

4.3 Flow of BEMCVT

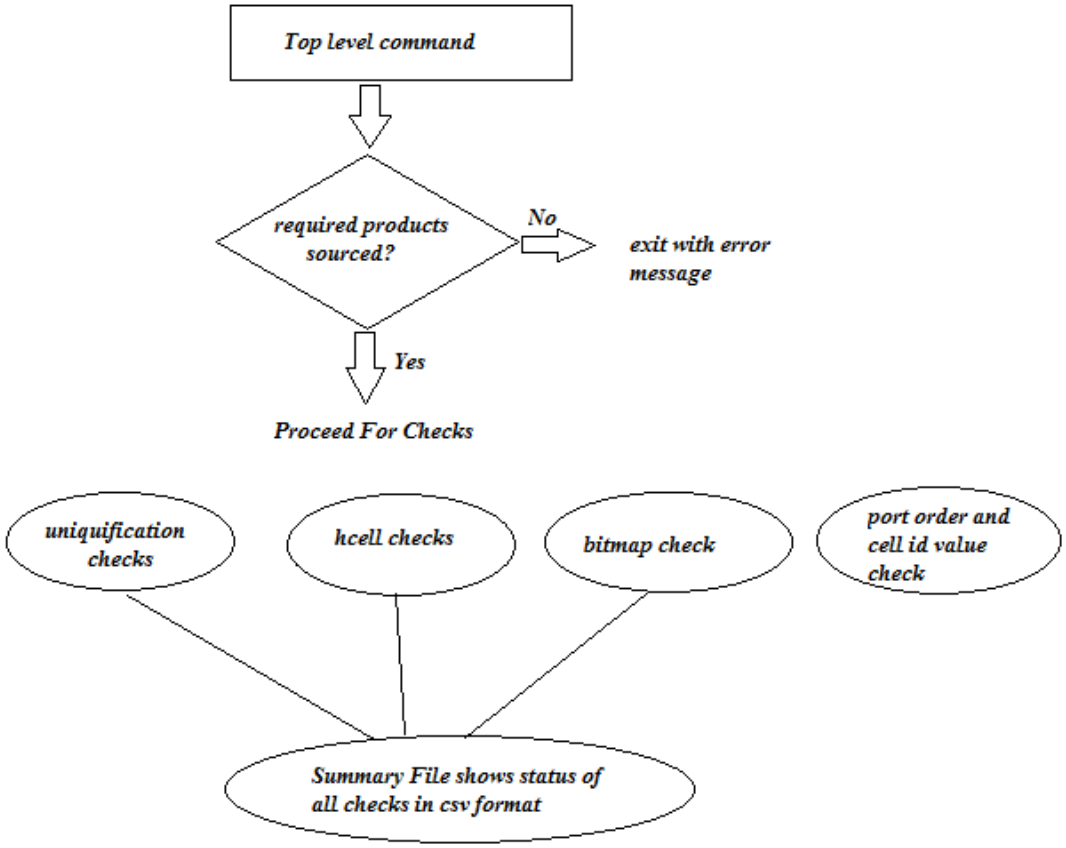


Figure 4.1: Project Checks Flow

4.4 Important Features of BEMCVT

Following are the main features of this validation tool:

4.4.1 Modular Code

Common methods or procedures are created in a separate script, all other checks make use of these procedures. Thus there is not much repetition of code. Similar code is handles in such common procedures. Hard coding is avoided.

4.4.2 Full Fledged Library Level Check

Executing a top level script allows testing of whole library containing single or multiple cuts. We can run checks on whole library without the need to run a check

again and again for each cut. A summary file is maintained that displays the status of each check and path of its log file.

4.4.3 Log Directory

A well structured log directory is created after each library level check execution. Check wise directory structure contains log files for each check and each cut. Log file can be analysed for full check details and errors can thus be removed.

4.4.4 Summary File

In log directory a csv file is created that contains all check names that are executed, their status whether passed or failed and log file path. This file displays whole result in a readable form and saves much time. After checking summary file, we can open any log file for further details.

4.4.5 Debug Mode

When certain environment variable DEBUG holds a value (i.e. variable is set), detailed error information or actual cell level information is displayed. This creates long log files but is of much use in case of identifying an error.

4.4.6 Dry run Provision

When we want to run checks only for single cut rather than a large number of cuts in the library, we can set environment variable DRYRUN. All checks will run only for single cut and final report will be generated. This saves time when there is no need to repeat the checks for all cuts present in the library.

4.4.6 Log/Report directory Structure

Final log files for all checks, resultant reports and Summary file is dumped in a directory called “Results” in present working directory (pwd).

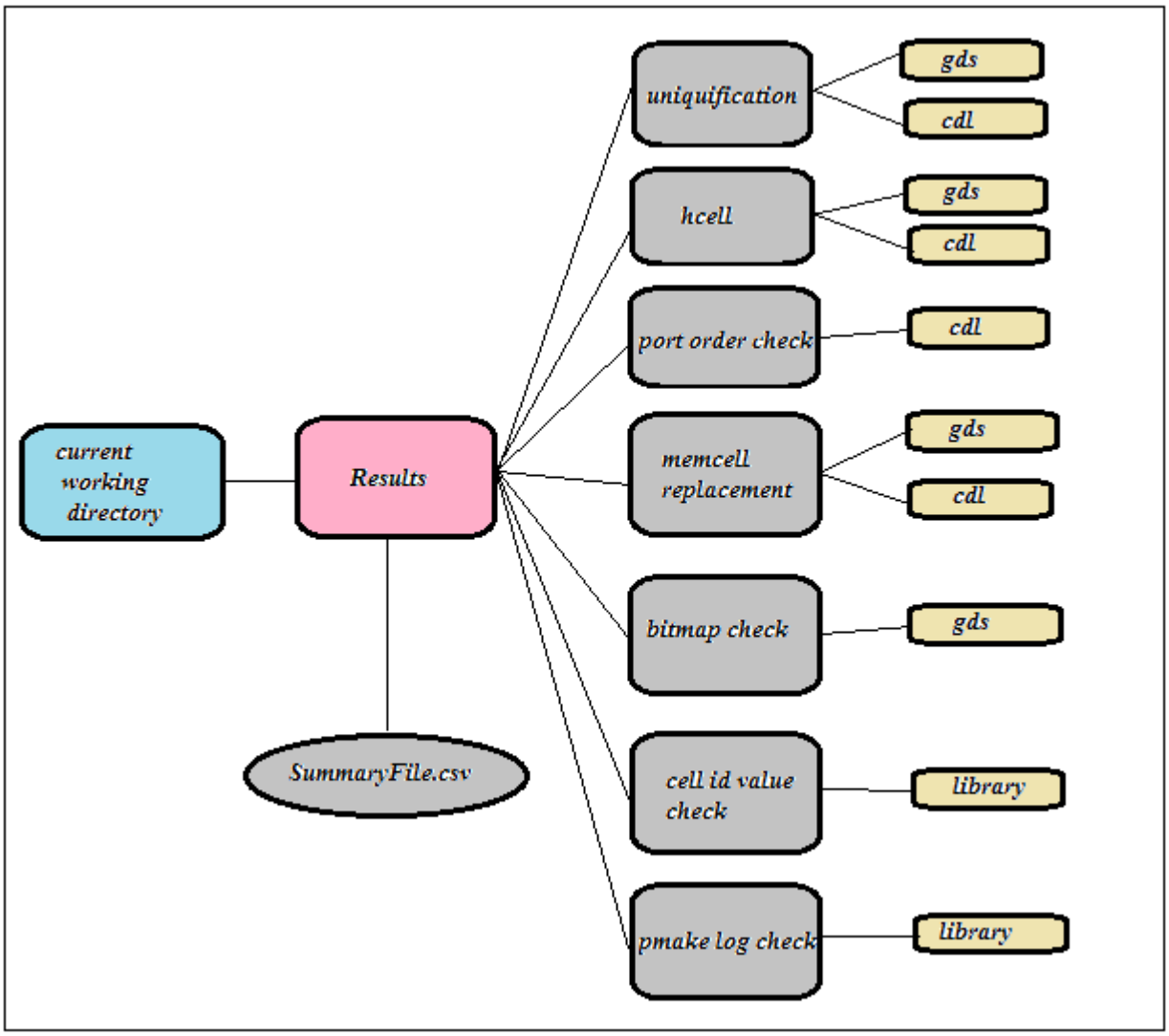


Figure 4.2: Reporting directory structure

CHAPTER 5

RESULTS AND ANALYSIS

5.1 Functional Analysis

S. No.	Evaluation Parameter	Previously	Using BEMCVT
1	Validation checks	All Manual except bitmap check	Automated
2	Repetition of checks	Easy	Very Difficult
3	Reports of checks	No reports	Dedicated reports
4	Granularity of validation (per-cell or per-view)	Low	High
5	Collaborative Reporting of all checks	None	Collaborative report in csv file

Table 5.1: Functional Analysis

5.2 Timing Analysis

Manual checks required very long time. If number of cuts is large, it is difficult to keep track in manual validation. To validate a cut library with large number of cuts, requires so many days. Manual validation may sometimes lead to delayed deliveries of the BE product.

Time analysis was done for a library containing 18 checks. Following table shows the time analysis for various checks on this library:

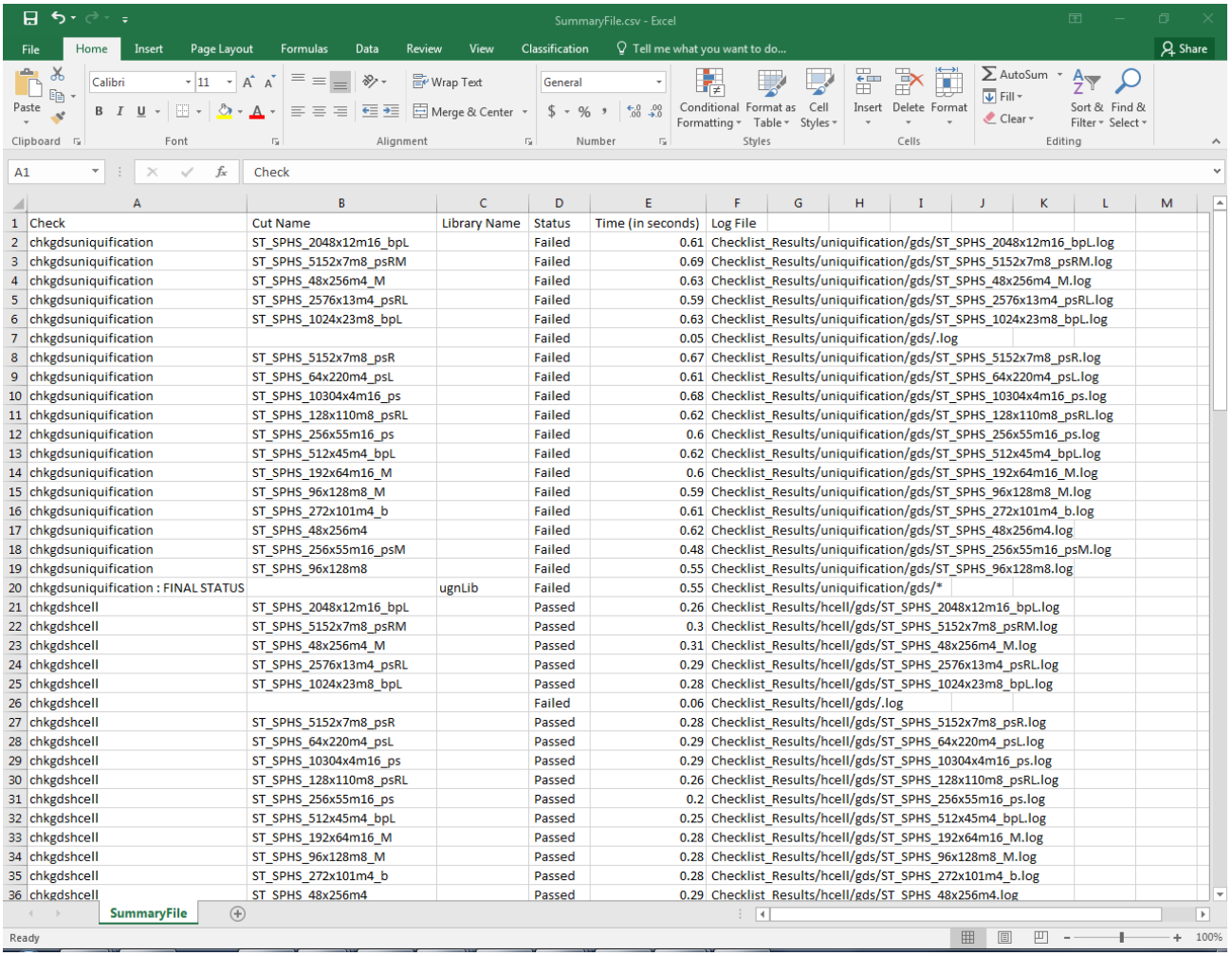
Check Name	Previous (Manual) Time	Time using BEMCVT
Uniquification Check	47min	0.54s per cut
Hcell Check	30min	0.29s per cut
Port Order Check	35min	0.19s per cut
Bitmap Check	2.15s (using another tool)	1.38s per cut
Memcell Replacement gds	1.49 hour	93.2s (18 cuts)

Memcell Replacement cdl	1.38 hour	90.4s (18 cuts)
Cell Id Value check	1.21 hour	14.34s
Pmake log check	1.76s	0.4s
Gds Cdl Diff	Many hours	1.03 hours

Table 5.2: Timing Analysis

5.3 Results

1. After execution, summary file is generated. It is a csv file that contains check name, cut name, status of check, time taken, log file path.
2. This csv (comma separated value) file also shows final status of each check executed on more than one cut.



SummaryFile.csv - Excel

	A	B	C	D	E	F
1	Check	Cut Name	Library Name	Status	Time (in seconds)	Log File
2	chkgdsuniquification	ST_SPHS_2048x12m16_bpL		Failed	0.61	Checklist_Results/uniquification/gds/ST_SPHS_2048x12m16_bpL.log
3	chkgdsuniquification	ST_SPHS_5152x7m8_psRM		Failed	0.69	Checklist_Results/uniquification/gds/ST_SPHS_5152x7m8_psRM.log
4	chkgdsuniquification	ST_SPHS_48x256m4_M		Failed	0.63	Checklist_Results/uniquification/gds/ST_SPHS_48x256m4_M.log
5	chkgdsuniquification	ST_SPHS_2576x13m4_psRL		Failed	0.59	Checklist_Results/uniquification/gds/ST_SPHS_2576x13m4_psRL.log
6	chkgdsuniquification	ST_SPHS_1024x23m8_bpL		Failed	0.63	Checklist_Results/uniquification/gds/ST_SPHS_1024x23m8_bpL.log
7	chkgdsuniquification			Failed	0.05	Checklist_Results/uniquification/gds/.log
8	chkgdsuniquification	ST_SPHS_5152x7m8_psR		Failed	0.67	Checklist_Results/uniquification/gds/ST_SPHS_5152x7m8_psR.log
9	chkgdsuniquification	ST_SPHS_64x220m4_psL		Failed	0.61	Checklist_Results/uniquification/gds/ST_SPHS_64x220m4_psL.log
10	chkgdsuniquification	ST_SPHS_10304x4m16_ps		Failed	0.68	Checklist_Results/uniquification/gds/ST_SPHS_10304x4m16_ps.log
11	chkgdsuniquification	ST_SPHS_128x110m8_psRL		Failed	0.62	Checklist_Results/uniquification/gds/ST_SPHS_128x110m8_psRL.log
12	chkgdsuniquification	ST_SPHS_256x55m16_ps		Failed	0.6	Checklist_Results/uniquification/gds/ST_SPHS_256x55m16_ps.log
13	chkgdsuniquification	ST_SPHS_512x45m4_bpL		Failed	0.62	Checklist_Results/uniquification/gds/ST_SPHS_512x45m4_bpL.log
14	chkgdsuniquification	ST_SPHS_192x64m16_M		Failed	0.6	Checklist_Results/uniquification/gds/ST_SPHS_192x64m16_M.log
15	chkgdsuniquification	ST_SPHS_96x128m8_M		Failed	0.59	Checklist_Results/uniquification/gds/ST_SPHS_96x128m8_M.log
16	chkgdsuniquification	ST_SPHS_272x101m4_b		Failed	0.61	Checklist_Results/uniquification/gds/ST_SPHS_272x101m4_b.log
17	chkgdsuniquification	ST_SPHS_48x256m4		Failed	0.62	Checklist_Results/uniquification/gds/ST_SPHS_48x256m4.log
18	chkgdsuniquification	ST_SPHS_256x55m16_psM		Failed	0.48	Checklist_Results/uniquification/gds/ST_SPHS_256x55m16_psM.log
19	chkgdsuniquification	ST_SPHS_96x128m8		Failed	0.55	Checklist_Results/uniquification/gds/ST_SPHS_96x128m8.log
20	chkgdsuniquification : FINAL STATUS		ugnLib	Failed	0.55	Checklist_Results/uniquification/gds/*
21	chkgdshcell	ST_SPHS_2048x12m16_bpL		Passed	0.26	Checklist_Results/hcell/gds/ST_SPHS_2048x12m16_bpL.log
22	chkgdshcell	ST_SPHS_5152x7m8_psRM		Passed	0.3	Checklist_Results/hcell/gds/ST_SPHS_5152x7m8_psRM.log
23	chkgdshcell	ST_SPHS_48x256m4_M		Passed	0.31	Checklist_Results/hcell/gds/ST_SPHS_48x256m4_M.log
24	chkgdshcell	ST_SPHS_2576x13m4_psRL		Passed	0.29	Checklist_Results/hcell/gds/ST_SPHS_2576x13m4_psRL.log
25	chkgdshcell	ST_SPHS_1024x23m8_bpL		Passed	0.28	Checklist_Results/hcell/gds/ST_SPHS_1024x23m8_bpL.log
26	chkgdshcell			Failed	0.06	Checklist_Results/hcell/gds/.log
27	chkgdshcell	ST_SPHS_5152x7m8_psR		Passed	0.28	Checklist_Results/hcell/gds/ST_SPHS_5152x7m8_psR.log
28	chkgdshcell	ST_SPHS_64x220m4_psL		Passed	0.29	Checklist_Results/hcell/gds/ST_SPHS_64x220m4_psL.log
29	chkgdshcell	ST_SPHS_10304x4m16_ps		Passed	0.29	Checklist_Results/hcell/gds/ST_SPHS_10304x4m16_ps.log
30	chkgdshcell	ST_SPHS_128x110m8_psRL		Passed	0.26	Checklist_Results/hcell/gds/ST_SPHS_128x110m8_psRL.log
31	chkgdshcell	ST_SPHS_256x55m16_ps		Passed	0.2	Checklist_Results/hcell/gds/ST_SPHS_256x55m16_ps.log
32	chkgdshcell	ST_SPHS_512x45m4_bpL		Passed	0.25	Checklist_Results/hcell/gds/ST_SPHS_512x45m4_bpL.log
33	chkgdshcell	ST_SPHS_192x64m16_M		Passed	0.28	Checklist_Results/hcell/gds/ST_SPHS_192x64m16_M.log
34	chkgdshcell	ST_SPHS_96x128m8_M		Passed	0.28	Checklist_Results/hcell/gds/ST_SPHS_96x128m8_M.log
35	chkgdshcell	ST_SPHS_272x101m4_b		Passed	0.28	Checklist_Results/hcell/gds/ST_SPHS_272x101m4_b.log
36	chkgdshcell	ST_SPHS_48x256m4		Passed	0.29	Checklist_Results/hcell/gds/ST_SPHS_48x256m4.log

Figure 5.1: Summary File

3. Report directory for bitmap also contains an html report file that shows the final status of all bitmap related checks.

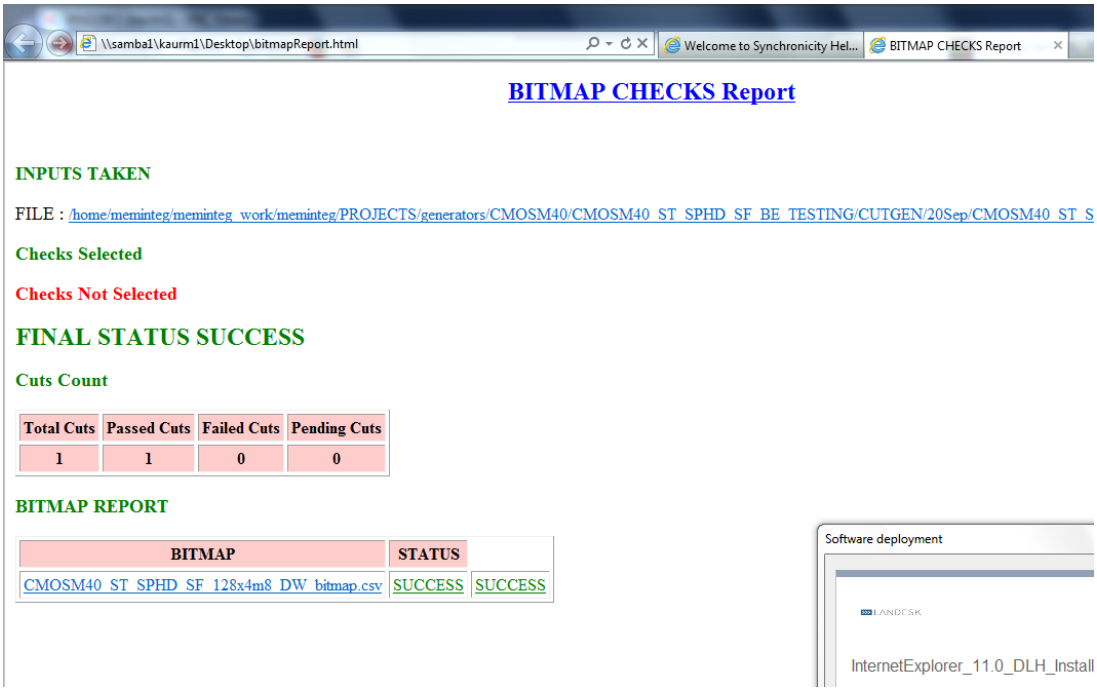


Figure 5.2: Bitmap check report

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

Back End Memory Compiler Validation Tool is a tool that validates BE Compiler's output thus validating the compiler code. This tool helps to detect any kind of issues or errors at initial stage of IP development. If failure is found at SOC design level, it will involve much effort, time and cost to overcome the failure. Thus, this tool reduces the cost and effort to much extent. It runs significant amount of checks on the cut library and maintains reports for each check.

Earlier the validation was done manually which took lots of time. Manual validation, there are always chances of missing something which involves risk at later stages. BE Memory Compiler Validation Tool maintains reports for each executed check. These reports are stored according to a well defined directory structure which is easy to locate any check's log file. To validate whole library having very large number of cuts, we can simply run the top level command of this tool and rest is automatic, we can continue with our work while the checks are executed on the cut library.

6.2 Future Scope

Currently, this tool supports 10 checks. More checks can be added in the future. For instance, new LG checks can be integrated in the tool. Few LG checks like delete label, pin promotion checking etc. are already in the list of to be integrated checks. These new checks will then be included in library level flow in top level script.

References

[1] K. Shuler, "What does it cost you when your SoC is late to market?" 2014. [Online]. Available: http://www.arteris.com/blog/bid/112221/What-Does-It-Cost-You-When-Your-SoC-is-Late-to-Market

[2] Jean-Pierre HELIOT, “LYS: A Solution for System on Chip (SoC) Production Cost and Time to Volume Reduction” in Proceedings of the Fourth International Symposium on Quality Electronic Design (ISQED’03)

[3] Ender Yılmaz, Günhan Dündar, “New Layout Generator For Analog CMOS Circuits”, 2007, IEEE.

[4] Ender Yilmaz, “Analog Layout Generator for CMOS Circuits” IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 28, NO. 1, JANUARY 2009

[5] Zhao-Yong Zhang, Chia-Cheng Chen * and Jian-Bin Zheng, “A 90-nm CMOS Embedded Low Power SRAM Compiler” , 2009 IEEE

[6] Antun Domic, Samuel Levitin, Nathan Philips, Channeary Thai, Tom Shiple, Dilip Bhavasar and Clint Bisell, “CLEO: A CMOS Layout Generator”, 1989, IEEE

[7] STMicroElectronics Internal Document , Library Views

[8] STMicroelectronics, “St documents on cut generation,” pp. 1 – 17, October 2015.

[9] STMicroelectronics, “St internal documents on memories and training manuals,” pp. 314 – 317, October, 2015.

[10] L. W. Wang and H. W. Luo, "Automated IP quality qualification for efficient system-on-chip design," in *International Conference on Electronic Packaging Technology & High Density Packaging*, 2012.

[11] R. B. Lin, I. H. Chou and C. M. Tsai, "Benchmark circuits improve the quality of a standard cell library," in *Asia and South Pacific Design Automation Conference*, 1999.

[12] Design And Reuse. (2017). IP Verification : What are the main challenges to successful IP integration?. [online] Available at: https://www.design-reuse.com/articles/3249/ip-verification-what-are-the-main-challenges-to- successful-ip-integration.html

[13] AnySilicon. (2017). Verification, Validation, Testing of ASIC/SOC designs - What are the differences? - AnySilicon. [online] Available at: http://anysilicon.com/verification-validation-testing-asicsoc-designs-differences/

[14] EETimes. (2017). Why We Don't Have IP Quality Yet | EE Times. [online] Available http://www.eetimes.com/author.asp?section_id=36&doc_id=1265541

[15] H. J. Brand, S. Rulke and M. Radetzki, "IPQ: IP qualification for efficient system design," *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720)*, 2004, pp. 478-482.

[16] A. Vorg, M. Radetzki and W. Rosenstiel, "Measurement of IP qualification costs and benefits," *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, 2004, pp. 996-1001 Vol.2.

[17] R. Seepold, N. Martínez Madrid, A. Vorg, W. Rosenstiel, M. Radetzki, P. Neumann and J. Haase, "A Qualification Platform for Design Reuse", Proceedings of Design, Automation and Test in Europe Conference and Exhibition 2012