Synthesis And APR Of Data-Path Module In Sensor Hub On 32nm/22nm Technology

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF TECHNOLOGY
IN
ELECTRONICS & COMMUNICATION ENGG.
(VLSI DESIGN)

UNDER THE GUIDANCE OF
Mr. Purav K Bhatt

Submitted by
Khamar Tapaswi J.(10MECV24)

Department of Electronics & Communication Engineering
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY,
AHMEDABAD 382481

Synthesis And APR Of Data-Path Module In Sensor Hub On 32nm/22nm Technology

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF TECHNOLOGY
IN
ELECTRONICS & COMMUNICATION ENGG.
(VLSI DESIGN)

UNDER THE GUIDANCE OF
Mr. Purav K Bhatt

Submitted by
Khamar Tapaswi J.(10MECV24)

Department of Electronics & Communication Engineering
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY,
AHMEDABAD 382481

CERTIFICATE

This is to certify that the M.Tech Dissertation report entitled "***Synthesis And APR Of Data-Path Module Of Sensor Hub On 32nm/22nm technology***" submitted by **Khamar Tapaswi J. (Roll no 10MECV24)** towards the partial fulfillment of the requirements for Master Of Technology (Electronics and Communication Engineering) in the field of VLSI Design of **Institute Of Technology,Nirma University, Ahmedabad** at **Intel Corporation, Bangalore** is the record of the work carried out by him under our supervision and guidance. The work submitted has in our opinion reached a level required for being accepted for examination. The results embodied in this Dissertation Project Work to the best of our knowledge have not been submitted to any other University or Institute for award of any degree or diploma.

Date:

Project Guide
Mr. Purav Bhatt
Component Design Engineer,
Intel Corporation,
Bangalore, India.

Internal Project Guide
Dr. N.M. Devashrayee
Institute of Technology,
Nirma University,
Ahmedabad, India.

Project Manager
Mr. Srinivas Lingam
Engineering Manager,
Intel Corporation,
Bangalore, India.

Ashok S. Ranade
EE Department
Institute of Technology,
Nirma University,
Ahmedabad, India.

ACKNOWLEDGEMENT

First and foremost, I would like to express my hearty thanks and indebtedness to my guide ***Mr.Purav Bhatt*** for his enormous help and encouragement throughout the course of this thesis**,** who happens to be my role model, has always given me a real example of how a researcher should be.

I express my soulful gratitude to ***Mr.Sumeet Aggarwal*** for their invaluable suggestion for my training. I specially want to acknowledge ***Intel Corporation*** who gave me opportunity to do my thesis work and provide all resources required for my project work. I also want to acknowledge Mrs Usha Mehta, Prof. B.G. Upadhyay, Prof. H.A.Patel, Mr. Panayil Samuel, Mr. Nagendra Gajjar, Mrs. Krupaxi Patel, Mr. Anuj Kumar Mevada, Mr Maulin Sheth, Prof. T Ram, Mr S. Perumal and Dr. C.G. Patel without whom this project work could not have seen the daylight and helping me with his constant involvement during my project tenure directly or indirectly.

I would like to express my sincere gratitude to ***Dr. N.M. Devashrayee (Program Co-Ordinator***, MTech VLSI design, Nirma University, Ahmedabad*)* for his continuous guidance, support and enthusiasm.

I acknowledge gratefully the help and suggestion of colleagues at Intel Corporation, friends mentors and each of them who in spite of their busy schedule and huge workload, were always eager to help me with their warm attitude and technical knowledge.

Finally, Tons of word won’t be enougsh to express my deepest reverence to my family without whom I wouldn’t have been able to reach at this position.

Khamar Tapaswi J.
10MECV24

Abstract

➔ The thesis produced here is the outcome of performing RTL to GDS operation of the industry's most talked digital circuit, i.e. Sensors. The thesis revolves around the entire SoC design cycle. It explores the entire design cycle step by step starting from RTL coding to the Sign-Off checks. The digital circuit presented here is essentially a datapath circuit which is responsible for taking in the data from general purpose Inputs and giving out the data to Outputs.

➔ The most important challenge faced by the industry is Power consumption. The thesis also explores the optimization strategies that can be used to optimize the Power. Together with the Power, TTM(Time To Market) is also an important aspect. The work done also focusses on the optimizing the digital circuit so that the tool is having only the required data to handle and thus the time of execution can be reduced.

➔ The Engineering Change Order script is developed to perform the ECO operation which can be used by any tool. The script is a tcl utility which utilizes the result of performing FEV, takes in the original netlist and the modified RTL. The script automatically changes the netlist as per the modified RTL.

Contents

Sr. No.	Chapter Name	Page No.
1.	Introduction & Background Of SoC Design	1
2.	Synthesis Using Design Compiler Basics	10
3.	Optimization Strategies	18
4.	Synthesis Outputs	24
5.	Automatic Place And Route	27
6.	Results Of Sign-Off Checks	35
7.	Conclusion	39
8.	Bibliography	41

List of figures

Sr. No.	Figure Description
Figure 1	Design Implementation Flow
Figure 2	Design Compiler Implementation Flow
Figure 3	Load/Driver Constraint Example
Figure 4	Timing Goal Example
Figure 5	Boundary Optimization
Figure 6	Automatic Place And Route Flow
Figure 7	Timing Sign Off Results
Figure 8	Power Sign Off Results
Figure 9	Functional Sign Off
Figure 10	Layout Versus Schematic Sign Off
Figure 11	Electrical Rule Check Sign Off

List Of Abbreviations

1. **SoC** – System On Chip
2. **DC** – Design Compiler
3. **ICC –** Integrated Circuit Compiler
4. **CTS** – Clock Tree Synthesis
5. **PNR** – Placement And Routing
6. **APR –** Automatic Place & Route
7. **AU-BO** – Automatic Ungrouping And Boundary Optimization
8. **LVS** – Layout Versus Schematic
9. **ERC** – Electrical Rule Check
10. **LEC** – Logic Equivalence Check
11. **FEV** – Functional/Formal Equivalence Verification
12. **PD –** Physical Design
13. **SD –** Structural Design
14. **ECO** – Engineering Change Order
15. **SDC** – Synopsys Design Constraints
16. **TEs** – Timing Exceptions

CHAPTER 1 :

Introduction & Background of SoC Design

1.1 Introduction

As deep sub-micron semiconductor geometries shrink, traditional methods of chip design have become increasingly difficult. In addition, an increasing numbers of transistors are being packed into the same die-size, making validation of the design extremely hard, if not impossible. Furthermore, under critical “time-to-market” pressure the chip design cycle has remained the same, or is constantly being reduced. To counteract these problems, new methods and tools have evolved to facilitate the SOC design methodology.

The main function of this chapter is to bring to the forefront different stages involved in chip design as we move deeper into the sub-micron realm. Various techniques that improve the design flow are also discussed. Since the last edition of this book, Synopsys introduced another tool called Physical Compiler. In the tool, synthesis and placement now are more tightly coupled. Consequently, there is a dramatic change in the traditional design flow. This chapter stresses the importance of the new techniques to the reader, and explains the necessity of these techniques in the design flow to achieve the maximum benefit, by reducing the overall cycle time. Since the tool is fairly new to the IC design world, and as yet, not embraced 100% by the SOC design community, both the traditional and the new flows are discussed.

This chapter focuses on the entire synthesis based SOC design flow methodology, from RTL coding to the final tape-out. Both the traditional and the Physical Compiler based flow are discussed.

1.1.1 Traditional Design Flow

The traditional SOC design flow contains the steps outlined below. Flow chart relating to the design flow described below. Subsequent chapters describe in detail synthesis related topics.

1. Architectural and electrical specification.
2. RTL coding in HDL.
3. DFT memory BIST insertion, for designs containing memory elements.
4. Exhaustive dynamic simulation of the design, in order to verify the functionality of the design.
5. Design environment setting. This includes the technology library to be used, along with other environmental attributes.
6. Constraining and synthesizing the design with scan insertion (and optional JTAG) using Design Compiler.
7. Block level static timing analysis, using Design Compiler’s built-in static timing analysis engine.

8. Formal verification of the design. RTL compared against the synthesized netlist, using Formality.
9. Pre-layout static timing analysis on the full design through PrimeTime.
10. Forward annotation of timing constraints to the layout tool.
11. Initial floorplanning with timing driven placement of cells, clock tree insertion and global routing
12. Transfer of clock tree to the original design (netlist) residing in Design Compiler.
13. In-place optimization of the design in Design Compiler.
14. Formal verification between the synthesized netlist and clock tree inserted netlist, using Formality.
15. Extraction of estimated timing delays from the layout after the global routing step.
16. Back annotation of estimated timing data from the global routed design, to PrimeTime.
17. Static timing analysis in PrimeTime, using the estimated delays extracted after performing global route.
18. Detailed routing of the design.
19. Extraction of real timing delays from the detailed routed design.
20. Back annotation of the real extracted timing data to PrimeTime.
21. Post-layout static timing analysis using PrimeTime.
22. Functional gate-level simulation of the design with post-layout timing (if desired).
23. Tape out after LVS and DRC verification.

The acronyms STA and CT represent static timing analysis and clock tree respectively. DC represents Design Compiler.

1.1.1 Specification and RTL Coding

Chip design commences with the conception of an idea dictated by the market. These ideas are then translated into architectural and electrical specifications. The architectural specifications define the functionality and partitioning of the chip into several manageable blocks, while the electrical specifications define the relationship between the blocks in terms of timing information. The next phase involves the implementation of these specifications. In the past this was achieved by manually drawing the schematics, utilizing the components found in a cell library. This process was time consuming and was impractical for design reuse. To overcome this problem, hardware description languages (HDL) were developed. As the name suggests, the functionality of the design is coded using the HDL. There are two main HDLs in use today, Verilog and VHDL. Both languages perform the same function, each having their own advantages and disadvantages. There are three levels of abstraction that may be used to represent the design; Behavioral, RTL (Register Transfer Level) and Structural. The Behavioral level code is at a higher level of abstraction. It is used primarily for translating the architectural

specification, to a code that can be simulated. Behavioral coding is initially performed to explore the authenticity and feasibility of the chosen implementation for the design. Conversely, the RTL coding actually describes and infers the structural components and their connections. This type of coding is used to describe the functionality of the design and is synthesizable to form a structural netlist. This netlist comprises of the components from a target library and their respective connections; very similar to the schematic based approach. The design is coded using the RTL style, in either Verilog or VHDL, or both. It can also be partitioned if necessary, into a number of smaller blocks to form a hierarchy, with a top-level block connecting all lower level blocks. Synopsys recently introduced Behavior Compiler, capable of synthesizing Behavior level style of coding. Since this is a major topic of discussion and is not relevant to this book, only RTL related synthesis is covered in this book.

1.1.2 Dynamic Simulation

The next step is to check the functionality of the design by simulating the RTL code. All currently available simulators are capable of simulating the behavior level as well as RTL level coding styles. In addition, they are also used to simulate the mapped gate-level design. The test bench is normally written in behavior HDL while the actual design is coded in RTL. Usually the simulators are language dependent (either Verilog or VHDL), although there are a few simulators in the market, capable of simulating a mixed HDL design. The purpose of the test bench is to provide necessary stimuli to the design. It is important to note that the coverage of the design is totally dependent on the number of tests performed and the quality of the test bench. This is the reason why a sound test bench is extremely critical to the design. During the simulation of the RTL, the component (or gate) timing is not considered. Therefore, to minimize the difference between the RTL simulation and the synthesized gate-level simulation at a later stage, the delays are usually coded within the RTL source, usually for sequential elements.

1.1.3 Constraints, Synthesis and Scan Insertion

For a long time, the HDLs were used for logic verification. Designers would manually translate the HDL into schematics and draw the interconnections between the components to produce a gate-level netlist. With the advent of synthesis tools, this manual task has been rendered obsolete. The tool has taken over and performs the task of reducing the RTL to the gate-level netlist. This process is termed as synthesis. Synopsys's Design Compiler (from now on termed as, DC) is the de-facto standard and by far the most popular synthesis tool in the SOC industry today. Synthesizing a design is an iterative process and begins with defining timing constraints for each block of the design. These timing constraints define the relationship of each signal with respect to the clock input for a particular block. In addition to the constraints, a file defining the synthesis environment is also needed. The environment file specifies the technology cell libraries

and other relevant information that DC uses during synthesis. DC reads the RTL code of the design and using the timing constraints, synthesizes the code to structural level, thereby producing a mapped gate level netlist. Usually, for small blocks of a design, DCs internal static timing analysis is used for reporting the timing information of the synthesized design. DC tries to optimize the design to meet the specified timing constraints. Further steps may be necessary if timing requirements are not met. Most designs today, incorporate design-for-test (DFT) logic to test their functionality, after the chip is fabricated. The DFT consists of logic and memory BIST (built-in-self-test), scan logic and Boundary Scan logic (JTAG) etc. The logic and memory BIST comprises of synthesizable RTL that is based upon controller logic and is incorporated in the design before synthesis. There are tools available in the market that may be used to generate the BIST controller and surrounding logic. Unfortunately, Synopsys does not provide this capability. The scan insertion may be performed using the test ready compile feature of DC. This procedure maps the RTL directly to scan-flops, before linking them in a scan-chain. An advantage of using this feature is its ability to enable DC to take the scan-flop timing into account while synthesizing. This technique is important since the scan-flops generally have different delays associated with them as compared to their non-scan equivalent flops (or normal flops). JTAG or boundary scan is primarily used for testing the board connections, without unplugging the chip from the board. The JTAG controller and surrounding logic may also be generated directly by DC.

1.1.4 Formal Verification

The concept of formal verification is fairly new to the SOC design community. Formal verification techniques perform validation of a design using mathematical methods without the need for technological considerations, such as timing and physical effects. They check for logical functions of a design by comparing it against the reference design. A number of EDA tool vendors have developed the formal verification tools. However, only recently, Synopsys also introduced to the market its own formal verification tool called Formality. The main difference between formal methods and dynamic simulation is that former technique verifies the design by proving that the structure and functionality of two designs are logically equivalent. Dynamic simulation methods can only probe certain paths of the design that are sensitized, thus may not catch a problem present elsewhere. In addition, formal methods consume negligible amount of time as compared to dynamic simulation. The purpose of the formal verification in the design flow is to validate the RTL against RTL, gate-level netlist against the RTL code, or the comparison between gate-level to gate-level netlists. The RTL to RTL verification is used to validate the new RTL against the old functionally correct RTL. This is usually performed for designs that are subject to frequent changes in order to accommodate additional features. When these features are added to the source RTL, there is always a risk of breaking the old functionally correct feature. To prevent this, formal verification may be performed between the old RTL and the new RTL to check the validity of the old functionality. The RTL to gate-level verification is

used to ascertain that the logic has been synthesized accurately by DC. Since the RTL is dynamically simulated to be functionally correct, the formal verification of the design between the RTL and the scan inserted gate-level netlist assures us that the gate-level also has the same functionality. In this instance if we were to use the dynamic simulation method to verify the gate-level, it would have taken a long time (days and weeks, depending on the size of the design) to verify the design. In comparison, the formal method would take a few hours to perform a similar verification. The last part involves verifying the gate-level netlist against the gate-level netlist. This too is a significant step for the verification process, since it is mainly used to verify what has gone into the layout versus what has come out of the layout. What comes out of the layout is obviously the clock tree inserted netlist (flat or hierarchical). This means that the original netlist that goes into the layout tool is modified. The formal technique is used to verify the logic equivalency of the modified netlist against the original netlist.

1.1.5 Static Timing Analysis using PrimeTime

As previously mentioned, the block level static timing analysis is done using DC. Although, the chip-level static timing can be performed using the above approach, it is recommended that PrimeTime, be used instead. PrimeTime is the Synopsys stand-alone sign-off quality static timing analysis tool that is capable of performing extremely fast static timing analysis on full chip-level designs. It provides a Tcl interface that provides a powerful environment for analysis and debugging of designs. The static timing analysis, to some extent, is the most important step in the whole SOC design process. This analysis allows the user to exhaustively analyze all critical paths of the design and express it in an orderly report. Furthermore, the report can also contain other debugging information like the fanout or capacitive loading of each net. The static timing is performed both for the pre and post-layout gate-level netlist. In the pre-layout mode, PrimeTime uses the wire load models specified in the library to estimate the net delays. During this, the same timing constraints that were fed to DC previously are also fed to PrimeTime, specifying the relationship between the primary I/O signals and the clock. If the timing for all critical paths is acceptable, then a constraints file may be written out from PrimeTime or DC for the purpose of forward annotation to the layout tool. This constraint file in SDF format specifies the timing between each group of logic that the layout tool uses, in order to perform the timing driven placement of cells. In the post-layout mode, the actual extracted delays are back annotated to PrimeTime to provide realistic delay calculation. These delays consist of the net capacitances and interconnect RC delays. Similar to synthesis, static timing analysis is also an iterative process. It is closely linked with the placement and routing of the chip. This operation is usually performed a number of times until the timing requirements are satisfied.

1.1.6 Placement, Routing and Verification

As the name suggests, the layout tool performs the placement and routing. There are a number of methods in which this step could be performed. However, only issues related to synthesis are discussed in this section. The quality of floorplan and placement is more critical than the actual routing. Optimal cell placement location, not only speeds up the final routing, but also produces superior results in terms of timing and reduced congestion. As explained previously, the constraint file is used to perform timing driven placement. The timing driven placement method forces the layout tool to place the cells according to the criticality of the timing between the cells. After the placement of cells, the clock tree is inserted in the design by the layout tool. The clock tree insertion is optional and depends solely on the design and users preference. Users may opt to use more traditional methods of routing the clock network, for example, using fishbone/spine structure for the clocks in order to reduce the total delay and skew of the clock. As technologies shrink, the spine approach is getting more difficult to implement due to the increase in resistance (thus, RC delays) of the interconnect wires. It is therefore the intent of this section (and the entire book) to stress solely on the clock tree synthesis approach. At this stage an additional step is necessary to complete the clock tree insertion. As mentioned above, the layout tool inserted the clock tree in the design after the placement of cells. Therefore, the original netlist that was generated from DC (and fed to the layout tool), lacks the clock tree information (essentially the whole clock tree network, including buffers and nets). Therefore, the clock tree must be re-inserted in the original netlist and formally verified. Some layout tools provide direct interface to perform this step. For the sake of simplicity, assume that the clock tree insertion to the original netlist has been performed. The layout tool generally performs routing in two phases: **global routing and detailed routing.** After placement, the design is globally routed to determine the quality of placement, and to provide estimated delays approximating the real delay values of the post-routed (after detailed routing) design. If the cell placement is not optimal, the global routing will take a longer time to complete, as compared to placing the cells. Bad placement also affects the overall timing of the design. Therefore, to minimize the number of synthesis-layout iterations and improve placement quality, the timing information is extracted from the layout, after the global routing phase. Although, these delay numbers are not as accurate as the numbers extracted after detailed routing, they do provide a fair idea of the post-routed timing. The estimated delays are back annotated to PrimeTime for analysis, and only when the timing is considered satisfactory, the remaining process is allowed to proceed. Detailed routing is the final step that is performed by the layout tool. After detailed route is complete, the real timing delays of the chip are extracted, and plugged into PrimeTime for analysis. These steps are iterative and depend on the timing margins of the design. If the design fails timing requirements, post-layout optimization is performed on the design before undergoing another iteration of layout. If the design passes static timing analysis, it is ready to undergo LVS (layout versus schematic) and DRC (design rule checking) before tape-out. It must be noted that all steps discussed above can also be applied for hierarchical place and route. In other words, one can repeat these steps for

each sub-block of the design before placing the sub-blocks together in the final layout and routing between the sub-blocks.

1.1.7 Engineering Change Order

This step is an exception to the normal design flow and should not be confused with the regular design cycle. Therefore, this step will not be explained in subsequent chapters. Many designers regard engineering change order (ECO) as the change required in the netlist at the very last stage of the SOC design flow. For instance, ECO is performed when there is a hardware bug encountered in the design at the very last stage (say, after tape-out), and it is necessary to perform a metal mask change by re-routing a small portion of the design. As a result ECO is performed on a small portion of the chip to prevent disturbing the placement and routing of the rest of the chip, thereby preserving the rest of the chips timing. Only the part that is affected is modified. This can be achieved, either by targeting the spare gates incorporated in the chip, or by routing only some of the metal layers. This process is termed as metal mask change. Normally, this procedure is executed for changes that require less than 10% modification of the whole chip (or a block, if doing hierarchical place and route). If the bug fix requires more than 10% change then it is best to repeat the whole procedure and re-route the chip (or the block). The latest version of DC incorporates the ECO compiler. It makes use of the mathematical algorithms (also used by the formal verification techniques), to automatically implement the required changes. Making use of the ECO compiler provides designers an alternative to the tedium of manually inserting the required changes in the netlist, thus minimizing the turn-around time of the chip. Some layout tools have incorporated the ECO algorithm within their tool. The layout tool has a built-in advantage that it does not suffer from the limitation of crossing the hierarchical boundaries associated with a design. Also, the layout tool benefits from knowing the placement location of the spare cells (normally included by the designers in the design), thus can target the nearest location of spare cells in order to implement the required ECO changes and achieve minimized routing.

1.2 Chapter Summary

In this chapter the SOC design flows incorporating the latest tools and technology for very deep sub-micron (VDSM) technologies were reviewed. The flow started with the definition of specification, and ended with physical layout. The significance was placed on logic and physical synthesis related topics. Also introduced was a new concept of physical synthesis as applicable to the design flow to shorten the design cycle of the chip. The need to perform physical synthesis was emphasized to get a better estimation of delays and shorten the time-to-market. ween the synthesis and the simulation environments. Other times, the control is needed simply to direct DC to map to certain types of components; or for embedding the constraints and attributes

directly in the HDL source code. DC provides a number of compiler directives targeted specifically for Verilog and VHDL design entry formats. These directives provide the means to control the outcome of synthesis, directly from the HDL source code. The directives are specified as comments in the HDL code, but have specific meaning for DC. These special comments alter the synthesis process, but have no effect on the simulation.

CHAPTER 2 :

Synthesis Basics

2. Synthesis Using Design Compiler Basics

This chapter provides basic information about Design Compiler functions. The chapter presents both high-level and basic synthesis design flows. Standard user tasks, from design preparation and library specification to compile strategies, optimization, and results analysis, are introduced as part of the basic synthesis design flow presentation. This chapter includes the following sections: The High-Level Design Flow Running Design Compiler Support for Multicore Technology Support for Multicorner-Multimode Designs. Figure 2 Basic High-Level Design Flow Using the design flow shown in Figure 2, you perform the following steps:

1. Start by writing an HDL description (Verilog or VHDL) of your design. Use good coding practices to facilitate successful Design Compiler synthesis of the design.
2. Perform design exploration and functional simulation in parallel.

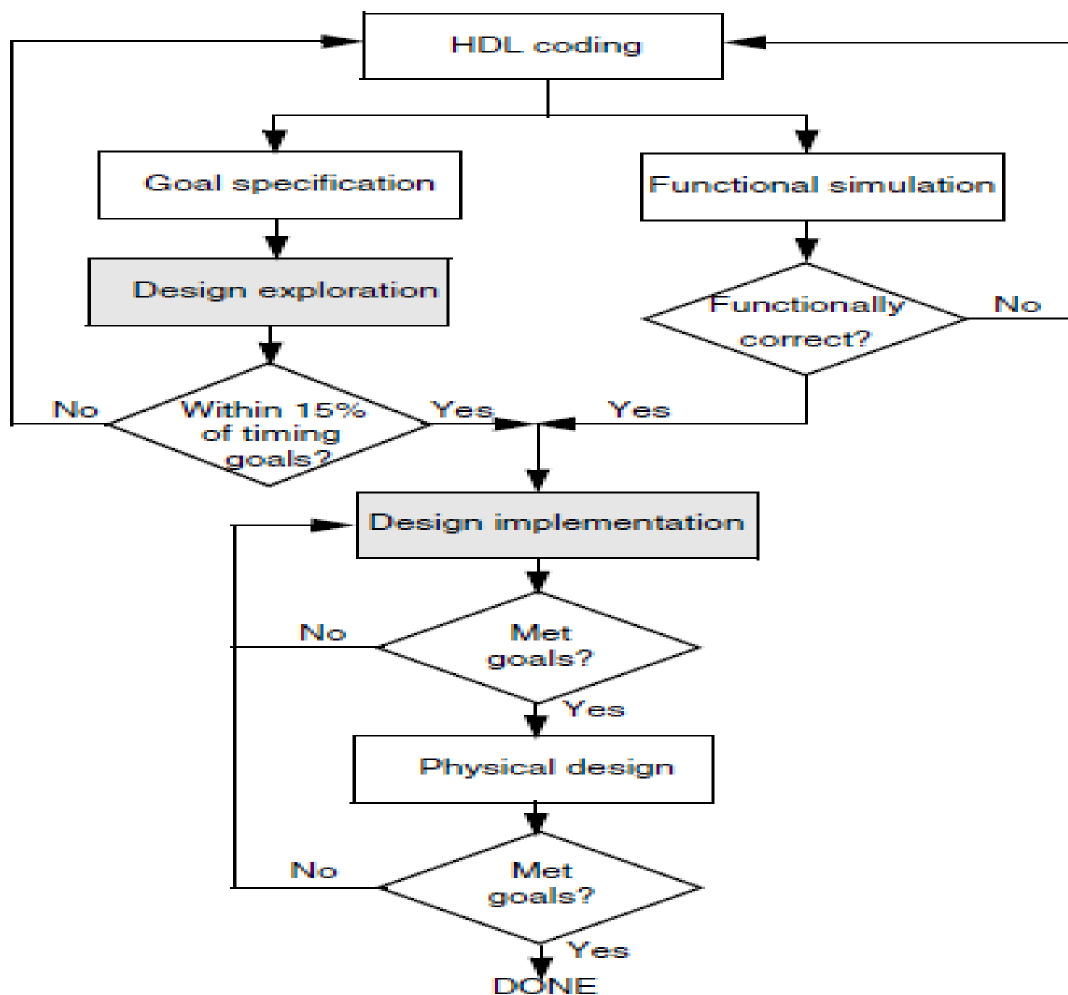

Figure 1

3. In design exploration,we use Design Compiler to

(a) Implement specific design goals (design rules and optimization constraints)
(b) Carry out a preliminary, default synthesis (using only the Design Compiler default options).

If design exploration fails to meet timing goals by more than 15 percent, modify your design goals and constraints, or improve the HDL code. Then repeat both design exploration and functional simulation. In functional simulation, determine whether the design performs the desired functions by using an appropriate simulation tool. If the design does not function as required, you must modify the HDL code and repeat both design exploration and functional simulation. Continue performing design exploration and functional simulation until the design is functioning correctly and is within 15 percent of the timing goals.

3. Perform design implementation synthesis by using Design Compiler to meet design goals. After synthesizing the design into a gate-level netlist, verify that the design meets your goals. If the design does not meet your goals, generate and analyze various reports to determine the techniques you might use to correct the problems.

After the design meets functionality, timing, and other design goals, complete the physical design (either in-house or by sending it to your semiconductor vendor). Analyze the physical designs performance by using back-annotated data. If the results do not meet design goals, return to step 3. If the results meet your design goals, you are finished with the design cycle.

[Please Turn Over For The Figure]

Following is the flow of the Design compiler which describes more:-

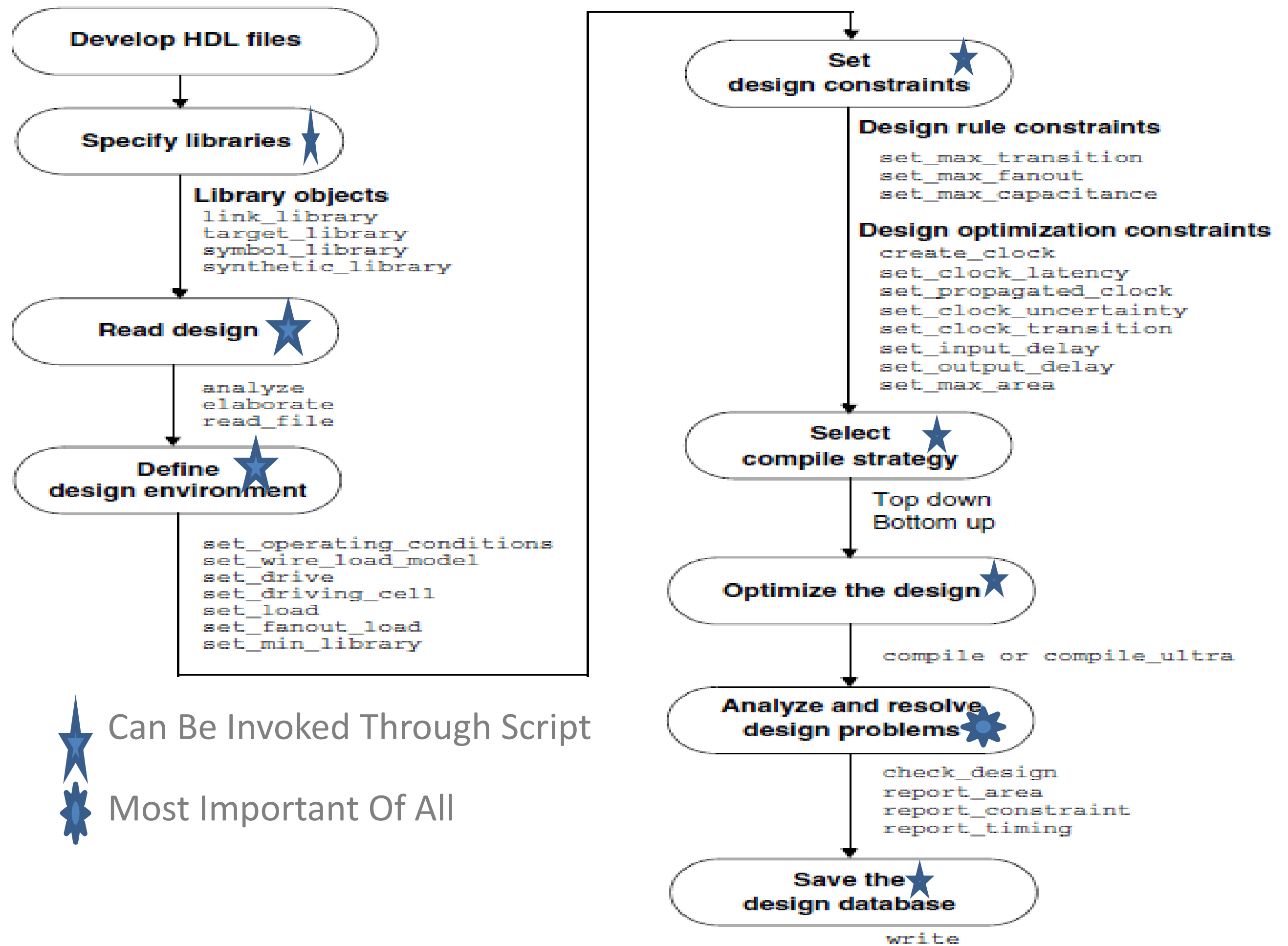

Figure 2

1. Develop HDL Files:- The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL. These design descriptions need to be written carefully to achieve the best synthesis results possible. When writing HDL code, you need to consider design data management, design partitioning, and your HDL coding style. Partitioning and coding style directly affect the synthesis and optimization processes. Note: This step is included in the flow, but it is not actually a Design Compiler step. You do not create HDL files with the Design Compiler tools.

2. Specify Libraries:- You specify the link, target, symbol, and synthetic libraries for Design Compiler by using the link_library, target_library, symbol_library, and synthetic_library commands. The link and target libraries are technology libraries that define the semiconductor vendors set of cells and related information, such as cell names, cell pin names, delay arcs, pin loading, design rules, and operating conditions. The symbol library defines the symbols for

schematic viewing of the design. You need this library if you intend to use the Design Vision GUI. In addition, you must specify any specially licensed DesignWare libraries by using the synthetic_library command. (You do not need to specify the standard DesignWare library.)

3. Read Design Design Compiler can read both RTL designs and gate-level netlists. Design Compiler uses HDL Compiler to read Verilog and VHDL RTL designs. It has a specialized netlist reader for reading Verilog and VHDL gate-level netlists. The specialized netlist reader reads netlists faster and uses less memory than HDL Compiler. Design Compiler provides the following ways to read design files: The analyze and elaborate commands The read_file command The read_vhdl and read_verilog commands. These commands are derived from the read_file -format VHDL and read_file -format verilog commands. Define Design Environment Design Compiler requires that you model the environment of the design to be synthesized. This model comprises the external operating conditions (manufacturing process, temperature, and voltage), loads, drives, fanouts, and wire load models.

4. Set Design Constraints Design Compiler uses design rules and optimization constraints to control the synthesis of the design. Design rules are provided in the vendor technology library to ensure that the product meets specifications and works as intended. Typical design rules constrain transition times (set_max_transition), fanout loads (set_max_fanout), and capacitances (set_max_capacitance). These rules specify technology requirements that you cannot violate. (You can, however, specify stricter constraints.) Optimization constraints define the design goals for timing (clocks, clock skews, input delays, and output delays) and area (maximum area). In the optimization process, Design Compiler attempts to meet these goals, but no design rules are violated by the process. You define these constraints by using commands such as those listed under this step in Figure 2. To optimize a design correctly, you must set realistic constraints. Note: Design constraint settings are influenced by the compile strategy you choose.

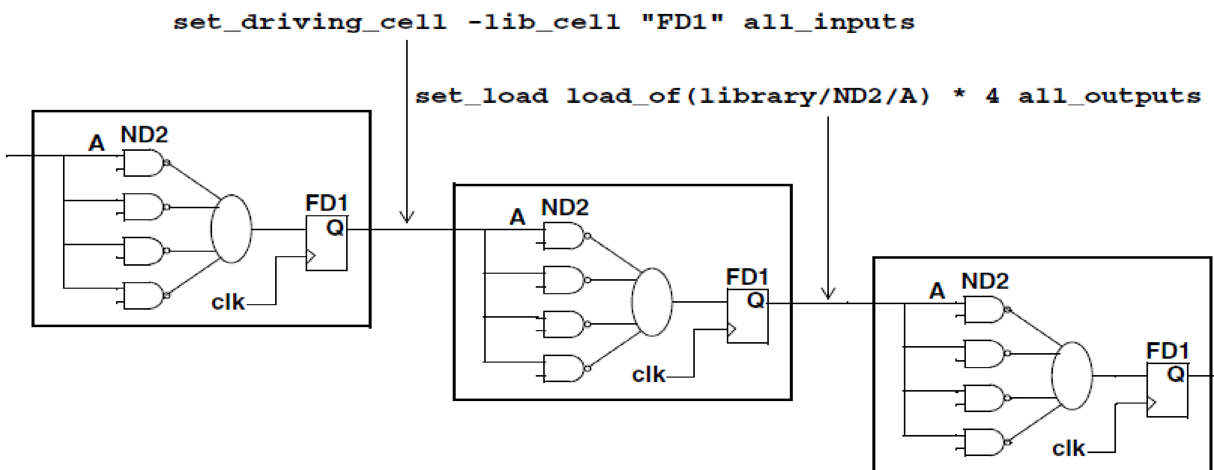

Figure 3

5. Select Compile Strategy The two basic compile strategies that you can use to optimize hierarchical designs are referred to as top down and bottom up. In the top-down strategy, the top-level design and all its sub-designs are compiled together. All environment and constraint settings are defined with respect to the top-level design. Although this strategy automatically takes care of inter-block dependencies, the method is not practical for large designs because all designs must reside in memory at the same time. In the bottom-up strategy, individual subdesigns are constrained and compiled separately. After successful compilation, the designs are assigned the dont_touch attribute to prevent further changes to them during subsequent compile phases. Then the compiled subdesigns are assembled to compose the designs of the next higher level of the hierarchy (any higher-level design can also incorporate unmapped logic), and these designs are compiled. This compilation process is continued up through the hierarchy until the top-level design is synthesized. This method lets you compile large designs because Design Compiler does not need to load all the uncompiled subdesigns into memory at the same time. At each stage, however, you must estimate the interblock constraints, and typically you must iterate the compilations, improving these estimates, until all subdesign interfaces are stable. Each strategy has its advantages and disadvantages, depending on your particular designs and design goals. You can use either strategy to process the entire design, or you can mix strategies, using the most appropriate strategy for each subdesign.

Note: The compile strategy you choose affects your choice of design constraints and the values you set. To Optimize the Design You use the compile_ultra command or compile command to invoke the Design Compiler synthesis and optimization processes. Several compile options are available with both commands. In particular, the map_effort option of the compile command can be set to medium or high. In a default compile, when you are performing design exploration, you use the medium map_effort option of the compile command. Because this option is the default, you do not need to specify map_effort in the compile command. In a final design implementation compile, you might want to set map_effort to high. Often setting map_effort to medium is sufficient. For designs that have significantly tight timing constraints, you can invoke a single DC Ultra command, compile_ultra, for better quality of results (QoR). The command is a push-button solution for timing-critical, high performance designs and encapsulates DC Ultra strategies into a single command.

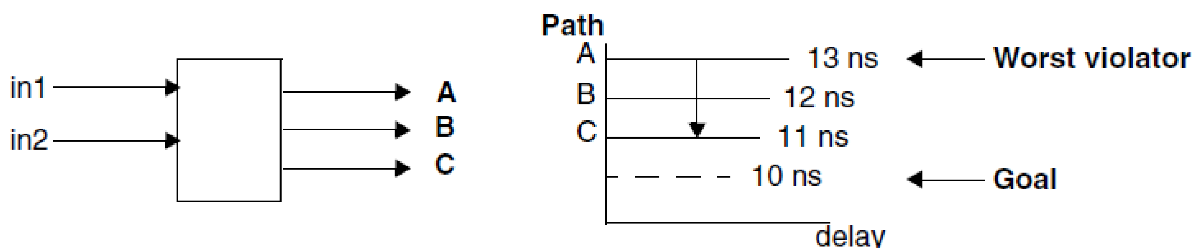

Figure 4

6. Analyze and Resolve Design Problems Design Compiler can generate numerous reports on the results of a design synthesis and optimization, for example, area, constraint, and timing reports. You use reports to analyze and resolve any design problems or to improve synthesis results. You can use the check_design command to check the synthesized design for consistency. Other check_ commands are available. To Save the Design Database You use the write command to save the synthesized designs. Remember that Design Compiler does not automatically save designs before exiting. You can also save in a script file the design attributes and constraints used during synthesis. Script files are ideal for managing your design attributes and constraints. The script contains comments that identify each of the steps in the flow. Some of the script command options and arguments have not yet been explained in this manual. Nevertheless, from the previous discussion of the basic synthesis flow, you can begin to understand this example of a top-down compile.

Top-Down Compile Script Example

```
 /* specify the libraries */
set target_library my_lib.db
set symbol_library my_lib.sdb
set link_library [list "*" $target_library]
/* read the design */
read_verilog Adder16.v
/* define the design environment */
set_operating_conditions WCCOM
set_wire_load_model "10x10"
set_load 2.2 sout set_load 1.5 cout
set_driving_cell -lib_cell FD1 [all_inputs] set_drive 0 clk
/* set the optimization constraints */
create_clock clk -period 10
set_input_delay -max 1.35 -clock clk {ain bin}
set_input_delay -max 3.5 -clock clk cin
set_output_delay -max 2.4 -clock clk cout
set_max_area 0
/* map and optimize the design */
compile
/* analyze and debug the design */
report_constraint -all_violators report_area
/* save the design database */
write -format ddc -hierarchy -output Adder16.ddc
```

We can execute these commands in any of the following ways: Enter dc_shell and type each command in the order shown in the example. Enter dc_shell and execute the script file, using the source command. For example, if you are running Design Compiler and the script is in a file called run.scr, you can execute the script file by entering the following command: dc_shell> source run.tcl Run the script from the UNIX command line by using the -f option of the dc_shell command.

2.2 Summary:-

From this chapter following are the take-aways:-

1. We should use the options of the DC judiciously, so that we don't compromise the compute intensiveness.
2. DC is able to optimize the Hardware on a specific constraint given to it.
3. DC takes in the RTL + Timing Constraints + Area Constraints to derive the synthesized netlist for the design.

CHAPTER 3 :

Optimization Strategies

3.1 Optimizing the Design

Optimization is the Design Compiler synthesis step that maps the design to an optimal combination of specific target library cells, based on the design's functional, speed, and area requirements. You use the compile_ultra command or the compile command to compile a design. Design Compiler provides options that enable you to customize and control optimization. Several of the many factors affecting the optimization outcome are discussed in this chapter.

This chapter has the following sections:

- The Optimization Process
- Selecting and Using a Compile Strategy
- Resolving Multiple Instances of a Design Reference
- Preserving Subdesigns
- Understanding the Compile Cost Function
- Performing Design Exploration
- Performing Design Implementation

3.1.1 The Optimization Process

Design Compiler performs the following three levels of optimization:

- **Architectural optimization**
- **Logic-level optimization**
- **Gate-level optimization**

The following sections describe these processes.

3.1.1.1 Architectural Optimization

Architectural optimization works on the HDL description. It includes such high-level synthesis tasks as

- **Sharing common subexpressions**
- **Sharing resources**
- **Selecting DesignWare implementations**
- **Reordering operators**
- **Identifying arithmetic expressions for data-path synthesis (DC Ultra only).**

Except for DesignWare implementations, these high-level synthesis tasks occur only during the optimization of an unmapped design. DesignWare selection can recur after gate-level mapping. High-level synthesis tasks are based on your constraints and your HDL coding style. After high-level optimization, circuit function is represented by GTECH library parts, that is, by a generic, technology-independent netlist.

3.1.1.2 Logic-Level Optimization

Logic-level optimization works on the GTECH netlist. It consists of the following two processes:

• Structuring

This process adds intermediate variables and logic structure to a design, which can result in reduced design area. Structuring is constraint based. It is best applied to noncritical timing paths. During structuring, Design Compiler searches for subfunctions that can be factored out
and evaluates these factors, based on the size of the factor and the number of times the factor appears in the design. Design Compiler turns the subfunctions that most reduce the logic into intermediate variables and factors them out of the design equations.

• Flattening

The goal of this process is to convert combinational logic paths of the design to a two-level, sum-of-products representation. Flattening is carried out independently of constraints. It is useful for speed optimization because it leads to just two levels of combinational logic. During flattening, Design Compiler removes all intermediate variables, and therefore all its associated logic structure, from a design.

3.1.1.3 Gate-Level Optimization

Gate-level optimization works on the generic netlist created by logic synthesis to produce a technology-specific netlist. It includes the following processes:

• Mapping

This process uses gates (combinational and sequential) from the target technology libraries to generate a gate-level implementation of the design whose goal is to meet timing and area goals. You can use the various options of the compile_ultra command or the compile command to control the mapping algorithms used by Design Compiler.

• Delay optimization

The process goal is to fix delay violations introduced in the mapping phase. Delay optimization does not fix design rule violations or meet area constraints.

• Design rule fixing

The process goal is to correct design rule violations by inserting buffers or resizing existing cells. Design Compiler tries to fix these violations without affecting timing and area results, but if necessary, it does violate the optimization constraints.

• Area optimization

The process goal is to meet area constraints after the mapping, delay optimization, and design rule fixing phases are completed. However, Design Compiler does not allow area recovery to introduce design rule or delay constraint violations as a means of meeting the area constraints.

3.2 Understanding the Optimization Cost Function

The compile cost function consists of design rule costs and optimization costs. By default, Design Compiler prioritizes costs in the following order:

1. **Design rule costs**
   a. Connection class
   b. Multiple port nets
   c. Maximum transition time
   d. Maximum fanout
   e. Maximum capacitance
   f. Cell degradation
2. **Optimization costs**
   a. Maximum delay
   b. Minimum delay
   c. Maximum power
   d. Maximum area

The compile cost function considers only those components that are active in your design. Design Compiler evaluates each cost function component independently, in order of Importance. When evaluating cost function components, Design Compiler considers only violators (positive difference between actual value and constraint) and works to reduce the cost
function to zero. The goal of Design Compiler is to meet all constraints. However, by default, it gives precedence to design rule constraints because design rule constraints are functional requirements for designs. Using the default priority, Design Compiler fixes design rule violations even at the expense of violating your delay or area constraints. You can change the priority of the maximum design rule costs and the delay costs by using the set_cost_priority command to specify the ordering. You must run the set_cost_priority command before running the compile

command. You can disable evaluation of the design rule cost function by using the -no_design_rule option when running the compile_ultra command or compile command. You can disable evaluation of the optimization cost function by using the -only_design_rule option when running the compile_ultra command or compile command.

3.3 Performing Design Implementation

The default compile generates good results for most designs. If your design meets the optimization goals after design exploration, you are finished. If not, try the techniques described in the following sections:

- **Optimizing High-Performance Designs**
- **Optimizing for Maximum Performance**
- **Optimizing for Minimum Area**
- **Optimizing Data Paths**

3.3.1 Optimizing High-Performance Designs

For high-performance designs that have significantly tight timing constraints, you can invoke a single DC Ultra command, compile_ultra, for better quality of results (QoR). This command allows you to apply the best possible set of timing-centric variables or commands during ompile for critical delay optimization as well as improvement in area QoR. Because compile_ultra includes all compile options and starts the entire compile process, no separate compile command is necessary. By default, if the dw_foundation.sldb library is not in the synthetic library list but the DesignWare license has been successfully checked out, the dw_foundation.sldb library is automatically added to the synthetic library list. This behavior applies to the current command only. The user-specified synthetic library and link library lists are not affected. In addition, all DesignWare hierarchies are, by default, unconditionally ungrouped in the second pass of the compile. You can prevent this ungrouping by setting the compile_ultra_ungroup_dw variable to false (the default is true).

3.3.2 Optimizing for Maximum Performance

If your design does not meet the timing constraints, you can try the following methods to improve performance:

- Create path groups
- Fix heavily loaded nets
- Auto-ungroup hierarchies on the critical path
- Perform a high-effort incremental compile
- Perform a high-effort compile

3.3.3 Optimizing for Minimum Area

If your design has timing constraints, these constraints always take precedence over area requirements. For area-critical designs, do not apply timing constraints before you compile.
If you want to view timing reports, you can apply timing constraints to the design after you compile.
If your design does not meet the area constraints, you can try the following methods to reduce the area:

• Disable total negative slack optimization
• Optimize across hierarchical boundaries

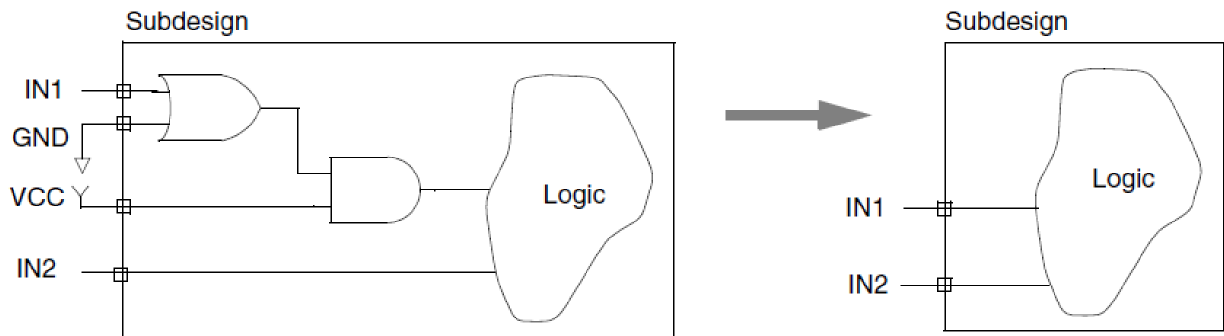

Figure 5

3.3.4 Optimizing Data Paths

Datapath design is commonly used in applications that contain extensive data manipulation, such as 3-D, multimedia, and digital signal processing (DSP). Datapath extraction transforms arithmetic operators (for example, addition, subtraction, and multiplication) into datapath blocks to be implemented by a datapath generator. This transformation improves the QOR by utilizing the carry-save arithmetic technique. Beginning with version W-2004.12, Design Compiler provides an improved datapath generator and better arithmetic components for both DC Expert and DC Ultra. To take advantage of these enhancements, make sure that the dw_foundation.sldb library is listed in the synthetic library. If necessary, use the set synthetic_library dw_foundation.sldb command. These enhancements require a DesignWare license. DC Ultra enables datapath extraction and explores various datapath and resource-sharing options during compile. DC Ultra datapath optimization provides the following benefits:

• Shares datapath operators
• Extracts the datapath
• Explores better solutions that might involve a different resource-sharing configuration
• Allows the tool to make better tradeoffs between resource sharing and datapath Optimization.

CHAPTER 4 :

Synthesis Outputs

4.1 Understanding Synthesis Outputs

Synthesis tool gives the Outputs in the following forms:-

1. Reports
2. Log
3. Output Netlist: In form Of ddc or Verilog

We present here the snaps of the reports which should be **clean** to proceed to the PNR step.

1. Reports

```
****************************************
Report : qor
Design : c69p0snsflistopp
Version: D-2010.03-SP5
Date   : Sun Oct 23 12:51:03 2011
****************************************


 Timing Path Group 'flis_gated_clk'
 -----------------------------------
 Levels of Logic:             2.00
 Critical Path Length:      536.79
 Critical Path Slack:      1242.90
 Critical Path Clk Period: 2500.00
 Total Negative Slack:        0.00
 No. of Violating Paths:      0.00
 Worst Hold Violation:        0.00
 Total Hold Violation:        0.00
 No. of Hold Violations:      0.00
 -----------------------------------


 Timing Path Group 'default'
 -----------------------------------
 Levels of Logic:             9.00
 Critical Path Length:      338.93
 Critical Path Slack:       -38.93
 Critical Path Clk Period:   Undef
 Total Negative Slack:    -1099.06
 No. of Violating Paths:     32.00
 Worst Hold Violation:        0.00
 Total Hold Violation:        0.00
 No. of Hold Violations:      0.00
 -----------------------------------


 Cell Count
 -----------------------------------
 Hierarchical Cell Count:        1
 Hierarchical Port Count:      840
 Leaf Cell Count:             1980
 Buf/Inv Cell Count:           569
 CT Buf/Inv Cell Count:          0
 -----------------------------------


 Area
 -----------------------------------
 Combinational Area:   1999.669256
 Noncombinational Area: 1714.509687
 Net Area:                0.000000
 -----------------------------------
 Cell Area:            3714.178943
 Design Area:          3714.178943


 Design Rules
 -----------------------------------
 Total Number of Nets:        2339
 Nets With Violations:         385
 Max Trans Violations:         271
 -----------------------------------
```

Design Data Synthesis Issue Summary

```
Warning: File reports/c69p0snsflistopp.cg_issues.syn.rpt could not be opened.  Clock-gate collapsing information unavailable.
/-----------------------------------------------------------------------------------------------------\
| Clock-gate |  Low   | Unclocked | Clock | Reset | Gclatchen | Latch | Latch enables | Combinational |
| collapsing | clock | Registers | logic | logic |   logic   | races | not driven by |  feedthroughs |
|          | gates |          |      |      |          |      |   gclatchen  |             |
|------------+-------+-----------+-------+-------+-----------+-------+---------------+---------------|
|   N/A    |  0  |    0    |  0  | 18  |    0    |  0  |      0      |     306     |
\-----------------------------------------------------------------------------------------------------/

/-------------------------------------------------------\
| Errors in | Bonus | Unloaded  | Unregistered | Latched |
|  logfile  | cells | gclatchen |   outputs    | outputs |
|          |      |   cells   |             |        |
|-----------+-------+-----------+--------------+---------|
|    0    |  0  |    0    |      32     |   0   |
\-------------------------------------------------------/

/------------------------------------------------------\
| Timing loops  |        Netlist syntax issues        |
|---------------+--------------------------------------|
| RTL | Netlist | Slashes | 1'b0/1'b1 | assigns | wands |
|-----+---------+---------+-----------+---------+-------|
|  0  |    0    |    0    |    458    |    5    |       |
\------------------------------------------------------/
```

```
Design Summary: MOST IMPORTANT OF ALL
===============
  Total Cell Count    : 1980
  Total Area          : 3714.17894400004
  Total Gate Count    : 3714.17894400004 / 0.645888 = 5750.50
  Total PWidth Zp     : 0.00
  Total NWidth Zn     : 0.00
  Total Width Z=Zp+Zn : 0.00
  Total Cell leakage power : 481.35
```

CHAPTER 5 :

Automatic Place And Route Flow

5. 1 IC Compiler Basic Design Flow

The IC Compiler design flow includes design planning, placement and optimization, clock tree synthesis and optimization, this flow produces GSDII output.For most designs, the placement and optimization, clock tree synthesis and optimization, and routing and postroute optimization steps are preset for optimal results.

The following sections describe the basic design flow:
5.1.1 Overview of the IC Compiler Design Flow
5.1.2 Design Planning and Power Planning
5.1.3 Placement and Optimization
5.1.4 Clock Tree Synthesis and Optimization
5.1.5 Routing and Postroute Optimization

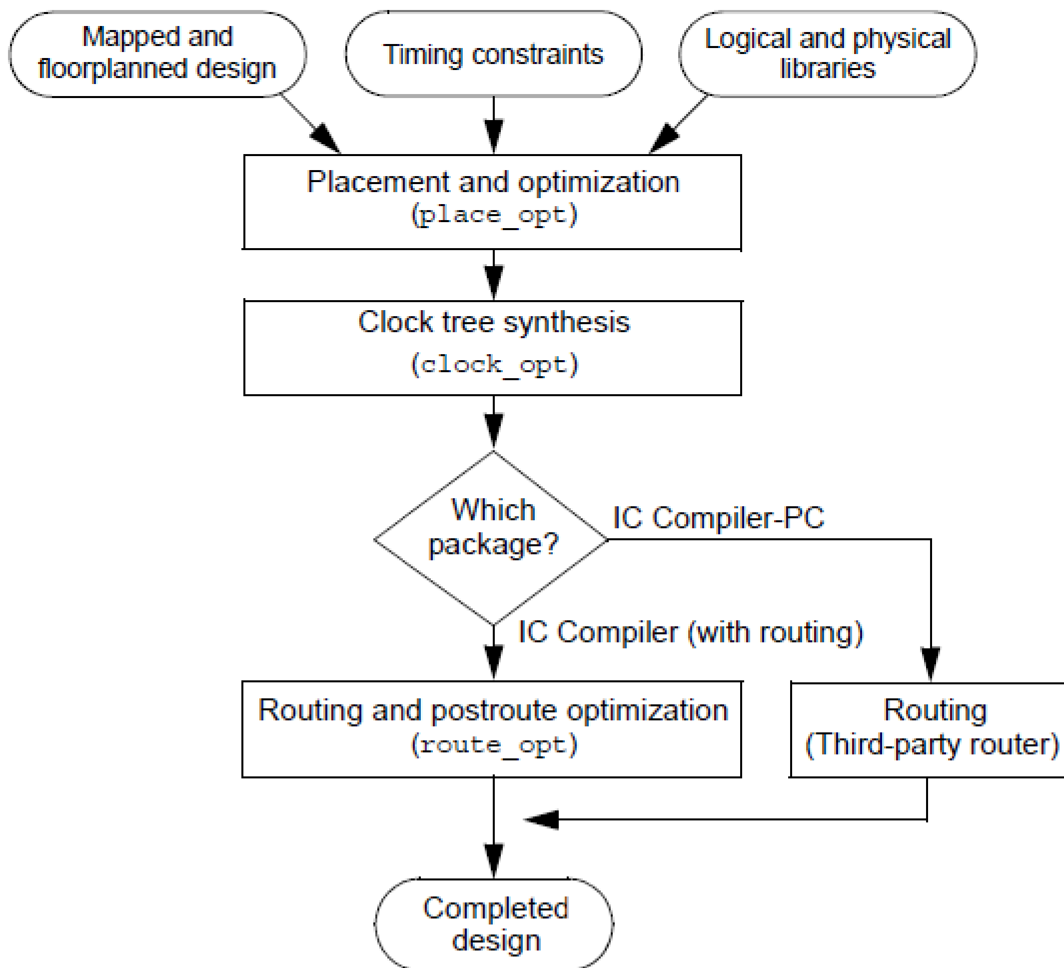

Figure 6

5.1.1 Overview of the IC Compiler Design Flow

As illustrated in Figure, the IC Compiler design flow is a single-pass flow with design planning and three basic steps for placement and optimization, clock tree synthesis and optimization, and routing and postroute optimization. Depending on the package of the IC Compiler tool you are using, this flow either includes routing and optimization or it outputs the design for routing by a routing tool not made by Synopsys. “Chip Finishing and Design for Manufacturing.”

5.1.2 Design Planning and Power Planning

After you have set up your design in IC Compiler, you can create a floorplan to determine the size of the design cell, create the boundary and core area, create site rows for placement of standard cells, set up the I/O pads, and create a power plan. To do these functions, you first need to read in the I/O constraints and initialize the floorplan.

5.1.3 Placement and Optimization

IC Compiler placement and optimization addresses and resolves timing closure for your design. This iterative process uses enhanced placement and synthesis technologies to generate a legalized placement for leaf cells and an optimized design. You can supplement this functionality by optimizing for power, recovering area for placement, minimizing congestion, and minimizing timing and design rule violations. The following sections explain how to set up and perform these placement functions:

Prerequisites for Placement and Optimization
- **Setting Up for Placement and Optimization**
- **Running Placement and Optimization**

5.1.3.1 Prerequisites for Placement and Optimization

For the best results, the placement and optimization process requires the following:
• The design has a realistic floorplan that includes
- Placed macros that are fixed in their locations
- Power and ground nets
- Standard cell utilization that is between 50 and 80 percent

• The design must be routable
This is necessary for timing convergence during placement and optimization.

5.1.3.2 Setting Up for Placement and Optimization

Placement and optimization is already set up to provide optimal results for most designs. If necessary, you can augment this by preventing IC Compiler from automatically buffering high-fanout nets. You can also perform incremental placement functions.

5.1.3.3 Running Placement and Optimization

1. Choose Placement > Core Placement and Optimization.
The Core Placement and Optimization dialog box appears.
2. Specify the placement and optimization settings you need. Table lists the ways you can change the placement and optimization settings in the Core Placement and Optimization dialog box.

TABLE 1

To do this	Specify
Improve the quality of results or reduce runtime.	Low, Medium, or High in the Effort section. Low minimizes runtime and High maximizes quality. Medium is the default.
Adjust the placement.	High in the Effort section but make sure that Congestion is not selected. You must previously set the `psynopt_enable_adjust_placement` variable to true.
Recover area for cells not on timing critical paths.	Area recovery (select "Recover area").
Optimize leakage power, dynamic power, perform power-aware placement, or a combination of these optimizations.	Power optimizations (select "Power optimization"). This activates all the power optimizations already enabled. By default, leakage power optimization is enabled.
To do this	Specify
Run clock tree synthesis and optimization with placement and optimization.	Clock tree synthesis and optimization (select "Clock compilation, optimization and routing"). Typically, this is used for the quick flow for designs that do not need extensive clock tree synthesis and optimization work. If you run clock tree synthesis with placement and optimization, you do not need to run clock tree synthesis after this step. Instead, you can run routing and postroute optimization.
Minimize congestion.	Congestion awareness (select "Minimize congestion.")
Reorder scan chains.	Reordering of scan chains (select "Reorder scan during placement").
Use additional CPUs.	Number of CPUs to use in the "Number of CPUs" box.

Power optimizations (select “Power optimization”). This activates all the power optimizations already enabled. By default, leakage power optimization is enabled. Alternatively, you can use the place_opt command.

Run clock tree synthesis and optimization with placement and optimization. Clock tree synthesis and optimization (select “Clock compilation, optimization and routing”). Typically, this is used for the quick flow for designs that do not need extensive clock tree synthesis and optimization work. If you run clock tree synthesis with placement and optimization, you do not need to run clock tree synthesis after this step. Instead, you can run routing and postroute optimization.

5.1.3.4 Analyzing Placement and Optimization

After you complete placement and optimization, you can analyze the design, as explained in “Analyzing Placement”. Optimize leakage power, dynamic power, perform power-aware placement, or a combination of the optimizations already enabled.

Run clock tree synthesis and optimization with placement and optimization. Typically, this is used for the quick flow for designs not requiring extensive clock tree synthesis and optimization work.

Clock tree synthesis. -cts
Use additional CPUs. -num_cpus number
Reorder scan chains. -optimize_dft
Minimize congestion. -congestion

5.2 Performing Clock Tree Synthesis

The following sections explain how you can set up and perform these clock tree synthesis functions:

- **Prerequisites for Clock Tree Synthesis**
- **Defining the Clock Trees**
- **Running Clock Tree Synthesis and Optimization**

5.2.1 Prerequisites for Clock Tree Synthesis

The clock tree synthesis and optimization process has the following prerequisites:

- The design must be placed, optimized, and legalized.If congestion issues are not resolved during placement and optimization, the addition of clock trees can increase congestion. If the design is congested, you can rerun place_opt with the -congestion and -effort high options, but the runtime can be long.

If the design is not legalized, clock tree synthesis might have runtime and QoR issues. To legalize the design, run the legalize_placement command.

- Clock references in the logical library should not have dont_use or dont_touch attributes.

If these attributes exist, remove them.

• The physical library should include
- All clock buffers and inverters
- Routing information, which includes layer information and nondefault rules

• TLUPlus models must exist.
Extraction requires these models to estimate the net resistance and capacitance.

5.2.2 Defining the Clock Trees

By default, IC Compiler uses the clock sources defined by the create_clock command as the clock roots and derives the clock endpoints by tracing through all cells in the transitive fanout of the clock root. The types of pins that become default clock sinks include
• Clock pins of sequential cells (latches or flip-flops)
• Clock pins of macro cells

When synthesizing the clock tree, the default set of clock tree references includes all of the buffers and inverters defined in your logic library. You can change this default behavior to meet the special needs of your design by changing the clock tree options, exceptions, and references.

5.2.3 Running Clock Tree Synthesis and Optimization

When you use IC Compiler to perform clock tree synthesis and embedded clock tree optimization, it does the following:

• Clock tree synthesis
During clock tree synthesis, IC Compiler builds clock trees that meet the clock tree design rule constraints while balancing the loads and minimizing the clock skew.

• Clock tree optimization
During clock tree optimization, IC Compiler fixes the placement of the clock sinks, performs incremental logic and placement optimization, and fixes the placement of both the buffers and registers on the clock tree. You can also choose to fix hold time violations and perform interclock delay balancing.

• Clock routing

• Preroute RC estimation

• Placement and timing optimization based on the propagated clock arrival times

To perform clock tree synthesis and embedded clock tree optimization,
1. Choose CTS > Core CTS and Optimization.
The Core CTS and Optimization dialog box appears.
2. Select hold time fixing, interclock delay balancing, and updating

of clock latency values, if desired.
3. Click OK or Apply.

Alternatively, you can use the clock_opt command. For example,

icc_shell> clock_opt -fix_hold -inter_clock_balance

5.3 Routing and Postroute Optimization

As part of routing and postroute optimization, IC Compiler performs global routing, track assignment, detail routing, search and repair, topological optimization, and engineering change order (ECO) routing. For most designs, the default routing and postroute optimization setup produces optimal results. If necessary, you can supplement this functionality by optimizing routing patterns and reducing crosstalk or by customizing the routing and postroute optimization functions for special needs

- **Prerequisites for Routing and Postroute Optimization**
- **Running Routing and Postroute Optimization**
- **Analyzing Routing and Postroute Optimization**

5.3.1 Prerequisites for Routing and Postroute Optimization

Before you can perform routing and postroute optimization, your design must meet the following conditions:

- Clock tree synthesis and optimization have been performed.
- There are no (or almost no) timing violations based on the estimated delays.
- There are no maximum capacitance or maximum transition time violations based on the estimated values.
- Estimated congestion is acceptable.

Routing and postroute optimization is already set up to provide optimal results for most designs. If necessary, you can modify the setup for routing and postroute optimization (Route > Core Routing and Postroute Optimization or route_opt).

Routing and postroute optimization does the following:

- Performs one or more of the following routing stages:

- Global routing

Assigns nets to specific metal layers and global routing cells while minimizing detours as it avoids congested global routing cells, power and ground nets, and routing blockages.

- Track assignment

Assigns each net in the design to a specific track and lays down the actual metal traces, which it attempts to make long and straight with the minimum number of vias.

- Detail routing

Attempts to fix all DRC violations after track assignment for each portion (defined by a switch box) of the design. Because the switch box size is fixed, detail routing might not fix all DRC violations.

- Search and repair

Fixes the remaining DRC violations through multiple iterations using progressively larger switch box sizes.

• Performs a topological optimization and incrementally reroutes the design to close timing, signal integrity, and routing violations.

• Performs postroute RC extraction.

By default, IC Compiler is already set up to provide optimal routing results for most designs, but you can customize this setup when you are debugging the design.

To perform routing and postroute optimization,

1. Choose Route > Core Routing and Optimization.
The Core Routing and Optimization dialog box appears.

2. If you need to customize the setup for debugging or special
Conditions.

3. Click OK or Apply.

- The output of all the above steps is a routed design. The design has to undergo several checks once is routed. For example Layout versus Schematic Checks, Design Rule Checks, FEV etc..
- The following chapters are restricted to the results earned by all these steps and the detailed explanation of all these is out of scope of this thesis.

CHAPTER 6 :

Results Of Sign-Off

This chapter summarizes all the results of the datapath block in the form of snapshots. All the snapshots are self-explainatory and so no written explaination is done. However, the Highlighting is used to focus on some key points. Following are the Sign-Off Checks Done on the block :

1. Timing Sign-Off
2. Power Sign-Off
3. Functional Sign-Off
4. Layout vs Schematic Sign-Off
5. Electrical Rule Checks

Snapshots Of Results:

1. Timing Sign-Off Results:-

This sign off check is done using Prime Time Tool. Without this check getting clean, The design is not assured to be working.

```
c69p0snsflistopp.worst_all_max.syn.rpt.gz =+ (/nfs/iind/d...S/syn/SBL2_Model_Runs/c69p0snsflistopp_frz_2/reports) -
File  Edit  Tools  Syntax  Buffers  Window  Help
38   flis_top_apb_addr[6] (net)                          12     88.65                     0.00      0.00 f
39   i_flis/apb_addr[6] (c69p0snsflistopp_flis_0)                                         0.00      0.00 f
40   i_flis/apb_addr[6] (net)                                   88.65                     0.00      0.00 f
41   i_flis/U239/b (yc8hbnna021nx901f)                                   300.00           0.00      0.00 f
42   i_flis/U239/o (yc8hbnna021nx901f)                                    56.19          64.51     64.51 r
43   i_flis/n1238 (net)                                   1      7.21                     0.00     64.51 r
44   i_flis/U143/a (yc8hbnno021nx701f)                                    56.19           0.00     64.51 r
45   i_flis/U143/o (yc8hbnno021nx701f)                                    22.34          29.82     94.33 f
46   i_flis/n725 (net)                                    2      9.15                     0.00     94.33 f
47   i_flis/U144/a (yc8hbnin001nx701f)                                    22.34           0.00     94.33 f
48   i_flis/U144/o (yc8hbnin001nx701f)                                    14.43          19.06    113.39 r
49   i_flis/n1033 (net)                                   4     17.29                     0.00    113.39 r
50   i_flis/U147/c (yc8hbnno131nx201f)                                    14.43           0.00    113.39 r
51   i_flis/U147/o (yc8hbnno131nx201f)                                    19.47          16.10    129.49 f
52   i_flis/n1289 (net)                                   1      1.69                     0.00    129.49 f
53   i_flis/U121/c (yc8hbnca011nx201f)                                    19.47           0.00    129.49 f
54   i_flis/U121/o (yc8hbnca011nx201f)                                    34.30          30.98    160.47 r
55   i_flis/n67 (net)                                     1      4.65                     0.00    160.47 r
56   i_flis/U477/d (yc8hbnna041nx501f)                                    34.30           0.00    160.47 r
57   i_flis/U477/o (yc8hbnna041nx501f)                                    30.11          49.98    210.45 f
58   i_flis/n83 (net)                                     1      3.21                     0.00    210.45 f
59   i_flis/U480/b (yc8hbnco011nx301f)                                    30.11           0.00    210.45 f
60   i_flis/U480/o (yc8hbnco011nx301f)                                    20.63          31.82    242.27 r
61   i_flis/n88 (net)                                     1      2.14                     0.00    242.27 r
62   i_flis/U226/b (yc8hbnna031nx201f)                                    20.63           0.00    242.27 r
63   i_flis/U226/o (yc8hbnna031nx201f)                                    24.15          32.25    274.52 f
64   i_flis/rddata[0] (net)                               1      3.22                     0.00    274.52 f
65   i_flis/rddata[0] (c69p0snsflistopp_flis_0)                                           0.00    274.52 f
66   n311 (net)                                                  3.22                     0.00    274.52 f
67   U191/a (yc8hbnbf001nx701f)                                           24.15           0.00    274.52 f
68   U191/o (yc8hbnbf001nx701f)                                           48.34          20.58    295.00 f
69   flis_top_rddata[0] (net)                             1    120.00                     0.00         f
70   flis_top_rddata[0] (out)                                             48.34           0.00         f
71   data arrival time
72
73   max_delay                                                                          300.72    300.7200
74   output external delay                                                                0.00    300.7200
75   data required time                                                                           300.7200
76   ---------------------------------------------------------------------------------------------------------
77   data required time                                                                           300.7200
78   data arrival time                                                                            295.00
79   ---------------------------------------------------------------------------------------------------------
80   slack (MET)                                                                                  5.72
```

Figure 7

2. **Power Sign-Off Results**

   Early in the design cycle the Target Power numbers are given to the designer. If this power numbers are not satisfied, the design can not be signed off for the Fabrication. The power numbers are distributed over the blocks on the basis of the expected gate counts. The target Power number given for this block was 5mW which was met with a good margin of about 200uW.

```
30 Global Operating Voltage = 0.9
31 Power-specific unit information :
32     Voltage Units = 1V
33     Capacitance Units = 1.000000ff
34     Time Units = 1ps
35     Dynamic Power Units = 1mW    (derived from V,C,T units)
36     Leakage Power Units = 1uW
37
38
39 ------------------------------------------------------------------------------------
40                                          Switch    Int      Leak     Total
41 Hierarchy                                Power     Power    Power    Power     %
42 ------------------------------------------------------------------------------------
43 c69p0snsflistopp                          1.643     2.858   371.86    4.873  100.0
44   i_flis (c69p0snsflistopp_flis_0)        1.007     2.770   331.19    4.108   84.3
45 1
```

Figure 8

3. **Functional Sign-Off (LEC)**

   This check is must for design to be functionally correct. This is the check which is done on each step of Physical design. Before taping in the design this check has to go through.

```
File  Edit  Tools  Syntax  Buffers  Window  Help

 1 ===========================================================================
 2 Compared points      PO     DFF     DLAT    BBOX      Total
 3 ---------------------------------------------------------------------------
 4 Equivalent           534    54898   9       5         55446
 5 ===========================================================================
 6 0 Non-equivalent point(s) reported
 7 0 compared point(s) reported
 8 ===========================================================================
 9 Compared points      PO     DFF     DLAT    BBOX      Total
10 ---------------------------------------------------------------------------
11 Equivalent           534    54898   9       5         55446
12 ===========================================================================
13 0 Abort point(s) reported
14 0 compared point(s) reported
15 ===========================================================================
16 Compared points      PO     DFF     DLAT    BBOX      Total
17 ---------------------------------------------------------------------------
18 Equivalent           534    54898   9       5         55446
19 ===========================================================================
20 0 compared point(s) reported
21 ===========================================================================
22 Compared points      PO     DFF     DLAT    BBOX      Total
23 ---------------------------------------------------------------------------
24 Equivalent           534    54898   9       5         55446
25 ===========================================================================
26 0 Not-compared point(s) reported
27 0 compared point(s) reported
28 ===========================================================================
29 Compared points      PO     DFF     DLAT    BBOX      Total
30 ---------------------------------------------------------------------------
31 Equivalent           534    54898   9       5         55446
32 ===========================================================================
```

Figure 9

4. Layout vs Schematic Checks

This check is done whether the circuit on the Silicon canvas is equivalent to the Schematic of the circuit built. This check is critical as it can spot any error the PNR tool has done in converting the circuit from netlist to the silicon level. Also shorts, Opens and other checks are being done.

```
+---------------------------------------------------------------------------+
|                      Hercules LVS Comparison Report                       |
+---------------------------------------------------------------------------+

COMPARE (R) Hierarchical Layout Vs. Schematic
        SUSE.64 Release B-2008.09.SP3.24010 2010/08/24
Copyright (C) Synopsys, Inc. All rights reserved.

----------------------------------------------------------------------------
LVS error file      = c69p0snsflistoppop.LVS_ERRORS
Layout error file   = c69p0snsflistoppop.LAYOUT_ERRORS
Schematic netlist   = /localdisk/coe69all_tjkhamar/tjkhamar.c69p0snsflistoppop.trcstdlvs.iss/
c69p0snsflistoppop.herc
Layout netlist      = c69p0snsflistoppop.net
Equivalence file    = [automatic]
Runset file         = runiss.tjkhamar.c69p0snsflistoppop.trcstdlvs.evx
Working directory   = /tmp/localdisk/coe69all_tjkhamar/tjkhamar.c69p0snsflistoppop.trcstdlvs.
iss
Compare directory   = ./compare
Compare start time  = 2011-09-28 16:26:43

----------------------------------------------------------------------------
Top block compare result: PASS

         [C69P0SNSFLISTOPP == C69P0SNSFLISTOPP]

----------------------------------------------------------------------------
Comparison summary

    74 successful equivalencies
     0 failed equivalencies
```

Figure 10

5. Electrical Rule Checks

This check is done to confirm that IOs, Nets and every other component complies the required electrical characteristics. These include Diode checks, Net Length check etc..

```
#-   ERC Run Log
#-   USER : tjkhamar
#-   Fub  : c69p0snsflistopp
#-   HOST : inlc6612
#-   PRJ  : coe69all
#-   CADROOT_VER  : 5.0.1.1_stPseOpt64X
#-   VER : 10ww41_10a
#-   Time : Tue Jul 19 12:14:52 2011
#-   run_erc.pl -nontiming -auto_clock -noarchive -ruleset IO_UPART_Ruleset -noparasitic -view sn -fub c69p0snsdcufamilytopp -noexcl
```

CHAPTER 7 :

Concusion

After completion of all the above mentioned tasks, It can be concluded that :

1. **SOC Design Cycle Consists Of Below Steps:-**

    a) RTL Coding
    b) RTL Verification
    c) Synthesis
    d) Floor Planning
    e) CTS
    f) APR
    g) Functional Equivalence Verification
    h) Layout Versus Schematic Verification
    i) Density Verification
    j) Diode Checks.

2. In the Design Cycle there may arise a need of doing a small change if bug is spot in the mid of the Design Cycle. There is generally no need to re-spin the design but, we can go over for a process called **Engineering Change Order**, which eradicates the need of Design Re-Spin.

3. Time To Market is an essential thing to hit. Due to which the product may fail or flourish in the market. Performing the ECO is therefore very crucial and the necessary step in the SoC Design cycle.

Bibliography

[1] E. Seevinck, F. J. List, and J. Lohstroh, "Static-noise margin analysis of MOS SRAM cells," *IEEE J. Solid-State Circuits,* vol. SC-22, pp. 748-754, Oct. 1987.
[2] A. J. Bhavnagarwala, T. Xinghai, and J. D. Meindl, "The impact of intrinsic device fluctuations on CMOS SRAM cell stability," *IEEE J. Solid-State Circuits,* vol. 36, pp. 658-665, Apr. 2001.
[3] M. Yamaoka, "0.4-V Logic-Library-Friendly SRAM array using rectangular-diffusion cell and delta-boosted-array voltage scheme," *IEEE J. Solid-State Circuits,* vol. 39, pp. 934-940, June 2004.
[4] A. Hardi, B. Bhaumik, P. Pradhan, R. Varambally, " A Low Power 256 KB SRAM Design,"
[5] A. Turier, L. Ben Ammar, A. Amara, " Static Power Consumption Management in CMOS memories," © *2001 IEEE.*
[6] M. Lee, Wing-II Sze, Chii-ming M. Wu, "Static noise margin and soft-error rate simulations for thin film transistor cell stability in a 4 Mbit SRAM Design," © *1995 IEEE.*
[8] ST internal site
[9] Neil H. E. Weste, Kamran Eshraghian, *the system perspective* PRINCIPLES OF CMOS VLSI DESIGN, Second Edition.
[10] John F. Wakerly, Digital Design *Principles & Practices*, Third Edition.
[11] Sung-Mo Kang, Yusuf Leblebici, CMOS Digital Integrated Circuits *Analysis & Design*, Third Edition..
[12] Hazapis, O.G.; Manolakos, E.S.Scalable FRM-SSA SoC Design for the Simulation of Networks in Real Time
[13] Design Compiler User Guide, Synopsys Inc.
[14] Integrated Circuit Compiler User Guide.