CALIFORNIA STATE UNIVERSITY NORTHRIDGE

Asynchronous Interface, Implementation of Complete ASIC Design Flow

A graduate project submitted in partial fulfillment of the requirements

For the degree of Masters of Science

In Electrical Engineering

By

Raviteja Podila

DECEMBER 2013

The graduate project of Raviteja Podila is approved:

Professor Vijay Bhatt	Date
Dr. Ali Amini	Date
Dr. Ramin Roosta, Chair	Date

California State University, Northridge

Acknowledgement

I would like to express my gratitude and appreciation to all those who gave me the possibility to complete this report. A special thanks to Dr. Ramin Roosta, whose help, stimulating suggestions and encouragement helped me to finish this project.

I would also like to acknowledge with much appreciation the crucial role of the staff of EE department and Dr. Ramin Roosta, who gave the twenty four hours access to use all required labs and the necessary material to complete the Synthesis part of my project.

I am also highly thankful to other faculty member in committee Dr. Ali Amini for his guidance both on this project and in my general academic pursuits and Professor Vijay Bhatt for his helpful suggestions and support.

Table Of Contents

List Of Figures

ABSTRACT

Asynchronous Interface, Implementation of Complete ASIC Design Flow

By

Raviteja Podila

Master of Science in Electrical Engineering

The aim of this project is to successfully complete ASIC design flow using the advanced industry level tools. This project provides a solid base and practical hands-on experience of these advanced tools. It also provides an overview of types of ASICs, detailed ASIC standard design flow (Front and Back end), Synopsys Design Compiler and IC compiler flow. 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.

The project describes the design and Implementation of an Asynchronous FIFO. The Asynchronous FIFO comparison method requires additional techniques to correctly synthesize and analyze the design, which are detailed in this project. In this project whole ASIC flow explained with the help of new advanced EDA tools like Synopsys Design Compiler, IC Compiler.

Chapter 1: Introduction of ASICs

Application Specific Integrated Circuit (ASIC) is designed for a specific function and is widely used in military, space, medical and commercial fields. They offer the best design features in terms of size, power, speed, and reliability ASIC applications are endless due to their customizability in size, power, speed, and reliability. Some commercial applications of these devices can be found in auto emission control, gaming device, iPod, and much more. Mostly all ICs in today's technology are developed using ASIC methodology.

ASICs are categorized into full-custom ASICs and semi-custom ASICs.

1.1 Different types of ASICs:

i. Full Custom ASICs.
ii. Gate Array ASICs.
iii. Standard Cell ASICs.

Full Custom ASICs: Full-custom ASICs are entirely tailor-made to a particular application from the very start. Since their ultimate design and functionality are pre-specified by the user, they are manufactured with all the photolithographic layers of the device already fully defined, similar to most off-the-shelf general purpose ICs. These designs are referred to "handcrafted" designs. The use of predefined masks for manufacturing leaves no option for circuit modification during fabrication, except perhaps for some minor fine-tuning or calibration. This means that a full-custom ASIC

cannot be modified to suit different applications, and is generally produced as a single, specific product for a particular application only.

The disadvantages of a full custom ASICs are:

1. Although the performance of a full custom ASIC remains unmatched, a gate array can often provide sufficient performance.
2. Tradeoff is a 2x reduction in performance for a 10x reduction in cost.
3. Full custom ASICs are complex, making them difficult to build.
4. Require many man hours to design and produce.
5. Requires a high volume of sales to offset the non-recurring engineering (NRE) cost of producing ASIC.

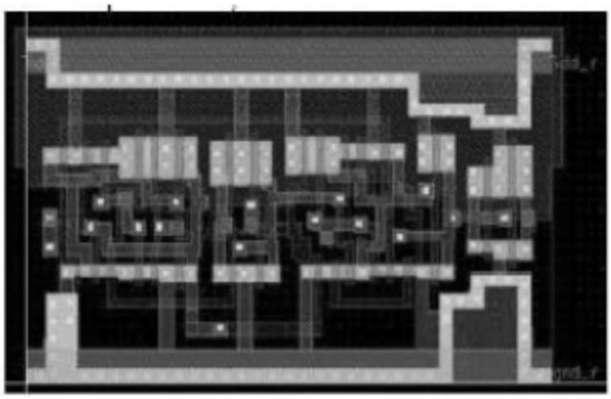

Figure 1.1: Full Custom ASIC

Gate Array ASICs:

Gate array ASICs are partially fabricated with rows of transistors and resistors built in but unconnected. The chip is completed by designing and adhering the top metal layers that provide the interconnecting pathways.

The Gate array is made of "basic cells", where each cell has a standard sized die containing some number of transistors and resistors depending on the vendor. Using a cell library (gates, registers, etc…) and a macro library (more complex functions), the customer designs the chip, and the vendor's software generates the interconnection masks.

These final masking stages are less costly than designing a full custom chip from scratch.

A Gate array circuit is a prefabricated circuit with no particular function, in which transistors, standard logic gates, and other active devices are placed at regular predefined positions and manufactured on a wafer, usually called Master Slice.

Creation of a circuit with a specified function is accomplished by adding metal interconnects to the chips on the master slice late in the manufacturing process, allowing the function of the chip to be customized as desired.

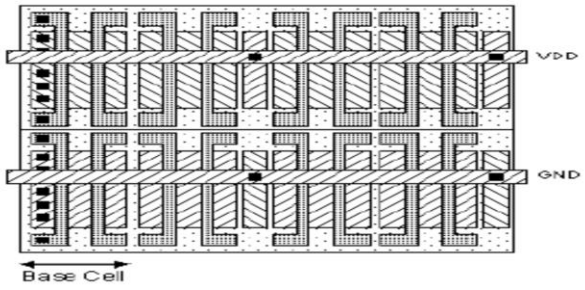

Figure 1.2: Gate Array Basic cell

Structured Gate Array ASICs:

Structured Gate Arrays combine some of the features of standard cell-based ASICs and MGAs. These are designed and produced from a tightly defined set of: 1) design methodologies; 2) intellectual properties (IPs); 3) well-characterized silicon, aimed at shortening the design cycle and minimizing the development costs of the ASIC. A platform ASIC is built from a group of "platform slices", a pre-manufactured device, system, or logic for that platform. Each slice used by the ASIC may be customized by varying its metal layers. The "re-use" of pre-manufactured and pre-characterized platform slices simply means that platform ASICs are not built from scratch, thereby minimizing design cycle time and costs.

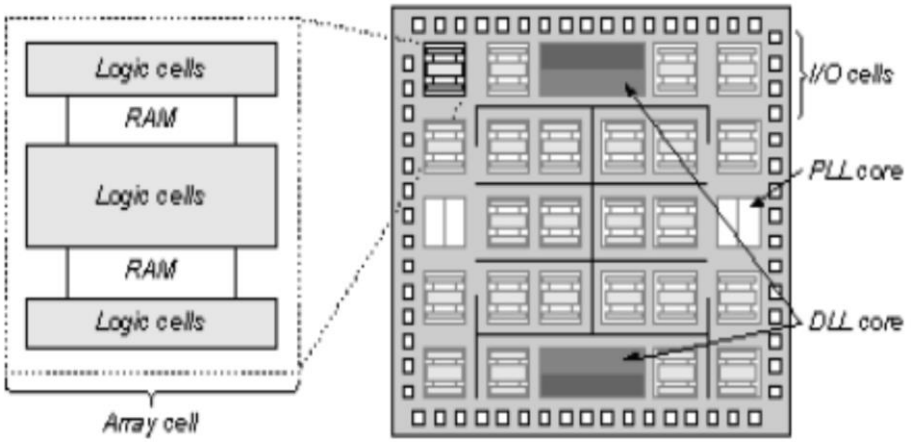

Figure 1.3: Structured Gate Array cells

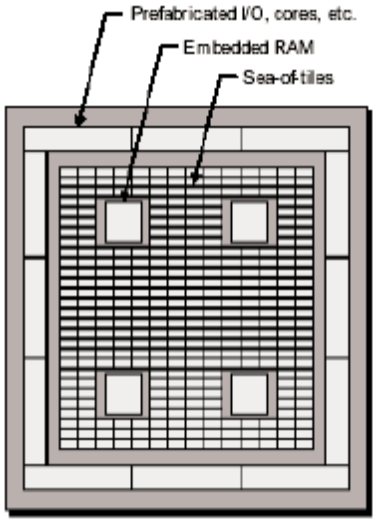

Figure 1.4: Structured Array

Standard Cell ASICs:

Cell-based ASICs (CBICs) make use of pre-designed, pre-tested, and pre-characterized logic cells such as AND gates, OR gates, multiplexers, and flip-flops for the design. These predefined cells are also referred to as standard cells. The standard-cell areas (also called flexible blocks) in a CBIC are built of rows of standard cells similar to a wall built of bricks. The standard-cell areas may be used in combination with larger pre-designed cells, known as megacells. Megacells are also known as mega-functions, full-custom blocks, system-level macros (SLMs), fixed blocks, cores, or Functional Standard Blocks (FSBs). Examples of megacells are microcontrollers and microprocessors. The ASIC designer needs to define only the placement of the standard cells and interconnect in a CBIC. The CBIC approach for ASIC fabrication saves time and money, reduces risk, and allows each standard cell to be optimized individually. For example, during the cell-library design, the designer can choose each transistor in a standard cell to maximize

speed or minimize area. The drawbacks of this approach are the time and expense of designing or buying the standard-cell library and the time needed to fabricate all layers of the ASIC for each new design.

1.2 ASIC Methodology:

To design a chip, one needs to have an idea about what exactly one wants to design. At every step in the ASIC flow, the idea conceived keeps changing forms. The first step to make the idea into a chip is to come up with the specifications or goals to be achieved:

- Goals and constraints of the design.
- Functionality i.e. what the chip will do.
- Performance figures like speed and power.
- Technology constraints like size and space (physical dimensions).
- Fabrication technology and design techniques.

The next step in the flow is to come up with the Structural and Functional Description. At this point, one has to decide what kind of architecture will be used for the design, e.g. RISC/CISC, ALU, pipelining, etc.

To facilitate the design of a complex system, it is normally broken down into several subsystems. The functionality of these subsystems should match the specifications. At this point, the relationship between different sub systems and the top level system is also defined.

The subsystems, as well as the top level system (once defined), need to be implemented.

It is implemented using logic representation (Boolean Expressions), finite state machines,

Combinational, sequential logic, schematics, etc. This phase is called logic design.

Once the previous goals and specifications are elucidated, one can initiate the practical implementation of the design.

Broadly, ASIC design flow can be divided into the following sections:

- RTL Description.
- Functional Simulation/Verification.
- Synthesis.
- Design Verification.
- Layout.

1.3 HDL Design Flow:

The HDL design flow is shown in the figure below. HDL simulation is performed using Cadence tools and TCL scripts. Synthesis is performed using Synopsys IC Compiler, and this tool allows design optimization in terms of time, area, and power.

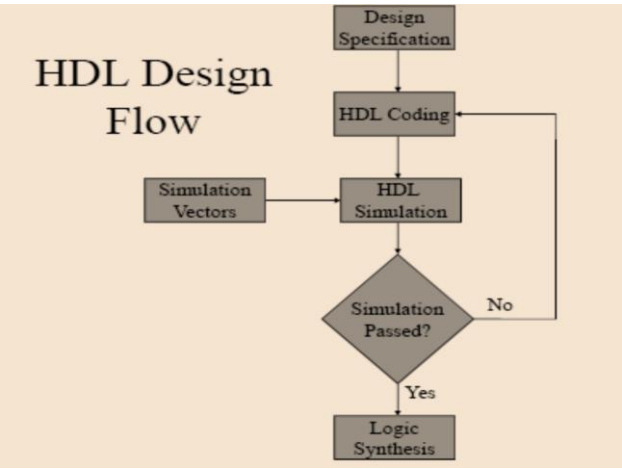

Figure 1.5: HDL Design flow

1.4 Logic Synthesis:

Logic synthesis is the process of converting a high-level description of design into an optimized gate-level representation. Logic synthesis uses a standard cell library, which contains simple cells, such as basic logic gates (i.e. and/or/nor) or macro cells (i.e. adders, multiplexers, memory, and flip-flops). Standard cells put together are referred to as a technology library. Normally, the technology library is known by the transistor size.

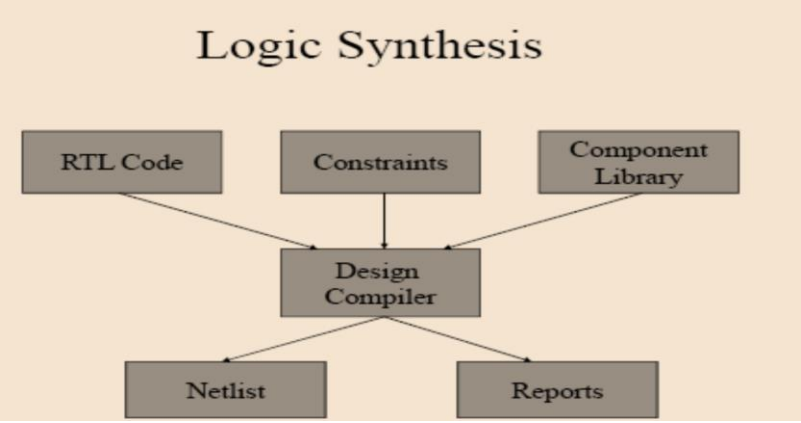

Figure 1.6: Logic Synthesis

ASIC design 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 linking 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).

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.

xi. Initialize the floor planning 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.

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.

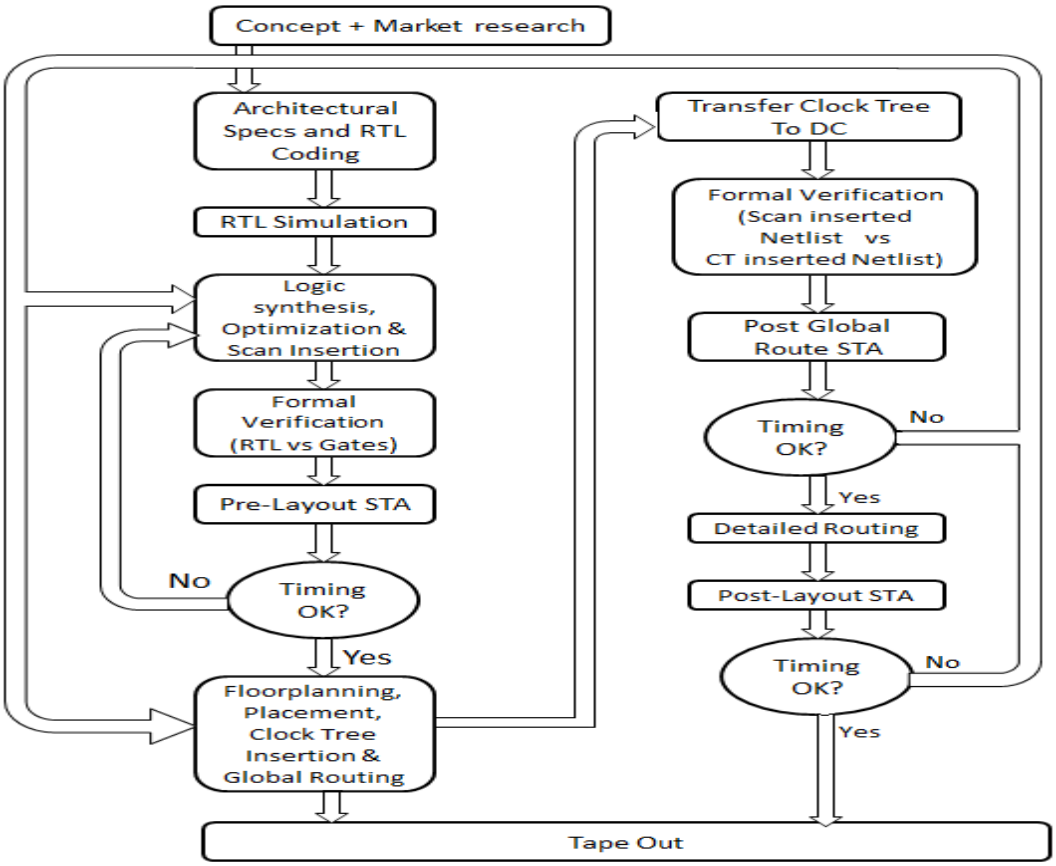

Figure 1.7: Design flow of ASIC

Figure 1.7 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).

Chapter 2: Asynchronous Interface

Asynchronous interface design is the circuitry in which set of signals that consists 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.

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.

2.1 FIFO DISCUSSION:

FIFO means First in First Out. It is the most widely preferred method of data transfer between modules. Data are queued up: first word written is first word read, but it may stay in the queue for a while.

FIFOs are one of the famous examples of Asynchronous Interface which are often used to safely pass data from one clock domain to another asynchronous clock domain. Using a FIFO to pass data from one clock domain to another clock domain requires multi-asynchronous clock design techniques. There are many ways to design a FIFO wrong.

There are many ways to design a FIFO right but still make it difficult to properly synthesize and analyze the design.

An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer 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.

There are many ways to design asynchronous FIFO design, including many wrong ways. Most incorrectly implemented FIFO designs still function properly 90% of the time. Most almost-correct FIFO designs functions properly 99%+ of the time.

2.2 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.

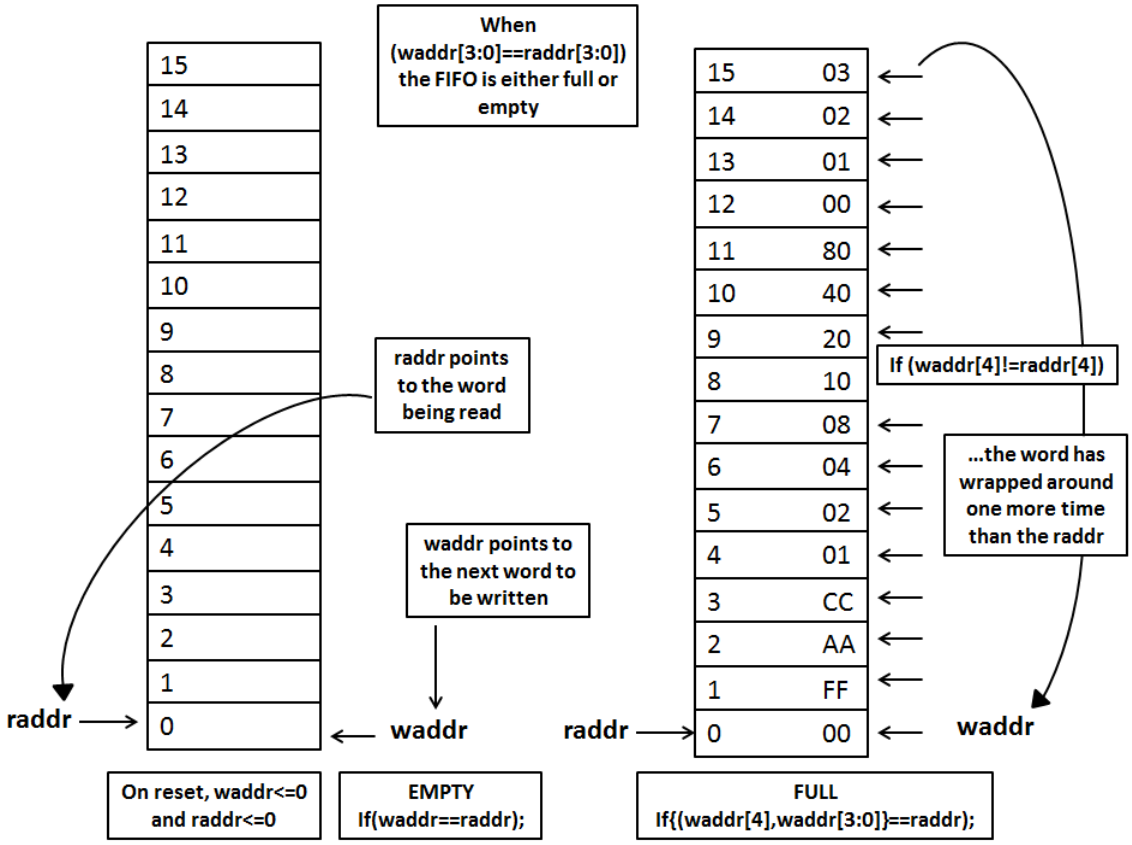

Figure 2.1: FIFO Pointers

Avoid Overflow/Underflow:

FIFO full occurs when the write pointer catches up to the synchronized and sampled read pointer. The write pointer to be compared with the read pointer has a different clock frequency than the read pointer; therefore write pointer should cross this clock domain using synchronizer that has the frequency of the read clock, so they can be compared on the same frequency.

The synchronized and sampled read pointer might not reflect the current value of the actual read pointer but the write pointer will not try to count beyond the synchronized read pointer value.

2.3 FIFO Flags:

Need Full/Empty flags. It is the most complicated part of FIFO. Need to avoid underflow/overflow. If the system was synchronous, it would be relatively easy. If the system was synchronous, we most likely wouldn't need the FIFO at all. Read pointer is formed from read clock; write pointer is formed from write clock.

One design technique used to distinguish between full and empty is to add an extra bit to each pointer. When the write pointer increments after 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 Fig. 2.1(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.

The FIFO to be designed is as shown in the block diagram:

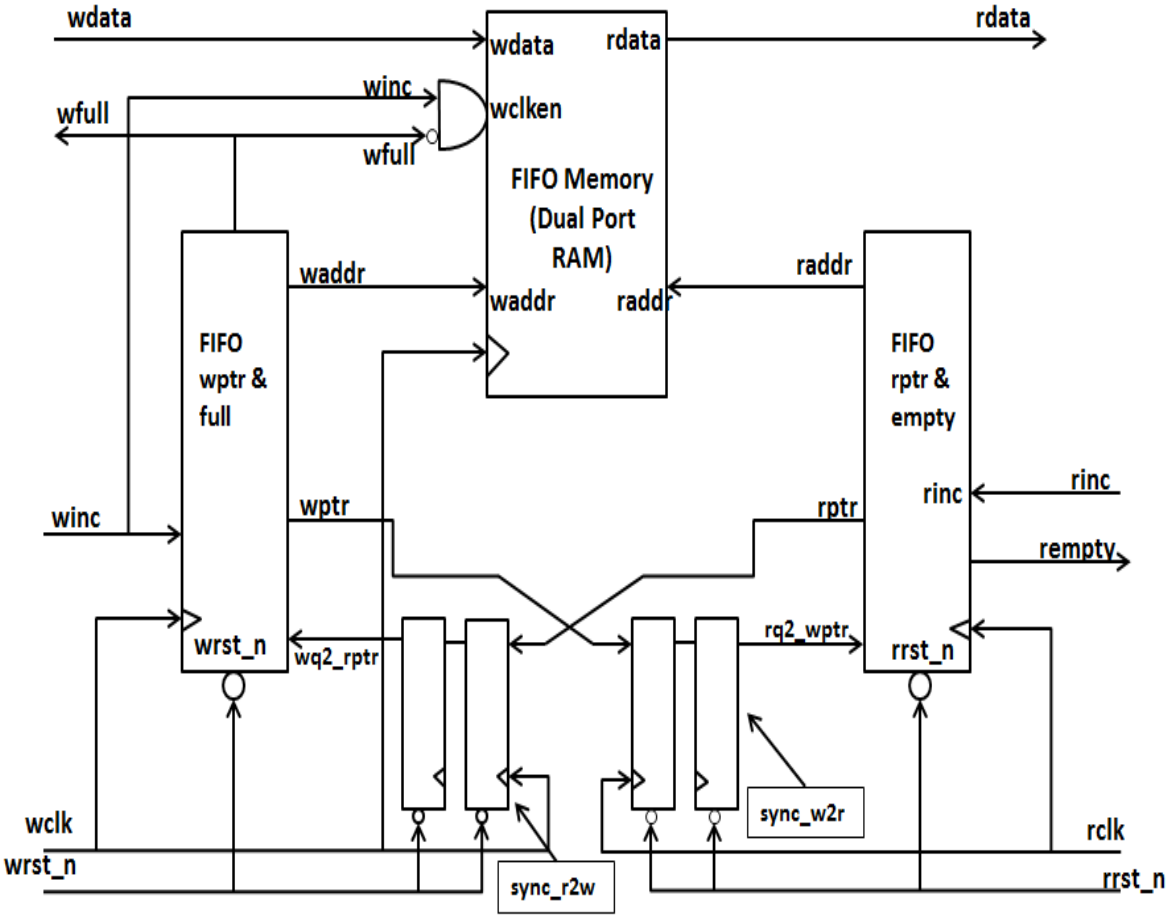

Figure 2.2: FIFO block diagram

Chapter 3: Design Compiler

Design compiler tool is the one of the Synopsys product. It contains tool that synthesize HDL designs in to a gate level netlist.

The main steps in Design Compiler flow are:

i) Load designs and libraries.

ii) Apply design Constraints.

iii) Synthesize the design.

iv) Write out design data.

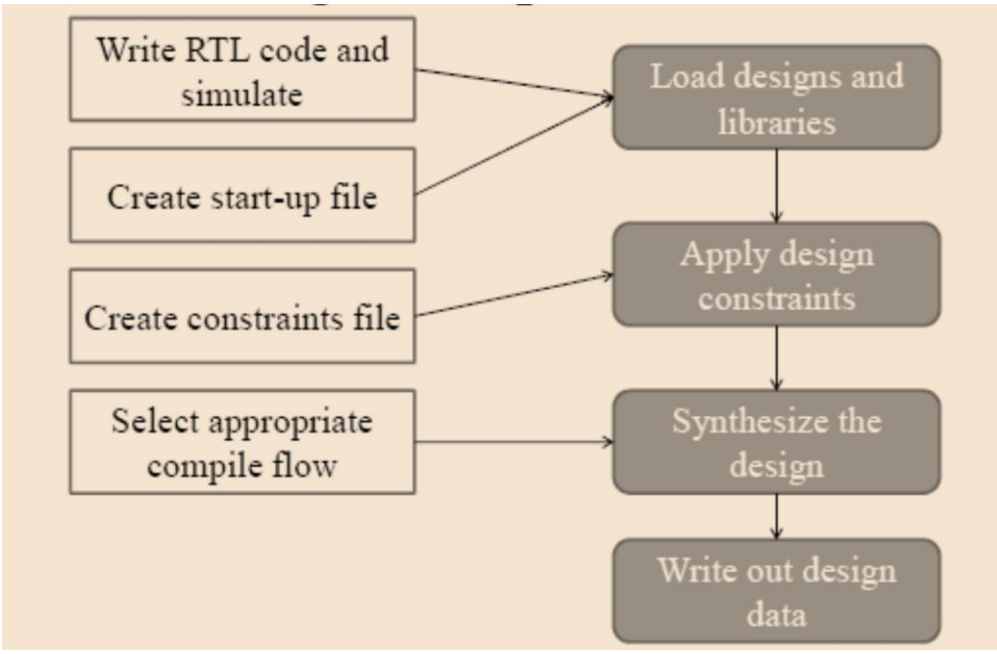

Figure 3.1: Design Compiler Flow

3.1 Synthesis:

Depending on the size and complexity of the RTL, logic synthesis may be a quick procedure on a single file or an iterative process with many files. Synthesis is multi-level logic optimization and technology mapping (Picking the right components).

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.

3.2 Synopsys Design Compiler Flow:

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 gate level-netlist. The gate level-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.

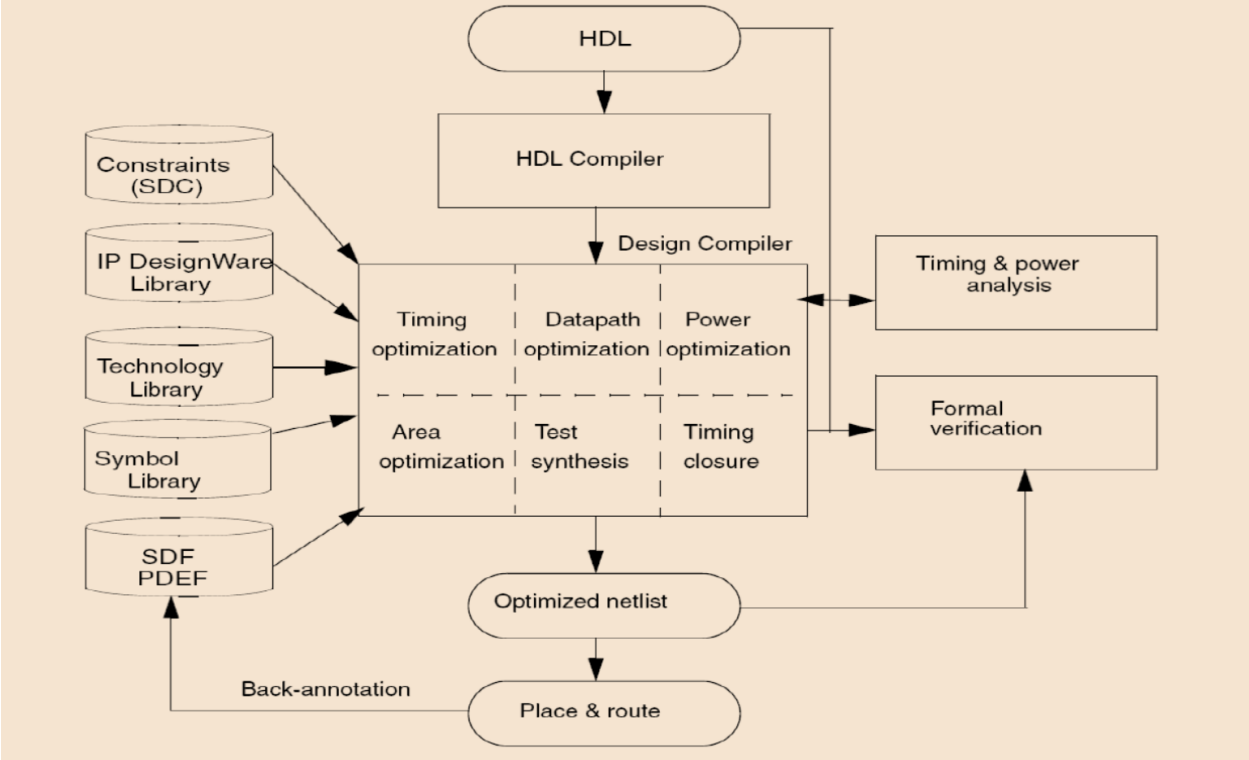

Figure 3.2: Synopsys Design Compiler flow

1. Design Compiler reads in technology libraries, Design Ware 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 Design Ware library. The technology library consists of basic logic

gates and flip-flops. The Design Ware 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.

3.3 Synthesis 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.

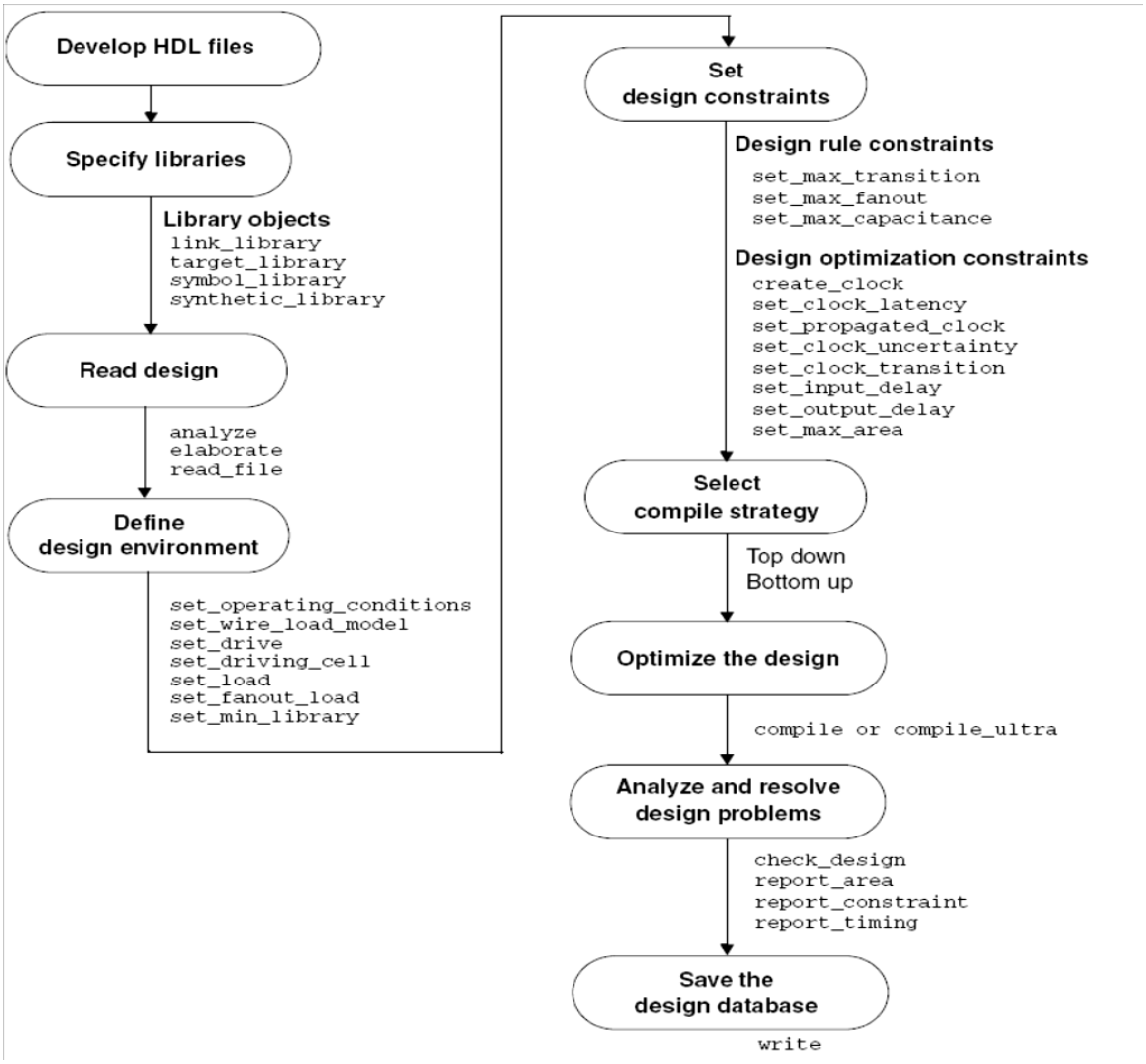

Figure 3.3 Synthesis Flow

3.4 Develop HDL Files:

The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL. These design descriptions need to be written carefully to achieve the best synthesis results possible. When writing HDL code, design

data management, design partitioning, and HDL coding style must be considered. Partitioning and coding style directly affect the synthesis and optimization processes.

Specify Libraries:

The Synopsys synthesis tool when invoked, through Design compiler command, reads a startup file, which must be present in the current working directory. This startup file is **synopsys_dc.setup** file. There should be two startup files present, one in the current working directory and other in the root directory in which Synopsys is installed. The local startup file in the current working directory should be used to specify individual design specifications. This file does not contain design dependent data. Its function is to load the Synopsys technology independent libraries and other parameters. The user in the startup files specifies the design dependent data. The settings provided in the current working directory override the ones specified in the root directory.

This step presents setup of basic library information. Design Compiler uses technology, symbol, and synthetic or Design Ware libraries to implement synthesis and to display synthesis results graphically. We should specify the link, target, symbol, and synthetic libraries for Design Compiler by using the *link_library*, *target_library*, *symbol_library*, and *synthetic_library* commands.

There are four important parameters that should be setup before one can start using the tool.

Search path:

This parameter is used to specify the synthesis tool all the paths that it should search when looking for a synthesis technology library for reference during synthesis.

Link Library:

Design Compiler uses the link library to resolve references. For a design to be complete, it must connect to all the library components and designs it references. This process is called linking the design or resolving references.

The *link_library* variable specifies a list of libraries and design files that Design Compiler can use to resolve references. When you load a design into memory, Design Compiler also loads all libraries specified in the *link_library* variable.

Target Library:

This library is used mainly for mapping all the logic gates from the target library. It also calculates the timing of the circuit, using the vendor-supplied timing data for these gates.

The *target_library* specification should only contain those standard cell libraries that you want Design Compiler to use when mapping your design's standard cells. Standard cells are cells such as combinational logic and registers. The *target_library* specification should not include any DesignWare libraries or macro libraries such as I/O pads or memories.

The *target_library* is a subset of the *link_library* and listed first in your list of link libraries.

Symbol Library:

It is the library that contains all the definitions of the graphic symbols that represent library cells in the design schematics, when you generate the design schematic, Design Compiler performs a one-to-one mapping of cells in the netlist to cells in the symbol library.

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.

Analyze and Elaborate:

Source code is usually best read in with analyze/elaborate

Executing *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.

Executing 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.

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

Design Rule Constraints DRCs:

Design rule constraints reflect technology-specific restrictions the design must meet in order to function as intended.

The design rule constraints include:

- Maximum transition time.
- Maximum fan-out.
- Minimum and maximum capacitance.
- Cell degradation.

DRCs are applied to the nets of the design in association to the pins of the cells from the technology library. Design compiler can't violate the DRCs, even if it means to violate the optimization constraints (area or speed). User can apply design rule constraints more restrictive than the default constraints set by the technology library, but these constraints can't be less restrictive.

Link:

Link resolves references in design hierarchy and link is done automatically with elaborate and also resolves both design units and library references.

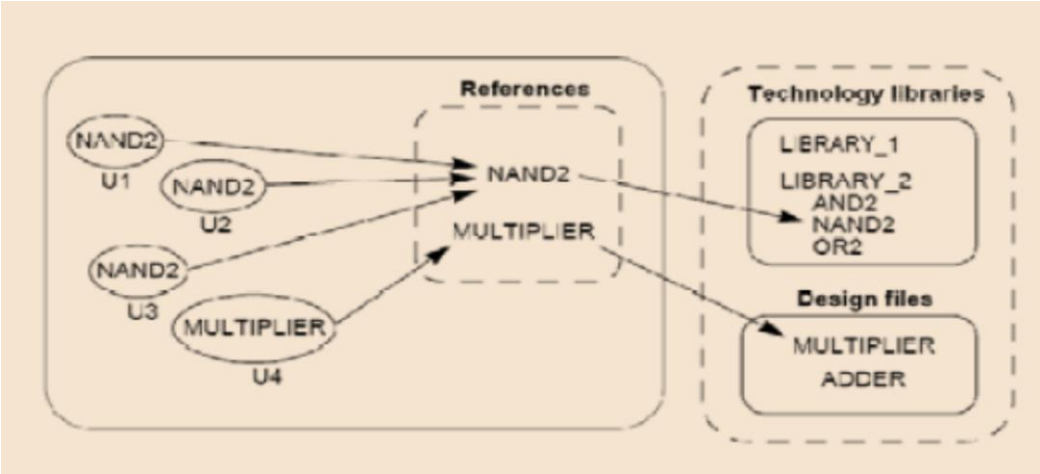

Figure 3.4: Linking hierarchy

3.5 Optimizing the Design:

Optimization is the Design Compiler synthesis step that maps the design to an optimal combination of specific target library cells, based on the design's functional, speed, and area requirements. Design Compiler provides options that enable you to customize and control optimization.

Design Compiler performs the following three levels of optimization:

- Architectural optimization.
- Logic-level optimization.
- Gate-level optimization.

Architectural Optimization:

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

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

Logic-Level Optimization:

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

- Flattening.
- Structuring.

Flattening:

Flattening is a common academic term for reducing logic to a 2-level AND/OR representation. DC uses this approach to remove all intermediate variables and

parenthesis (using Boolean distributive laws) in order to optimize the design. This option is set to “false” by default. It is useful for speed optimization because it leads to just two levels of combinational logic.

Structuring:

Structuring is used for designs containing regular structured logic, for e.g., a carry-look-ahead adder. It is enabled by default for timing only. When structuring, DC adds intermediate variables that can be factored out. This enables sharing of logic that in turn results in reduction of area. Structuring comes in two flavors: timing (default) and Boolean optimization. The latter is a useful method of reducing area, but has a greater impact on timing.

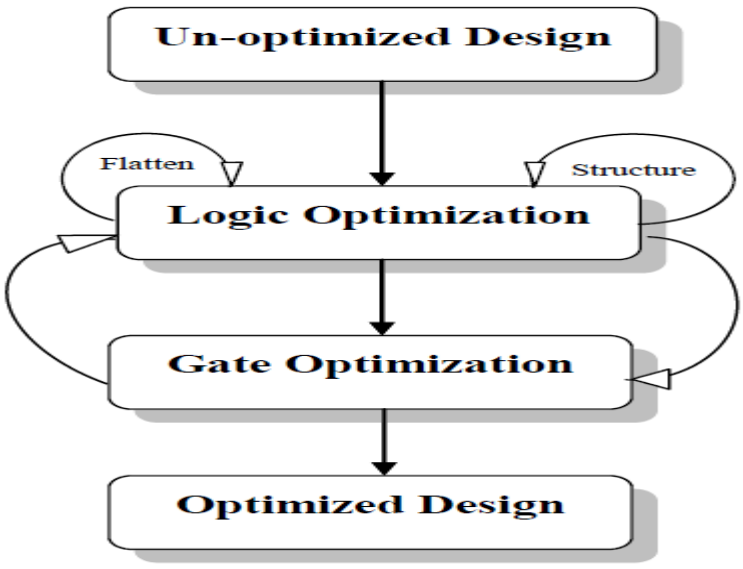

Figure 3.5: Optimization Flow

Gate Level Optimization:

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

- Mapping.
- Delay Optimization.
- Design Rule Fixing.

Mapping:

- This process uses gates (combinational and sequential) from the target technology libraries to generate a gate-level implementation of the design whose goal is to meet timing and area goals.

Delay Optimization:

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

Design Rule Fixing:

- The process goal is to correct design rule violations by inserting buffers or resizing existing cells. Design Compiler tries to fix these violations

without affecting timing and area results, but if necessary, it does violate the optimization constraints.

3.6 Analyze and Resolve design problems:

In this stage Design Compiler generates reports which are used to analyze and debug the design. Reports can be generated before and after compiling the design. Reports that have been generated before compiling the design are used to check that you have set all the attributes, constraints, and design rules properly, while reports that have been generated after compiling the design are used to analyze the results and debug the design for meeting the design goals that have been required.

Once the reports and various output files of different extensions and usages have been generated after compiling the design, these are fed into the IC Compiler to be the input starting data for the back end.

The Design compiler generates a gate-level netlist file with the timing and area constraints which is the input file to the IC Compiler. The output synthesized file from DC can be directly fed to IC for physical implementation, which is a Verilog file with .v extension or synthesized constrained file with .ddc extension. It is always preferred to use .ddc file as input synthesized file as it contains constraints applied for optimizations.

Chapter 4: Analysis and Conclusion

Asynchronous FIFO design requires careful attention to details from pointer generation techniques to full and empty generation. Ignorance of important details will generally result in a design that is totally wrong. Synchronization of FIFO pointers into the opposite clock domain is safely accomplished using Gray code pointers. Careful partitioning of the FIFO modules along clock boundaries with all outputs registered can facilitate synthesis and static timing analysis within the two asynchronous clock domains.

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 test bench 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.

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.

Future Work:

i) **Formal Verification:** Formal verification techniques perform validation of a design using mathematical methods without the need for technological considerations, such as timing and physical effects. They check for logical

functions of a design by comparing it against the reference design. A number of EDA tool vendors have developed the formal verification tools. However, only recently, Synopsys also introduced to the market its own formal verification tool called Formality. The purpose of the formal verification in the design flow is to validate the RTL against RTL, gate-level netlist against the RTL code, or the comparison between gate-level to gate-level netlists.

ii) **Post Silicon Validation:** is the last step in the development of a semiconductor integrated circuit. During the pre-silicon process, engineers test devices in a virtual environment with sophisticated simulation, emulation, and formal verification tools. In contrast, post-silicon validation tests occur on actual devices running at-speed in commercial, real-world system boards using logic analyzer and assertion-based tools. **Testing:** A speck of dust on a wafer is sufficient to kill chip. Yield of any chip is < 100%. Tests after manufacturing are must before delivery to customers to only ship good parts.

REFRENCES

1) ASIC/IC Design-for-Test Process Guide, Software Version 8.6_1, Mentor Graphics.
2) http://semicon.sanyo.com/en/asic/user/cts.php (11/12/2012)
3) http://people.ee.duke.edu/~krish/teaching/Lectures/Testing.1.pdf(11/15/2012)
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. <http://www.encyclopedia.com>. (11/6/2012)
6) Clifford E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," SNUG 2002 – www.sunburst-design.com/papers/CummingsSNUG2002SJ_ FIFO1.pdf.(11/4/2012)
7) Clifford E. Cummings, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs," SNUG 2001 – www.sunburst-design.com/papers/ CummingsSNUG2001SJ_AsyncClk.pdf(11/4/2012)
8) http://asic-soc.blogspot.com/2007/11/asynchronous-fifo-design.html(11/8/2012)
9) "FIFO Architecture, Functions, and Applications" - Texas Instruments(11/8/2012)
10) Synopsys Design Complier User Guide Version D-2010.03, March2012.
11) ASIC Chip Synthesis 2ed from www.scribd.com(10/12/2012)
12) Synopsys IC Complier User Guide Version D-2010.03, March2012.

13) http://www.utdallas.edu/~zhoud/Lecture1.pdf(11/04/2012)

14) Synopsys Low-Power Flow User Guide, Version F-2011.09, September 2012.

15) Power Compiler User Guide , Version F-2011.09, September 2012.

APPENDIX

1) Basic Commands for Defining Design rules:

The commands that define the design environment are:

- set_max_capacitance

  Sets a maximum capacitance for the nets attached to the specified ports or to all the nets in a design.

- set_max _fanout

  Sets the expected fan-out load value for output ports.

- set_max_transition

  Sets a maximum transition time for the nets attached to the specified ports or to all the nets in a design.

- xset_min_capacitance

  Sets a minimum capacitance for the nets attached to the specified ports or to all the nets in a design.

1.1) Commands for Defining Design Environments

The commands that define the design environment are:

- set_drive

  Sets the drive value of input or in-out ports. The set_drive command is superseded by the set_driving_cell command.

- set_driving_cell

  Sets attributes on input or in-out ports, specifying that a library cell or library pin drives the ports. This command associates a library pin with an input port so that delay calculators can accurately model the drive capability of an external driver.

- set_fanout_load

  Defines the external fan-out load values on output ports.

- set_load

  Defines the external load values on input and output ports and nets.

- set_operating_conditions

  Defines the operating conditions for the current design.

- set_wire_load_model

  Sets the wire load model for the current design or for the specified ports. With this command, one can specify the wire load model to use for the external net connected to the output port.

1.2) Commands for Setting Design Constraints

- create_clock

  Creates a clock object and defines its waveform in the current design.

  set_clock_latency, set_clock_uncertainty, set_clock_transition

  Sets clock attributes on clock objects or flip-flop clock pins.

- set_input_delay

  Sets input delay on pins or input ports relative to a clock signal.

- set_max_area

  Specifies the maximum area for the current design.

- set_output_delay

  Sets output delay on pins or output ports relative to a clock signal.

- group_path

  Groups a set of paths or endpoints for cost function calculation. This command is used to create path groups, to add paths to existing groups, or to change the weight of existing groups.

- set_false_path

  Marks paths between specified points as false. This command eliminates the selected paths from timing analysis.

- set_max_delay

  Specifies a maximum delay target for selected paths in the current design.

- set_min_delay

  Specifies a minimum delay target for selected paths in the current design.

1.3) Commands for Analyzing and Resolving Design Problems

- all_connected

  Lists all fan outs on a net.

- all_registers

  Lists sequential elements or pins in a design.

- check_design

  Checks the internal representation of the current design for consistency and issues error and warning messages as appropriate.

- check_timing

  Checks the timing attributes placed on the current design.

- get_attribute

  Reports the value of the specified attribute.

- link

  Locates the reference for each cell in the design.

- report_area

  Provides area information and statistics on the current design.

- report_attribute

  List the attributes and their values for the selected object. An object can be a cell, net, pin, port, instance, or design.

- report_cell

Lists the cells in the current design and their cell attributes.

- report_clock

  Displays clock-related information on the current design.

- report_constraint

  Lists the constraints on the current design and their cost, weight, and weighted cost.

- report_delay_calculation

  Reports the details of a delay arc calculation.

- report_design

  Displays the operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs defined for the current design.

- report_net

  Displays net information for the design of the current instance, if set; otherwise, displays net information for the current design.

- report_path_group

Lists all timing path groups in the current design.

- report_port

  Lists information about ports in the current design.

- report_qor

  Displays information about the quality of results and other statistics for the current design.

- report_resources

  Displays information about the resource implementation.

- report_timing

  Lists timing information for the current design.

- report_timing_requirements

- Lists timing path requirements and related information.

- report_transitive_fanin

  Lists the fan-in logic for selected pins, nets, or ports of the current instance.

- report_transitive_fanout

Lists the fan-out logic for selected pins, nets, or ports of the current instance.

2) Script file to synthesize the FIFO design top down:

****

Author: Raviteja Podila

Filename: fifo_top.tcl

This script file synthesizes the FIFO design top down.

****

#analyzing and elaborating the low level design

analyze -f verilog full_flag.v

elaborate full_flag

check_design

analyze -f verilog empty_flag.v

elaborate empty_flag

check_design

analyze -f verilog write_pntr.v

elaborate write_pntr

```
check_design

analyze -f verilog read_pntr.v

elaborate read_pntr

check_design

analyze -f verilog memory.v

elaborate memory

check_design

analyze -f verilog aasd.v

elaborate aasd

check_design

analyze -f verilog fifo_top.v

elaborate fifo_top

check_design

#ungrouping the design hierarchy

ungroup -all –flatten
```

```
#generating write clock and read clock

create_clock wrt_clk -period 5

create_clock rd_clk -period 25

#setting false path for the signals crossing the clock domain

set_false_path -from [get_clocks wrt_clk] -to [get_clocks rd_clk]

set_false_path -from [get_clocks rd_clk] -to [get_clocks wrt_clk]

#to constraint the area

set_max_area 29790 -ignore_tns

#compiling the design

compile

check_design

#generating the SDF files

write_sdf fifo.sdf

#reporting the area and timing

report_area >> fifo_area_new_1.rpt
```

report_timing >> fifo_timing_new_1.rpt

2.1) Area report for top down synthesis:

****

Report: area

Design: fifo_top

****

Information: Updating design information... (UID-85)

Library(s) Used: saed90nm_max

(File:/opt/ECE_Lib/SAED_EDK90nm/Digital_Standard_Cell_Library/synopsys/models/saed90nm_max.db)

Number of ports: 62

Number of nets: 1726

Number of cells: 1684

Number of references: 20

Combinational area: 15767.195304

Noncombinational area: 14713.237717

Net Interconnect area: 2577.459163

Total cell area: 30480.433021

Total area: 33057.89218

2.2) Timing report for top down synthesis:

```
****************************************

Report: timing

        -path full

        -delay max

        -max_paths 1

Design: fifo_top

****************************************

Operating Conditions: BEST Library: saed90nm_max

Wire Load Model Mode: enclosed

Start point: rpt/read_address_reg [1]

         (rising edge-triggered flip-flop clocked by rd_clk)

Endpoint: mem/rdata_reg [2]

         (rising edge-triggered flip-flop clocked by rd_clk)

Path Group: rd_clk
```

Path Type: max

Des/Clust/Port	Wire Load Model	Library
fifo_top	35000	saed90nm_max

Point	Incr	Path
clock rd_clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
rpt/read_address_reg [1]/CLK (DFFARX1)	0.00	0.00 r
rpt/read_address_reg [1]/Q (DFFARX1)	0.59	0.59 f
U1744/Q (NBUFFX2)	0.21	0.80 f
U796/Q (NBUFFX2)	0.15	0.94 f
U1621/Q (MUX41X1)	0.08	1.03 r
U1725/Q (MUX41X1)	0.08	1.11 r
U1723/Q (MUX21X1)	0.06	1.17 r

mem/rdata_reg [2]/D (DFFX1)		0.00	1.17 r
data arrival time	1.17		
clock rd_clk (rise edge)		25.00	25.00
clock network delay (ideal)		0.00	25.00
mem/rdata_reg [2]/CLK (DFFX1)		0.00	25.00 r
library setup time		-0.03	24.97
data required time	24.97		

---

data required time	24.97
data arrival time	-1.17

---

slack (MET) 23.80

Start point: wpt/write_address_reg [0]

(rising edge-triggered flip-flop clocked by wrt_clk)

Endpoint: mem/memory_reg [28] [6]

(rising edge-triggered flip-flop clocked by wrt_clk)

Path Group: wrt_clk

Path Type: max

Des/Clust/Port Wire Load Model Library

---

fifo_top 35000 saed90nm_max

Point	Incr	Path	
clock wrt_clk (rise edge)	0.00	0.00	
clock network delay (ideal)	0.00	0.00	
wpt/write_address_reg [0]/CLK (DFFARX1)		0.00	0.00 r
wpt/write_address_reg [0]/Q (DFFARX1)	0.58	0.58 f	
U1544/Q (NBUFFX2)	0.23	0.81 f	
U760/Q (NBUFFX2)	0.15	0.96 f	

U895/Q (MUX41X1)		0.11	1.07 f
U831/Q (MUX41X1)		0.07	1.13 f
U830/Q (MUX21X1)		0.05	1.18 f
U829/Q (AO22X1)		0.47	1.65 f
U768/Q (NBUFFX2)		0.14	1.80 f
U1251/Q (AO22X1)		0.07	1.87 f
mem/memory_reg [28] [6]/D (DFFX1)		0.00	1.87 f
data arrival time	1.87		
clock wrt_clk (rise edge)		5.00	5.00
clock network delay (ideal)		0.00	5.00
mem/memory_reg [28] [6]/CLK (DFFX1)		0.00	5.00 r
library setup time		-0.03	4.97
data required time	4.97		

---

data required time	4.97

data arrival time -1.87

---

slack (MET) 3.10

Start point: wpt/write_address_reg [0]

(rising edge-triggered flip-flop clocked by wrt_clk)

Endpoint: mem/memory_reg [28][6]

(rising edge-triggered flip-flop clocked by wrt_clk)

Path Group: wrt_clk

Path Type: max

Des/Clust/Port Wire Load Model Library

---

fifo_top 35000 saed90nm_max

Point Incr Path

---

clock wrt_clk (rise edge) 0.00 0.00

clock network delay (ideal)	0.00	0.00
wpt/write_address_reg[0]/CLK (DFFARX1)	0.00	0.00 r
wpt/write_address_reg[0]/Q (DFFARX1)	0.58	0.58 f
U1544/Q (NBUFFX2)	0.23	0.81 f
U760/Q (NBUFFX2)	0.15	0.96 f
U895/Q (MUX41X1)	0.11	1.07 f
U831/Q (MUX41X1)	0.07	1.13 f
U830/Q (MUX21X1)	0.05	1.18 f
U829/Q (AO22X1)	0.47	1.65 f
U768/Q (NBUFFX2)	0.14	1.80 f
U1251/Q (AO22X1)	0.07	1.87 f
mem/memory_reg[28][6]/D (DFFX1)	0.00	1.87 f
data arrival time	1.87	
clock wrt_clk (rise edge)	5.00	5.00
clock network delay (ideal)	0.00	5.00

mem/memory_reg[28][6]/CLK (DFFX1) 0.00 5.00 r

library setup time -0.03 4.97

data required time 4.97

---

data required time 4.97

data arrival time -1.87

---

slack (MET) 3.10

2.3) Script file to synthesize the FIFO design bottom up:

****

Author: Raviteja Podila

Filename: fifo_up.tcl

This script file synthesizes the FIFO design bottom up.

****

#reading the lower level compiled files

read_file -f ddc full_flag_bot_up_II.ddc

read_file -f ddc empty_flag_bot_up_II.ddc

```
read_file -f ddc write_pntr_bot_up_II.ddc

read_file -f ddc read_pntr_bot_up_II.ddc

read_file -f ddc memory_bot_up_II.ddc

read_file -f ddc aasd_bot_up_II.ddc

analyze -f verilog fifo_top.v

elaborate fifo_top

check_design

create_clock wrt_clk -period 5

create_clock rd_clk -period 25



#setting false path for the signals crossing the clock domain

set_false_path -from [get_clocks wrt_clk] -to [get_clocks rd_clk]

set_false_path -from [get_clocks rd_clk] -to [get_clocks wrt_clk]

#compiling the empty flag design

analyze -f verilog empty_flag.v
```

```
elaborate empty_flag

check_design

create_clock -period 10 -name sys_clock

compile

check_design

write -f ddc -o empty_flag_bot_up.ddc

quit

#compiling the write pointer design

analyze -f verilog write_pntr.v

elaborate write_pntr

check_design

create_clock clock -period 5

compile

check_design

write -f ddc -o write_pntr_bot_up.ddc
```

```
quit

#compiling the read pointer design

analyze -f verilog read_pntr.v

elaborate read_pntr

check_design

create_clock clock -period 10

compile

check_design

write -f ddc -o read_pntr_bot_up.ddc

quit

#compiling the memory design

analyze -f verilog memory.v

elaborate memory

check_design

create_clock wclk -period 5
```

```
create_clock rclk -period 10

compile

check_design

write -f ddc -o memory_bot_up.ddc

quit

#compiling the synchronizer design

analyze -f verilog aasd.v

elaborate aasd

check_design

create_clock clock -period 5

compile

check_design

write -f ddc -o aasd_bot_up.ddc

quit
```

2.4) Area report for top level module:

```
****************************************
Report : area
Design : fifo_top
****************************************
Library(s) Used:

gtech (File: /usr/synopsys/A-2007.12-SP5/libraries/syn/gtech.db)

saed90nm_max (File:

/opt/ECE_Lib/syn_lib_2.2/DATABASE_SAED90NM_08_September_2008/synopsys/m
odels/saed90nm_max.db)

Number of ports:                62

Number of nets:                 77

Number of cells:                 9

Number of references:            7

Combinational area:    7748.808294

Noncombinational area:          0.000000

Net Interconnect area:          undefined (No wire load specified)
```

Total cell area: 7748.808294

Total area: undefined

2.5) Timing report for top level module:

****

Report : timing

-path full

-delay max

-max_paths 1

Design : fifo_top

****

A fan-out number of 1000 was used for high fan-out net computations.

Operating Conditions: WORST Library: saed90nm_max

Wire Load Model Mode: top

Start point: rpt/read_address_reg[0]

(rising edge-triggered flip-flop clocked by rd_clk)

Endpoint: mem/rdata_reg[0]

(rising edge-triggered flip-flop clocked by rd_clk)

Path Group: rd_clk

Path Type: max

Point	Incr	Path
clock rd_clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
rpt/read_address_reg[0]/clocked_on (**SEQGEN**)	0.00	0.00 r
rpt/read_address_reg[0]/Q (**SEQGEN**)	0.00	0.00 f
rpt/read_address[0] (read_pntr_size6)	0.00	0.00 f
mem/raddr[0] (memory_width16_addr5)	0.00	0.00 f
mem/B_7/Z (GTECH_BUF)	0.00	0.00 f
mem/C1243/S0 (fifo_top_MUX_OP_32_5_16)	0.00	0.00 f
alt42/U170/Q (MUX41X1)	0.63	0.63 f
alt42/U167/Q (MUX41X1)	0.60	1.23 f

alt42/U166/Q (MUX21X1)		0.29	1.52 f
mem/C1243/Z_15 (fifo_top_MUX_OP_32_5_16)		0.00	1.52 f
mem/rdata_reg[0]/next_state (**SEQGEN**)		0.00	1.52 f
data arrival time	1.52		
clock rd_clk (rise edge)		25.00	25.00
clock network delay (ideal)		0.00	25.00
mem/rdata_reg[0]/clocked_on (**SEQGEN**)		0.00	25.00 r
library setup time		0.00	25.00
data required time	25.00		

---

data required time	25.00
data arrival time	-1.52

---

slack (MET) 23.48

Start point: wpt/write_address_reg[0]

(rising edge-triggered flip-flop clocked by wrt_clk)

Endpoint: mem/memory_reg[0][0]

(rising edge-triggered flip-flop clocked by wrt_clk)

Path Group: wrt_clk

Path Type: max

Point	Incr	Path
clock wrt_clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
wpt/write_address_reg[0]/clocked_on (**SEQGEN**)	0.00	0.00 r
wpt/write_address_reg[0]/Q (**SEQGEN**)	0.00	0.00 f
wpt/write_address[0] (write_pntr_size6)	0.00	0.00 f
mem/waddr[0] (memory_width16_addr5)	0.00	0.00 f
mem/B_2/Z (GTECH_BUF)	0.00	0.00 f
mem/C1242/S0 (fifo_top_MUX_OP_32_5_16_1)	0.00	0.00 f

alt45/U170/Q (MUX41X1)	0.63	0.63 f
alt45/U167/Q (MUX41X1)	0.60	1.23 f
alt45/U166/Q (MUX21X1)	0.29	1.52 f
mem/C1242/Z_15 (fifo_top_MUX_OP_32_5_16_1)	0.00	1.52 f
mem/C1241/Z_0 (*SELECT_OP_2.16_2.1_16)	0.00	1.52 f
mem/memory_reg[0][0]/next_state (**SEQGEN**)	0.00	1.52 f
data arrival time	1.52	

---

data required time	5.00
data arrival time	-1.52

---

slack (MET)	3.48

3) Gate files created using compiler lower level module (FIFO):

3.1) Empty Flag Gates:

```
**********************************************************************************************
Author: Raviteja Podila
Filename: fifo_eflag.v
Gate file for FIFO empty flag.
**********************************************************************************************
`timescale 1ns/1ns

module empty_flag_gates ( read_pointer, write_pointer, empty_flag );

input [5:0] read_pointer;

input [5:0] write_pointer;

output empty_flag;

wire n3, n4, n5, n6, n7, n8, n9;

NOR4X0 U3 ( .IN1(n3), .IN2(n4), .IN3(n5), .IN4(n6), .QN(empty_flag) );

XOR2X1 U4 ( .IN1(write_pointer[4]), .IN2(read_pointer[4]), .Q(n6) );

XOR2X1 U5 ( .IN1(write_pointer[1]), .IN2(read_pointer[1]), .Q(n5) );

XOR2X1 U6 ( .IN1(write_pointer[0]), .IN2(read_pointer[0]), .Q(n4) );
```

```
NAND3X0 U7 ( .IN1(n7), .IN2(n8), .IN3(n9), .QN(n3) );

XNOR2X1 U8 ( .IN1(write_pointer[2]), .IN2(read_pointer[2]), .Q(n9) );

XNOR2X1 U9 ( .IN1(write_pointer[3]), .IN2(read_pointer[3]), .Q(n8) );

XNOR2X1 U10 ( .IN1(write_pointer[5]), .IN2(read_pointer[5]), .Q(n7) );

endmodule
```

3.2) Full Flag Gates:

```
**********************************************************************************************

Author: Raviteja Podila

Filename: fifo_fflag.v

Gate file for FIFO full flag.

**********************************************************************************************

`timescale 1ns/1ns

module full_flag_gates ( write_pointer, read_pointer, full_flag, clk_en );

input [5:0] write_pointer;

input [5:0] read_pointer;

input clk_en;

output full_flag;
```

```
wire n9, n10, n11, n12, n13, n14, n15, n16;

NAND2X0 U10 ( .IN1(clk_en), .IN2(n9), .QN(full_flag) );

NAND4X0 U11 ( .IN1(n10), .IN2(n11), .IN3(n12), .IN4(n13), .QN(n9) );

NOR3X0 U12 ( .IN1(n14), .IN2(n15), .IN3(n16), .QN(n13) );

XOR2X1 U13 ( .IN1(write_pointer[1]), .IN2(read_pointer[1]), .Q(n16) );

XOR2X1 U14 ( .IN1(write_pointer[3]), .IN2(read_pointer[3]), .Q(n15) );

XOR2X1 U15 ( .IN1(write_pointer[2]), .IN2(read_pointer[2]), .Q(n14) );

XNOR2X1 U16 ( .IN1(write_pointer[0]), .IN2(read_pointer[0]), .Q(n12) );

XOR2X1 U17 ( .IN1(write_pointer[4]), .IN2(read_pointer[4]), .Q(n11) );

XOR2X1 U18 ( .IN1(write_pointer[5]), .IN2(read_pointer[5]), .Q(n10) );

endmodule
```

3.3) Read Pointer Gates:

```
**********************************************************************************************
Author: Raviteja Podila
Filename: fifo_rpointer.v
Gate file for FIFO read pointer.
**********************************************************************************************
`timescale 1ns/1ns

module read_pntr_gates ( clock, reset, incrementer, flag, read_pointer, read_address);

output [5:0] read_pointer;

output [4:0] read_address;

input clock, reset, incrementer, flag;

wire N2, N3, N4, N5, N6, n11, n14, n17, n20, n22, n24, n36, n37, n38, n39,

n40, n41, n42, n43, n44, n45, n46, n47;

wire [5:0] b_next;

DFFARX1 \b_next_reg[0] ( .D(n24), .CLK(clock), .RSTB(reset), .Q(b_next[0]),

.QN(n43) );

DFFARX1 \b_next_reg[1] ( .D(n22), .CLK(clock), .RSTB(reset), .Q(b_next[1]),
```

.QN(n44));

DFFARX1 \b_next_reg[2] (.D(n20), .CLK(clock), .RSTB(reset), .Q(b_next[2]),

.QN(n45));

DFFARX1 \b_next_reg[3] (.D(n17), .CLK(clock), .RSTB(reset), .Q(b_next[3]),

.QN(n46));

DFFARX1 \b_next_reg[4] (.D(n14), .CLK(clock), .RSTB(reset), .Q(b_next[4]),

.QN(n47));

DFFARX1 \b_next_reg[5] (.D(n11), .CLK(clock), .RSTB(reset), .Q(b_next[5]),

.QN(n42));

3.4) Write Pointer Gates:

****

Author: Raviteja Podila

Filename: fifo_wpointer.v

Gate file for FIFO write pointer.

****

`timescale 1ns/1ns

module write_pntr_gates (clock, reset, incrementer, write_flag, empty_flag, wclk_en,

```
write_pointer, write_address );

output [5:0] write_pointer;

output [4:0] write_address;

input clock, reset, incrementer, write_flag, empty_flag;

output wclk_en;

wire N3, N4, N5, N6, N7, n16, n17, n19, n21, n23, n25, n27, n39, n40, n41,

n42, n43, n44, n45, n46, n48, n49, n50, n51, n52, n53;

wire [5:0] bnext;

DFFX1 wclk_en_reg ( .D(n16), .CLK(clock), .Q(wclk_en) );

DFFARX1 \bnext_reg[0] ( .D(n27), .CLK(clock), .RSTB(reset), .Q(bnext[0]),

.QN(n49) );

DFFARX1 \bnext_reg[1] ( .D(n25), .CLK(clock), .RSTB(reset), .Q(bnext[1]),

.QN(n50) );

DFFARX1 \bnext_reg[2] ( .D(n23), .CLK(clock), .RSTB(reset), .Q(bnext[2]),

.QN(n51) );
```

DFFARX1 \bnext_reg[3] (.D(n21), .CLK(clock), .RSTB(reset), .Q(bnext[3]),

.QN(n52));

DFFARX1 \bnext_reg[4] (.D(n19), .CLK(clock), .RSTB(reset), .Q(bnext[4]),

.QN(n53));

3.5) Memory Gates (FIFO Ram):

****

Author: Raviteja Podila

Filename: fifo_mgates.v

Gate file for FIFO memory module.

****

`timescale 1ns/1ns

module memory_gates (wdata, waddr, w_clken, wclk, raddr, rclk, rdata);

input [15:0] wdata;

input [4:0] waddr;

input [4:0] raddr;

output [15:0] rdata;

input w_clken, wclk, rclk;

wire N10, N11, N12, N13, N14, N15, N16, N17, N18, N19, \memory[31][15] ,

\memory[31][14] , \memory[31][13] , \memory[31][12] ,

\memory[31][11] , \memory[31][10] , \memory[31][9] , \memory[31][8] ,

\memory[31][7] , \memory[31][6] , \memory[31][5] , \memory[31][4] ,

\memory[31][3] , \memory[31][2] , \memory[31][1] , \memory[31][0] ,

\memory[30][15] , \memory[30][14] , \memory[30][13] ,

\memory[30][12] , \memory[30][11] , \memory[30][10] , \memory[30][9] ,

\memory[30][8] , \memory[30][7] , \memory[30][6] , \memory[30][5] ,

\memory[30][4] , \memory[30][3] , \memory[30][2] , \memory[30][1] ,

\memory[30][0] , \memory[29][15] , \memory[29][14] , \memory[29][13] ,

\memory[29][12] , \memory[29][11] , \memory[29][10] , \memory[29][9] ,

\memory[29][8] , \memory[29][7] , \memory[29][6] , \memory[29][5] ,

\memory[29][4] , \memory[29][3] , \memory[29][2] , \memory[29][1] ,

\memory[29][0] , \memory[28][15] , \memory[28][14] , \memory[28][13] ,

\memory[28][12] , \memory[28][11] , \memory[28][10] , \memory[28][9] ,

\memory[28][8] , \memory[28][7] , \memory[28][6] , \memory[28][5] ;

4) Functional Verification: Simulation Results of the Project Design (ASYNCHRONOUS FIFO INTERFACE)

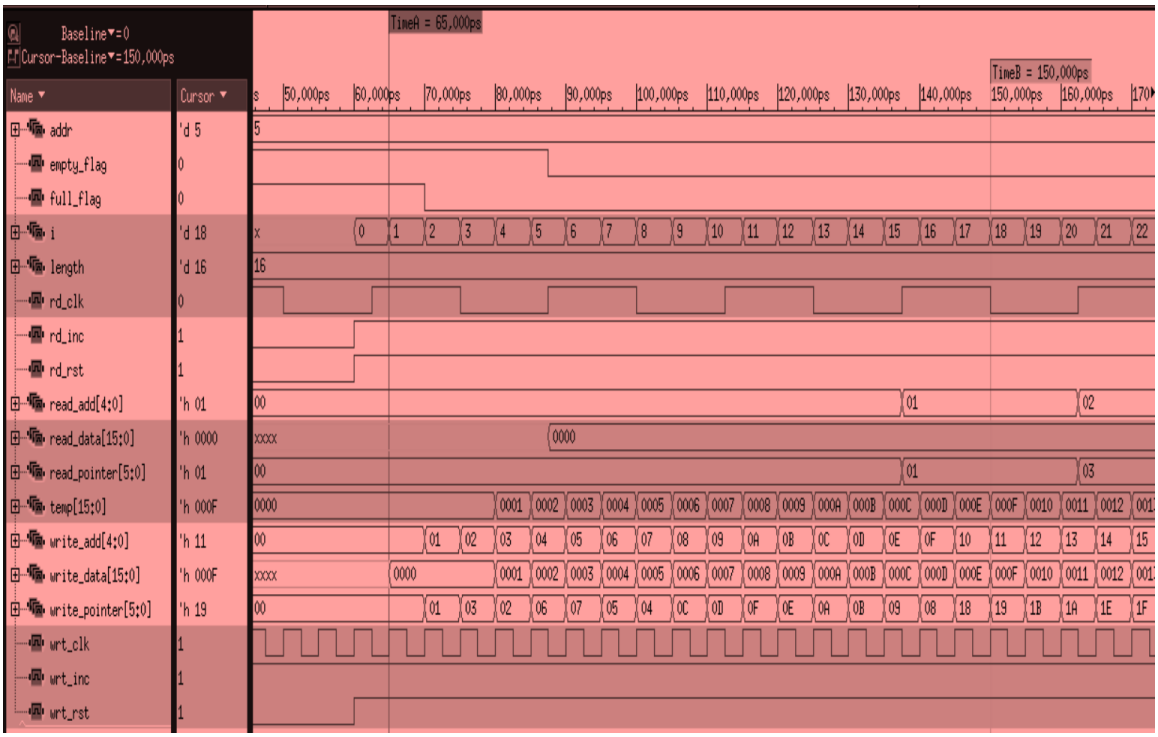

Figure A.1 Write into Memory:

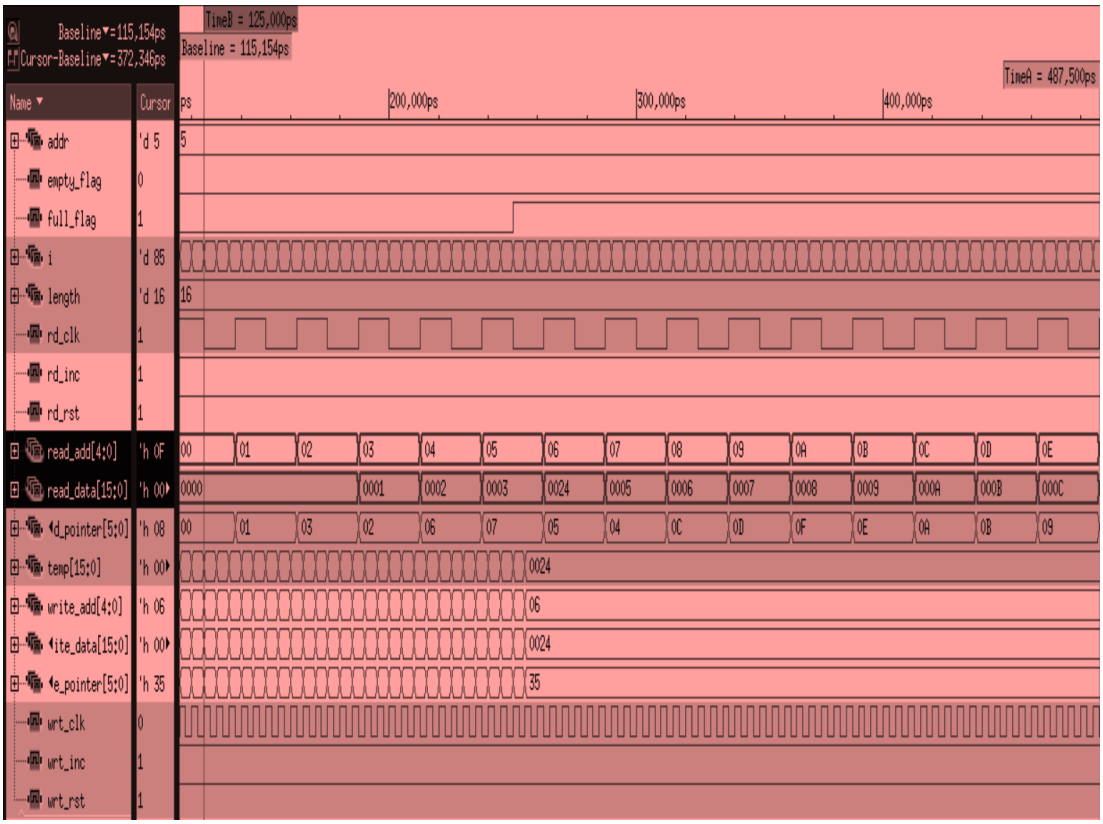

Figure A.2 Read From Memory:

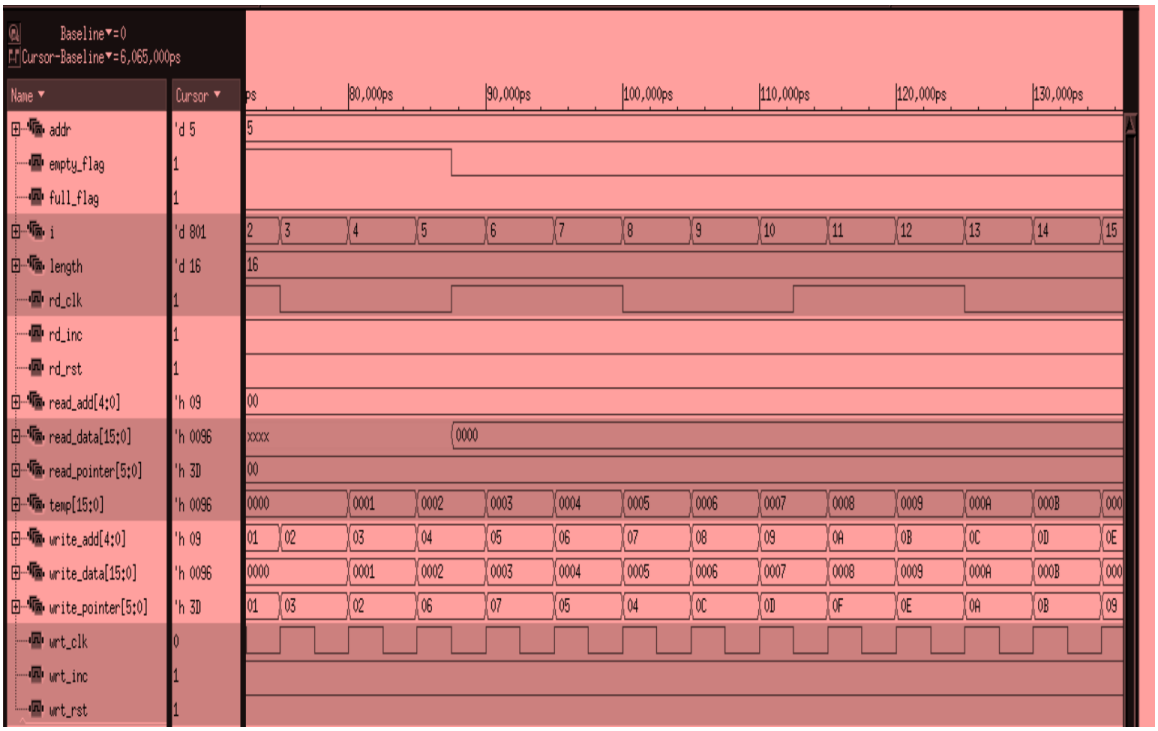

Figure A.3 Empty Flag:

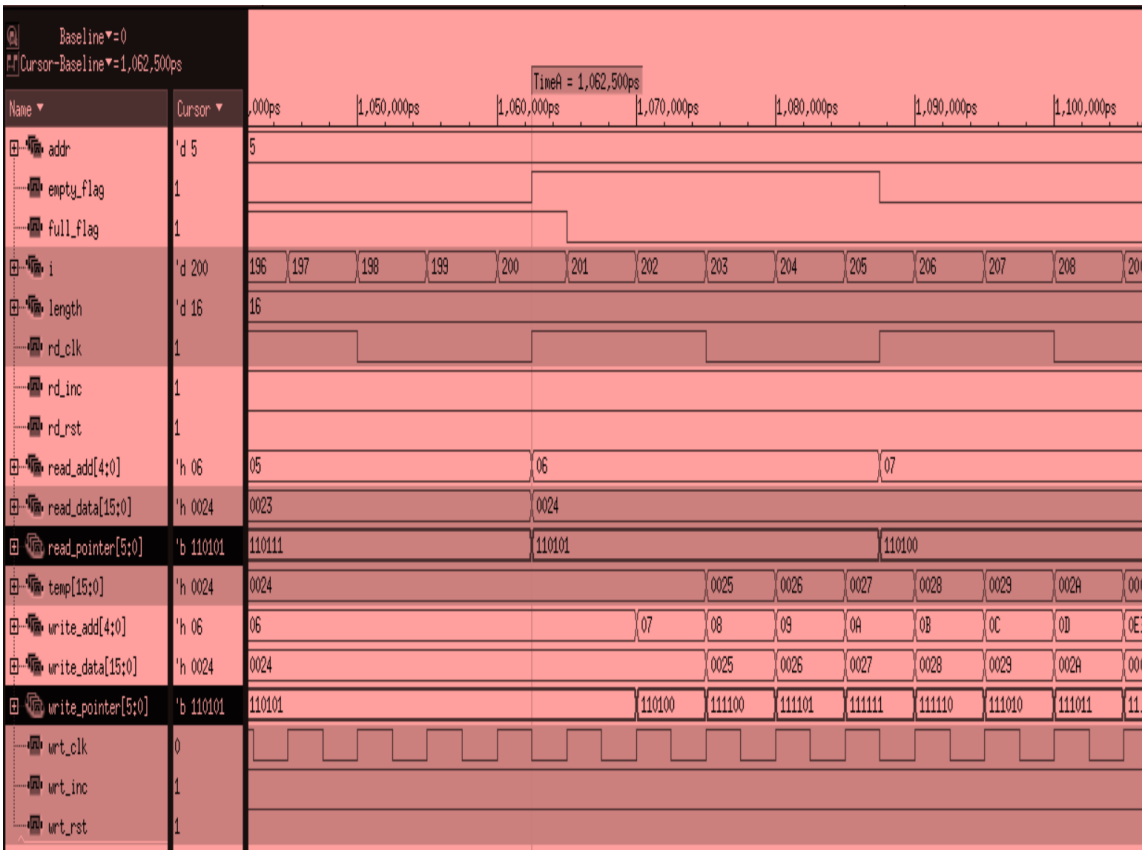

Figure A.4 Clear Empty Flag:

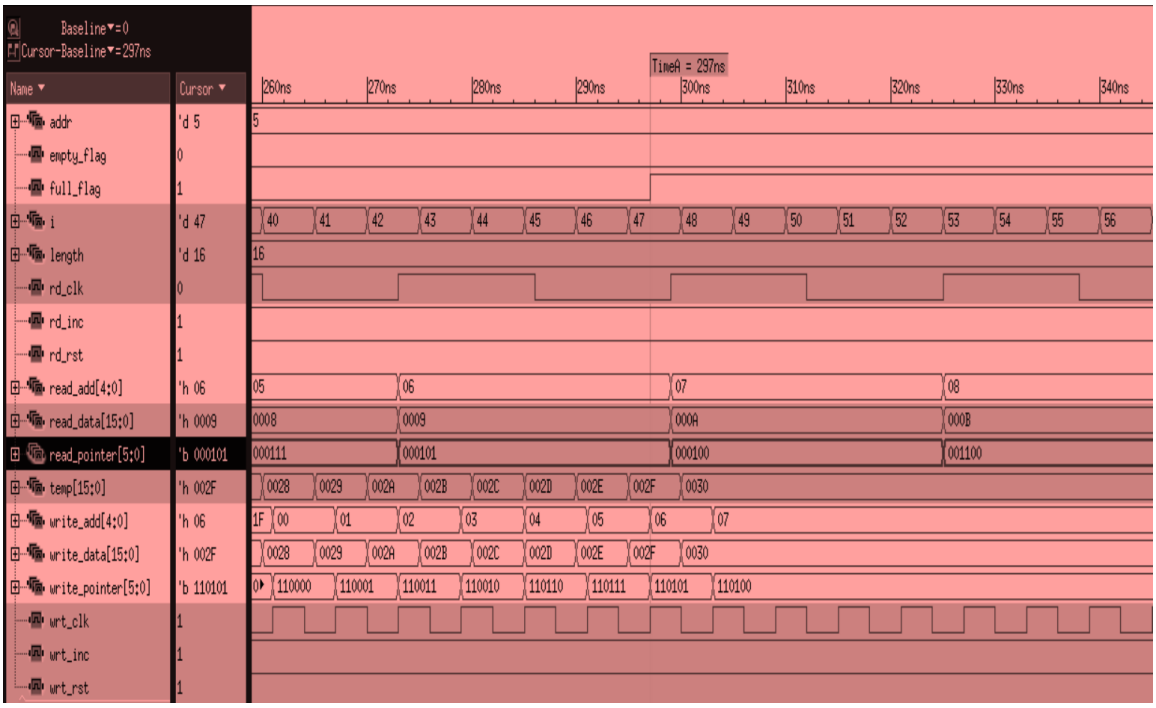

Figure A.5 Full Flag High:

5) This is the Fault Summary Report for Front-end; Showing the Test Coverage of100%.

```
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.
```

6) Boundary Scan and Logic Scan Insertion Report:

```
###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 ;
```

7) 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
```