CALIFORNIA STATE UNIVERSITY NORTHRIDGE

Asynchronous Interface, ASIC Flow (RTL-to-GDSII) using Cadence and Synopsys Tools

A graduate project submitted in partial fulfillment of the requirements
For the degree of Masters of Science
in Electrical Engineering

By

Tejas Raval

May 2012

The graduate project of Tejas Raval is approved:

Dr. Somnath Chattopadhyay	Date
Dr. Ali Amini	Date
Dr. Ramin Roosta, Chair	Date

California State University, Northridge

Acknowledgement

This project work would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this project.

First and foremost, my utmost gratitude to Dr. Ramin Roosta, Dr. Roosta has been my inspiration as I hurdle all the obstacles in the completion this project. Dr. Roosta's continuous guidance, support and resourcefulness in the field of ASIC/FPGA designing and verification helped to conquer every big and small milestone. Without his constant support and help this project would not have been materialized. It was an honor for me to work under his guidance.

I would like to show my gratitude to Dr. Ali Amini for being a mentor and for his continuous guidance during the full course of my graduate studies and also for his support as a member of project committee. I would like to thank Dr. Somnath Chattopadhyay for his valued attention, information and support.

Thanks are also due to Ms. Tanya, System Administrator, for being a patient listener to the technical complaints and issues regarding the lab soft-wares that our team faced during this project and for solving them at the earliest as possible. Special thanks to Dr. Dave Wilder, Synopsys lecturer, for training me and to clear my concepts in Backend Physical Design.

Most importantly, I like to thank my friends, my parents, Mr. Ashok Raval, Mrs. Parul Raval and my sister Mrs. Deepa Raval, for their constant support, prayers and wishes for my success and well-being.

Table of Contents

List of Figures

List of Tables

Abstract

Implementation of Asynchronous Interface ASIC Flow (RTL-to-GDSII) using Cadence and Synopsys Tools

By

Tejas Raval

Master of Science in Electrical Engineering

As the technology sizes of semiconductor devices continue to decrease, the effect of nanometer technologies on congestion, signal integrity, crosstalk etc. are becoming more significant. These all factors are affecting and forcing various technological methodologies throughout the design flow to constantly fight and keep updating the EDA tools to cop-up with these issues.

The aim of this project is to successfully complete ASIC design flow from RTL to GDS-II, using the advance industry level tools. This project provides a solid base and practical hands-on experience of advanced tools like Cadence NC Simulator (Behavioral Simulation and Post Synthesis Simulation), Synopsys Design Compiler (Logic Synthesis) , Synopsys DFT Compiler (Logic Scan Insertion and Boundary Scan Insertion), Synopsys Power Compiler (Power Optimization using clock gating), Synopsys Tetra Max (Determine Fault Coverage) and Synopsys IC Compiler (Design planning, Power Network Synthesis, Clock Tree Synthesis, Place and Route and Chip Finishing). Along with this, the analysis of various design factors affecting the performance of the final chip such as power, area and timing is also performed.

CHAPTER 1: ASIC Design Flow

1.1 Introduction to ASIC

An application-specific integrated circuit (ASIC) is an integrated circuit(IC) customized for a particular use, rather than intended for general-purpose use. In today's world, ASICs offer many advantages over off-the-shelf devices.

1. Smaller die size leads to board size reduction
2. Reduced power consumption, less heat dissipation
3. Lower costs under mass production
4. Improved performance
5. Better radiation tolerance
6. Improved testability
7. Enhanced reliability
8. Proprietary design implementation

1.2 Standard-Cell–Based ASIC

A cell-based ASIC uses predefined logic cells like AND gates, OR gates, multiplexers, and flip-flops known as standard cells. The flexible blocks in a CBIC are built of rows of standard cells. Placement of the standard cells and the interconnect is defined by an ASIC designer in a CBIC. The advantage of CBICs is that they can be designed in less time with small amount of money compared to full-custom ASICs, and also the most important thing is it reduce the risk by using a predesigned, pretested, and pre-characterized standard-cell library which can be optimized individually. At the same time, the disadvantages are the time or expense of designing or buying the standard-cell library and the time needed to fabricate all layers of the ASIC for each new design. Figure-1.1 shows a CBIC.

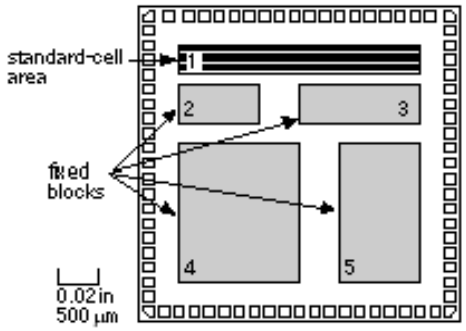

Figure: 1.1- Cell based ASIC (CBIC) [1]

Each standard cell in the library is constructed using full-custom design methods, but you can use these predesigned and pre-characterized circuits without having to do any full-custom design yourself. This design style gives you the same performance and a flexibility advantage of a full-custom ASIC but reduces design time and reduces risk. [1]

1.3 Need for Low Power ASIC

For early digital circuits, high speed and minimum area were the main design constraints. Most of the EDA tools were designed specifically to meet these criteria. Power consumption was never highly visible. Nowadays, the area reduction of digital circuits is no longer a big issue as with the latest sub-micron techniques, many millions of transistors can be fit in a single IC. Smaller chip size eventually leads to high demand for portable and handheld devices. More and more applications are battery powered, and low power IC's are the key to extend the usage time in between battery recharge, and in turn increase battery life and reliability of the product. Also in submicron technologies, there is a limitation on the proper functioning of circuits due to heat generated by power dissipation. Market forces are demanding low power for not only longer battery life but also reliability, portability, performance, cost and time to market. This is very true in the field of personal computing devices, wireless communications systems, home entertainment systems, which are becoming popular now-a-days. Implantable medical devices, such as pace maker, deep brain system for Parkinson's disease, and spinal cord stimulator for pain management, particularly need to dissipate less power for longer battery life and improved component reliability and safety.

As process technology reduces into 90nm and below, performance and density are taken to new levels, yet power loss in both switching and leakage makes designing with these devices a major challenge. Leakage power reduction is essential in sustaining the scaling of the CMOS process. Leakage power is now becoming proportional to dynamic or

switching power loss as shown in Figure below. While lowering of the threshold voltage leads to significant increase in sub-threshold leakage current, the increase in gate tunneling leakage current is caused by thinner gate oxides. While scaling improves transistor density, functionality, and higher performance on chip, it also results in power dissipation increase. Therefore, it has become necessary to use new techniques to manage energy at the system level.

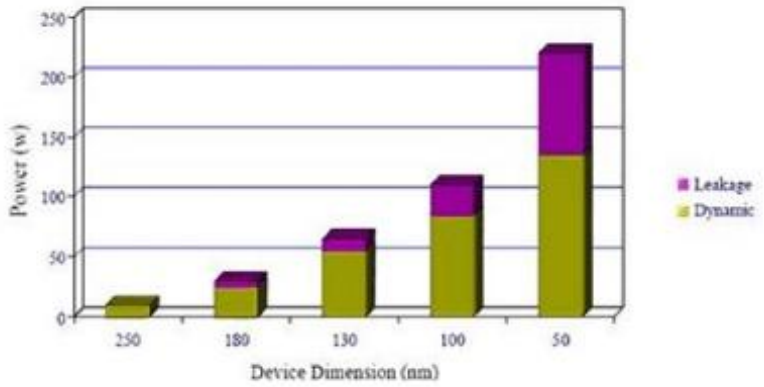

Figure: 1.2- Dynamic and Leakage Power Comparison [2]

Bottom line, low power budget has become one of the most important design parameters for VLSI (Very Large Scale Integration) systems.

1.3.1 ASIC Flow:

The traditional ASIC design flow contains the steps outlined below:

i. Prepare requirement specification and create a Micro-Architecture document.
ii. RTL design and development of IP's.
iii. After the previous step DFT memory BIST insertion can also be implemented, if the design contains any memory element.
iv. Functional verification all the IPS. Check whether the RTL is free from linting errors and analyze whether the RTL is synthesis friendly.
    a. Perform cycle based verification (functional) to verify the protocol behavior of the RTL.
    b. Perform the property checking to verify the RTL implementation and the specification understanding is matching.

v. Design environment setting. This includes the technology file to be used along with other environmental attributes.

vi. Prepare the design constraints file to perform synthesis, usually called as an SDC synopsys_constraints or dc_synopsys_setup file, specific to synthesis tool (design compiler).

    a. Once the constraints file is set. For performing synthesis inputs to the DC are the library file (for which the synthesis needs to be targeted for, which has the functional/timing information available for the standard cell library and the wire load models for the wires based on the fan-out length of the connectivity), RTL files and the design constraints files, so that the synthesis tool can perform the synthesis of the RTL files and map and optimize to meet the design constraints requirements. After performing the synthesis, scan insertion and JTAG scan chain insertions are implemented and then synthesis is repeated.

vii. Check whether the design is meeting the requirements after synthesis. Perform block level static timing analysis using Design compiler's built-in static timing analysis engine.

viii. Perform Formal verification between RTL and the synthesized netlist to confirm that the synthesis tool has not altered the functionality.

ix. Perform the pre-layout STA (static timing analysis) using PrimeTime with the SDF (standard delay format) file and synthesized netlist file to check whether the design is meeting the timing requirements.

x. Once the synthesis is performed the synthesized netlist file (VHDL/Verilog format) and the SDC (constraints file) is passed as input files to the Placement and routing tool to perform the back-end activities. The tool used is IC Compiler. [3]

xi. Initialize the floorplanning with timing driven placement of cells, clock tree insertion and global routing.

xii. Transfer of clock tree to the original design (netlist) residing in Design Compiler.

xiii. In-place optimization of the design in Design Compiler.

xiv. Formal verification between the synthesized netlist and clock tree inserted netlist, using Formality.

xv. Extraction of estimated timing delays from the layout after the global routing step.
xvi. Back annotation of estimated timing data from the global routed design, to PrimeTime.
xvii. Static timing analysis in PrimeTime, using the estimated delays extracted after performing global route.
xviii. Detailed routing of the design.
xix. Extraction of real timing delays from the detailed routed design.
xx. Back annotation of the real extracted timing data to PrimeTime.
xxi. Post-layout static timing analysis using PrimeTime.
xxii. Functional gate-level simulation of the design with post-layout timing (if desired).
xxiii. Tape out after LVS and DRC verification.[4]

CAD tools are involved in all stages of VLSI design flow–Different tools can be used at different stages due to EDA common data formats. CAD tools provide several advantages:

- Ability to evaluate complex conditions in which solving one problem creates other problems
- Use analytical methods to assess the cost of a decision
- Use synthesis methods to help provide a solution
- Allows the process of proposing and analyzing solutions to occur at the same time

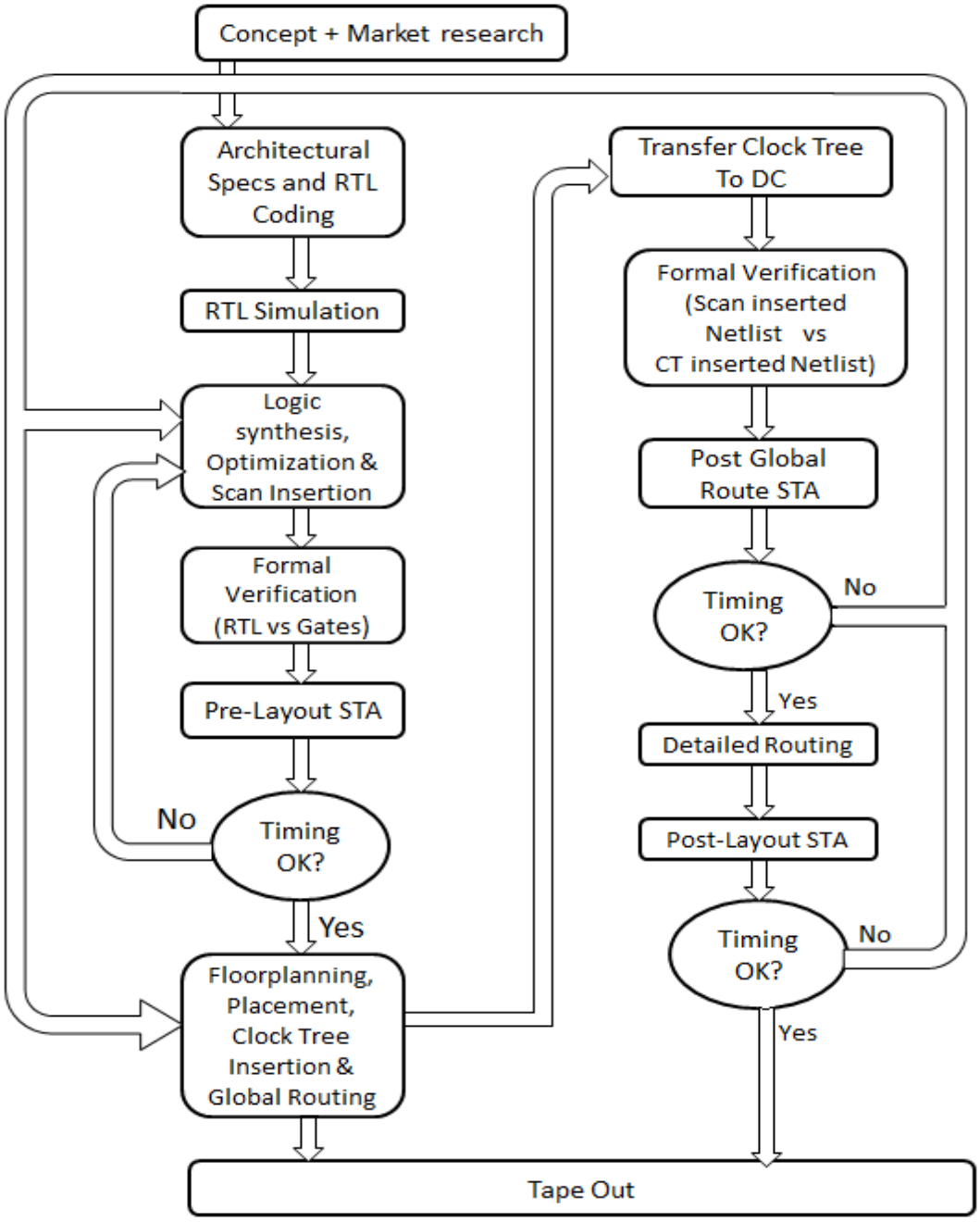

Figure 1.3 Traditional ASIC Design Flow [4]

Figure 1.3 graphically illustrates the typical ASIC design flow discussed above. The acronyms STA and CT represent static timing analysis and clock tree respectively. DC represents Design Compiler Synopsys CAD tool for Physical Design is called Integrated Circuit Compiler (ICC). [4]

CHAPTER 2: Asynchronous Interface Overview

2.1 Introduction: Asynchronous Interface

Asynchronous interface design is the circuitry in which set of signals that comprises the connection between devices of a computer system where the transfer of information between devices is organized by the exchange of signals not synchronized to some controlling clock. A request signal from an initiating device indicates the requirement to make a transfer; an acknowledging signal from the responding device indicates the transfer completion. This asynchronous interchange is also widely known as Handshaking. [5]

Most of the time, asynchronous designs are referred to as the designs with no clocks, but this project asynchronous FIFO interface circuit incorporates multiple clocks for transmitting and receiving the data values. The description of the design is explained below along with the top module diagram of the design.

An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer (RAM) from one clock domain and the data values are read from the same FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other. Asynchronous FIFOs are used to safely pass data from one clock domain to another clock domain. [6]

There are a lot of different ways to design asynchronous FIFO interface design, the method used in this project is “FIFO partitioning with synchronized pointer comparison”; for comparing and synchronizing the design working on two clocks one for transmitting and one for receiving, uses gray counters for comparison of full and empty registers of RAM which is FIFO buffer for writing and reading the data values.

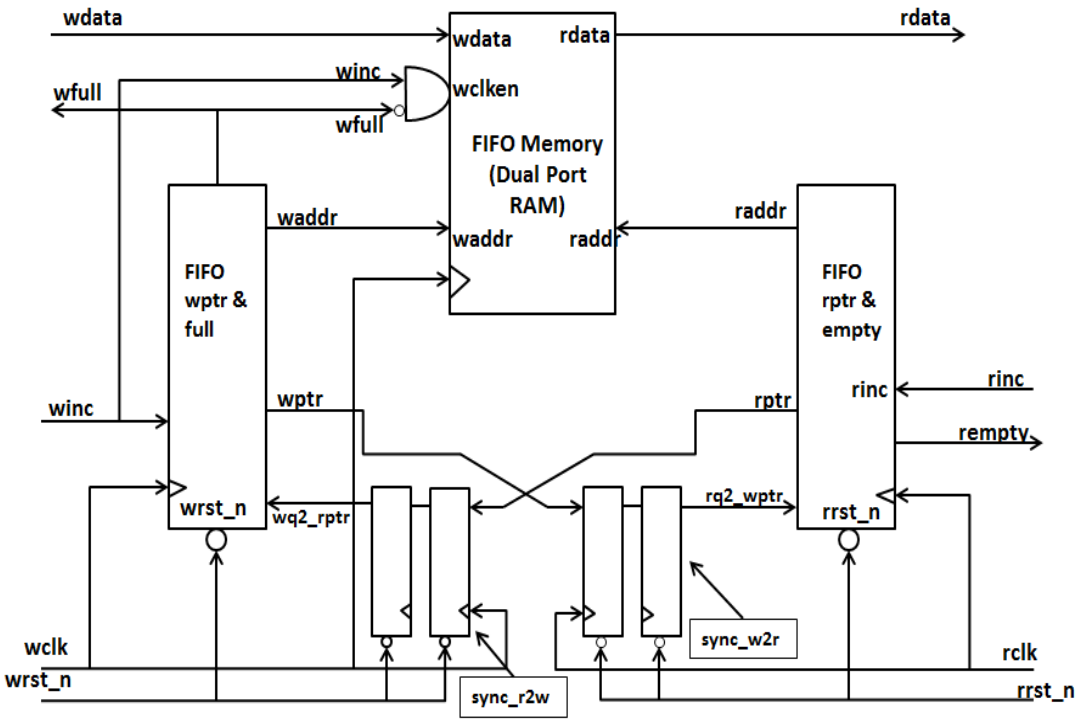

Figure 2.1: Internal Block Diagram of FIFO partitioning with synchronized pointer comparison [6]

Data words are placed into a FIFO buffer memory array by control signals in one clock domain, and the data words are removed from another port of the same FIFO buffer memory array by control signals from a second clock domain. The difficulty associated with doing FIFO design is related to generating the FIFO pointers and finding a reliable way to determine full and empty status on the FIFO. [6]

Generally FIFOs are used where write operation is faster than read operation. However, even with the different speed and access types the average rate of data transfer remains constant. FIFO pointers keep track of number of FIFO memory locations read and written and corresponding control logic circuit prevents FIFO from either under flowing or overflowing. FIFO architectures inherently have a challenge of synchronizing itself with the pointer logic of other clock domain and control the read and write operation of FIFO memory locations safely. [7]

2.2 Issues in Designing Asynchronous FIFO

Although the design states that the circuitry is asynchronous and is working in multiclock environment, it is essential to synchronize the two clocks as the data can be lost due to setup and hold violations. It is very important to understand the signal stability in multi clock domains since for a traveling signal the new clock domain appears to be asynchronous. If the signal is not synchronized to new clock, the first storage element of the new clock domain may go to metastable state and the worst case is that resolution time cannot be predicted. It can traverse throughout the new clock domain resulting in failure of functionality. To prevent such failures setup time and hold time specification has to be obeyed in the design. Manufacturers provide statistics of probability of failure of flip-flops due to metastability characters in terms of MTBF (Mean Time before Failure). Synchronizers are used to prevent the downstream logic from entering into the metastable state in multiclock domain with multibit data values.

Thus, for efficient working of FIFO architecture designing of FIFO pointers is the key issue. At this point, deep understandings of the FIFO read and write pointers become necessary. On reset both read and write pointers are pointing to the starting location of the FIFO. This location is also the first location where data has to be written at the same time this first location happens to be first read location. Therefore, in general, read pointer always points to the word to be read and write pointer always points to the next location to which data has to be written. [8]

2.3 Operation of the Design

2.3.1 Data write operation:

When both read and write pointers are pointing to first location of FIFO empty flag is asserted indicating the FIFO status as empty. Now data writing can be performed. Data will be written to the location where the write pointer is pointing and after the data write operation write pointer gets incremented pointing to the next location to be written. At the same time, empty flag is de-asserted which indicates that FIFO is not empty, some

data is available. One notable point regarding read pointer is with empty flag active the data pointed out by the read pointer is always invalid data. When first data written and empty flag status cleared (i.e. empty flag inactive) read pointer logic immediately drives the data from the location to which it was pointing to the read port of the dual port RAM, ready to be read by read logic. With this implementation of read logic the biggest advantage is that only one clock pulse is required to read from read port since previous clock cycle has already incremented read pointer and drives the data to read port. This will help in reducing latency in detecting empty and full pointer flag status. Empty status flag can be asserted in one more condition. After some n number of data write operations if same n number of read is performed then both pointers are again equal. Hence, if both pointers "catch up" each other, then empty flag is asserted. [9].

2.3.2 FIFO full status:

When write pointer reaches the top of the FIFO, it is pointing towards the location, which can be written and is the last location to be written. No read operation is performed yet and read pointer is pointing to first location itself. This is one method is to generate FIFO full condition. When write pointer reaches the top of the FIFO, if full flag is asserted then it is not the actual FIFO full condition, this is only 'almost full' as there is one location which can be written. Similarly almost empty condition can exist in FIFO. Now a write operation causes the location to be written and increment of write pointer. Since the location was the last one write pointer wraps up to first location. Now both read and write pointers are equal and hence empty flag is asserted instead of full flag assertion, which is a fatal mistake. Hence wrap around condition of a full pointer may be a FIFO full condition.

After writing the data to FIFO (consider write pointer is in top of FIFO) some data has been read and read pointer is somewhere in between FIFO. One more write operation causes the write pointer to wrap. Note that even though write pointer is pointing to first location of FIFO this is NOT FIFO full condition, since read pointer has moved up from the first location. Further data writing pushes write pointer up. Imagine read pointer

wraps around after some more read operation. Present condition is that both pointers have wrapped around but there is no FIFO full or FIFO empty condition. Data can be written to FIFO or read from the FIFO. [6]. The disadvantage of a FIFO of this kind is that the status signals cannot be fully synchronized with the read and write clock. [9].

2.3.3 Asynchronous FIFO pointers:

FIFO is full when the pointers are equal, that is, when the write pointer has wrapped around and caught up to the read pointer. This is a problem. Considering that point, it is difficult to decide which condition has occurred; the FIFO is either empty or full when the pointers are equal.

One design technique used to distinguish between full and empty is to add an extra bit to each pointer. Whenever the write pointer increments past the final FIFO address, the write pointer will increment the unused MSB while setting the rest of the bits back to zero as shown in Figure below (the FIFO has wrapped and toggled the pointer MSB). The same is done with the read pointer. If the MSBs of the two pointers are different, it means that the write pointer has wrapped one more time that the read pointer. If the MSBs of the two pointers are the same, it means that both pointers have wrapped the same number of times. [6]

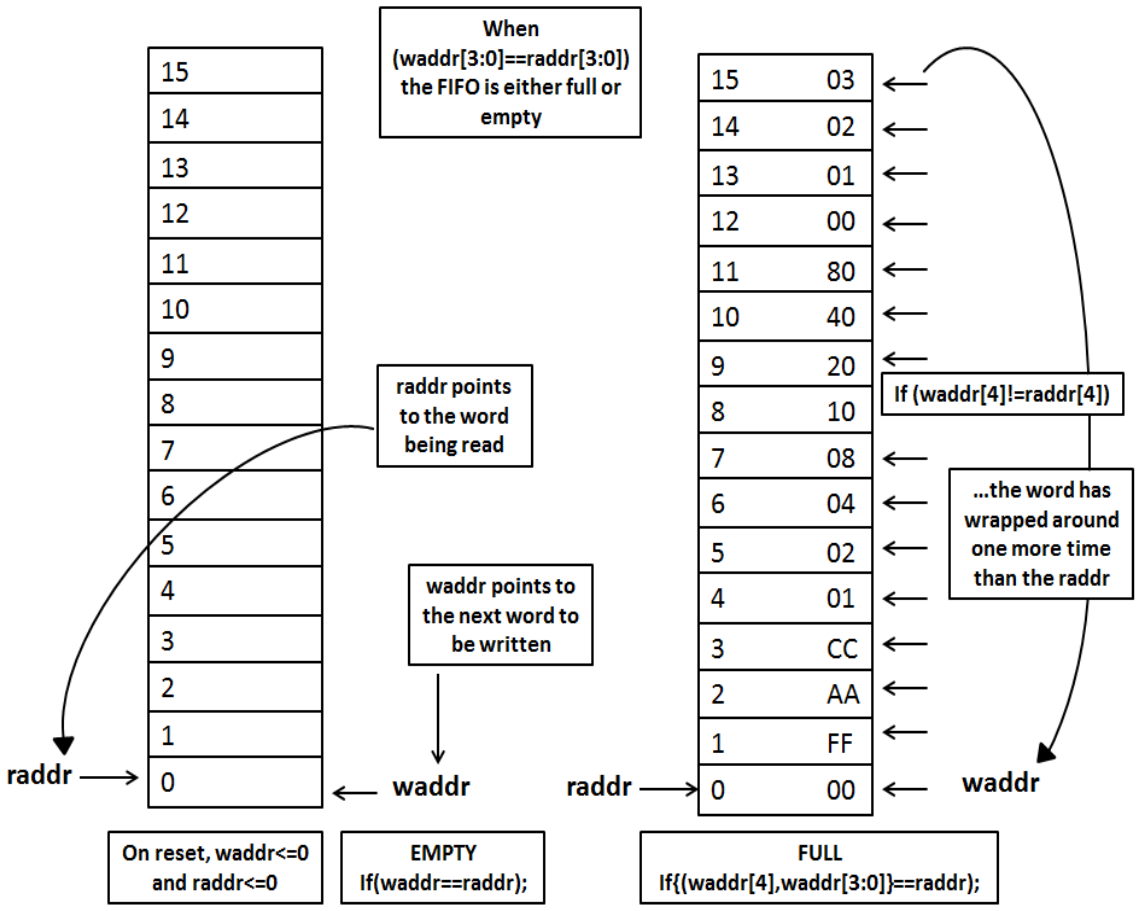

Figure2.2: FIFO full and empty conditions [6]

Using n-bit pointers where (n-1) is the number of address bits required to access the entire FIFO memory buffer; the FIFO is empty when both pointers, including the MSBs are equal. And the FIFO is full when both pointers, except the MSBs are equal. The FIFO design uses n-bit pointers for a FIFO with 2(n-1) write-able locations to help handle full and empty conditions.

The counters designed to synchronize the signals are Gray code counters. The reason to choose gray coder counter and not the binary code counter is that, trying to synchronize a binary count value from one clock domain to another is problematic because every bit of an n-bit counter can change simultaneously (example 7->8 in binary numbers is 0111->1000, all bits changed). Gray codes only allow one bit to change for each clock transition, eliminating the problem associated with trying to synchronize multiple

changing signals on the same clock edge. It is desirable to create both an n-bit Gray code counter and an (n-1)-bit Gray code counter. It would certainly be easy to create the two counters separately, but it is also easy and efficient to create a common n-bit Gray code counter and then modify the 2nd MSB to form an (n-1)-bit Gray code counter with shared LSBs. This will be called a "dual n-bit Gray code counter." [6].

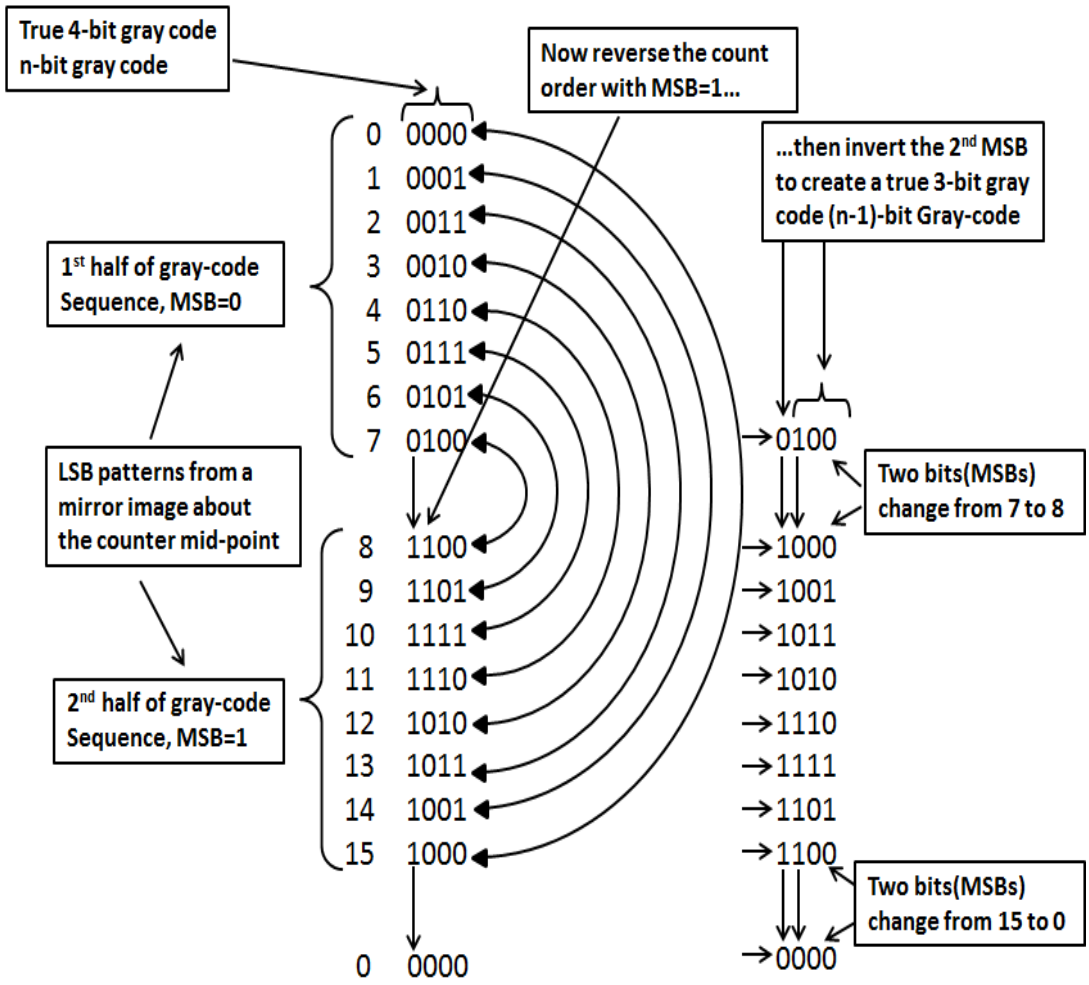

Figure 2.3: n-bit Gray code converted to an (n-1)-bit Gray code [6]

It is obvious that inverting the second MSB of the second half of the 4-bit Gray code will produce the desired 3-bit Gray code sequence in the three LSBs of the 4-bit sequence. The only other problem is that the 3-bit Gray code with extra MSB is no longer a true Gray code because when the sequence changes from 7 (Gray 0100) to 8 (~Gray 1000)

and again from 15 (~Gray 1100) to 0 (Gray 0000), two bits are changing instead of just one bit. A true Gray code only changes one bit between counts.

2.4 Handling full and empty conditions

Exactly how FIFO full and FIFO empty are implemented is design-dependent. The FIFO design in this paper assumes that the empty flag will be generated in the read-clock domain to insure that the empty flag is detected immediately when the FIFO buffer is empty, that is, the instant that the read pointer catches up to the write pointer (including the pointer MSBs).The FIFO design in this paper assumes that the full flag will be generated in the write-clock domain to insure that the full flag is detected immediately when the FIFO buffer is full, that is, the instant that the write pointer catches up to the read pointer (except for different pointer MSBs).

2.4.1 Generating empty flag

The FIFO is empty when the read pointer and the synchronized write pointer are equal. The empty comparison is simple to do. Pointers that are one bit larger than needed to address the FIFO memory buffer are used. If the extra bits of both pointers (the MSBs of the pointers) are equal, the pointers have wrapped the same number of times and if the rest of the read pointer equals the synchronized write pointer, the FIFO is empty. The Gray code write pointer must be synchronized into the read-clock domain through a pair of synchronizer registers found in the sync_w2r module. Since only one bit changes at a time using a Gray code pointer, there is no problem synchronizing multi-bit transitions between clock domains. In order to efficiently register the rempty output, the synchronized write pointer is actually compared against the rgraynext (the next Gray code that will be registered into the rptr). The empty value testing and the accompanying sequential always block has been extracted from the rptr_empty.v

2.4.2 Generating full flag

Since the full flag is generated in the write-clock domain by running a comparison between the write and read pointers, one safe technique for doing FIFO design requires that the read pointer be synchronized into the write clock domain before doing pointer comparison. The full comparison is not as simple to do as the empty comparison. Pointers that are one bit larger than needed to address the FIFO memory buffer are still used for the comparison, but simply using Gray code counters with an extra bit to do the comparison is not valid to determine the full condition. [6]

2.5 Procedure to Design FIFO Module:

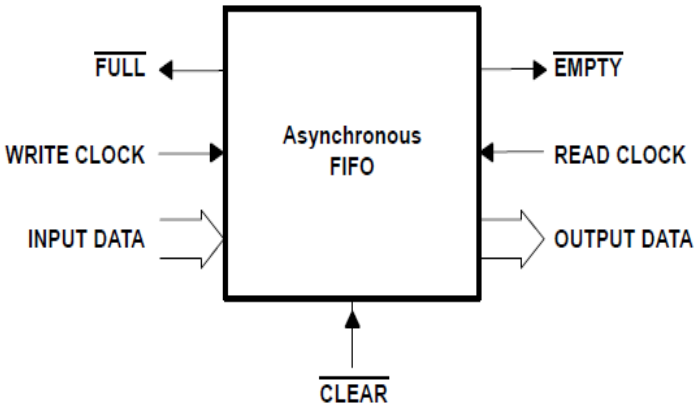

Figure 2.4: Top Module of asynchronous fifo [9]

In order to perform FIFO full and FIFO empty tests using this FIFO style, the read and write pointers must be passed to the opposite clock domain for pointer comparison.

- Create a Verilog model fifo_model.v which asserts full_flag and empty_flag if the fifo memory is full or empty.
- Use parameter for width of the data and keep default to 16 bits.
- Use parameter for width of write and read pointer. We have to keep write pointer and read pointer width one bit more than width of address for flag comparision.
- Make wr_pointer and rd_pointer 6-bit and wr_address and rd_pointer 5-bit.
- Create an always block where we write data into fifo memory at wr_clk and increment wr_pointer if wr_enable and full_flag are low.

- Now create an always block where we read data from fifo memory at rd_clk and increment rd_pointer if rd_enable and empty_flag are low.
- Pass rd_pointer through 2 stage synchronizer flip-flops which run on wr_clk. Pass wr_pointer through 2 stage synchronizer flip-flops which run on rd_clk. Before passing this pointer through 2 flip flops they have to be converted into gray-code because only one bit should change at a given point of time between consecutive pointer values. Rise and fall times are different so 0 to 1 transition and 1 to 0 transition never occur at same time.
- After passing through the 2-stage synchronizers read pointer and write pointer are converted back into binary and their values are compared to generate full flag and empty flag. Converting back into binary result into slow design than comparing the pointer values by modifying the gray code. Gray code can be modified by modifying MSB-1 bit, by using exclusive or gate between the MSB and MSB-1 bit. Keeping only one exclusive or gate can result into faster design.
- We convert back into binary and compare the pointers. We following use the logic for generating flags:

```
assign depth_rd = modified_rdclk_wrpointer-rd_pointer;
assign depth_wr = wr_pointer-modified_wrclk_rdpointer;
assign empty_flag  = (depth_rd == 6'b000000 ) ? 1'b1 : 1'b0;
assign full_flag = (depth_wr == 6'b100000 ) ? 1'b1 :1'b0;
```

- Then we create a test fixture tb_fifo.v to test fifo_model.v. We give values to wr_data at wr_clk. We use a counter to write data. Data write starts from 0 and keeps writing 0,1,2,3… if wr_rst is high and wr_enable(put) is low. We keep rd_rst high and rd_enable(get) low to read data. Fifo buffers all asynchronous data and data is not lost if both enables are low.
- To check for flags generation keep wr_enable high empty flag is asserted after all data is read. Full flag is asserted if rd_enable is high after some time.

CHAPTER 3: SYNTHESIS USING SYNOPSYS DESIGN COMPILER

3.1 Synthesis and its Basic Flow

Synthesis is the process that generates a gate-level netlist for an IC design that has been defined using a Hardware Description Language (HDL). Synthesis includes reading the HDL source code and optimizing the design from that description. Using the technology library's cell logical view, the Logic Synthesis tool performs the process of mathematically transforming the ASIC's register-transfer level (RTL) description into a technology-dependent netlist. This process is similar to a software compiler converting a high-level C-program listing into a processor-dependent assembly-language listing. The netlist is the standard-cell representation of the ASIC design, at the logical view level. It consists of instances of the standard-cell library gates, and port connectivity between gates. Proper synthesis techniques ensure mathematical equivalency between the synthesized netlist and original RTL description. The netlist contains no unmapped RTL statements and declarations.

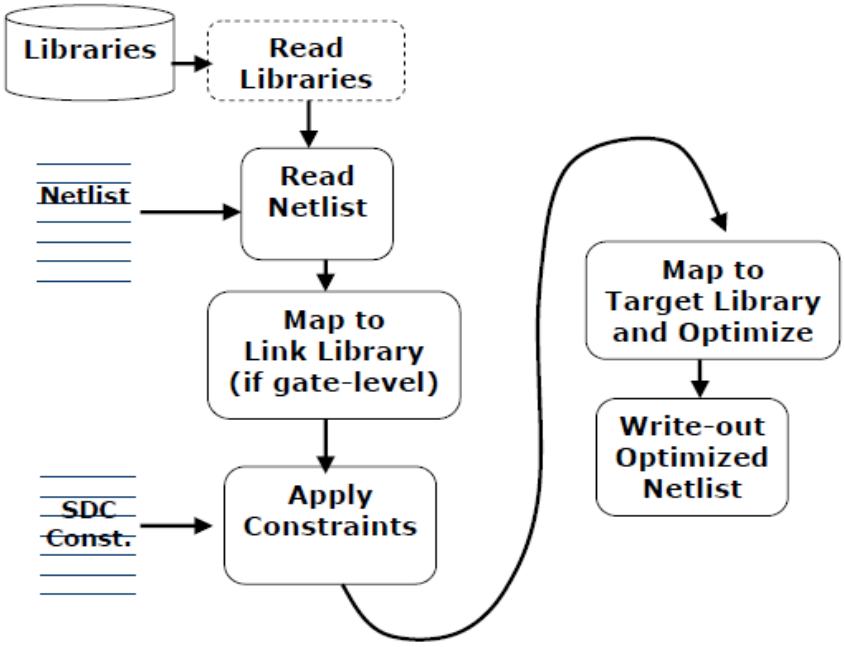

Figure 3.1: The Basic Synthesis flow [10]

3.2 Synopsys Design Compiler Flow for Synthesis

The Design Compiler is a synthesis tool from Synopsys Inc. In simple terms, synthesis tool takes a RTL [Register Transfer Logic] hardware description written in either Verilog or VHDL and standard cell library as input and the resulting output would be a technology dependent gatelevel-netlist. The gatelevel-netlist is nothing but structural representation of only standard cells based on the cells in the standard cell library. The synthesis tool internally performs many steps, which are listed below. Also below is the flowchart of synthesis process.

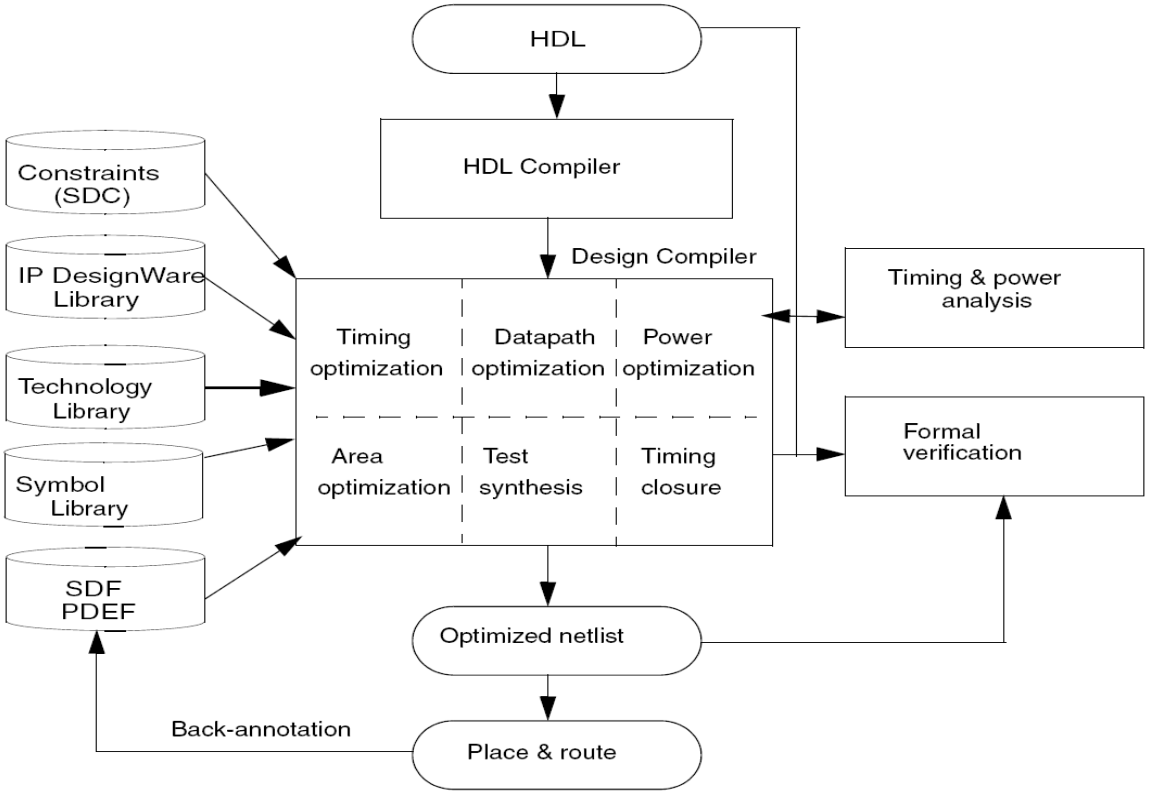

Figure 3.2: Design Compiler Flow [11]

1. Design Compiler reads in technology libraries, DesignWare libraries, and symbol libraries to implement synthesis. During the synthesis process, Design Compiler [DC] translates the RTL description to components extracted from the technology library and DesignWare library. The technology library consists of basic logic gates and flip-flops.

The DesignWare library contains more complex cells for example adders and comparators which can be used for arithmetic building blocks. DC can automatically determine when to use Design Ware components and it can then efficiently synthesize these components into gate-level implementations.

2. Design Compiler also needs the RTL designed by the designer. It reads the RTL hardware description written in either Verilog/VHDL.

3. The synthesis tool now performs many steps including high-level RTL optimization, RTL to un-optimized Boolean logic, technology independent optimizations, and finally technology mapping to the available standard cells in the technology library, known as target library. This resulting gate-level-netlist also depends on constrains given. Constraints are the designer's specification of timing and environmental restrictions [area, power, process etc] under which synthesis is to be performed. As an RTL designer, it is good to understand the target standard cell library, so that one can get a better understanding of how the RTL coded will be synthesized into gates.

4. After the design is optimized, it is ready for DFT [design for test/ test synthesis]. DFT is test logic; designers can integrate DFT into design during synthesis. This helps the designer to test for issues early in the design cycle and also can be used for debugging process after the chip comes back from fabrication.

5. After test synthesis, the design is ready for the place and route tools. The Place and route tools place and physically interconnect cells in the design. Based on the physical routing, the designer can back-annotate the design with actual interconnect delays; DC can be used again to resynthesize the design for more accurate timing analysis. [10]

While running DC, it is important to monitor/check the log files, reports, scripts etc to identity issues which might affect the area, power and performance of the design.

3.3 Design Flow

3.3.1 Read Design

Design Compiler reads designs into memory from design files. Many designs can be in memory at any time. After a design is read in, you can change it in numerous ways, such as grouping or ungrouping its sub designs or changing sub design references. Design Compiler provides the following ways to read design files:

• The analyze and elaborate commands
• The read_file command

Using the analyze and elaborate Commands

The analyze command does the following:

• Reads an HDL source file
• Checks it for errors (without building generic logic for the design)
• Creates HDL library objects in an HDL-independent intermediate format
• Stores the intermediate files in a location you define

If the analyze command reports errors, fix them in the HDL source file and run analyze gain. After a design is analyzed, you must reanalyze it only when you change it.

The elaborate command does the following:

•Translates the design into a technology-independent design (GTECH) from the intermediate files produced during analysis.
• Allows changing of parameter values defined in the source code.
• Allows VHDL architecture selection.
• Replaces the HDL arithmetic operators in the code with DesignWare components.
• Automatically executes the link command, which resolves design references.

Resolving the reference means that the design library or file containing the detailed design data for the sub-block can be found and processed. If any references in the netlist cannot be resolved, the link command will issue warnings as to which sub-component designs are not available.

3.3.2 Define Design Environment

In order to obtain optimum results from DC, designers have to methodically constrain their designs by describing the design environment, target objectives and design rules. The constraints may contain timing and/or area information, usually derived from design specifications. DC uses these constraints to perform synthesis and tries to optimize the design with the aim of meeting target objectives. You define the environment by specifying operating conditions, wire load models, and system interface characteristics. Operating conditions include temperature, voltage, and process variations. Wire load models estimate the effect of wire length on design performance. System interface characteristics include input drives, input and output loads, and fan-out loads. The environment model directly affects design synthesis results. [11]

- set_operating_conditions describes the process, voltage and temperature conditions of the design. The Synopsys library contains the description of these conditions, usually described as WORST, TYPICAL and BEST case. The names of operating conditions are library dependent. Users should check with their library vendor for correct setting. By changing the value of the operating condition command, full ranges of process variations are covered. The WORST case operating condition is generally used during pre-layout synthesis phase, thereby optimizing the design for maximum setup-time. The BEST case condition is commonly used to fix the hold-time violations. The TYPICAL case is mostly ignored, since analysis at WORST and BEST case also covers the TYPICAL case.

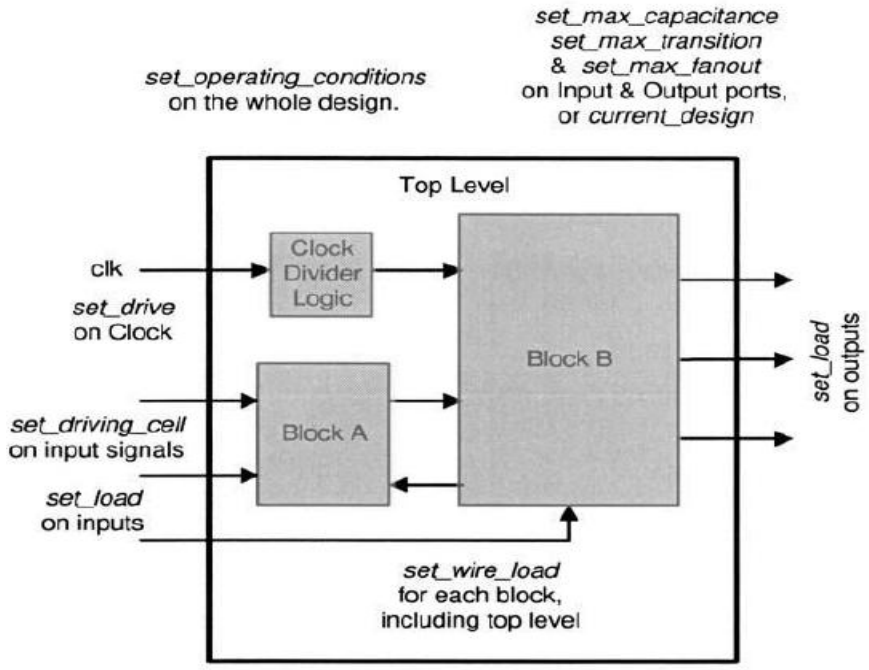

Figure 3.3: Basic Design Environment [11]

set_operating_conditions <name of operating conditions>

It is possible to optimize the design both with the WORST and the BEST case, simultaneously. The optimization is achieved by using the –min and –max options in the above command, as illustrated below. This is very useful for fixing the design for possible hold-time violations. [4]

dc_shell> set_operating_conditions–min BEST –max WORST

- **set_wire_load_model** command is used to provide estimated statistical wire-load information to DC, which in turn, uses the wire-load information to model net delays as a function of loading. Generally, a number of wire-load models are present in the Synopsys technology library, each representing a particular size block. In addition, designers may also choose to create their own custom wire-load models to accurately model the net loading of their blocks.

set_wire_load_model -name<wire-load model>

- set_drive and **set_driving_cell** are used at the input ports of the block. set_drive command is used to specify the drive strength at the input port. It is typically used to model the external drive resistance to the ports of the block or chip. The value of 0 signifies highest drive strength and is commonly utilized for clock ports. Conversely, set_driving_cell is used to model the drive resistance of the driving cell to the input ports. This command takes the name of the driving cell as its argument and applies all design rule constraints of the driving cell to the input ports of the block.

set_drive <value> <object list>

set_driving_cell –cell <cell name>

–pin <pin name> <object list>

- Design Rule Constraints or DRCs consist of **set_max_transition**, **set_max_fanout** and **set_max_capacitance** commands. These rules are generally set in the technology library and are determined by the process parameters. These rules should not be violated in order to achieve working silicon. The DRC commands can be applied to input ports, output ports or on the current_design. Furthermore, if the value set in the technology library is not adequate or is too optimistic, then these commands may also be used at the command line, to control the buffering in the design.

set_max_transition <value> <object list>

set_max_capacitance <value> <object list>

set_max_fanout <value> <object list>

3.3.3 Design Constraints

Design constraints describe the goals for the design. They may consist of timing or area constraints. Depending on how the design is constrained, DC tries to meet the set objectives. It is imperative that designers specify realistic constraints, since unrealistic specification results in excess area, increased power and/or degradation in timing. The basic commands to constrain a design are shown in Figure 3.4. [4]

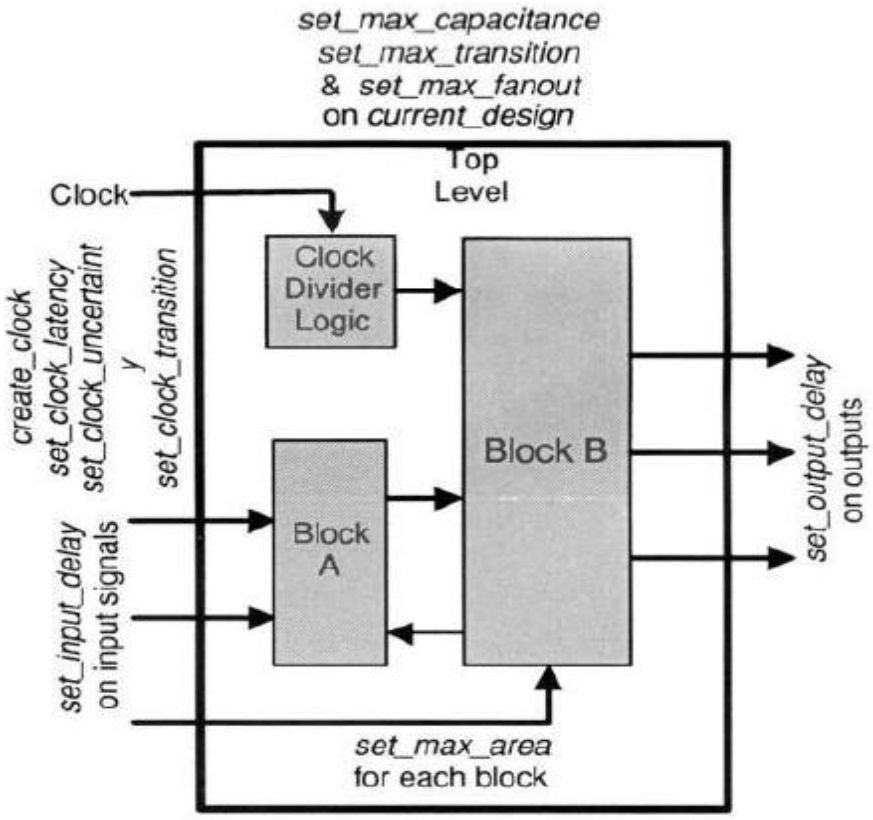

Figure 3.4: Design Constraints for synthesis [11]

There are two categories of design constraints:

• Design rule constraints

• Design optimization constraints

Design rule constraints are supplied in the technology library we specify. They are referred to as the implicit design rules. These rules are established by the library vendor, and, for the proper functioning of the fabricated circuit, they must not be violated. We can, however, specify stricter design rules if appropriate. The rules you specify are referred to as the explicit design rules.

Design optimization constraints define timing and area optimization goals for Design Compiler. These constraints are user-specified. Design Compiler optimizes the synthesis of the design, in accordance with these constraints, but not at the expense of the design

rule constraints. That is, Design Compiler attempts never to violate the higher-priority design rules. [4]

- **create_clock** command is used to define a clock object with a particular period and waveform. The –period option defines the clock period, while the –waveform option controls the duty cycle and the starting edge of the clock. This command is applied to a pin or port, object types.

In some cases, a block may only contain combinational logic. To define delay constraints for this block, one can create a virtual clock and specify the input and output delays in relation to the virtual clock. To create a virtual clock, designers may replace the port name (CLK, in the above example) with the –name <virtual clock name>, in the above command. Alternatively, one can use the set_max_delay or set_min_delay commands to constrain such blocks.

-create_generated_clock command is used for clocks that are generated internal to the design. This command may be used to describe frequency divided/multiplied clocks as a function of the primary clock.

```
create_generated_clock –name <clock name>
                       –source <clock source>
                       –divide_by <factor> | –multiply_by <factor>
                        …….
```

- **set_dont_touch** is used to set a dont_touch property on the current_design, cells, references or nets. This command is frequently used during hierarchical compilation of the blocks. Also, it can be used for, preventing DC from inferring certain types of cells present in the technology library.

-set_input_delay specifies the input arrival time of a signal in relation to the clock. It is used at the input ports to specify the time it takes for the data to be stable after the clock edge. The timing specification of the design usually contains this information, as the

setup/hold time requirements for input signals. Given the top-level timing specification of the design, this information may also be extracted for the sub-blocks of the design.

In Figure 3.5, the maximum input delay constraint of 23ns and the minimum input delay constraint of 0ns is specified for the signal *datain* with respect to the clock signal *CLK*, with a 50% duty cycle and a period of 30ns. In other words the setup-time requirement for the input signal *datain* is 7ns, while the hold-time requirement is 0ns.

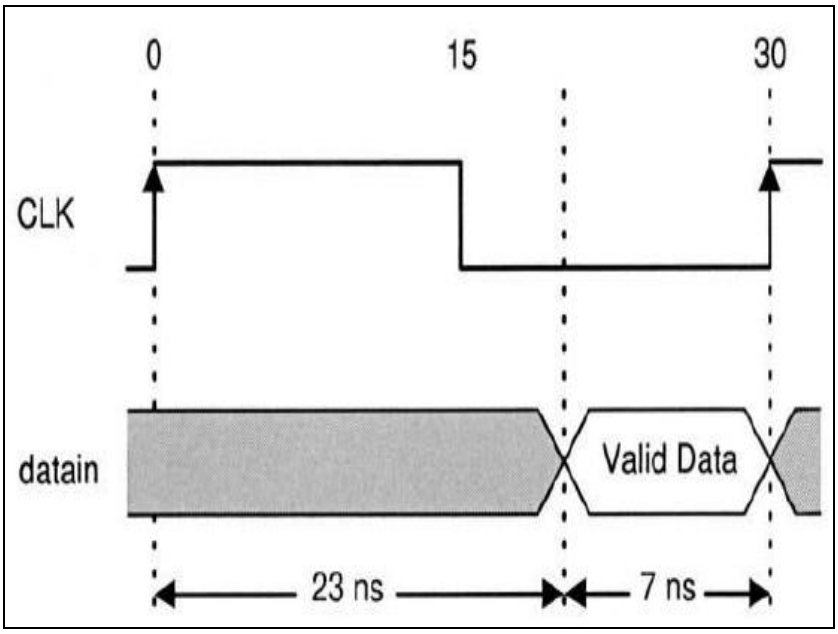

Figure 3.5: Specification of the Input Delay [11]

- set_output_delay command is used at the output port to define the time it takes for the data to be available before the clock edge. The timing specification of the design usually contains this information. Given the top-level timing specification of the design, this information may also be extracted for the sub-blocks of the design.

In Figure 3.6, the output delay constraint of 19ns is specified for the signal *dataout* with respect to the clock signal *CLK*, with a 50% duty cycle and a period of 30ns. This means that the data is valid for 11ns after the clock edge.

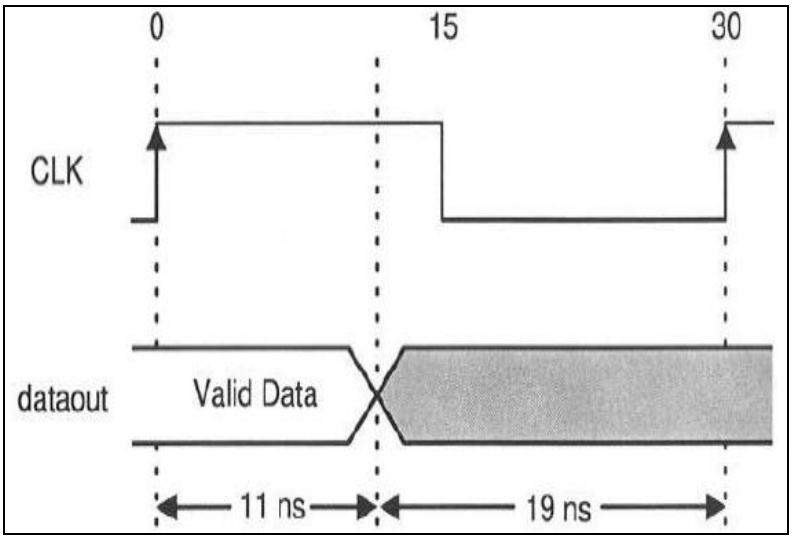

Figure 3.6: Specification of the Output Delay [11]

- set_clock_latency command is used to define the estimated clock insertion delay during synthesis. This is primarily used during the prelayout synthesis and timing analysis. The estimated delay number is an approximation of the delay produced by the clock tree network insertion (done during the layout phase).

- **set_clock_uncertainty** command lets the user define the clock skew information. Basically this is used to add a certain amount of margin to the clock, both for setup and hold times. During the pre-layout phase one can add more margin as compared to the post-layout phase.

- set_false_path is used to instruct ICC to ignore a particular path for timing or optimization. Identification of false paths in a design is critical. Failure to do so compels DC to optimize all paths in order to reduce total negative slack. Consequently, the critical timing paths may be adversely affected due to optimization of all the paths, which also includes the false paths. The valid start point and endpoint to be used for this command are the input ports or the clock pins of the sequential elements, and the output ports or the data pins of the sequential cells.

- **set_max_delay** defines the maximum delay required in terms of time units for a particular path. In general, it is used for the blocks that contain combinational logic only. However, it may also be used to constrain a block that is driven by multiple clocks, each with a different frequency.

- **set_min_delay** is the opposite of the set_max_delay command, and is used to define the minimum delay required in terms of time units for a particular path.

CHAPTER 4: Design for Testability: Boundary Scan and Logic Scan Insertion

4.1 What is Design-for-Test?

Testability is a design attribute that measures how easy it is to create a program to comprehensively test a manufactured design's quality. Traditionally, design and test processes were kept separate, with test considered only at the end of the design cycle. But in contemporary design flows, test merges with design much earlier in the process, creating what is called a design-for-test (DFT) process flow. Testable circuitry is both controllable and observable. In a testable design; setting specific values on the primary inputs results in values on the primary outputs which indicate whether or not the internal circuitry works properly. [12]

4.2 Understanding Boundary Scan

Boundary scan, sometimes referred to as JTAG (for Joint Test Action Group, the committee that formulated IEEE standard 1149.1 describing boundary scan), is a DFT technique that facilitates the testing of printed circuit board interconnect circuitry and, to a limited extent, the chips on those boards. Boundary scan test structures greatly improve board-level testing, thus shortening the manufacturing test and diagnostics processes.

4.2.1 Boundary Scan Overview

When used on a board, boundary scan stitches the input and output ports of the chips together into a long scan path. Data shifts along the scan path, starting at the edge-connector input TDI (test data in) and ending at the edge-connector output TDO (test data out). In between, the scan path connects all the devices on the board that contain boundary scan circuitry. The TDO of one chip feeds the TDI of the next, all the way around the board. The inputs TCK (test clock) and TMS (test mode select) connect, in parallel, to each boundary scan device in the scan path. With this configuration one can test board interconnections, perform a snapshot of normal system data, or test individual chips. The TAP (test access port) controller is a state machine that controls the operation of the boundary scan circuitry. Boundary scan circuitry's primary use is in board-level

testing, but it can also control circuit-level test structures, such as BIST or internal scan. By adding boundary scan circuitry to your design, you create a standard interface for accessing and testing chips at the board level. [12]

Figure 4.1 shows the general configuration of a chip after the addition of boundary scan logic.

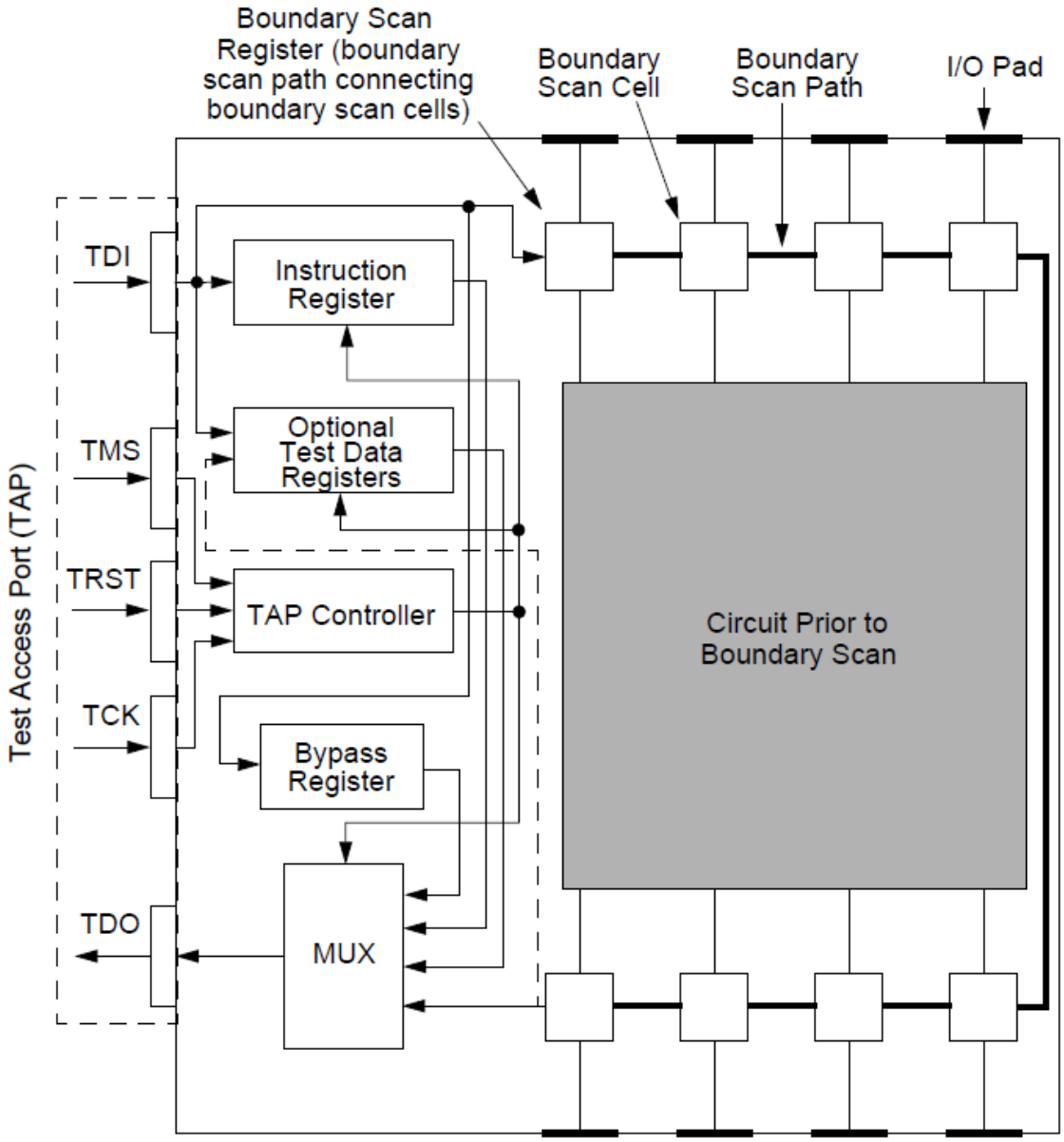

Figure 4.1 - Boundary Scan Architecture [12]

A simple boundary scan architecture consists of the following:

- Circuit Prior to Boundary Scan
- Boundary Scan Cells
- Test Access Port (TAP)
- TAP Controller
- Boundary Scan Register
- Bypass register
- Optional Test Data Registers
    - Device identification (optional)
    - Data-specific (optional)

These registers allow access to the chip's test support features, such as BIST and internal scan paths.

- Instruction Register

  The instruction register controls the boundary scan circuitry by connecting a specific test data register between the TDI and TDO pins and controlling the operation affecting the data in that register, using a predefined set of instructions. Three instructions are mandatory, and several others are optional.

The mandatory instructions include:

- EXTEST
- SAMPLE/PRELOAD
- BYPASS

The optional instructions include:

- INTEST
- IDCODE
- USERCODE
- CLAMP
- HIGHZ
- RUNBIST

This instruction executes the circuit's internal BIST procedure. [12]

4.3 Understanding Scan Design

This section gives an overview of scan design and how it works.

4.3.1 Scan Design Overview

The goal of scan design is to make a difficult-to-test sequential circuit behave (during the testing process) like an easier-to-test combinational circuit. Achieving this goal involves replacing sequential elements with scannable sequential elements (scan cells) and then stitching the scan cells together into scan registers, or scan chains. You can then use these serially-connected scan cells to shift data in and out when the design is in scan mode.

The design shown in Figure 4.2 contains both combinational and sequential portions. Before adding scan, the design had three inputs, A, B, and C, and two outputs, OUT1 and OUT2. This “Before Scan” version is difficult to initialize to a known state, making it difficult to both control the internal circuitry and observe its behavior using the primary inputs and outputs of the design.

After adding scan circuitry, the design has two additional inputs, sc_in and sc_en, and one additional output, sc_out. Scan memory elements replace the original memory elements so that when shifting is enabled (the sc_en line is active), scan data is read in from the sc_in line.

The operating procedure of the scan circuitry is as follows:

1. Enable the scan operation to allow shifting (to initialize scan cells).
2. After loading the scan cells, hold the scan clocks off and then apply stimulus to the primary inputs.
3. Measure the outputs.
4. Pulse the clock to capture new values into scan cells.
5. Enable the scan operation to unload and measure the captured values while simultaneously loading in new values via the shifting procedure. [12]

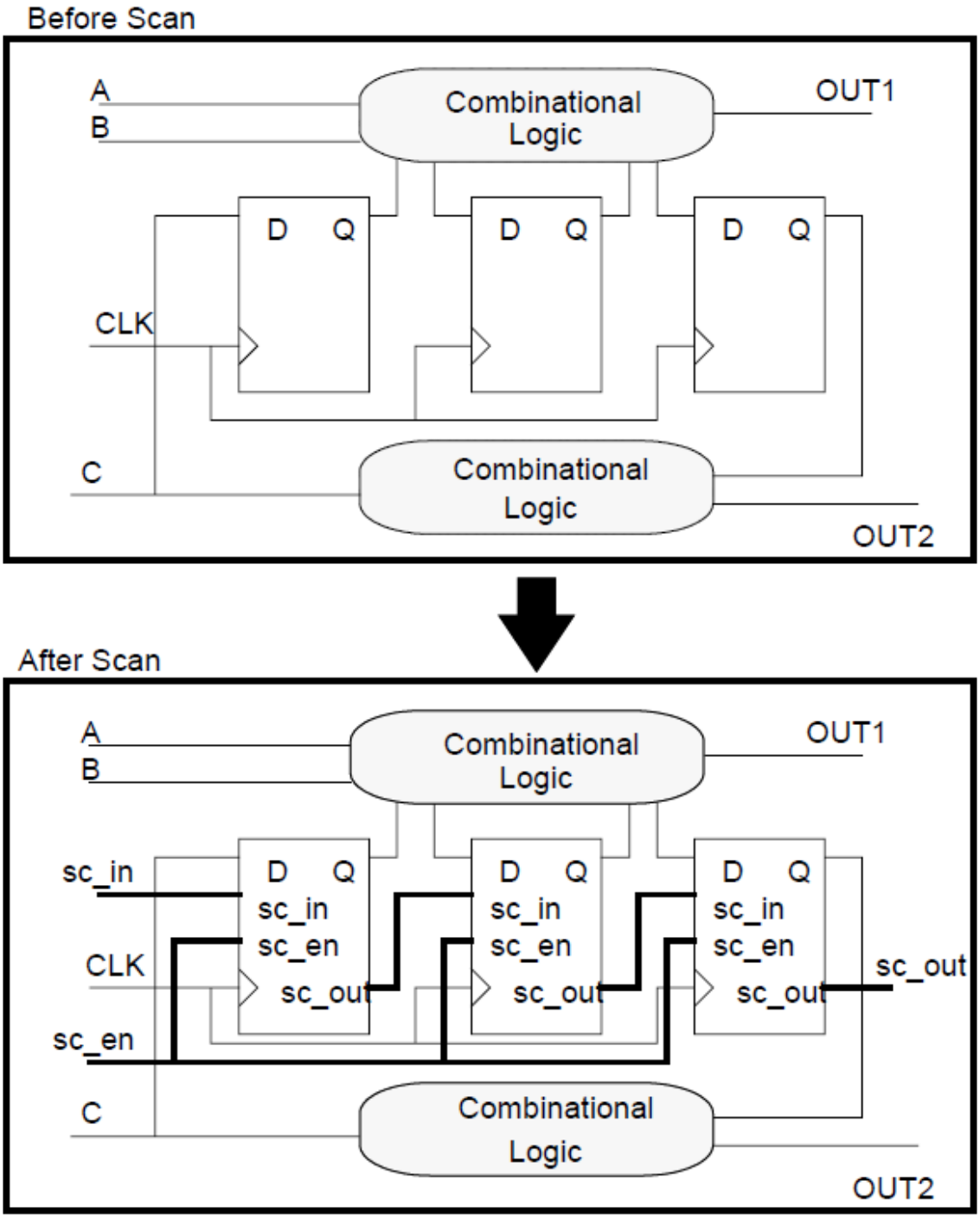

Figure 4.2 - Design Before and After Adding Scan [12]

4.3.2 Full Scan

Full scan is a scan design methodology that replaces all memory elements in the design with their scannable equivalents and then stitches (connects) them into scan chains. The idea is to control and observe the values in all the design's storage elements so you can make the sequential circuit's test generation and fault simulation tasks as simple as those of a combinational circuit. [12]

4.3.3 Full Scan Benefits

The following are benefits of employing a full scan strategy:

- Highly automated process
- Highly-effective, predictable method
- Ease of use
- Assured quality

4.3.4 Partial Scan

Because full scan design makes all storage elements scannable, it may not be acceptable for all designs because of area and timing constraints. Partial scan is a scan design methodology where only a percentage of the storage elements in the design are replaced by their scannable equivalents and stitched into scan chains. Using the partial scan method, one can increase the testability of your design with minimal impact on the design's area or timing. In general, the amount of scan required to get an acceptable fault coverage varies from design to design. [12]

4.3.5.Partial Scan Benefits

The following are benefits of employing a partial scan strategy:

- Reduced impact on area
- Reduced impact on timing
- More flexibility between overhead and fault coverage
- Re-use of non-scan macros

CHAPTER 5: Power Optimization: Clock Gating

5.1 Introduction to Power Compiler

This chapter describes the Power Compiler methodology and describes power library models and power analysis technology. Power Compiler is part of Synopsys's Design Compiler synthesis family. It performs both RTL and gate-level power optimization and gate-level power analysis. By applying Power Compiler's various power reduction techniques, including clock-gating, operand isolation, multivoltage leakage power optimization, and gate-level power optimization, you can achieve power savings, and area and timing optimization in the front-end synthesis domain.

In earlier generations of IC design technologies, the main parameters of concern were timing and area. EDA tools were designed to maximize the speed while minimizing area. Power consumption was a lesser concern. CMOS was considered a low-power technology, with fairly low power consumption at the relatively low clock frequencies used at the time, and with negligible leakage current.

In recent years, however, device densities and clock frequencies have increased dramatically in CMOS devices, thereby increasing the power consumption dramatically. At the same time, supply voltages and transistor threshold voltages have been lowered, causing leakage current to become a significant problem. As a result, power consumption levels have reached their acceptable limits, and power has become as important as timing or area. High power consumption can result in excessively high temperatures during operation. [13]

5.2 Clock Gating

Power optimization at high levels of abstraction has a significant impact on reduction of power in the final gate-level design. Clock gating is an important high-level technique for reducing the power consumption of a design.

5.2.1 Introduction to Clock Gating

Clock gating applies to synchronous load-enable registers, which are groups of flip-flops that share the same clock and synchronous control signals and that are inferred from the same HDL variable. Synchronous control signals include synchronous load-enable, synchronous set, synchronous reset, and synchronous toggle. The registers are implemented by Design Compiler by use of feedback loops. However, these registers maintain the same logic value through multiple cycles and unnecessarily use power. Clock gating saves power by eliminating the unnecessary activity associated with reloading register banks. Designs that benefit most from clock gating are those with low-throughput datapaths. Designs that benefit less from RTL clock gating include designs with finite state machines or designs with throughput-of-one datapaths.

Power Compiler allows you to perform clock gating with the following techniques:

- RTL-based clock gate insertion on unmapped registers. Clock gating occurs when the register bank size meets certain minimum width constraints.
- Gate-level clock gate insertion on both unmapped and previously mapped registers. In this case, clock gating is also applied to objects such as IP cores that are already mapped.
- Power-driven gate-level clock gate insertion, which allows for further power optimizations because all aspects of power savings, such as switching activity and the flip-flop types to which the registers are mapped, are considered.

We can choose the type of clock-gating circuit inserted. Following are some of the choices:

1) Choose an integrated or nonintegrated cell with latch-based clock gating
2) Choose an integrated or nonintegrated cell with latch-free clock gating
3) Insert logic to increase testability
4) Specify a minimum number of bits below which clock gating is not inserted
5) Explicitly include signals in clock gating

6) Explicitly exclude signals from clock gating
7) Specify a maximum number for the fanouts of each clock-gating element
8) Move a clock-gated register to another clock-gating cell
9) Resize the clock-gating element

Figure 5.1 shows a latch-based clock-gating style using a 2-input AND gate; however, depending on the type of register and the gating style, gating can use NAND, OR, and NOR gates instead.

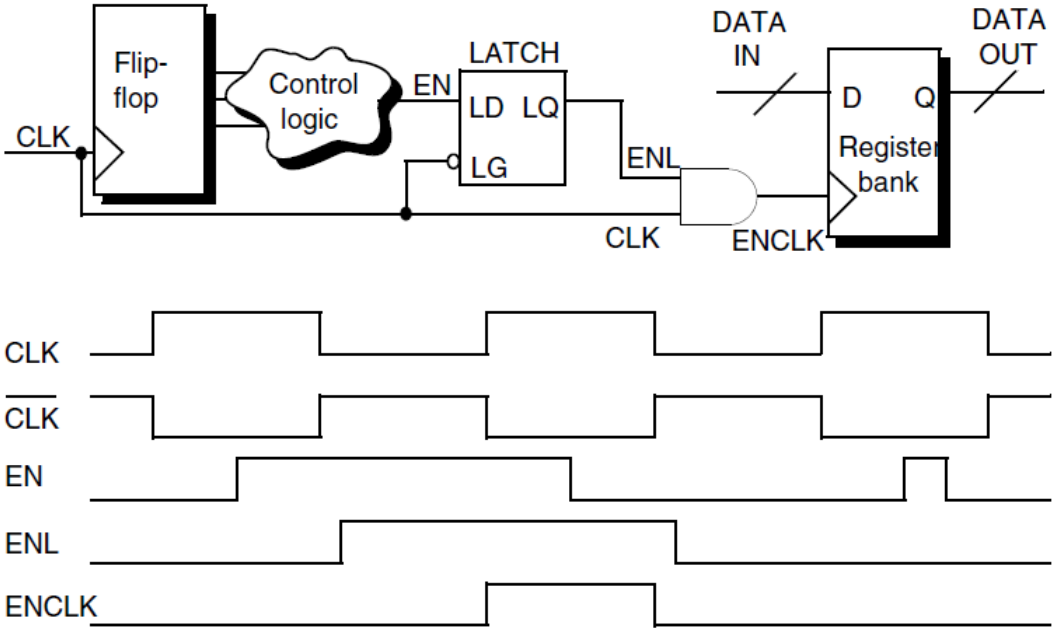

Figure 5.1 - Latch Based Clock Gating [14]

At the bottom of Figure 5.1, waveforms of the signals are shown with respect to the clock signal, CLK. The clock input to the register bank, ENCLK, is gated on or off by the AND gate. ENL is the enabling signal that controls the gating. The register bank is triggered by the rising edge of the ENCLK signal.

The latch prevents glitches on the EN signal from propagating to the register's clock pin. When the CLK input of the 2-input AND gate is at logic state 1, any glitching of the EN signal could, without the latch, propagate and corrupt the register clock signal. The latch

eliminates this possibility because it blocks signal changes when the clock is at logic state 1.

In latch-based clock gating, the AND gate blocks unnecessary clock pulses by maintaining the clock signal's value after the trailing edge. For example, for flip-flops inferred by HDL constructs of rising-edge clocks, the clock gate forces the gated clock to 0 after the falling edge of the clock.

By controlling the clock signal for the register bank, you can eliminate the need for reloading the same value in the register through multiple clock cycles. Clock gating inserts clock-gating circuitry into the register bank's clock network, creating the control to eliminate unnecessary register activity.

5.2.2 Inserting Clock Gates in RTL Design

Power Compiler inserts clock-gating cells to your design if we compile our design using the *-gate_clock* option of the compile or *compile_ultra* command. We can also insert clock gates to your design using the *insert_clock_gating* command.

To insert clock gating logic in your RTL design and to synthesize the design with the clock-gating logic, follow these steps:

1. Read the RTL design.
2. Use the *compile_ultra -gate_clock* command to compile your design.

During the compilation process clock gate is inserted on the registers qualified for clock-gating. By default, during the clock-gate insertion the *compile_ultra* command uses the default settings of the *set_clock_gating_style* command, and also honors the setup, hold, and other constraints specified in the technology libraries. To override the setup and hold values specified in the technology library, use the *set_clock_gating_style* command before compiling your design. You can also use the *insert_clock_gating* command to insert the clock-gating cells. Both, *compile_ultra* and *insert_clock_gating* commands use the default settings of the *set_clock_gating_style* command, during the clock-gate

insertion. The default of the *set_clock_gating_style* command is suitable for most designs.

If you are using testability in your design, use the *insert_dft* command to connect the *scan_enable* and the *test_mode* ports or pins of the integrated clock-gating cells.

Use the *report_clock_gating* command to report the registers and the clock gating cells in the design. Use the *report_power* command to get details of the dynamic power utilized by your design after the clock-gate insertion.

dc_shell> read_verilog design.v

dc_shell> create_clock -period 10 -name CLK

dc_shell> compile_ultra -gate_clock -scan

dc_shell> insert_dft

dc_shell> report_clock_gating

dc_shell> report_power

CHAPTER 6: Introduction to IC Complier

6.1 Introduction

IC Compiler is a single, convergent netlist-to-GDSII or netlist-to-clock-tree-synthesis design tool for chip designers developing very deep submicron designs. It takes as input a gate-level netlist, a detailed floorplan, timing constraints, physical and timing libraries, and foundry-process data, and it generates as output either a GDSII-format file of the layout or a Design Exchange Format (DEF) file of placed netlist data ready for a third-party router. IC Compiler can also output the design at any time as a binary Synopsys Milkyway database for use with other Synopsys tools based on Milkyway or as ASCII files (Verilog, DEF, and timing constraints) for use with tools not from Synopsys. [15]

6.2 User Interfaces

IC Compiler uses the tool command language (Tcl), which is used in many applications in the EDA industry. Using Tcl, you can extend the IC Compiler command language by writing reusable procedures and scripts.

IC Compiler provides two user interfaces:

- Shell interface (icc_shell) – The IC Compiler command-line interface is used for scripts, batch mode, and push-button operations.
- Graphical user interface (GUI) – The IC Compiler graphical user interface is an advanced analysis and physical editing tool. IC Compiler can perform certain tasks, such as very accurately displaying the design and providing visual analysis tools, only from the GUI.

The IC Compiler design flow is an easy-to-use, single-pass flow that provides convergent timing closure. Figure 6.1 shows the basic IC Compiler design flow, which is centered around three core commands that perform placement and optimization (place_opt), clock tree synthesis and optimization (clock_opt), and routing and postroute optimization (route_opt). [15]

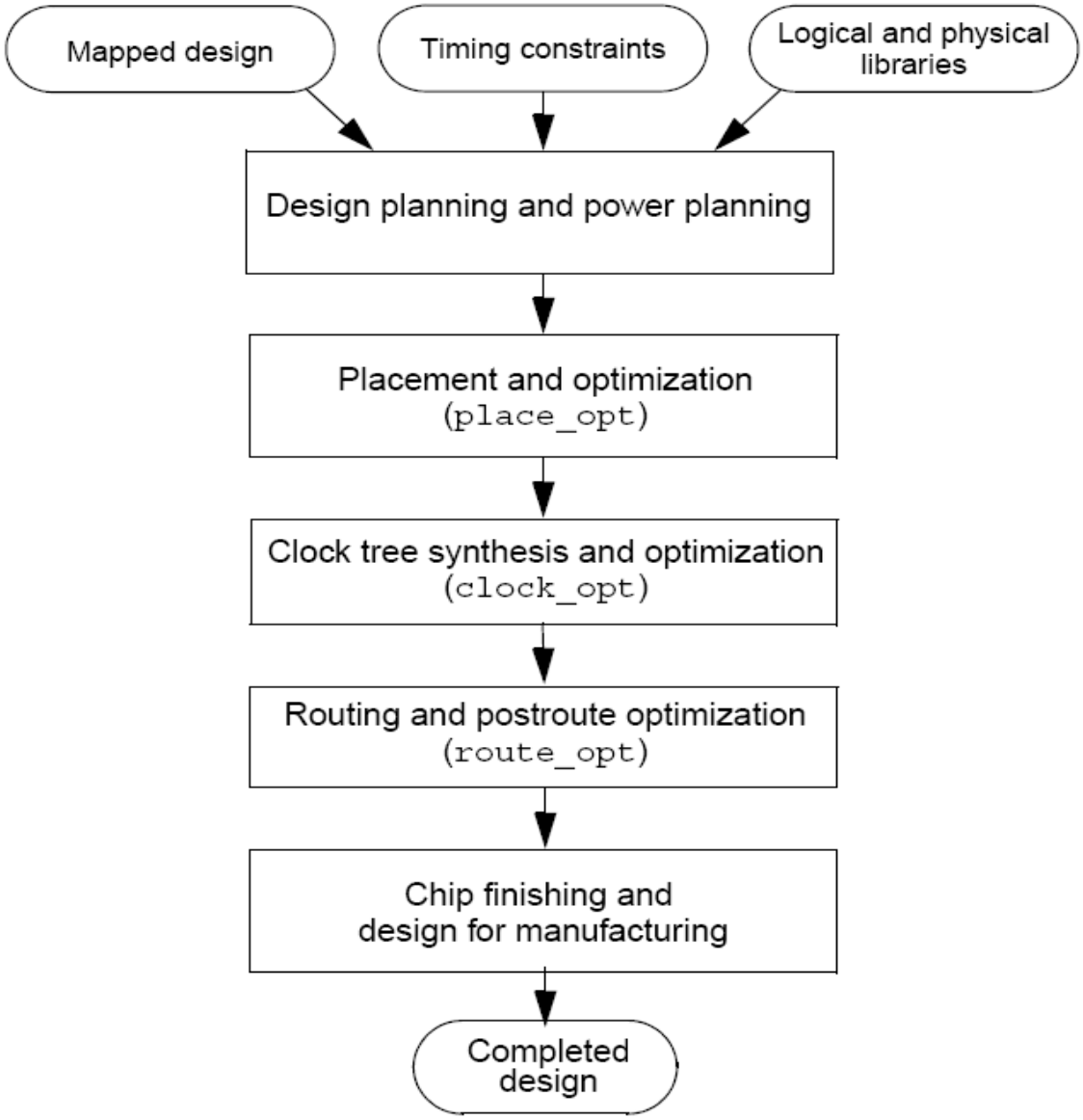

Figure 6.1 - IC Compiler Design Flow [15]

For most designs, the *place_opt*, *clock_opt*, and *route_opt* steps are preset for optimal results. IC Compiler also provides additional placement, clock tree synthesis, and routing technologies, as well as extended physical synthesis tools, that you can use to further improve the quality of results for your design.

To run the IC Compiler design flow,

1. Set up the libraries and prepare the design data.
2. Perform design planning and power planning.

   When you perform design planning and power planning, you create a floorplan to determine the size of the design, create the boundary and core area, create site rows for the placement of standard cells, set up the I/O pads, and create a power plan.
3. Perform placement and optimization.

   To perform placement and optimization, use the *place_opt* core command (or choose Placement > Core Placement and Optimization in the GUI).

   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.
4. Perform clock tree synthesis and optimization.

   To perform clock tree synthesis and optimization, use the *clock_opt* core command (or choose Clock > Core Clock Tree Synthesis and Optimization in the GUI).

   IC Compiler clock tree synthesis and embedded optimization solve complicated clock tree synthesis problems, such as blockage avoidance and the correlation between preroute and postroute data. Clock tree optimization improves both clock skew and clock insertion delay by performing buffer sizing, buffer relocation, gate sizing, gate relocation, level adjustment, reconfiguration, delay insertion, dummy load insertion, and balancing of interclock delays.
5. Perform routing and postroute optimization.

   To perform routing and postroute optimization, use the *route_opt* core command (or choose Route > Core Routing and Optimization in the GUI).

   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.

6. Perform chip finishing and design for manufacturing tasks.

   IC Compiler provides chip finishing and design for manufacturing and yield capabilities that you can apply throughout the various stages of the design flow to address process design issues encountered during chip manufacturing.

7. Save the design.

   Save your design in the Milkyway format. This format is the internal database format used by IC Compiler to store all the logical and physical information about a design. [15]

6.3 How to Invoke the IC Compiler

1. Log in to the UNIX environment with the user id and password .
2. Start IC Compiler from the UNIX promt:

   UNIX$ icc_shell

   The xterm unix prompt turns into the IC Compiler shell command prompt.

3. Start the GUI.

   icc_shell> start_gui

   This window can display schematics and logical browsers, among other things, once a design is loaded.

6.4 Preparing the Design

IC Compiler uses a Milkyway design library to store design and its associated library information. This section describes how to set up the libraries, create a Milkyway design library, read your design, and save the design in Milkyway format.

These steps are explained in the following sections:

• Setting up the Libraries

• Setting up the Power and Ground Nets
• Reading the Design
• Annotating the Physical Data
• Preparing for Timing Analysis and RC Calculation
• Saving the Design

6.4.1 Setting Up the Libraries

IC Compiler requires both logic libraries and physical libraries. The following sections describe how to set up and validate these libraries.

• Setting Up the Logic Libraries:

IC Compiler uses logic libraries to provide timing and functionality information for all standard cells. In addition, logic libraries can provide timing information for hard macros, such as RAMs.

IC Compiler uses variables to define the logic library settings. In each session, you must define the values for the following variables (either interactively, in the .synopsys_dc.setup file, or by restoring the values saved in the Milkyway design library) so that IC Compiler can access the libraries:

• *search_path*

Lists the paths where IC Compiler can locate the logic libraries.

• *target_library*

Lists the logic libraries that IC Compiler can use to perform physical optimization.

• *link_library*

Lists the logic libraries that IC Compiler can search to resolve references.

• Setting Up the Physical Libraries:

IC Compiler uses Milkyway reference libraries and technology (.tf) files to provide physical library information. The Milkyway reference libraries contain physical information about the standard cells and macro cells in your technology library. In addition, these reference libraries define the placement unit tile. The technology files

provide technology-specific information such as the names and characteristics (physical and electrical) for each metal layer.

The physical library information is stored in the Milkyway design library. For each cell, the Milkyway design library contains several views of the cell, which are used for different physical design tasks.

If you have not already created a Milkyway library for your design (by using another tool that uses Milkyway), you need to create one by using the IC Compiler tool. If you already have a Milkyway design library, you must open it before working on your design.

This section describes how to perform the following tasks:

• Create a Milkyway design library

To create a Milkyway design library, use the create_mw_lib command (or choose File > Create Library in the GUI).

• Open a Milkyway design library

To open an existing Milkyway design library, use the open_mw_lib command (or choose File > Open Library in the GUI).

• Report on a Milkyway design library

To report on the reference libraries attached to the design library, use the *-mw_reference_library* option.

icc_shell>report_mw_lib-mw_reference_library\ design_library_name

To report on the units used in the design library, use the report_units command.

icc_shell> report_units

• Change the physical library information

To change the technology file, use the *set_mw_technology_file* command (or choose File > Set Technology File in the GUI) to specify the new technology file name and the name of the design library.

• Save the physical library information

To save the technology or reference control information in a file for later use, use the *write_mw_lib_files* command (or choose File > Export > Write Library File in the GUI). In a single invocation of the command, you can output only one type of file. To

output both a technology file and a reference control file, you must run the command twice.

• Verifying Library Consistency:

Consistency between the logic library and the physical library is critical to achieving good results. Before you process your design, ensure that your libraries are consistent by running the *check_library* command. [15]

icc_shell> check_library

6.4.2 Setting Up the Power and Ground Nets

IC Compiler uses variables to define names for the power and ground nets. In each session, you must define the values for the following variables (either interactively or in the .synopsys_dc.setup file) so that IC Compiler can identify the power and ground nets:

• *mw_logic0_net*

By default, IC Compiler VSS as the ground net name. If you are using a different name, you must specify the name by setting the *mw_logic0_net* variable.

• *mw_logic1_net*

By default, IC Compiler uses VDD as the power net name. If you are using a different name, you must specify the name by setting the *mw_logic1_net* variable.

6.4.3 Reading the Design

IC Compiler can read designs in either Milkyway or ASCII (Verilog, DEF, and SDC files) format.

• Reading a Design in Milkyway Format

• Reading a Design in ASCII Format

6.4.4 Annotating the Physical Data

IC Compiler provides several methods of annotating physical data on the design:

• Reading the physical data from a DEF file

To read a DEF file, use the *read_def* command (or choose File > Import > Read DEF in

the GUI).

icc_shell> read_def -allow_physical design_name.def

• Reading the physical data from a floorplan file

A floorplan file is a file that you previously created by using the *write_floorplan* command (or by choosing Floorplan > Write Floorplan in the GUI).

icc_shell> read_floorplan floorplan_file_name

• Copying the physical data from another design

To copy physical data from the layout (CEL) view of one design in the current Milkyway design library to another, use the *copy_floorplan* command (or choose Floorplan > Copy Floorplan in the GUI). [15]

icc_shell> copy_floorplan -from design1

6.4.5 Preparing for Timing Analysis and RC Calculation

IC Compiler provides RC calculation technology and timing analysis capabilities for both preroute and postroute data. Before you perform RC calculation and timing analysis, you must complete the following tasks:

• Set up the TLUPlus files

You specify these files by using the *set_tlu_plus_files* command (or by choosing File > Set TLU+ in the GUI).

*icc_shell> set_tlu_plus_files *

*-tech2itf_map ./path/map_file_name.map *

*-max_tluplus ./path/worst_settings.tlup *

-min_tluplus ./path/best_settings.tlup

• (Optional) Back-annotate delay or parasitic data

To back-annotate the design with delay information provided in a Standard Delay Format (SDF) file, use the *read_sdf* command (or choose File > Import > Read SDF in the GUI).

To remove annotated data from design, use the *remove_annotations* command.

• Set the timing constraints

At a minimum, the timing constraints must contain a clock definition for each clock signal, as well as input and output arrival times for each I/O port. This requirement ensures that all signal paths are constrained for timing.

To read a timing constraints file, use the *read_sdc* command (or choose File > Import > Read SDC in the GUI).

icc_shell> read_sdc -version 1.7 design_name.sdc

• Specify the analysis mode

Semiconductor device parameters can vary with conditions such as fabrication process, operating temperature, and power supply voltage. The *set_operating_conditions* command specifies the operating conditions for analysis.

• (Optional) Set the derating factors

If your timing library does not include minimum and maximum timing data, you can perform simultaneous minimum and maximum timing analysis by specifying derating factors for your timing library. Use the *set_timing_derate* command to specify the derating factors.

• Select the delay calculation algorithm

By default, IC Compiler uses Elmore delay calculation for both preroute and postroute delay calculations. For postroute delay calculations, you can choose to use Arnoldi delay calculation either for clock nets only or for all nets. Elmore delay calculation is faster, but its results do not always correlate with the PrimeTime and PrimeTime SI results. The Arnoldi calculation is best used for designs with smaller geometries and high resistive nets, but it requires more runtime and memory. [15]

6.4.6 Saving the Design

To save the design in Milkyway format, use the *save_mw_cel* command (or choose File > Save Design in the GUI). [15]

CHAPTER 7: Design Planning

7.1 Introduction

Design planning in IC Compiler provides basic floorplanning and prototyping capabilities such as dirty-netlist handling, automatic die size exploration, performing various operations with black box modules and cells, fast placement of macros and standard cells, packing macros into arrays, creating and shaping plan groups, in-place optimization, prototype global routing analysis, hierarchical clock planning, performing pin assignment on soft macros and plan groups, performing timing budgeting, converting the hierarchy, and refining the pin assignment.

Power network synthesis and power network analysis functions, applied during the feasibility phase of design planning, provide automatic synthesis of local power structures within voltage areas. Power network analysis validates the power synthesis results by performing voltage-drop and electromigration analysis. [15]

7.2 Tasks to be performed during Design Planning

- Initializing the Floorplan
- Automating Die Size Exploration
- Handling Black Boxes
- Performing an Initial Virtual Flat Placement
- Creating and Shaping Plan Groups
- Performing Power Planning
- Performing Prototype Global Routing
- Performing Hierarchical Clock Planning
- Performing In-Place Optimization
- Performing Routing-Based Pin Assignment
- Performing RC Extraction

• Performing Timing Analysis
• Performing Timing Budgeting
• Committing the Physical Hierarchy
• Refining the Pin Assignment

7.3 Initializing the Floorplan

The steps in initializing the floorplan are described below.

• Reading the I/O Constraints:

To load the top-level I/O pad and pin constraints, use the *read_io_constraints* command.

• Defining the Core and Placing the I/O Pads:

To define the core and place the I/O pads and pins, use the *initialize_floorplan* command.

• Creating Rectilinear-Shaped Blocks:

Use the *initialize_rectilinear_block* command to create a floorplan for rectilinear blocks from a fixed set of L, T, U, or cross-shaped templates. These templates are used to determine the cell boundary and shape of the core. To do this, use *initialize_rectilinear_block -shape L|T|U|X*.

• Writing I/O Constraint Information:

To write top-level I/O pad or pin constraints, use the *write_io_constraints* command.

Read the Synopsys Design Constraints (SDC) file (*read_sdc* command) to ensure that all signal paths are constrained for timing.

• Adding Cell Rows:

To add cell rows, use the *add_row* command.

• Removing Cell Rows:

To remove cell rows, use the *cut_row* command.

• Saving the Floorplan Information:

To save the floorplan information, use the *write_floorplan* command.

•Writing Floorplan Physical Constraints for Design Compiler Topographical Technology:

IC Compiler can now write out the floorplan physical constraints for Design Compiler

Topographical Technology (DC-T) in Tcl format. The reason for using floorplan physical constraints in the Design Compiler topographical technology mode is to accurately represent the placement area and to improve timing correlation with the post-place-and-route design. The command syntax is:

write_physical_constraints -output output_file_name -port_side [15]

7.4 Automating Die Size Exploration

This section describes how to use MinChip technology in IC Compiler to automate the processes exploring and identifying the valid die areas to determine smallest routable, die

size for your design while maintaining the relative placement of hard macros, I/O cells, and a power structure that meets voltage drop requirements. The technology is integrated into the Design Planning tool through the *estimate_fp_area* command. The input is a physically flat Milkyway CEL view.

7.5 Handling Black Boxes

Black boxes can be represented in the physical design as either soft or hard macros. A black box macro has a fixed height and width. A black box soft macro sized by area and utilization can be shaped to best fit the floorplan.

To handle the black boxes run the following set of commands.

set_fp_base_gate
estimate_fp_black_boxes
flatten_fp_black_boxes
create_fp_placement
place_fp_pins
create_qtm_model qtm_bb
set_qtm_technology -lib library_name
create_qtm_port -type clock $port
report_qtm_model
write_qtm_model -format qtm_bb
report_timing qtm_bb

7.6 Performing an Initial Virtual Flat Placement

The initial virtual flat placement is very fast and is optimized for wire length, congestion, and timing.

The way to perform an initial virtual flat placement is described below.

• Evaluating Initial Hard Macro Placement:

No straightforward criteria exist for evaluating the initial hard macro placement. Measuring the quality of results (QoR) of the hard macro placement can be very subjective and often depends on practical design experience.

• Specifying Hard Macro Placement Constraints:

Different methods can be use to control the preplacement of hard macros and improve the QoR of the hard macro placement.

1. Creating a User-Defined Array of Hard Macros
2. Setting Floorplan Placement Constraints On Macro Cells
3. Placing a Macro Cell Relative to an Anchor Object
4. Using a Virtual Flat Placement Strategy
5. Enhancing the Behavior of Virtual Flat Placement With the macros_on_edge Switch
6. Creating Macro Blockages for Hard Macros
7. Padding the Hard Macros

• Padding the Hard Macros:

To avoid placing standard cells too close to macros, which can cause congestion or DRC violations, one can set a user-defined padding distance or keepout margin around the macros. One can set this padding distance on a selected macro's cell instance master.During virtual flat placement no other cells will be placed within the specified distance from the macro's edges. [15]

To set a padding distance (keepout margin) on a selected macro's cell instance master, use the *set_keepout_margin* command.

• Placing Hard Macros and Standard Cells:

To place the hard macros and standard cells simultaneously, use the *create_fp_placement* command.

• Performing Floorplan Editing:

IC Compiler performs the following floorplan editing operations.

1. Creating objects
2. Deleting objects
3. Undoing and redoing edit changes
4. Moving objects
5. Changing the way objects snap to a grid
6. Aligning movable objects

7.7 Creating and Shaping Plan Groups

This section describes how to create plan groups for logic modules that need to be physically implemented. Plan groups restrict the placement of cells to a specific region of the core area. This section also describes how to automatically place and shape objects in a design core, add padding around plan group boundaries, and prevent signal leakage and maintain signal integrity by adding modular block shielding to plan groups and soft macros.

The following steps are covered for Creating and Shaping Plan Groups.

• Creating Plan Groups:

To create a plan group, *create_plan_groups* command.

To remove (delete) plan groups from the current design, use the *remove_plan_groups* command.

• Automatically Placing and Shaping Objects In a Design Core:

Plan groups are automatically shaped, sized, and placed inside the core area based on the distribution of cells resulting from the initial virtual flat placement. Blocks (plan groups, voltage areas, and soft macros) marked fix remain fixed; the other blocks, whether or not they are inside the core, are subject to being moved or reshaped.

To automatically place and shape objects in the design core, *shape_fp_blocks* command.

• Adding Padding to Plan Groups:

To prevent congestion or DRC violations, one can add padding around plan group boundaries. Plan group padding sets placement blockages on the internal and external edges of the plan group boundary. Internal padding is equivalent to boundary spacing in the core area. External padding is equivalent to macro padding.

To add padding to plan groups, *create_fp_plan_group_padding* command.

To remove both external and internal padding for the plan groups, use the *remove_fp_plan_group_padding* command.

• Adding Block Shielding to Plan Groups or Soft Macros:

When two signals are routed parallel to each other, signal leakage can occur between the signals, leading to an unreliable design. One can protect signal integrity by adding modular block shielding to plan groups and soft macros. The shielding consists of metal rectangles that are created around the outside of the soft macro boundary in the top level of the design, and around the inside boundary of the soft macro.

To add block shielding for plan groups or soft macros, use the *create_fp_block_shielding* command.

To remove the signal shielding created by modular block shielding, use the *remove_fp_block_shielding* command. [15]

7.8 Performing Power Planning

After completed the design planning process and have a complete floorplan, one can perform power planning, as explained below.

• Creating Logical Power and Ground Connections:

To define power and ground connections, use the *connect_pg_nets* command.

• Adding Power and Ground Rings:

After doing floorplanning, one need to add power and ground rings.

To add power and ground rings, use the *create_rectangular_rings* command.

• Adding Power and Ground Straps:

To add power and ground straps, use the *create_power_straps* command.

• Prerouting Standard Cells:

To preroute standard cells, use the *preroute_standard_cells* command.

• Performing Low-Power Planning for Multithreshold-CMOS Designs:

One can perform floorplanning for low-power designs by employing power gating. Power gating has the potential to reduce overall power consumption substantially because it reduces leakage power as well as switching power.

• Performing Power Network Synthesis:

As the design process moves toward creating 65-nm transistors, issues related to power and signal integrity, such as power grid generation, voltage (IR) drop, and electromigration, have become more significant and complex. In addition, this complex technology lengthens the turnaround time needed to identify and fix power and signal integrity problems.

By performing power network synthesis one can preview an early power plan that reduces the chances of encountering electromigration and voltage drop problems later in the detailed power routing.

To perform the PNS, one can run the set of following commands. [15]

synthesize_fp_rail
set_fp_rail_constraints
set_fp_rail_constraints -set_ring
set_fp_block_ring_constraints
set_fp_power_pad_constraints
set_fp_rail_region_constraints
set_fp_rail_voltage_area_constraints
set_fp_rail_strategy

• Committing the Power Plan:

Once the IR drop map meets the IR drop constraints, one can run the *commit_fp_rail* command to transform the IR drop map into a power plan.

• Handling TLUPlus Models in Power Network Synthesis:

Power network synthesis supports TLUPlus models.

set_fp_rail_strategy -use_tluplus true

• Checking Power Network Synthesis Integrity:

Initially, when power network synthesis first proposes a power mesh structure, it assumes that the power pins of the mesh are connected to the hard macros and standard cells in the design. It then displays a voltage drop map that one can view to determine if it meets the voltage (IR) drop constraints. After the power mesh is committed, one might discover "problem" areas in design as a result of automatic or manual cell placement. These areas are referred to as chimney areas and pin connect areas.

To Check the PNS Integrity one can run the following set of commands.

set_fp_rail_strategy -pns_commit_check_file
set_fp_rail_strategy -pns_check_chimney_file
set_fp_rail_strategy -pns_check_chimney_file pns_chimney_report
set_fp_rail_strategy -pns_check_hor_chimney_layers
set_fp_rail_strategy -pns_check_chimney_min_dist
set_fp_rail_strategy -pns_check_pad_connection file_name
set_fp_rail_strategy -pns_report_pad_connection_limit
set_fp_rail_strategy -pns_report_min_pin_width
set_fp_rail_strategy -pns_check_hard_macro_connection file_name
set_fp_rail_strategy -pns_check_hard_macro_connection_limit
set_fp_rail_strategy -pns_report_min_pin_width

• Analyzing the Power Network:

One perform power network analysis to predict IR drop at different floorplan stages on both complete and incomplete power nets in the design.

To perform power network analysis, use the *analyze_fp_rail* command.

To add virtual pads, use the *create_fp_virtual_pad* command.

To ignore the hard macro blockages, use the *set_fp_power_plan_constraints* command.

• Viewing the Analysis Results:

When power and rail analysis are complete, one can check for the voltage drop and electromigration violations in the design by using the voltage drop map and the electromigration map. One can save the results of voltage drop and electromigration current density values to the database by saving the CEL view that has just been analyzed.

• Reporting Settings for Power Network Synthesis and Power Network Analysis Strategies:

To get a report of the current values of the strategies used by power network synthesis

and power network analysis by using the *report_fp_rail_strategy* command. [15]

7.9 Performing Prototype Global Routing

One can perform prototype global routing to get an estimate of the routability and congestion of the design. Global routing is done to detect possible congestion “hot spots” that might exist in the floorplan due to the placement of the hard macros or inadequate channel spacing.

To perform global routing, use the *route_fp_proto* command.

7.10 Performing Hierarchical Clock Planning

This section describes how to reduce timing closure iterations by performing hierarchical clock planning on a top-level design during the early stages of the virtual flat flow, after plan groups are created and before the hierarchy is committed. One can perform clock planning on a specified clock net or on all clock nets in the design.

- Setting Clock Planning Options:

  To set clock planning options, use the *set_fp_clock_plan_options* command.
- Performing Clock Planning Operations:

  To perform clock planning operations, use the *compile_fp_clock_plan* command.
- Generating Clock Tree Reports:

  To generate clock tree reports, use the *report_clock_tree* command.
- Using Multivoltage Designs in Clock Planning:

  Clock planning supports multivoltage designs. Designs in multivoltage domains operate at various voltages. Multivoltage domains are connected through level-shifter cells. A level-shifter cell is a special cell that can carry signals across different voltage areas.
- Performing Plan Group-Aware Clock Tree Synthesis in Clock Planning:

  With this feature, clock tree synthesis can generate a clock tree that honors the plan groups while inserting buffers in the tree and prevent new clock buffers from being placed on top of a plan group unless they drive the entire subtree inside that particular

plan group. This results in a minimum of clock feedthroughs, which makes the design easier to manage during partitioning and budgeting. [15]

7.11 Performing In-Place Optimization

In-place optimization is an iterative process that is based on virtual routing. Three types of optimizations are performed: timing improvement, area recovery, and fixing DRC violations. These optimizations preserve the netlist's logical hierarchy as well as the physical locations of the cells.

To perform in-place optimization, use the *optimize_fp_timing* command.

7.12 Performing Routing-Based Pin Assignment

IC Compiler provides two ways to perform pin assignment: on soft macros (traditional pin assignment) or on plan groups (pin cutting flow).
To assign pin constraints, use the *set_fp_pin_constraints* command.
To assign soft macros pins, use the *place_fp_pins* command.
To perform Block Level Pin Assignmentuse, use the *place_fp_pins -block_level* command.
To align soft macro pins, use the *align_fp_pins* command.

To remove soft macro pin overlaps, use the *remove_fp_pin_overlaps* command.

7.13 Performing RC Extraction

Perform postroute RC estimation by using the *extract_rc* command.

7.14 Performing Timing Analysis

Use the *report_timing* command to generate timing reports for the design. Depending on the options selected, one can report valid paths for the entire design or for specific paths. The timing report helps evaluate why some parts of a design might not be optimized.

7.15 Performing Timing Budgeting

During the design planning stage, timing budgeting is an important step in achieving timing closure in a physically hierarchical design. The timing budgeting algorithm determines the corresponding timing boundary constraints for each top-level soft macro or plan group (block) in a design. If the timing boundary constraints for each block are met when they are implemented, the top-level timing constraints are satisfied.

Timing budgeting distributes positive and negative slack between blocks and then generates timing constraints in the Synopsys Design Constraints (SDC) format for block-level implementation.

To generate a pre-budgeting timing analysis report file, use the *check_fp_timing_environment* command.

To run the timing budgeter, use the *allocate_fp_budgets* command.

Immediately after budgeting a design, you can use the *check_fp_budget_result* command to perform post-budget analysis. [15]

7.16 Committing the Physical Hierarchy

This section describes how to commit the physical hierarchy after finalizing the floorplan by converting plan groups to soft macros. Committing the hierarchy creates a new level of physical hierarchy in the virtual flat design by creating CEL views for selected plan groups. After committing the physical hierarchy, you can also "uncommit" the physical hierarchy by converting the soft macros back into plan groups.

In addition, this section also describes how to propagate top-level preroutes into soft macros, recover all pushed-down objects in child cells to the top-level, and uncommit the physical hierarchy by converting soft macros back into plan groups.

To convert plan groups to soft macros, use the *commit_fp_plan_groups* command.

To push down physical objects to the soft macro level, use the *push_down_fp_objects* command.

To push up physical objects to the soft macro level, use the *push_up_fp_objects* command.

To uncommit the physical hierarchy, use the *uncommit_fp_soft_macros* command. [15]

7.17 Refining the Pin Assignment

One can analyze and evaluate the quality of the pin assignment results by checking the placement of soft macros pins in the design and the pin alignment.
To check the placement of soft macro pins, use the *check_fp_pin_assignment* command.
To check the pin alignment, use the *check_fp_pin_alignment* command. [15]

CHAPTER 8: Placement

Placement is an essential step in electronic design automation - the portion of the physical design flow that assigns exact locations for various circuit components within the chip's core area. An inferior placement assignment will not only affect the chip's performance but might also make it nonmanufacturable by producing excessive wirelength, which is beyond available routing resources. Consequently, a placer must perform the assignment while optimizing a number of objectives to ensure that a circuit meets its performance demands. Typical placement objectives include

- Total wirelength
- Timing
- Power
- A secondary objective is placement runtime minimization

8.1 Tasks to be performed during Placement

• Defining Placement Blockages

• Setting Placement Options

• Inserting Port Protection Diodes

• Preparing for High-Fanout Net Synthesis

• Analyzing Placement and Optimization Feasibility

• Performing Clock Tree Synthesis During Placement

• Performing Placement and Optimization

• Using Physical Optimization

• Performing Layer Optimization

• Performing Preroute RC Estimation

• Analyzing Placement

• Refining Placement

8.2 Defining Placement Blockages

Placement blockages are areas that leaf cells must avoid during placement and legalization, including overlapping any part of the placement blockage. Placement blockages can be hard or soft. A hard blockage prevents cells from being put in the blockage area. A soft blockage restricts the coarse placer from putting cells in the blockage area, but optimization and legalization can place cells in a soft blockage area.

To define a hard global keepout margin, specify the width, in microns, of the keepout margin by setting the *physopt_hard_keepout_distance* variable.

To define a soft global keepout margin, specify the width, in microns, of the keepout margin by setting the *placer_soft_keepout_channel_width* variable.

To define a keepout margin with different widths on each side or to define a keepout margin for specific cells, use the *set_keepout_margin* command.

To create placement blockages, use the *create_placement_blockage* command.

To return a collection of placement blockages in the current design that match certain criteria, use the *get_placement_blockages* command.

To remove placement blockages from the design, use the *remove_placement_blockage* command.

To create placement blockages that fit in the thin channels to improve the quality of results at the top level, use the *derive_placement_blockages* command. [15]

8.3 Setting Placement Options

IC Compiler attempts to minimize congestion during placement and optimization. Congestion occurs when the number of wires going through a region exceeds the capacity of that region. This condition is detected by global routing. IC Compiler places and moves cells in such a manner to avoid congestion and to fix congestion problems when they occur.

One can set certain options related to congestion avoidance with the *set_congestion_options* command.

To report and remove congestion option settings, use the *report_congestion_options* and *remove_congestion_options* commands respectively.

To report congestion conditions, including DRC violations and routing density, use the *report_congestion* command.

For placement that is not set up for congestion removal, one can control how densely cells can be packed by using the placer_max_cell_density_threshold variable.

To create a move bound in IC Compiler, use the *create_bounds* command.

To report the move bounds in your design, use the *report_bounds* command.

To return a collection of move bounds in the current design that match certain criteria, use the *get_bounds* command.

To remove move bounds from your design, use the *remove_bounds* command.

To define the intercell spacing rules, using the *set_lib_cell_spacing_label* command.

Define the spacing requirements between the labels by using the *set_spacing_label_rule* command.

To report both the intercell and boundary spacing rules, use the *report_spacing_rules* command with the *-all* option. To report the intercell spacing rules for a specific collection of library cells, use the *report_spacing_rules* command with the *-of_library_cells* option.

To remove all spacing rules, use the *remove_all_spacing_rules* command.

One can specify a list of preferred buffers to fix hold violations during preroute optimization by using the *set_prefer* and *set_fix_hold_options* commands.

To enable multithreading, use the *set_host_options* command with the *–max_cores* option.

The *set_auto_disable_drc_nets* command enables DRC fixing on constant nets.

One can also insert tie cells manually with the connect_tie_cells command.

To preserv dont_touch Nets during optimization, use the *set_dont_touch_network* command. [15]

8.4 Inserting Port Protection Diodes

IC Compiler can automatically insert protection diodes on subdesign ports to prevent antenna violations at the top level.

Use the *insert_port_protection_diodes* command to add diodes to the specified ports to your netlist, one diode per port.

To report the port protection diodes that are inserted in your design, use the *report_port_protection_diodes* command.

8.5 Preparing for High-Fanout Net Synthesis

During placement and optimization, IC Compiler does not buffer clock nets as defined by the *create_clock* command, but it does, by default, buffer other high-fanout nets, such as resets or scan enables, using a built-in high-fanout synthesis engine.
Before running high-fanout net synthesis during the *place_opt* step, define the buffering options by using the *set_ahfs_options* command.

During high-fanout net synthesis, IC Compiler automatically analyzes the buffer trees to determine the fanout thresholds by default, and then it removes and builds buffer trees.To get information about the buffer trees in your design, use the *report_buffer_tree* command.To remove buffer trees from your design, use the *remove_buffer_tree* command.To perform statistical analysis, specify the *-design_statistics* option with the *check_physical_design* command. [15]

8.6 Analyzing Placement and Optimization Feasibility

During placement and optimization, one might need to run the *place_opt* command multiple times to obtain optimal results. To improve runtime and to reduce iterations of placement and optimization during early design stages, one can use the *place_opt_feasibility* command to analyze placement and optimization feasibility.

Running the feasibility flow can minimize the number of iterations, validate timing constraints, and report all paths that are not feasible.

8.7 Performing Clock Tree Synthesis During Placement

If the design contains simple clock tree structures and uses the same design rule constraints for placement and clock tree synthesis, one can simplify the design flow by performing clock tree synthesis and optimization during placement.

To perform clock tree synthesis and optimization during placement, use the *set_place_opt_cts_strategy* command.

8.8 Performing Placement and Optimization

To perform placement and optimization, use the *place_opt* command.

The *place_opt* command performs coarse placement, high-fanout net synthesis, physical optimization, and legalization.During placement and optimization, the *place_opt* command does not touch the clock networks in the design.

For most designs, using the default setting for the *placer_enable_enhanced_router* variable should meet the congestion optimization requirements during placement.

To improve congestion for a complex floorplan or to improve timing for the design, one can use magnet placement to specify fixed objects as magnets and have IC Compiler move their connected standard cells close to them.To perform magnet placement, use the *magnet_placement* command.To return a collection of cells that can be moved with magnet placement, use the *get_magnet_cells* command. [15]

8.9 Using Physical Optimization

One can run incremental placement-based optimization that supports area recovery, design rule fixing, sizing, and route-based optimization by using the *psynopt* command. By default, this command performs timing optimization and design rule fixing, based on the maximum capacitance and maximum transition settings while keeping the clock networks untouched. It can also perform power optimizations. The *psynopt* command continues to optimize until no more optimizations can be performed. When done, it completes the optimizations with a legalized placement of the design. [15]

8.10 Performing Layer Optimization

One can use the *preroute_focal_opt* command to perform preroute optimization to fix high-fanout nets, setup, hold, and logical DRC violations after the *place_opt* or *clock_opt* stage but before the *route_opt* stage. One can also use the command to perform layer optimization to control the tradeoff between buffering and layer assignment by specifying the *-layer_optimization* option. Layer optimization works only in the current scenario.

To change the default settings of layer optimization, use the *set_preroute_focal_opt_strategy* command. [15]

8.11 Performing Preroute RC Estimation

IC Compiler automatically performs preroute RC estimation when running the following commands: *place_opt*, *clock_opt*, *create_placement*, *legalize_placement*, and *psynopt*. In addition, one can explicitly perform preroute RC estimation by running the *extract_rc* command.

To report the delay estimation coefficients, use the *report_delay_estimation_options* command.

To set net-based layer constraints, use the *set_net_routing_layer_constraints* command.

To report and remove routing layer constraints, use the *report_net_routing_layer_constraints* and *remove_net_routing_layer_constraints* commands. [15]

8.12 Analyzing Placement

After placement, one can view and analyze the results. One can report the area utilization with the *report_placement_utilization* command, the design timing with the *report_timing* command, the power consumption characteristics with the *report_power* command, and the QoR with the *create_qor_snapshot* command.

After setting the timing constraints such as clocks, input delays, and output delays, it is a good idea to use the *check_timing* command to check for timing setup problems and timing conditions such as incorrectly specified generated clocks and combinational feedback loops. The command checks the timing attributes of the current design and issues warning messages about any unusual conditions found.

Use the *report_timing* command to report the worst-case timing paths in the design.

To get a detailed report on the delay calculation at a given point along a timing path, use the *report_delay_calculation* command.

The *report_power* command calculates and reports power for a design. The command uses the user-annotated switching activity to calculate the net switching power, cell internal power, and cell leakage power, and it displays the calculated values in a power report. [15]

8.13 Refining Placement

If the design shows large timing or violations after running the *place_opt* command, adjust the *place_opt* options and rerun *place_opt*, as described in "Performing Placement and Optimization".

If the design shows small timing or violations after running *place_opt*, running *psynopt* to fix these violations, as described in "Using Physical Optimization".

If the design has congestion violations after running *place_opt*, rerun *place_opt* with high-effort congestion reduction (-*congestion* option). If the design still has congestion violations, one can refine the placement to fix these violations.

To refine the placement, use the *refine_placement* command. The *refine_placement* command performs incremental placement and legalization.[15]

CHAPTER 9: Clock Tree Synthesis

Introduction

A clock (buffer) tree is built to balance the output loads and minimize the clock skew a delay line can be added to the network to meet the minimum insertion delay (clock balancing), buffers are used to speed up the clock signals.

Effects of CTS:

- Several (Hundreds/Thousands) of clock buffers added to the design
- Placement / Routing congestion may increase
- Non-clock cells may have been moved to less ideal locations
- Timing violations can be introduced

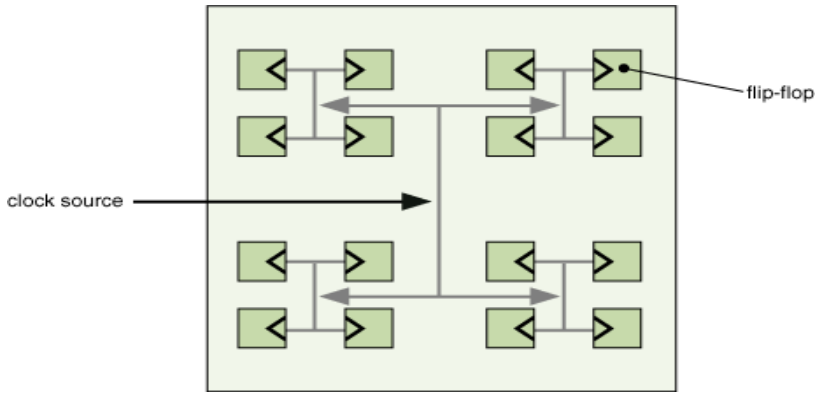

Figure 9.1 - Clock tree Synthesis [18]

This chapter provides detailed information about the IC Compiler clock tree synthesis capability. This chapter contains the following sections:

- Prerequisites for Clock Tree Synthesis
- Analyzing the Clock Trees
- Defining the Clock Trees
- Specifying Clock Tree Exceptions
- Specifying the Clock Tree References
- Defining Clock Cell Spacing Rules

- Specifying Clock Tree Synthesis Options
- Specifying Clock Tree Optimization Options
- Inserting User-Specified Clock Trees
- Handling Specific Design Characteristics
- Verifying the Clock Trees
- Implementing the Clock Trees
- Implementing Clock Meshes
- Using Batch Mode to Reduce Runtime
- Analyzing the Clock Tree Results
- Fine-Tuning the Clock Tree Synthesis Results
- Analyzing and Refining the Design

9.1 Prerequisites for Clock Tree Synthesis

9.1.1 Design Prerequisites

Before running clock tree synthesis, the design should meet the following requirements:

- The design is placed and optimized.

  Use the *check_legality -verbose* command to verify that the placement is legal. The estimated QoR for the design should meet your requirements before you start clock tree synthesis. This includes acceptable results for:

- Congestion

  If congestion issues are not resolved before clock tree synthesis, 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.
- Timing
- Maximum capacitance
- Maximum transition time

To ensure that the clock tree can be routed, verify that the placement is such that the clock sinks are not in narrow channels and that there are no large blockages between the clock root and its sinks. If these conditions occur, fix the placement before running clock tree synthesis.

- The power and ground nets are prerouted.
- High-fanout nets, such as scan enables, are synthesized with buffers. [15]

9.1.2 Library Prerequisites

Also libraries must meet the following requirements:

- Any cell in the logic library that you want to use as a clock tree reference (a buffer or inverter cell that can be used to build a clock tree) or for sizing of gates on the clock network must be usable by clock tree synthesis and optimization. By default, clock tree synthesis and optimization cannot use buffers and inverters that have the *dont_use* attribute to build the clock tree.
- The physical library should include

  All clock tree references (the buffer and inverter cells that can be used to build the clock trees). Routing information, which includes layer information and non-default routing rules.
- Tuples models must exist.

  Extraction requires these models to estimate the net resistance and capacitance. [15]

9.2 Analyzing the Clock Trees

Before running clock tree synthesis, analyze each clock tree to determine its characteristics and its relationship to other clock trees in the design. For each clock tree, determine

- What the clock root is?
- What the desired clock sinks and clock tree exceptions are?
- Whether the clock tree contains preexisting cells, such as clock-gating cells.

- Whether the clock tree converges, either with itself (a convergent clock path) or with another clock tree (an overlapping clock path).
- Whether the clock tree has timing relationships with other clock trees in the design, such as interclock skew requirements.
- What the logical design rule constraints (maximum fanout, maximum transition time, and maximum capacitance) are?
- What the routing constraints (routing rules and metal layers) are?

9.3 Defining the Clock Trees

IC Compiler uses the clock sources defined by the *create_clock* command as the clock roots and derives the default set of clock sinks by tracing through all cells in the transitive fanout of the clock roots. To disallow clock sources defined on a hierarchical pin, set the *cts_enable_clock_at_hierarchical_pin* variable to false before using the *create_clock* command. This variable is true by default. [15]

9.3.1 Cascaded Clocks

If a nested clock tree has its own source, IC Compiler considers the source pin of the driven clock to be an implicit exclude pin of the driving clock. Sinks of the driven clock are not considered sinks of the driving clock. To verify that the clock sources are correctly defined, use the *check_clock_tree* command. [15]

9.3.2 Cascaded Generated Clocks

If a nested clock tree has a generated source, IC Compiler traces back to the master-clock source from which the generated clock is derived and considers the sinks of the generated clock to be the sinks of the driving clock tree. Incorrectly defining the master-clock source, results in poor skew and timing QoR.

If IC Compiler cannot trace back to the master-clock source, the tool cannot balance the sinks of the generated clock with the sinks of its source. If the master-clock source is not a clock source defined by the *create_clock* or *create_generated_clock* command, IC Compiler cannot synthesize a clock tree for the generated clock or its source. Use the

check_clock_tree command to verify that your master-clock sources are correctly defined. [15]

9.3.3 Identifying the Clock Tree Endpoints

Clock paths have two types of endpoints:

- **Stop pins:** Stop pins are the endpoints of the clock tree that are used for delay balancing. During clock tree synthesis, IC Compiler uses stop pins in calculations and optimizations for both design rule constraints and clock tree timing (skew and insertion delay). Stop pins are also referred to as sink pins.
- **Exclude pins**: Exclude pins are clock tree endpoints that are excluded from clock tree timing calculations and optimizations. IC Compiler uses exclude pins only in calculations and optimizations for design rule constraints.

Verify that the default sink pins (implicit stop pins), implicit nonstop pins, and implicit exclude pins are accurate by generating a clock tree exceptions report. If the default sink pins, implicit nonstop pins, and implicit exclude pins are correct, you are done with the clock tree exception definition. Otherwise, first identify any timing settings, such as disabled timing arcs and case analysis settings, that affect the clock tree traversal. To identify disabled timing arcs in your design, use the *report_disable_timing* command. To identify case analysis settings in your design, use the *report_case_analysis* command. Remove any timing settings that cause an incorrect clock tree definition. [15]

9.3.4 Analyzing Clock Sink Groups

A clock sink group is a group of clock sinks driven directly by a single net. The sink group assumes the net name. Sink groups can have timing relationships when an endpoint in a sink group has one or more startpoints or endpoints in another sink group. Each startpoint-and-endpoint pair forms one timing relationship path.

To report sink groups and their timing relationships, specify the *-sink_group* option with the *report_clock_tree* command. Sink groups can overlap because each sink group can be

in more than one clock domain. When sink group overlapping occurs, the *report_clock_tree -sink_group* command issues a warning message by default. To suppress the warning message, set the *timing_enable_multiple_clocks_per_reg* variable to true, changing it from its default of false.

9.3.5 Ignoring Clock Tree Exceptions

By default, the *report_clock_tree -sink_group* command respects the clock tree exceptions when traversing the clock tree network. To ignore the clock tree exceptions, specify the *-sink_group_ignore_cts_exceptions* option. The following restrictions apply:

- When an explicit stop pin or a float pin is not an endpoint of a timing relationship, the tool does not count the timing relationship.
- The tool does not support per-clock exceptions on sinks.

9.3.6 Ignoring Buffers and Inverters

- By default, the *report_clock_tree -sink_group* command does not allow the driving nets to pass through preexisting buffers and inverters.

9.3.7 Defining the Clock Root Attributes

If the clock root is an input port (without an I/O pad cell), must accurately specify the driving cell of the input port. A weak driving cell does not affect logic synthesis, because logic synthesis uses ideal clocks. However, during clock tree synthesis, a weak driving cell can cause IC Compiler to insert extra buffers as the tool tries to meet the clock tree design rule constraints, such as maximum transition time and maximum capacitance. If not specified a driving cell (or drive strength), IC Compiler assumes that the port has infinite drive strength. If the clock root is an input port with an I/O pad cell, must accurately specify the input transition time of the input port. [15]

9.4 Specifying Clock Tree Exceptions

To define clock tree exceptions, use the *set_clock_tree_exceptions* command. You can set clock tree exceptions on pins or hierarchical pins. If a pin is on paths in multiple clock domains, you can define different clock tree exceptions for each clock domain. By default, while using the *set_clock_tree_exceptions* command, the clock tree exceptions apply to all clocks for all multicorner-multimode scenarios. When you use the *set_clock_tree_exceptions* command without the -clocks option, the tool

- Applies the clock tree exceptions to all clocks for all multicorner-multimode scenarios. To specify clock tree exceptions for the current scenario only, use the -*current_scenario option*. Do not use the -*current_scenario* option with the -*clocks* option.
- Defines the clock tree exceptions for all master clocks that contain the paths of the pin, including the paths of the generated clocks.

To see the clock tree exceptions defined in your design, generate a clock tree exceptions report by running the *report_clock_tree -exceptions* command. One can remove clock tree exceptions based on how they are defined, If a clock tree exception is defined without the -clocks option, use the *remove_clock_tree_exceptions* command without the -clocks option. If a clock tree exception is defined with the -clocks option, use the *remove_clock_tree_exceptions -clocks* command. [15]

9.4.1 Precedence of Clock Tree Exceptions

Issue the *set_clock_tree_exceptions* command multiple times for the same pin, the pin keeps the highest-priority exception. IC Compiler prioritizes the clock tree pin exceptions in the following order:

- Nonstop pins:
  To specify a nonstop pin, use the *set_clock_tree_exceptions -non_stop_pins* command.
- Exclude pins:

To specify an exclude pin, use the *set_clock_tree_exceptions -exclude_pins* command.

- Float pins:

  To specify a float pin and its timing characteristics, use the following *set_clock_tree_exceptions* options:

  - *-float_pins [get_pins pin_list]*
  - *-float_pin_max_delay_fall max_delay_fall_value*
  - *-float_pin_max_delay_rise max_delay_rise_value*
  - *-float_pin_min_delay_fall min_delay_fall_value*
  - *-float_pin_min_delay_rise min_delay_rise_value*
  - *-float_pin_logic_level logic_level_value*

- Stop pins:

  To specify a stop pin, use the *set_clock_tree_exceptions -stop_pins* command.

When multiple exceptions are defined on the same pin, the *set_clock_tree_exceptions* command with the *-clocks* option always overrides the command without the *-clocks* option. The same rule applies even when the command without the -clocks option sets a higher-priority exception than the command with the -clocks option. [15]

9.5 Specifying the Clock Tree References

IC Compiler uses four clock tree reference lists:

- One for clock tree synthesis.
- One for boundary cell insertion.
- One for sizing.
- One for delay insertion

By default, each clock tree reference list contains all the buffers and inverters in your technology library. To fine-tune the results, one can restrict the set of buffers and inverters used for one or more of these operations. To define a clock tree reference list, use the *set_clock_tree_references* command. When a clock tree reference list is defined,

ensure that the buffers and inverters are specified with a wide range of drive strengths, so that clock tree synthesis can select the appropriate buffer or inverter for each cluster. [15]

9.6 Defining Clock Cell Spacing Rules

Clock cells consume more power than cells that are not in the clock network. Clock cells that are clustered together in a small area increase current densities for the power and ground rails, where a potential electromigration problem might occur. One way to avoid the problem is to set spacing requirements between clock cells. Set the spacing requirements by defining clock cell spacing rules for inverters, buffers, and integrated clock-gating cells in the clock network. Thus, one can prevent local clumping of the clock cells along a standard cell power rail between the perpendicular straps.

To define clock cell spacing rules, use the *set_clock_cell_spacing* command.

To report clock cell spacing rules, use the *report_clock_cell_spacing* command.

To remove clock cell spacing rules, use the *remove_clock_cell_spacing* command. [15]

9.7 Specifying Clock Tree Synthesis Options

To define clock tree synthesis options, use the *set_clock_tree_options* command.

9.7.1 Specifying the Clock Tree Synthesis Goals

The optimization goals used for synthesizing the design and the optimization goals used for synthesizing the clock trees might differ. Perform the following steps to ensure the proper constraints usage:

- Set the clock tree design rule constraints.
- Set the clock tree timing goals.

IC Compiler prioritizes the clock tree synthesis optimization goals as follows:

- Design rule constraints.

a. Meet maximum capacitance constraint

b. Meet maximum transition time constraint

c. Meet maximum fanout constraint.

- Clock tree timing goals
  a. Meet maximum skew target

  b. Meet minimum insertion delay target

9.7.2 Setting Clock Tree Design Rule Constraints

IC Compiler supports the following design rule constraints for clock tree synthesis:

- Maximum capacitance (*set_clock_tree_options -max_capacitance*)

  If this constraint is not specified, the clock tree synthesis default is 0.6 pF.
- Maximum transition time (*set_clock_tree_options -max_transition*)

  If this constraint is not specified, the clock tree synthesis default is 0.5 ns.
- Maximum fanout (*set_clock_tree_options -max_fanout*)

  If this constraint is not defined, the clock tree synthesis default is 2000.

9.7.3 Setting Clock Tree Timing Goals

During clock tree synthesis, IC Compiler considers only the clock tree timing goals. It does not consider the latency (as specified by the *set_clock_latency command*) or uncertainty (as specified by the *set_clock_uncertainty* command).

One can specify the following clock tree timing goals for a clock tree:

- Maximum skew (*set_clock_tree_options -target_skew*)
- Minimum insertion delay (*set_clock_tree_options -target_early_delay*).

9.7.4 Setting Clock Tree Routing Options

IC Compiler allows to specify the following options to guide the clock tree routing:

- Which routing rule (type of wire) to use.

  Specifying Routing Rules:

If not specified which routing rule to use for clock tree synthesis, IC Compiler uses the default routing rule (default wires) to route the clock trees.

To see the current routing rule definitions, run the *report_routing_rules* command. To set the clock tree routing rule, use the *set_clock_tree_options -routing_rule* command.

- Which clock shielding methodology to use.

  Shielding Clock Nets:

  IC Compiler implements clock shielding using nondefault routing rules. One can choose either to shield clock nets before routing signal nets or vice versa.

  To define nondefault routing rules for clock shielding, use the *define_routing_rule* command. The syntax is *define_routing_rule rule_name*

  [-snap_to_track]

  [-shield_spacings shield_spacings]

  [-shield_widths shield_widths]

- Which routing layers to use.

  Specifying Routing Layers

  If not specified which routing layers to use for clock tree synthesis, IC Compiler can use any routing layers. For more control of the clock tree routing, one can specify preferred routing layers by using the *set_clock_tree_options -layer_list* command.

- Which nondefault routing rules to use with which cell types.

  Association of Nondefault Routing Rules With Reference Cells

  Electromigration problems result from an increase in current densities, which often occurs when strong cells drive thin nets. Electromigration can lead to opens and shorts due to metal ion displacement caused by the flow of electrons and can lead to the functional failure of the IC device. To prevent these problems in clock networks, associate reference cells with compatible nondefault routing rules by using the *set_reference_cell_routing_rule* command. [15]

9.8 Specifying Clock Tree Optimization Options

IC Compiler can perform several optimization tasks, which can be enabled or disabled by setting the appropriate options.

9.8.1 Controlling Embedded Clock Tree Optimization

To set the optimization options for embedded clock tree optimization, use the *set_clock_tree_options* command.

9.8.2 Controlling Preroute Clock Tree Optimization

During *clock_opt*, the clock tree optimization phase performs all the optimization tasks:
-buffer_relocation: Optimizes the placement of the buffers and inverters in the synthesized clock trees.
-buffer_sizing: Optimizes the sizing of the buffers and inverters in the synthesized clock trees.
-delay_insertion: Inserts delays on clock paths to reduce the clock skew, while at the same time ensuring that the longest clock path does not change.
-gate_relocation: Optimizes the placement of the preexisting gates in the clock trees by moving them closer to the clock sinks. Gates marked as fixed are not moved.
-gate_sizing: Optimizes the sizing of the preexisting gates in the clock trees
One cannot independently control these tasks for clock_opt. To set the optimization options for standalone preroute clock tree optimization, specify the options when running the *optimize_clock_tree* command. [15]

9.8.3 Controlling Postroute Clock Tree Optimization

To set the optimization options for postroute clock tree optimization, specify the options when you run the *optimize_clock_tree* command.

9.9 Inserting User-Specified Clock Trees

IC Compiler supports the use of a clock configuration file to enable user specification of the clock tree structure. The following sections describe how to:

- Read a clock configuration file before clock tree synthesis.

  If one have a configuration file that describes the desired clock tree structure, run the *set_clock_tree_options -config_file_read config_file* command before running clock tree synthesis.

- Reduce skew variation by using RC constraint-based clustering.

  To enable this capability, set the cts*_enable_rc_constraints* variable to true before running clock tree synthesis.

- Save a clock configuration file after clock tree synthesis.

  To save the generated clock tree structure in a clock configuration file after clock tree synthesis, run the *set_clock_tree_options -config_file_*write *config_file* command before running clock tree synthesis.

- Describe clock tree structure in a clock configuration file.

  Use the following syntax to define the structure for each user-specified clock tree in design:

  begin_clock_tree number_of_levels
  clock_net *net_name* [routing_rule *rule_name*]
  [routing_layer_constraints *min_layer max_layer*]
  {buffer_level *reference_cell number_of_buffers*
  [buffer_level_pin *instance*/*pin*]
  [routing_rule *rule_name*]
  [routing_layer_constraints *min_layer max_layer*]
  ...}
  ... end_clock_tree
  begin_clock_tree *number_of_levels*

9.10 Handling Specific Design Characteristics

Several design styles might require special considerations during clock tree synthesis. These design styles include:

- Multicorner-multimode designs.

- Hard macro cells.
- Preexisting clock trees.
- Non-unate gated clocks.
- Integrated clock-gating (ICG) cells.
- Multiple clocks (with balanced skew).
- Hierarchical designs.
- Extracted timing models.
- Multivoltage designs.

The following sections describe how to use IC Compiler clock tree synthesis with these design styles.

9.11 Verifying the Clock Trees

Before you synthesize the clock trees, use the *check_clock_tree* command to verify that the clock trees are properly defined. If not specified the -clocks option, IC Compiler checks all clocks in the current design. The *check_clock_tree* command checks for the following issues:

- Hierarchical pin defined as a clock source
- Generated clock without a valid master clock source.

For multicorner-multimode designs, the *check_clock_tree* command checks all scenarios automatically for the following issues:

- Conflicting per-clock exception settings for each scenario
- Conflicting balancing settings in merged scenarios
- Conflicting multicorner-multimode interclock delay balancing settings

9.12 Implementing the Clock Trees

The recommended process for implementing the clock trees in design is to use the *clock_opt* command, which performs clock tree synthesis and incremental physical optimization. This process results in a timing optimized design with fully implemented clock trees.

9.12.1 Optimizing Clock Tree Synthesis Only and Clock Tree Synthesis Hold Only Scenarios

- To enable the clock tree synthesis only scenarios, run the *set_scenario_options* command with the following settings before running the clock_opt command: icc_shell> **set_scenario_options –cts_mode true –setup false –hold false \ -cts_corner *value* -scenarios *cts_scenario***

  Here, *value* could be any of *max, min, or min_max*.
- To enable the clock tree synthesis hold only scenarios, run the set_scenario_options command before running the clock_opt command: icc_shell> **set_scenario_options –cts_mode true –setup false –hold true \ -cts_corner *value* -scenarios *cts_scenario***

  Here, *value* could be any of *max, min, or min_max*.

9.12.2 Analyzing Optimization Feasibility After Clock Tree Synthesis

After you finish implementing the clock trees, evaluate the design constraints for setup and hold time by analyzing optimization feasibility. Perform the feasibility analysis at an early design stage to fine-tune the design constraints for optimization. To perform the feasibility analysis, use the *clock_opt_feasibility* command with the *-only_psyn* option. Running optimization feasibility provides the following benefits:

- Improves timing QoR by resolving setup time violations and performing hold time fixing analysis.
- Reduces the runtime relative to the comparable *clock_opt* flow for optimization.

9.12.3 Standalone Clock Tree Synthesis Capabilities

Using the *clock_opt* command is the recommended method for performing clock tree synthesis and optimization with IC Compiler. However, in cases where finer control is required, IC Compiler also provides the following standalone clock tree synthesis capabilities:

- Clock tree power optimization
- Clock tree synthesis
- High-fanout net synthesis
- Clock tree optimization
- Interclock delay balancing
- I/O timing adjustment
- Clock Tree Routing

9.12.4 Performing Clock Tree Synthesis

Clock tree synthesis is performed during the *clock_opt* process and can also be run as a standalone process. IC Compiler clock tree synthesis is blockage-aware by default. The blockage-aware capability avoids routing and placement blockages to reduce DRC violations in designs with complex floorplans. Furthermore, it implements clock trees with minimum clock insertion delay, small clock skew, low buffer count, and small clock cell area to produce the best quality of results (QoR). During clock tree synthesis, IC Compiler:

- Upsizes and moves the existing clock gates, which can improve the QoR and reduce the number of clock tree levels. To prevent upsizing of specific cells during this process, use the *set_clock_tree_exceptions -dont_size_cells* command.
- Inserts buffers and inverters to build clock trees that meet the clock tree design rule constraints, while balancing the loads and minimizing the clock skew.
- Fixes DRC violations beyond clock exceptions without balancing the skew if the *cts_fix_drc_beyond_exceptions* variable is set to *true* (the default).

- Builds a blockage map infrastructure per voltage area to identify whether a location is blocked for routing or placement, so the legalizer can move buffers to the nearest unblocked locations toward clock sources.
- Locates the shortest blockage-avoiding route path from a startpoint to an endpoint with minimum delay to prevent DRC violations.

If design has logical hierarchy, IC Compiler uses the lowest common parent of a buffer's fanout pins to determine where to insert the buffers.

- If the lowest common parent is not the top level of the design, the buffer is inserted in the lowest common parent.
- If the lowest common parent is the top level of the design, the buffer is inserted in the block that contains the driving pin of the buffer.

To perform standalone clock tree synthesis, use the *compile_clock_tree* command. [15]

9.12.5 Performing Clock Tree Optimization

Clock tree optimization improves the clock skew and clock insertion delay by applying additional optimization iterations. Clock tree optimization is performed during the *clock_opt* process and can also be run as a standalone process before clock routing, after clock tree routing, or after detail routing.

To perform standalone clock tree optimization, use the *optimize_clock_tree* command. IC Compiler provides the following incremental optimization capabilities:

- Buffer relocation by using the *-buffer_relocation option*.
- Buffer sizing by using the *-buffer_sizing option*.
- Delay insertion by using the *-delay_insertion option*
- Gate relocation by using the *-gate_relocation option.*
- Gate sizing by using the *-gate_sizing option.*

9.12.6 Performing Clock Routing

After you finish clock tree optimization, you can perform clock routing using either the default router, Zroute, or the classic router. Both Zroute and the classic router support the integrated clock global router and balanced-mode routing. To achieve the best correlation results, IC Compiler uses the integrated clock global router and saves clock global routing information. When using Zroute (the default router), use the *route_zrt_group -all_clock_nets* command to perform clock routing. Must also use the *-reuse_existing_global_route* option so that Zroute detects the clock global routing information in the Milkyway database and performs incremental global routing. [15]

9.13 Implementing Clock Meshes

Clock meshes are homogeneous shorted grids of metal that are driven by many clock drivers. The purpose of a clock mesh is to reduce clock skew in both nominal designs and designs across variations such as on-chip variation (OCV), chip-to-chip variation, and local power fluctuations. A clock mesh reduces skew variation mainly by shorting the outputs of many clock drivers. [15]

9.13.1 Prerequisites for Creating Clock Meshes

Before running clock mesh commands, design should meet the following requirements:

- The design should be mesh-conducive. A basic mesh-conducive design contains at least one high-fanout clock net that has no more than two levels below the proposed mesh. If necessary, use the *remove_clock_gating* command in Power Compiler or the *flatten_clock_gating* command in IC Compiler to flatten the circuitry under the proposed mesh.
- The design should have enough room to place mesh drivers near the mesh loads for driving the mesh optimally.
- To analyze clock mesh circuits, you must have the NanoSim or HSIM transistor models for all the clock mesh gates. A circuit simulator is needed because static timing tools cannot handle clock meshes.

- One should be able to run NanoSim from the shell where you invoke the IC Compiler. [15]

9.13.2 Implementing Hierarchical Clock Meshes

Compiler clock mesh technology supports hierarchical designs by using ILMs. During the block-level flow for your hierarchical design, you can also create a clock mesh for the clocks inside a block. The timing information of the block is available at the top level with the use of ILMs. Block abstraction models are not supported by the hierarchical clock mesh flow.

9.13.3 Prerequisites for the Hierarchical Clock Mesh Flow

The IC before to start creating a clock mesh in a block, make sure to complete the following steps.

- Complete the hierarchical design planning of the design, and create the Milkyway design libraries for the top-level and lower-level block.
- Select a block that is clock mesh conducive.

9.13.4 Procedures to Create Hierarchical Clock Mesh

To create a clock mesh for the clocks inside a block:

- Complete the hierarchical design planning, open the block, and complete the placement process.
- Create the clock mesh by using the clock mesh flow.
- Run the *analyze_subcircuit* command to analyze the clock mesh.
- Use the *create_ilm* command to generate the ILM for the block. The timing information is saved in the ILM view.
- Create the FRAM view for the block to be used at the top level.

- Return to the top level of the design and include the block with its FRAM and ILM views.
- Propagate block-level timing to the top level and then proceed to complete the top-level flow. This is a required step in the hierarchical clock mesh flow. [15]

9.14 Using Batch Mode to Reduce Runtime

While running multiple *compile_clock_tree* commands on your design, the overhead of repeated clock tree synthesis preprocessing and postprocessing increases runtime. To reduce the *compile_clock_tree* runtime, you can enable the clock tree synthesis batch mode by using the *set_cts_batch_mode* command. The *reset_cts_batch_mode* command disables batch mode and invokes the postprocessing cleanup of excess clock trees that were inserted. The *report_cts_batch_mode* command displays whether the flow is in batch mode or normal mode. Using batch mode for the *compile_clock_tree* command saves an average of 15 percent in runtime with a minimum impact on QoR.To use batch mode with the *clock_opt* command, set the *cts_clock_opt_batch_mode* variable to *true.* [15]

9.15 Analyzing the Clock Tree Results

After synthesizing the clock trees, analyze the results to verify that they meet your requirements. Typically the analysis process consists of the following tasks:

- Analyzing the clock tree reports
- Analyzing the clock tree timing
- Reporting QoR
- Verifying the placement of the clock instances,

If the clock trees meet the requirements, then its ready to analyze the entire design for quality of results. If the synthesis results do not meet your requirements, IC Compiler can help you debug the results by outputting additional information during clock tree synthesis.

9.16 Fine-Tuning the Clock Tree Synthesis Results

IC Compiler supports the following methods for fine-tuning the clock tree synthesis results:

- Using the Useful Skew Technique
- Balancing the Skew Using Skew Groups
- Resynthesizing the Clock Trees
- Modifying the Nondefault Routing Rule
- Modifying Clock Trees in the GUI

9.17 Analyzing and Refining the Design

After getting the satisfied results with the clock tree synthesis results, analyze the QoR of the entire design by reporting on constraints (use the *report_constraint* command) and timing (use the *report_timing* command). Use the reports to check the following parameters:

- Worst negative slack (WNS)
- Total negative slack (TNS)
- Design rule constraint violations [15]

CHAPTER 10: Routing Using Zroute

Introduction

This chapter describes the routing capabilities of Zroute, which is the default router for all IC Compiler packages, with the exception of the ICC-XP package. Zroute is architected for multicore hardware and efficiently handles advanced design rules for 45 nm and below technologies and design-for-manufacturing (DFM) tasks. [15]

10.1 Tasks to be performed during Routing

- Zroute Features
- Basic Zroute Flow
- Prerequisites for Routing
- Checking Routability
- Setting Up for Routing
- Routing Clock Nets
- Routing Critical Nets
- Routing Signal Nets
- Performing ECO Routing
- Cleaning Up Routed Nets
- Saving the Routing Information
- Analyzing the Routing Results
- Routing Nets and Buses in the GUI
- Postroute RC Extraction

10.2 Zroute Features

Zroute has five routing engines: global routing, track assignment, detail routing, ECO routing, and routing verification. You can invoke global routing, track assignment, and detail routing by using the *route_opt* core command; by using task-specific commands; or by using an automatic routing command.

Zroute includes the following main features.

- Multithreading on multicore hardware for all routing steps, including global routing, track assignment, and detail routing.
- A realistic connectivity model where Zroute recognizes electrical connectivity if the rectangles touch; it does not require the center lines of wires to connect.
- A dynamic maze grid that permits Zroute to go off-grid to connect pins, while retaining the speed advantages of gridded routers.
- A polygon manager, which allows Zroute to recognize polygons and to understand that design rule checks (DRCs) are aimed at polygons.
- Concurrent optimization of design rules, antenna rules, wire optimization, and via optimization during detail routing.
- Concurrent redundant via insertion during detail routing.
- Support for soft rules built into global routing, track assignment, and detail routing.
- Timing- and crosstalk-driven global routing, track assignment, detail routing, and ECO routing.
- Intelligent design rule handling, including merging of redundant design rule violations and intelligent convergence.
- Net group routing with layer constraints and nondefault routing rules.
- Clock routing.
- Route verification.
- Optimization for DFM and design-for-yield (DFY) using a soft rule approach. [15]

10.3 Basic Zroute Flow

Figure 10-1 shows the basic Zroute flow, which includes clock routing, signal routing, DFM optimizations, and route verification.

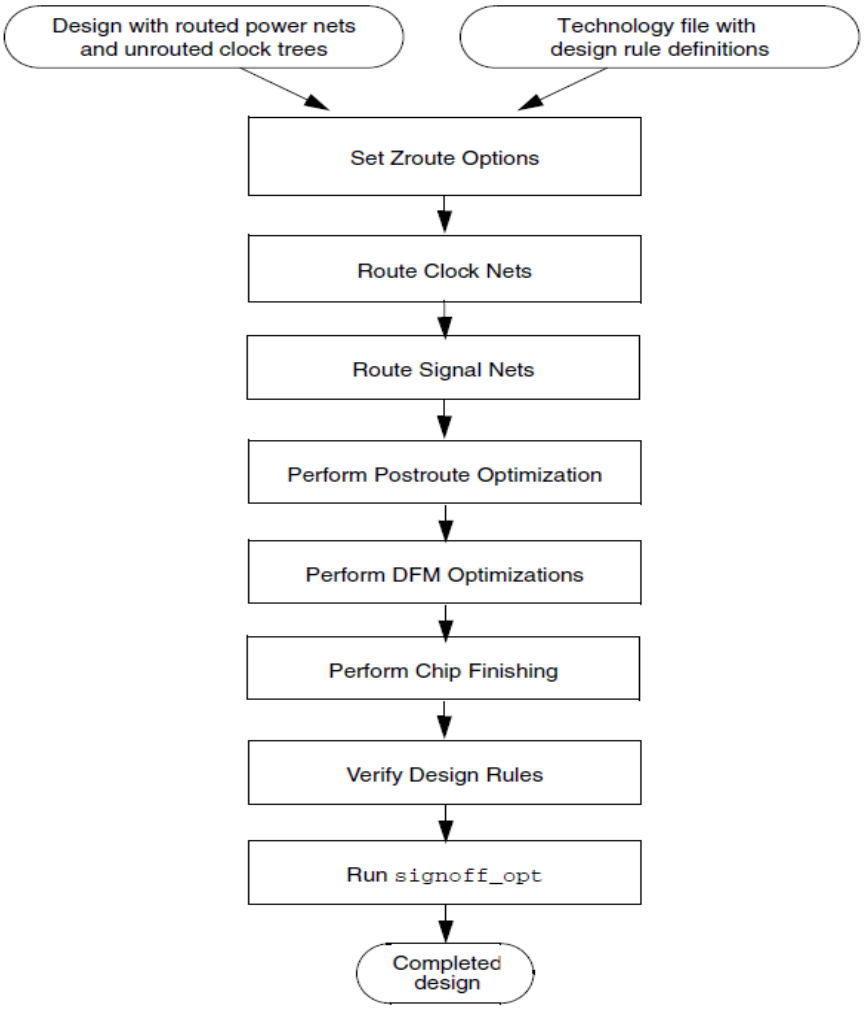

Figure 10.1 - Basic Zroute Flow [15]

10.4 Prerequisites for Routing

Before running Zroute, one must ensure that the design and physical library meet the following requirements

• Library requirements:

Zroute gets all of the design rule information from the Milkyway technology file; therefore, one must ensure that all design rules are defined in the technology file before starting routing.

In addition, Zroute uses only default vias for nonpin access. Make sure that all vias that one want Zroute to use are defined as default vias (*isDefaultContact* attribute is 1) in the technology file.

• Design requirements:

1. Before performing routing, the design must meet the following conditions.
2. Power and ground nets have been routed after design planning and before placement.
3. Clock tree synthesis and optimization have been performed.
4. Estimated congestion is acceptable.
5. Estimated timing is acceptable (about 0 ns of slack).
6. Estimated maximum capacitance and transition have no violations.

10.5 Checking Routability

After placement is completed, one can have IC Compiler check whether the design is ready for detail routing. The tool checks pin access points, cell instance wire tracks, pins out of boundaries, minimum grid and pin design rules, and blockages to make sure they meet design requirements. It creates an error file named after the top-level design (*top_design_name*.err), with a list of violations that you should correct before performing detail routing.

To verify that the design is ready for detail routing, use the *check_routeability* command. [15]

10.6 Setting Up for Routing

The following steps describe how to specify general routing setups for Zroute to use whenever perform routing.

• Enabling Multicore Processing

• Creating Route Guides

• Setting the Preferred Routing Direction

• Controlling Pin Connections

• Using Nondefault Routing Rules

• Specifying the Routing Layers
• Setting Zroute Options
• Setting Signal Integrity Options

• Setting Multivoltage Options

10.7 Routing Clock Nets

One can use Zroute for initial routing of clock nets, redundant via insertion on clock nets, shielding of clock nets, and ECO routing of clock nets, using following sets of commands.

route_zrt_group -all_clock_nets
create_zrt_shield
optimize_clock_tree

10.8 Routing Critical Nets

To route a group of critical nets before routing the rest of the nets in the design, use the *route_zrt_group* command.

10.9 Routing Signal Nets

Before route the signal nets, all clock nets must be routed without violations, using following set of commands.

route_zrt_global
route_zrt_track
route_zrt_detail
route_zrt_auto
set_route_opt_strategy
focal_opt

10.10 Performing ECO Routing

Whenever modify the nets in the design, one need to run engineering change order (ECO) routing to reconnect the routing.

To run ECO routing, use the *route_zrt_eco* command.

10.11 Cleaning Up Routed Nets

After routing is complete, one can clean up the routed nets by running the *remove_zrt_redundant_shapes* command.

10.12 Saving the Routing Information

To save the routing information, use the the *write_route* command. This command generates a script file that contains the Tcl commands to generate the current routing.

10.13 Analyzing the Routing Results

One can analyze the routing results by reporting on the cell placement and routing statistics. The following steps describe how to perform these tasks.

- Reporting Cell Placement and Routing Statistics:
  To view the place and route summary report, run the *report_design_physical -verbose* command.
- Analyzing Congestion:
  To generate a congestion report, run the *report_congestion* command.
- Performing Design Rule Checking Using IC Compiler:
  To use the IC Compiler DRC engine to check the routing design rules defined in the Milkyway technology file, run the *verify_zrt_route* command.
- Performing Signoff Design Rule Checking:
  To perform signoff design rule checking, run the *signoff_drc* command.
- Analyzing DRC Violations:

After having generated a Milkyway error view, either by running one of the Synopsys design rule checking commands, by reading a route guidance file, or by converting the Calibre results to an error view, one can analyze the DRC violations by, using the DRC query command *get_drc_errors* or using the error browser. [15]

10.14 Routing Nets and Buses in the GUI

The layout editor in the IC Compiler GUI provides the following routing capabilities.

- Routing Single Nets
- Creating Physical Buses
- Modifying Buses
- Routing Buses or Multiple Nets
- Creating Custom Wires

10.15 Postroute RC Extraction

IC Compiler automatically performs postroute RC estimation when running the *route_opt* command and when running any of the following timing analysis commands on a routed but not extracted design: *report_timing, report_qor, report_constraint, report_delay_calculation, report_net, report_clock, report_clock_timing,* or *report_clock_tree*. In addition, one an explicitly perform postroute RC extraction by running the *extract_rc* command.

Chapter 11: Chip Finishing and Design For Manufacturing

11.1 Overview:

Figure 11.1 shows the design for manufacturing and chip finishing tasks supported by IC Compiler and how these tasks fit into the overall IC Compiler design flow. The following sections describe how to perform these tasks.

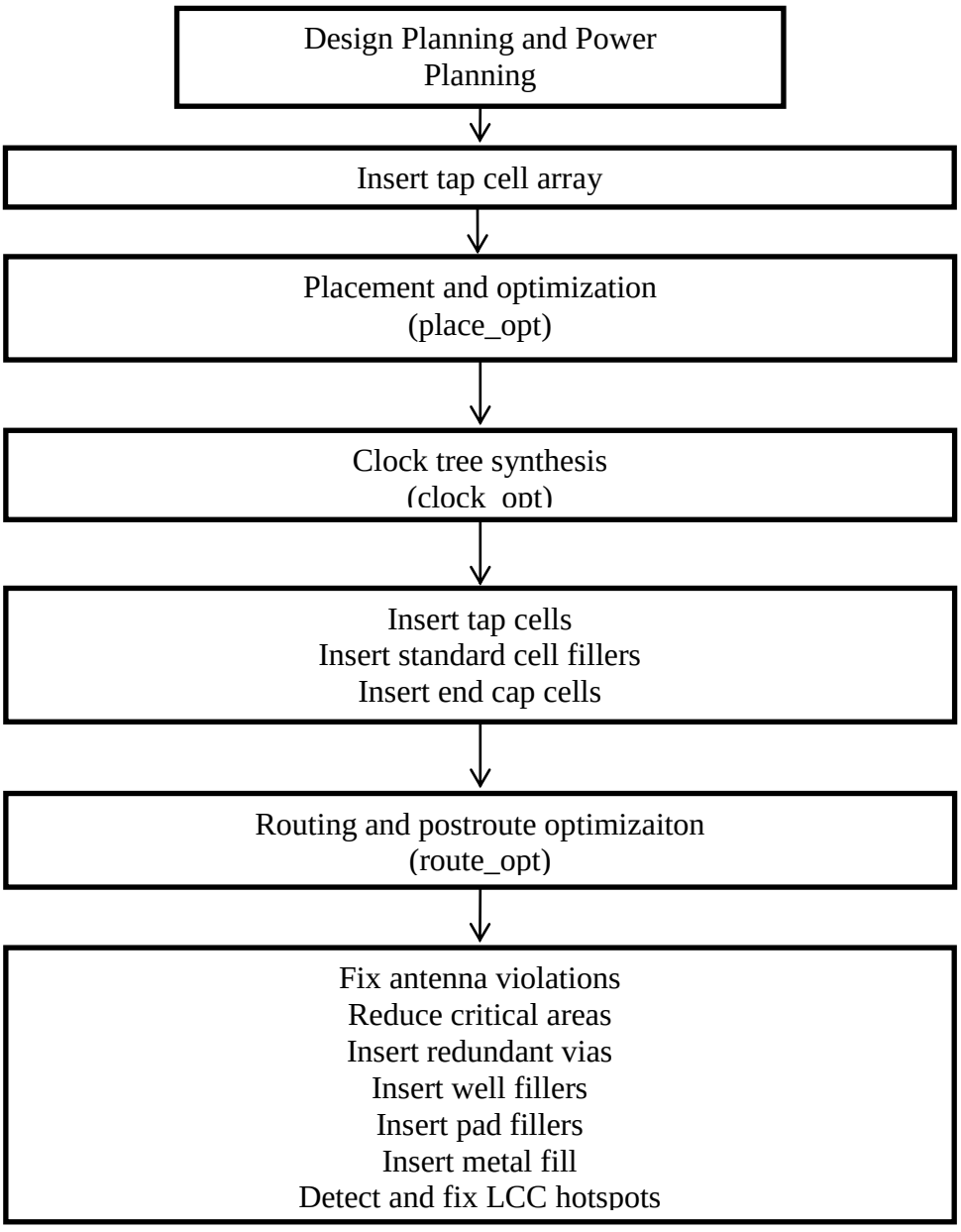

Figure11.1: Design for Manufacturing and Chip Finishing Tasks in the Design Flow

11.2 Inserting Tap Cells

Tap cells are a special non logic cell with well and substrate ties. These cells are typically used when most or all of the standard cells in the library contain no substrate or well taps. Generally, the design rules specify the maximum distance allowed between every transistor in a standard cell and a well or the substrate ties.

- Tap cell arrays can be inserted before placement to ensure that the placement complies with the maximum diffusion-to-tap limit.
- Tap cells can be inserted after placement to fix maximum diffusion-to-tap violations.

11.2.1 Adding Tap Cell Arrays

Before global placement (during the floorplanning stage), you can add tap cells to the design that form a two-dimensional array structure to ensure that all standard cells placed subsequently will comply with the maximum diffusion-to-tap distance limit.

To add a tap cell array, use the add_tap_cell_array command.

11.2.2 Fixing Tap Spacing Violations

You can add tap cells to comply with the diffusion-to-substrate or -well-contact maximum spacing design rule. After global placement (typically), you can insert tap cells “by rules” so that all existing standard cells comply with the maximum diffusion-to-tap distance limit.

To insert tap cells to satisfy to the diffusion-to-tap design rules, use the insert_tap_cells_by_rules command.

11.2.3 Removing Tap Cells

To remove tap cells, use the *remove_stdcell_filler* -tap command.

11.3 Finding and Fixing Antenna Violations

In chip manufacturing, gate oxide can be easily damaged by electrostatic discharge. The static charge that is collected on wires during the multilevel metallization process can damage the device or lead to a total chip failure. The phenomenon of an electrostatic charge being discharged into the device is referred to as either antenna or charge-collecting antenna problems.

The antenna flow consists of the following steps:

- Define the antenna rules
  Define the global metal layer antenna rules by using the define_antenna_rule command.
- Specify the antenna properties of the pins and ports
  Specify the pin and port antenna properties either in a cell library format (CLF) file or by using the set_route_zrt_detail_options command to set the following detail route options:
  • default_diode_protection
  • default_gate_size
  • default_port_external_gate_size
  • default_port_external_antenna_area
  • port_antenna_mode
- Analyze and fix the antenna violations [15]

11.4 Inserting Redundant Vias

Redundant via insertion can be performed in the following ways:

- Postroute redundant via insertion
- Concurrent soft-rule-based redundant via insertion
- Concurrent hard-rule-based redundant via insertion

In general, postroute redundant via insertion should be started. If doing so provides good results, one of the concurrent redundant via insertion methods can be used to improve the redundant via rate. If postroute redundant via insertion results in a redundant via rate of at least 80 percent, the redundant via rate can be improved by using concurrent soft-rule-based redundant via insertion. If postroute redundant via insertion results in a redundant via rate of at least 90 percent, the redundant via rate can be improved by using concurrent hard-rule-based redundant via insertion. [15]

11.5 Reducing Critical Areas

A critical area is a region of the design where, if the center of a random particle defect falls there, the defect causes circuit failure, thereby reducing yield. A conductive defect causes a short fault, and a nonconductive defect causes an open fault.

The following sections describe how to

- Report critical areas
- Display critical area maps
- Reduce critical area short faults by performing wire spreading
- Reduce critical area open faults by performing wire widening

11.6 Shielding Nets

The router shields routed nets by generating shielding wires that are based on the shielding widths and spacing defined in the shielding rules. In addition to shielding nets on the same layer, there is also an option to shield one layer above or one layer below or the layer above and the layer below. Shielding above or below the layer is called coaxial shielding. Coaxial shielding provides even better signal insulation than same-layer shielding, but it uses more routing resources. Shielding can be performed either before or after signal routing. Shielding before signal routing, which is referred to as preroute shielding, provides better shielding coverage but can result in congestion issues during signal routing. Preroute shielding is typically used to shield critical clock nets. Shielding

after signal routing, which is referred to as post-route shielding, has a very minimal impact on routability, but provides less protection to the shielded nets. [15]

11.7 Inserting Filler Cells

Filler cells fill gaps in the design to ensure that all power nets are connected and the spacing requirements are met.

- Before routing, these can be performed
    - Insert standard cell fillers
    - Insert end cap cells
- After routing, these operations can be performed
    - Insert well fillers
    - Insert pad fillers

11.8 Inserting Metal Fill

After routing, empty spaces in the design can be filled with metal wires to meet the metal density rules required by most fabrication processes. Before inserting metal fill, the design should be close to meeting timing and have only a very few or no DRC violations.

Metal fill can be inserted by running the signoff_metal_fill command. This command requires a Hercules license.

11.9 Signal Integrity

Signal integrity is the ability of an electrical signal to carry information reliably and to resist the effects of high-frequency electromagnetic interference from nearby signals. The following conditions can impact signal integrity:

- **Crosstalk**

  Crosstalk is the undesirable electrical interaction between two or more physically adjacent nets due to capacitive coupling. Crosstalk can lead to crosstalk-induced delay changes or static noise.

- **Signal electromigration**

  Electromigration is the permanent physical movement of metal in thin wire connections resulting from the displacement of metal ions by flowing electrons. Electromigration can lead to shorts and opens in wire connections, causing functional failure of the IC device. The problem is more severe in modern technologies due to smaller wire widths and increased current densities.

  IC Compiler supports signal integrity analysis and optimization in either a flat flow or a hierarchical flow (using interface logic models). When creating an interface logic model for use in a hierarchical signal integrity flow, one must use the -include_xtalk option while using run the create_ilm command.

IC Compiler signal integrity analysis and optimization supports multivoltage and multimode-multicorner designs. [15]

Chapter 12: Analysis and Conclusion

- Using Verilog HDL An asynchronous interface for 16-bit wide (parameter default) data is designed using Verilog HDL.
- We have to keep write pointer and read pointer width one bit more than width of address for flag comparison.
- We pass rd_pointer through 2 stage synchronizer flip-flops which run on wr_clk, pass wr_pointer through 2 stage synchronizer flip-flops which run on rd_clk. Before passing this pointer through 2 flip flops they have to be converted into gray-code because only one bit should change at a given point of time between consecutive pointer values. Rise and fall times are different so 0 to 1 transitions and 1 to 0 transitions never occur at same time.
- After passing through the 2-stage synchronizers read pointer and write pointer are converted back into binary and their values are compared to generate full flag and empty flag.
- Using Synopsys Design Compiler, we synthesized top-down. Area and timing reports are shown. We further do post-synthesis simulation and verify the netlist using the previous testbench using NC Verilog simulator.
- We can see discrepancies in post-synthesis simulation like few glitches, can be seen in simulation waveform 4. Flag assertion is verified using NC Verilog simulator.
- In long term we can see some loss of data so frequency drift is an issue in long term. It is not self-corrected. Only thing we can do is by generating signals indicating loss of data or data is not correctly buffered.

- Following are the results obtained for Timing, Area and Power during front-end and back-end design:

	Timing (ns)	**Area**	**Power(uW)**	**Slack (MET)**
Front-end	Write Clock: 5 , Read Clock: 25	6334.464145	**Dynamic:** 65.2868 **Leakage:** 29.4684	Write Clock:0, Read Clock: 3.59
Back-end	Write Clock: 5 , Read Clock: 25	6444.726117	**Dynamic:** 854.0466 **Leakage:** 30.7003	Write Clock:0.7, Read Clock: 3.42

Table 12.1 Table showing the result for timing, area and power.

In DC Complier and IC Compiler, Cb13fs120_tsmc_max 90nm library is used to optimize the design for smallest area, fastest speed and lowest power. It is clear from the above results that the design meets the required setup-time with a slack value of 0.7ns and 3.42ns for write clock and read clock respectively.

Clock gating and other power optimization methods like LPP (Low Power Placement) and GLPO (Gate Level Power Optimization) are implemented in design to reduce dynamic power to 656.779 µw and leakage power to 36.20 µw.

The main objectives of this project were to implement RTL to GDSII flow using Cadence and Synopsys tools. Complete ASIC design is successfully implemented using Cadence NC Verilog Simulator, Synopsys Design Compiler, Synopsys Tetra-Max and Synopsys IC Compiler. Also power optimization and DFT methodologies like logic scan, JTAG boundary scan were implemented using Power Compiler and DFT Compiler respectively. Also, main emphasis is given to physical implementation of standard cell based ASICs using IC Compiler. This part covers the detailed back-end design flow including analyzing and fixing of congestion issues, timing closure and DRC and LVS violations.

This project provided the strong foundation to excel career as an ASIC designer, backed-up with sufficient knowledge and exposure to complete ASIC design flow, with hands on experience of the above mentioned EDA tools.

REFRENCES

1. Michael John Sebastian Smith [June 1997], “Application-Specific Integrated Circuits”, Addison-Wesley Publishing Company, VLSI Design Series.
2. Vishwani D. Agrawal, Srivaths Ravi, “Low-Power Design and Test”, July 2007.
3. http://www.vlsichipdesign.com/index.php/Chip-Design-Articles/analogy-of-vlsi-and-building-architecture-a-thought-process.html(11/3/2011)
4. Himanshu Bhatnagar, “Advanced ASIC chip Synthesis Using Synopsys Design Compiler, Physical Compiler and Primetime” 2004.
5. JOHN DAINTITH. "asynchronous interface." A Dictionary of Computing. 2004. Encyclopedia.com. 6 Nov. 2011 <http://www.encyclopedia.com>. (date 11/6/2011)
6. Clifford E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” SNUG 2002 – www.sunburst-design.com/papers/CummingsSNUG2002SJ_ FIFO1.pdf(11/4/2011)
7. Clifford E. Cummings, “Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs,” SNUG 2001 – www.sunburst-design.com/papers/ CummingsSNUG2001SJ_AsyncClk.pdf(11/4/2011)
8. http://asic-soc.blogspot.com/2007/11/asynchronous-fifo-design.html (11/8/2011)
9. "FIFO Architecture,Functions,and Applications" - Texas Instruments(11/8/2011)
10. ASIC Design Flow Tutorial Using Synopsys Tools By Hima Bindu Kommuru, Hamid Mahmoodi, Nano-Electronics & Computing Research Lab, School of Engineering, San Francisco State University San Francisco, CA
11. Synopsys Design Complier User Guide Version D-2010.03, March2010
12. ASIC/IC Design-for-Test Process Guide, Software Version 8.6_1, Mentor Graphics.
13. Synopsys Low-Power Flow User Guide, Version F-2011.09, September 2011
14. Power Compiler User Guide , Version F-2011.09, September 2011
15. Synopsys IC Complier User Guide Version D-2010.03, March2010
16. IC Compiler User Guide Version A-2007.12, December 2007
17. http://semicon.sanyo.com/en/asic/user/cts.php (11/12/2011)
18. Low power implementation of SiAc system controller / by Jess Shi.
19. Implementation of Complete ASIC Design Flow, RTL to GDSII, Maninder Singh

APPENDIX

1. Simulation Waveforms:

a) Behavioral Simulation Waveform showing empty flag asserted, when wr_enable is high after some time:

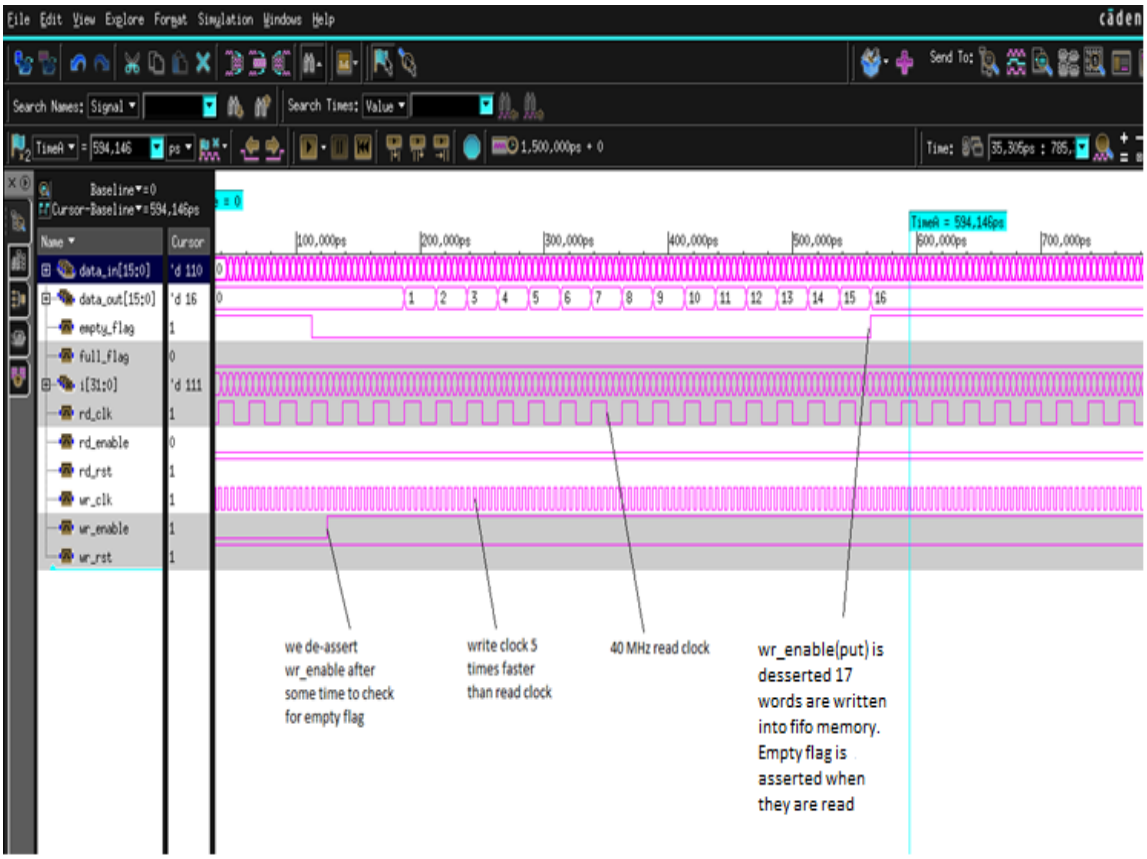

Figure A.1. Waveform showing empty flag asserted, when wr_enable is high after some time

b) Behavioral Simulation Waveform showing full flag remaining high, when rd_enable is high after some time:

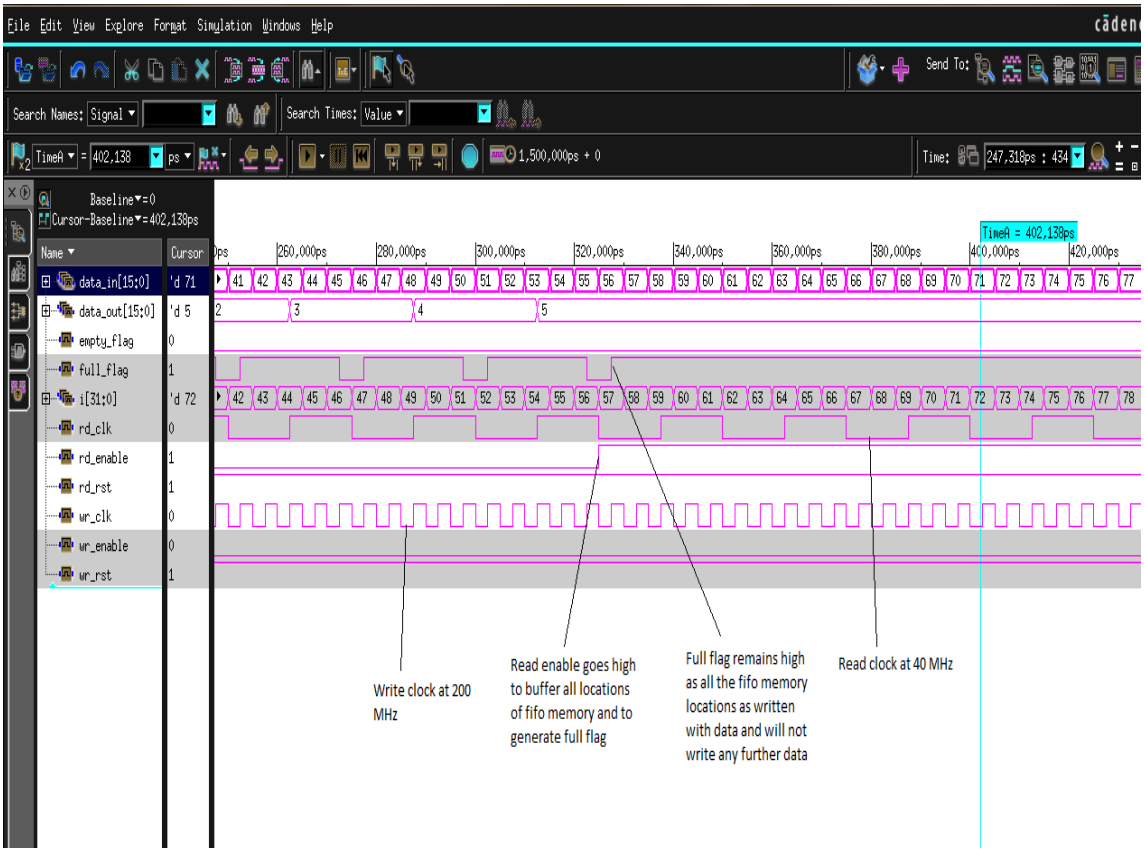

Figure A.2. Waveform showing full flag remaining high, when rd_enable is high after some time

c) Behavioral Simulation Waveform showing fifo memory buffers all the data if wr_enable(put) and rd_enable(get) are high all the time:

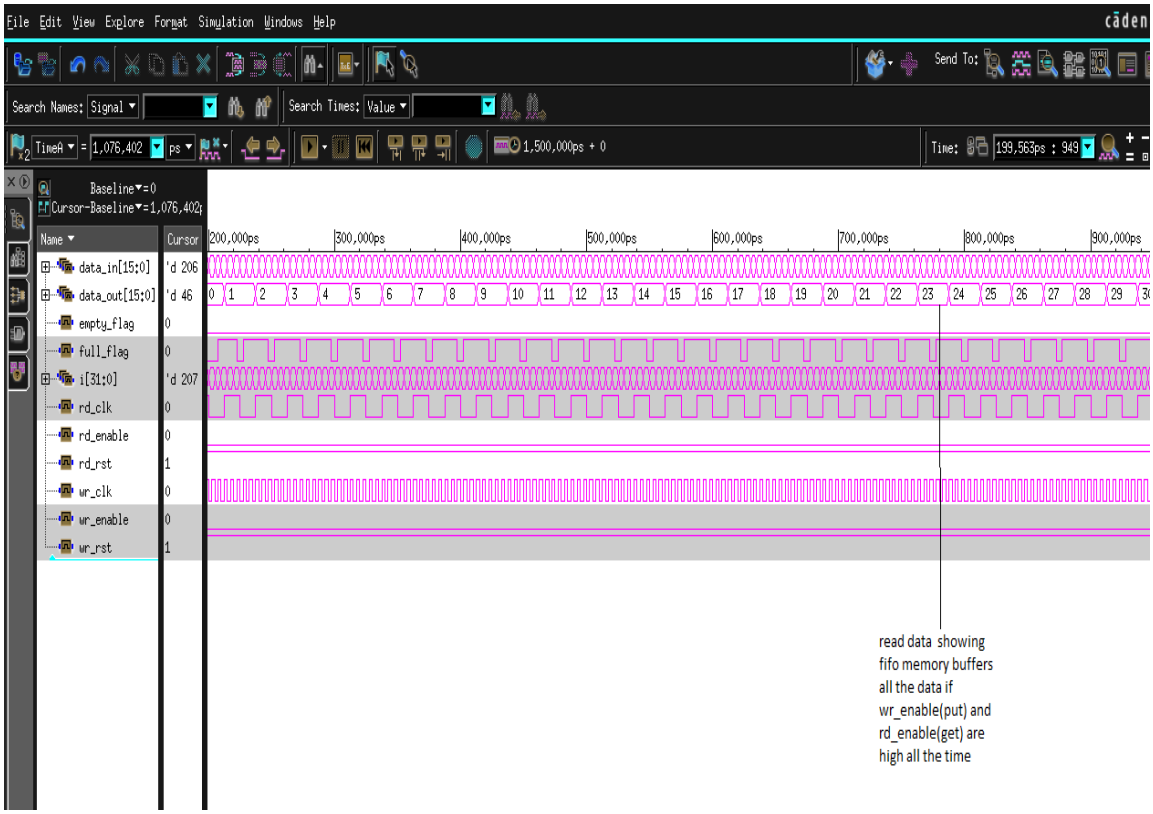

Figure A.3. Waveform showing showing fifo memory buffers all the data if wr_enable(put) and rd_enable(get) are high all the time

d) Post-Synthesis Simulation Waveform showing empty flag asserted, when wr_enable is high after some time:

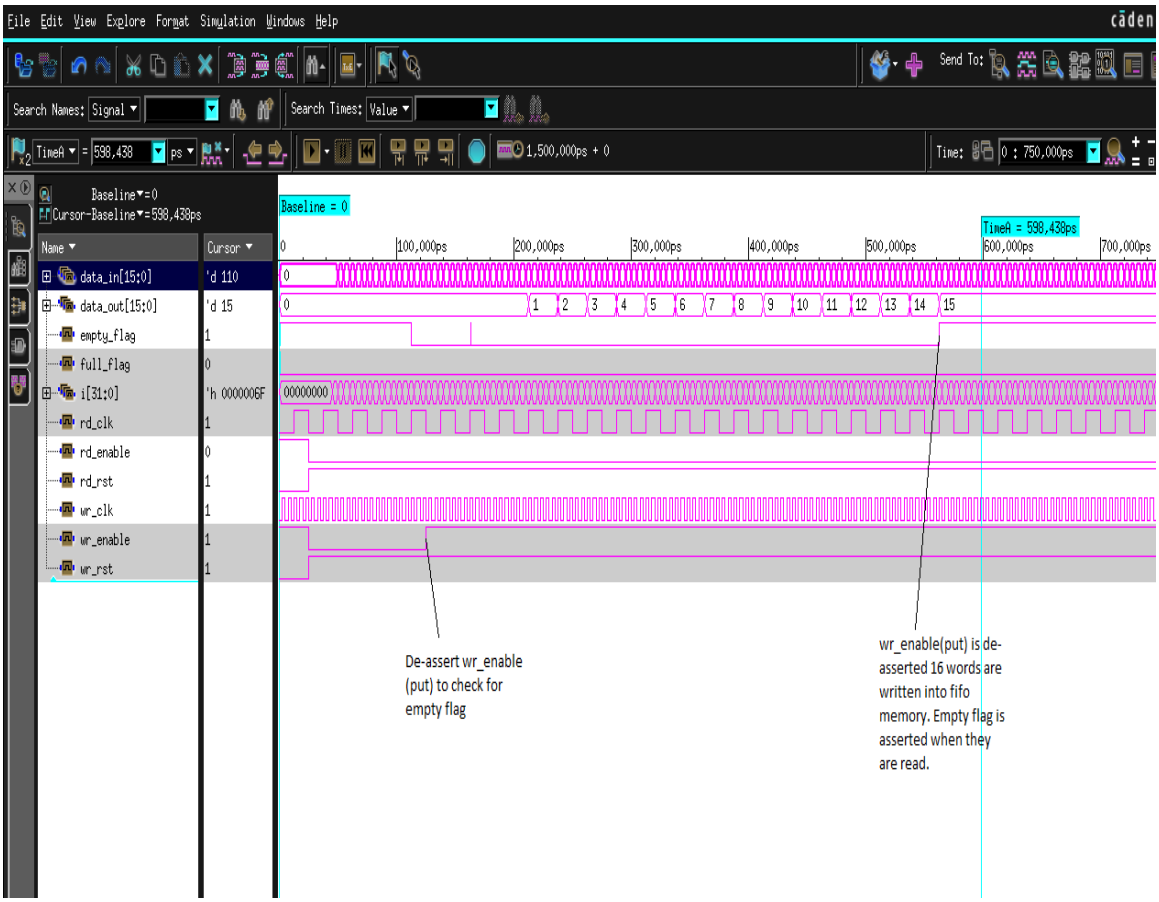

Figure A.4. Post-synthesis simulation waveform showing Post-Synthesis Simulation Waveform showing empty flag asserted, when wr_enable is high after some time

e) Post-Synthesis Simulation Waveform showing full flag remaining high, when rd_enable is high after some time:

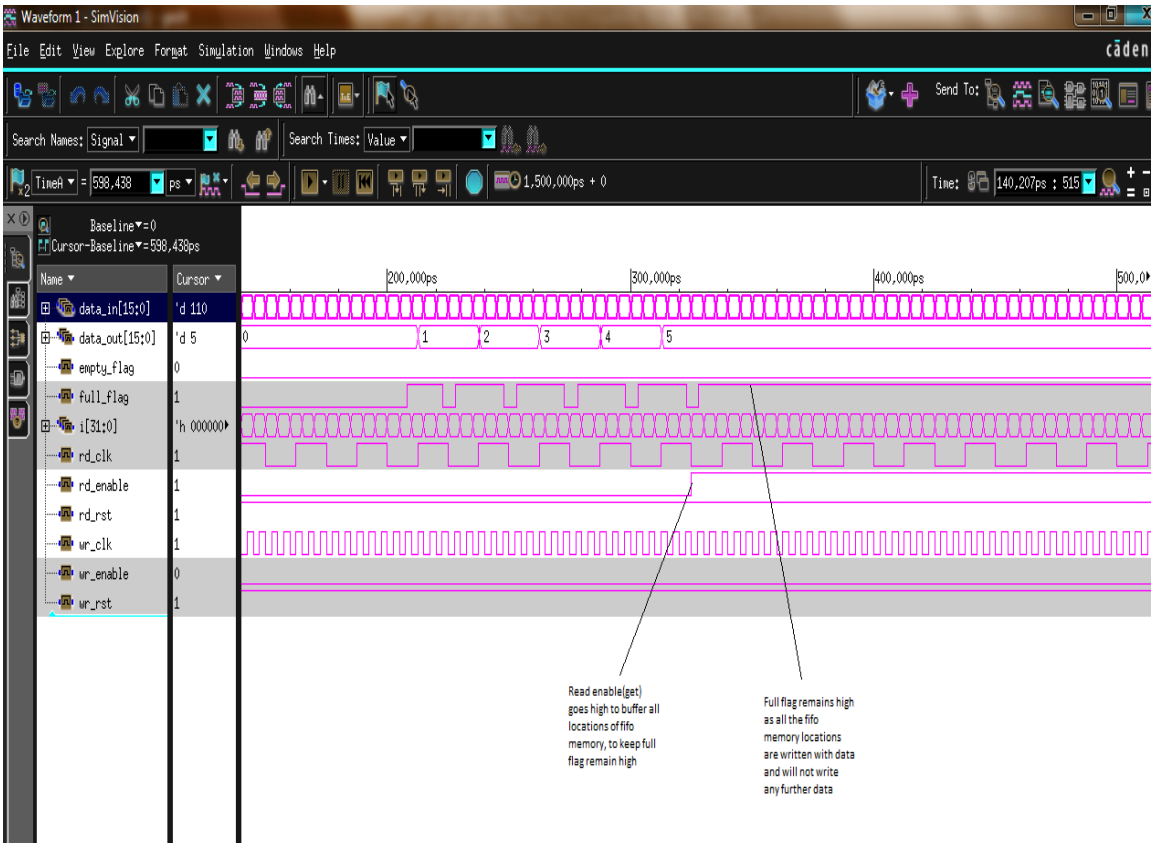

Figure A.5. Post-synthesis simulationwaveform showing full flag remaining high, when rd_enable is high after some time

f) Post-synthesis simulation waveform showing fifo memory buffers all the data if wr_enable(put) and rd_enable(get) are high all the time

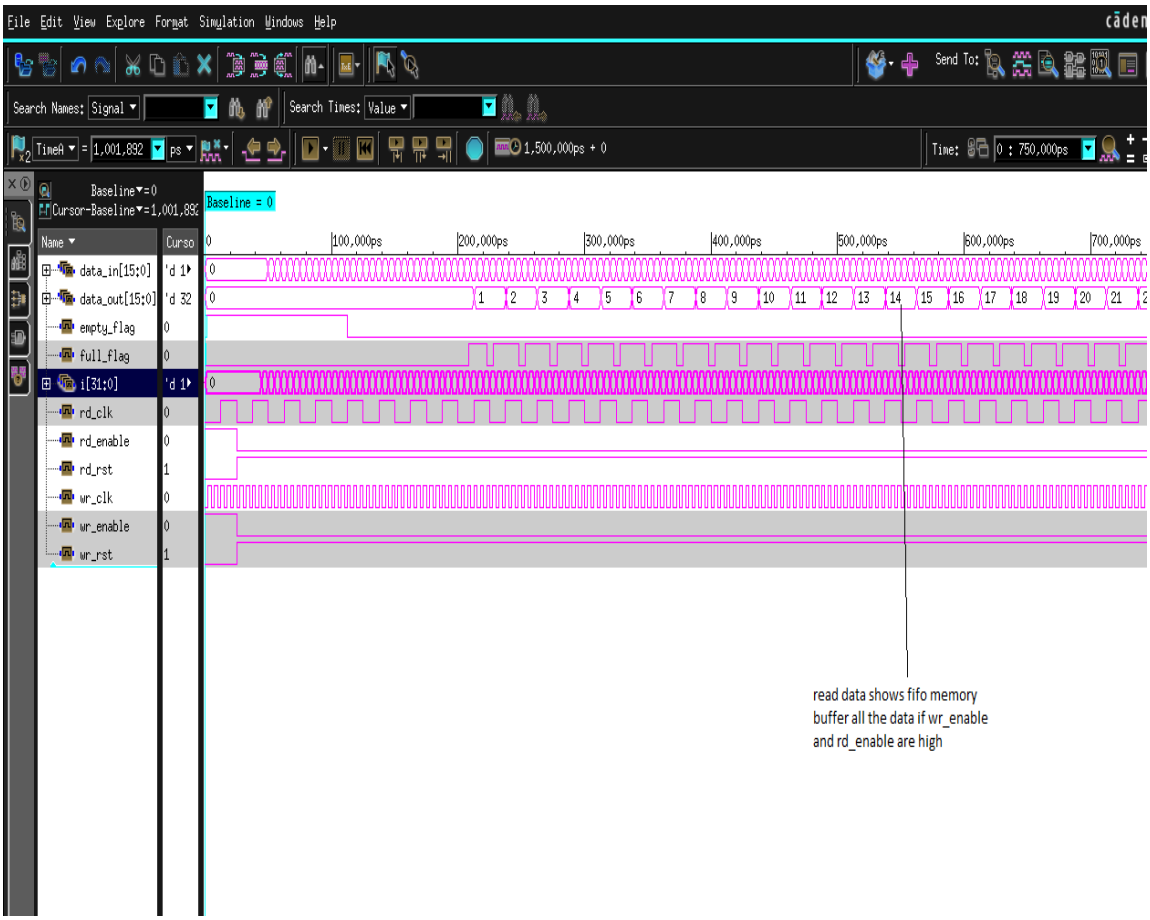

Figure A.6. Post-synthesis simulation waveform showing fifo memory buffer all the data if wr_enable(put) and rd_enable(get) are high all the time

2. Boundary Scan and Logic Scan Insertion Script:

```
###scan chain insertion
set_svf dmx_svffile.svf ;
### setup the scan style
set test_default_scan_style multiplexed_flip_flop ;
set_scan_configuration -create_dedicated_scan_out_ports true ;

#set scan cell  dtrsp_2 ;
#set_scan_register_type -type dtrsp_2 ;

### setup the view
create_port -direction "in"  {Scan_I Scan_E} ;
create_port -direction "out" {Scan_O} ;

set_dft_signal -view spec -type ScanDataIn  -port Scan_I ;
set_dft_signal -view spec -type ScanDataOut -port Scan_O ;
set_dft_signal -view spec -type ScanEnable  -port Scan_E -active_state 1 ;

# set_dft_signal -view spec -type RST -port RST -active_state 1 ;

### RTL-LEVEL DRC (DESIGN RULE CHECK)
###

set_dft_signal -view existing_dft -type ScanClock -port CLK -timing [list 45 55] ;
#set_dft_signal -view existing_dft -type Reset -port RST -active_state 0 ;

#boundary scan insertion
set_bsd_instruction {EXTEST} -code {1100} -reg BOUNDARY
set_bsd_instruction {SAMPLE} -code {1110} -reg BOUNDARY
set_bsd_instruction {PRELOAD} -code {1110} -reg BOUNDARY
set_bsd_instruction {BYPASS} -code {1111} -reg BYPASS
#set_bsd_--#Fields for Version, Part Number and Manufacturer Identity
set_bsd_instruction {IDCODE} -code {1010} -capture_value \
{32'h55555555}
set_bsd_configuration -style synchronous -asynchronous_reset true
set test_default_period 100
```

```
set_dft_signal -type tck -port CLK
set_dft_signal -type trst -port RST -active_state 0
set_dft_signal -type tms -port TMS
set_dft_signal -type tdi -port TDI
set_dft_signal -type tdo -port TDO

# create_test_protocol ;
create_test_protocol -infer_clock -infer_async ;

### rtl-level drc
dft_drc -verbose ;

check_design;

### SCAN SYNTHESIS
###

### one-pass scan synthesis (insert scan-ffs, but not yet routed)

#compile_ultra -incremental -scan ;
#compile_ultra -scan ;
compile -scan ;
#report_constraint -all_violators ;
#compile;

### scan insertion
preview_dft ;
# preview_dft -show all ;

insert_dft;

#insert_dft -physical ;
#insert_dft -ignore_compile_design_rules ;
#insert_dft -no_scan ;

set_scan_configuration -replace false ;
insert_dft ;
```

```
### post-scan drc
dft_drc ;

### SCAN EXTRACTION AND REPORT
###

write_scan_def -output dmx_scandeffile.scandef ;
write_test_protocol -o test_protocol_file.0.stil ;

#check_scan_def                        >  ./report/scan.$timestamp ;
#dft_drc -coverage_estimate                >> ./report/scan.$timestamp ;
#report_dft_configuration                >> ./report/scan.$timestamp ;
#report_scan_configuration                >> ./report/scan.$timestamp ;
#report_dft_signal                        >> ./report/scan.$timestamp ;
#report_autofix_configuration                >> ./report/scan.$timestamp ;
#report_scan_path -view existing_dft -chain all >> ./report/scan.$timestamp ;
#report_scan_path -view existing_dft -cell all  >> ./report/scan.$timestamp ;

report_area   >  area_report.rpt ;
report_timing > timing_report.rpt ;
```

3. Power optimization Script:

```
# Power Optimization Section
set power_driven_clock_gating true
# The following setting can be used to enable global clock gating.
# With global clock gating, common enables are extracted across hierarchies
# which results in fewer redundant clock gates.
#set compile_clock_gating_through_hierarchy true
# clock_gating_style
set_clock_gating_style -sequential_cell latch -control_point before -control_signal
scan_enable
# Apply Power Optimization Constraints
set_max_dynamic_power 0
set_max_leakage_power 0
compile_ultra -scan -gate_clock
```

4. This is the front-end timing report , Write clock runs at 5ns and Read clock runs at 25ns.

```
Information: Updating graph... (UID-83)
Information: Updating design information... (UID-85)

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : fifo_model
Version: E-2010.12-SP4
Date   : Fri Mar 16 18:34:11 2012
****************************************

Operating Conditions: cb13fs120_tsmc_max   Library: cb13fs120_tsmc_max
Wire Load Model Mode: enclosed

  Startpoint: fifo_mem_reg[14][8]
              (rising edge-triggered flip-flop clocked by wr_clk)
  Endpoint: rd_data_reg[8]
            (rising edge-triggered flip-flop clocked by rd_clk)
  Path Group: rd_clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  fifo_model         8000                  cb13fs120_tsmc_max

  Point                                    Incr       Path
  ----------------------------------------------------------
  clock wr_clk (rise edge)                20.00      20.00
  clock network delay (ideal)              0.00      20.00
  fifo_mem_reg[14][8]/CP (dfcrq1)          0.00      20.00 r
  fifo_mem_reg[14][8]/Q (dfcrq1)           0.32      20.32 f
  U1599/Z (aor22d1)                        0.20      20.52 f
  U1600/ZN (nr04d0)                        0.19      20.71 r
  U1601/ZN (oai2222d1)                     0.43      21.14 f
```

rd_data_reg[8]/U4/Z (mx02d0) 0.17 21.31 f
rd_data_reg[8]/D (dfcrb1) 0.00 21.31 f
data arrival time 21.31

clock rd_clk (rise edge) 25.00 25.00
clock network delay (ideal) 0.00 25.00
rd_data_reg[8]/CP (dfcrb1) 0.00 25.00 r
library setup time -0.10 24.90
data required time 24.90
---
data required time 24.90
data arrival time -21.31
---
slack (MET) 3.59

Startpoint: wrclk_rdpointer_reg[4]
(rising edge-triggered flip-flop clocked by wr_clk)
Endpoint: fifo_mem_reg[31][0]
(rising edge-triggered flip-flop clocked by wr_clk)
Path Group: wr_clk
Path Type: max

Des/Clust/Port Wire Load Model Library
---
fifo_model 8000 cb13fs120_tsmc_max

Point Incr Path
---
clock wr_clk (rise edge) 0.00 0.00
clock network delay (ideal) 0.00 0.00
wrclk_rdpointer_reg[4]/CP (dfcrq1) 0.00 0.00 r
wrclk_rdpointer_reg[4]/Q (dfcrq1) 0.31 0.31 f
U1386/Z (mx02d2) 0.28 0.58 f
U1397/ZN (xn02d2) 0.33 0.91 f
U1396/ZN (xn02d2) 0.29 1.21 f
U1395/ZN (xn02d2) 0.33 1.54 f
U1699/ZN (xn02d1) 0.30 1.84 f
U1700/ZN (nr02d0) 0.08 1.91 r
U1406/CO (cg01d1) 0.21 2.12 r

U1405/ZN (inv0d1) 0.05 2.17 f
U1407/CO (cg01d1) 0.18 2.35 f
U1337/ZN (nd12d0) 0.10 2.45 r
U1336/ZN (nd02d1) 0.11 2.56 f
U1403/Z (ora31d1) 0.24 2.80 f
U1702/Z (aor311d1) 0.25 3.04 f
U1704/Z (aor211d1) 0.21 3.26 f
U1394/ZN (nr04d0) 0.19 3.45 r
U1393/Z (or02d2) 0.26 3.71 r
U1709/ZN (nr02d0) 0.20 3.91 f
U759/ZN (nd03d2) 0.27 4.18 r
U695/ZN (nr02d2) 0.41 4.59 f
U1101/Z (mx02d0) 0.30 4.89 r
fifo_mem_reg[31][0]/D (dfcrq1) 0.00 4.89 r
data arrival time 4.89

clock wr_clk (rise edge) 5.00 5.00
clock network delay (ideal) 0.00 5.00
fifo_mem_reg[31][0]/CP (dfcrq1) 0.00 5.00 r
library setup time -0.10 4.90
data required time 4.90
---
data required time 4.90
data arrival time -4.89
---
slack (MET) 0.00

1

5. This is the back-end timing report , Write clock runs at 5ns and Read clock runs at 25ns.

```
Loading db file '/usr/synopsys/E-2010.12-SP4-ICC/libraries/syn/gtech.db'
Information: linking reference library : /usr/synopsys/E-2010.12-SP2/ref/mw_lib/io. (PSYN-878)
Information: linking reference library : /usr/synopsys/E-2010.12-SP2/ref/mw_lib/sc. (PSYN-878)
Information: linking reference library : /usr/synopsys/E-2010.12-SP2/ref/mw_lib/ram16x128. (PSYN-878)

 Linking design 'fifo_model'
 Using the following designs and libraries:
 --------------------------------------------------------------------------
 fifo_model              chip_finish_final.CEL
 cb13fs120_tsmc_max (library) /usr/synopsys/E-2010.12-SP2/ref/db/sc_max.db
 cb13fs120_tsmc_max (library) /usr/synopsys/E-2010.12-SP2/ref/db/sc_pg_max.db
 cb13io320_tsmc_max (library) /usr/synopsys/E-2010.12-SP2/ref/db/io_max.db
 cb13special_max (library)   /usr/synopsys/E-2010.12-SP2/ref/db/special_max.db
 ram16x128_max (library)     /usr/synopsys/E-2010.12-SP2/ref/db/ram16x128_max.db
 ram32x64_max (library)      /usr/synopsys/E-2010.12-SP2/ref/db/ram32x64_max.db

Load global CTS reference options from NID to stack
Information: The design has horizontal rows, and Y-symmetry has been used for sites. (MWDC-217)
Floorplan loading succeeded.

 Loading design 'fifo_model'
```

Information: Library Manufacturing Grid(GridResolution) : 5
Information: Time Unit from Milkyway design library: 'ns'
Information: Design Library and main library timing units are matched - 1.000 ns.
Information: Resistance Unit from Milkyway design library: 'kohm'
Information: Design Library and main library resistance units are matched - 1.000 kohm.
Information: Capacitance Unit from Milkyway design library: 'pf'
Information: Design Library and main library capacitance units are matched - 1.000 pf.
Warning: Inconsistent library data found for layer CP. (RCEX-018)

TLU+ File = /usr/synopsys/E-2010.12-SP2/ref/tlup/cb13_6m_max.tluplus
TLU+ File = /usr/synopsys/E-2010.12-SP2/ref/tlup/cb13_6m_min.tluplus

--------- Sanity Check on TLUPlus Files ----------
1. Checking the conducting layer names in ITF and mapping file ...
[Passed!]
2. Checking the via layer names in ITF and mapping file ...
[Passed!]
3. Checking the consistency of Min Width and Min Spacing between MW-tech and ITF ...
[Passed!]
---------- Check Ends ----------
Information: The distance unit in Capacitance and Resistance is 1 micron. (RCEX-007)
Information: The RC model used is detail route TLU+. (RCEX-015)
Information: Start mixed mode parasitic extraction. (RCEX-023)
Information: Start rc extraction...
Information: Parasitic source is LPE. (RCEX-040)
Information: Parasitic mode is RealRC. (RCEX-041)
Information: Using virtual shield extraction. (RCEX-081)
Information: Extraction mode is MIN_MAX. (RCEX-042)
Information: Extraction derate is 125/125/125. (RCEX-043)
Information: Coupling capacitances are lumped to ground. (RCEX-044)
Information: Start back annotation for parasitic extraction. (RCEX-023)
Information: End back annotation for parasitic extraction. (RCEX-023)
Information: Start timing update for parasitic extraction. (RCEX-023)
Information: End timing update for parasitic extraction. (RCEX-023)
Information: End parasitic extraction. (RCEX-023)
Information: Updating graph... (UID-83)

Information: Updating design information... (UID-85)
Information: Input delay ('rise') on clock port 'wr_clk' will be added to the clock's propagated skew. (TIM-112)
Information: Input delay ('fall') on clock port 'wr_clk' will be added to the clock's propagated skew. (TIM-112)
Information: Input delay ('rise') on clock port 'rd_clk' will be added to the clock's propagated skew. (TIM-112)
Information: Input delay ('fall') on clock port 'rd_clk' will be added to the clock's propagated skew. (TIM-112)

****
Report : timing
-path full
-delay max
-max_paths 1
Design : fifo_model
Version: E-2010.12-ICC-SP4
Date : Sat Apr 14 02:31:06 2012
****

* Some/all delay information is back-annotated.

Operating Conditions: cb13fs120_tsmc_max Library: cb13fs120_tsmc_max

Information: Percent of Arnoldi-based delays = 0.00%

Startpoint: wrclk_rdpointer_reg[4]
(rising edge-triggered flip-flop clocked by wr_clk)
Endpoint: full_flag (output port clocked by rd_clk)
Path Group: OUTPUTS
Path Type: max

Point	Incr	Path
clock wr_clk (rise edge)	20.00	20.00
clock network delay (propagated)	0.38	20.38
wrclk_rdpointer_reg[4]/CP (dfcrq2)	0.00	20.38 r
wrclk_rdpointer_reg[4]/Q (dfcrq2)	0.36	20.74 r
U31/ZN (inv0d2)	0.04 &	20.77 f
U70/ZN (nd02d1)	0.06 &	20.83 r

U72/ZN (nd02d2) 0.10 & 20.93 f
U12/ZN (invbd2) 0.04 & 20.97 r
U79/ZN (nd02d2) 0.06 & 21.03 f
U80/ZN (nd02d2) 0.07 & 21.09 r
U49/ZN (inv0d2) 0.05 & 21.14 f
U96/ZN (nd02d2) 0.04 & 21.19 r
U50/ZN (nd02d2) 0.06 & 21.25 f
U90/ZN (invbd2) 0.04 & 21.28 r
U30/ZN (nd02d2) 0.06 & 21.34 f
U45/ZN (nd02d2) 0.06 & 21.40 r
U60/ZN (nd02d1) 0.06 & 21.47 f
U52/ZN (nd03d0) 0.08 & 21.55 r
U32/CO (cg01d1) 0.21 & 21.75 r
U1407/CO (cg01d1) 0.19 & 21.94 r
U1337/ZN (nd12d1) 0.06 & 22.01 f
U1336/ZN (nd02d1) 0.04 & 22.05 r
U1404/CO (cg01d1) 0.19 & 22.24 r
U1701/ZN (nr02d0) 0.07 & 22.31 f
U1702/Z (aor311d2) 0.24 & 22.55 f
U105/ZN (aoim211d1) 0.20 & 22.75 r
U109/ZN (nd02d2) 0.07 & 22.81 f
U112/ZN (inv0d4) 0.05 & 22.86 r
U38/ZN (invbd7) 0.06 & 22.92 f
U23/ZN (invbdk) 0.31 & 23.23 r
full_flag (out) 0.16 & 23.39 r
data arrival time 23.39

clock rd_clk (rise edge) 25.00 25.00
clock network delay (ideal) 1.00 26.00
output external delay -2.00 24.00
data required time 24.00
---
data required time 24.00
data arrival time -23.39
---
slack (MET) 0.61

Startpoint: fifo_mem_reg[14][10]
(rising edge-triggered flip-flop clocked by wr_clk)

Endpoint: rd_data_reg[10]
(rising edge-triggered flip-flop clocked by rd_clk)
Path Group: rd_clk
Path Type: max

Point	Incr	Path
clock wr_clk (rise edge)	20.00	20.00
clock network delay (propagated)	0.39	20.39
fifo_mem_reg[14][10]/CP (dfcrq1)	0.00	20.39 r
fifo_mem_reg[14][10]/Q (dfcrq1)	0.37	20.76 f
U1641/Z (aor22d1)	0.19 &	20.95 f
U1642/ZN (nr04d0)	0.18 &	21.13 r
U1643/ZN (oai2222d1)	0.42 &	21.55 f
rd_data_reg[10]/U4/Z (mx02d0)	0.16 &	21.71 f
rd_data_reg[10]/D (dfcrb1)	0.00 &	21.71 f
data arrival time		21.71
clock rd_clk (rise edge)	25.00	25.00
clock network delay (propagated)	0.20	25.20
rd_data_reg[10]/CP (dfcrb1)	0.00	25.20 r
library setup time	-0.07	25.14
data required time		25.14
data required time		25.14
data arrival time		-21.71
slack (MET)		3.42

Startpoint: wrclk_rdpointer_reg[4]
(rising edge-triggered flip-flop clocked by wr_clk)
Endpoint: fifo_mem_reg[6][3]
(rising edge-triggered flip-flop clocked by wr_clk)
Path Group: wr_clk
Path Type: max

Point	Incr	Path
clock wr_clk (rise edge)	0.00	0.00

clock network delay (propagated) 0.38 0.38
wrclk_rdpointer_reg[4]/CP (dfcrq2) 0.00 0.38 r
wrclk_rdpointer_reg[4]/Q (dfcrq2) 0.36 0.74 r
U31/ZN (inv0d2) 0.04 & 0.77 f
U70/ZN (nd02d1) 0.06 & 0.83 r
U72/ZN (nd02d2) 0.10 & 0.93 f
U12/ZN (invbd2) 0.04 & 0.97 r
U79/ZN (nd02d2) 0.06 & 1.03 f
U80/ZN (nd02d2) 0.07 & 1.09 r
U49/ZN (inv0d2) 0.05 & 1.14 f
U96/ZN (nd02d2) 0.04 & 1.19 r
U50/ZN (nd02d2) 0.06 & 1.25 f
U90/ZN (invbd2) 0.04 & 1.28 r
U30/ZN (nd02d2) 0.06 & 1.34 f
U45/ZN (nd02d2) 0.06 & 1.40 r
U60/ZN (nd02d1) 0.06 & 1.47 f
U52/ZN (nd03d0) 0.08 & 1.55 r
U32/CO (cg01d1) 0.21 & 1.75 r
U1407/CO (cg01d1) 0.19 & 1.94 r
U1337/ZN (nd12d1) 0.06 & 2.01 f
U1336/ZN (nd02d1) 0.04 & 2.05 r
U1404/CO (cg01d1) 0.19 & 2.24 r
U1701/ZN (nr02d0) 0.07 & 2.31 f
U1702/Z (aor311d2) 0.24 & 2.55 f
U111/Z (aor211d1) 0.20 & 2.75 f
U1394/ZN (nr04d0) 0.09 & 2.85 r
U1393/Z (or02d2) 0.24 & 3.08 r
U1709/ZN (nr02d0) 0.15 & 3.24 f
U793/ZN (nd03d0) 0.29 & 3.53 r
U783/ZN (nr02d0) 0.70 & 4.23 f
U1194/Z (mx02d0) 0.36 & 4.59 r
fifo_mem_reg[6][3]/D (dfcrq1) 0.00 & 4.59 r
data arrival time 4.59

clock wr_clk (rise edge) 5.00 5.00
clock network delay (propagated) 0.36 5.36
fifo_mem_reg[6][3]/CP (dfcrq1) 0.00 5.36 r
library setup time -0.06 5.29
data required time 5.29
---

data required time	5.29
data arrival time	-4.59
---	---
slack (MET)	0.70

1

6. Area Report for Front-end design.

```
****************************************
Report : area
Design : fifo_model
Version: E-2010.12-SP4
Date   : Fri Mar 16 18:34:11 2012
****************************************

Library(s) Used:

    cb13fs120_tsmc_max (File: /usr/synopsys/E-2010.12-SP2/ref/db/sc_max.db)

Number of ports:                 40
Number of nets:                1635
Number of cells:               1603
Number of combinational cells:         1035
Number of sequential cells:             568
Number of macros:                         0
Number of buf/inv:                       28
Number of references:                    36

Combinational area:       2115.750000
Noncombinational area:    3151.000000
Net Interconnect area:    1067.714145

Total cell area:          5266.750000
Total area:               6334.464145
1
```

7. This is Area Report for Back-end design.

```
****************************************
Report : area
Design : fifo_model
Version: E-2010.12-ICC-SP4
Date   : Sat Apr 14 02:37:12 2012
****************************************

Library(s) Used:

    cb13fs120_tsmc_max (File: /usr/synopsys/E-2010.12-SP2/ref/db/sc_max.db)

Number of ports:                 40
Number of nets:                1712
Number of cells:               1680
Number of combinational cells:         1112
Number of sequential cells:             568
Number of macros:                         0
Number of buf/inv:                       68
Number of references:                    41

Combinational area:       2216.250000
Noncombinational area:    3151.000000
Net Interconnect area:    1077.476117

Total cell area:          5367.250000
Total area:               6444.726117
1
```

8. This is the Fault Summary Report (ATPG fault coverage report) for Front-end; Showing the Test Coverage of 100%.

```
In mode: Internal_scan...
  Design has scan chains in this mode
  Design is scan routed
  Post-DFT DRC enabled

Information: Starting test design rule checking. (TEST-222)
  Loading test protocol
  ...basic checks...
  ...basic sequential cell checks...
  ...checking vector rules...
  ...checking clock rules...
  ...checking scan chain rules...
  ...checking scan compression rules...
  ...checking X-state rules...
  ...checking tristate rules...
  ...extracting scan details...

-------------------------------------------------------------------
  DRC Report

  Total violations: 0

-------------------------------------------------------------------


Test Design rule checking did not find violations

-------------------------------------------------------------------
  Sequential Cell Report

  32 out of 592 sequential cells have violations

-------------------------------------------------------------------

SEQUENTIAL CELLS WITH VIOLATIONS
      *  32 cells are clock gating cells
SEQUENTIAL CELLS WITHOUT VIOLATIONS
      * 548 cells are valid scan cells
      *  12 cells are non-scan shift-register cells

Information: Test design rule checking completed. (TEST-123)
  Running test coverage estimation...
 14224 faults were added to fault list.
 ATPG performed for stuck fault model using internal pattern source.
 ------------------------------------------------------------
 #patterns     #faults     #ATPG faults  test      process
 stored     detect/active  red/au/abort  coverage  CPU time
 ---------  -------------  ------------  --------  --------
 Begin deterministic ATPG: #uncollapsed_faults=10125, abort_limit=10...
 0             6331   3794         0/0/0    73.33%      0.02
 0             1617   2177         0/0/0    84.69%      0.02
 0              770   1407         0/0/0    90.11%      0.03
 0              648    758         1/0/0    94.67%      0.03
 0              266    492         1/0/0    96.54%      0.04
 0              128    364         1/0/0    97.44%      0.04
 0               83    281         1/0/0    98.02%      0.05
 0              106    174         2/0/0    98.78%      0.05
 0               63    111         2/0/0    99.22%      0.05
 0              103      8         2/0/0    99.94%      0.06
 0                8      0         2/0/0   100.00%      0.06
```

```
           Pattern Summary Report
-----------------------------------------------
#internal patterns                             0
-----------------------------------------------


    Uncollapsed Stuck Fault Summary Report
-----------------------------------------------
fault class                    code   #faults
------------------------------ ----  ---------
Detected                        DT      14222
Possibly detected               PT          0
Undetectable                    UD          2
ATPG untestable                 AU          0
Not detected                    ND          0
-----------------------------------------------
total faults                              14224
test coverage                            100.00%
-----------------------------------------------
 Information: The test coverage above may be inferior
              than the real test coverage with customized
              protocol and test simulation library.
```

9. Power report for Front-end from Design compiler:

```
Loading db file '/usr/synopsys/E-2010.12-SP2/ref/db/sc_max.db'
Information: Propagating switching activity (low effort zero delay simulation). (PWR-6)
Warning: Design has unannotated primary inputs. (PWR-414)
Warning: Design has unannotated sequential cell outputs. (PWR-415)

****************************************
Report : power
        -analysis_effort low
Design : fifo_model
Version: E-2010.12-SP4
Date   : Sun Apr 15 21:43:09 2012
****************************************


Library(s) Used:

    cb13fs120_tsmc_max (File: /usr/synopsys/E-2010.12-SP2/ref/db/sc_max.db)


Operating Conditions: cb13fs120_tsmc_max   Library: cb13fs120_tsmc_max
Wire Load Model Mode: enclosed

Design        Wire Load Model           Library
------------------------------------------------
fifo_model            8000            cb13fs120_tsmc_max


Global Operating Voltage = 1.08
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =  36.8412 uW   (56%)
  Net Switching Power  =  28.4456 uW   (44%)
```

Total Dynamic Power = 65.2868 uW (100%)

Cell Leakage Power = 29.4684 uW

1

10. Power report for Backend Design from IC Compiler:

```
****************************************
Report : power
        -analysis_effort low
Design : fifo_model
Version: E-2010.12-ICC-SP4
Date   : Sat Apr 14 02:49:17 2012
****************************************


Library(s) Used:

    cb13fs120_tsmc_max (File: /usr/synopsys/E-2010.12-SP2/ref/db/sc_max.db)


Operating Conditions: cb13fs120_tsmc_max   Library: cb13fs120_tsmc_max
Wire Load Model Mode: enclosed

Design       Wire Load Model          Library
------------------------------------------------
fifo_model           8000           cb13fs120_tsmc_max


Global Operating Voltage = 1.08
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  = 358.4537 uW   (42%)
  Net Switching Power  = 495.5928 uW   (58%)
```

Total Dynamic Power = 854.0466 uW (100%)

Cell Leakage Power = 30.7003 uW

1

11. This is the Back-end Quality Of Results (QoR) report after performing routing.

```
****************************************
Report : qor
Design : fifo_model
Version: E-2010.12-ICC-SP4
Date   : Sat Apr 14 02:55:43 2012
****************************************


  Timing Path Group 'OUTPUTS'
  -----------------------------------
  Levels of Logic:             26.00
  Critical Path Length:         3.03
  Critical Path Slack:          0.61
  Critical Path Clk Period:     25.00
  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 'rd_clk'
  -----------------------------------
  Levels of Logic:             4.00
  Critical Path Length:         1.34
  Critical Path Slack:          3.42
  Critical Path Clk Period:     25.00
  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 'wr_clk'
  -----------------------------------
  Levels of Logic:             28.00
  Critical Path Length:         4.23
```

Critical Path Slack: 0.70
Critical Path Clk Period: 5.00
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

Cell Count

Hierarchical Cell Count: 0
Hierarchical Port Count: 0
Leaf Cell Count: 1680
Buf/Inv Cell Count: 68
CT Buf/Inv Cell Count: 10
Combinational Cell Count: 1112
Sequential Cell Count: 568
Macro Count: 0

Area

Combinational Area: 2216.250000
Noncombinational Area: 3151.000000
Net Area: 1077.476117
Net XLength : 53642.67
Net YLength : 53238.28

Cell Area: 5367.250000
Design Area: 6444.726117
Net Length : 106880.95

Design Rules

Total Number of Nets: 1712
Nets With Violations: 0

```
-----------------------------------


Hostname: dcd157.ecs.csun.edu

Compile CPU Statistics
-----------------------------------------
Resource Sharing:                0.00
Logic Optimization:              0.00
Mapping Optimization:           10.26
-----------------------------------------
Overall Compile Time:           10.46
Overall Compile Wall Clock Time:    10.53
```

1

12. This is the Back-end LVS report after fixing Short and Open Nets.

```
-- LVS START : --
WARNING : The boundary ((-0.060,0.000),(217.520,216.640)) of metal Layer3 is out of top
cell boundary ((0.000,0.000),(217.460,216.640)).
Total area error in layer 0 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 1 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 2 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
ERROR : area  [(0.000,49.385), (0.430,49.585)]  0.0860 um sqr.
ERROR : area  [(217.115,72.755), (217.460,72.955)]      0.0690 um sqr.
ERROR : area  [(0.000,143.685), (0.360,143.885)]        0.0720 um sqr.
ERROR : area  [(217.030,167.055), (217.460,167.255)]    0.0860 um sqr.
Total area error in layer 3 is 4.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 4 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 5 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 6 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 7 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 8 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 9 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 10 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 11 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 12 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 13 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 14 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
Total area error in layer 15 is 0.  Elapsed =    0:00:00, CPU =    0:00:00
** Total Floating ports are 0.
** Total Floating Nets are 0.
** Total SHORT Nets are 0.
** Total OPEN Nets are 0.
** Total Electrical Equivalent Error are 0.
** Total Must Joint Error are 0.

-- LVS END : --
Elapsed =    0:00:00, CPU =    0:00:00
Update error cell ...
```

13. Layout showing Standard Cells and IOs Placed on top of each other at the bottom left corner.

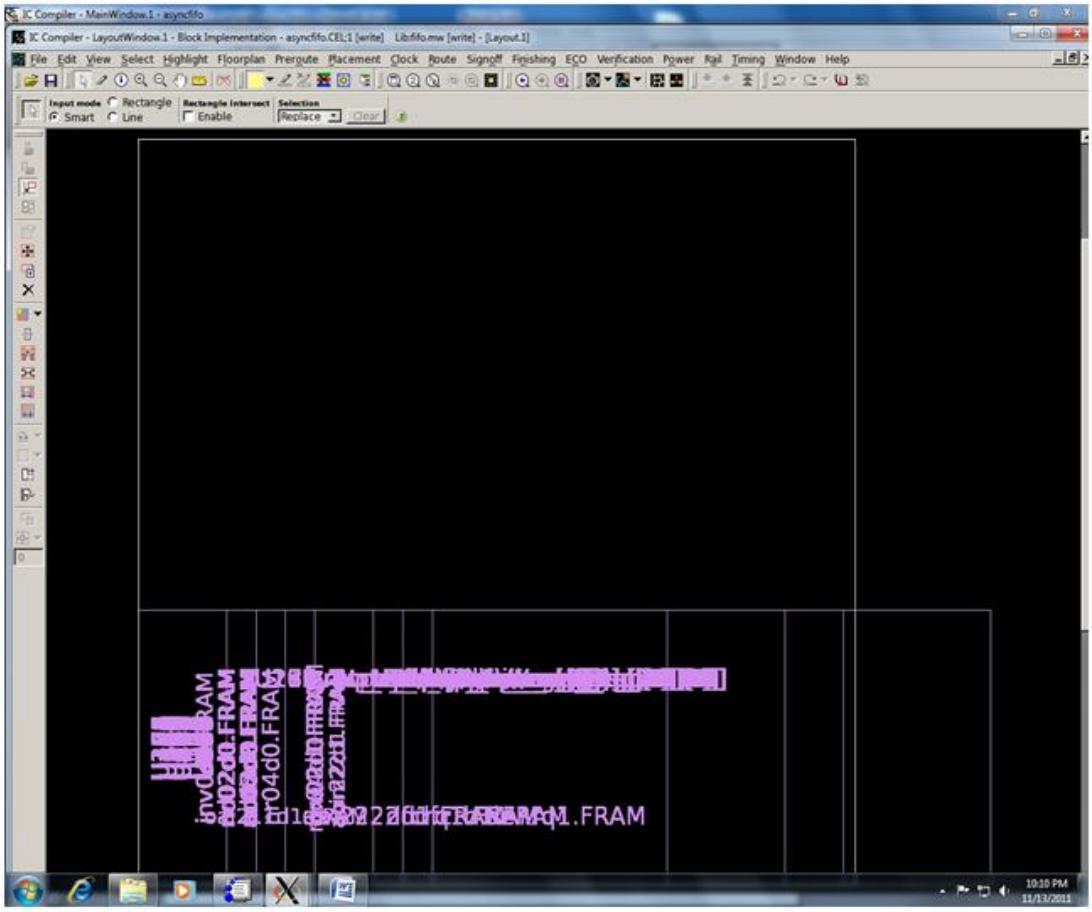

Figure A.7: Layout showing Standard Cells and IOs Placed on top of each other

14. Layout clearly showing Standard Cell palced outside the Core Area (INITIAL FLOORPLAN)

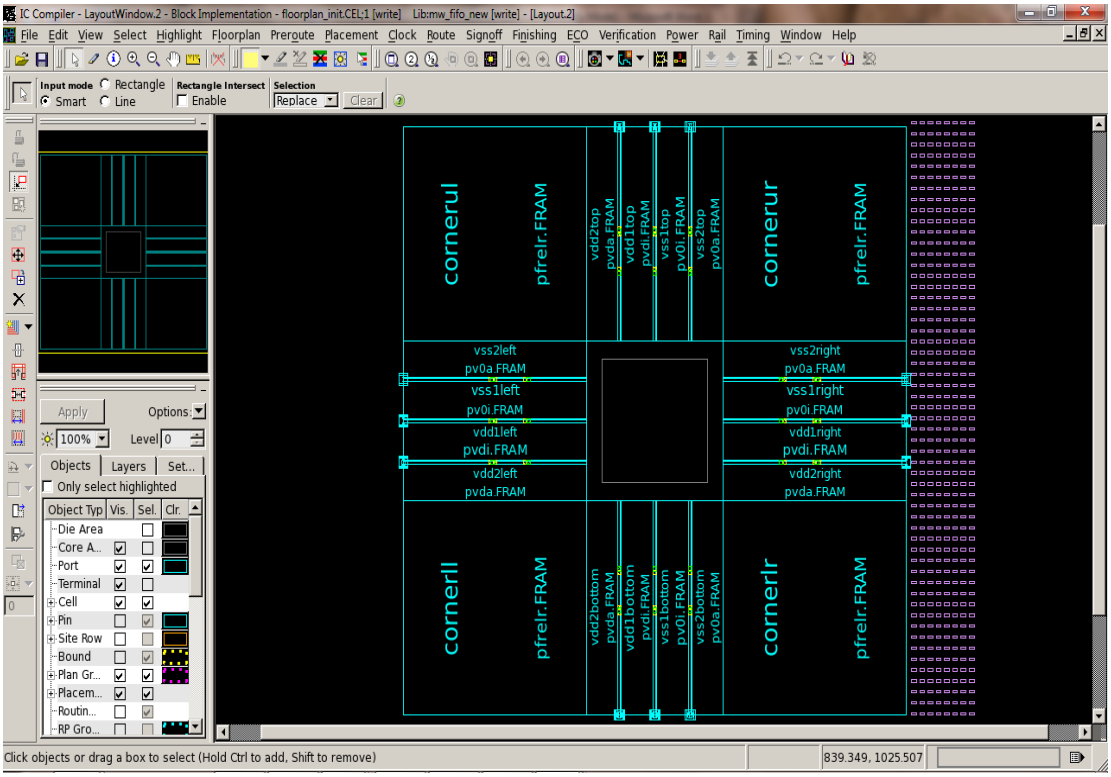

Figure A.8: Initial Floorplan

15. Layout showing Standard Cell Placed Inside the Core Area.

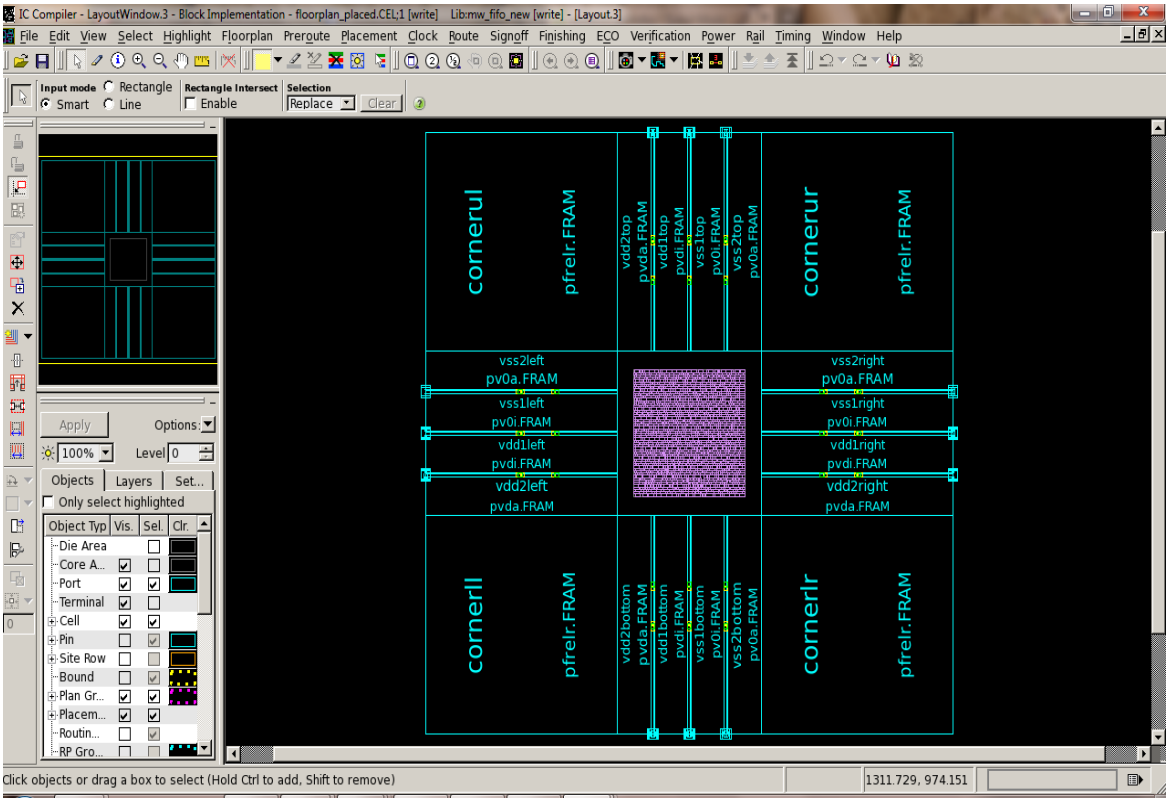

Figure A.9: Placement

16. Layout showing Power Network Synthesis (Power Rings and Power Straps).

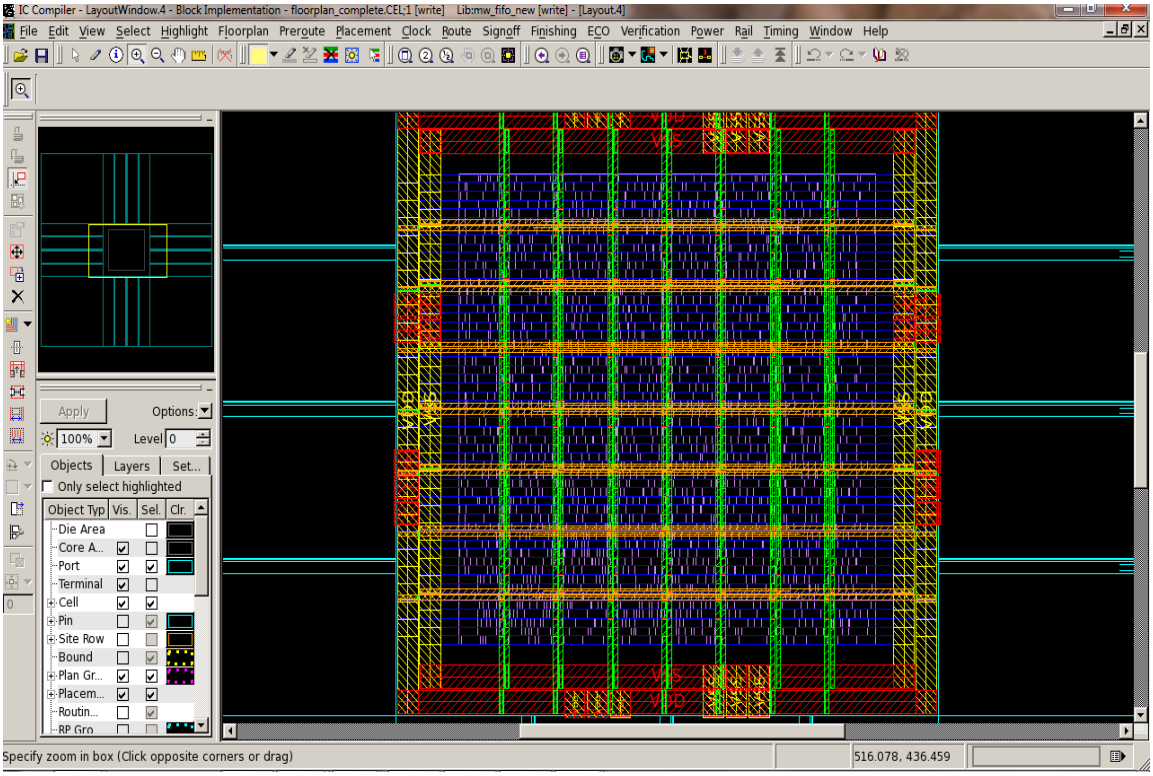

Figure A.10: Layout showing Power Network Synthesis.

17. Layout Showing Clock Tree Synthesis.

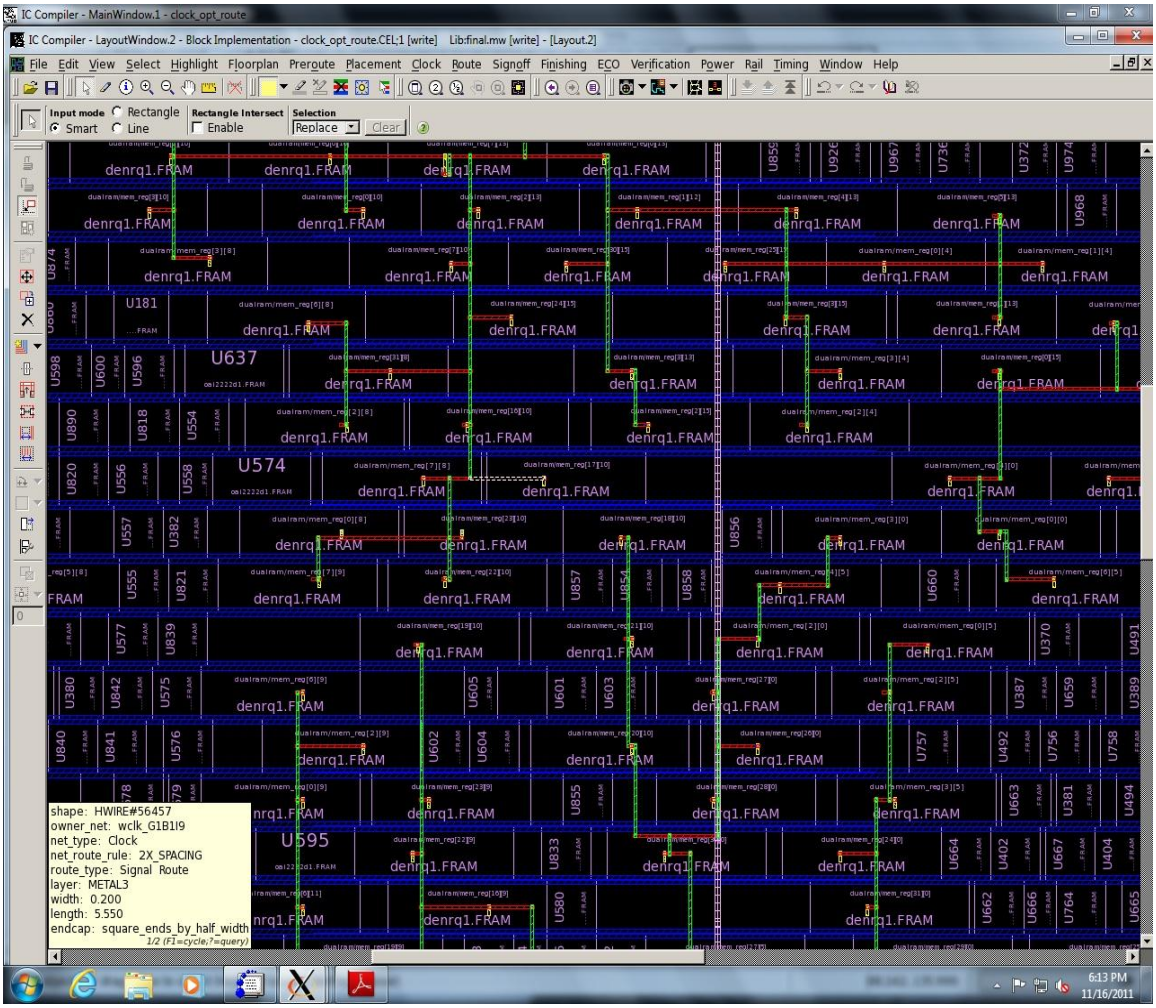

Figure A.11: Layout Showing CTS

18. Layout showing Routing.

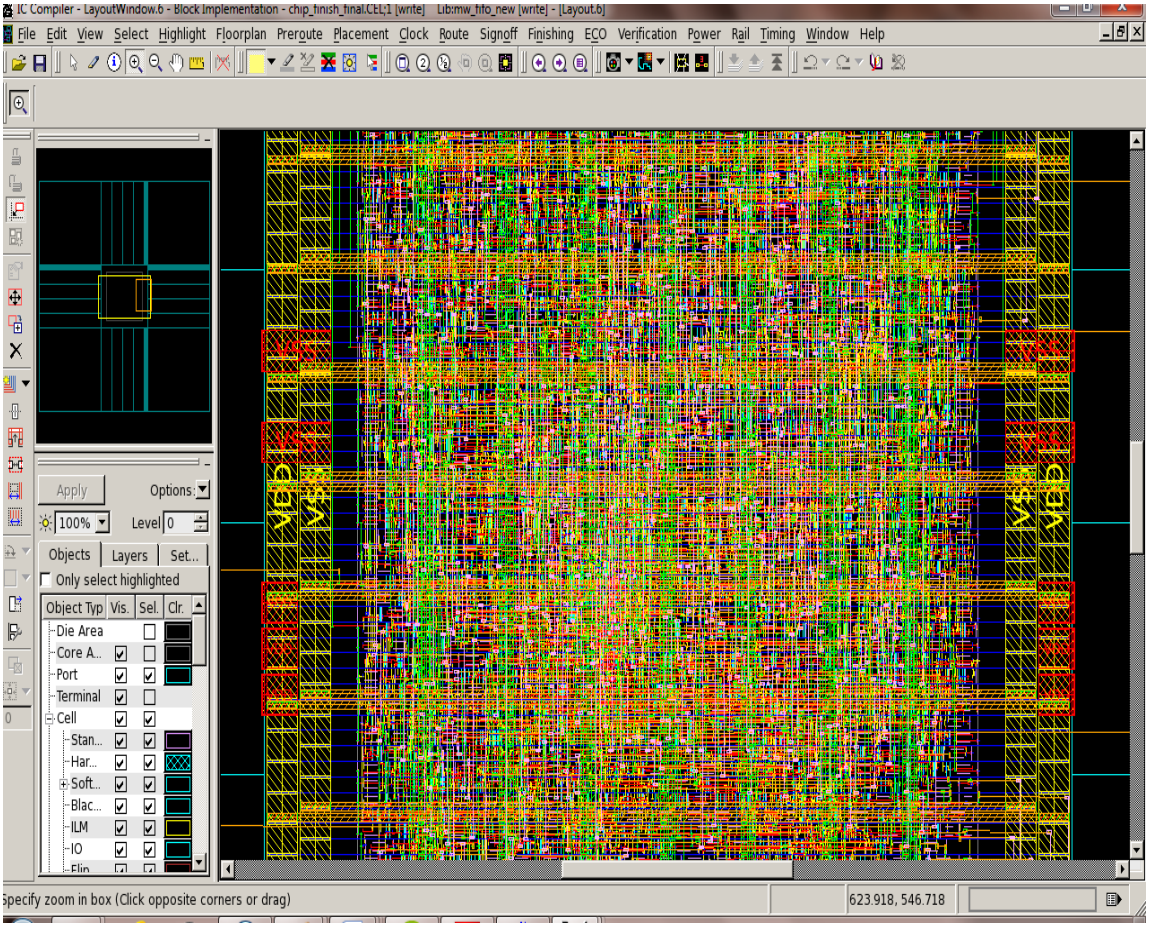

Figure A.12: Routing.

19. Final Layout after chip finishing steps:

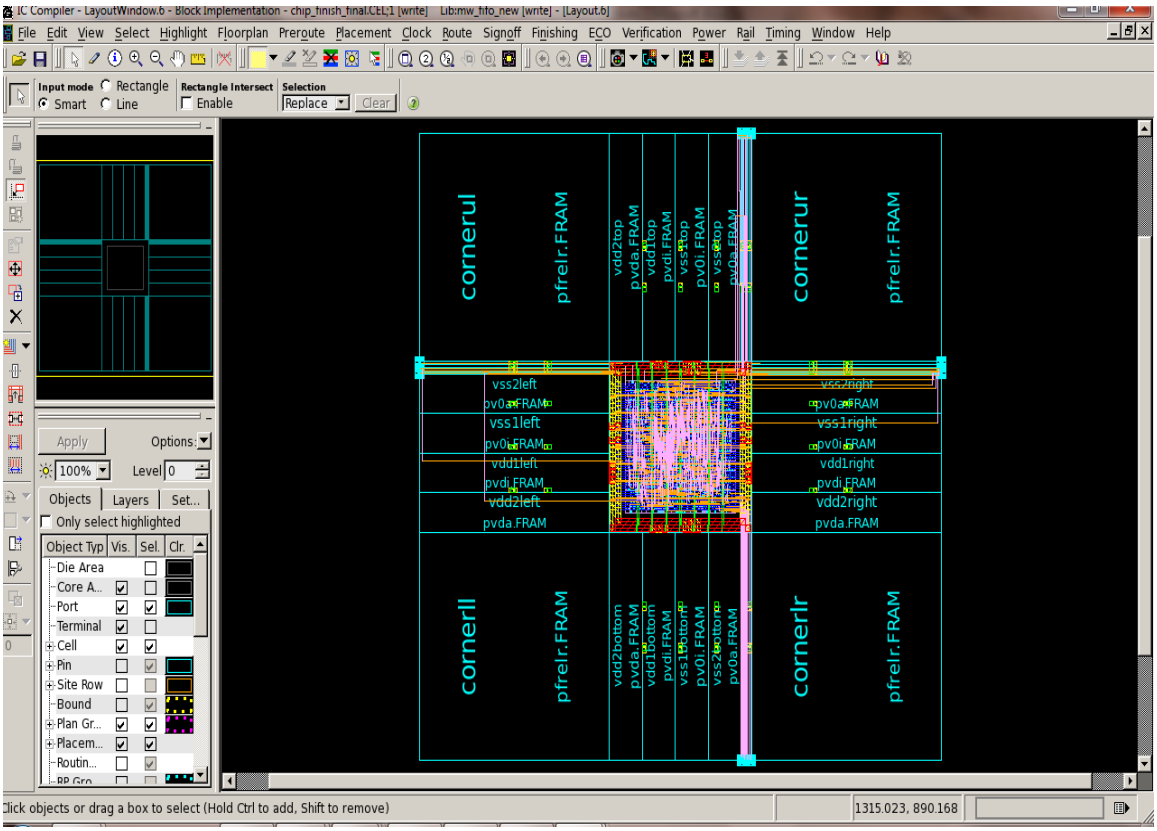

Figure A.13: Final Layout of the chip