CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Low Power ASIC Design, a Comparative Study

A graduate project submitted in partial fulfillment of

the requirements for the degree of Masters of Science

in Electrical Engineering

By:

Silvia Said

May 2012

The graduate project of Silvia Said is approved:

Dr. Ronald W. Mehler Date

Dr. Kang J Chang Date

Dr. Ramin Roosta, Chair Date

California State University, Northridge

Acknowledgement

Apart from the efforts of me, the success of any project depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this project.

I would like to show my greatest appreciation to Dr. Ramin Roosta for encouraging me to take my project in the field of ASIC Design and learn how to implement a design on ASIC, starting from the front end and continuing to the back end. I would thank Dr. Roosta once again for sending me to take the Synopsys training for IC Compiler1 in which I learned a lot from it and it was a very successful training.
I could not say thank you enough for his tremendous support and help. Without his encouragement and guidance this project would not have been materialized.

I also thank Dr. Ronald W. Mehler as he taught me the front end of the ASIC flow design, which was valuable and help me a lot for saving time in the project as I had already known a lot of tricks.
I also thank Dr. Kang J Chang for his acceptance to be in my project committee.

I would also like to thank Maninder Singh and Harshit Jariwala, as they were a really good support for me and helped me in facing all the problems I had for completing this project.

Most importantly, I would like to thank all my family, friends and especially my husband who was a good support for me in the entire Master Program and he was the first one who encourages me to go forward and take this step, in which without his support I wouldn’t have done it.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

LOW POWER ASIC DESIGN, A COMPARITIVE STUDY

By

Silvia Said

Master of Science in Electrical Engineering

In the earlier generations the main focus was on the timings and area constraints. Power consumption was considered significant when there was a drastic increase of device densities from 130nm on. As technology advanced from 130nm to 90nm and below there was a significant increase in leakage current due to lesser threshold voltage. High power consumption will cause different problems such as increasing the cost of the product, reducing the reliability, reducing the battery life and many others. Therefore EDA tools were designed to maximize the speed while minimizing area.

The main objective of this project is to successfully complete a comparative study of an ASIC design flow using 90 nm SYNOPSYS Design Compiler (DC) to generate the net list of the physical design and then use SYNOPSYS IC Compiler to perform the placement and optimization followed by clock tree synthesis, routing and lastly chip design. This project gives an overview of different types of ASIC, front end and back end design using SYNOPSYS Design Compiler and IC compiler flow. In this project FIFO design flow as is considered as an example will be done through the entire flow two times. Firstly, without applying any power techniques optimization through the front end as well as in the back end and secondly, by applying the low power techniques that can be implemented within the SYNOPSYS license that we have in school in front end as well as in back end also. First synthesis is done in front end using SYNOPSYS Design Compiler and net list is generated. Then the physical implementation of the design is done using SYNOPSYS IC Compiler. Power optimization using RTL level optimization and setting up power constraints is mainly considered.

Chapter 1: Introduction

1.1 What is an ASIC?

Integrated Circuits are made from silicon wafer, with each wafer holding hundreds of die.

An ASIC is an Application Specific Integrated Circuit. An Integrated Circuit designed is called an ASIC if we design the ASIC for the specific application. Examples of ASIC include, chip designed for a satellite, chip designed for a car, chip designed as an interface between memory and CPU etc. Examples of IC's which are not called ASIC include Memories, Microprocessors etc.

1.2 Types of ASICs:

Application specific integrated circuits are categorized according to the technology used for manufacturing them. These types are full-custom ASICs and semi-custom and semi-custom can be further classified as standard cell based ICs (CBICs), Gate Array (GA) type.

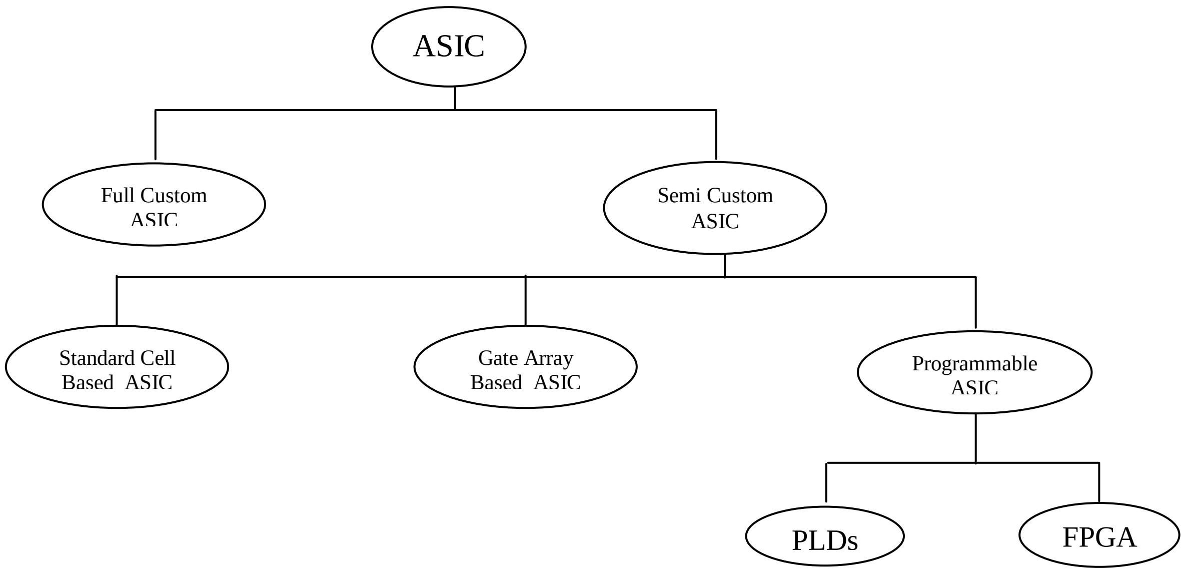


Fig 1.1: Types of ASICs chart

1.2.1 Full Custom ASICs:

All mask layers are customized and unique in a full-custom ASIC.

It only makes sense to design a full-custom IC if there are no libraries available.

Full-custom offers the highest performance and great for optimizing data paths and lowest part cost (smallest die size) with the disadvantages of increased design time, complexity, design expense, and highest risk and density.

Microprocessors were exclusively full-custom, but designers are increasingly turning to semicustom ASIC techniques in this area too.

Other examples of full-custom ICs or ASICs are requirements for high-voltage automobile), analog/digital (communications), or sensors and actuators.

Early in the 80's, a typical full custom ASIC would cost $10 million.

The disadvantages of full custom ASIC that makes it loses customers are:

1. Performance was still unmatched but often a gate array could give enough performance.
2. Tradeoff was a 2x reduction in performance for a 10x reduction in cost.
3. Full custom ASICs are just too difficult to build.
4. Take a lot of man hours.
5. Most products won't be sold enough to make up for all that development cost.

1.2.2. Semi-Custom ASICs:

Semi-custom ASIC's, can be partly customized to serve different functions within its general area of application, unlike full-custom ASIC's. Semi-custom ASIC's is designed to allow a certain degree of modification during the manufacturing process. A semi-custom ASIC is manufactured with the masks for the diffused layers already fully defined, so the transistors and other active components of the circuit are already fixed for that semi-custom ASIC design. The customization of the final ASIC product to the intended application is done by varying the masks of the interconnection layers, e.g., the metallization layers.

1.2.2.1 Standard Cell ASICs:

A standard cell-based ASIC uses predesigned logic cells (AND gates, OR gates, multiplexers, and flip-flops, for example) known as standard cells. Designer could apply the term CBIC to any IC that uses cells, but it is generally accepted that a cell-based ASIC or CBIC means a standard-cell–based ASIC. The standard-cell areas (also called flexible blocks) in a CBIC are built of rows of standard cells like a wall built of bricks. The standard-cell areas may be used in combination with larger predesigned cells, perhaps microcontrollers or even microprocessors, known as mega cells. Mega cells are also called mega functions, full-custom blocks, system-level macros (SLMs), fixed blocks, cores, or Functional Standard Blocks (FSBs).

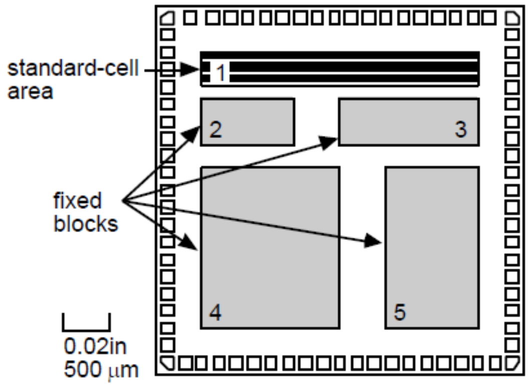


Fig. 1.2: Standard Cell ASICs

1.2.2.2 Gate Array ASICs:

Gate array ASICs is partially finished 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.

Gate array drawbacks are low density and performance than full custom or standard cell ASICs.

Die size is inflexible, which is one of the disadvantages of GA. For example, if a design requires 5001 transistors, the designer should upgrade his chip to an extra step up in the size.
NRE's are still substantial, it is much lower for a gate array than for a full custom, also fabrication time is modest.

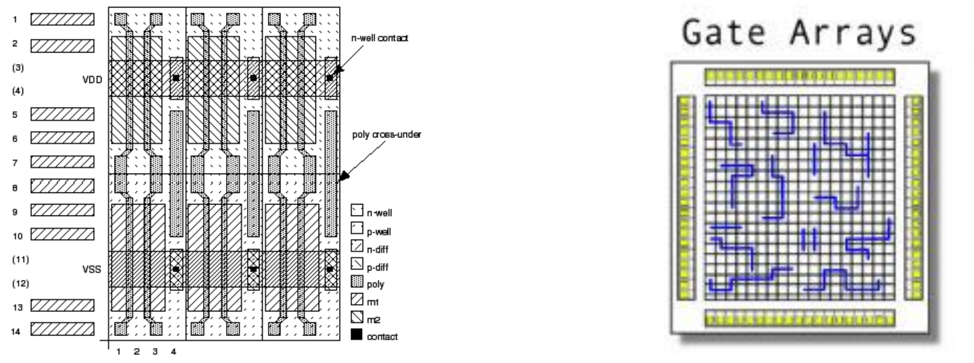


Fig. 1.3: Gate Array ASIC structure

1.2.3. Programmable ASICs:

Programmable logic devices (PLDs) are standard ICs that are available in standard configurations from a catalog of parts and are sold in very high volume to many different customers. However, PLDs may be configured or programmed to create a part customized to a specific application, and so they also belong to the family of ASICs. PLDs use different technologies to allow programming of the device.
All PLDs are having these features in common:

- No customized mask layers or logic cells
- Fast design turnaround
- A single large block of programmable interconnect
- A matrix of logic macro cells that usually consist of programmable array logic followed by a flip-flop or latch.

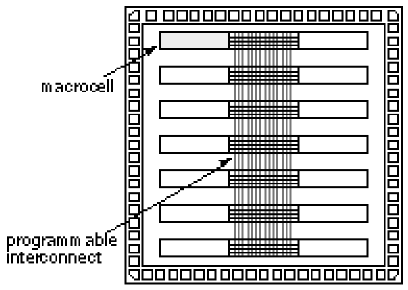


Fig. 1.4: Programmable ASICs

FPGA (Field Programmable Gate Array) are first developed in the mid of 80's by Xilinx, they are a step upper than PLD in complexity and density of the design.
FPGAs are composed of logic blocks instead of unwired transistors.
The core is a regular array of programmable basic logic cells that can implement combinational as well as sequential logic (flip-flops).

FPGAs have the following features:

- Less dense than custom mask.
- Higher unit cost.
- No NRE at all
- Ready to be used as soon as you get it.
- Reprogrammable.
- Performance can never catch up to mask-programmed devices.
- Programmable I/O cells surround the core.

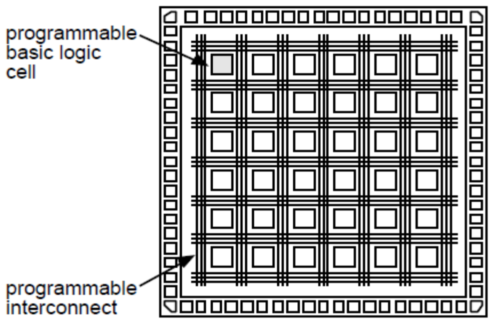


Fig. 1.5: FPGA Architecture

1.3 Design Flow of ASICs:

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 (what will the chip 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.** It means that at this point one has to decide what kind of architecture you would like to use for the design, e.g. RISC/CISC, ALU, pipelining etc…. .

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

The sub systems, top level systems once defined, need to be implemented. It is implemented using logic representation (Boolean Expressions), finite state machines,
Combinatorial, Sequential Logic, Schematics etc.... This step is called Logic Design

After figuring out the previous goals and specifications, then it's time to start the practical implementation of the design.

Broadly ASIC design flow can be divided into following sections:

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

RTL Description:

RTL stands for Register Transfer Level. RTL description of a design describes the design in terms of registers and logic that resides between them. This captures the timing constraints of the design as well as to be seen in behavioral description of the design. RTL is used in hardware description languages (HDLs) like Verilog and VHDL which are the most popular languages that are used to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Design at the RTL level is typical practice in modern digital design.

Functional Simulation/Verification:

Here the RTL description is tested for functional correctness of the design. Simulation involves event and cycle based simulation. While assertion based verification can also be used where formal methods are used to verify the functional correctness of the design.

Synthesis:

A design description that is functionally correct is fed to a Synthesis tool which takes an RTL hardware description and a standard cell library as input and produces a gate-level netlist as output. Standard cell library is the basic building block for today's IC design. Constraints such as timing, area, testability, and power are considered. Synthesis tools try to meet constraints, by calculating the cost of various implementations. It then tries to generate the best gate level implementation for a given set of constraints, target process. The resulting gate-level netlist is a completely structural description with only standard cells at the leaves of the design.

Design Verification:

Formal verification methods are used to test the functional correctness of gate-level netlist. Testing functional correctness involves testing an optimized design against a golden design description.

Layout:

This phase involves the back end of ASIC design flow which is implemented using IC Compiler by SYNOPSYS which includes: floor planning, Placement of cells on the chip area, placement of

Input/Output pads on the chip area. Clock tree synthesis is performed in order to minimize space and power consumed by clock signal. Placement and routing is carried out on this design. So far the interconnect delays and parasitic values are based on wire-load model. Now resistance, capacitance and inductance (latest feature) is calculated for a placed and routed netlist. Design rule violations are identified and corrected. Static timing analysis is carried out to find timing violations.

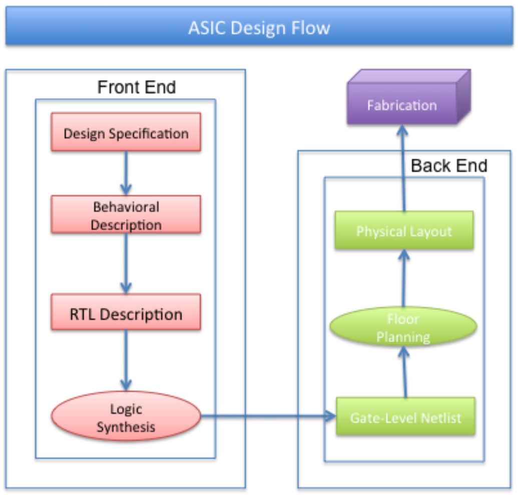


Fig 1.6(a): ASIC design flow.

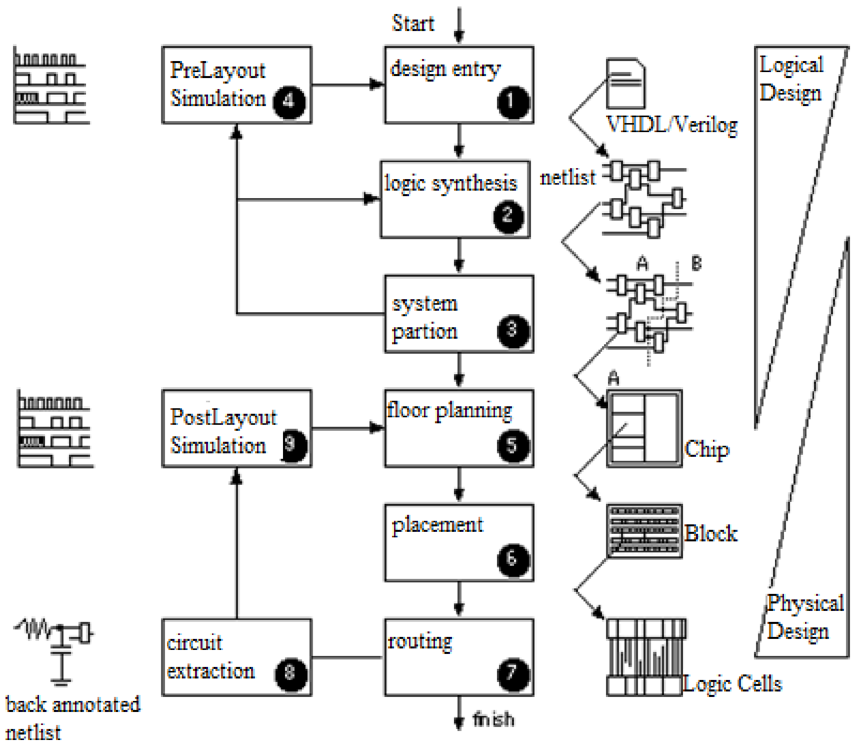


Fig 1.6(b): ASIC design flow.

Chapter 2: Project Design Overview

2.1 Introduction to Asynchronous Interface:

Asynchronous interface is a 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.

Asynchronous design has been an active area of research since at least the mid 1950's, but has yet to achieve widespread use.

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.

This paper discusses one FIFO design style and important details that must be considered when doing asynchronous FIFO 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.

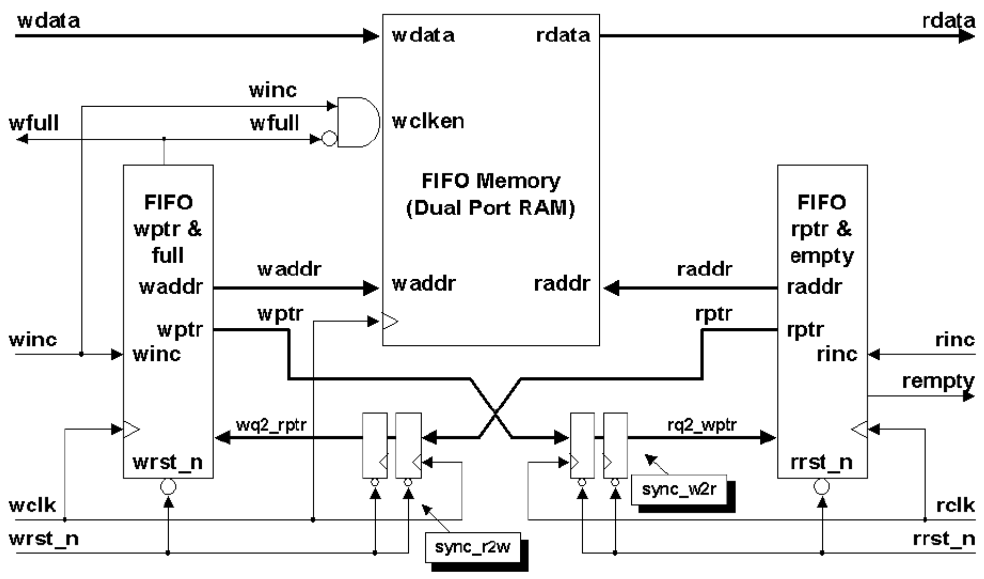


Fig. 2.1: FIFO Block diagram

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, therefore FIFO needs to buffer the data that need to be stored during the write cycle, until it has been read in the read clock domain. The depth of the FIFO depends on the data to be stored.

2.2 Problems in implementing Asynchronous FIFOs:

The main problems in Asynchronous FIFO are as follows:

- Crossing the clock domains.
- Avoid Metastability.
- Generating the FIFO pointers because there might be multiple bits change per clock edge.
- Need to avoid overflow / underflow.
- Finding a reliable way to determine full and empty status flag on the FIFO.

2.2.1: Crossing Clock Domains:

It is very important in designing FIFO is to understand the main problems that will face you while designing asynchronous FIFO. In designing Asynchronous FIFO, the most important issue that should be resolved is to adjust crossing clock domains.

Since one clock domain might be faster than the other, therefore we need to buffer the data words properly so we don’t lose any data words or duplicate them, or let it go to a metastable state.

2.2.2 Avoid Metastability:

If the signal is not synchronized to the new clock, the first storage element of the new clock domain may go to metastable state and the worst case is that the resolution time cannot be predicted. It can traverse throughout the new clock domain resulting in failure of functionality. To prevent such failures setup 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.

2.2.3 Generating FIFO pointers to avoid multi-bit changes per clock pulse:

We have to understand how the FIFO pointers work. When reset is asserted, both read/write pointers are set to zero, FIFO empty flag is asserted and the read pointer points to invalid data.

When write operation starts, after writing the first data, the FIFO empty flag is de-asserted and the write pointer increments, read pointer is still addressing the contents of the first FIFO memory word, immediately drives that first valid word onto the FIFO data output port, to be read by the receiver logic.

The main problem in the write/read pointers are the multi bit transition in binary counters, which will give false data. For example, when counter goes from 7 to 8, the 4-bits changes (three ones to zero and one zero to one), which will be harmful due to multi bit changes per clock pulse, which will give false data.

In order to solve this problem, gray code will be used as it is based on 1 transition per clock edge.

In this table below, therefore if we take look on gray code below, by converting binary to gray, multi bit transition problem is solved.

Decimal	Binary	Gray
0	0000	0000
1	0001	0001
….	….	……
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
……	…..	…..
14	1110	1001
15	1111	1000

Table 2.1: Binary and Gray counters

2.2.4 Generating full/empty status flag:

The FIFO is empty when the read and write pointers are both equal. This condition happens when both pointers are reset to zero during a reset operation, or when the read pointer catches up to the write pointer, having read the last word from the FIFO.

A FIFO is full when the pointers are again equal, that is, when the write pointer has wrapped around and caught up to the read pointer. This is a problem. The FIFO is either empty or full when the pointers are equal, but which?

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. 2.2 (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.

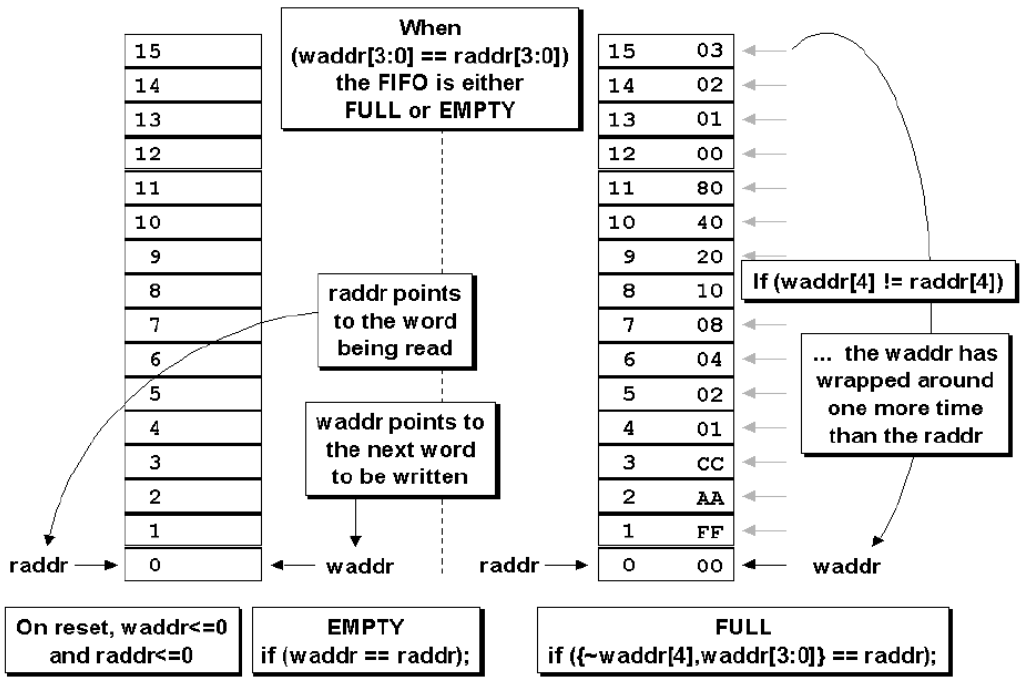

When (waddr[3:0] == raddr[3:0]) the FIFO is either FULL or EMPTY
raddr points to the word being read
waddr points to the next word to be written
raddr
waddr
raddr
waddr
If (waddr[4] != raddr[4])
... the waddr has wrapped around one more time than the raddr
On reset, waddr<=0 and raddr<=0
EMPTY if (waddr == raddr);
FULL if ({~waddr[4],waddr[3:0]} == raddr);

Fig. 2.2: FIFO full and empty conditions

2.2.5 Avoid Overflow/Underflow:

The binary counter values are sampled periodically and not all of the binary counter values can be passed to a new clock domain. The question is do we need to be concerned about the case where a binary counter might continue to increment and overflow or underflow the FIFO between sampled counter values? The answer is no.

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. **Overflow will not occur.**

FIFO empty occurs when the read pointer catches up to the synchronized and sampled write pointer. The synchronized and sampled write pointer might not reflect the current value of the actual write pointer but the read pointer will not try to count beyond the synchronized write pointer value. **Underflow will not occur.**

2.3 Operation of the Design (Asynchronous FIFO):

Asynchronous FIFO is a design where data words are written to a RAM through an input port using write clock frequency and then read the data word through an output port from the RAM using read clock frequency.

2.3.1 Write Operation:

Initially after resetting the FIFO, read and write pointers are set to zero location, and empty flag is asserted (**rempty**), which indicates that the FIFO is empty. When the first data word is being written in the RAM, the write pointer increments and locates the next location, also at the same time the empty flag is de-asserted. When empty flag is cleared and write pointer moves to the next location, read pointer points to the location where the first word has been written in and automatically using the read logic circuit to read the data word through the output port of the RAM and increment the read pointer to the next location.

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.

2.3.2 Asynchronous FIFO pointers:

After resetting, write and read pointers are set to zero and empty status flag is asserted. After n times of write operation, with each write clock cycle, the pointer increments and points to the next location where a new data word will be written in, also with each read clock cycle, the read pointer increments to the next location in the RAM to read the data word. When write pointer reaches the top and read pointer is somehow in the middle, the full flag is not asserted although the write pointer reaches the top of the RAM because a read operation has been done during the writing operation.

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 status flag.

When FIFO pointer wrap to the first location after reaching the top of the RAM by writing in all the RAM location addresses, FIFO write pointer wrap to the first location where no read operation is performed and so they will be both equal which will cause the empty status flag to be asserted falsely as in the case after resetting the pointers.

In order to avoid the flags to be set falsely, an extra bit will be added to the pointer width, in which (n-1) bits will be the size of the address bits, and MSB will be used for identifying which flag is to be asserted.

Therefore, in case of resetting the FIFO pointers, both the read and write pointer MSB are equal and so empty flag is asserted.

When write pointer reaches the top of the FIFO and then wrap to go the first location, therefore the write pointer will be the same in (n-1) bits while MSB will be different from the read pointer and so the full flag will be asserted.

Gray Counter is the best counter to be used as it is based on a fact that the code distance between any two adjacent words is just 1 (only one bit can change from one Gray count to another). The second fact to remember about a Gray code counter is that most useful Gray code counters must have power-of-2 counts in the sequence. It is possible to make a Gray code counter that counts an even number of sequences but conversions to and from these sequences are generally not as simple to do as the standard Gray code. Also note that there is no odd-count-length Gray code sequences so one cannot make a 23-deep Gray code. This means that the technique described in this paper is used to make a FIFO that is 2n deep.

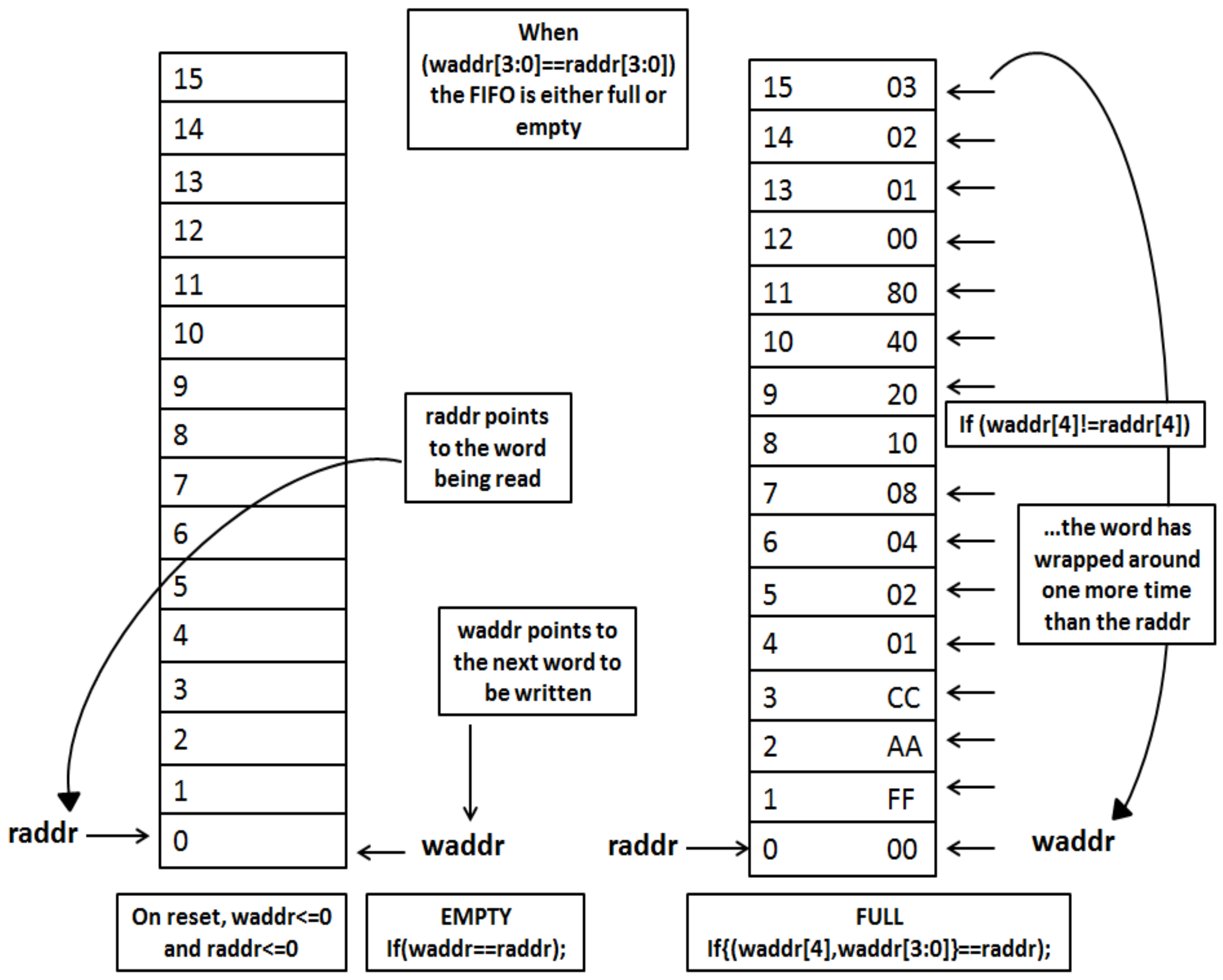

When (waddr[3:0]==raddr[3:0]) the FIFO is either full or empty
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
raddr points to the word being read
waddr points to the next word to be written
raddr
waddr
15 03
14 02
13 01
12 00
11 80
10 40
9 20
8 10
7 08
6 04
5 02
4 01
3 CC
2 AA
1 FF
0 00
If (waddr[4]!=raddr[4])
...the word has wrapped around one more time than the raddr
raddr
waddr
On reset, waddr<=0 and raddr<=0
EMPTY If(waddr==raddr);
FULL If{(waddr[4],waddr[3:0]}==raddr);

Fig. 2.3: Empty and Full flag generation

The best Gray code to be used in FIFO pointer counters is:

1- **Dual n-bit Gray code counter-style #1:**

The dual n-bit gray code counter that generates both an n-bit Gray code sequence and an (n-1)-bit Gray code sequence. The (n-1)-bit Gray code is simply generated by doing an exclusive-or operation on the two MSBs of the n-bit Gray code to generate the MSB for the (n-1)-bit Gray code. This is combined with the (n-2) LSBs of the n-bit Gray code counter to form the (n-1)-bit Gray code counter.

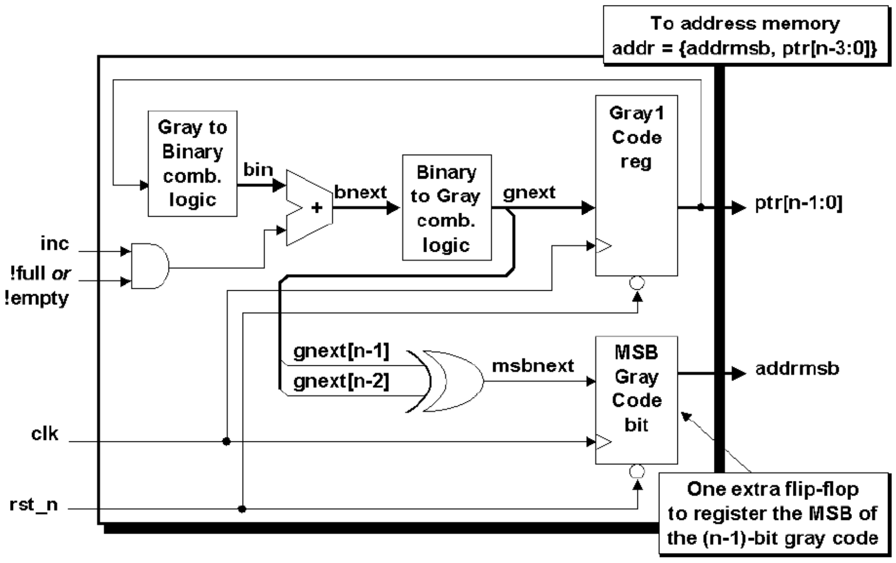

To address memory
addr = {addrmsb, ptr[n-3:0]}
Gray to Binary comb. logic
bin
bnext
Binary to Gray comb. logic
gnext
Gray1 Code reg
ptr[n-1:0]
inc
!full or !empty
gnext[n-1]
gnext[n-2]
msbnext
MSB Gray Code bit
addrmsb
clk
rst_n
One extra flip-flop to register the MSB of the (n-1)-bit gray code

Fig. 2.4: Gray code counter-Style#1

2- Dual n-bit Gray code counter-style #2:

Which actually employs two sets of registers to eliminate the need to translate Gray pointer values to binary values? The second set of registers (the binary registers) can also be used to address the FIFO memory directly without the need to translate memory addresses into Gray codes. The n-bit Gray-code pointer is still required to synchronize the pointers into the opposite clock domains, but the n-1-bit binary pointers can be used to address memory directly.

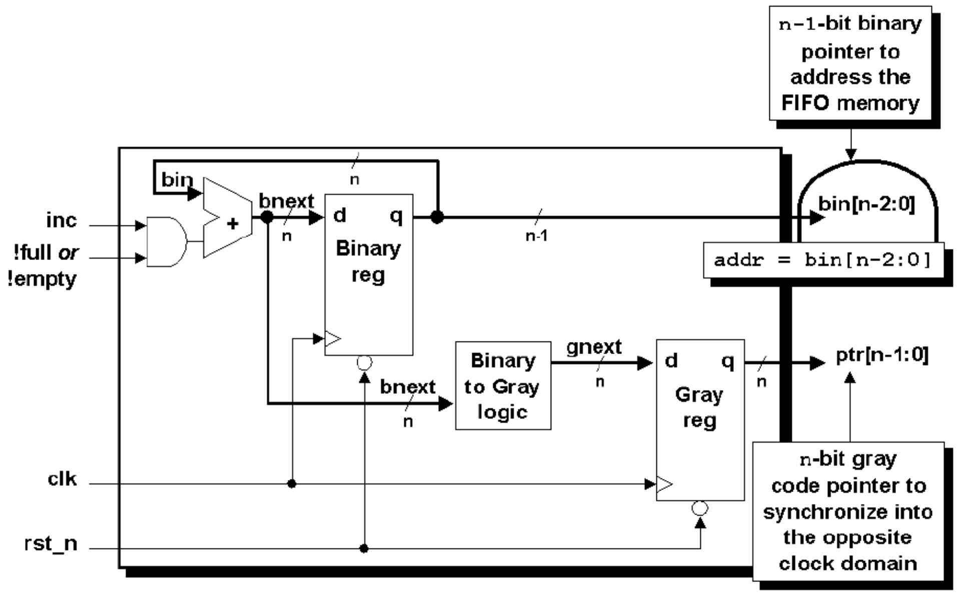


Fig. 2.5: Gray Code Counter-style#2

2.3.3 Generating Empty flag:

Initially FIFO pointers are reset to zero locations where write and read pointers are equal including MSB, which lead to assertion of the empty flag.

After n times of write operation, FIFO write pointer will reach the top of the FIFO and then wrap down to zero location, if no read operation is performed then empty flag won’t be asserted because according to the gray code counter used in designing the FIFO pointers, an extra bit is added to both pointers. Therefore, in this case the read pointer will be all zeroes including MSB while the write pointer will be of equal (n-1) bits to read pointer but differs in MSB due to wrap more than read pointer, and so the empty flag won’t be asserted falsely as discussed previously due to the extra bit added to the pointers.

Therefore, Empty flag is asserted if and only if read and write pointers are equal including MSB.

2.3.4 Generating Full flag:

When resetting the FIFO pointers, both the read and write pointers are set to zero location, where the empty flag is asserted due to the equalization of both the read and write pointers.

After n times of write operation where no read operation is done, the write pointer reaches the top of the FIFO and wrap to zero location. In this case, write pointer MSB will not be equal to read pointer MSB, but (n-1) bits of read and write pointers are equal. Therefore, Full flag is asserted indicating full FIFO memory as no read operation has been performed during the write operation.

If write operation has been performed for n times and during this operation read operation has been performed. Therefore, when write operation reaches the top of the FIFO, the full flag won’t be asserted because MSB of both read and write should be unequal in addition to (n-1) bits of both pointers should be equal.

2.4 Modules of the design:

2.4.1 FIFO.v:

It is the top level module of asynchronous FIFO. The top module is only used as a wrapper to instantiate all of the other FIFO modules used in the design. Also, the top level module in this design includes the synchronizer module that is used to synchronize the read pointer into the write-clock domain and vice versa.

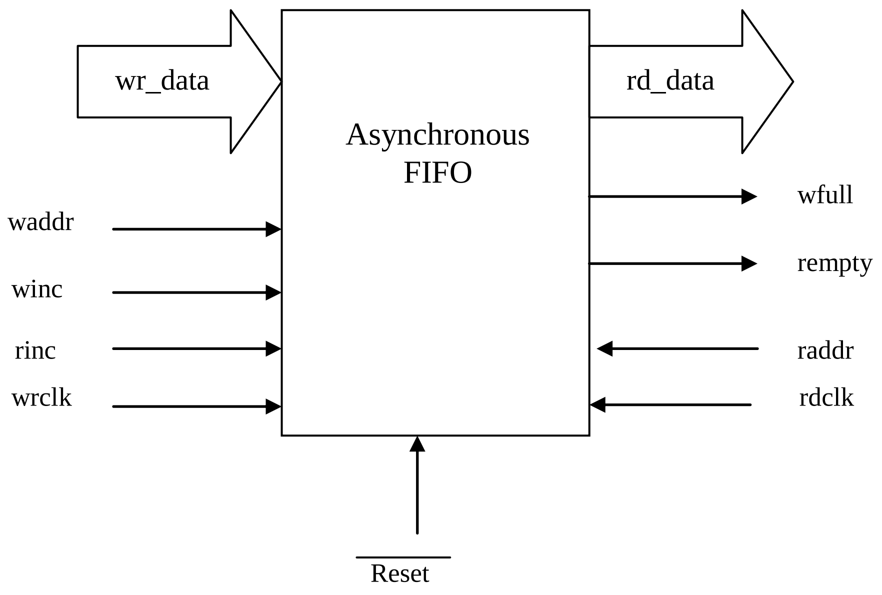


Fig. 2.6: Top Module of Asynchronous FIFO

2.4.2 mem_blk.v: This is the FIFO memory buffer that is accessed by both the write and read clock domains. This buffer is most likely an instantiated, synchronous dual-port RAM. Other memory styles can be adapted to function as the FIFO buffer.

2.4.3 rptr_ctrl.v: This module includes the gray counter conversion logic that will makes the read pointer works properly and efficiently, also this module include the generation of the Empty flag.

2.4.4 wptr_ctrl.v: This module includes the gray counter conversion logic that will makes the write pointer works properly and efficiently, also this module include the generation of the Full flag.

Chapter3: Introduction to Design Compiler

3.1 Introduction:

The Design Compiler tool is the core of the Synopsys synthesis products. Design Compiler optimizes designs to provide the smallest and fastest logical representation of a given function. It comprises tools that synthesize your HDL designs into optimized technology-dependent, gate-level designs. It supports a wide range of flat and hierarchical design styles and can optimize both combinational and sequential designs for speed, area, and power.

You use Design Compiler for logic synthesis, which is the process of converting a design description written in a hardware description language such as Verilog or VHDL into an optimized gate-level netlist mapped to a specific technology library.

The resulting gate-level netlist is a completely structural description with only standard cells at the leaves of the design. Internally, a synthesis tool performs many steps including high-level RTL optimizations, RTL to un-optimized Boolean logic, technology independent optimizations, and finally technology mapping to the available standard cells. A synthesis tool is only as good as the standard cells which it has at its disposal. Good RTL designers will familiarize themselves with the target standard cell library so that they can develop a solid intuition on how their RTL will be synthesized into gates.

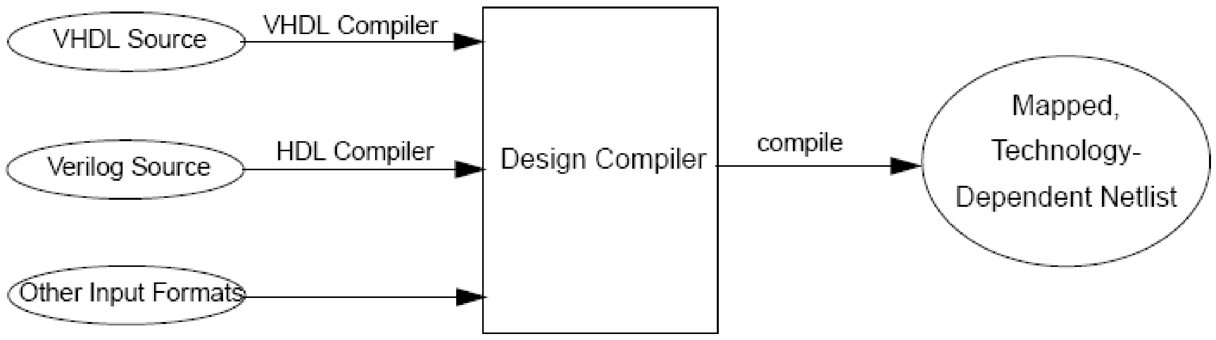


Fig. 3.1: Design Compiler flow

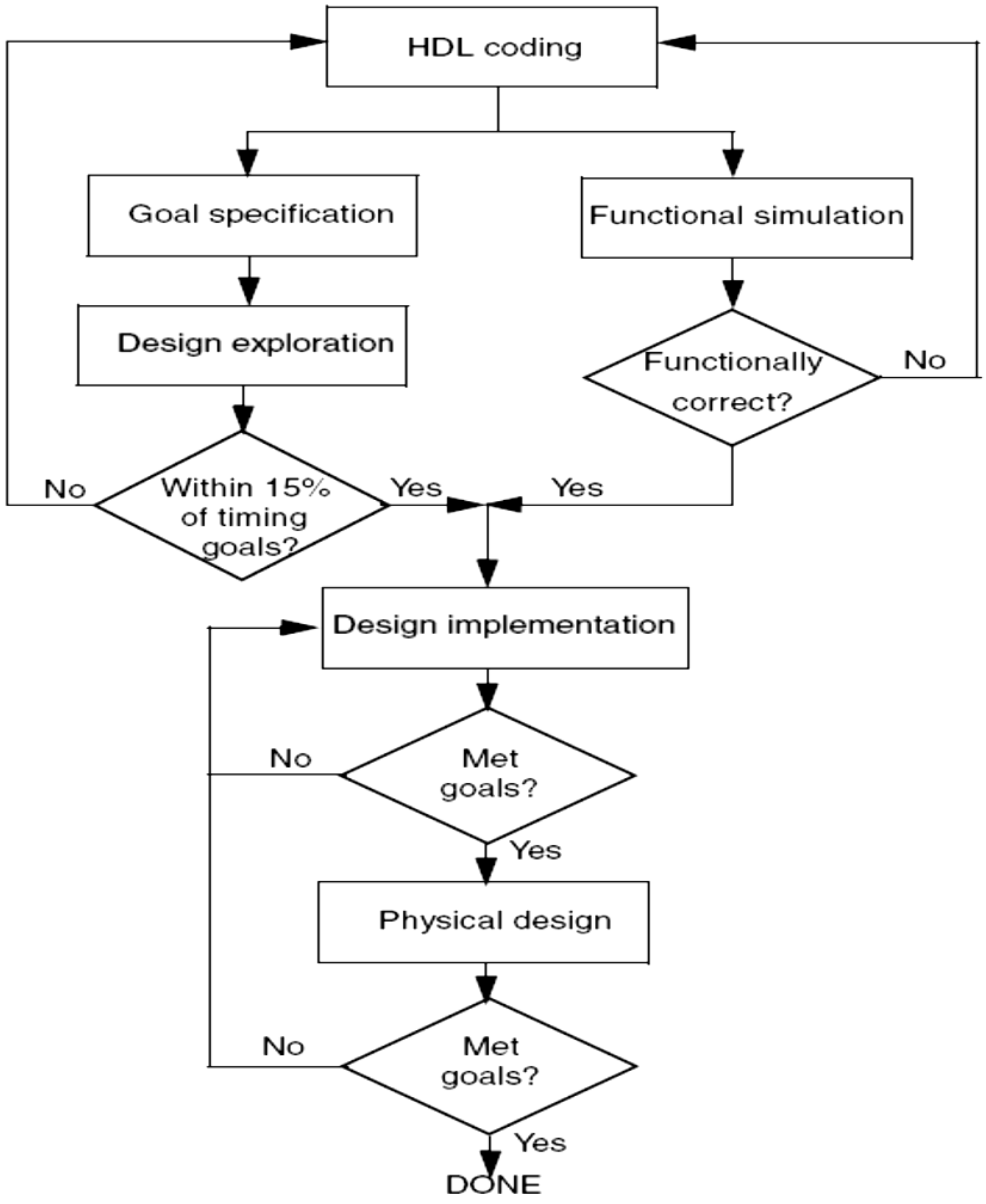


Fig. 3.2 Design flow chart

3.2 Design flow:

According to the figure above, there are many steps to be done through the design flow to get the design ready for synthesis tool (Design compiler):

1. Write the design to be implemented in a Hardware description language (HDL) such as VHDL or Verilog HDL.

2. Perform Functional simulation/verification for the code to make sure it works properly as it is designed for.

   If the design does not function as required, then the designer must modify the HDL code and repeat the functional simulation/verification.

   Continue performing the functional simulation/verification until the design is functioning correctly.

3.3 Design Compiler flow:

The steps in the synthesis process are as follows:

1- Writing always the input design files for Design Compiler using a hardware description language (HDL) such as Verilog or VHDL.

2- Perform design exploration, in which specific design goals are implemented to the design (Design Rules and Optimization Constraints), and carryout a preliminary synthesis (default synthesis) using Design Compiler.

   If design exploration fails to meet timing goals by more than 15 percent, modify the design goals and constraints implemented, or improve the HDL code. Then repeat the design exploration again.

3- Design Compiler uses technology libraries, synthetic or Design Ware libraries, and symbol libraries to implement synthesis and to display synthesis results graphically.

   During the synthesis process, Design Compiler translates the HDL description to components extracted from the generic technology (GTECH) library and Design Ware library. The GTECH library consists of basic logic gates and flip-flops. The Design Ware library contains more complex cells such as adders and comparators. Both the GTECH and Design Ware libraries are technology independent, that is, they are not mapped to a specific technology library. Design Compiler uses the symbol library to generate the design schematic.

4- After translating the HDL description to gates, Design Compiler optimizes and maps the design to a specific technology library, known as the target library. The process is constraint driven. Constraints are the designer's specification of timing and environmental restrictions under which synthesis is to be performed. Verify at this stage that all the design rules and the timing constraints have been met before proceed to the synthesis part, and if not repeat the constraints differently until it meets the goal for the design and then proceed to the next step.

5- After the design is optimized, it is ready for test synthesis. Test synthesis is the process by which designers can integrate test logic into a design during logic synthesis. Test synthesis enables designers to ensure that a design is testable and resolve any test issues early in the design cycle. The result of the logic synthesis process is an optimized gate-level netlist, which is a list of circuit elements and their interconnections. Verify that the design meets the goals. If the design does not meet goals, generate and analyze various reports to determine the techniques which might use to correct the problems.

6- After test synthesis, the design is ready for the place and route tools, which place and interconnect cells in the design. Based on the physical routing, the designer can back-annotate the design with actual interconnect delays; Design Compiler can then re-synthesize the design for more accurate timing analysis.

3.4 Design Compiler Synthesis detailed 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.

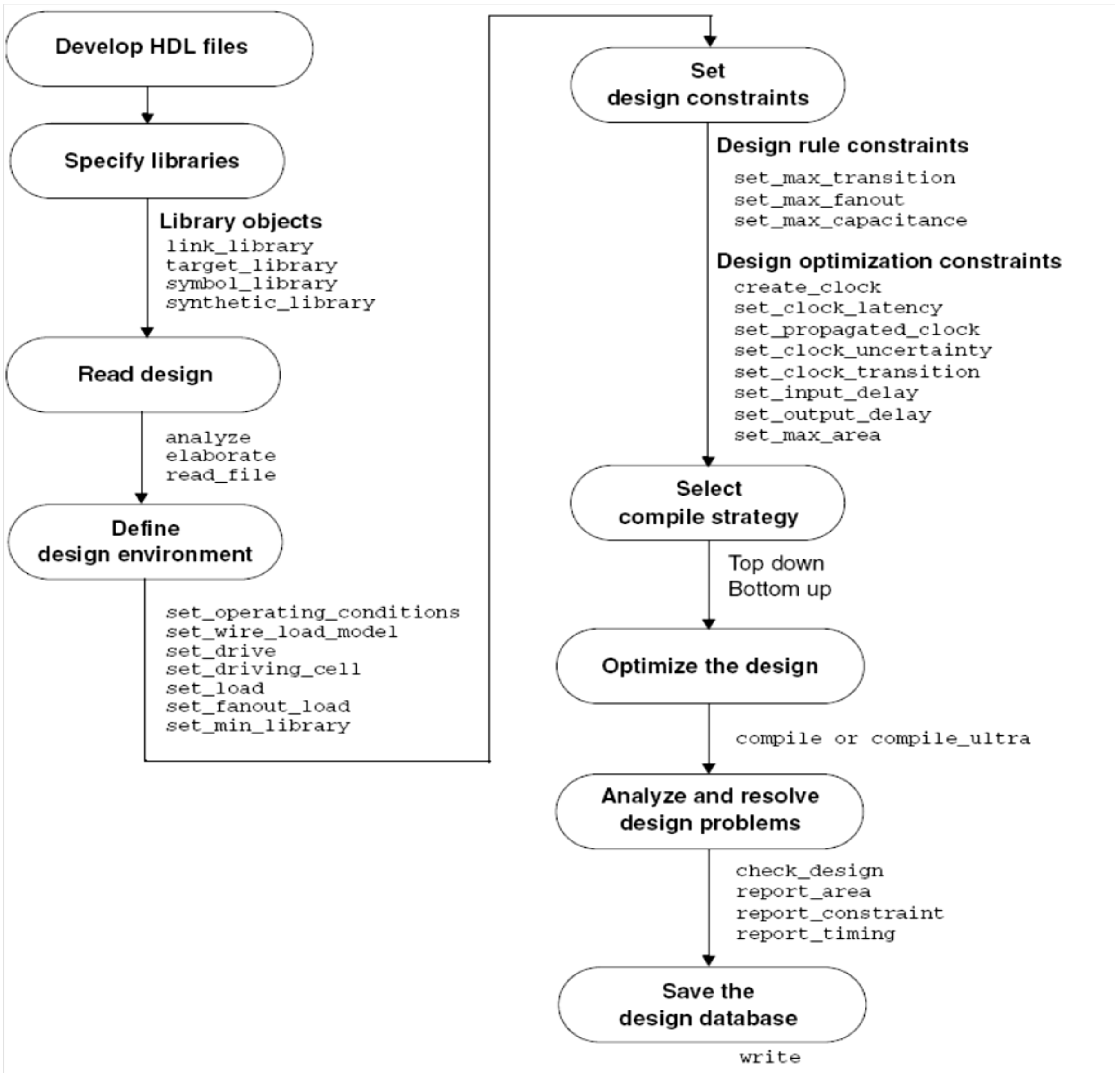


Fig. 3.3: Design Compiler Synthesis Flow

3.4.1 Develop HDL Files:

The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL. These design descriptions need to be written carefully to achieve the best synthesis results possible. When writing HDL code, design data management, design partitioning, and HDL coding style must be considered. Partitioning and coding style directly affect the synthesis and optimization processes.

There are few techniques in partitioning and coding style that helps the optimization and synthesis part for the design:

3.4.1.1 Group related combinational logic and its destination register together:

When working with the complete combinational path, Design Compiler has the flexibility to merge logic, resulting in a smaller, faster design. Grouping combinational logic with its destination register also simplifies the timing constraints and enables sequential optimization.

3.4.1.1.1 Eliminate glue logic:

Glue logic is the combinational logic that connects blocks. Moving this logic into one of the blocks improves synthesis results by providing Design Compiler with additional flexibility. Eliminating glue logic also reduces compile time, because Design Compiler has fewer logic levels to optimize.

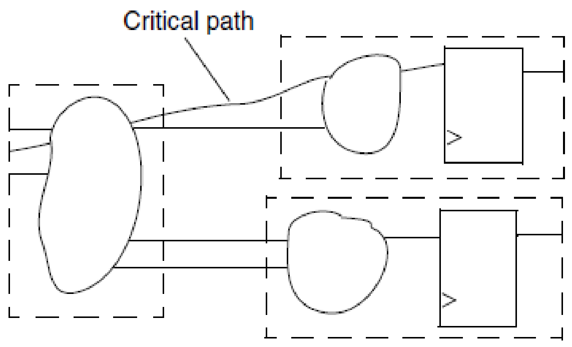


Fig. 3.4: Poor partitioning of related logic

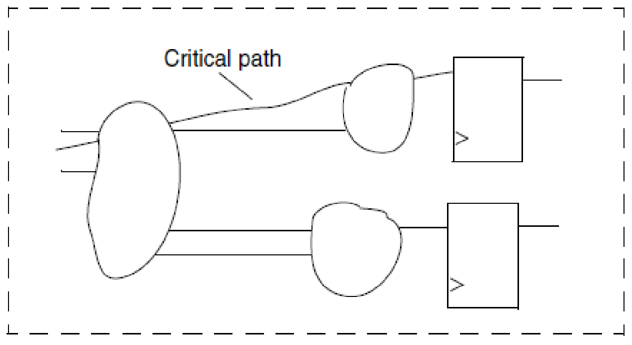


Fig. 3.5: Good partitioning techniques

3.4.1.1.2 Registering Block Outputs:

This method enables you to constrain each block easily because

- The drive strength on the inputs to an individual block always equals the drive strength of the average input drive.
- The input delays from the previous block always equal the path delay through the flip-flop.

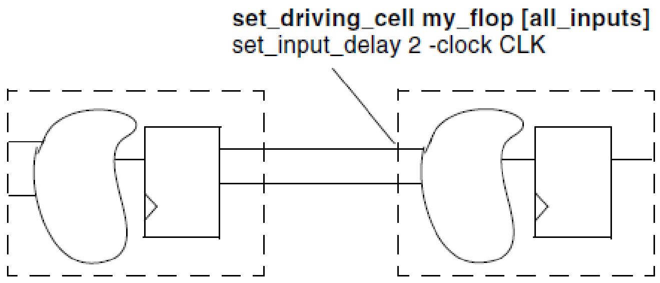

set_driving_cell my_flop [all_inputs]
set_input_delay 2 -clock CLK

Fig. 3.6: Registering Block outputs

This partitioning method can improve simulation performance. With all outputs registered, a module can be described with only edge-triggered processes. The sensitivity list contains only the clock and, perhaps, a reset pin. A limited sensitivity list speeds simulation by having the process triggered only once in each clock cycle.

3.4.1.2 Partition by design goals:

The goal is to partition logic with different design goals into separate blocks. Use this method when certain parts of a design are more area and timing critical than other parts.

To achieve the best synthesis results, isolate the noncritical speed constraint logic from the critical speed constraint logic. By isolating the noncritical logic, you can apply different constraints, such as a maximum area constraint, on the block.

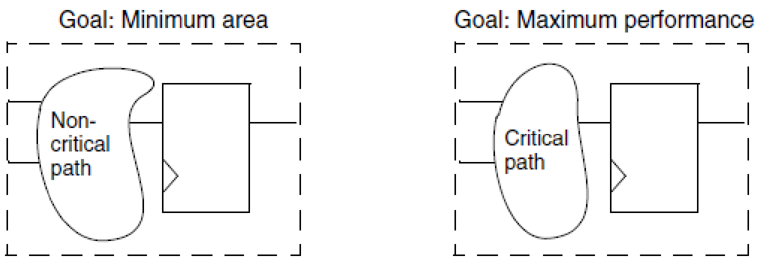


Fig. 3.7: Blocks with different Constraints

3.4.1.3 Keeping Sharable Resources together:

Design Compiler can share large resources, such as adders or multipliers, but resource sharing can occur only if the resources belong to the same VHDL process or Verilog always block.

For example, if two separate adders have the same destination path and have multiplexed outputs to that path, keep the adders in **one VHDL process or Verilog always block**. This approach allows Design Compiler to share resources (using one adder instead of two) if the constraints allow sharing.

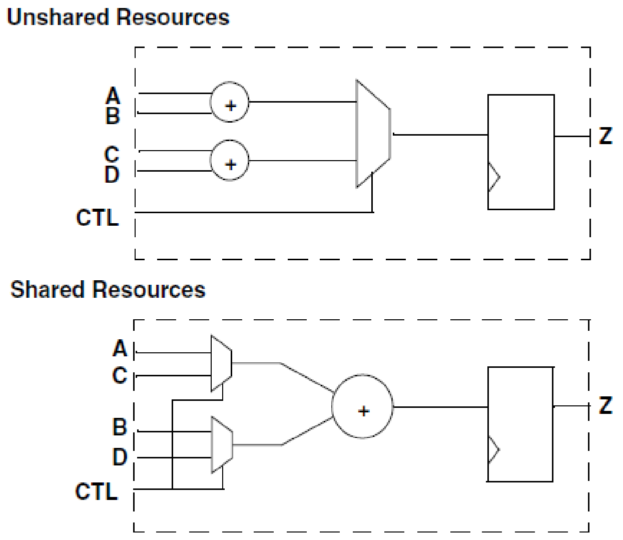


Fig. 3.8: Keeping Sharable Resources in one block

3.4.1.4 Keeping User-Defined Resources With the Logic They Drive:

User-defined resources are user-defined functions, procedures, or macro cells, or user-created DesignWare components. Design Compiler cannot automatically share or create multiple instances of user-defined resources. Keeping these resources with the logic they drive, however, gives you the flexibility to split the load by manually inserting multiple instantiations of a user-defined resource if timing goals cannot be achieved with a single instantiation.

Fig. 3.9 illustrates splitting the load by multiple instantiation when the load on the signal PARITY_ERR is too heavy to meet constraints.

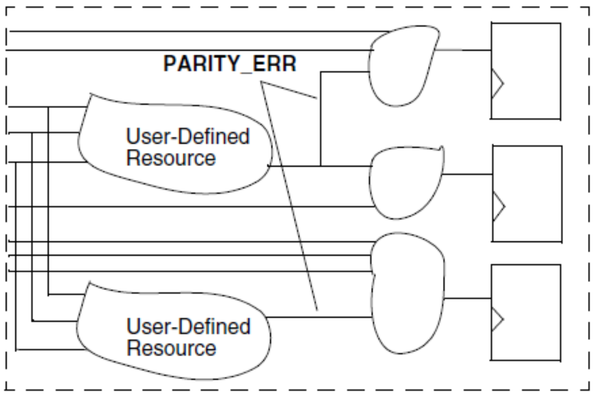


Fig. 3.9: Duplicate user-defined resources

3.4.1.5 Isolating Special Functions:

Isolate special functions (such as I/O pads, clock generation circuitry, boundary-scan logic, and asynchronous logic) from the core logic. It is the recommended partitioning technique for the top level of the design.

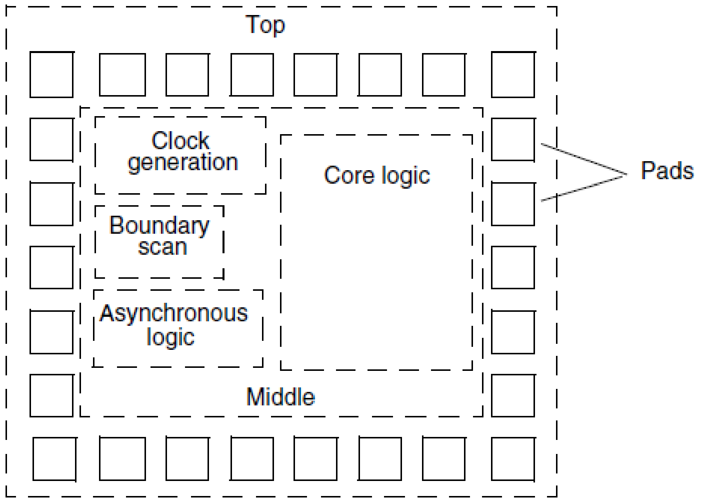


Fig. 3.10: Recommended top level partitioning technique

The top level of the design contains the I/O pad ring and a middle level of hierarchy that contains sub-modules for the boundary-scan logic, the clock generation circuitry, the asynchronous logic, and the core logic. The middle level of hierarchy exists to allow the flexibility to instantiate I/O pads. Isolation of the clock generation circuitry enables instantiation and careful simulation of this module. Isolation of the asynchronous logic helps confine testability problems and static timing analysis problems to a small area.

3.4.2 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.

3.4.2.1 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.

3.4.2.2 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.

3.4.2.3 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.

3.4.2.4 Symbol Library:

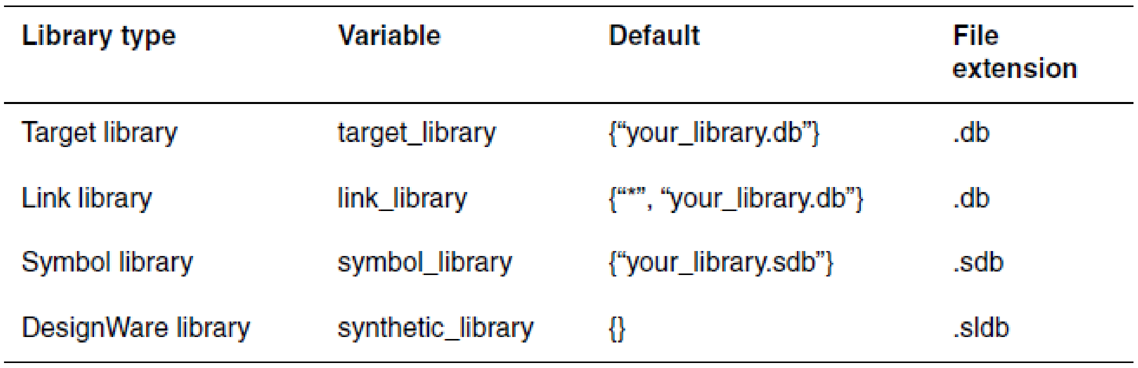
It is the library that contains all the definitions of the graphic symbols that represent library cells in the design schematics, when you generate the design schematic, Design Compiler performs a one-to-one mapping of cells in the netlist to cells in the symbol library.

3.4.2.5 Synthetic Library:

The user doesn't need to specify the standard synthetic library (standard.sldb), which implements the built-in HDL operators. These operators include +, -, *, <, >, <=, >=, and the operations defined by if and case statements.

The Design Compiler software automatically uses this library without even loading in the setup file (.synopsys_dc.setup).

If you are using additional DesignWare libraries, you must specify these libraries by using the *synthetic_library* variable (for optimization purposes) and the *link_library* variable (for cell resolution purposes).



Library type	Variable	Default	File extension
Target library	target_library	{“your_library.db”}	.db
Link library	link_library	{“*”, “your_library.db”}	.db
Symbol library	symbol_library	{“your_library.sdb”}	.sdb
DesignWare library	synthetic_library	{}	.sldb

Fig. 3.11: Library Variables

All of the different libraries that we talked about previously should be located in a special format in a file under a specific name (.synopsys_dc.setup) which must be located in the working directory of the user so it can be invoked by the tool.

Here is a sample of the (.synopsys_dc.setup) file:

search_path = “. /synopsys/libraries/syn/cell_library/libraries/syn”
Set target_library [list of standard cell library files **for mapping**]
Set synthetic_library [list of sldb files for designware, and so on]
Set *additional_link_lib_files* [list of additional libraries for linking: pads, macros, and so on]
Set link_library [list * $target_library $additional_link_lib_files \ $synthetic_library]

3.4.3 Read Design:

After setting the startup file for Design compiler and specifying all the required libraries that needed in your design.

After invoking the tool using command **dc_shell-xg-t**

Now it’s the time for reading the design which is having to hierarchies to be done:

- Top Bottom
- Bottom Up.

In our case study due to the small size of the design, top-bottom will be beneficial than bottom-up.

Due to the Top-bottom hierarchy strategy, all the sub modules will be read first and then the top level.

This particular step is done through two ways:

3.4.3.1 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

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.

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.

N.B.: Analyze operates on filename while Elaborate operates on module name.

For example, enter

dc_shell> **analyze -format vhdl *RISC_CORE.vhd***

dc_shell> **elaborate *RISC_CORE***

3.4.3.2 Read_file:

Executing *read_file* command does the following:

- Reads several different formats (.ddc, .vhd, .v, .db).
- Performs the same operations as analyze and elaborate in a single step
- Creates .mr and .st intermediate files for VHDL
- Does not execute the link command automatically.

- Does not create any intermediate files for Verilog (However, you can have the read_file command create intermediate files by setting the hdlin_auto_save_templates variable to true.

N.B.: Previously compiled designs (netlists) must be read in with the read command.

For example, enter

dc_shell> **read_file -format *verilog RISC_CORE*.*v***

3.4.4. Define design environment:

After reading the design files, and before doing any optimization. The operating environment where the design is expected to operate in should be defined.

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 fanout loads. The environment model directly affects design synthesis results. In Design Compiler, the model is defined by a set of attributes and constraints that you assign to the design, using specific dc_shell commands.

- ***set_operating_conditions***

Most technology libraries used has different and predefined operating conditions. We have to report the library using command *report_lib* to list the operating conditions defined in the library. The library should be loaded first in memory before running the command *report_lib*.

In order to see the list of libraries loaded in memory, use command *list_libs.*

Operating condition describes the Voltage, Temperature, Interconnect model and process of the design.

Each operating condition predefined with its specific temperature, voltage, interconnect model.

There are mostly common in most technology libraries WORST, BEST and TYPICAL but the names are library dependent. Users should ask the vendor which is the best operating condition to be used.

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.

dc_shell> **set_operating_conditions *BEST***

In the following example, it illustrates that the design use the BEST case values for minimum delay analysis and uses the WORST case values for maximum delay analysis.

dc_shell> **set_operating_conditions –min *BEST* –max *WORST***

- ***set_wire_load_model***

Wire load modeling allows the user to estimate the effect of wire length and fanout on the resistance, capacitance, and area of nets. Design Compiler uses these physical values to calculate wire delays and circuit speeds.
The vendors usually develop wire load model, depending on some statistical information. The models include values for area, capacitance, and resistance per unit length.

dc_shell> **set_wire_load_model –name <*model_name*>**

Design Compiler supports three modes for determining which wire load model to use for nets that cross hierarchical boundaries:

- TOP

This wire load model is used as if there is no hierarchy, in which the compiler models all the nets in the top level or in the sub-designs using the wire load model specified for the top level model.

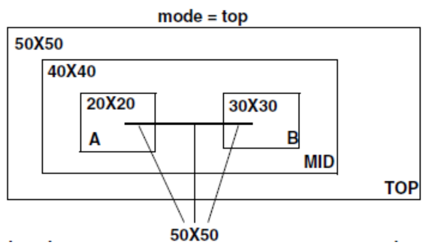


Fig. 3.12: Top wire load model

- ENCLOSED

Design Compiler uses the wire load model of the smallest design in the hierarchy. If the design enclosing the net has no wire load model specified, the tool keep tracing in upward direction until it finds a wire load model. ENCLOSED wire load model is more accurate than top model when cells in the same design are placed near to each other during layout.

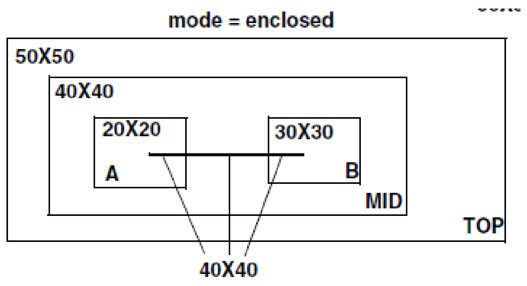


Fig. 3.13: Enclosed wire load model

- SEGMENTED

Design Compiler determines the wire load model of each segment of a net by the design encompassing the segment. Nets crossing boundaries are divided into segments.
For each net segment, Design Compiler uses the wire load model of the design containing the segment.
If the design has a net that has no wire load model specified, the tool keep tracing in upward direction until it finds a wire load model.

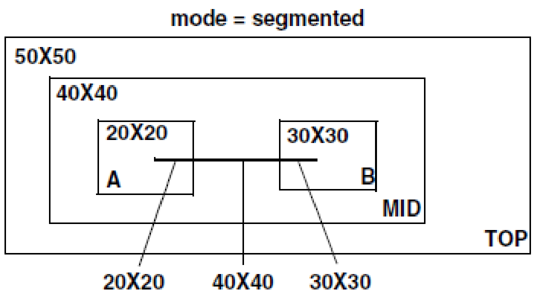


Fig. 3.14: Segmented wire load model

- ***set_drive***

This command uses to specify resistance values on specified Input and InOut ports in the current design.

They are used to set the drive resistance of ports in the top level module only.

dc_shell > **set_drive <*num*> <*port*>**

dc_shell > **set_drive *1.5 {I1 I2}***

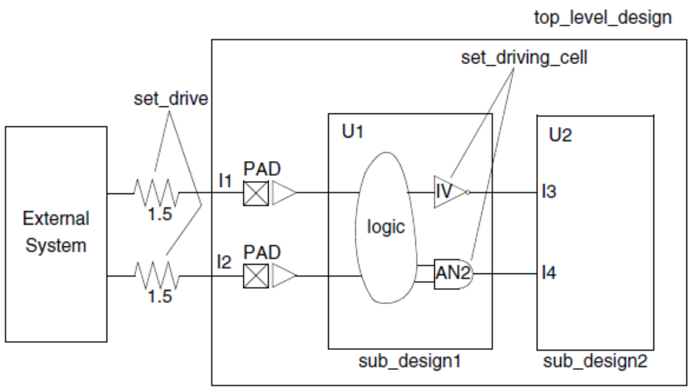


Fig. 3.15: Driving Characteristics

- ***set_driving_cell***

This command sets attributes on the specified Input or InOut ports in the current design to associate an external driving cell with the ports. The drive capability of the port is the same as if the specified driving cell were connected in the same context to allow accurate modeling of the port drive capability for nonlinear delay models.

dc_shell > **set_driving_cell -lib_cell *IV {I3}***

dc_shell> **set_driving_cell -lib_cell *AN2* -pin *Z* -from_pin *B {I4}***

- ***set_load***

Design Compiler by default assumes that all ports have zero capacitive loads.
This command is used to set the value of the capacitive load on Input or Output ports in the design.

```
dc_shell> set_load 30 {out1}
```

- ***set_fanout_load***

Design Compiler tries to ensure that the sum of the fanout load on the output port plus the fanout load of cells connected to the output port driver is less than the maximum fanout limit of the library, library cell, and design.
Fanout load is unit less and not the same as load. Fanout represents a numerical contribution of the total fanout load, while load represents a capacitive value for the ports.

```
dc_shell> set_fanout_load 4 {out1}
```

3.4.5 Set Design Constraints:

When Design Compiler comes to optimize the design, it uses two types of constraints:
Design rule constraints: These are implicit constraints; the technology library defines them. These constraints are requirements for a design to function correctly, and they apply to any design using the library. You can make these constraints more restrictive than optimization constraints.
Optimization constraints: These are explicit constraints; you define them. Optimization constraints apply to the design on which you are working for meeting the design’s goals. They must be realistic.
Design Compiler tries to meet both design rule constraints and optimization constraints, but design rule constraints take precedence.

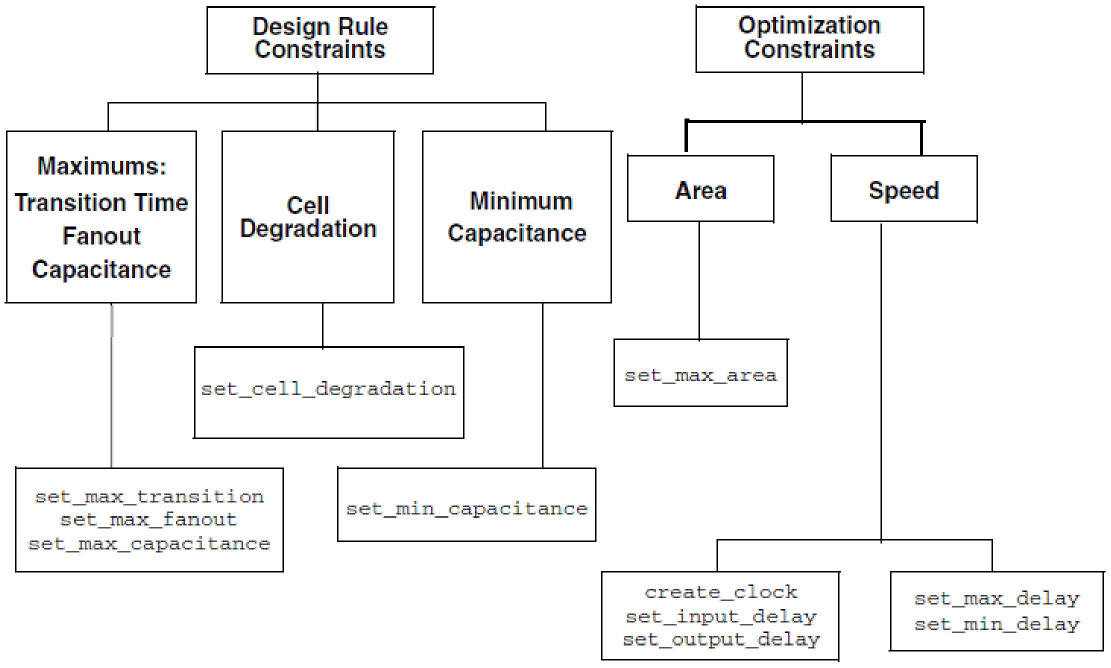


Fig. 3.16: Design Compiler Constraints

3.4.5.1. 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 fanout
- 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.

- ***set_max_transition***

This command is used to set the longest time required for a driving pin to change its logic value.

- ***set_max_fanout***

This command sets the maximum allowable fanout load for the listed input ports.
Most technology libraries place fanout restrictions on driving pins, creating an implicit fanout constraint for every driving pin in designs using that library.

- ***set_max_capacitance***

This command is used to define the maximum total capacitive load that an output pin can drive. In which the pin cannot connect to a net that has a total capacitance greater than or equal to the maximum capacitance defined at the pin.

3.4.5.2 Design Optimization Constraints:

These types of constraints are being set to meet the design goals or requirements in terms of area, power and speed. It is recommended that designers specify realistic constraints, since unrealistic specification results in excess area, increased power and/or degradation in timing.
The optimization constraints include

- Timing constraints (performance and speed)
    - Input and output delays (synchronous paths)
    - Minimum and maximum delay (asynchronous paths)
- Maximum area (number of gates)

There are a set of commands that can be used to set the optimization constraints in order to meet the user goals:

- ***create_clock***

This command is used to define the clock that will be used in the design. This command doesn't imply that the synthesizer will create a circuit to implement the waveform. It tells the synthesizer that the specified periodic waveform will be input at a clock port and it needs to make a circuit that will work with it.

dc_shell> **create_clock** ***-period -waveform***

-period *specifies the period of the clock waveform in library time units.*

-waveform *specifies the rise and fall edge times, in library time units, of the clock over an entire clock period.*

- ***set_clock_latency***

This is used to specify that there will be a delay of unit time delays through buffers that are not modeled in the design. It is very useful in hierarchical designs, when constraining sub-modules.

This is primarily used during the pre-layout 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).

dc_shell> **set_clock_latency** ***<time unit> [get_clocks <port name>]***

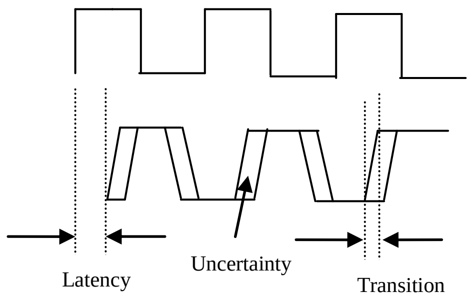


Fig. 3.17: Clock waveform constraints

- ***set_clock_uncertainty***

This command is used for setting a margin for setup and hold time of the clock. When clock arrives simultaneously to every register, the arrival time won't be the same at every register. There will be skew. During the pre-layout phase one can add more time margin as compared to the post-layout phase.

This is to specify the worst case sum of clock skew and jitter.

dc_shell> **set_clock_uncertainty** ***<time_unit> [get_clocks <port name>]***

This is to specify setup and hold uncertainty parameter:

dc_shell> **set_clock_uncertainty *–setup <time_unit> [get_clocks <port name>]***

dc_shell> **set_clock_uncertainty *–hold <time_unit> [get_clocks <port name>]***

Setup is always a problem in worst case condition, while hold is a problem in best case condition.

- ***set_input_delay***

This command is used to specify the arrival of the signal coming from outside that will not arrive at the beginning of the clock period. The compiler should know this.
It is used at the input ports to specify the time it takes for the data to be stable after the clock edge. It is the delay provided at the input port by the external logic.

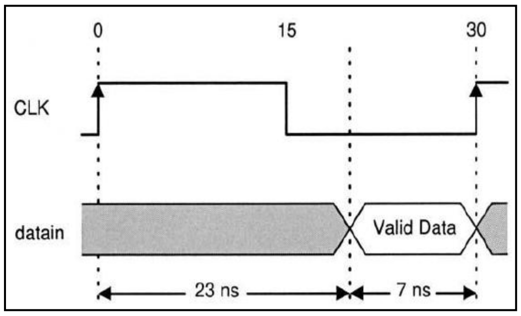


Fig. 3.18: Input delay Waveform specification

Input delay = FF delay + external Combinational delay

dc_shell> **set_input_delay –max *23* –clock *CLK [get_ports datain]***

- ***set_output_delay***

This command is used to specify the time taken by the signal to be available before the clock edge. It is used at the output ports to specify the time it takes for the data to be stable before the clock edge. It is the delay at the output port that has been caused by the external logic delay.
Maximum Output delay = Setup time of FF + max. external logic delay
Minimum Output delay = min. external logic delay – hold time of FF

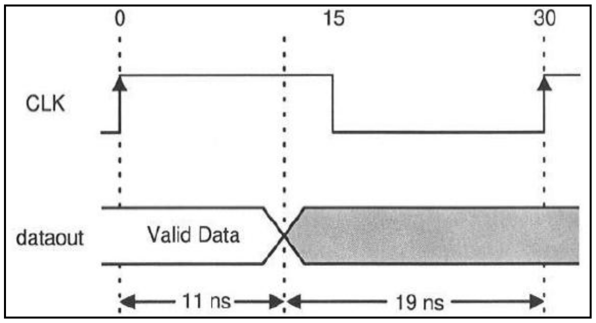


Fig. 3.19: Output delay waveform specification

dc_shell> **set_output_delay -max** ***19*** **–clock** ***CLK [get_ports dataout]***

- ***set_false_path***

Most timing arcs are between registers within a module. These arcs are mostly within a synchronous system (same clock at both registers).
False path command is mostly used across clock domains. 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 start point of the path is identified by [-from] which may include clocks, ports, or pins. If the start point is not specified this may cause the tool to disable all the paths to the [–to port_list]
The end point of the path is identified by [-to] which may include clocks, ports, pins or cells. If the end point is not specified, this may cause the tool to disable all the paths that starts with the specified starting point.

dc_shell> **set_false_path –from <*port*> -to <*port*>**

- ***set_max_delay***

This command 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***

This command 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.

- ***set_max_area***

This command specifies the maximum allowable area for the current design. Design Compiler computes the area of a design by adding the areas of each component on the lowest level of the design hierarchy (and the area of the nets). Maximum area represents the number of gates in the design, not the physical area the design occupies.

3.4.6. Select Compile Strategy:

Synopsys recommends the following compilation strategies that depend entirely on how your design is structured and defined. It is up to user discretion to choose the most suitable compilation strategy for a design.

3.4.6.1 Top-Down Hierarchical compile method:

Top-Down hierarchical compile method was generally used to synthesize very small designs (less than 10K gates). Using this method, the source is compiled by reading the entire design. Based on the design specifications, the constraints and attributes are applied, only at the top level. Although, this method provides an easy push-button approach to synthesis, it was extremely memory intensive and viable only for very small designs.

The advantages and disadvantages are summarized below:

- **Advantages:**

a. Only top level constraints are needed.
b. Better results due to optimization across entire design.

- **Disadvantages:**

a. Long compile times.
b. Incremental changes to the sub-blocks require complete re-synthesis.

c. Does not perform well, if design contains multiple clocks or generated clocks.

3.4.6.2 Bottom-Up Hierarchical compile method:

The designer manually specifies the timing requirements for each block of the design, thereby producing multiple synthesis scripts for individual blocks. The synthesis is usually performed bottom-up i.e., starting at the lowest level and ascending to the topmost level of the design. This method targets medium to very large designs and does not require large amounts of memory.

The advantages and disadvantages are listed below:

- **Advantages:**

a. Easier to manage the design because of individual scripts.
b. Incremental changes to the sub-blocks do not require complete re-synthesis of the entire design.
c. Does not suffer from design style e.g., multiple and generated clocks are easily managed.
d. Good quality results in general because of flexibility in targeting and optimizing individual blocks.

- **Disadvantages:**

a. Tedious to update and maintain multiple scripts.
b. Critical paths seen at the top-level may not be critical at lower level.
c. The design may need to be incrementally compiled in order to fix the DRC’s.

3.4.7. 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

3.4.7.1 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).

3.4.7.2 Logic-Level Optimization:

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

- 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.

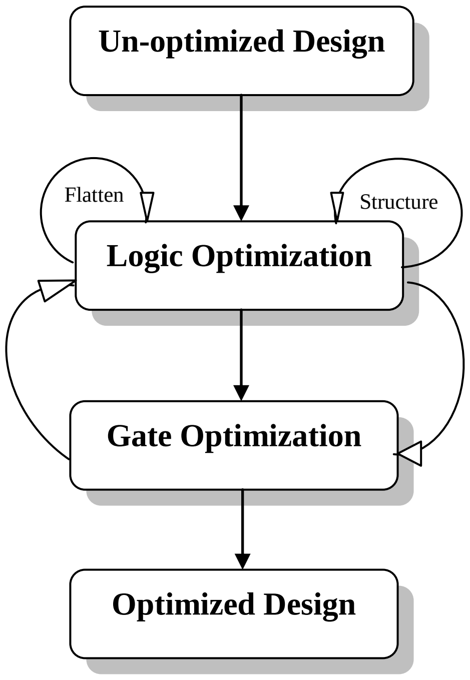


Fig. 3.20: Optimization Flow

3.4.7.3 Gate Level Optimization:

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

- Mapping
- Delay Optimization
- Design Rule Fixing
- Area Optimization

➢ **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.

- **Area Optimization:**

The process goal is to meet area constraints after the mapping, delay optimization, and design rule fixing phases are completed.

3.4.8. 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.

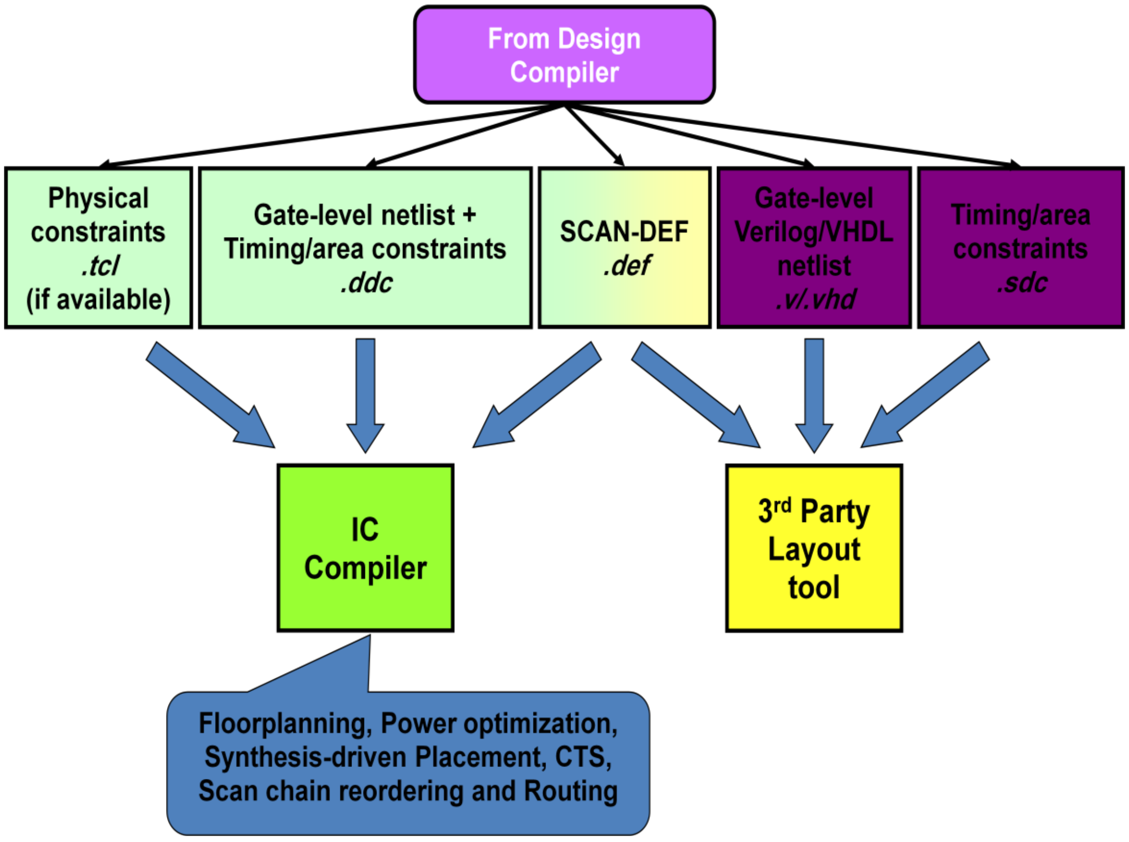

From Design Compiler
Physical constraints .tcl (if available)
Gate-level netlist + Timing/area constraints .ddc
SCAN-DEF .def
Gate-level Verilog/VHDL netlist .v/.vhd
Timing/area constraints .sdc
IC Compiler
3rd Party Layout tool
Floorplanning, Power optimization, Synthesis-driven Placement, CTS, Scan chain reordering and Routing

Fig. 3.21: Design Compiler flow to IC Compiler

Chapter 4: Power Optimization Techniques

4.1 Introduction:

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. This means that expensive ceramic chip packaging must be used instead of plastic, and complex and expensive heat sinks and cooling systems are often required for product operation. Laptop computers and hand-held electronic devices can become uncomfortably hot to the touch. Higher operating temperatures also reduce reliability because of electro migration and other heat-related failure mechanisms.

High power consumption also reduces battery life in portable devices such as laptop computers, cell phones, and personal electronics. As more features are added to a product, power consumption increases and the battery life is reduced, requiring a larger, heavier battery or shorter life between charges. Battery technology has lagged behind the increased demands for power.

Another aspect of power consumption is the sheer cost of electrical energy used to power millions of computers, servers, and other electronic devices used on a large scale, both to run the devices themselves and to cool the machines and buildings in which they are used. Even a small reduction in power consumption of a microprocessor or other device used on a large scale can result in large aggregate cost savings to users and can provide significant benefits to the environment as well.

4.2 Power Consumption:

CMOS technology has two main factors for power consumption

$P_{total} = P_{dynamic} + P_{static}$

In order to know how to minimize the power consumption, we have to learn first the various causes of power consumption.

4.2.1 Static (Leakage) Power:

It is the power dissipated when the gate is not working or switching between logic values (inactive or static). Static power is dissipated in several ways.

There are two main sources of leakage current in which static power is dissipated:

4.2.1.1 Sub-threshold current (**I_{sub}**): The current which flows from the drain to the source current of a transistor operating in the weak inversion region, in other words when threshold voltage is reduced that lead to prevent the gate from completely turning off.

4.2.1.2 Gate Leakage (I_{gate}): The current which flows directly from the gate through the oxide to the substrate due to gate oxide tunneling and hot carrier injection.

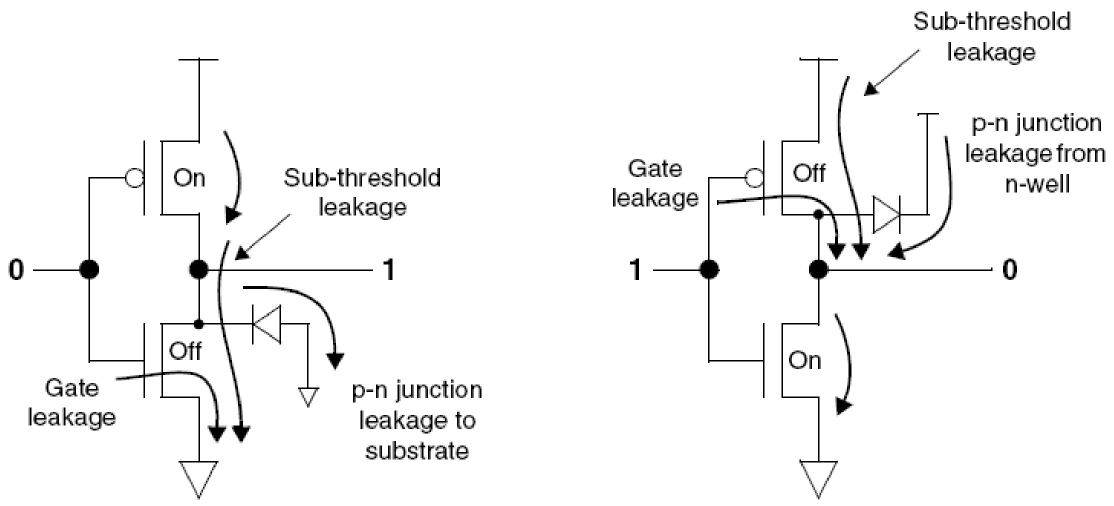


Fig. 4.1: Static Leakage Currents

Leakage currents occur whenever power is applied to the transistor, irrespective of the clock speed or switching activity. Leakage cannot be reduced by slowing or stopping the clock. However, it can be reduced or eliminated by lowering the supply voltage or by switching off the power to the transistors entirely.

4.2.2 Dynamic Power:

Dynamic power is the power consumed when the device is active. A circuit is active anytime the voltage on net changes due to some stimulus applied to the circuit. Because voltage on an input net can change without necessarily resulting in logic transition on the output, dynamic power can be dissipated even when an output net does not change its logic state. It consists of two components, switching power and internal power.

4.2.2.1 Switching Power: It is the power results from the charging and discharging of the external capacitive load on the output of a cell. In one complete cycle of CMOS logic, current flows from V_{DD} to the load capacitance to charge it and then flows from the charged load capacitance to ground during discharge.

Therefore in one complete charge/discharge cycle, a total of $Q=C_LV_{DD}$ is thus transferred from V_{DD} to ground. Multiply by the switching frequency on the load capacitances to get the current used, and multiply by voltage again to get the characteristic switching power dissipated by a CMOS device.

Switching Current ➔ $\mathbf{I = C_L.V.F}$

Power dissipated/transition ➔ $\mathbf{P = 1/2\ C_L.F.V^2}$ where C_L is the capacitive load.

4.2.2.2 Internal Power: It is the power resulted from short circuit current that flows through the P-N stack during a transition; both the transistors will be ON for a small period of time in which current will find a path directly from V_{DD} to ground, hence creating a short circuit current. Short circuit power dissipation increases with rise and fall time of the transistors.

A transition from 0 to 1 on the output of the inverter charges the capacitive load of the output net through the PMOS transistor. A transition from 1 to 0 discharges the same capacitive load through the NMOS transistor.

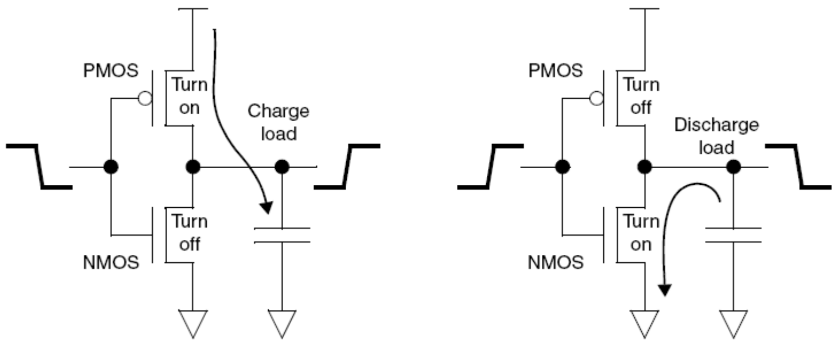


Fig 4.2: Dynamic Leakage Currents

An additional form of power consumption became significant in the 1990s as wires on chip became narrower and the long wires became more resistive. CMOS gates at the end of those resistive wires see slow input transitions. During the middle of these transitions, both the NMOS and PMOS logic networks are partially conductive and current flows directly from V_{dd} to V_{SS}. The power thus used is called crowbar power. Careful design which avoids weakly driven long skinny wires has ameliorated this effect, and crowbar power is nearly always substantially smaller than switching power.

To speed up designs, manufacturers have switched to constructions that have lower voltage thresholds and smaller rise/fall time.

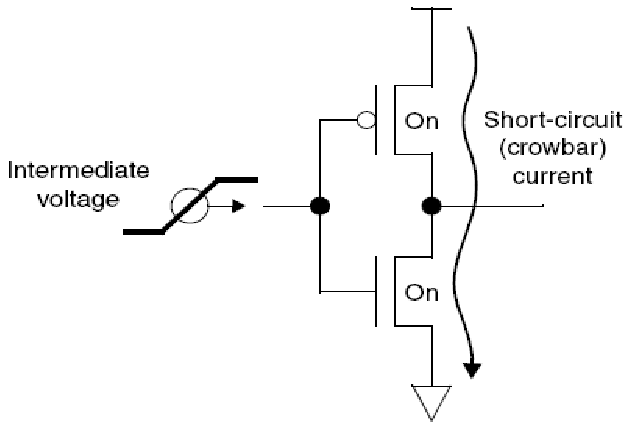


Fig 4.3: Internal Power (Short Circuit)

4.3 Power Optimization Methodology:

There are several different RTL and gate-level design strategies for reducing power. Some methods, such as clock gating, have been used widely and successfully for many years. Others, such as dynamic voltage and frequency scaling, have not been used much in the past due to the difficultly of implementing them. As power becomes increasingly important in advanced technologies, more methods are being exploited to achieve the design requirements.

4.3.1 Supply Voltage Reduction:

The most basic way to reduce power is to reduce the supply voltage. Power usage is proportional to the square of the supply voltage, as shown below.

A 50 percent reduction in the supply voltage results in a 50 percent reduction in current and a 75 percent reduction in power. A similar degree of power reduction can be achieved in CMOS circuits for both dynamic and static power.

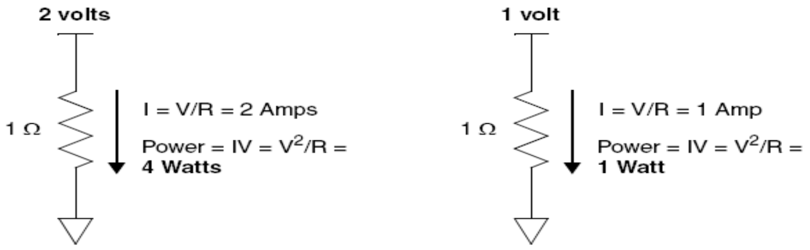


Fig. 4.4: Power versus Supply Voltage

Successive generations of CMOS technologies have used lower and lower supply voltages to reduce power. Starting at 5 volts for compatibility with bipolar TTL circuits in the 1980s, supply voltages have fallen to about 1 volt for advanced technologies. Each lowering of the supply voltage reduces per-gate power consumption, but also lowers the switching speed. In addition, the transistor threshold voltage must be lowered, causing more problems with noise immunity, crowbar currents, and sub-threshold leakage. The lower voltage swing makes it more difficult to interface the chip with external devices that require larger voltage swings.

4.3.2 Clock Gating:

Clock gating is an important high-level technique for reducing the power consumption of a design. Clock gating is a dynamic power reduction method in which the clock signals are stopped for selected register banks during times when the stored logic values are not changing. One possible implementation of clock gating is shown in **Fig. 4.5**

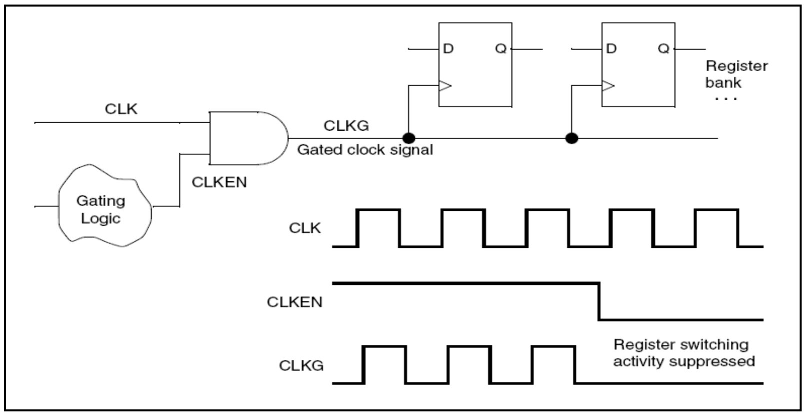


Fig. 4.5: Clock Gating Example.

Clock gating is particularly useful for registers that need to maintain the same logic values over many clock cycles. Shutting off the clocks eliminates unnecessary switching activity that would otherwise occur to reload the registers on each clock cycle. The main challenges of clock gating are finding the best places to use it and creating the logic to shut off and turn on the clock at the proper times.

Clock gating is a well-established power-saving technique that has been used for years. Synthesis tools such as Power Compiler can detect low-throughput data paths where clock gating can be used with the greatest benefit, and can automatically insert clock-gating cells in the clock paths at the appropriate locations. Clock gating is relatively simple to implement because it only requires a change in the netlist. No additional power supplies or power infrastructure changes are required.

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.

Also we can choose which clock gating insertion circuit is to be implemented from the following options:

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

In **Fig.4.6** 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.

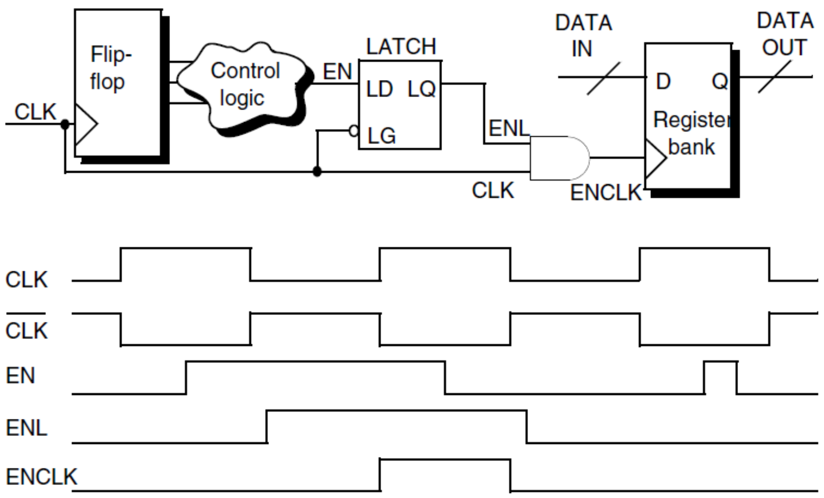


Fig. 4.6: Latch-based Clock Gating Style

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 glitches at 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.

Clock gating inserts clock-gating circuitry into the register bank's clock network, creating the control to eliminate unnecessary register activity.

4.3.2.1 Inserting Clock gating circuitry in RTL design:

For implementing low power techniques on a design, Clock Gating technique is added to the RTL design during the Front End of the ASIC design flow. 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. This can be achieved by following some steps:

1. Read the entire design.
2. Set the design environment.
3. Set all the timing and area constraints to be met by the design.
4. Set the power optimization true.

Power Compiler has to use *compile_ultra –gate_clock* option for compiling the design to insert clock-gating cells.

Before compiling the design some settings have to be done for implementing low power technique properly.

1. Designer has to set the *power_driven_clock_gating* constraint to true as the default is false.
2. Designer has to set the clock-gating style will be used by the tool.

set_clock_gating_style

This command sets the clock-gating style to be used for clock-gating with the *compile_ultra -gate_clock* command, as well as power-driven and gate-level clock gating enabled by specifying the -*gate_clock* option to *compile* or *compile_ultra* commands.

Also there are some options that can be set for more specification:

-sequential _cell   ***none | latch***

This option specifies the cell that is used to delay the enable signal gating the clock. To select a latch-based style, use latch. By default, the tool selects which latch cell to use.

-control_point   ***none | before | after***

This option specifies where to insert a control point in the clock gating circuitry. With before or after control styles, the tool implements an OR gate with the original register enable signal and a test control signal as inputs. If you *set_clock_gating_style* specify a *latch-based style,* the before or after control styles determine the location of the OR gate with respect to the latch. The tool creates a new input port to provide the test signal. The control points must be hooked up to the design level test_mode or scan_enable port using the insert_dft command.

-control_signal   ***scan_enable | test_mode***

This option specifies the test control signal. If an input port is created and the argument of the *control_signal* option is *scan_enable,* the name of the port is determined by the *test_scan_enable_port_naming_style* variable, and a *test_scan_enable signal_type* attribute is put on the new port. If an input port is created and the argument of the *-control_signal* option is *test_mode*, the name of the port is determined by the *test_mode_port_naming_style* variable, and a test_hold attribute is put on the new port.

-positive_edge_logic   cell_list | integrated

Specifies the circuitry used to gate the clock of a flip-flop that is inferred by a positive edge clock construct in the HDL code. You can specify the circuitry in one of the following ways:

- *cell_list*

This option specifies a list of 1 to 4 strings that describe the gates inserted in the clock network. The circuitry must be compatible with the latch style specified by the -*sequential_cell* option (for example, AND or NAND functionality for a latch-based style and OR or NOR functionality for a latch-free style).

- *integrated*

This option uses a single special integrated cell instead of the clock-gating circuitry.
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> set power_driven_clock_gating true
dc_shell> set_clock_gating_style     -sequential_cell latch   -control_point before
              -control_signal scan_enable       -positive_edge_logic integrated
dc_shell> compile_ultra -gate_clock -scan
dc_shell> insert_dft
dc_shell> report_clock_gating
dc_shell> report_power
```

4.3.3 Multiple V_{th} Library Cells:

Some CMOS technologies support the fabrication of transistors with different threshold voltages (V_{th} values). In that case, the cell library can offer two or more different cells to implement each logic function, each using a different transistor threshold voltage. For example, the library can offer two inverter cells: one using low V_{th} transistors and another using high V_{th} transistor. A low V_{th} cell has higher speed, but higher sub-threshold leakage current. A high V_{th} cell has low leakage current, but less speed. The synthesis tool can choose the appropriate type of cell to use based on the tradeoff between speed and power. For example, it can use low V_{th} cells in the timing-critical paths for speed and high V_{th} cells everywhere else for lower leakage power.

4.3.4 Multi-Voltage Design:

Different parts of a chip might have different speed requirements. For example, the CPU and RAM blocks might need to be faster than a peripheral block. As mentioned earlier, a lower supply voltage reduces power consumption but also reduces speed. To get maximum speed and lower power at the same time, the CPU and RAM can operate with a higher supply voltage while the peripheral block operates with a lower voltage, as shown in **Fig. 4.7**

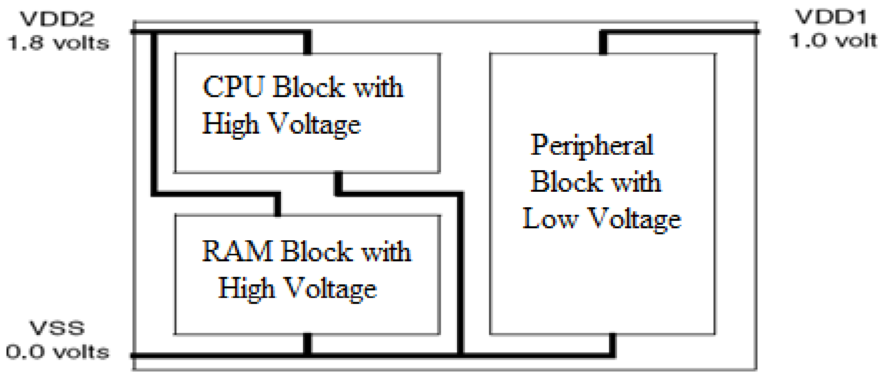


Fig 4.7: Multi-Voltage Chip Design

Providing two or more supply voltages on a single chip introduces some complexities and costs. Additional device pins must be available to supply the chip voltages, and the power grid must distribute each of the voltage supplies separately to the appropriate blocks. Where a logic signal leaves one power domain and enters another, if the voltages are significantly different, a level-shifter cell is necessary to generate a signal with the proper voltage swing. In the example shown in **Fig. 4.8**, a level shifter converts a signal with 1.8-volt swing to a signal with a 1.0-volt swing. A level shifter cell itself requires two power supplies that match the input and output supply voltages.

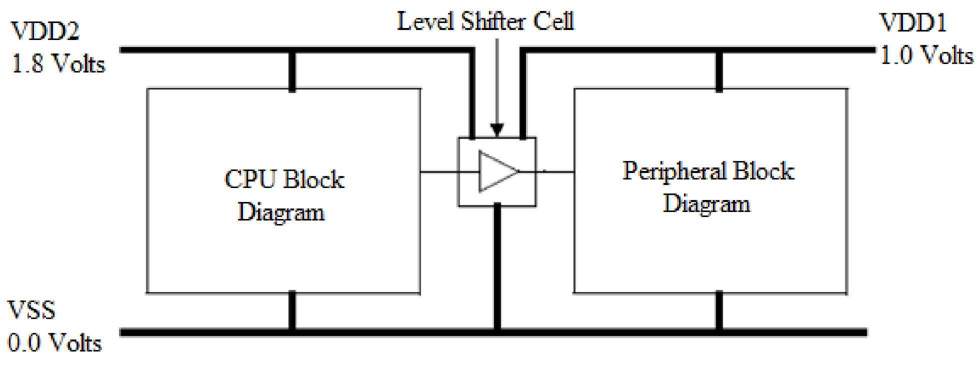


Fig 4.8: Level Shifter

4.3.5 Power Switching:

Power switching is another power-saving technique in which portions of the chip are shut down completely during periods of inactivity. For example, in a cell phone chip, the block that performs voice processing can be shut down when the phone is in standby mode. When the user places a call or receives an outside call, the voice processing block must “wake up” from its powered-down state.

Power switching has the potential to reduce overall power consumption substantially because it lowers leakage power as well as switching power. It also introduces some additional challenges, including the need for a power controller, a power-switching network, isolation cells, and retention registers.

A power controller is a logic block that determines when to power down and power up a specific block. There is a certain amount of time and power cost for powering down and powering up a block, so the controller should determine the appropriate power-down times with a high degree of certainty.

A block that can be powered down must receive its power through a power-switching network, consisting of a larger number of transistors with source-to-drain connections between the always-on power supply rail and the power pins of the cells. The power switches are distributed physically around or within the block. The network, when switched on, connects the power to the logic gates in the block. When switched off, the power supply is effectively disconnected from the logic gates in the block. High-V_{th} transistors from a Multiple-Threshold CMOS technology are used for the power switches because they minimize leakage and their switching speed is not critical.

There are two strategies for applying power switching technique:

1. Coarse-grain strategy because the power switching is applied to the whole block. Multiple transistors in parallel drive a common supply net for the block.
2. Fine-grain strategy, each library cell has its own power switch, allowing fine-grain control over which cells are powered down. The fine-grain approach has better potential for power savings, but requires significantly more area.

Any use of power switching requires isolation cells where signals leave a powered-down block and enter a block that is always on (or currently powered up). An isolation cell provides a known, constant logic value to an always-on block when the power-down block has no power, thereby preventing unknown or intermediate values that could cause crowbar currents. One simple implementation of an isolation cell is shown in **Fig. 4.9**. When the block on the left is powered up, the signal P_UP is high and the output signal passes through the isolation cell unchanged (except for a gate delay). When the block on the left is powered down, P_UP is low, holding the signal constant at logic 0 going into the always-on block. Other types of isolation cells can hold logic 1 rather than 0, or can hold the signal value latched at the time of the power-down event. Isolation cells must themselves have power during block power down periods.

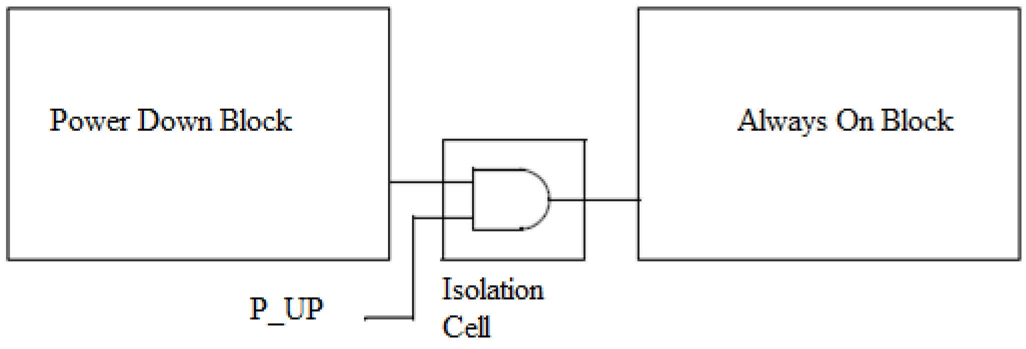


Fig. 4.9: Isolation Cell

The power switching technique can be combined with Multi-Voltage operation. Different blocks can be designed to operate at different voltages and also to be separately powered down when they are not needed. In that case, the interface cells between different blocks must perform both level shifting and isolation functions, depending on whether the two blocks are operating at different voltages or one is shut down. A cell that performs both functions is called an enable level shifter. This cell must have two separate power supplies, just like any other level shifter. When a block is powered down and then powered back up, it is often desirable for the block to be restored to the state it was in prior to the power-down event. There are several strategies for doing this:

1. The block register contents can be copied to RAM outside of the block before power-down, and then copied back after power-up.
2. Retention registers in the power-down block. A retention register can retain data during power-down by saving the data into a shadow register (also known as the bubble register) prior to power-down. Upon power-up, it restores the data from the shadow register to the main register. The shadow register has an always-on power supply, but it is constructed with high-V_{th} transistors to minimize leakage during the power-down period. The main register is built with fast but leaky low-V_{th} transistors.

One type of retention register implementation is shown in **Fig. 4.10**. The SAVE signal saves the register data into the shadow register prior to power-down and the RESTORE signal restores the data after power-up. Instead of using separate, edge-sensitive SAVE and RESTORE signals, a retention register could use a single level-sensitive control signal.

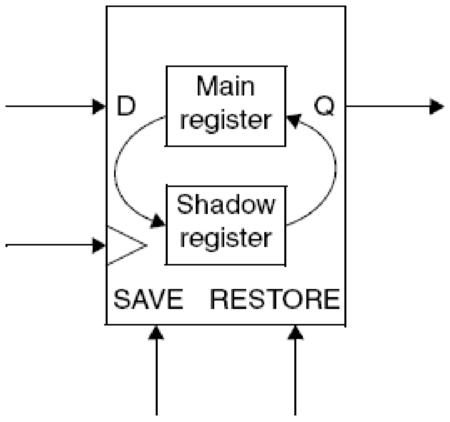


Fig. 4.10: Retention Register

A retention register occupies a larger area than an ordinary register, and it requires an always-on power supply connection for the shadow register in addition to the power-down supply used by the rest of the device. However, restoring the data to the registers after power-up is fast and simple compared with other strategies.

4.3.6 Dynamic Voltage and Frequency Scaling:

The principle of Multi-Voltage operation can be extended to allow the voltage to be changed during operation of the chip to match the current workload. For example, a math processor chip in a laptop computer might operate at a lower voltage and lower clock frequency during simple spreadsheet computations, thereby saving power; and then at a higher voltage and higher clock frequency during 3-D image rendering when the highest performance is needed. The changing of supply voltage and operating frequency during operation to meet workload requirements is called dynamic voltage and frequency scaling.

The chip and voltage supply can be designed to use a number of established levels, or even a continuous range. Dynamic voltage scaling requires a multilevel power supply and a logic block to determine the best voltage level to use for a given task. Design, implementation, verification, and testing of the device can be especially challenging because of the ranges and combinations of voltage levels and operating frequencies that must be analyzed and accommodated. Dynamic voltage scaling can be combined with power switching technology so that each block in the design can operate at multiple voltage levels for different performance requirements, or shut off completely when not needed at all.

Chapter 5: Design For Testability

5.1 Introduction:

DFT techniques are increasingly gaining momentum among ASIC designers. These techniques provide measures to comprehensively test the manufactured device for quality and coverage. 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.DFT has two main parts, Controllability and Observability

- Controllability: The ability to set some circuit nodes to a certain states or logic values.
- Observability: The ability to observe the state or logic values of internal nodes.

5.2 Types of DFT:

Many vendors, including Synopsys provide solutions for incorporating testability in the design. Synopsys adds the DFT capabilities to DC through its DFT Compiler (DFTC) that is incorporated within the DC suite of tools.

The main DFT techniques that are currently in use today are:

1. Scan insertion
2. Memory BIST insertion
3. Logic BIST insertion
4. Boundary-Scan insertion

5.2.1 Scan Insertion:

It is the most widely used DFT technique of all the other ones. It is used to test the chip for manufacture defects such as stuck-at-faults. It has high possibility to attain a very high fault coverage percentage (usually above 95%).

The scan insertion technique involves replacing all the flip-flops in the design, with special flops that contain built-in logic, solely for testability. The most prevalently used architecture is the multiplexed flip-flop. This type of architecture incorporates a 2-input MUX at the input of the D-type flip-flop. The select line of the MUX determines the mode of the device, i.e., it enables the MUX to be either in the normal operational mode (functional mode with normal data going in) or in the test mode (with scanned data going in). These scan-flops are linked together (using the scan-data input of the MUX) to form a scan-chain, that functions like a serial shift register. During scan mode, a combination of patterns are applied to the primary input, and shifted out through the scan-chain. If done correctly, this technique provides a very high coverage for all the combinational and sequential logic within a chip.

Other architectures available along with multiplexed type flip-flops are the *lssd* structure, *clocked scan* structure etc. As mentioned above, the most commonly used architecture is the multiplexed flip-flop. Scan can also be used to test the DUT (Design Under Test) for any possible timing violations.

Basically, scan uses two cycles: Capture and Shift. Scan data is injected from the primary inputs into the device where it is captured by the flops (going through logic) and is then shifted out to primary outputs where it is compared against the expected results. The signal that selects between the capture and shift cycle is usually called the *scan_enable* signal. In addition another signal is also used which is usually called the *scan_mode* signal. *Scan mode* signal is used to put the DUT in test conditions. In general, designs are modified such that under test conditions the device behaves different as opposed to the normal functional behavior. This modification is desired in order to achieve greater control and/or observability. Sometimes it is done simply to comply with the strict DFT rules.

Scan Insertion is done through some steps:

1. Load/Unload scan chain (shift cycle)
2. Force primary inputs (except clocks)
3. Measure primary outputs
4. Pulse clock to capture functional data (capture cycle)

The most important two processes in scan Insertion technique are to understand Shift and Capture cycles.

5.2.1.1 Shift Cycle:

It is a cycle where Data traverses the entire design through a big chain of registers. All the registers are linked together to behave as a shift register, that's the reason for naming this cycle by **Shift Cycle.** The Shift Cycle uses the **SD** input in the FF. This is the main difference between shift and capture cycle.

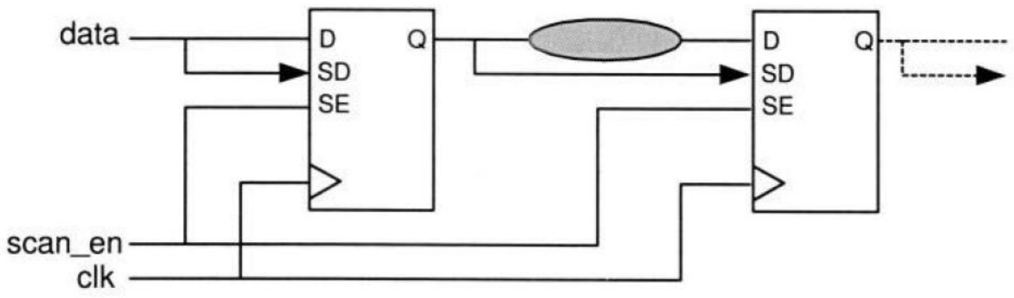


Fig. 5.1: Shift Cycle

Scan_enable signal controls which input to be used. In other words, Shift cycle needs several clock cycles (depending on the length of the scan chain) to be done because for each clock cycle, the data input will be shifted through one register from the whole number of linked registers. Data is injected through the primary input and shifted out of the device through the **SD** input port of the FF.

5.2.1.2 Capture Cycle:

It is a cycle where Data traverses the functional path. In other words, Data goes through the logic of the design as if the device is working under normal conditions. If the clock pulse has the same frequency as the functional one, all timing relationships between registers can be checked during the scan testing process. Data is injected to the device through **D** input port in the FF. *Scan_enable* signal is used to control which input to be used. In other words, one clock pulse is needed to capture the data that will be shifted out to be compared by the tester.

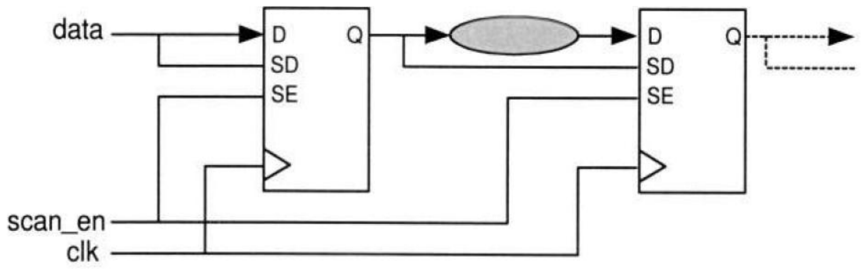


Fig. 5.2: Capture Cycle

5.2.1.3 Example for clarification:

Assume *scan_enable* port is active high for shift operation. Once the chain has been flushed out and compared, the *scan_enable* signal is toggled (driven low). Now a single clock pulse is applied to capture the data into the flops through the “D” inputs, before the *scan_enable* is toggled again (driven high) and the data shifted out for comparison.

Most designers when using scan insertion method for scan testing, they add a test signal called *scan_mode* port which is connected to a 2x1 MUX where the 2-inputs are the functional clock pulse and its inversion. During normal functional operation, an inverted clock is fed to the second register, while in case of testing mode of operation; the *scan_mode* signal selects the non-inverted clock. In this case the capture cycle differs from the functional cycle. Therefore, testing the timing for the original paths can’t be performed.

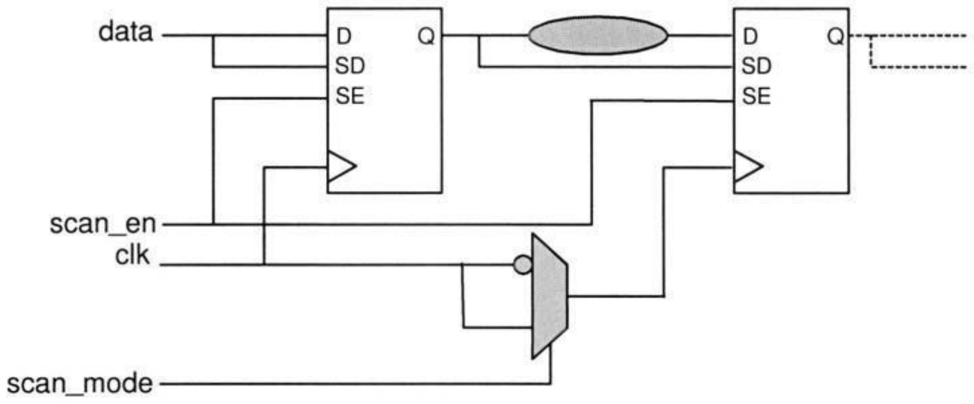


Fig. 5.3: Capture and Functional cycle differs

5.2.2 Memory BIST insertion:

The Memory BIST includes of controller logic that uses different algorithms to generate input patterns that are used to exercise the memory elements of a design (say a RAM). The BIST logic is automatically generated based upon the size and configuration of the memory element. It is generally in the form of synthesizable HDL language (Verilog or VHDL), which is inserted in the RTL source with hookups, leading to the memory elements. Upon triggering, the BIST logic generates input patterns that are based upon predefined algorithm, to fully examine the memory elements. The output result is fed back to the BIST logic, where a comparator is used to compare what went in, against what was read out. The output of the comparator generates a pass/fail signal that signifies the authenticity of the memory elements.

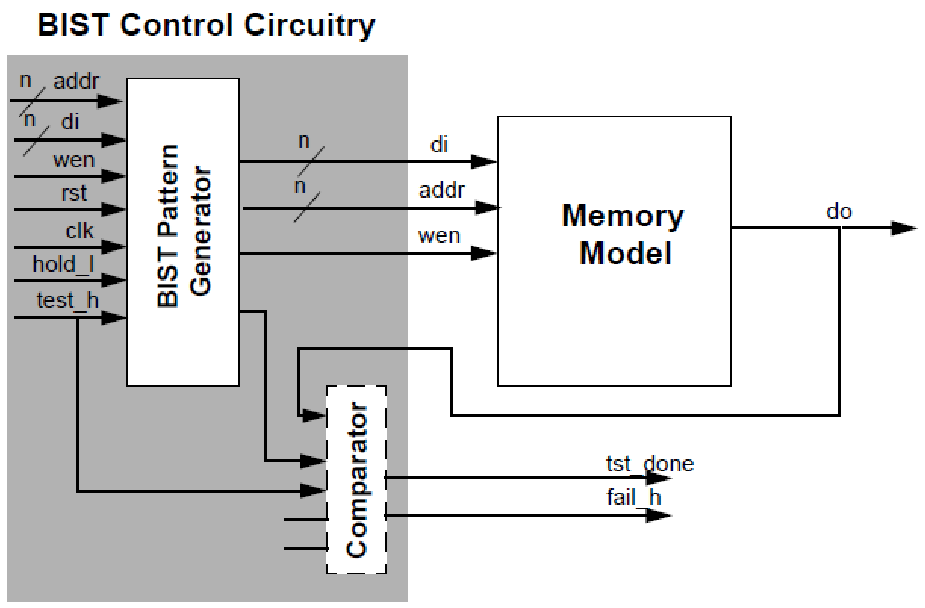


Fig. 5.4: Memory BIST insertion

5.2.3 Logic BIST insertion:

Logic BIST uses the same approach but targets the logic part of the design. The logic BIST uses a random pattern generator to exercise the scan chains in the design. The output is a compressed signature that is compared against simulated signature. If the signature of the device under test (DUT) matches the simulated signature, the device passes otherwise it fails. The main advantage of using logic BIST is that it eliminates the need for test engineers to generate huge scan vectors as inputs to the DUT. This saves tremendous amount of test time. The disadvantage is that additional logic (thus area) is incorporated within the design, just for testing purposes.

5.2.4 Boundary-Scan Insertion:

Boundary Scan was developed by the Joint Test Action Group of JTAG to perform basic test operation on circuitry; it involves serially connected registers that are found in the “boundary” of the chip. Because they are serially connected, data can be shifted in and shifted out of the device. This allows the tester to interact with the chip in a meaningful manner to determine the functionality of the chip.

Boundary Scan is primarily used for board test to detect: missing devices, damaged devices, shorts and opens, misaligned devices, wrong devices.

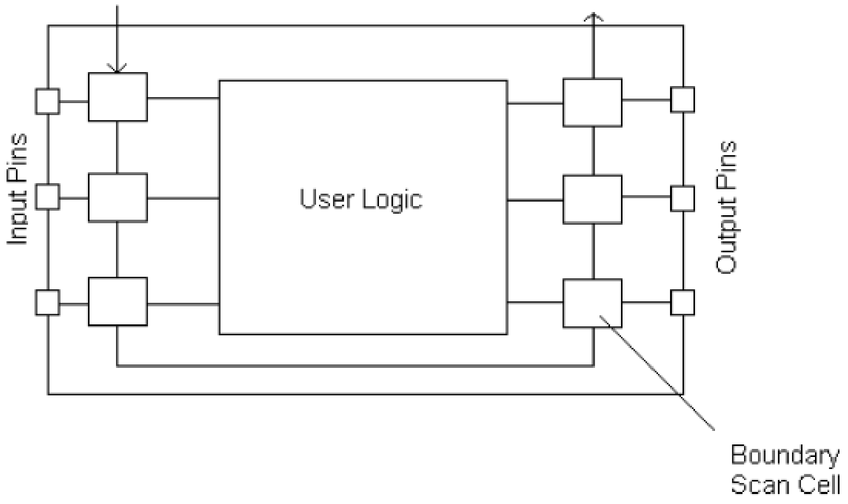


Fig. 5.5: Boundary Scan Insertion

Boundary Scan is very useful for testing board level interconnections which is called as EXTEST. It can be tested independently of the logic in any chip; therefore it leads to easier fault identification. Also Boundary Scan can be used to test the internal logic of the chip by driving the inputs to user logic from the boundary scan cells and shifting out the results; and this is called INTEST. It can also initiate BIST.

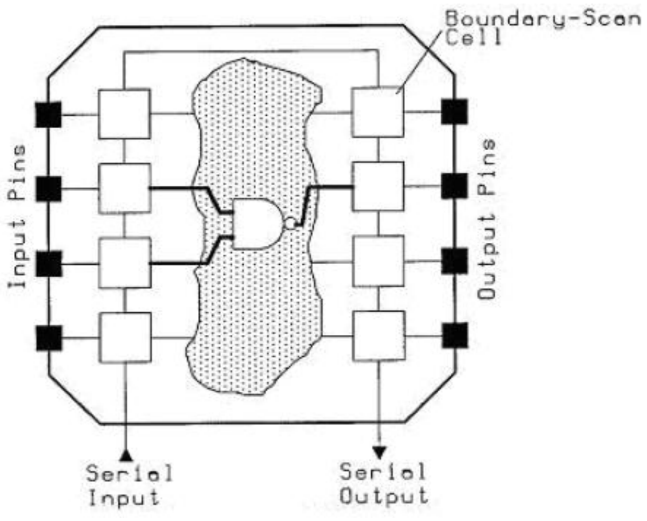


Fig. 5.6: Intest Boundary Scan

Boundary-Scan Insertion links all the input and output ports of the chips together into a long scan path, in which the output of the first chip is the input of the second. Data started in the scan path at the scan input port TDI (Test Data in) and ends at scan output port TDO (Test Data Out).

There are two more scan ports TCLK (Test Clock) and TMS (Test Mode Select) which are connected in parallel to each boundary scan cell in the path. The TAP (Test Access Port) controller is a state machine that controls the operation of the boundary scan circuitry.

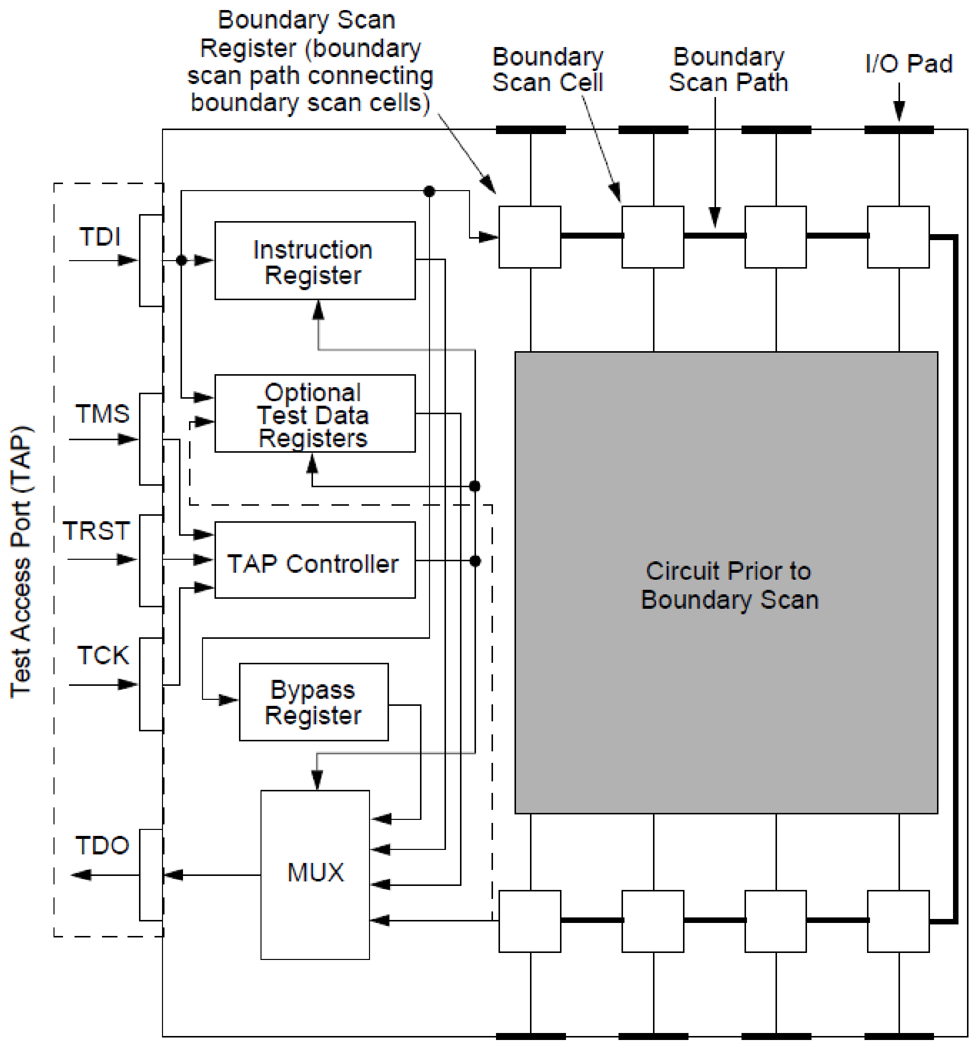


Fig. 5.7: Boundary Scan Architecture

Boundary Scan Architecture consists of:

1- Boundary Scan Registers
2- Test Access Ports (TDI, TDO, TMS, TRST, TCK)
3- Tap Controller
4- Instruction Register
5- By-pass Register
6- Optional Test Data Register

5.3 How to implement Scan chain using Design Compiler:

5.3.1 Modifying the top level source code:

As explained above, source code needs to have a minimal set of test scan ports added. So that when Design Compiler invokes the design, it can create the scan chain path in the design.
As mentioned before there are three main ports that must be added by the user in case of design for testability (DFT); TDI (test data in), TDO (test data out), TSE (scan enable).
Tclk and Trst do not need to be unique for scan, it can share regular clock and reset.

```
module design (din, dout, clk, rst, TDI, TDO, TSE)
    input din, clk, rst, TDI, TSE;
    output dout, TDO;
```

5.3.2 Set DFT settings:

There are many settings that has to be defined/specified for DFT setup

5.3.2.1 Specify signal type for DFT insertion:

set_dft_signal

-view existing_dft | spec

- ***existing_dft*** implies that the specification refers to the existing usage of a port. For example, when working with a DFT-inserted design, the command to indicate that port A is used as a scan enable follows this syntax:

  icc_shell > **set_dft_signal -view *existing_dft* -port *clk* -type *ScanClock***

- ***spec*** (the default value) implies that the specification refers to ports that the tool must use during DFT insertion. For example, when preparing a design for DFT insertion, the command to specify that port A is used as a test enable follows this syntax:

  icc_shell > **set_dft_signal -view *spec* -port *clk* -type *ScanClock***

-type Reset | ScanDataIn | ScanDataOut | ScanEnable | ScanClock

This specifies the signal type.

-timing

This specifies the rise and fall time for clocks. Used only when specifying clock port.

icc_shell> **set_dft_signal -view *existing_dft* -type *ScanClock* -port *clk* –timing *[list 45 55]***

-usage clock_gating | scan

This specifies the usage of the signal. You must also specify ***-view spec*** with this option. This option specifies that ***insert_dft*** is to use the specified signal for that purpose only.

- ***clock_gating*** specifies that the signal is to be connected to the test pin of clock gating cells during **insert_dft**. And as discussed earlier if you ***set_clock_gating_style*** and specify a ***latch-based style***, the tool creates a new input port to provide the test signal. The control points must be hooked up to the design level test_mode or scan_enable port using the ***insert_dft*** command.

  icc_shell> **set_dft_signal -view *spec* -port *TSE* - type *ScanEnable* -usage *clock_gating***
- ***scan*** specifies that the signal is to be connected to the enable of scan elements during ***insert_dft***. This usage is valid only for ***-type*** ScanEnable.

  icc_shell> **set_dft_signal -view *spec* -port *TSE* - type *ScanEnable* -usage *scan***

5.3.2.2 Define Scan Insertion Style:

set_scan_configuration

-style multiplexed_flip_flop | clocked_scan | none

This identifies the scan style. Select *none* if you have not selected a scan style for the design. By default, ***insert_dft*** uses the scan style value specified by environment variable ***test_default_scan_style*** in your .synopsys_dc.setup file.

5.3.2.3 Set DFT auto-fix Configuration and test protocol:

set_dft_configuration

This command specifies the DFT configuration for a design. The DFT configuration is specific to the current design.

-fix_clock enable | disable

This option enables or disables the clock Auto-Fix utility. By default, clocks are not auto-fixed.

-fix_reset enable | disable

This option enables or disables the reset Auto-Fix utility. By default, resets are not auto-fixed.

-fix_set enable | disable

This option enables or disables the set Auto-Fix utility. By default, sets are not auto-fixed.

icc_shell> **set_dft_configuration –fix_clock *enable* –fix_reset *enable* –fix_set *enable***

set_dft_drc_configuration

This command specifies the configuration of DFT DRC.

-allow_se_set_reset_fix true | false

If the value is set to true, it means that the user has bypass logic for internally generated reset using scan_enable. The switch can be used to include such scan flops in the scan chain which would be otherwise marked violated and excluded from the scan chains. The default value is false.

create_test_protocol

This command creates a test protocol for the current design based on user specifications issued prior to running this command. Such specifications are made using commands such as **set_dft_signal**.

-infer_clock

If this option is specified, it means it infers test clocks in the design

-infer_asynch

If this option is specified, it means it infers asynchronous set and reset signals in the design.

5.3.2.4 Check the settings:

dft_drc

This command checks the current design against test design rules

-verbose

This option controls the amount of detail when displaying violations. If specified, every violation instance is displayed. By default, only the first instance and the number of instances are displayed.

5.3.2.5 Preview and inserting DFT:

preview_dft

Previews, but does not implement, the test points, scan chains, and on-chip clocking control logic to be added to the current design.

-show

This option reports information for the specified objects as values.

-test_points

This option reports all test points information, in addition to the summary report the ***preview_dft*** command produces by default. The information displayed includes names assigned to the test points, locations of violations being fixed, names of test mode tags, logic states that enable the test mode, and names of data sources or sinks for the test points.

icc_shell> **preview_dft –show *all* –test_points *all***

insert_dft

This command adds internal scan or boundary scan circuitry to the design.

5.3.3 Generating Reports:

report_scan_configuration

This command displays the options that have been specified by the ***set_scan_configuration*** command.

The command reports the chain count, scan style, maximum scan chain length, rebalance scan chain length, preserve multi-bit segments, clock mixing, internal clocks, add lockup, insert terminal lockup, create dedicated scan out ports, shared scan in, and bidirectional mode.

report_scan_path

Displays scan paths and scan cells specified by the ***set_scan_path*** command and displays scan paths inserted by the ***insert_dft*** command.

-view existing_dft | spec

This option specifies the view from which to report. Valid options are **spec** and **existing_dft**. The **spec** view is used to report scan paths specified with the **set_scan_path** command. The **existing_dft** view shows scan paths already inserted by the **insert_dft** command. If not specified, the default is to use the **spec** view.

-cell chain_name | all

This option specifies chain names to report on scan cells and their order. By default the cells of the chains specified through the **-chain** option are not shown. With this option all cells in the chain are shown in the correct order. If **all** is specified, the scan cell order for all scan paths is reported.

icc_shell> **report_scan_path –view *existing_dft* –cell *all***

write_scan_def

Writes scan chain information in SCANDEF format for performing scan chain reordering using place and route tools. This file will be used in IC Compiler in placement stage.

icc_shell> **write_scan_def -output *./reports/FIFO.scandef***

write_saif

This writes a backward Switching Activity Interchange Format (SAIF) file.

-output file_name

This option specifies the name of the output SAIF file

-instances instances

This option specifies a list of instances for which the SAIF file will be generated. If not specified, SAIF file will be generated for the current instance.

icc_shell> **write_saif –output *FIFO.saif* –instances *FIFO***

Chapter 6: Introduction to IC Compiler

6.1 Introduction:

IC Compiler is an integral part of the Synopsys Galaxy Design Platform that delivers a complete design solution, including synthesis, physical implementation, low-power design, and design for manufacturability (DFM).

IC Compiler is a single, convergent chip-level physical implementation tool and also a 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 also includes flat and hierarchical design planning, placement and optimization, clock tree synthesis, routing, DFM, and low-power capabilities that enable designers to implement today's high-performance, complex designs.

6.2 IC Compiler – Overview

IC Compiler provides a comprehensive DFM solution that concurrently optimizes for yield with timing, area, power, test, and routeability. IC Compiler increases manufacturability of the design, optimizing both functional and parametric yield.

IC Compiler with concurrent hierarchical design enables powerful design planning and chip-level analysis features to handle large, complex designs. Providing early analysis and feasibility exploration, IC Compiler delivers smaller die size and achieves predictable design closure to reduce the cost of design.

IC Compiler with Zroute technology utilizes advanced routing algorithms, concurrent DFM optimizations and multithreading to deliver a combined speed increase of more than 10X in routing

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

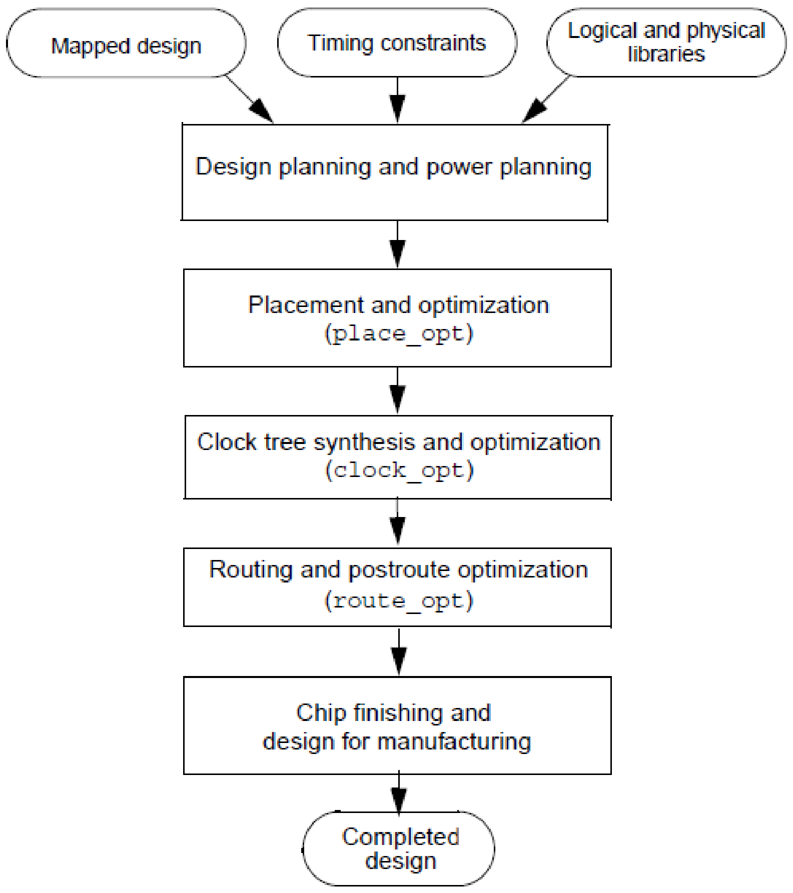

Mapped design
Timing constraints
Logical and physical libraries
Design planning and power planning
Placement and optimization (place_opt)
Clock tree synthesis and optimization (clock_opt)
Routing and postroute optimization (route_opt)
Chip finishing and design for manufacturing
Completed design

Fig. 6.1: IC Compiler (Back End) design flow

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.
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.
IC Compiler clock tree synthesis and embedded optimization solve complicated clock tree synthesis problems, such as blockage avoidance and the correlation between pre-route and post-route 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 inter-clock delays.

5. Perform routing and post-route optimization.

To perform routing and post-route optimization, use the route_opt core command.
As part of routing and post-route 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 post-route 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 post-route 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.

6.3 How to invoke IC Compiler:

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.

For invoking IC Compiler:

1. Log in to the UNIX environment with the user id and password
2. Change the working directory of the Unix to the Synopsys version E-2010.12-SP2
   UNIX$ cd   /usr/synopsys/E-2010.12-SP2
3. Change the environment variable for IC Compiler version E-2010.12-SP2
   UNIX$ cd   .setup-ic.sh
4. Go back again to your home directory as it was when you logged in.
   UNIX$ cd   /home/users (number)/your_username
5. Start IC Compiler from the UNIX promt:
   UNIX$ icc_shell
   The xterm UNIX prompt turns into the IC Compiler shell command prompt.
6. 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 Getting Started:

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

6.4.1 Setting Up Libraries:

6.4.1.1 Logical Libraries:

IC Compiler uses logical libraries as they provide timing and functionality information for all standard cells (and, or, flip-flop ….). It is also provide timing information for hard macros (IP, RAM, ROMs …….).

Logical libraries define drive/load design rules (max fanout, max transition, max/min capacitance) and they are usually the same ones used by Design Compiler during front end synthesis.

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*

It lists the path for which the libraries are located so IC Compiler can invoke them.

lappend search_path [………….]

- *target_library*

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

set_app_var target_library "sc_max.db"

- *link_library*

It lists the logic libraries that IC Compiler can search to resolve all the instantiated components in a netlist.

*set_app_var link_library "*sc_max.db io_max macros_max.db"*

The *set_min_library* command is used to define the "fast corner" library that is intended to be used with its corresponding "slow corner" library.

set_min_library sc_max.db –min_version sc_min.db

These settings can be re-applied in each new IC Compiler session, or more conveniently, entered once in the *.synopsys_dc.setup* file, which is already done when invoking design compiler and automatically read by the tool when ICC is invoked.

6.4.1.2 Physical Reference Libraries:

IC Compiler uses Milkyway reference libraries and technology (.tf) files to provide physical library information. It contains physical information of standard cells, macros, and pad cells necessary for placement and routing.

It defines placement unit tile:

- Height of placement rows
- Minimum width resolution
- Preferred routing directions
- Pitch of routing tracks
- ……

➢ **The technology file (.tf file)**

It is unique to each technology; it contains metal layer technology parameters:

- Number and name designations for each layer/via
- Physical and electrical characteristics of each layer/via
- Design rules for each layer/via (Minimum wire widths and wire-to-wire spacing, etc.)
- Units and precision for electrical units
- Colors and patterns of layers for display

Specifying the *tech* file is written inside *.synopsys_dc.setup* file using command:

Set tech_file "…../usr/synopsys/E-2010.12-SP2/ref/tech/cb13_6m.tf"

6.4.2 Creating a Milkyway Design Library:

The design library is a Milkyway UNIX-based database eventually structure, created by the user, which will contain all the associated input data required for placement, CTS, routing, etc., as well as the physical “design cell” or layout.

The first step in data-setup is to create the design library. This entails giving the library a user-defined name and specifying the technology file as well as the physical reference libraries, (layout cells for standard cells, macro, and IO pad cells).

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.

It is specified with the command:
create_mw_lib –mw_reference_library "./libs/sc ./libs/macros " –technology ./libs/abc_6m.tf
or if using GUI: File> create_library

6.4.3 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.

icc_shell> **check_library**

By default, the *check_library* command performs consistency checks between the logic libraries specified in the *link_library* variable and the physical libraries in the current Milkyway design library.

By default, the *check_library* command performs the following consistency checks between the logic library and the physical library:

- Verify that all cells exist in both the logic library and the physical library, including any physical-only cells.
- Verify that the pins on each cell are consistent between the logic library and the physical library.
- Verify that the area for each cell (except pad cells) is consistent between the logic library and the physical library.
- Verify that the cell footprints are consistent between the logic library and the physical library.
- Verify that the same bus naming style is used in the logic library and the physical library.

6.4.4 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 uses 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.

6.4.5 Reading a 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

6.4.6 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).

```
icc_shell> copy_floorplan -from design1
```

6.4.7 Preparing for Timing analysis and RC Calculation:

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

6.4.7.1 Setting up TLUPlus files:

TLUPlus is a binary table format that stores the RC coefficients. The TLUPlus models enable accurate RC extraction results by including the effects of width, space, density, and temperature on the resistance coefficients.

User must specify the following files:

- The map file

The map file matches the layer and via names in the Milkyway technology file with the names in the ITF file.

```
icc_shell> set_tlu_plus_files –tech2itf_map   ./libs/abc.map
```

- The maximum TLUPlus model files

```
icc_shell> set_tlu_plus_files –max_tluplus   ./libs/abc_max.tlup
```

- The minimum TLUPlus model files (Optional)

```
icc_shell> set_tlu_plus_files –min_tluplus   ./libs/abc_min.tlup
```

6.4.7.2 Back annotate delay or parasitic delay:

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 back-annotate the design with parasitic capacitance and resistance information, use the *read_parasitics* command. IC Compiler accepts parasitic data in Synopsys Binary Parasitic Format (SBPF) or Standard Parasitic Exchange Format (SPEF).

6.4.7.3 Setting 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 ./scipts/design_name.sdc***

6.4.7.4 Specifying 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, so that IC Compiler uses the appropriate set of parameter values in the technology library.

6.4.7.5 Set the delay calculation algorithm:

IC Compiler uses Elmore delay calculation for both pre-route and post-route delay calculations. For post-route 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 the Arnoldi calculation is best used for designs with smaller geometries and high resistive nets, but it requires more runtime and memory.

To enable Arnoldi delay calculation for clock nets only, enter the following command:

icc_shell> **set_delay_calculation -clock_arnoldi**

To enable Arnoldi delay calculation for all nets, enter the following command:

icc_shell> **set_delay_calculation –arnoldi**

6.4.8 Save the design:

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

Chapter 7: Design Planning

7.1 Introduction:

Design planning in IC Compiler provides you with 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 electro migration analysis.

Design Planning is intended to be used for both a fast exploration of the design to reduce die size and to implement a final, optimized and detailed floorplan. Min-Chip technology in IC Compiler allows designers to automatically implement the smallest routable die for their design while power network synthesis and analysis automatically creates a power network that meets IR drop requirements.

This chapter contains the following sections:

- 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

7.2 Initialize Floorplan:

The steps in initializing the floorplan are described in the following sections:

- Reading I/O Constraints:

  You place the top-level I/O pads and pins according to the constraints. The I/O constraints are used for pad and terminal placement. You can use the *read_io_constraints* command.

- Defining the core and placing the I/O pads:

  You can estimate the die size for the core area of the design, based on input control parameters such as target utilization, aspect ratio, core size, row number, and boundary, and then place the I/O pads and pins. You can 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*.

- Adding Cell Rows:

  You can add cell rows in a specified area by entering the coordinates of diagonal corners of a rectangle. You can also create cell rows with different heights, depending on the unit tile you specify. You can use the *add_row* command

- Removing Cell Rows:

You can remove (cut) all existing cell rows from the current design, or you can cut single cell rows in a specified area. You can use *cut_row* command

- Save the floorplan information:

You can write (save) the floorplan information from the top cell of the design, regardless of the current instance. The output is a command script file that contains information about the floorplan; therefore, you can load it as a common script file. The script file describes the floorplan information of the current Milkyway CEL view. You can use the *write_floorplan* command

- Writing Floorplan Physical Constraints for Design Compiler Topographical Mode:

IC Compiler can write out the floorplan physical constraints in Tcl format for use by Design Compiler topographical mode. The reason for using floorplan physical constraints in the Design Compiler topographical mode is to accurately represent the placement area and to improve timing correlation with the post-place-and-route design. Also, using floorplan information is recommended if the floorplan is very complicated, for example, if the design contains many macros or a large number of blockages and keepout regions, or if the core area contains unusual shapes. The command syntax is:
write_physical_constraints -output output_file_name   -port_side

7.3 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.

7.4 Handling Black Boxes:

Milkyway defines a black box as any instance in the netlist that does not have either a corresponding leaf cell in a reference library or a netlist module defined in terms of a leaf cell.
Therefore, a black box can have an empty module represented in the netlist where its ports and their direction are defined. If the black box instance in the netlist does not have a corresponding module, the ports of the black box are defined in the Milkyway database with inout directions.
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.

7.5 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 in the following sections:

- 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 pre-placement of hard macros and improve the QoR of the hard macro placement.

➢ Creating a User-Defined Array of Hard Macros

A macro array is treated as a single object during placement – standard cells cannot be placed inside the array.

set_fp_macro_array -name *array_name*
[-elements *list_of_macro_cell_objects*]
[-x_offset] [-y_offset]

```
                    [-use_keepout_margin]
                    [-align_edge t | b | l | r | c]
                    [-align_pins {list of two pin objects}]
                    [-align_2d lb | lc | lt | rb | rc | rt | cb | cc | cr]
                    [-vertical]
                    [-rectilinear]
                    [-reset]
```

➢ Setting Floorplan Placement Constraints On Macro Cells

User can define placement constraints that restrict specified macro cells and macro arrays to particular regions, orientations, alignments, and so forth by using the *set_fp_macro_options* command.

```
set_fp_macro_options        $ list_of_macro_objects
                    [-Iegal_orientations {legal_orientations}]
                    [-anchor_bound t | b | l | r | tl | tr | bl | br | tm | bm | lm | rm | c]
                    [-x_offset     distance_from_core_edge]
                    [-y_offset     distance_from_core_edge]
                    [-align_pins   ref_port_of_block     constrained_pin]
                    [-side_channel {Ieft_dist    right_dist    top_dist      bottom_dist}]
                    [-reset]
```

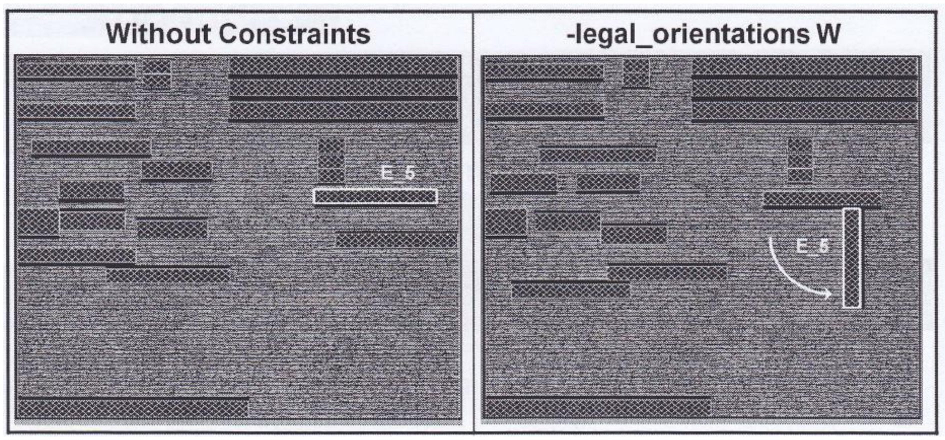


Fig. 7.1: Macro Constraints [Legal Orientation]

set_fp_macro_options [get_cells E-5] –legal_orientations W

➢ Placing a Macro Cell Relative to an Anchor Object

User can define placement constraints that restrict specified macro cells and macro arrays to particular regions, orientations, alignments, and so forth by using the *set_fp_macro_options* command.

```
set_fp_relative_location
                    -name           constraint_name
                    -target_cell    cell__name
                    [-target_orientation N | S | E | W | FN | FS | FE | FW]
                    [-target_corner   bl | br | tl | tr]
                    [-anchor_object   object_name]
                    [-anchor_corner   bl | br | tl | tr]
                    [-x_offset      distance]
                    [-y_offset      distance]
```

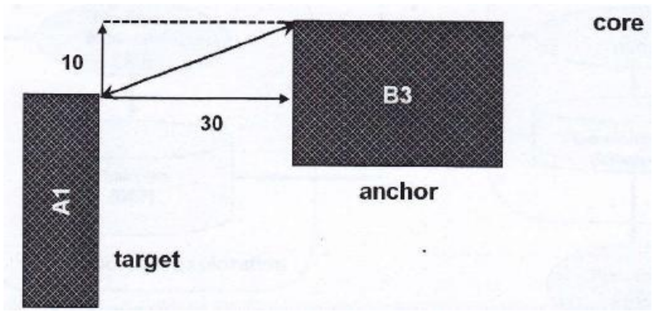


Fig. 7.2: Macro Constraints [Relative Location]

icc_shell> **set_fp_relative_location –name *RP1* –target_cell *“A1”* –target_orientation *“S”* \ -target_corner *“tr”* –anchor_object *“B3”* –anchor_corner*”tl”* –x_offset *30* –y_offset *10***

- Using a Virtual Flat Placement Strategy

This section describes the constraints you can use to control the virtual flat placement.

set_fp_placement_strategy

-macro_orientation	**automatic** | all | N
-auto_grouping	none | user-only | **low** | high
-macro_setup_only	on | **off**
-macros_on_edqe	on | **off**
-sliver_size	<**0.00**>
-snap_macros-to-user-grid	on | **off**
-fix_macros	**none** | soft_macros_only | all
-congestion_effort	**low** | high

-IO_net_weight	<**1.0**>
-plan_group_interface_net_weight	<**1.0**>
-voltage_area_interface_net_weight	<**1.0**>
-voltage_area_net_weight_LS_only	on | **off**
-spread_spare_cel1s	**on** | off
-legalizer_effort	low | **high**
-virtual_IPO	on | **off**
-pin_routing_aware	on | **off**

icc_shell> **set_fp_placement_strategy -sliver_size *10* -virtual_IPO *on***

In the example above, standard cells won't be placed in channels between macros of less than 10 microns, also with VIPO turned on; sizing and buffer insertion is performed in memory, thus relaxing the need to unnecessarily shorten the distance between the critical cells.

- Creating Macro Blockages for Hard Macros

If you do not want hard macros to be placed in certain areas of your design, you can create a macro blockage to restrict the placement of hard macros.
There might be a channel between two plan groups or two hard macros that is large enough for standard cells to go through, but if a hard macro is placed in the channel, routing would be adversely affected. In this case, placing a macro blockage in the channel restricts the placement of the hard macro.
This can be done by using command *create_placemen_blockage*

- Padding the Hard Macros

To avoid placing standard cells too close to macros, which can cause congestion or DRC violations, you can set a user-defined padding distance or keepout margin around the macros. During virtual flat placement, no other cells will be placed within the specified distance from the macro's edges.

icc_shell> **set_keepout_margin -type *hard* | *soft* –outer *{left bottom right top}***

7.6 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.

7.7 Perform Power Planning:

After you have completed the design planning process and have a complete floorplan, you can perform power planning.

- Creating Logical Power and Ground Connections:

You must create logical power and ground connections throughout the design hierarchy if the power and ground nets are not explicitly defined in your netlist. To define these connections, you need to associate port patterns on the standard cells or macros with a power or ground net.

icc_shell> **derive_pg_connection -power_net *VDD* -power_pin *VDD* -ground_net *VSS***
-ground_pin *VSS*
icc_shell> **derive_pg_connection -power_net *VDD* -ground_net *VSS* -tie**

- Adding Power and Ground Rings:

After you do floorplanning, you need to add power and ground rings using command *create_rectangular_rings.*

- Adding Power and Ground Straps:

After you add the power and ground rings, you need to add the power and ground straps. The straps are automatically connected to the closest power and ground ring at, or beyond, both ends of the straps, using command *create_power_straps.*

- Pre-route Standard Cells:

You can connect power and ground pins in standard cells to the straps and rings of the power mesh and connect power and ground rails in the standard cells. To make sure the global router can recognize the routing obstruction, pre-route the standard cells before performing global routing. User can use *preroute_standard_cells* command.

- 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 electro migration, 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.
To perform the PNS, one can run the set of following commands:

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
commit_fp_rail

- Analyzing Power Network:

You can perform power network analysis to predict IR drop at different floorplan stages on both complete and incomplete power nets in your design. Power network analysis consists of extraction and analysis of the power or ground rail specified by the given net names. You can analyze IR drop and electro migration effects during initial power grid planning while the floorplan with top-level blocks is still being implemented in the design and the power ports of the standard cells are not yet connected. User can use *analyze_fp_rail* command.

- Reporting Settings for Power Network Synthesis and Power Network Analysis Strategies

You can 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.

7.8 Performing Prototype Global Routing:

Global routing is done to detect possible congestion "hot spots" that might exist in your floorplan due to the placement of the hard macros or inadequate channel spacing. You can perform prototype global routing to get an estimate of the routeability and congestion of your design. This can be done using *route_fp_proto* command.

7.9 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. You can perform clock planning on a specified clock net or on all clock nets in your design.
Clock planning tries to minimize clock skew by running block-level and top-level clock tree synthesis during the early stages of the design flow to determine the clock budgets, allocate resources for clock buffers and clock routes, determine optimal clock pin locations for soft macros, and provide an estimate of the block-level insertion delays and skew for each plan group prior to finalizing the floorplan. This can be done by setting some specifications:

- Setting Clock Planning Options:

Before you can compile clock trees inside the plan groups and build clock trees at the top level, you must first set different clock planning options such as anchor cell insertion, specifying nets for clock planning, and whether or not to route the clock nets after clock planning.
To set clock planning options use command *set_fp_clock_plan_options.*

- Performing Clock Planning Operations:

You can perform clock planning operations to compile the clock trees inside plan groups and build clock trees at the top-level based on the options you have selected in the Set Clock Plan Options.

To set clock planning operations use command *compile_fp_clock_plan.*

- Generating Clock Tree Reports:

You can use clock skew analysis to generate skew report to report skew information for a specified clock (or for all the clocks within a design) before or after routing using command *report_clock_tree*.

- 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.

7.10 Performing In-place Optimization:

In-place optimization is an iterative process that is based on virtual routing. You can run in-place optimization before or after global routing. If you decide to run in-place optimization after you have run global routing, you must first delete the global routes. This is necessary because in-place optimization has changed the netlist by adding or removing buffers or resizing existing buffers, and therefore, the original global routes are now invalid.

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.

Methods for improving timing include:

- Inserting buffers (legal locations are found automatically)
- Increasing or decreasing cell size.
- Moving cells

This can be done using command

icc_shell> **optimize_fp_timing –effort *low* | *medium* |*high*   -fix_design_rule   -area_recovery**

7.11 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 flows).
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 Assignment use, 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.

7.12 Performing RC Extraction:

Perform post-route RC estimation by using the *extract_rc* command.

7.13 Performing Timing Analysis:

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

7.14 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.
Timing budgeting distributes the timing constraints from a top design level constraint set into individual block level constraint sets for each of the top-level blocks in the design. Budgeting also allocates slack between blocks for timing paths crossing the block boundaries. Timing budgeting propagates the timing constraints downwards one hierarchical level at a time.

Timing budgeting also propagates timing exceptions to block-level constraint sets.

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.

7.15 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.

Converting plan groups to soft macros using command *commit_fp_plan_groups*

Pushing physical objects down to the soft macro level using command *push_down_fp_objects*

Pushing physical objects up to the top level using command *push_up_fp_objects*

Converting soft macros to plan groups using command *uncommit_fp_soft_macros*

7.16 Refining the Pin Assignment:

You 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. You can then refine the pin assignment based on the results.

Checking the pin assignment using command *check_fp_pin_assignment*

Checking the pin alignment using command *check_fp_pin_alignment*

Chapter 8: Placement

8.1 Introduction:

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 non-manufacture product by producing excessive wire length, 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 wire length: Minimizing the total wire length, or the sum of the length of all the wires in the design, is the primary objective of most existing placers. This not only helps minimize chip size, and hence cost, but also minimizes power and delay, which are proportional to the wire length (This assumes long wires have additional buffering inserted; all modern design flows do this.)
- Timing: The clock cycle of a chip is determined by the delay of its longest path, usually referred to as the critical path. Given a performance specification, a placer must ensure that no path exists with delay exceeding the maximum specified delay.
- Congestion: While it is necessary to minimize the total wire length to meet the total routing resources, it is also necessary to meet the routing resources within various local regions of the chip’s core area. A congested region might lead to excessive routing detours, or make it impossible to complete all routes.
- Power: Power minimization typically involves distributing the locations of cell components so as to reduce the overall power consumption, alleviate hot spots, and smooth temperature gradients.
- A secondary objective is placement runtime minimization.

A placer takes a given synthesized circuit netlist together with a technology library and produces a valid placement layout. The layout is optimized according to the aforementioned objectives and ready for cell resizing and buffering a step essential for timing and signal integrity satisfaction. Clock-tree synthesis and routing follow, completing the physical design process. In many cases, parts of, or the entire, physical design flow are iterated a number of times until design closure is achieved.

In the case of application-specific integrated circuits, or ASICs, the chip's core layout area comprises a number of fixed height rows, with either some or no space between them. Each row consists of a number of sites which can be occupied by the circuit components. A free site is a site that is not occupied by any component. Circuit components are standard cells, macro blocks, or I/O pads. Standard cells have a fixed height equal to a row's height, but have variable widths. The width of a cell is an integral number of sites. On the other hand, blocks are typically larger than cells and have variable heights that can stretch a multiple number of rows. Some blocks can have pre-assigned locations say from a previous floorplanning process — which limits the placer's task to assigning locations for just the cells. In this case, the blocks are typically referred to by fixed blocks. Alternatively, some or all of the blocks may not have pre-assigned locations. In this case, they have to be placed with the cells in what is commonly referred to as mixed-mode placement.

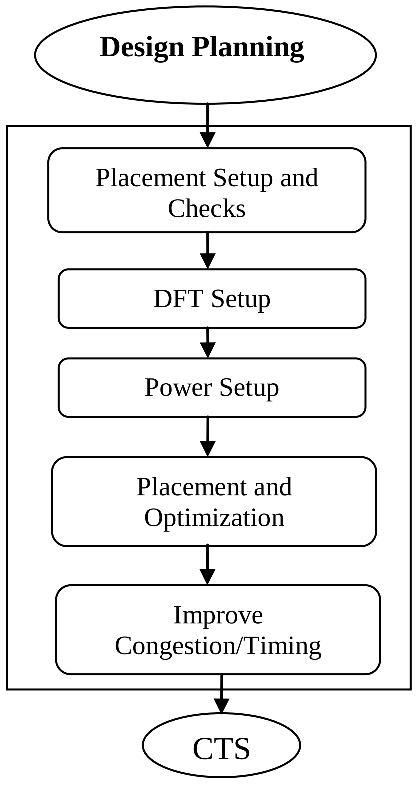

Design Planning
Placement Setup and Checks
DFT Setup
Power Setup
Placement and Optimization
Improve Congestion/Timing
CTS

Fig. 8.1: Placement flow

8.2. Tasks to be performed during Placement:

8.2.1 Placemen Setup and Checks:

8.2.1.1 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.

If you define both hard and soft placement blockages in your design, the hard placement blockages take priority over the soft placement blockages in places where they overlap.

To set hard global keepout margin use command *set physopt_hard_keepout_distance <0.00>*

To set soft global keepout margin use command *set physop_soft_keepout_distance <0.00>*

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

icc_shell> **set_keepout_margin -type *hard | soft* -outer *{left bottom right top}***

To report the cell-specific keepout margins in your design, use the *report_keepout_margin* command.

To report the placement blockages in your design, use the *report_placement_blockages* command.

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

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

8.2.1.2 Defining Move Bounds:

Move bounds are placement constraints that control the placement of groups of leaf cells and hierarchical cells. Move bounds can be either soft (the default), hard, or exclusive.

- Soft move bounds specify placement goals, with no guarantee that the cells will be placed inside the bounds.
- Hard move bounds force placement of the specified cells inside the bounds.

- Exclusive move bounds force the placement of the specified cells inside the bounds. All other cells must be placed outside the bounds.

Defining a move bound allows you to group cells to minimize wire length and place the cells at the most appropriate locations.
This can be done using command

icc_shell> **create_bounds -name *bound_name* -type *hard* | *soft* -exclusive -coordinate *{coord_list}***

8.2.1.3 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.
The high-fanout synthesis engine does not buffer nets that are set as ideal nets or nets that are disabled with respect to design rule constraints. It also does not work if only inverters are available for buffering.
In order to perform high-fanout net synthesis during *place_opt*, you can use the *create_buffer_tree* command to run standalone high-fanout net synthesis.
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.

8.2.1.4 Performing Clock Tree Synthesis:

If your design contains simple clock tree structures and uses the same design rule constraints for placement and clock tree synthesis, you can simplify the design flow by performing clock tree synthesis and optimization during placement.
To identify the clock tree synthesis options by using the *set_place_opt_cts_strategy* command.

8.2.2 DFT Setup:

If your design doesn’t include scan chains, skip this part completely. If your design flow includes “design for test”, the netlist will contain “scan chains”. Scan chain paths are active only during “test mode” not during “functional mode”.

IC Compiler can perform both placement-aware scan chain reordering of scan cells and Clock-aware scan chain reordering.

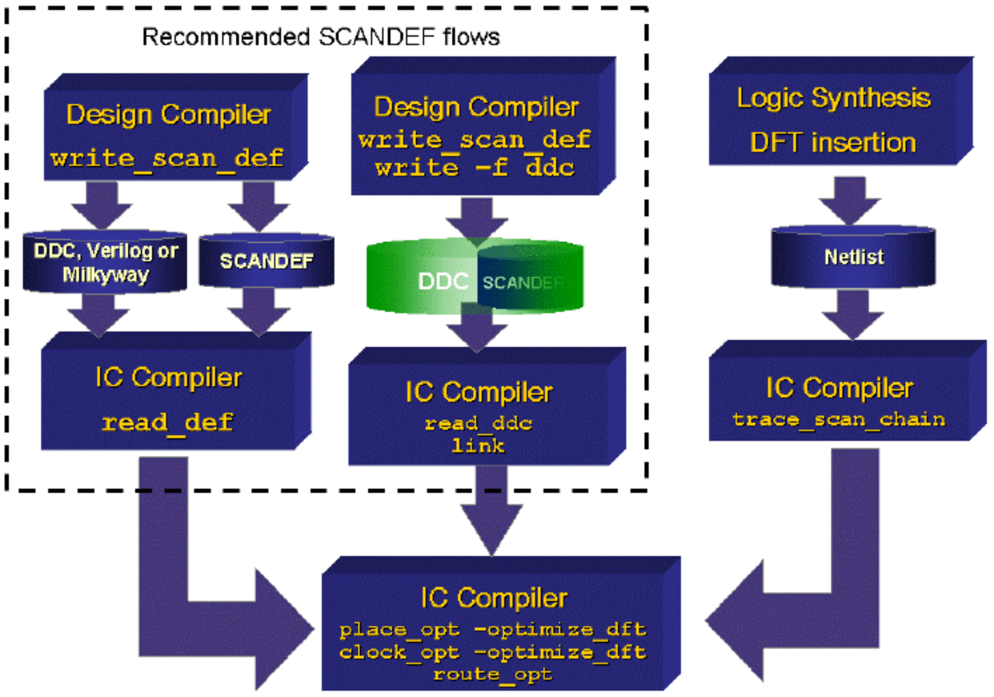


Fig. 8.2: Physical DFT flow in IC Compiler

The scan chain information from front end synthesis tool (Design Compiler) can be transferred to IC Compiler in two ways:

- By loading the netlist in .ddc format (imbedded SCANDEF).

During data setup, if the netlist from synthesis was read in as ddc format, then the SCANDEF information is included and is automatically ported over to IC Compiler using command *read_ddc.*

- By loading the netlist in Verilog or VHDL format

An explicit SCANDEF file must be loaded prior to placement. This SCANDEF file is typically written out during synthesis, after scan insertion, with the command *write_scan_def.*
The ASCII SCANDEF file can be loaded using the *read_def* command. To generate detailed information about errors and warnings that occur when reading the SCANDEF file, use the -*verbose* option.

icc_shell> **read_def *./myscan.def***

The SCANDEF data from the .ddc file can be loaded by using the read_file –format ddc, or read_ddc, or import_design command, followed by link as shown in the following example.

```
read_ddc ./design_with_scandef.ddc
link
check_scan_chain
report_scan_chain
place_opt -optimize_dft
```

Loading SCANDEF data from the .ddc file ensures that the netlist and the SCANDEF data are consistent. It also simplifies file management.

8.2.3 Power Setup:

If power optimization is not critical in your design, you can skip this part.

When you activate power optimization, IC Compiler can perform the following types of power optimization:

- Low-Power Placement
- Leakage Power Optimization
- Dynamic Power Optimization

The default setup for power optimization performs only leakage power optimization during placement or routing. To perform other power optimizations, you must enable each of them separately. You can also change the constraints and other settings that are used for these power optimizations. The following section explains the setup and use of the power optimization techniques.

For the best results for total power optimization, you should annotate the high-quality switching activity information to the design before you perform these power optimization functions. SAIF files are generated by simulators. A SAIF file has header information while the body contains hierarchical switching activity information.

To read in the switching activity information, use the *read_saif* command with the following syntax:

```
icc_shell> read_saif -input saif_file -instance path
```

Alternatively, you can use the *set_switching_activity* command. If no switching activity has been annotated to the design, IC Compiler applies the default toggle rate to the primary inputs and black box outputs and then propagates it throughout the design.
After setting up the power by reading the SAIF file or read in a user-generated *toggle-rate.tcl* file, it is now the time to start the power optimization techniques

- **Low Power Placement:**

Low-power placement reduces capacitance between nets with high switching activity. The reduction in capacitance is achieved by shortening the nets during optimization and placement.
To set up for low-power placement option

```
icc_shell> set_power_options     -low_power_placement true
```

- **Leakage Power Optimization:**

Leakage power optimization is an additional step to timing optimization that minimizes leakage power without degrading performance.
Leakage power optimization is enabled by default. To disable leakage power optimization, set the *-leakage* option to false using the *set_power_options* command. You can setup multiple threshold voltage libraries to reduce the leakage power, though leakage optimization works on single threshold libraries also.

- **Dynamic Power Optimization:**

Gate-level dynamic power optimization, is an additional step to timing optimization, to further reduce the dynamic power during a placement function. This optimization targets a dynamic power constraint, which is set, by default, to 0 for a well-balanced runtime and quality of results. You can run this optimization when performing placement and optimization or placement alone.

```
icc_shell> set_power_options     -dynamic true
```

8.2.4 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.

The *place_opt* command has a lot of options as follows:

-effort low |medium | high	Improve the quality of results or reduce runtime. Higher effort levels use additional runtime in an attempt to improve the quality of results. The default is **medium.**
-area_recovery	Recover area for cells not on timing-critical paths.
-power	Perform low-power placement. The power optimizations and low-power placement capability must be previously enabled
-optimize_dft	Reorder scan chains
-congestion	Reduce congestion for improved routeability. For best results, use this option only for congested designs.

You 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, but it can also perform power optimizations if you specify power constraints.

The *psynopt* command has a lot of options that can be used for incremental optimization during placement:

-area_recovery	Enable area recovery within the cluster boundary for noncritical paths. Using this switch can have a severe runtime impact if the design contains many high-fanout nets.
-power	Open power optimization. This activates the power optimizations enabled by the set_power_options command. By default, only leakage power optimization is enabled, but you can also enable dynamic power optimization for use with the psynopt command.
-optimize_dft	Reorder scan chains
-congestion	Reduce congestion to improve routeability. For best results, use this option only for congested designs.
-only_design_rule	Perform design rule fixing without performing timing optimizations. This option uses *psynopt* placement to target DRC problems. You cannot use *-only_design_rule* with *-no_design_rule* or *-only_area_recovery*.

8.2.5 Performing Pre-route RC extraction:

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

When performing pre-route RC estimation, IC Compiler performs the following tasks:

- Determine RC Coefficients
- Estimates the routing topology
- Calculates RC values
- Performs delay estimation
- Saves the timing data and attributes in the Milkyway design library.

8.2.6 Analyze placement:

After you perform placement, you can analyze the following:

- Power

To report dynamic and static power statistics, use the *report_power* command.

To display power density maps for leakage power, dynamic power, switching power, internal power, and total power, choose Power > *map_name*, where *map_name* is the name of the power density map you need to see.

- Timing

  To report timing statistics for the design, use the *report_timing* command.

Chapter 9: Clock Tree Synthesis (CTS)

9.1 Introduction:

In a synchronous digital system, the clock signal is used to define a time reference for the movement of data within that system. The **clock distribution network** (or **clock tree**, when this network forms a tree) distributes the clock signal(s) from a common point to all the elements that need it. Since this function is vital to the operation of a synchronous system, much attention has been given to the characteristics of these clock signals. Clock signals are often regarded as simple control signals; however, these signals have some very special characteristics and attributes.

Finally, the control of any differences and uncertainty in the arrival times of the clock signals can severely limit the maximum performance of the entire system and create catastrophic race conditions in which an incorrect data signal may latch within a register.

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.

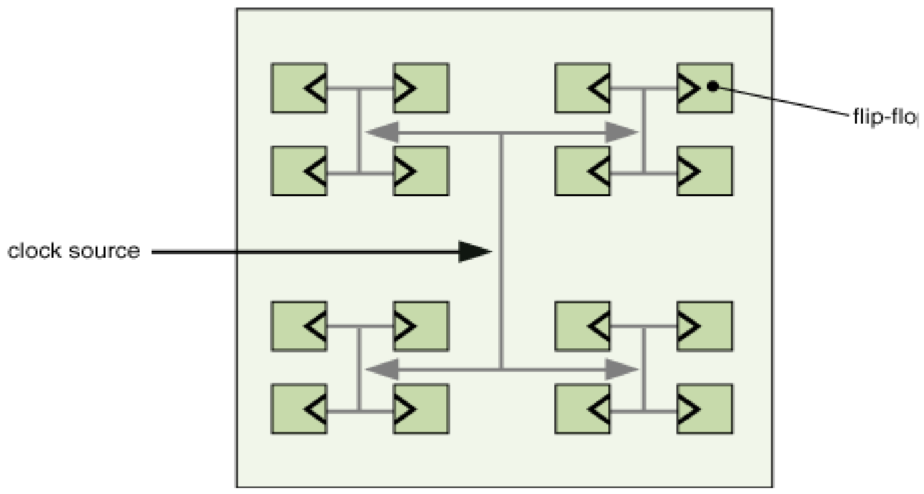


Fig. 9.1: Clock Tree Synthesis

The clock distribution network often takes a significant fraction of the power consumed by a chip. Furthermore, significant power can be wasted in transitions within blocks, even when their output is not needed. These observations have led to a power saving technique called clock gating, which involves adding logic gates to the clock distribution tree, so portions of the tree can be turned off when not needed. (When a clock can be safely gated may be determined either through automatic analysis of the circuit, or specified by the designer). The exact savings are very design dependent, but around 20-30% is often achievable.

This chapter provides detailed information about the IC Compiler clock tree synthesis flow.

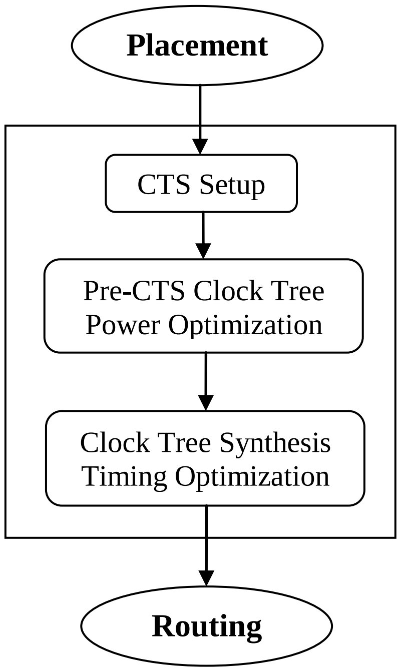

Placement
CTS Setup
Pre-CTS Clock Tree Power Optimization
Clock Tree Synthesis Timing Optimization
Routing

Fig. 9.2: Clock Tree Synthesis Flow

This chapter contains the following sections:

- CTS Setup and prerequisite checks
- Pre-CTS Clock tree and power optimization
- Clock Tree Synthesis and Timing Optimization

9.2 CTS Setup and prerequisite checks:

9.2.1 Prerequisites for Clock Tree Synthesis:

Before you run clock tree synthesis, ensure that your design and libraries meet the prerequisites described in the following sections:

➢ Design Prerequisites:

Before running clock tree synthesis, your 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 power and ground nets are pre-routed.
- High-fanout nets, such as scan enables, are synthesized with buffers.

➢ Library Prerequisites:

Before you run clock tree synthesis, your libraries must meet the following requirements:

• Any cell in the logic library that you want to use as a clock tree reference 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. To use these cells during clock tree synthesis and optimization, you can either remove the *dont_use* attribute by using the *remove_attribute* command or you can override the *dont_use* attribute by specifying the cell as a clock tree reference by using the *set_clock_tree_references* command.

• 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

• TLUPlus models must exist.

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

9.2.2 Analyzing the Clock Trees:

Before running clock tree synthesis, analyze each clock tree in your design to determine its characteristics and its relationship to other clock trees in the design. For each clock tree you have to 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 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 inter-clock 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?

9.2.3 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.
In addition to simple clock trees, IC Compiler also supports cascaded clock trees (a clock tree that contains another clock tree in its fanout). The nested clock tree can either have its own source (identified by the *create_clock* command) or be a generated clock (identified by the *create_generated_clock* command).
Use the *check_clock_tree* command to verify that your master clock sources are correctly defined.

9.2.3.1 Cascaded Clock:

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.

9.2.3.2 Cascaded Generated Clock:

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.

9.2.3.3 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.
IC Compiler defines the following clock endpoints as implicit stop pins:

- Clock pins of sequential cells
- Clock pins of macros.

- 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.
IC Compiler defines the following clock endpoints as implicit exclude pins:

- Source pins of clock trees in the fanout of another clock
- Non-clock input pins of sequential cells
- Multiplexer select pins
- Three-state enable pins
- Output ports

- Incorrectly defined clock pins (for example, the clock pin does not have trigger edge information or does not have a timing arc to the output pin)
- Buffer or inverter input pins that are held constant (by using *set_case_analysis*)
- Input pins of combinational cells or integrated clock gating cells that do not have any fanout or that do not have any enabled timing arcs.

9.2.3.4 Defining the clock root attributes:

If the clock root is an input port (without an I/O pad cell), you 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 clock tree CLK1 has input port CLK1 as its root and CLK1 is driven by cell CLKBUF, enter

icc_shell> **set_driving_cell -lib_cell *mylib/CLKBUF* [get_ports *CLK1*]**

If the clock root is an input port with an I/O pad cell, you must accurately specify the input transition delay of the input port.

icc_shell> **set_input_transition -rise *0.3* [get_ports *CLK1*]**
icc_shell**> set_input_transition -fall *0.2* [get_ports *CLK1*]**

9.2.4 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. To see the clock tree exceptions defined in your design, generate a clock tree exceptions report by running the *report_clock_tree -exceptions* command.
If you 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 as follows:

1. Nonstop pins
2. Exclude pins
3. Float pins
4. Stop pins

9.2.5 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

To define a clock tree reference list, use the *set_clock_tree_references* command. When you define a clock tree reference list, ensure that the buffers and inverters that you specify have a wide range of drive strengths, so that clock tree synthesis can select the appropriate buffer or inverter for each cluster.

```
icc_shell> set_clock_tree_references   -references {clk1a6 clk1a9 clk1a15}
```

9.2.6 Specifying Clock Tree Synthesis Options:

Some options can be applied either on all clocks (globally) or on a per-clock basis, while other options can be applied globally only. To define clock tree synthesis options, use the *set_clock_tree_options* command.
To see the current settings for the clock tree synthesis options, use the *report_clock_tree -settings* command

9.2.6.1 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 you do not specify this constraint, the clock tree synthesis default is 0.6 pF.

- Maximum transition time (*set_clock_tree_options -max_transition*)

If you do not specify this constraint, the clock tree synthesis default is 0.5 ns.

- Maximum fanout (*set_clock_tree_options -max_fanout*)

If you do not specify this constraint, the clock tree synthesis default is 2000.

You can specify the clock tree design rule constraints for a specific clock (by using the -*clock_trees* option to specify the clock) or for all clocks (by omitting the -*clock_trees* option).

9.2.6.2 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).

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

- Maximum skew (*set_clock_tree_options -target_skew*)

If you do not specify a maximum skew value, IC Compiler uses 0 as the maximum skew.

- Minimum insertion delay (*set_clock_tree_options -target_early_delay*)

If you do not specify a minimum insertion delay value, IC Compiler uses 0 as the minimum insertion delay.

You can specify the clock tree timing goals for a specific clock (by using the -*clock_trees* option to specify the clock) or for all clocks (by omitting the -*clock_trees* option).

9.2.6.3 Clock Tree Routing Options:

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

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

If you do not specify which routing rule to use for clock tree synthesis, IC Compiler uses the default routing rule (default wires) to route the clock trees. To reduce the wire delays in the clock trees, you can use wide wires instead. Wide wires are represented by non-default routing rules. Before you can use a non-default routing rule, the rule must either exist in the Milkyway design library or have been previously defined by using the *define_routing_rule* command.

icc_shell> **define_routing_rule *clk_rule* -widths *{M1 0.28 M2 0.28 M3 0.28 M4 0.28}* **

-spacings *{M1 0.28 M2 0.28 M3 0.28 M4 0.28}*

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.

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

icc_shell> **set_clock_tree_options -clock trees *CLK1* -routing_rule *clk_rule* **

-layer list {*metal4 metal5*}

9.2.6.4 Selecting the Clock Tree Synthesis Mode:

IC Compiler provides three modes for clock tree synthesis:

- Block mode (default)
- Top mode
- Logic-level balancing mode

➢ **Top Mode**

The top-mode algorithm is more aware of blockages in design. If design contains many hard macro cells or blockages or if insertion delay has a higher priority than skew, then top-mode algorithm is used .Top mode is enabled through command

icc_shell> **set_clock_tree_options -top_mode *true***

If you enable the top-mode algorithm, it applies to all clock trees. You cannot select the clock tree synthesis algorithm on a per-clock basis.

➢ **Logic-Level Balancing**

If on-chip variation is an issue for your design, use logic-level balancing mode. Logic level balancing is enabled through *set_clock_tree_options -logic_level_balance true*

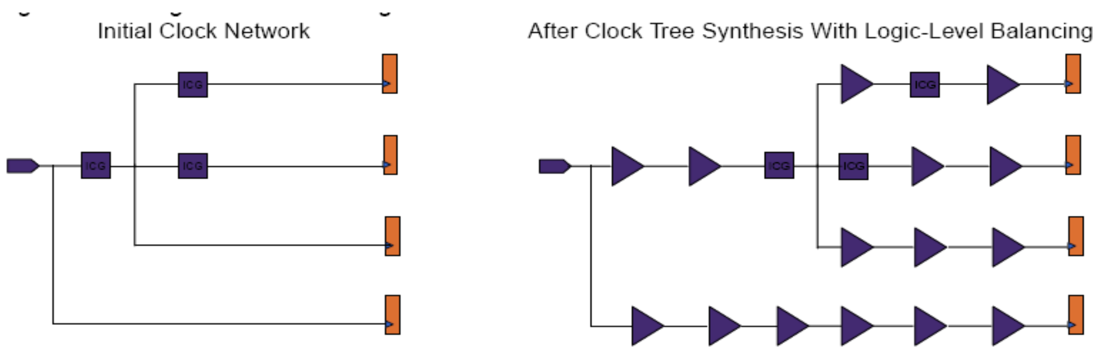


Fig 9.3: Before and after clock tree synthesis

By default, IC Compiler balances the delay in each branch of the clock tree, but it does not consider the number of logic levels in each branch. IC Compiler can take into account both delay and the number of logic levels when balancing the clock tree. This feature is called *logic-level balancing.*

To enable logic-level balancing, use the *set_clock_tree_options -logic_level_balance true* command

9.3 Pre-CTS Clock Tree Optimization:

During *clock_opt*, the clock tree optimization phase performs all the optimization tasks listed below. To set the optimization options for stand-alone pre-route clock tree optimization, specify the options when you run the *optimize_clock_tree* command.

-buffer_relocation	Optimizes the placement of the buffers and inverters in the synthesized clock trees. The default is true
-buffer_sizing	Optimizes the sizing of the buffers and inverters in the synthesized clock trees. The default is true
-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. The default is true

-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. The default is true
-gate_sizing	Optimizes the sizing of the preexisting gates in the clock trees. The default is true

Also for power optimization:

When your goal is minimizing power, you can use a large (or unlimited) maximum clock-gating fanout during insertion of the integrated clock gating cells (ICGs).

After placing your design, perform clock tree power optimization before running clock tree synthesis. To perform clock tree power optimization you can use *optimize_pre_cts_power* command. This command performs and under-the-hood check clock tree estimation. The movement of registers and data cells to reduce overall clock network capacitance is based on this estimated clock tree. Only small displacements of registers and data cells are allowed at this stage. At the end the estimated clock tree is discarded.

IC Compiler performs the following tasks during clock tree power optimization:

a. Merges integrated clock gating cells that have the same enable signal.

Only integrated clock gating cells with positive slack are merged; cells with negative slack are not merged.

b. Performs high-fanout net synthesis to fix design rule violations on the enable signals.
c. Performs power-aware placement of the integrated clock gating cells and registers

9.4 Clock Tree Synthesis and Timing Optimization:

9.4.1 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.
- Multi-voltage designs.

9.4.2 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.

```
icc_shell> check_clock_tree -clocks my_clk
```

If you do not specify the *-clocks* option, IC Compiler validates 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
- Clock (master or generated) with no sinks
- Looping clock
- Cascaded clock with an un-synthesized clock tree in its fanout
- Multiple-clocks-per-register propagation not enabled, but the design contains overlapping clocks
- Ignored clock tree exceptions
- Stop pin or float pin defined on an output pin

Each message generated by the *check_clock_tree* command has a detailed man page that describes how to fix the identified issue. Fixing these issues can improve the clock tree and timing QoR.

9.4.3 Implementing the Clock Trees:

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

By default, the *clock_opt* command uses virtual routing throughout the clock tree synthesis and optimization process to estimate the wire delay and capacitance. For better correlation with post-route timing, you can use the integrated clock global router to estimate the wire delay and capacitance during the clock tree optimization and interclock delay balancing phases. To enable the integrated clock global router for these phases, set the *cts_integrated_global_router* variable to true before running *clock_opt*.

The *clock_opt* command does the following:

- Performs clock tree power optimization **(Optional)**

To perform clock tree power optimization during the *clock_opt* process, enable physical optimization of the integrated clock gating cells and power-aware placement and use the *-power* option of the *clock_opt* command.

- Synthesizes the clock trees

Before implementing the clock trees, IC Compiler upsizes, and possible moves, the existing clock gates, which can improve the quality of results (QoR) and reduce the number of clock tree levels.

- Optimizes the clock trees

During clock tree optimization, IC Compiler uses the optimization techniques, such as buffer relocation, buffer sizing, delay insertion, gate sizing, and gate relocation, to further improve the skew.

- Performs interclock delay balancing **(Optional)**

To perform interclock delay balancing during the *clock_opt* process, define the interclock delay balancing requirements, and use the *-inter_clock_balance* option of the *clock_opt* command.

- Performs detail routing of the clock nets.
- Designer can also perform detail routing of the clock nets as a stand-alone process.
- Performs RC extraction of the clock nets and computes accurate clock arrival times.

- Adjusts the I/O timing **(Optional)**

To adjust the input and output delay based on the actual clock arrival times; use the -*update_clock_latency* option of the *clock_opt* command. IC Compiler uses the adjusted input and output delays during placement and timing optimization.

- Optimizes the scan chains **(Optional)**

To optimize the scan chains by reordering the chains to minimize the number of buffer crossings in the scan chain, use the -*optimize_dft* option of the *clock_opt* command.

- Fixes the placement of the clock tree buffers and inverters.
- Performs placement and timing optimization

If designer specify -*update_clock_latency*, IC Compiler uses the adjusted input and output delays during placement and timing optimization. IC Compiler uses propagated arrival times for all clock sinks.

- Performs power optimization **(Optional)**

Designer can perform leakage power optimization and dynamic power optimization during the *clock_opt* process by enabling the selected optimizations with the *set_power_options* command and using the -power option of the *clock_opt* command. To enable leakage power optimization, use the *set_power_options -leakage* command. To enable dynamic power optimization, use the *set_power_options -dynamic* command.

- Fixes hold time violations **(Optional)**

To fix hold time violations during the *clock_opt* process, use the -*fix_hold_all_clocks* option of the *clock_opt* command

Summary of the most important options that can be used with *clock_opt* command:

-area_recovery	Enables area recovery during placement and timing optimization.
-only_cts	Do not perform extraction, optimization, or hold time fixing
-only_psyn	Do not perform initial gate upsizing, clock tree synthesis, or clock tree optimization.

-no_clock_route	Do not perform clock routing.
-optimize_dft	Enables scan chain optimization.
-power	Perform the power optimizations enabled by *set_power_options*.
-inter_clock_balance	Enables interclock delay balancing.
-fix_hold_all_clocks	Fix hold time violations.

9.5 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.
- Verifying the placement of the clock instances

If the clock trees meet your requirements, you are 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. Set the *cts_use_debug_mode* variable to true before running clock tree synthesis to output the following additional information:

- User-defined design rule constraints (maximum transition time, maximum capacitance, and maximum fanout)
- User-defined clock tree timing constraints (skew and insertion delay)
- User-defined level restrictions (maximum level count)
- Clustering targets set by IC Compiler (maximum transition time and maximum capacitance)

9.5.1 Generating Clock Tree Reports:

To generate clock tree reports, use the *report_clock_tree* command. To limit the report to specific clock trees, use the -*clock_trees*. To generate the report for a specific operating condition, use the -*operating_condition* option.

These are some other options for *report_clock_tree* command:

-drc_violators	Lists the design rule violations that occur in the clock trees up to the endpoints, except for violations on don't buffer nets, nets inside interface logic models, nets connected to boundary cells or pad cells.
-all_drc_violators	Lists all design rule violations that occur in the clock trees up to the default sinks
-exceptions	Lists the stop pins, exclude pins, float pins, nonstop pins, don't touch subtrees, don't buffer nets, and don't size cells.
-summary	Reports the global clock skew summary.
-settings	Lists the following global clock tree settings: clock tree synthesis algorithm, logic-level balancing, clock tree design rule constraints, clock tree configuration files non-default routing rules, on-chip-variation-aware clustering. Lists the following per-clock tree settings: clock tree timing constraints, clock tree design rule constraints, maximum buffer levels, clock tree optimizations

9.5.2 Analyzing Clock Tree Timing:

The timing characteristics of the clock network are important in any high-performance design. To obtain detailed information on the clock networks in a design, use the *report_clock_timing* command. This command reports the clock latency, transition time, and skews characteristics at specified clock pins of sequential elements in the network.

In the *report_clock_timing* command, use the -*type* option to specify the type of report you want, the scope of the design to analyze, and any desired filtering or ordering options for the report.

9.5.3 Verifying the placement of the Clock Instances:

IC Compiler provides several GUI windows that you can use to analyze the clock trees:

- Layout window

The clock tree visual modes allow you to view the structure or timing distribution of the clock trees in the layout window.

- Interactive clock tree synthesis window

The interactive clock tree synthesis window provides information about all the clocks defined in your design.

- Clock tree option viewer

The clock tree option viewer displays the current settings of the clock tree synthesis options.

- Clock tree hierarchy browser

The clock tree hierarchy browser provides a hierarchical listing of the clock tree.

- Fanin and Fanout schematics
- Clock arrival histogram

After you are satisfied with the clock tree synthesis results, analyze the QoR of the entire design by reporting on constraints by using the *report_constraint* command

Use the reports to check the following parameters:

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

Chapter 10: Routing

10.1 Introduction:

Routing is a crucial step in the design of integrated circuits. It builds on a preceding step, called placement and clock tree synthesis, which determines the location of each active element of an IC and connects the clock to each sequential element in the design. The routing step adds wires needed to properly connect the placed components while obeying all design rules for the IC.

The task of all routers is the same. They are given some pre-existing polygons consisting of pins (also called terminals) on cells, and optionally some pre-existing wiring called pre-routes. Each of these polygons is associated with a net. The primary task of the router is to create geometries such that all terminals assigned to the same net are connected, no terminals assigned to different nets are connected, and all design rules are obeyed.

Before designer can perform routing, design must meet the following conditions:

- Power and ground nets have been routed after design planning and before placement.
- Clock tree synthesis and optimization have been performed.
- Estimated congestion is acceptable.
- Estimated timing is acceptable (about 0 ns of slack).
- Estimated maximum capacitance and transition have no violations.

10.2 Checking Routeability:

After placement is completed, designer can have IC Compiler check whether 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 cell in design (top_cell_name.err), with a list of violations designer should correct before performing detail routing. To verify that design is ready for detail routing, use the *check_routeability* command

After you run the *check_routeability* command, you can use the *report_error_coordinates* command to report the location of each error. You can also use the error browser to examine the errors in the GUI.

10.3 Types of Zroute Routing Operations:

10.3.1 Global Routing (GR):

Global routing maps general pathways through the design for each un-routed net (signal nets and clock nets). The global router uses a two-dimensional array of global routing cells to model the demand and capacity of global routing. IC Compiler uses the average height of the standard cells to create the height and width of each global routing cell.

During global routing, IC Compiler assigns nets to the global routing cells through which they pass. For each global routing cell, the routing capacity is calculated according to the blockages, pins, and routing tracks inside the cell. Although the nets are not assigned to the actual wire tracks during global routing, the number of nets assigned to each global routing cell is noted. IC Compiler calculates the demand for wire tracks in each global routing cell and reports the overflows, which are the number of wire tracks still needed after IC Compiler assigns nets to the available wire tracks in a global routing cell.

Zroute Global routing option should be set to true first using command *set_route_zrt_global_options true*

IC Compiler might reduce overflows by detouring nets around congested areas and increasing the wire length. Designer can examine the global routing report that appears in the command window and display congestion maps to help designer decide whether design can be routed. The global router considers spacing and wide-wire variable routing rules, as well as shielding variable routing rules, when calculating congestion.

And routing can be done by using the *route_opt -initial_route_only –stage global* command.

10.3.2 Track Assignment:

Before you perform detail routing, run track assignment to specify which tracks within each global routing cell are to be used for each net. Track assignment operates on the entire design at once; it can make long routes straight and reduce the number of vias, whereas the detail routing routes a small area at a time.

After track assignment finishes, all nets are routed, but not very carefully. There are many violations, particularly where the routing connects to pins. Detail routing works to correct those violations.

Zroute track assignment routing option should be set to true first using command *set_route_zrt_track_options true*

And routing can be done using the *route_opt –initial_route_only –stage track* command.

When you specify initial routing only and the track assignment stage, IC Compiler runs global routing and track assignment.

10.3.3 Detail Routing:

Detail routing uses the general pathways suggested by global routing and track assignment to route the nets (paths and contacts). During search-and-repair routing passes, IC Compiler searches for DRC violations and reroutes wires in an effort to avoid violations. IC Compiler does not add metal stubs on frozen nets to fix DRC violations, even when the nets have DRC violations.

Zroute detail routing option should be set to true first using command *set_route_zrt_detail_options true*

And routing can be done using the *route_opt –initial_route_only –stage detail* command.

10.3.4 ECO Routing:

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

To perform ECO routing,

- Specify the ECO options as follows:
  - ➢ Run

Select the routing operations (“Global route,” “Track assignment,” and “Detail routing”) to be performed in sequence on the modified nets.

If designer need to use detail routing first to connect broken or new nets, select Auto and specify the region-based nets. If detail routing fails to connect those nets because the scope of change is too large for detail routing, IC Compiler uses global routing to connect only the broken nets and then performs track assignment or detail routing to complete the ECO process.

- Scope

To connect or route the modified nets and fix DRC violations, select Global.
To connect or route the modified nets without fixing DRC violations, select Local.

- Maximum Search & Repair cycles

Specify the maximum number of search-and-repair cycles to allow.

- Freeze routing on layers

If designer need to prevent ECO routing changes on any layers, select those layers from this list of layers.
If designer need to also prevent ECO changes to the vias on the selected layers, select “Freeze vias on frozen metal.”

- Reroute nets

Indicate the types of nets to reroute.

- To utilize dangling wires during ECO route, select the “Utilize dangling wires” check box.

Alternatively, designer can use the *route_zrt_eco* command

10.4 Routing Setting Up:

10.4.1 Setting Up Routing Options:

You can specify global routing, track assignment, detail routing, and other routing options for IC Compiler to use whenever you perform routing functions.
This can be done by using the *set_route_options* command.
To report the settings, use the *report_route_options* command.

10.4.2 Setting Route Guides:

You can create or remove route guides that do the following for a specific area of your design:

- Prevent routing for signals or pre-routed nets
- Change the wiring direction
- Control wiring density
- Fix violations

Specify the route guide characteristics.

- Specify the route guide by typing its name in the Name box.
- To repair violations, select the “Repair as single SBox” check box.
- To prevent the routing of signal wires on specific layers, select the “No signal route on layers” check box and select the layers in the list below the check box.
- To prevent the routing of wires within a certain distance of the route guide (where there is a keepout margin), do not select the “Zero min-spacing” check box.
- To prevent automatic pre-routes on layers, select the “No automatic pre-routes on layers” check box and select the layers in the list below the check box.
- To specify the allowable horizontal and vertical track utilizations, select the “Route track utilization” check box and specify the horizontal and vertical track utilization percentages.
- To switch the preferred wiring direction, select the “Switch preferred direction” check box.
- Specify the coordinates for the route guide by typing the coordinates in the dialog box or drawing the rectangle in the layout view.
- Also, specify that the route guide should snap to the minimum grid, placement site, routing track (the default), middle routing track, or user grid.
- Alternatively, this can be done by using *create_route_guide* command with appropriate options as follows:

Find route guides that meet the conditions of an expression	-filter expression

Find route guides that are totally within a rectangular area	-within x1 y1 x2 y2
Find route guides that touch or cross over the border of a rectangular area	-touch x1 y1 x2 y2

To remove route guides, Use the *remove_route_guides* command

10.4.3 Setting the Preferred routing direction:

Use the *set_preferred_routing_direction* command to reset and override the default preferred routing direction specified in the library, or the design, for a specific layer.
The layer direction set with this command applies to the current design only.
The syntax is
set_preferred_routing_direction -layers *list_of_layers* -direction *horizontal* | *vertical*
Use the *remove_preferred_routing_direction* command to remove the user-defined directions for a specific layer from the design. The syntax is
remove_preferred_routing_direction -layers *list_of_layers*

10.4.4 Specifying Net Layer Constraints:

IC Compiler lets you specify which layers can be ignored for routing and which layers are to be ignored for RC and congestion estimation. If you do not specify layers to ignore for RC and congestion estimation, those functions use the layers you specify for routing. You can specify these layers in the following ways:

- Specify a minimum routing layer

It sets that layer and the layers above it for routing. If you don't specify layers to ignore for RC and congestion estimation, those functions use all of the routing layers.

icc_shell> **set_ignored_layers -min_routing_layer *layer_name***

- Specify a maximum routing layer

It sets that layer and all layers below it for routing. If you don't specify layers to ignore for RC and congestion estimation, those functions use all of the routing layers.

icc_shell> **set_ignored_layers -max_routing_layer *layer_name***

- Specify minimum and maximum routing layers

Sets the layers you specify and all layers between them for routing. If you don't specify layers to ignore for RC estimation and congestion, those functions use all of the routing layers.

```
icc_shell> set_ignored_layers -min_routing_layer layer_name \
                    -max_routing_layer layer_name
```

- Specify the layers to ignore for RC and congestion estimation

```
icc_shell> set_ignored_layers   -rc_congestion_ignored_layers layer_name
```

10.4.5 Setting Routing Types:

For debugging, you can set or change the routing types of wires, vias, or paths to indicate which IC Compiler routing engine is to route them.

10.5 Routing Clock Nets:

You can pre-route a group of nets, such as clock nets, before routing the rest of the nets in the design. Taking this step can help you find timing problems. For example, you can route clock and bus pins to pre-routed clock and bus wires and then perform timing analysis on these nets. If the timing results are satisfactory, you can route the remaining nets.
This can be done using the *route_zrt_group* command.

10.6 Setting the Routing and Optimization Strategy:

IC Compiler provides strategy controls that you set to guide how routing and post-route optimizations are run. Before you apply the strategy settings, make sure you enable hold fixing by using the *set_fix_hold* command.
Specify the following options or keep defaults:

- Maximum number of crosstalk reduction optimization cycles

By default, IC Compiler performs up to two cycles. To change the maximum number of cycles, specify a number in the "Maximum crosstalk reduction cycles" box.

- Maximum number of initial routing search-and-repair cycles

By default, IC Compiler performs as many as 10 search-and-repair cycles. To change the maximum number of cycles, specify a number in the “Maximum initial route Search and Repair cycles” box.

- Maximum number of wire and via optimization search-and-repair cycles

By default, IC Compiler performs up to five search-and-repair cycles for wire and via optimizations. To change the maximum number of cycles, specify a number in the “Maximum optimize wire Search and Repair cycles” box.

- Maximum number of ECO search-and-repair cycles

By default, IC Compiler performs up to five search-and-repair cycles. To change the maximum number of cycles, specify a number in the “Maximum ECO Search and Repair cycles” box.

- Threshold number of routing violations

By default, IC Compiler allows as many as 3,000 routing violations before limiting search-and-repair to one cycle. To change the threshold, specify a number in the “Route violations threshold to trigger the reduction Search and Repair loop” box.

- Runtime limitation

By default, IC Compiler does not have a runtime limitation for wire and via optimization (this optimization decreases wire length and the number of vias). To limit the time spent on this runtime, specify the number of minutes for the limitation in the “Run time limit” box.
Alternatively, you can use the *set_route_opt_strategy* command.
To report the settings, use the *report_route_opt_strategy* command.

10.7 Routing and Post-route Optimization of Signal Nets:

After you have specified your routing options and routing and post-route optimization strategy, you are ready to perform routing and post-route optimization of your design.

The routing and post-route optimization command runs global routing in timing-driven mode and track assignment in crosstalk-prevention mode. These modes automatically enable the density-driven mode for global routing and track assignment.

By default, IC Compiler performs optimization after all the routing steps (global routing, track assignment, and detail routing) are completed, but you can have it occur after global routing, track assignment, or detail routing.

Also By default, IC Compiler does not perform signal integrity optimization ("No optimization" radio button). You can perform crosstalk reduction and signal integrity optimization in addition to routing (by selecting the "Crosstalk reduction and SI optimization" radio button) or only crosstalk reduction and signal integrity optimization

To perform incremental optimization steps, select "Incremental mode." You can select "All incremental mode optimization steps" or select from the following options to perform individual optimization steps:

- Timing optimization only with wire sizing only
- Hold timing optimization only. You must first enable hold fixing by using the *set_fix_hold* command.
- Area recovery only
- Fix design rules only
- Power recovery only

These can be done using command *route_opt* with some options as follows:

-*effort low* | *medium* | *high*	Change the optimization effort level
-*stage global* |*track* | *detail*	Specify the routing stage after which optimization occurs
-*power*	Perform leakage power optimization
-*xtalk_reduction*	Perform crosstalk reduction optimization in addition to signal integrity optimization
-*initial_route_only*	Perform only initial routing without post-route optimization (this option works in conjunction with the -stage option)
-*skip_initial_route*	Perform post-route optimization without routing
-*optimize_wire_via*	Optimize wires and vias (this option decreases wire length and number of vias)

-*area_recovery*	Recover area for cells not on critical paths
-*incremental*	Perform incremental post-route optimization
-*wire_size*	Determine whether wire sizing is used to fix setup time violations
-*incremental -only_wire_size*	Perform only wire sizing to resolve setup time violations during incremental mode
-*incremental -only_hold_time*	Resolve only hold time violations during incremental mode (use the *set_fix_hold* command to enable hold time fixing)
-*incremental -only_area_recovery*	Perform only area recovery during incremental Mode
-*incremental -only_design_rule*	Perform only logical design rule fixing during incremental mode
-*incremental only_power_recovery*	Perform only power recovery during incremental mode

10.8 Reporting Cell Placement and Routing Characteristics:

You can report on cell placement and routing statistics to help you determine whether to perform further optimizations to improve timing. For example, if the statistics show that an area has high congestion, an additional optimization might not have room to add or size up standard cells.
This can be done using the *report_design -physical* command.

10.9 Verifying the Routing:

After you have completed the routing, you can check it for DRC violations and unconnected nets by using the *verify_zrt_route* command.
By default, the routing is verified for the entire design. To verify the routing only in a specific region, specify the coordinates of the region by using the -*bounding_box* option or by specifying the coordinates in the “DRC Area” area in the GUI.

10.10 Post-route RC Extraction:

IC Compiler automatically performs post-route RC estimation when you run the *route_opt* command and when you run 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, you can explicitly perform post-route RC extraction by running the *extract_rc* command.

Chapter 11: Chip Finishing and Design for Manufacturing

11.1 Introduction:

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

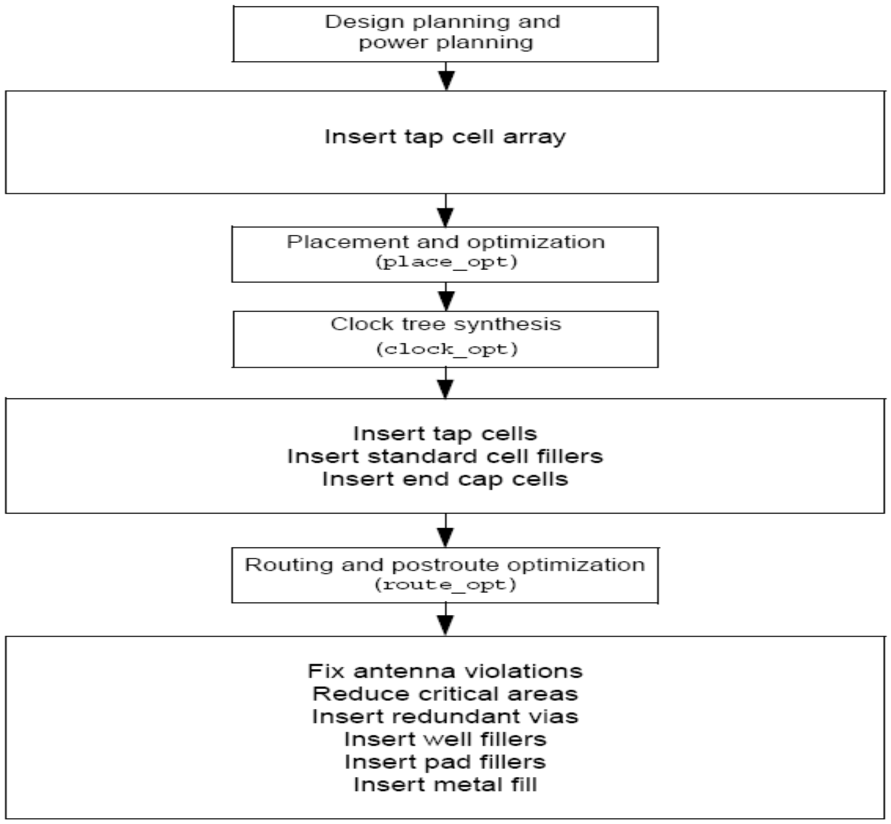

Design planning and power planning
Insert tap cell array
Placement and optimization (place_opt)
Clock tree synthesis (clock_opt)
Insert tap cells
Insert standard cell fillers
Insert end cap cells
Routing and postroute optimization (route_opt)
Fix antenna violations
Reduce critical areas
Insert redundant vias
Insert well fillers
Insert pad fillers
Insert metal fill

Fig 11.1: Chip Finishing and Manufacturing 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.

You can insert tap cells in your design before or after placement:

- You can insert tap cell arrays before placement to ensure that the placement complies with the maximum diffusion-to-tap limit.
- You can insert them after placement to fix maximum diffusion-to-tap violations.

11.3 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.

You can insert diodes into the design to fix antenna violations. Antenna fixing is a two-step process. First, IC Compiler either runs its internal antenna checker or uses the Hercules tool output to identify the antenna violations. Next, a diode cell (or multiple diodes when the antenna ratio requires the protection of more than one diode) is placed for each violation.

To minimize the impact on the existing placement and routing, you can choose to freeze the existing cell placement and the routing. In this case, the diode cells can be placed in existing empty spaces only. You can also choose to complete the routing of the diode cells during the insertion process when the cell placement is already frozen. Whenever possible, IC Compiler ties the diode cells to the existing wires without ripping up and rerouting the entire net.

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

11.4 Reducing Critical Areas:

A critical area is a region where circuit failure will occur (yield loss) if the center of a random defect falls in it. Critical area regions vary with defect size. For a particular layout, the larger the defect size, the larger the critical area. Larger defects are statistically less likely. A conductive defect causes a short fault, and a nonconductive defect causes an open fault.

This flow has the following steps:

- Reporting the Critical Area using command

```
report_critical_area   -particle_distr_func_file
                       -input_layers layer_list
                       -fault_type {short | open}
```

- Display Critical Area map
- Perform wire spreading to reduce short faults using command

```
spread_zrt_wires       -timing_preserve_setup_slack_threshold
                       - timing_preserve_hold_slack_threshold
```

- Perform wire widening to reduce open faults using command

```
widen_zrt_wires        -timing_preserve_setup_slack_threshold
                       - timing_preserve_hold_slack_threshold
```

11.5 Insert Redundant Vias:

After routing and post-routing optimization, you can replace single-cut vias with multiple-cut via arrays (a process sometimes called via doubling) or with a different via with the same layers. Via is replaced only if the via array or the new via does not introduce DRC violations. Some vias may be better for DFM, while others are better for routeability.

This can be done using commands:

```
insert_zrt_redundant_vias   -list_only
```

If changes are required, user has to define new redundant via rules

define_zrt_redundant_vias -from_via -to_via -to_via_x_size -to_via_y_size

Also *–to_via_weights* option can be used to set a priority. Weight is 1 to 10, higher weight via will be tried first. With equal weights, prioritization is based on routeability.

11.6 Inserting Filler Cells:

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

After placement (and clock tree synthesis, if applicable) is complete, you can fill empty spaces in the standard cell rows with instances of master filler cells in the library, to make sure all power nets are connected. Multi-height filler cells are allowed.

Filler cells with metal are inserted only if no DRC violations result using command

insert_stdcell_filler -cell_with_metal "…." –connect_to_power VSS –connect_to_ground VSS –between_std_cells_only

Filler cells without metal are inserted without checking DRC's next using command

insert_stdcell_filler -cell_without_metal "…." –connect_to_power VSS –connect_to_ground VSS –between_std_cells_only

You should insert cell with metal first then follow with cell without metal.

11.7 Performing Metal Density Filling:

After routing and before layout versus schematic (LVS) connectivity verification, design rule checking (DRC), and layout parasitic extraction, you can fill the empty tracks in your design with metal wires, to meet the metal density rules required by most fabrication processes.

When you define minimum and maximum metal density rules in the technology file, IC Compiler tries to create fills within the specified ranges.

This can be done using *insert_metal_filler* command with option *–timing_driven* to preserve timing on critical nets.

11.8 Performing Notch and Gap filling:

After routing is complete, you can fill notches and gaps that are smaller than the minimum distance limit between objects of the same net on the same layer. The generated notch-and-gap-filling information is stored in the FILL view cell and can be used when you translate your design data to GDSII format.
This can be done using *insesrt_ng_filler* command.

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. This chapter describes how to use IC Compiler to analyze and correct signal integrity problems.
The following phenomena 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 or static noise.

- Signal electro-migration

Signal electro-migration is an increase in current density caused by the use of smaller line widths and higher operational speeds in IC designs. Electro-migration can lead to shorts or opens due to ion displacement caused by the flow of electrons.

IC Compiler signal integrity analysis and optimization supports multi-voltage and multimode-multicorner designs.

11.10 Final Validation:

This is the last stage in the back end. There are some files produced in this stage that are used as input for other tools for more analysis:

- Parasitics (SPEF or SBPF)

This is a file that includes Wire parasitic for PrimeTime that is provided by either any of these files .SPEF or .SBPF
This can be done using command

write_parasitics –output <file_name> -format <SPEF | SBPF> -compress –no_name_mapping

- Netlist Output

Netlists for STA (Static Timing Analysis) do not require output of "Physical only cells" like:

  - ➢ Corner pad cells
  - ➢ Pad/Core filler cells
  - ➢ Unconnected cell instances

Unconnected cell instances (e.g.: spare cells) are needed for LVS
This can be done using command
write_verilog –no_corner_pad_cells …… final.v

- GDSII Output:

GDSII output file that is for external physical verification can be generated from IC Compiler.

This output file requires "physical only cells" like:

  - ➢ Corner pad cells
  - ➢ Pad/Core filler cells
  - ➢ Unconnected cell instances

This can be done using command
write_stream –cells DFM_clean   file_name.gdsii

Chapter 12: Analysis of Results and Conclusion

12.1 Functional Verification:

This project design is based on designing an asynchronous interface for n-bit wide data. This is implemented by designing a parameterized asynchronous FIFO, in which the depth and width of the FIFO can be changed easily by setting the required parameter for them.

The designing strategy was to design all the sub-modules (mem_blk.v , rptr_ctrl.v, wptr_crl.v) and then instantiate them all in the top level module (FIFO.v).

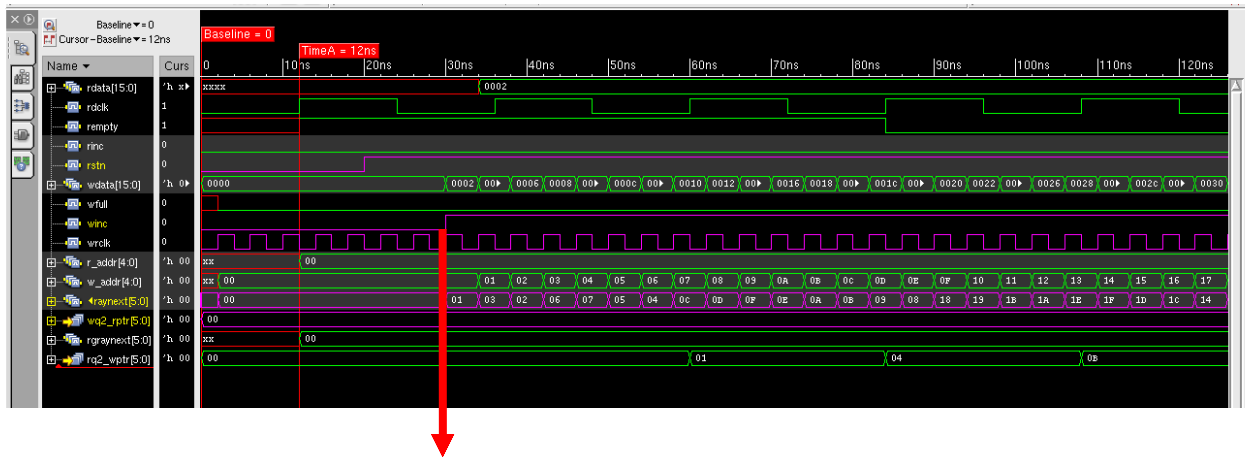
Then a test bench should be created to test the design if it works as it is designed for or not.

A functional/verification simulation is used to test the design if it works properly or no.

By running the UNIX$ ncverilog –f run.f command, the simulation window popped up and design can be verified, there are many stages that will be functionally examined to get sure the design works properly:

12.1.1 Set write enable while read enable is disabled:

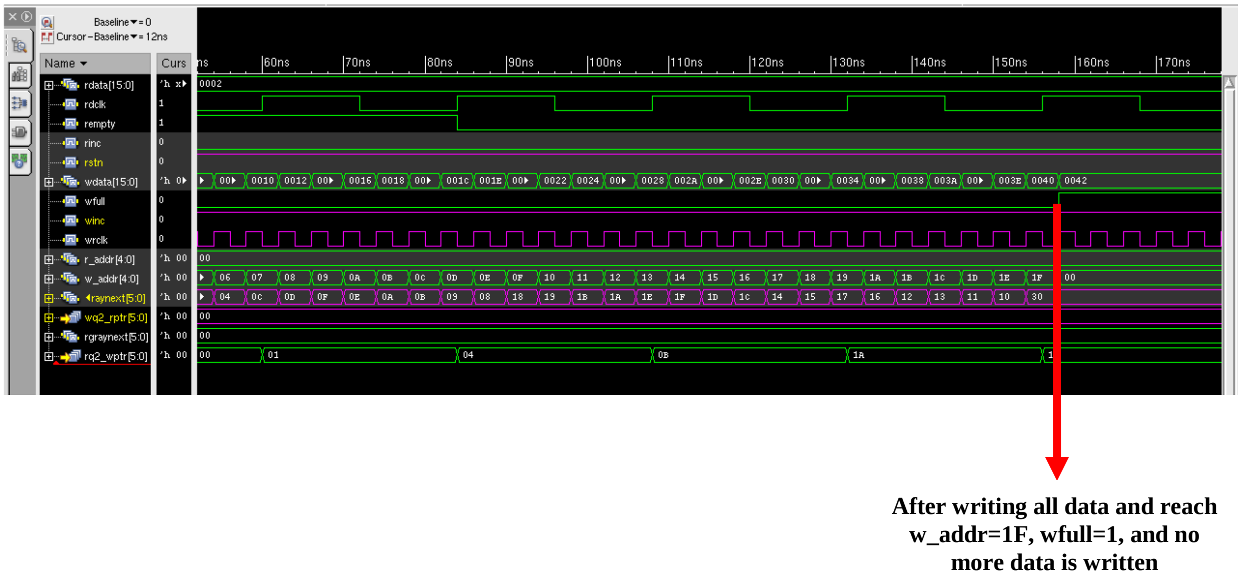
The objective is to test that the writing process is synchronous with the wrclk and enable works properly, winc =1 and rinc =0.



When winc=1, first word is written in addr 01

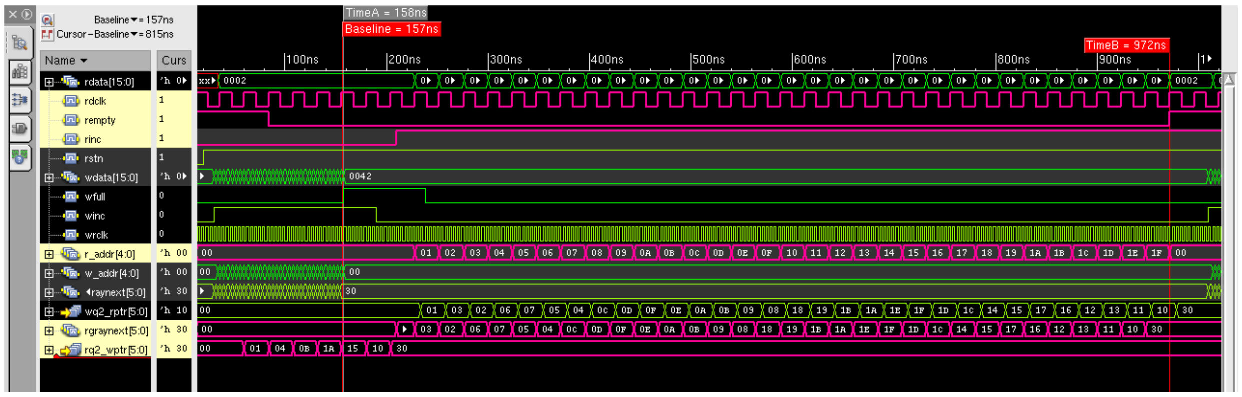
12.1.2 Test the full flag assertion and overflow:

The objective is to test if the full flag will be asserted properly or no.
After writing all the data, the write address wraps to 00 (beginning address of FIFO), while comparing wgraynext (x”30”) and wq2_rptr(x” 00”) ..
Since the 2MSB are opposite, full flag sets.
The other objective is to check the overflow, although the write enable is still ON , FIFO doesn’t write any more data, which means there is no over flow



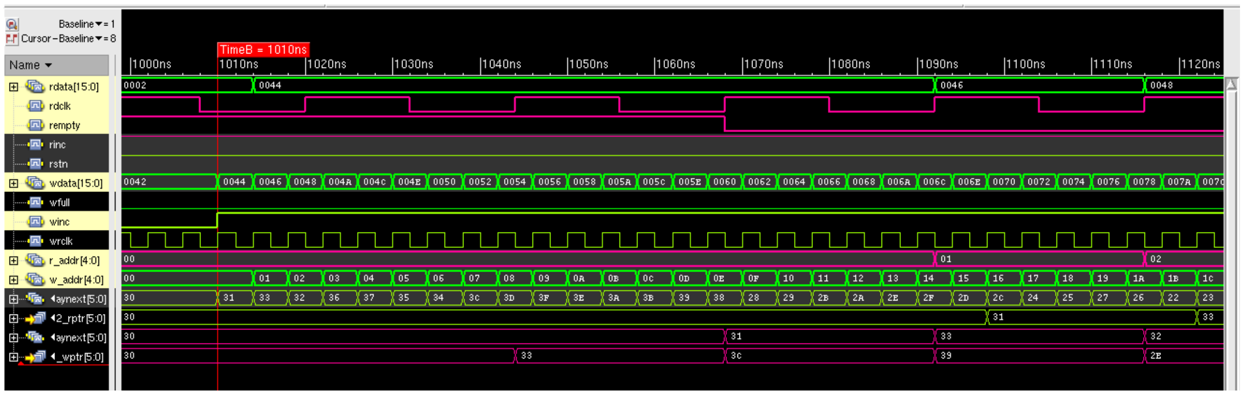
12.1.3 Test the empty flag assertion:

As shown in the waveform, after reaching the last word location (1F), comparison occurred between read pointer and synchronized write pointer. Empty flag is set when both pointers are of equal bits.

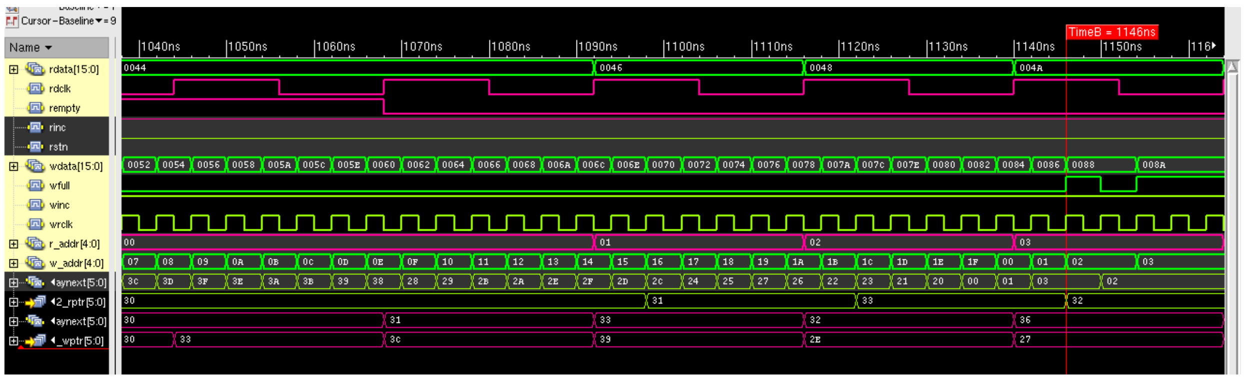


12.1.4 Enabling both read and write enables:

As shown in the waveform, when winc is enabled and since full flag is de-asserted FIFO starts to write, but since empty flag takes a few cycles to de-assert, therefore FIFO starts to read on the next positive edge of the rdclk.



This will be illustrated in the next waveform fig. , in which full flag will be asserted after writing two more data words to the FIFO.



12.2 Design Compiler Testing Strategy: (Front End)

The strategy in this project is based on a comparative study; the comparative study is being done on this design to observe the effect of low power techniques implemented on the design in the front end using Synopsys Design Compiler and back end using Synopsys IC Compiler.

Firstly we will synthesize the design without implementing any scan chains or low power techniques.

Then we will synthesize the design with scan chain insertion, to observe the expanding in area and power consumption which is normal.

Lastly we will implement the Low Power Techniques on the design that has scan chain insertion, because as explained before in chapter 5 which stated that by setting the clock gating style command to latch, an input port is created which must be hooked to the scan enable signal or test mode.

12.3 Design Compiler Synthesis Analysis:

Design	**Area**	**Power**	**Slack**
FIFO design	6794	Leakage: 36.32 uw	Rdclk: 0.09 (Met)
		Dynamic: 420.03 uw	Wrclk: 0.02 (Met)
FIFO w/ scan chain insertion	8264	**Leakage: 39.0974 uw**	Rdclk: 0.01 (Met)
		Dynamic: 2.1632 mw	Wrclk: 0.00 (Met)
FIFO w/ scan chain insertion and low power techniques	7111	**Leakage: 34.29 uw**	Rdclk: 0.01 (Met)
		Dynamic: 2.0975 mw	Wrclk: 0.00 (Met)

Table 12.1: Design Compiler Analysis

As shown in the table 12.1, by implementing the scan chain insertion for DFT, area and power consumption goes higher for adding more standard cells (multiplexed FF) which increases the power consumption of the whole design.

And by implementing low power techniques which are practically done using clock gating style, it is observed according to the data above that power consumption has been slightly reduced, also timing is better.

12.4 IC Compiler Testing Strategy: (Back End)

The testing strategy of the back end is based on the IC Compiler basics which have been explained in the previous chapters. According to the IC Compiler design flow; the netlist that has been created from the Design Compiler (front end) is loaded to the IC Compiler to be the input file that has the design to be implemented in IC Compiler which is either in (.v or .ddc) format.

The first two steps in the IC Compiler design flow which is the data setup and design planning of the design, has been done by loading the netlist of the FIFO design that has been created from the front end synthesis part which doesn’t include the scan chain insertion and the low power techniques.

The testing strategy will be based on a comparative study; firstly, we will complete all the IC Compiler till the chip finishing stage with the primary netlist that has been loaded in the data setup step.

Secondly, after finishing the first two steps of the IC Compiler design flow (data setup and design planning), and when approaching the placement stage; the SAIF file will be read into the design to load the switching activity information so low power techniques can be implemented.

And then I will complete all the IC Compiler stages till chip finishing stage, and then compare all the results.

References:

1- IC Compiler User Guide Version A-2007.12-SP2, March 2008. [Dec 2011]

2- Himanshu Bhatnagar, "Advanced ASIC chip Synthesis Using Synopsys Design Compiler, Physical Compiler and Primetime" 2004. [Jan 2012]

3- Design Compiler User Guide Version A-2007.12, December 2007 [Nov 2011]

4- Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, kaijian Shi, " Low Power Methodology Manual" [Dec 2011]

5- IC Compiler 1 workshop lab guide Version 2010.12-SP2 [Jan 2012]

6- IC Compiler 1 Manual by synopsys Version 2010.12-SP2 [Nov 2011]

7- Hima Bindu Kommuru, Hamid Mahmoodi, "ASIC Design Flow Tutorial Using Synopsys Tools" , Nano-Electronics & Computing Research Lab, School of Engineering, San Francisco State University San Francisco, CA, Spring 2009 [Jan 2012]

8- "A VHDL Package Implementation of Boundary Scan", ECE 623 [Oct 2011]

9- Clifford E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," SNUG San Jose 2002 - www.csee.umbc.edu/~tinoosh/cmpe415/tutorials/FIFO.pdf [Oct 2011]

10- http://www.asic-world.com/examples/verilog/asyn_fifo.html [Oct 2011]

11- http://asic-soc.blogspot.com/2007/11/asynchronous-fifo-design.html [Oct 2011]

12- Ronald W.Mehler "ECE 527 Lecture Notes" December 2011 [Sep 2011]

Appendix A: (Source Code)

A.1. mem_blk.v:

```
/* dual port RAM module for FIFO */

`timescale 1 ns / 1 ns

module mem_blk (clk    ,        //clk Input
                wr_addr  ,      // write address Input
                wr_data  ,      // write data
                wr_en     ,     // Write Enable
                rd_addr  ,      // read address Input
                rd_data );      // read data

parameter f_width = 16,         //width of FIFO
          f_depth = 32,         //Depth of FIFO
          f_cwidth = 5;         //width of address Counter

input clk,wr_en;
input [f_cwidth-1 :0] wr_addr,rd_addr;
input [f_width-1 :0]  wr_data;
output [f_width-1 :0] rd_data;

wire wr_en;
wire  [f_width-1 :0] rd_data;
reg  [f_width-1 :0] mem_data [f_depth-1 :0];

// ---------------------------------------------------------------
// # Write Functional Section
// ---------------------------------------------------------------
always @ (posedge clk) begin
          if (wr_en)
                mem_data [wr_addr] <= wr_data;
          else
                mem_data [wr_addr] <= mem_data [wr_addr];
end

//---------------------------------------------------------------
//# Read Functional Section
//---------------------------------------------------------------
assign rd_data = mem_data [rd_addr];

endmodule
```

A.2. rptr_ctrl.v:

/* read_pointer and empty flag control circuit*/

```
`timescale 1 ns/ 1 ns
module rptr_ctrl (r_addr,       //read address
                  r_ptr  ,      //read pointer
                  rempty_ptr,   //empty flag
                  r_inc  ,      //read increment signal
                  rdclk  ,      //read clock Input
                  rstn   ,      //read reset Input
                  rq2_wptr );   //synchronized write to read pointer


parameter fcwidth = 5;

input [fcwidth :0] rq2_wptr;
input      r_inc;
input      rdclk;
input      rstn;
output reg       rempty_ptr;
output [fcwidth-1:0]   r_addr;
output reg [fcwidth :0] r_ptr;

reg [fcwidth:0] rbin;
wire [fcwidth:0] rgraynext, rbinnext;
reg rempty_val;

//-------------------------------
// GRAYSTYLE2 pointer
//-------------------------------
always @(posedge rdclk or negedge rstn)begin
if (!rstn) begin
           rbin <= 0;
           r_ptr <= 0;
           end
else
           {rbin, r_ptr} <= {rbinnext, rgraynext};
end

assign r_addr = rbin[fcwidth-1:0];
assign rbinnext = rbin + (r_inc & ~rempty_ptr);
assign rgraynext = (rbinnext>>1) ^ rbinnext;

//--------------------------------------------------------------
// FIFO empty when the next rptr == synchronized wptr or on reset
//--------------------------------------------------------------
```

```
always @ ( rgraynext or rq2_wptr)begin
if (rgraynext[fcwidth:0] == rq2_wptr[fcwidth:0])
            rempty_val = 1'b1;
else if(rgraynext[fcwidth:0] != rq2_wptr[fcwidth:0])
            rempty_val = 1'b0;
end

always @(posedge rdclk or negedge rstn) begin
if (!rstn) rempty_ptr <= 1'b1;
else rempty_ptr <= rempty_val;
end

endmodule
```

A.3. <u>wptr_ctrl.v:</u>

```
/* write_pointer and full flag control circuit*/

`timescale 1 ns/ 1 ns

module wptr_ctrl (w_addr      ,//write address
                  w_ptr       ,//write pointer
                  wfull_ptr   ,//full flag
                  w_inc       ,//write increment signal
                  wrclk       ,//write clock Input
                  rstn        ,//write reset Input
                  wq2_rptr );  //synchronized write to read pointer

parameter fc_width = 5;

input [fc_width :0]     wq2_rptr;
input      w_inc;
input      wrclk;
input      rstn;
output reg      wfull_ptr;
output [fc_width-1:0] w_addr;
output reg [fc_width :0] w_ptr;

reg [fc_width:0] wbin;
wire [fc_width:0] wgraynext, wbinnext;
wire wfull_val;

//--------------------------------
// GRAYSTYLE2 pointer
//--------------------------------
always @(posedge wrclk or negedge rstn)begin
```

```
if (!rstn)
            {wbin,w_ptr} <= 0;
else
            {wbin,w_ptr} <= {wbinnext, wgraynext};
end

assign w_addr = wbin[fc_width-1:0];
assign wbinnext = wbin + (w_inc & ~wfull_ptr);
assign wgraynext = (wbinnext>>1) ^ wbinnext;
```

//---
// Simplified version of the three necessary full-tests:
// assign wfull_val=((wgnext[ADDRSIZE] !=wq2_rptr[ADDRSIZE]) &&
// (wgnext[ADDRSIZE-1] !=wq2_rptr[ADDRSIZE-1]) &&
// (wgnext[ADDRSIZE-2:0]==wq2_rptr[ADDRSIZE-2:0]));
//---

```
assign wfull_val = (wgraynext=={~ wq2_rptr[fc_width:fc_width-1],wq2_rptr[fc_width-2:0]});

always @(posedge wrclk or negedge rstn) begin
if (!rstn) wfull_ptr <= 1'b0;
else wfull_ptr <= wfull_val;
end

endmodule
```

A.4. <u>FIFO.v:</u>

/* Main module for FIFO */

```
`timescale 1 ns / 1 ns

module FIFO (rdclk    ,         //Read Clock Input
             wrclk    ,         //Write Clock Input
             rstn     ,         //Reset Input
             wdata    ,         //write data Input
             rdata    ,         //Data Output
             wfull    ,         //full flag output
             rempty   ,         //Empty flag output
             winc     ,         //write Increment Input
             rinc     ,         //Read Increment Input
             TEST_SI,           //Scan dataIn
             TEST_SO,           //Scan dataOut
             TEST_SE);          //Scan Enable

parameter    fdepth = 32,
                        fwidth = 16,
                        fcwidth = 5;

input       rdclk,wrclk,rstn;
input       rinc,winc,TEST_SI,TEST_SE;
input       [fwidth-1: 0] wdata;
output      wfull,rempty,TEST_SO;
output      [fwidth-1: 0] rdata;

reg         wclken;
wire        [fcwidth: 0] rptr,wptr;
wire        rstn;
reg         [fcwidth: 0] rq2_wptr,wq2_rptr;
reg         [fcwidth: 0] rq1_wptr,wq1_rptr;
wire        [fcwidth-1 :0] r_addr;
wire        [fcwidth-1 :0] w_addr;


//Synchronizing the read clock domain
always@ (posedge rdclk or negedge rstn) begin
            if (!rstn)
                  {rq2_wptr,rq1_wptr} <= 0;
            else
                  {rq2_wptr,rq1_wptr} <= {rq1_wptr,wptr};
            end

//Synchronizing the write clock domain
always@ (posedge wrclk or negedge rstn) begin
            if (!rstn)
                  {wq2_rptr,wq1_rptr} <= 0;
            else
                  {wq2_rptr,wq1_rptr} <= {wq1_rptr,rptr};
            end

//write clock enable for memory
//assign wclken = winc & ~wfull;

always@ ( wfull or winc) begin
            if ( winc & ~wfull)
                  wclken = 1'b1;
            else
                  wclken = 1'b0;
            end

//Instantiating submodules
rptr_ctrl #(fcwidth) C1 (       .r_addr(r_addr),
                                        .r_ptr (rptr)      ,
                                        .rempty_ptr (rempty),
                                        .r_inc (rinc)     ,
                                        .rdclk (rdclk)   ,
                                        .rstn (rstn),
                                        .rq2_wptr (rq2_wptr));

wptr_ctrl #(fcwidth) C2 (       .w_addr(w_addr),
                                        .w_ptr (wptr),
                                        .wfull_ptr (wfull),
                                        .w_inc (winc)  ,
                                        .wrclk (wrclk) ,
                                        .rstn (rstn),
                                        .wq2_rptr (wq2_rptr));

mem_blk #(fwidth,fdepth,fcwidth) C3 (         .clk(wrclk),
                                                  .wr_addr(w_addr),
                                                  .wr_data(wdata),
                                                  .wr_en(wclken),
                                                  .rd_addr(r_addr),
                                                  .rd_data(rdata));
endmodule
```

A.5. <u>FIFO.scr: (script file)</u>

```
analyze -f verilog mem_blk.v
elaborate mem_blk
analyze -f verilog rptr_ctrl.v
elaborate rptr_ctrl
analyze -f verilog wptr_ctrl.v
elaborate wptr_ctrl
analyze -f verilog FIFO.v
elaborate FIFO

current_design FIFO
check_design
link
ungroup -all -flatten
#set_operating_conditions BEST

#Defining Clocks
create_clock -period 2.5 [get_ports rdclk]
create_clock -period 3 [get_ports wrclk]
set_false_path -from wrclk -to rdclk
set_false_path -from rdclk -to wrclk
set_drive 0 rdclk
set_drive 0 wrclk
set_drive 0 rstn
#Power Optimization Section
set power_driven_clock_gating true
set_clock_gating_style -sequential_cell latch -positive_edge_logic integrated -control_point
before -control_signal TEST_SE
set_max_dynamic_power 0
set_max_leakage_power 0

#Check and Compile the Design with clock gating power optimization and DFT scan chain
insertion
check_design -summary
compile_ultra -gate_clock -scan

#DFT Optimization Section (Identifying ports)
set_dft_signal -view existing_dft -type ScanClock -port rdclk -timing [list 45 55]
set_dft_signal -view existing_dft -type Reset -port rstn -active_state 0
set_dft_signal -view existing_dft -type ScanDataIn -port TEST_SI
set_dft_signal -view existing_dft -type ScanDataOut -port TEST_SO
set_dft_signal -view spec -type ScanEnable -port TEST_SE

#Scan Testability Environment
set test_default_scan_style multiplexed_flip_flop
set_scan_configuration -chain_count 2 -style multiplexed_flip_flop
set hdlin_enable_rtldrc_info true
set_dft_insertion_configuration
set test_rldrc_latch_check_style transparent

#Automatic fix configuration for DFT
set_dft_drc_configuration -allow_se_set_reset_fix true
set_dft_configuration -fix_clock enable -fix_reset enable -fix_set enable

# Creation of test protocol and DFT violation checks
create_test_protocol -infer_clock -infer_asynch
dft_drc
dft_drc -verbose

# DFT Scan Chain Insertion
preview_dft
preview_dft -show all -test_points all
insert_dft

#Compile the Design with clock gating power optimization and DFT scan chain insertion
check_design -summary
compile_ultra -scan
#compile_ultra
```

#Generating Reports

```
dft_drc -coverage_estimate > ../grad_proj/reports/lp_coverage_estimate.rpt
report_power > ../grad_proj/reports/lp_power.rpt
report_area > ../grad_proj/reports/lp_area.rpt
report_timing > ../grad_proj/reports/lp_timing.rpt
report_clock_gating > ../grad_proj/reports/lp_clock.rpt
report_port > ../grad_proj/reports/lp_io_port.rpt
report_timing_requirements > ../grad_proj/reports/lp_timing_req.rpt
report_constraint -all_violators > ../grad_proj/reports/lp_constraints.rpt
report_scan_configuration > ../grad_proj/scan_reports/lp_scan_configuration.rpt
write_sdf ../grad_proj/reports/lp_FIFO.sdf
write_sdc ../grad_proj/reports/lp_FIFO.sdc
write -f ddc -o ../grad_proj/reports/lp_FIFO_gates.ddc
write -f verilog -o ../grad_proj/reports/lp_FIFO_gates.v
write_parasitics -o ../grad_proj/reports/lp_FIFO.sbpf
write_parasitics -o ../grad_proj/reports/lp_FIFO.spef
```

Appendix B: (Analysis Reports)

B.1. Front End Reports without scan insertion or power optimization:

B.1.1. Timing Report:

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : FIFO
Version: E-2010.12-SP4
Date   : Fri Apr  6 11:42:52 2012
****************************************

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

  Startpoint: C1/rbin_reg[3]
              (rising edge-triggered flip-flop clocked by rdclk)
  Endpoint: C1/rempty_ptr_reg
            (rising edge-triggered flip-flop clocked by rdclk)
  Path Group: rdclk
  Path Type: max

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

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock rdclk (rise edge)                  0.00       0.00
  clock network delay (ideal)              0.00       0.00
  C1/rbin_reg[3]/CP (dfcrq1)               0.00       0.00 r
  C1/rbin_reg[3]/Q (dfcrq1)                0.36       0.36 r
  U319/ZN (nd02d1)                         0.50       0.86 f
  U645/ZN (nd12d0)                         0.24       1.10 f
  U146/Z (mx02d0)                          0.31       1.41 f
  U257/Z (mx02d0)                          0.31       1.71 f
  U388/ZN (inv0d1)                         0.06       1.77 r
  U290/Z (mx02d0)                          0.16       1.94 r
  U657/ZN (nd03d0)                         0.10       2.04 f
  U408/Z (or04d1)                          0.26       2.31 f
  C1/rempty_ptr_reg/D (dfcrb1)             0.00       2.31 f
  data arrival time                                   2.31

  clock rdclk (rise edge)                  2.50       2.50
  clock network delay (ideal)              0.00       2.50
  C1/rempty_ptr_reg/CP (dfcrb1)            0.00       2.50 r
  library setup time                      -0.10       2.40
```

```
data required time                                  2.40
-----------------------------------------------------------------------
data required time                                  2.40
data arrival time                                   -2.31
-----------------------------------------------------------------------
slack (MET)                                         0.09


Startpoint: C2/wbin_reg[0]
          (rising edge-triggered flip-flop clocked by wrclk)
Endpoint: C3/mem_data_reg[9][2]
        (rising edge-triggered flip-flop clocked by wrclk)
Path Group: wrclk
Path Type: max

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

Point                                        Incr      Path
--------------------------------------------------------------------
clock wrclk (rise edge)                      0.00      0.00
clock network delay (ideal)                  0.00      0.00
C2/wbin_reg[0]/CP (dfcrn1)                   0.00      0.00 r
C2/wbin_reg[0]/QN (dfcrn1)                   0.42      0.42 f
U220/ZN (invbd4)                             0.10      0.52 r
U382/ZN (nr03d0)                             0.23      0.75 f
U137/Z (bufbd1)                              0.47      1.21 f
U711/Z (aor22d1)                             0.30      1.52 f
U400/Z (or04d1)                              0.17      1.68 f
U720/Z (aoim22d1)                            0.30      1.98 r
U248/ZN (oai211d2)                           0.49      2.47 f
U249/Z (bufbd1)                              0.34      2.81 f
C3/mem_data_reg[9][2]/D (denrq1)             0.00      2.81 f
data arrival time                                      2.81

clock wrclk (rise edge)                      3.00      3.00
clock network delay (ideal)                  0.00      3.00
C3/mem_data_reg[9][2]/CP (denrq1)            0.00      3.00 r
library setup time                           -0.17     2.83
data required time                                    2.83
-----------------------------------------------------------------------
data required time                                    2.83
data arrival time                                     -2.81
----------------------------------------------------------------------
slack (MET)                                           0.02

1
```

B.1.2. Area Report:

```
****************************************
Report : area
Design : FIFO
Version: E-2010.12-SP4
Date   : Fri Apr  6 11:42:52 2012
****************************************

Library(s) Used:

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

Number of ports:                    39
Number of nets:                     1448
Number of cells:                    1426
Number of combinational cells:      866
Number of sequential cells:         560
Number of macros:                   0
Number of buf/inv:                  73
Number of references:               31

Combinational area:                 1773.000000
Noncombinational area:              3976.500000
Net Interconnect area:              1044.771567

Total cell area:                    5749.500000
Total area:                         6794.271567
1
```

B.1.3. Power Report:

```
Information: Updating design information... (UID-85)
Information: Propagating switching activity (low effort zero delay simulation). (PWR-6)

****************************************
Report : power
        -analysis_effort low
Design : FIFO
Version: E-2010.12-SP4
Date   : Fri Apr  6 11:42:52 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                               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        =   232.0549 uW   (55%)
  Net Switching Power        =   187.9766 uW   (45%)
                       ---------
Total Dynamic Power          =   420.0314 uW   (100%)

Cell Leakage Power           =   36.3260 uW

1
```

B.1.4. Clock Gating Report:

```
****************************************
Report:  clock_gating
Design: FIFO
Version: E-2010.12-SP4
Date   : Fri Apr 6 11:42:52 2012
****************************************
                    Clock Gating Summary
        ------------------------------------------------------------------------
        |  Number of Clock gating elements  |          0                       |
        |                                   |                                  |
        |  Number of Gated registers        |          0 (0.00%)               |
        |                                   |                                  |
        |  Number of Ungated registers      |    560 (100.00%)                 |
        |                                   |                                  |
        |  Total number of registers        |          560                     |
        ------------------------------------------------------------------------

1
```

B.2. Front End power report with scan insertion only:

```
Loading db file '/usr/synopsys/E-2010.12-SP2/ref/db/sc_max.db'
Information: Updating design information... (UID-85)
Information: Propagating switching activity (low effort zero delay simulation). (PWR-6)

****************************************
Report:   power
          -analysis_effort low
Design: FIFO
Version: E-2010.12-SP4
Date   : Mon Apr 16 11:30:00 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           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          =   1.4790 mW      (68%)
  Net Switching Power          =   684.2270 uW   (32%)
                         ---------
Total Dynamic Power            =   2.1632 mW      (100%)

Cell Leakage Power             =   39.0974 uW

1
```

B.3. Front End Reports with scan insertion and power optimization

B.3.1. Timing Report:

```
****************************************
Report : timing
            -path full
            -delay max
            -max_paths 1
Design : FIFO
Version: E-2010.12-SP4
Date   : Mon Apr 16 11:35:22 2012
****************************************

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

  Startpoint: C1/rempty_ptr_reg
              (rising edge-triggered flip-flop clocked by rdclk)
  Endpoint: C1/rempty_ptr_reg
            (rising edge-triggered flip-flop clocked by rdclk)
  Path Group: rdclk
  Path Type: max

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

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock rdclk (rise edge)                  0.00       0.00
  clock network delay (ideal)              0.00       0.00
  C1/rempty_ptr_reg/CP (sdprb1)            0.00       0.00 r
  C1/rempty_ptr_reg/Q (sdprb1)             0.36       0.36 r
  U1373/ZN (nd12d1)                        0.12       0.49 r
  U1408/ZN (nr02d0)                        0.14       0.63 f
  U1406/ZN (nd02d1)                        0.14       0.77 r
  U1410/ZN (nr02d0)                        0.16       0.92 f
  U1352/ZN (nd12d1)                        0.08       1.00 r
  U1372/Z (mx02d1)                         0.29       1.29 f
  U1287/Z (mx02d0)                         0.29       1.58 r
  U1286/Z (mx02d0)                         0.24       1.82 r
  U1414/ZN (nr04d0)                        0.09       1.91 f
  C1/rempty_val_reg/Q (slnhq1)             0.36       2.27 f
  C1/rempty_ptr_reg/D (sdprb1)             0.00       2.27 f
  data arrival time                                   2.27

  clock rdclk (rise edge)                  2.50       2.50
  clock network delay (ideal)              0.00       2.50
  C1/rempty_ptr_reg/CP (sdprb1)            0.00       2.50 r
  library setup time                      -0.23       2.27
```

```
data required time                                          2.27
---------------------------------------------------------------------------
data required time                                          2.27
data arrival time                                          -2.27
---------------------------------------------------------------------------
slack (MET)                                                 0.01


Startpoint: C2/wbin_reg[3]
          (rising edge-triggered flip-flop clocked by wrclk)
Endpoint: C3/mem_data_reg[3][1]
         (rising edge-triggered flip-flop clocked by wrclk)
Path Group: wrclk
Path Type: max

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

Point                                            Incr       Path
--------------------------------------------------------------------------------
clock wrclk (rise edge)                          0.00       0.00
clock network delay (ideal)                      0.00       0.00
C2/wbin_reg[3]/CP (sdcrb1)                       0.00       0.00 r
C2/wbin_reg[3]/Q (sdcrb1)                        0.37       0.37 f
U1282/ZN (nd02d0)                                0.40       0.77 r
U1263/ZN (nd12d1)                                0.32       1.08 r
U1274/ZN (nr02d0)                                0.15       1.24 f
U1236/Z (bufbd1)                                 0.40       1.63 f
U1463/Z (aor22d1)                                0.29       1.92 f
U1467/ZN (nr03d0)                                0.11       2.03 r
U1238/ZN (nd04d1)                                0.37       2.40 f
U1292/Z (buffd1)                                 0.39       2.78 f
C3/mem_data_reg[3][1]/D (sdnrq1)                 0.01       2.79 f
data arrival time                                           2.79

clock wrclk (rise edge)                          3.00       3.00
clock network delay (ideal)                      0.00       3.00
C3/mem_data_reg[3][1]/CP (sdnrq1)                0.00       3.00 r
library setup time                              -0.20       2.80
data required time                                          2.80
-----------------------------------------------------------------------------
data required time                                          2.80
data arrival time                                          -2.79
-----------------------------------------------------------------------------
slack (MET)                                                 0.00

1
```

B.3.2. Area Report:

```
****************************************
Report : area
Design : FIFO
Version: E-2010.12-SP4
Date   : Mon Apr 16 11:35:22 2012
****************************************

Library(s) Used:

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

Number of ports:                  43
Number of nets:                   1548
Number of cells:                  1511
Number of combinational cells:    918
Number of sequential cells:       561
Number of macros:                 0
Number of buf/inv:                54
Number of references:             72

Combinational area:               1730.750000
Noncombinational area:            4348.000000
Net Interconnect area:            1032.483822

Total cell area:                  6078.750000
Total area:                       7111.233822
1
```

B.3.3. Power Report:

```
Information: Updating design information... (UID-85)
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
Version: E-2010.12-SP4
Date   : Mon Apr 16 11:35:22 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                              8000           cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_0
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_31
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_30
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_29
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_28
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_27
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_26
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_25
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_24
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_23
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_22
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_21
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_20
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_19
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_18
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_17
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_16
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_15
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_14
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_13
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_12
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_11
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_10
                                  ForQA            cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_9
```

```
                                   ForQA          cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_8
                                   ForQA          cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_7
                                   ForQA          cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_6
                                   ForQA          cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_5
                                   ForQA          cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_4
                                   ForQA          cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_3
                                   ForQA          cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_2
                                   ForQA          cb13fs120_tsmc_max
SNPS_CLOCK_GATE_HIGH_FIFO_1
                                   ForQA          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          =   1.0445 mW   (50%)
Net Switching Power          =   1.0530 mW   (50%)
                    ---------
Total Dynamic Power          =   2.0975 mW   (100%)

Cell Leakage Power           =   34.2957 uW

1
```

B.3.4. Clock Gating Report:

```
 ****************************************
Report: clock_gating
Design: FIFO
Version: E-2010.12-SP4
Date   : Mon Apr 16 11:35:22 2012
****************************************

                    Clock Gating Summary
       ---------------------------------------------------------------
       |   Number of Clock gating elements   |        32         |
       |                                     |                   |
       |   Number of Gated registers         |   512 (91.43%)    |
       |                                     |                   |
       |   Number of Ungated registers       |    48 (8.57%)     |
       |                                     |                   |
       |   Total number of registers         |       560         |
       -----------------------------------------------------------------

1
```

B.4. Front End Coverage Estimate Report:

(W/ scan insertion and power optimization techniques)

```
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 593 sequential cells have violations
-------------------------------------------------------------
SEQUENTIAL CELLS WITH VIOLATIONS
      *   32 cells are clock gating cells
SEQUENTIAL CELLS WITHOUT VIOLATIONS
      * 549 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...
 14256 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=10154, abort_limit=10...
 0           5778   4376            0/0/8                     69.30%           0.04
 0           1586   2790            0/0/16                    80.42%           0.04
 0           1018   1772            0/0/30                    87.57%           0.05
 0           587    1185            0/0/46                    91.69%           0.05
 0           395    790             0/0/71                    94.46%           0.06
 0           266    524             0/0/88                    96.32%           0.07
 0           189    335             0/0/131                   97.65%           0.08
 0           132    203             0/0/155                   98.58%           0.08
 0           98     105             0/0/169                   99.26%           0.09
 0           38     67              0/0/169                   99.53%           0.09

          Pattern Summary Report
 ---------------------------------------------------
 #internal patterns                       0
 ---------------------------------------------------
    Uncollapsed Stuck Fault Summary Report
 ------------------------------------------------------
 fault class                         code    #faults
 ------------------------------    ----------   ----------
 Detected                            DT     14185
 Possibly detected                   PT        0
 Undetectable                        UD         4
 ATPG untestable                     AU         0
 Not detected                        ND        67
 ------------------------------------------------------
 total faults                                 14256
 test coverage                                99.53%
 ------------------------------------------------------
  Information: The test coverage above may be inferior
              than the real test coverage with customized
              protocol and test simulation library.
1
```

B.5. Back End Report without power optimization techniques:

B.5.1. Timing Report:

```
****************************************
Report: timing
             -path full
             -delay max
             -max_paths 1
Design: FIFO
Version: E-2010.12-ICC-SP2
Date   : Mon Apr 16 12:20:18 2012
****************************************

 * Some/all delay information is back-annotated.

Operating Conditions: cb13fs120_tsmc_max   Library: cb13fs120_tsmc_max

Information: Percent of Arnoldi-based delays = 27.74%

  Startpoint: C1/rbin_reg[3]
           (rising edge-triggered flip-flop clocked by rdclk)
  Endpoint: C1/rempty_ptr_reg
          (rising edge-triggered flip-flop clocked by rdclk)
  Path Group: rdclk
  Path Type: max

  Point                                          Incr       Path
  --------------------------------------------------------------------------
  clock rdclk (rise edge)                        0.00       0.00
  clock network delay (propagated)               0.01       0.01
  C1/rbin_reg[3]/CP (dfcrq1)                     0.00       0.01 r
  C1/rbin_reg[3]/Q (dfcrq1)                      0.35       0.35 r
  U319/ZN (nd02d1)                               0.37 &     0.72 f
  U645/ZN (nd12d0)                               0.20 &     0.92 f
  U146/Z (mx02d0)                                0.28 &     1.21 f
  U257/Z (mx02d0)                                0.28 &     1.48 f
  U388/ZN (inv0d1)                               0.05 &     1.54 r
  U290/Z (mx02d0)                                0.16 &     1.69 r
  U657/ZN (nd03d0)                               0.09 &     1.79 f
  U408/Z (or04d1)                                0.26 &     2.04 f
  C1/rempty_ptr_reg/D (dfcrb1)                   0.00 &     2.04 f
  data arrival time                                         2.04

  clock rdclk (rise edge)                        2.50       2.50
  clock network delay (propagated)               0.01       2.51
  clock uncertainty                              -0.10      2.41
  C1/rempty_ptr_reg/CP (dfcrb1)                  0.00       2.41 r
  library setup time                             -0.09      2.32
  data required time                                        2.32
  --------------------------------------------------------------------------
```

```
data required time                                        2.32
data arrival time                                         -2.04
---------------------------------------------------------------------------
slack (MET)                                               0.27


Startpoint: C2/wbin_reg[0]
        (rising edge-triggered flip-flop clocked by wrclk)
Endpoint: C3/mem_data_reg[31][11]
       (rising edge-triggered flip-flop clocked by wrclk)
Path Group: wrclk
Path Type: max

Point                                       Incr       Path
---------------------------------------------------------------------------
clock wrclk (rise edge)                     0.00       0.00
clock network delay (propagated)            0.19       0.19
C2/wbin_reg[0]/CP (dfcrn1)                  0.00       0.19 r
C2/wbin_reg[0]/QN (dfcrn1)                  0.40       0.59 f
U1/ZN (invbd2)                              0.10 &     0.70 r
U384/ZN (nr03d2)                            0.74 &     1.44 f
U903/Z (aor22d1)                            0.31 &     1.75 f
U394/Z (or04d1)                             0.25 &     1.99 f
U909/Z (aoim22d1)                           0.29 &     2.28 r
U233/ZN (oai211d1)                          0.64 &     2.93 f
C3/mem_data_reg[31][11]/D (denrq1)          0.00 &     2.93 f
data arrival time                                      2.93

clock wrclk (rise edge)                     3.00       3.00
clock network delay (propagated)            0.18       3.18
clock uncertainty                           -0.10      3.08
C3/mem_data_reg[31][11]/CP (denrq1)          0.00      3.08 r
library setup time                          -0.14      2.93
data required time                                      2.93
---------------------------------------------------------------------------
data required time                                      2.93
data arrival time                                      -2.93
---------------------------------------------------------------------------
slack (MET)                                            0.01

1
```

B.5.2. Area Report:

```
****************************************
Report : area
Design : FIFO
Version: E-2010.12-ICC-SP2
Date   : Mon Apr 16 12:28:26 2012
****************************************

Library(s) Used:

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

Number of ports:                   39
Number of nets:                  1416
Number of cells:                 1394
Number of combinational cells: 834
Number of sequential cells:       560
Number of macros:                 0
Number of buf/inv:                41
Number of references:             31

Combinational area:              1773.250000
Noncombinational area:           3976.500000
Net Interconnect area:           1112.274726

Total cell area:          5749.750000
Total area:             6862.024726
1
```

B.5.3. Power Report:

```
****************************************
Report : power
        -analysis_effort low
Design : FIFO
Version: E-2010.12-ICC-SP2
Date   : Mon Apr 16 12:31:03 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                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          = 920.7864 uW   (51%)
  Net Switching Power          = 901.1628 uW   (49%)
                       ---------
Total Dynamic Power            =   1.8219 mW   (100%)

Cell Leakage Power             =   36.5097 uW

1
```

B.5.4. Quality of Results Report:

```
****************************************
Report : qor
Design : FIFO
Version: E-2010.12-ICC-SP2
Date   : Mon Apr 16 12:38:28 2012
****************************************
  Timing Path Group 'rdclk'
  -----------------------------------
  Levels of Logic:               8.00
  Critical Path Length:          2.03
  Critical Path Slack:           0.27
  Critical Path Clk Period:      2.50
  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 'wrclk'
  -----------------------------------
  Levels of Logic:               6.00
  Critical Path Length:          2.74
  Critical Path Slack:           0.01
  Critical Path Clk Period:      3.00
  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:                1394
Buf/Inv Cell Count:             41
CT Buf/Inv Cell Count:          13
Combinational Cell Count:       834
Sequential Cell Count:          560
Macro Count:                    0
-----------------------------------------------


Area
-----------------------------------------------------
Combinational Area:             1773.250000
Noncombinational Area:          3976.500000
Net Area:                       1112.274726
Net XLength        :            51935.60
Net YLength        :            52973.64
-----------------------------------------------------
Cell Area:                      5749.750000
Design Area:                    6862.024726
Net Length        :             104909.23

Design Rules
-----------------------------------------------------
Total Number of Nets:         1416
Nets With Violations:            0
-----------------------------------------------------

Hostname: dcd154.ecs.csun.edu

Compile CPU Statistics
-----------------------------------------------------
Resource Sharing:                           0.00
Logic Optimization:                         0.00
Mapping Optimization:                       1.56
---------------------------------------------------
Overall Compile Time:                       1.70
Overall Compile Wall Clock Time:  2.97

1
```

B.6. Back End Reports with power optimization techniques:\

B.6.1 Timing Report:

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : FIFO
Version: E-2010.12-ICC-SP2
Date   : Mon Apr 16 14:01:33 2012
****************************************

 * Some/all delay information is back-annotated.

Operating Conditions: cb13fs120_tsmc_max   Library: cb13fs120_tsmc_max
    Parasitic source    : LPE
    Parasitic mode      : RealRC
    Extraction mode     : MIN_MAX
    Extraction derating : 125/125/125

Information: Percent of Arnoldi-based delays = 11.66%

  Startpoint: winc (input port clocked by wrclk)
  Endpoint: C2/wfull_ptr_reg
            (rising edge-triggered flip-flop clocked by wrclk)
  Path Group: INPUTS
  Path Type: max

  Point                                    Incr       Path
  ---------------------------------------------------------------------
  clock wrclk (rise edge)                  0.00       0.00
  clock network delay (ideal)              0.00       0.00
  input external delay                     0.10       0.10 f
  winc (in)                                0.02       0.12 f
  U136/ZN (nd12d0)                         0.22 &     0.34 r
  U363/ZN (nr03d0)                         0.24 &     0.58 f
  U270/ZN (nd02d0)                         0.14 &     0.72 r
  U362/ZN (nr02d0)                         0.19 &     0.91 f
  U286/Z (an02d0)                          0.18 &     1.08 f
  U265/Z (mx02d0)                          0.29 &     1.37 f
  U258/Z (mx02d0)                          0.23 &     1.60 f
  U390/ZN (inv0d0)                         0.08 &     1.68 r
  U298/Z (mx02d0)                          0.16 &     1.84 r
  U653/ZN (nd03d0)                         0.10 &     1.95 f
  U654/ZN (nr04d0)                         0.17 &     2.12 r
  C2/wfull_ptr_reg/D (dfcrq1)              0.00 &     2.12 r
  data arrival time                                   2.12

  clock wrclk (rise edge)                  3.00       3.00
clock network delay (propagated)        0.00        3.00
clock uncertainty                       -0.10       2.90
C2/wfull_ptr_reg/CP (dfcrq1)            0.00        2.90 r
library setup time                      -0.10       2.80
data required time                                  2.80
----------------------------------------------------------------------
data required time                                  2.80
data arrival time                                   -2.12
----------------------------------------------------------------------
slack (MET)                                         0.68


Startpoint: C1/rbin_reg[3]
          (rising edge-triggered flip-flop clocked by rdclk)
Endpoint: C1/rempty_ptr_reg
         (rising edge-triggered flip-flop clocked by rdclk)
Path Group: rdclk
Path Type: max

Point                                   Incr        Path
----------------------------------------------------------------------
clock rdclk (rise edge)                 0.00        0.00
clock network delay (propagated)        0.00        0.00
C1/rbin_reg[3]/CP (dfcrq1)              0.00        0.00 r
C1/rbin_reg[3]/Q (dfcrq1)               0.33        0.33 r
U319/ZN (nd02d1)                        0.36 &      0.70 f
U645/ZN (nd12d0)                        0.20 &      0.90 f
U146/Z (mx02d0)                         0.28 &      1.18 r
U257/Z (mx02d0)                         0.27 &      1.45 f
U388/ZN (inv0d0)                        0.08 &      1.53 r
U290/Z (mx02d0)                         0.16 &      1.69 r
U657/ZN (nd03d0)                        0.09 &      1.78 f
U408/Z (or04d0)                         0.17 &      1.95 f
C1/rempty_ptr_reg/D (dfcrb1)            0.00 &      1.95 f
data arrival time                                   1.95

clock rdclk (rise edge)                 2.50        2.50
clock network delay (propagated)        0.00        2.50
clock uncertainty                       -0.10       2.40
C1/rempty_ptr_reg/CP (dfcrb1)           0.00        2.40 r
library setup time                      -0.10       2.30
data required time                                  2.30
----------------------------------------------------------------------
data required time                                  2.30
data arrival time                                   -1.95
----------------------------------------------------------------------
slack (MET)                                         0.35

Startpoint: C2/wbin_reg[0]
          (rising edge-triggered flip-flop clocked by wrclk)
Endpoint: C3/mem_data_reg[17][3]
```

```
          (rising edge-triggered flip-flop clocked by wrclk)
  Path Group: wrclk
  Path Type: max

  Point                                          Incr       Path
  -----------------------------------------------------------------------------
  clock wrclk (rise edge)                        0.00       0.00
  clock network delay (propagated)               0.00       0.00
  C2/wbin_reg[0]/CP (dfcrn1)                     0.00       0.00 r
  C2/wbin_reg[0]/QN (dfcrn1)                     0.35       0.35 f
  U220/ZN (inv0d1)                               0.14 &     0.48 r
  U384/ZN (nr03d0)                               0.17 &     0.65 f
  U5/Z (bufbd1)                                  0.55 &     1.20 f
  U735/Z (aor22d1)                               0.30 &     1.50 f
  U397/Z (or04d0)                                0.25 &     1.75 f
  U741/Z (aoim22d1)                              0.29 &     2.04 r
  U238/ZN (oai211d1)                             0.64 &     2.69 f
  C3/mem_data_reg[17][3]/D (denrq2)              0.00 &     2.69 f
  data arrival time                                          2.69

  clock wrclk (rise edge)                        3.00        3.00
  clock network delay (propagated)               0.00        3.00
  clock uncertainty                              -0.10       2.90
  C3/mem_data_reg[17][3]/CP (denrq2)             0.00        2.90 r
  library setup time                             -0.17       2.73
  data required time                                         2.73
  -----------------------------------------------------------------------------
  data required time                                         2.73
  data arrival time                                         -2.69
  -----------------------------------------------------------------------------
  slack (MET)                                                 0.04

1
```

B.6.2 Area Report:

```
****************************************
Report : area
Design : FIFO
Version: E-2010.12-ICC-SP2
Date   : Mon Apr 16 14:08:24 2012
****************************************

Library(s) Used:

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

Number of ports:                   39
Number of nets:                  1433
Number of cells:                 1411
Number of combinational cells:    851
Number of sequential cells:       560
Number of macros:                   0
Number of buf/inv:                 58
Number of references:              31

Combinational area:            1780.750000
Noncombinational area:         4077.250000
Net Interconnect area:         1076.898201

Total cell area:          5858.000000
Total area:               6934.898201
1
```

B.6.3 Power Report:

```
****************************************
Report : power
        -analysis_effort low
Design : FIFO
Version: E-2010.12-ICC-SP2
Date   : Mon Apr 16 14:12:00 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                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            =   3.1520 mW      (89%)
  Net Switching Power            =   401.8876 uW   (11%)
                   ---------
Total Dynamic Power              =   1.6539 mW      (100%)

Cell Leakage Power               =   35.0765 uW

1
```

B.6.4 Quality Of Results Report:

```
****************************************
Report : qor
Design : FIFO
Version: E-2010.12-ICC-SP2
Date   : Mon Apr 16 14:40:38 2012
****************************************

  Timing Path Group 'INPUTS'
  -----------------------------------
  Levels of Logic:                11.00
  Critical Path Length:           2.02
  Critical Path Slack:            0.68
  Critical Path Clk Period:       3.00
  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 'rdclk'
  -----------------------------------
  Levels of Logic:                8.00
  Critical Path Length:           1.95
  Critical Path Slack:            0.35
  Critical Path Clk Period:       2.50
  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 'wrclk'
-----------------------------------------------------
Levels of Logic:                    7.00
Critical Path Length:               2.69
Critical Path Slack:                0.04
Critical Path Clk Period:           3.00
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:                    1411
Buf/Inv Cell Count:                  58
CT Buf/Inv Cell Count:               13
Combinational Cell Count:           851
Sequential Cell Count:              560
Macro Count:                          0
-----------------------------------------------------
Area
-----------------------------------------------------
Combinational Area:                 1780.750000
Noncombinational Area:              4077.250000
Net Area:                           1076.898201
Net XLength        :                54752.52
Net YLength        :                54701.40
-----------------------------------------------------
Cell Area:                          5858.000000
Design Area:                        6934.898201
Net Length        :                 109453.92

Design Rules
-----------------------------------------------------
Total Number of Nets:               1433
Nets With Violations:                0
-----------------------------------------------------
Hostname: dcd154.ecs.csun.edu
Compile CPU Statistics
-----------------------------------------------------
Resource Sharing:                            0.00
Logic Optimization:                          0.00
Mapping Optimization:                        7.66
-----------------------------------------------------
Overall Compile Time:                        8.21
Overall Compile Wall Clock Time:             9.96

1
```

Appendix C: (Layout Figures)

C.1 Layout after Data_setup showing Standard Cells and IOs Placed on top of each other at the bottom left corner.

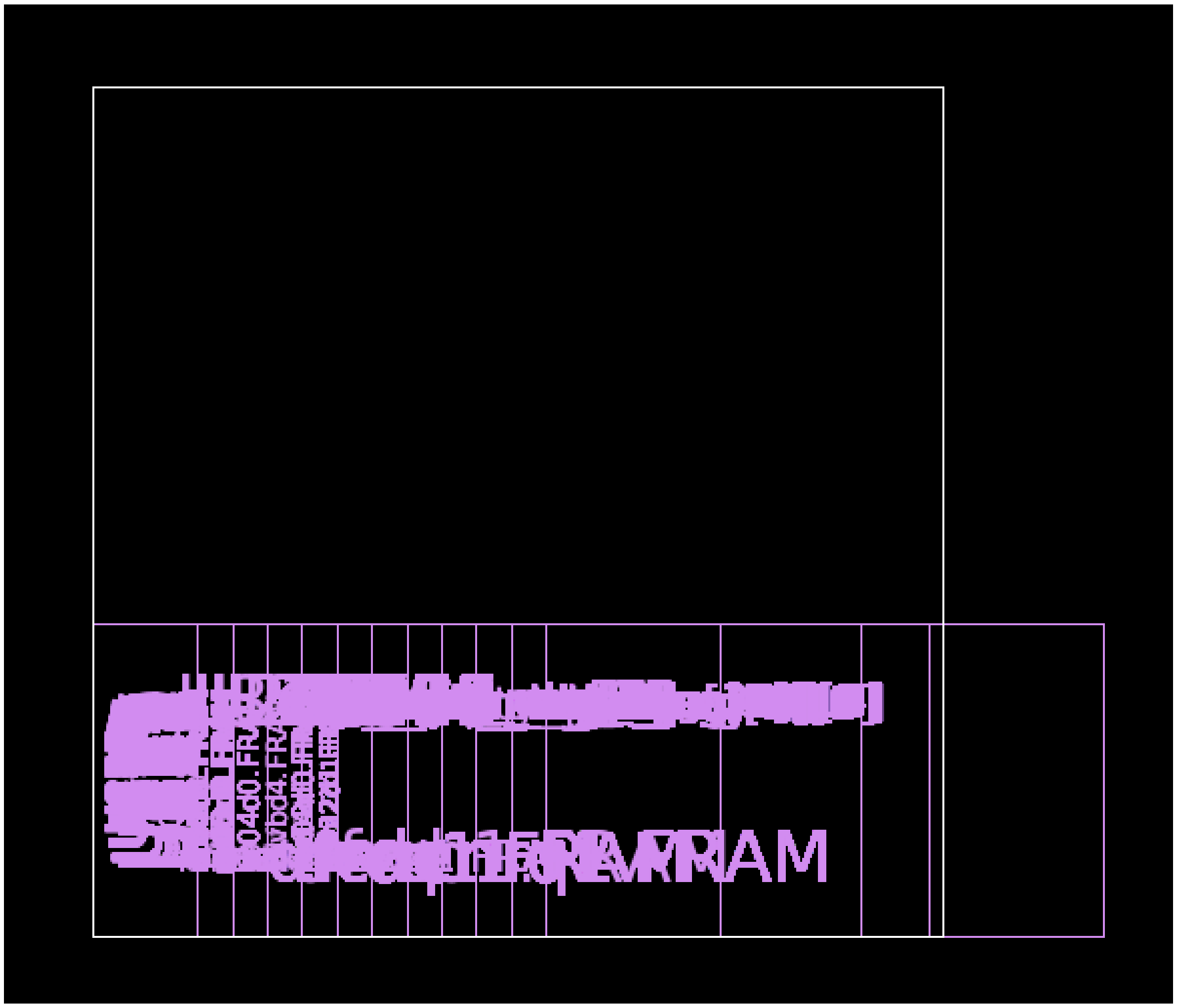


Fig. C.1: Layout showing IOs and standard cells on top of each other

C.2 **Layout after floorplan initialization:**

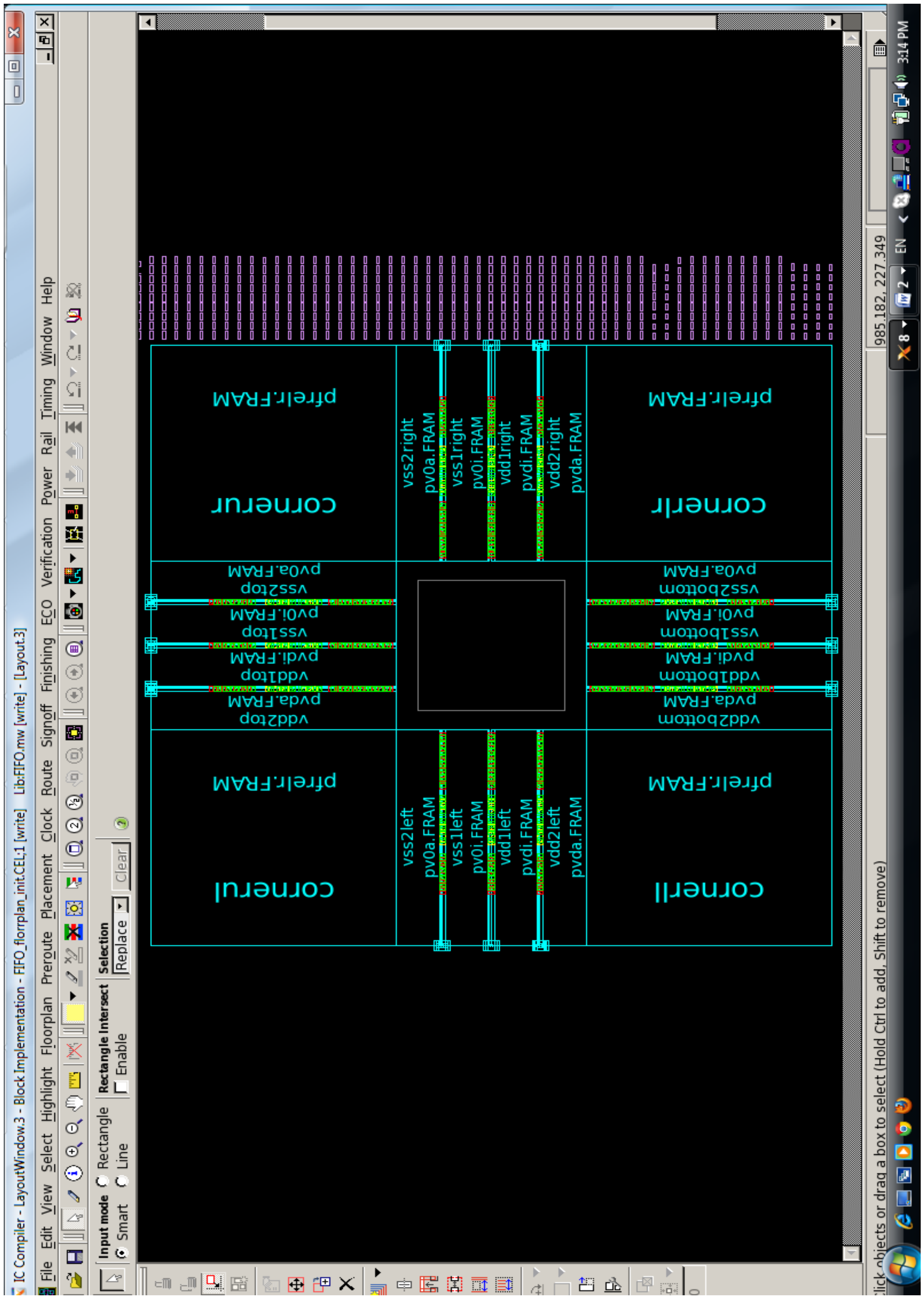


Fig. C.2: Floorplan Initialization layout

C.3 Layout after placing all the standard Cells:

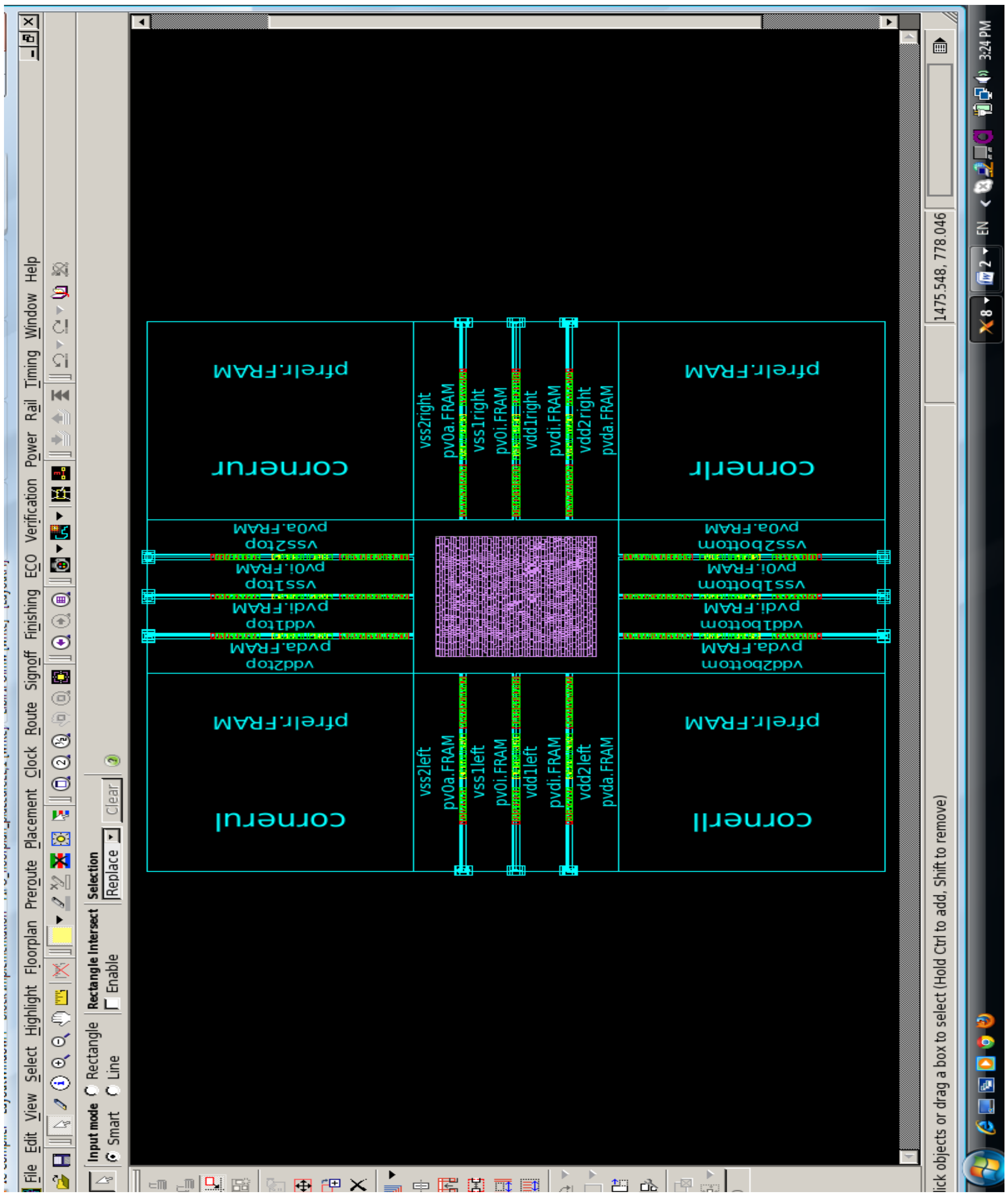
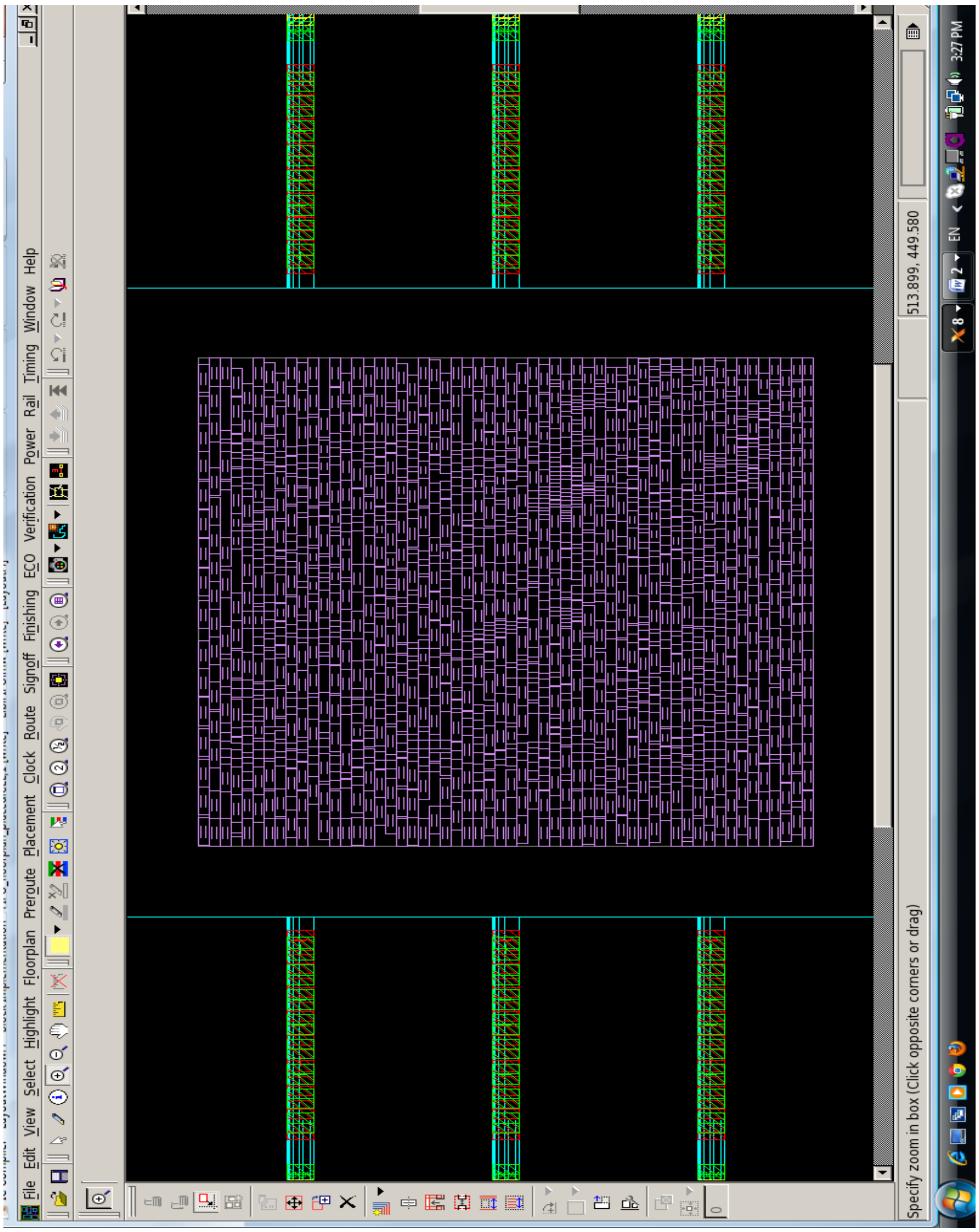


Fig. C.3: Layout after placing Standard Cells

C.4 Zoomed layout after placing standard cells:

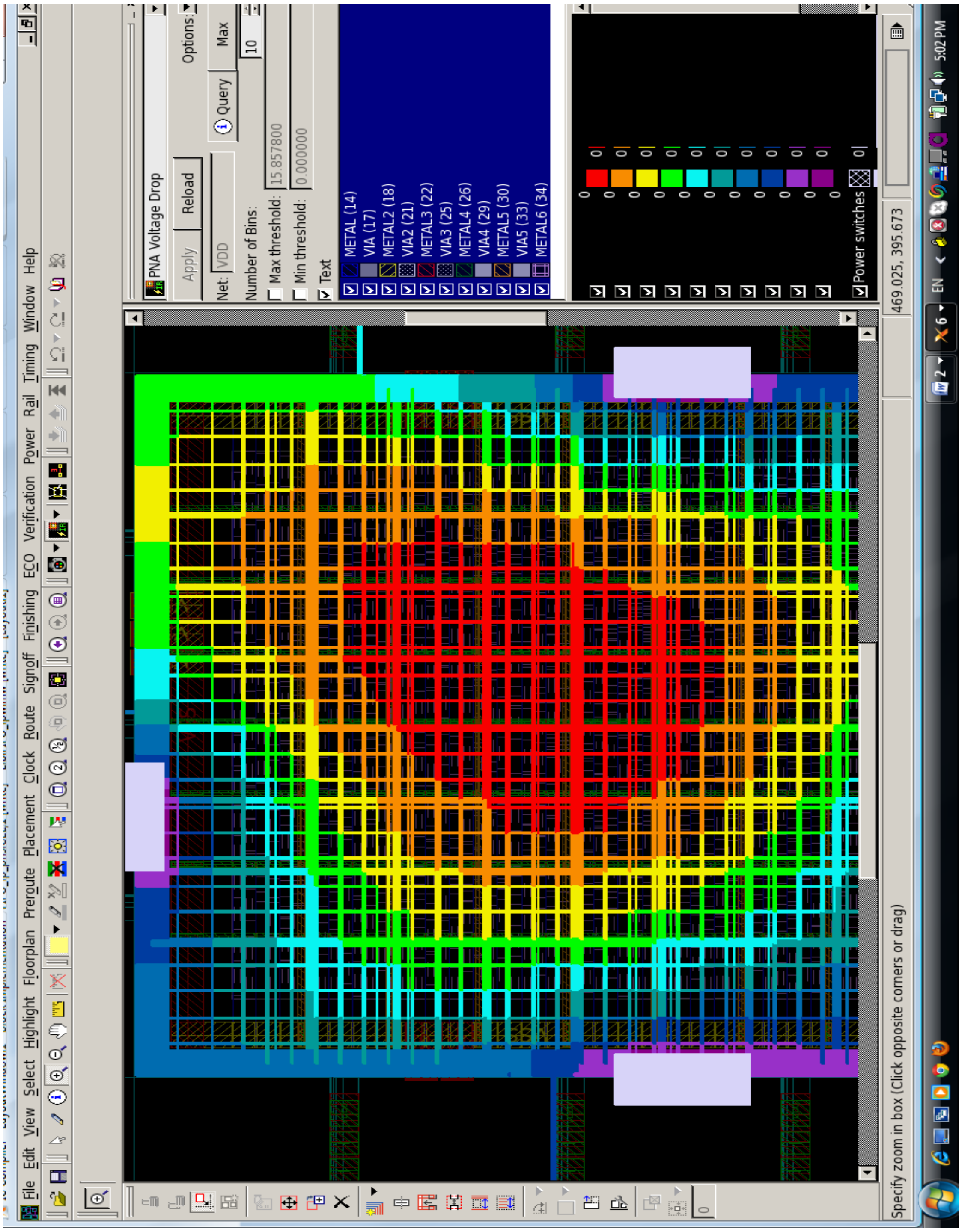


513.899, 449.580

Specify zoom in box (Click opposite corners or drag)

Fig. C.4: Zoomed layout for placed standard cells

C.5 Layout showing Voltage drop map after Power Network Synthesis:



Net: VDD
Number of Bins: 10
Max threshold: 15.857800
Min threshold: 0.000000

469.025, 395.673

Specify zoom in box (Click opposite corners or drag)

Fig. C.5: Voltage Drop Map

C.6 Layout after all the placement setup and optimization:

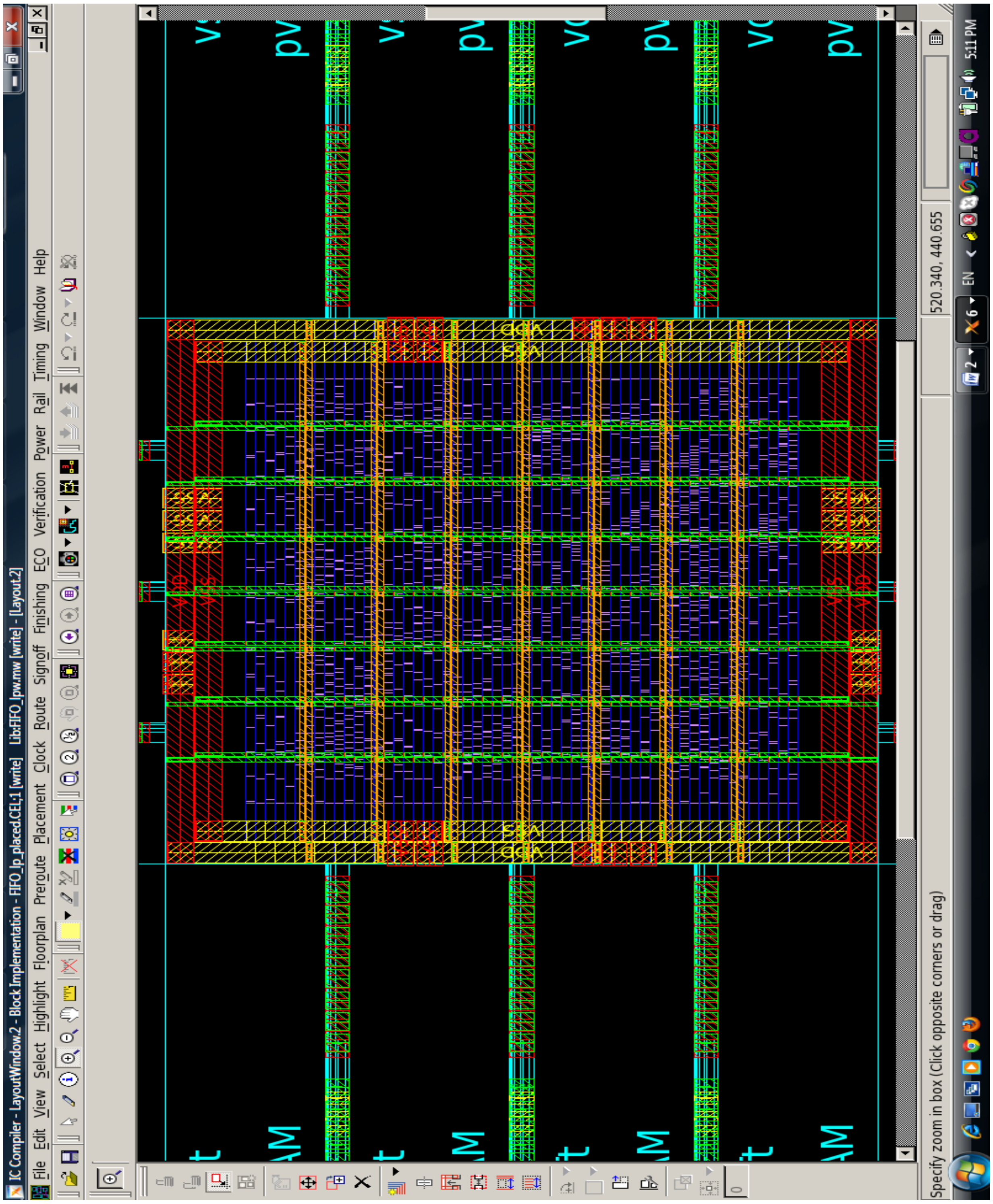

IC Compiler - LayoutWindow.2 - Block Implementation - FIFO_lp_placed.CEL;1 [write] Lib:FIFO_lpw.mw [write] - [Layout.2]
File Edit View Select Highlight Floorplan Preroute Placement Clock Route Signoff Finishing ECO Verification Power Rail Timing Window Help
520.340, 440.655
Specify zoom in box (Click opposite corners or drag)
VDD
VSS
EN
5:11 PM

Fig. C.6: Layout after placement setup and optimization

C.7 Layout showing the clock tree map:

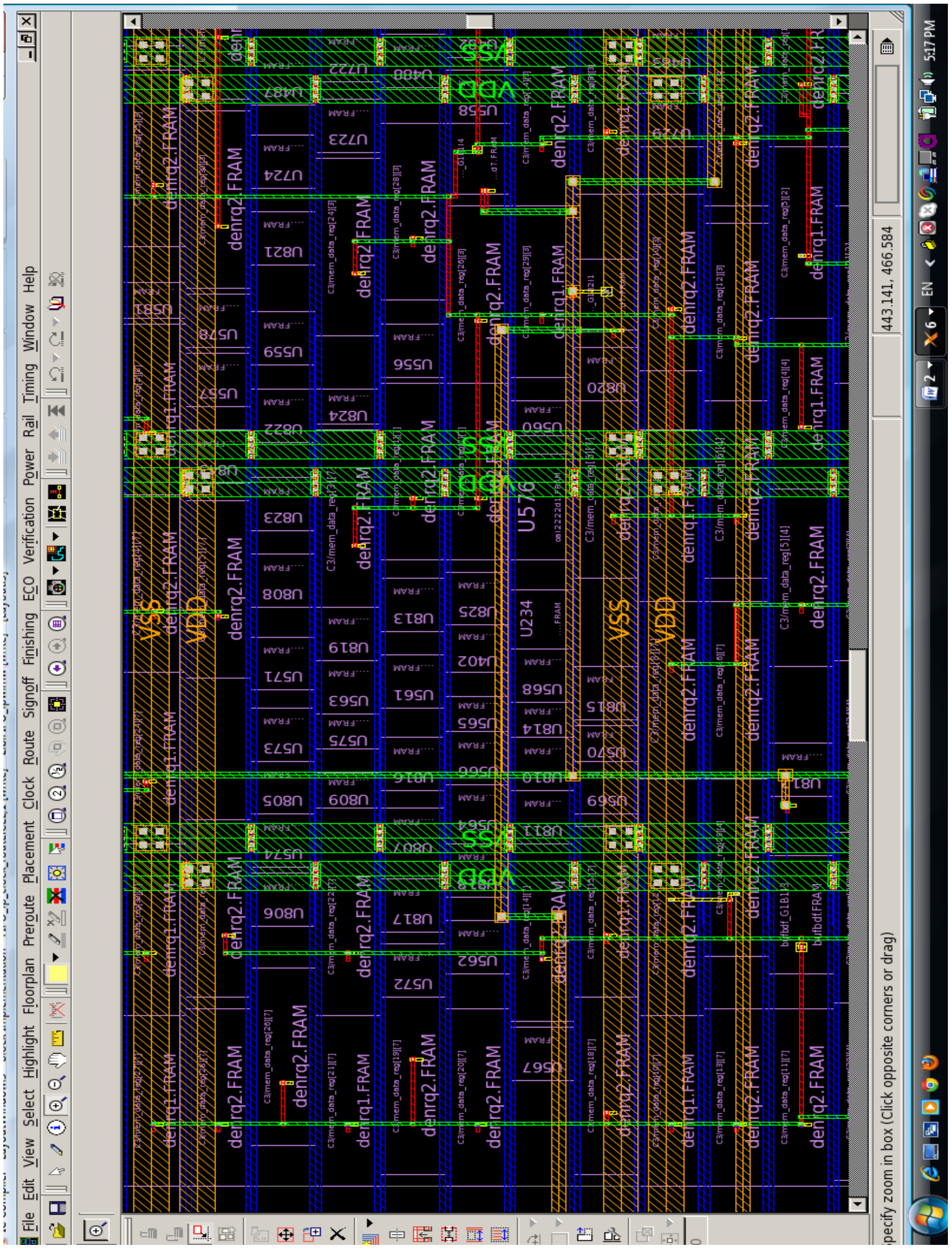


Fig. C.7: Layout after routing the clock

C.8 Layout showing routing:

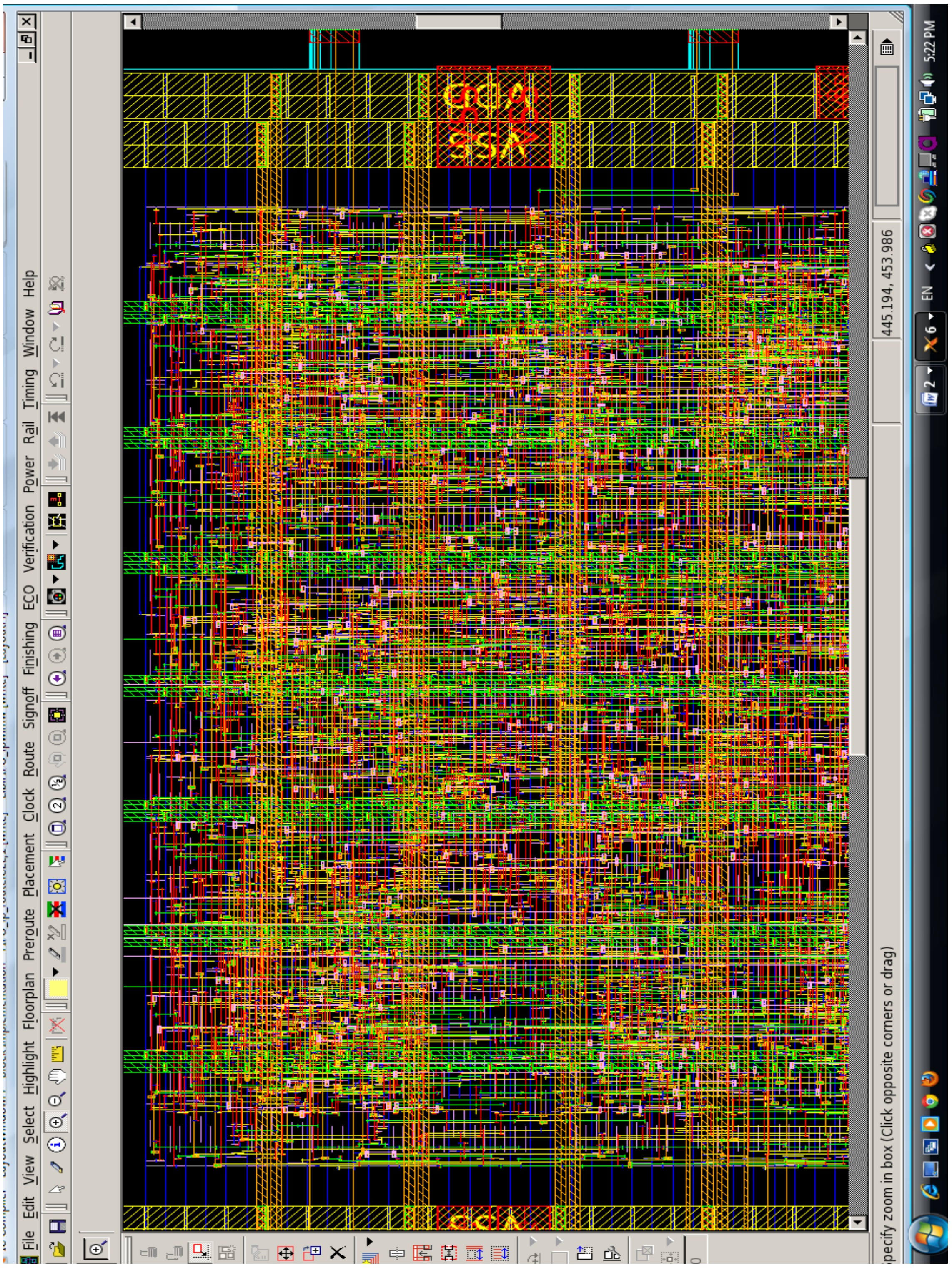

File Edit View Select Highlight Floorplan Preroute Placement Clock Route Signoff Finishing ECO Verification Power Rail Timing Window Help
445.194, 453.986
Specify zoom in box (Click opposite corners or drag)
5:22 PM

Fig. C.8: Layout Showing Routing