DEVELOPMENT OF RUNTIME ABSTRACTION UTILITY AGNOSTIC TO THE FLOW TYPE AND IMPLEMENTATION OF RUNTIME ENHANCED PROCEDURES IN THE MAIN FLOW OF PHYSICAL DESIGN

A Thesis Submitted in Partial Fulfillment of the Requirement for the Award of the Degree of

MASTER OF TECHNOLOGY
in VLSI Design

Submitted By
AASHISH JOSHI

601662001

Under Supervision of
Mr. Srinivas R STG
Engineering Manager
Intel Technology India Pvt. Ltd. Bangalore
and
Dr. Sanjay Kumar
Associate Professor, ECED, TIET

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY PATIALA, PUNJAB, INDIA
JUNE, 2018

INTEL TECHNOLOGY INDIA PVT. LTD.
Bangalore – 560103, Karnataka, India

Date: June 12, 2018

CERTIFICATE

This is to certify that **Aashish Joshi (Regn. No. 601662001)**, a student of M.Tech.(VLSI Design), **Thapar Institute of Engineering and Technology, Patiala** has successfully completed one year (August 2017 – June 2018) Internship programme at **Intel Technology India Pvt. Ltd, Bangalore.** His title of dissertation is "**Development of Runtime Abstraction utility agnostic to the flow type and Implementation of runtime enhanced procedures in the main flow of Physical design**".

During the period of his internship programme, he was punctual and hardworking.

I wish him every success in life.

Srinivas R STG
Engineering Manager
Intel, Bangalore

DECLARATION

I, **Aashish Joshi,** hereby declare that the thesis entitled **"Development of Runtime Abstraction utility agnostic to the flow type and Implementation of runtime enhanced procedures in the main flow of physical design"** is a record of my own work carried out towards the partial fulfilment for the award of degree of Master of Technology in VLSI Design at Thapar Institute of Engineering and Technology, Patiala, under the guidance of **Mr. Srinivas R STG** (Engineering Manager, Intel Technology India Pvt. Ltd, Bangalore) and **Dr. Sanjay Kumar,** Associate Professor , Electronics and Communication Engineering Department, Thapar Institute of Engineering and Technology, Patiala, India, from Aug 2017 – June 2018.

The matter presented in this thesis has not been submitted either in part or full to any other university or institute for the award of any other degree.

Date: 12 JUN 2018

Aashish Joshi

601662001

It is certified that the above statement made by the Student/ Intern (Mr. Aashish Joshi) is correct to the best of my knowledge and belief.

Mr. Srinivasa R STG

Engineering Manager

PDS, Product development and Solutions

Intel Technology India Pvt. Ltd, Bangalore

Dr. Sanjay Kumar

Associate Professor

ECED, TIET

Patiala-147004, (Punjab)

ACKNOWLEDGEMENT

Though my name appears on the cover of this disquisition, a group of great people have also contributed to its production. I owe my gratitude to all those people who have made this dream possible and because of whom my graduate experience has been one that will be cherished forever.

Foremost, I would like to express my sincere remittance towards my guide **Dr. Sanjay Kumar (Associate Professor, Department of Electronics and Communication Engineering, Thapar institute of Engineering and Technology, Patiala)**, for his consistent support and feedback, which helped me throughout the thesis work. I would like to thank my manager, **Mr. Srinivasa R STG (Engineering Manager, Intel Technology India Private Limited, Bangalore)** and my functional manager **Mrs. Chetana Halakatti (CAD Engineer, Intel Technology India Private Limited, Bangalore)** for their valuable suggestions, feedback and providing me freedom to work in my way. Their immense knowledge, guidance and support have helped me a lot during my internship.

I would like to thank my manager **Mr. Manoj K Dadhich (Engineering Manager, Intel Technology),** for providing me great support and guidance during the internship.

I am highly indebted to my mentors **Mr. Mrugen J. Purohit (CAD Engineer, Intel Technology), Mr. Rajesh K. Vembu (CAD Engineer, Intel Technology)** for their constant support and utmost cooperation during my internship at Intel.

I also very thankful to **Dr. R. S. Kaler (Deputy Director, Thapar institute of Engineering and Technology, Patiala)**, **Dr. S. S. Bhatia (DOAA, Thapar institute of Engineering and Technology, Patiala)** and **Dr. Alpana Aggarwal (Head of ECED)** for allowing me to do an internship and carrying out my thesis work at **Intel Technology India Pvt. Ltd. Bangalore.** Last but not the least, I would like to thank my parents and friends for timely care and support to make me feel strong enough to complete this thesis.

AASHISH JOSHI

601662001

ABSTRACT

From the semiconductor chip having two transistors, to billions of transistors in a single chip, Very Large Scale Integration (VLSI) industry is growing continuously while obeying Moore's Law. While, product quality is the important criterion for a SoC, time-to-market (TTM) also plays a crucial role for the company to stay ahead of the competitors. While the smaller SoCs like smart phone take 6 months to get implemented, the implementation of server SoC is actually a time hogging step; and in general, it can span from one year to two years, which includes a series of RTL design, synthesis and place and route flow to run multiple times until the design converges. As we are concerned about the back-end design process, our area of interest limits to synthesis and place & route process for now.

The aim of our thesis work is to pavé a path for faster TTM. The first step towards achieving faster TTM or smaller runtime is to find out the reasons behind longer runtime which should be followed by modifications in the routines/methods used in synthesis, and place & route flow.

In our thesis work, a runtime abstraction utility is proposed and implemented to pick out the actual runtime hoggers in the physical design flows. The utility is able to abstract runtime information at different levels irrespective to the flow type into consideration. A graphical user interface (GUI) is also developed for the same utility which provides interactive way of running the utility.

As we know that, finding the actual runtime hogger module is the first step towards the path of providing faster TTM, the thesis work also analyzes a test design having a huge runtime; and provides a set of application variables introduced in the runtime hogger module to reduce the runtime for the synthesis and APR flow.

Keywords: *VLSI, SoC, TTM, RTL, GUI, SYN, APR.*

TABLE OF CONTENTS

CERTIFICATE II

DECLARATION III

ACKNOWLEDGEMENT IV

ABSTRACT V

LIST OF FIGURES VIII

LIST OF TABLES IX

ABBREVIATIONS AND ACRONYMS X

CHAPTER 1 INTRODUCTION 1

1.1 Sytem Design Approach 1

1.1.1 Programmable Processor 2

1.1.2 Hardware software co design 2

1.1.3 IP reuse 2

1.2 VLSI Design Cycle 2

1.2.1 System Specification 3

1.2.2 Architectural Design 3

1.2.3 Behavioral or Functional Design 3

1.2.4 Logic Design 3

1.2.5 Circuit Design 3

1.2.6 Physical Design 3

1.2.7 Physical Verification 4

1.2.8 Fabrication 5

1.2.9 Packaging, Testing and Debugging 5

1.3 Application Specific Integrated Circuit (ASICs) 5

1.3.1 Full-custom Design 5

1.3.2 Standard Cell based Design 5

1.3.3 Macro cells 6

1.3.4 Gate arrays 6

1.3.5 Field programmable gate arrays (FPGAs) 6

1.3.6 Structured-ASICs (channel-less gate arrays) 6

1.4 Basic EDA Terminology 7

1.5 Design Compiler® **8**
1.5.1 RTL Synthesis flow 8

1.6 Profiler **10**
1.6.1 Types of profiling 10

1.7 Tcl/Tk: A perspective **10**

1.8 Problem Statement **11**

1.9 Organization of the Thesis **12**

1.10 Chapter summary **12**

CHAPTER 2 LITERATURE SURVEY **13**

CHAPTER 3 PROFILER FOR APR AND SYNTHESIS FLOW **15**

3.1 Motivation behind the profiler/ Runtime abstraction utility **15**

3.2 A gUI ANALYZER for apr and synthesis flow **16**
3.2.1 Designed interface 16
3.2.2 Runtime abstraction utility 17

3.3 Sample Reports: **18**

3.4 Chapter summary **20**

CHAPTER 4 RESULT & DISCUSSION **21**

4.1 Finding the actual runtime hogger **21**

4.2 Running compile stage with two different Synopsys binaries **21**

4.3 Running the compile step with new sb and proposed app variables **24**

4.4 Entire Synthesis flow with new application variables, 6/8 cores with new SB: **27**

CHAPTER 5 CONCLUSION AND FUTURE SCOPE **30**

REFERENCES **32**

LIST OF FIGURES

Sr. No. **Figure Details** **Page No.**

Fig 1.1 SOC system Design Approach 1

Fig 1.2 VLSI Design Cycle 4

Fig 1.3 A standard-cell layout routed through feedthrough cell from A-A 6

Fig 1.4 Standard-cell layout with net A-A’ routed using over-the-cell (OTC) routing 6

Fig 1.5 Diode as a component 7

Fig 1.6 Synthesis Flow in Dc compiler with commands 9

Fig 1.7 Sample report generated by UNIX Profiler 10

Fig 3.1 Developed utility (Runtime Abstraction) before analysis 16

Fig 3.2 Developed utility after analysis 16

Fig 3.3 Logical tree of execution for import design stage 18

Fig 3.4 report generated after detailed analysis 19

Fig 3.5 Report generated after non detailed analysis 19

Fig 3.6 Snapshot of reports generated after critical analysis 20

Fig 4.1 Clustered data of cell count of test design with 8 cores and new SB 22

Fig 4.2 Clustered data of area report of test design with 8 cores and new SB 23

Fig 4.3 Clustered data for Design TNS, WNS and violating paths 23

Fig 4.4 Clustered data for cell count with new SB, 6/8 cores and introduced variables 25

Fig 4.5 Clustered data for area utilization with new SB, 6/8 core and introduced variables 26

Fig 4.6 Clustered data for TNS, WNS and violating path for 8/6 cores with new SB and proposed application variables 26

Fig 4.7 Clustered data of cell count for 6/8 cores, new SB and introduced variables for entire synthesis flow 28

Fig 4.8 Clustered data of area optimization for 6/8 cores, new SB and introduced variable 29

Fig 4.9 Clustered data of TNS, WNS and violating paths for 6/8 cores, new SB and introduced variables 29

LIST OF TABLES

Sr. No. **Table Details** **Page No.**

Table 4.1 Runtime and QoR report of compile stage with two different Synopsys binaries 21

Table 4.2 Runtime and QoR report of compile stage with two different Synopsys binaries and 8/6 cores with proposed application variables 24

Table 4.3 runtime and QoR data for TNS, WNS and violating path for 8/6 cores with new SB and proposed application variables 27

ABBREVIATIONS AND ACRONYMS

APR	Automatic Place and Route
ASIC	Application Specific IC
BGA	Ball Grid Array
CAD	Computer Aided Design
CISC	Complex Instruction Set Computing
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chemical Mechanical Polishing
CPU	Central Processing Unit
DIP	Dual Inline Packaging
DRAM	Dynamic Random Access Memory
DRC	Design Rule Check
EDA	Engineering Design Automation
ERC	Electrical Rule Checking
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HDL	Hardware Description Language
IC	Integrated Circuits
LE	Logical Elements
LUT	Look Up Table
LVS	Layout Vs. Schematic
MAS	Micro Architectural System
OTC	Over the Cell
PGA	Pin Grid Array
PVT	Process, Voltage, Temperature
QFP	Quad Flat Packaging
QoR	Quality of Results
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
RTL	Register Transfer Logic
SB	Synopsys Binaries
SYN	Synthesis
TTM	Time to Market
UNIX	UNIplexed Information and Computing System
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

VLSI	Very Large Scale Integration
w.r.t	With respect to
TCL	Tool Command Language
.mw	Milky way database
.ddc	Synopsys internal database format

CHAPTER 1
INTRODUCTION

VLSI industry has been growing with Moore's law [1, 2, 3], where a single VLSI chip can accommodate multiple processors. Designing of such huge systems is impossible without Electronic Device Automation (EDA) tools. Even though EDA tools are time efficient, but for a bigger system like Server, microprocessor etc., the design process becomes a time-hogging process. To monitor the amount of time spent at various abstraction level in design flow, Profiler is much needed. Yet, there are profilers available in UNIX which can profile system procedures, but there is no profiler available today, which can profile the entire design flow. This Chapter necessitates the requirement of such profiler while providing a brief information about the design flow.

1.1 SYTEM DESIGN APPROACH

A wealth of knowledge has been accumulated in the VLSI and application-specific system design community in the past two decades. Many such design strategies remain valid and are indeed very effective for SoC systems [4]. For a complex system design, divide and conquer policy is used as shown in figure 1.1. Considering our design is meant for digital applications, the implementation of algorithms can be divided into two parts, software codes on programmable processors and hardware accelarators [4]. The harware accelarators consists of intellectual properties (IP) and fully custom designed ICs.

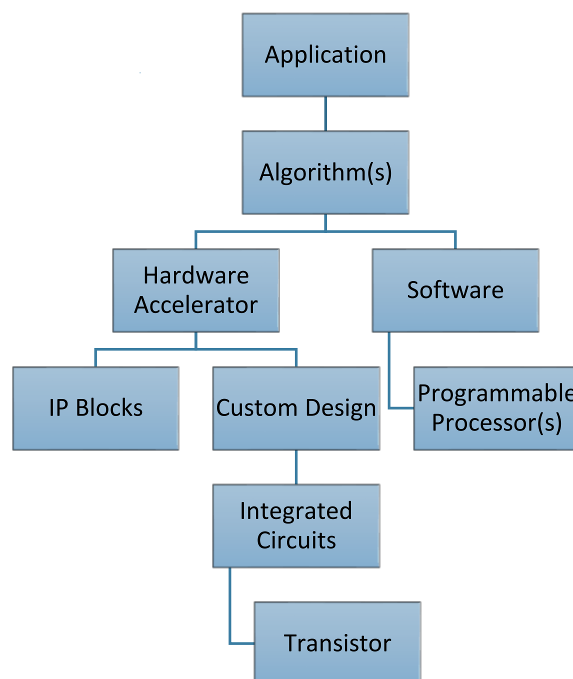

Fig 1.1 SOC system Design Approach

Major design elements in this strategy can be briefed as following,

1.1.1 PROGRAMMABLE PROCESSOR

To achieve high computational and energy efficiency and to fulfill the requirement of diversified computing, programmable processor approach is one of the strategies available. 90% of the SoC products in 130 nm technology have one programmable core [5]. With programmable cores, functional modification of a specific system can be done by software upgradation. RISC and/or DSP cores are popular implementations of such cores.

1.1.2 HARDWARE SOFTWARE CO DESIGN

Several applications can be split into two parts: (A) complex data-dependent and decision making routines, and (B) computation-intensive regular tasks. Programmable controller is dedicated to perform control-intensive tasks and, a dedicated hardware module for computation-intense tasks.

1.1.3 IP REUSE

Faster TTM is possible if the SoCs are built from previously designed blocks. Those previously designed high-performance IP, can save time and resources for a designer. The IP reuse is not only restricted to hardware reuse but also but also covers software domain by enabling the usage of reusable software.

1.2 VLSI DESIGN CYCLE

The information revolution has transformed our lives. It has changed our lifestyle, work and provided better entertainment tools. The revolution in Integrated Circuit (IC) technology has transformed computation and communication engineering and is used in computers, networks, memory and MEMs (Micro Electro Mechanical) systems.

The VLSI design cycle begins with a formal specification of a VLSI chip and ends with production of packaged chip. Figure 1.2 depicts the block diagram of VLSI design cycle which can be outlined as following.

1. System Specification
2. Architectural Design
3. Behavioral or Functional Design
4. Logic Design
5. Circuit Design
6. Physical Design
7. Fabrication
8. Packaging, Testing and Debugging

1.2.1 SYSTEM SPECIFICATION

System representation can be viewed as high level representation of the system. Performance, physical dimensions, functionality, fabrication technology and design techniques are taken in to account in this process, which results in specifications for the size, speed, power, and functionality of the VLSI system.

1.2.2 ARCHITECTURAL DESIGN

In this step basic architecture of the system is specified. Decisions are made to choose RISC (Reduced Instruction Set Computer) vs CISC (Complex Instruction Set Computer), number of arithmetic units, Floating Point units and pipelines. Architectural design generates a Micro-Architectural Specification (MAS). While MAS is basically a textual description, architects can accurately predict the performance, power and die size of the design based on such a description.

1.2.3 BEHAVIORAL OR FUNCTIONAL DESIGN

Main functional units are identified for a system. Identification of area, power and other parameter for each unit is done with interconnect requirements between them. A black- box overview is specified without specifying the internal structure. It’s a manual part in designing.

1.2.4 LOGIC DESIGN

In this step the control flow, word widths, register allocation, arithmetic operations, and logic operations of the design that represent the functional design are derived and tested[6], which is known as Register Transfer Level (RTL) description. RTL is manifested in a Hardware Description Language (HDL). Primary examples of such languages are Verilog or VHDL. This description can be used in simulation and verification. This description consists of Boolean expressions and timing information. For some special systems, logic design can be automated using *high level synthesis* tools capable of generating RTL description from behavioral description of the design.

1.2.5 CIRCUIT DESIGN

The purpose of circuit design is developing a circuit representation based on the logic design. RTL description is translated into a circuit representation while considering the speed and power requirements of the original design. *Circuit simulation* is used to verify the correctness and timing of each component. The netlist representation, obtained in this step can also be created automatically from RTL description by using *logic synthesis* tools.

1.2.6 PHYSICAL DESIGN

Obtained netlist is converted into a geometric representation of the circuit known as *layout.* Layout is created by converting logic components into geometric representation. Multiple layers are used for depicting the connections between them. Being a very complex process, Physical Design process is

broken down in various sub-steps. For the designs requiring faster time to market (TTM), layout is done automatically using Layout Synthesis tools. The layout of a high performance design (such as a microprocessor/ server) is done using manual design or semi- automated design.

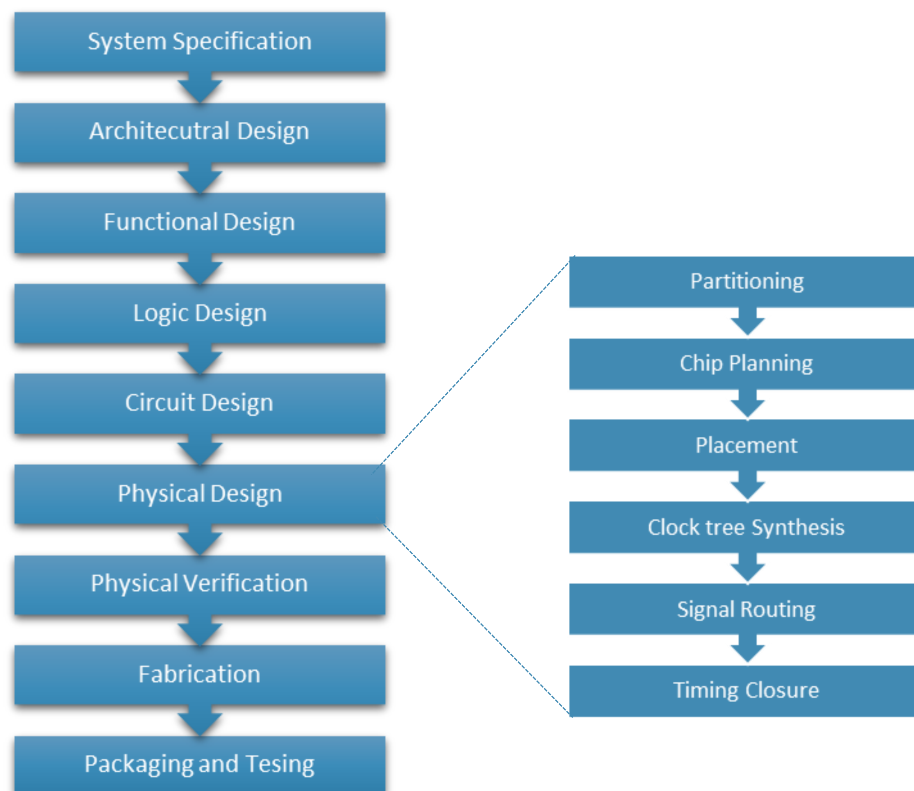

Fig 1.2 VLSI Design Cycle

1.2.7 PHYSICAL VERIFICATION

Several design rules are imposed to validate the correctness of the layout, which are based on process limitations. To authenticate the whole process, various validation and verification checks are performed.

- *Design rule checking* (*DRC*) verifies that the layout meets all technology imposed constraints. DRC also verifies layer density for *chemical-mechanical polishing* (*CMP*).
- *Layout vs. schematic* (*LVS*) checking verifies the functionality of the design. A derived netlist from the layout is compared with the original netlist produced from logic synthesis (circuit design).
- *Parasitic extraction* derives electrical parameters of the layout elements from their geometric representations; with the netlist, these are used to verify the electrical characteristics of the circuit.
- *Antenna rule checking* seeks to prevent *antenna effects*, which can damage transistor gates during plasma-etch steps through accumulation of excess charge on metal wires that are not connected to PN-junction nodes.

- *Electrical rule checking* (*ERC*) verifies the correctness of power and ground connections, and that signal transition times (*slew*), capacitive loads and *fanouts* are appropriately bounded.

1.2.8 FABRICATION

Layout data (GDSII stream format) is sent to fabrication on a tape and the data release is known as *Tape Out.* The data is converted into photo-lithographic masks. Masks are responsible to define the area of diffusion, deposition and even removal of the materials on wafer. A huge number of masks (Several dozen) may be used to complete the fabrication process.

1.2.9 PACKAGING, TESTING AND DEBUGGING

The wafer is diced after fabrication to separate individual chips. Different packaging techniques are used to package individual chips, which are then tested to ensure proper functionality. Following are the different packaging schemes used in VLSI design:

1. Dual In-line Package (DIP)
2. Pin Grid Array (PGA)
3. Ball Grid Array (BGA)
4. Quad Flat Package (QFP)

1.3 APPLICATION SPECIFIC INTEGRATED CIRCUIT (ASICS)

Application Specific Integrated Circuit (ASIC) is function specific ICs. The application field of ASICs varies from military, medical, satellite chips, ROM, DRAM, Cell phone chips etc. ASIC reduces the necessity of additional circuits, even one integrated chip can serve the purpose of different discrete chips. ASICs offer best design features w.r.t size, power, speed and reliability.

VLSI design can be categorized as *full-custom* and *semi-custom.* Full-custom design is basically associated with high-volume parts such as FPGAs and *microprocessors*. Semi-custom design reduces the complexity of the design flow which in-turn reduces the TTM and cost. *Cell-based* and *Array-based* semi-custom design styles are frequently used.

1.3.1 FULL-CUSTOM DESIGN

This technique results in highly optimized and compact chip. This style is error-prone, laborious and time consuming with low number of constraints. Full-custom scheme is primarily used for FPGAs and microprocessors.

1.3.2 STANDARD CELL BASED DESIGN

The digital standard cell has fixed size and functionality and is predefined. Standard-cell placement has less freedom which results in reduced complexity of design. This can reduce TTM at the cost of power efficiency or layout density. They are routed through either *feedthrough* (empty) cells or available routing tracks across rows. Available space between cell rows is termed as channels. *Over-*

the-cell (OTC) routing has turned to be most popular routing scheme and used up to 8-12 metal layers. Different routing schemes for standard cell are shown in figure 1.3 and 1.4.

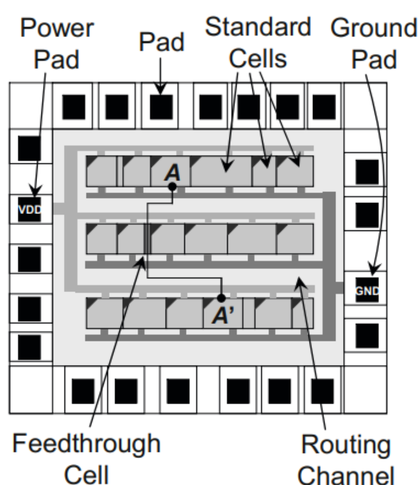

Fig 1.3 A standard-cell layout routed through feedthrough cell from A-A

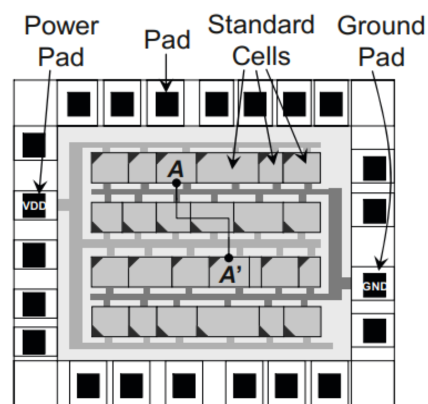

Fig 1.4 Standard-cell layout with net A-A' routed using over-the-cell (OTC) routing

1.3.3 MACRO CELLS

Macro cells have reusable functionality and range from simple to highly complex. They can be placed anywhere in the design and used for routing optimization. In several cases, the entire design can be assembled from existing macros to perform desired functionality.

1.3.4 GATE ARRAYS

Gate arrays are silicon chips with standard logic functionality. For e.g. NAND and NOR gates without connection. When chip requirements are known, routing layers are added. They can be produced at mass level due to their no predefined customization. They are cheaper and faster to produce as compared to standard and macro cell based designs and are used for low volume productions.

1.3.5 FIELD PROGRAMMABLE GATE ARRAYS (FPGAS)

FPGA comes with prefabricated but not-configured logic and interconnect components. *Logic elements (LEs)* are implemented using *lookup tables (LUTs),* while interconnects are configured using *switchboxes (SBs)* that join wires in adjacent routing channels. Due to the ability of customization without involving the fabrication facility, FPGAs has lower cost and faster TTM as compared to their counterparts. However, FPGAs typically run much slower and dissipate more power than ASICs [7].

1.3.6 STRUCTURED-ASICS (CHANNEL-LESS GATE ARRAYS)

A channel-less gate array is similar to an FPGA, except that the cells are usually not configurable. Unlike traditional gate arrays, sea of gate designs have many interconnect layers, removing the need for routing channels and thus improving density. Interconnects (sometimes, only via layers) are mask-

programmed in the foundry, and are not field-programmable. The modern incarnation of the channel-less gate array is the *structured ASIC*.

1.4 BASIC EDA TERMINOLOGY

A brief list of common terms used in EDA is as following. Many of these terms may be used subsequently in next chapters.

A *component* is the basic functional element of a circuits. For e.g. transistors, diode, resistors and capacitors. A diode as a component is shown in figure 1.5.

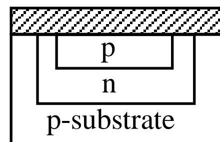

Fig 1.5 Diode as a component

A *module* can be viewed as a circuit partition or a collection of components.

A *block* is a module of defined shape/ boundary.

A *cell* is a logical and functional unit built from different components. A cell can be a gate, standard cell or macro.

A *pin* interfaces an electrical terminal to its external environment. At internal IC level, I/O pins exist on lower metal layers such as M1, M2 and M3.

A *layer* is a manufacturing process level in which design components are patterned. Different components are assigned to different layers at the time of physical design. These layers are routed according to netlist.

A *contact* is made between metal and silicon layer and used mostly inside cells.

A *via* is a connection between metal layers, usually to connect routing structures on different layers.

A *net* or *signal* is a set of pins or terminals that must be connected to have the same potential.

A *netlist* collects all the nets and components connected by them in the design, or, a list of nets and connecting pins of subdivision of the design. The netlist can be organized in two ways:

1. *Pin-oriented:* Each component is having a list of nets associated to it.
2. *Net*-oriented: Each net possesses a list mentioning associated design components.

A *net weight* is the numerical value assigned to a specific net, which shows its priority or criticality. Net weight helps in optimizing the net length, routing, and minimizing the distance between cells.

1.5 DESIGN COMPILER®

The Design compiler is the core of synthesis products provided by Synopsys. It optimizes designs to their smallest and fastest logical representation. HDL descriptions are converted into optimized, technology-dependent, gate-level designs. Figure 1.6 shows the synthesis flow in Design compiler with relevant commands.

1.5.1 RTL SYNTHESIS FLOW

Synthesis flow can be done in the following steps:

1. Develop the HDL files: design files are written in a hardware description language, VHDL or Verilog.
2. Specification of libraries: physical, link, target libraries are specified.
3. Reading the design: HDL compiler is used to read RTL designs, gate-level netlists.
4. Defining design environment: defining the environment refers to setting up the external operating conditions such as process, voltage and temperature (PVT) and loads, drivers, fanouts etc.
5. Setting up the design constraints: constraints are defined using several native commands as,
    a. set_max_transition
    b. set_max_fanout
    c. set_max_capacitance
    d. create_clock
    e. set_clock_latency
    f. set_propogated_clock
    g. set_clock_uncertainity
    h. set_clock_transision
    i. set_input_delay
    j. set_output_delay
    k. set_max_area
6. Select compile strategy: depending on the design, compile strategy can be chosen either bottom-up or top-bottom, or mixed.
7. Synthesize: compile or compile_ultra command is used for synthesizing the design.
8. Scan-chain insertion
9. Incremental synthesis
10. Resolving design process: timing, area, constraint, and timing reports are used as guidelines to optimize the design.
11. Saving the design: .ddc, .MW, or Verilog format is used to save the design.

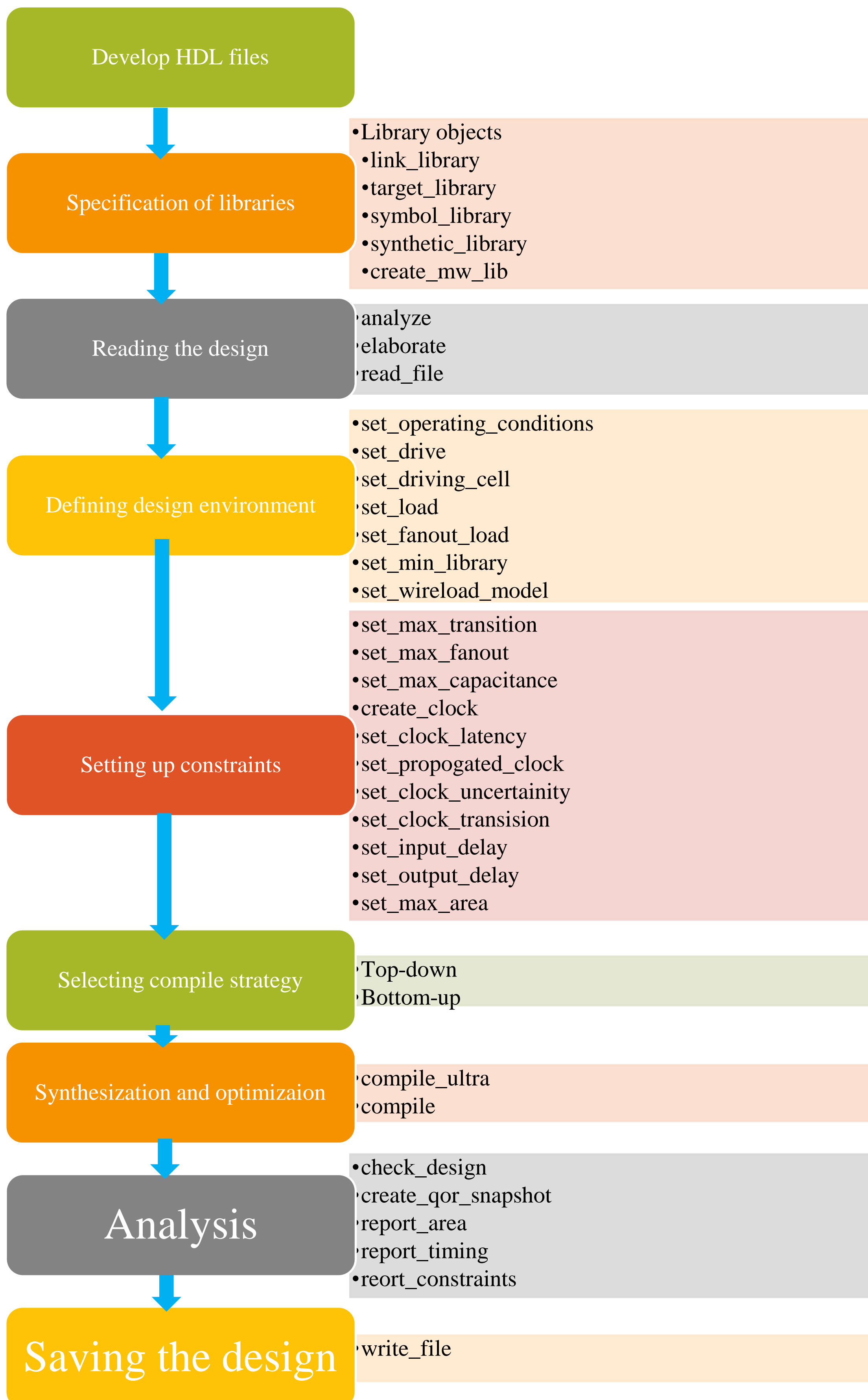

Fig 1.6 Synthesis Flow in Dc compiler with commands

1.6 PROFILER

Profiling always impacts the performance of the application being analyzed [8]. While the traditional profiling tools perform great on the host language code itself, they usually fail to provide meaningful results if the user builds and use abstractions on top of the host language. Thus a dedicating tool is required to profile the process.

Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters. Profilers are used in the performance engineering process.

They are critical to understand the program behavior. With the help of such tool, CAD engineers can estimate, how the procedures will work on new designs. A statistical example profiler report is shown in figure 1.7.

```
/* ------------ source------------------------- count */
0001             IF X = "A"                     0055
0002                THEN DO
0003                  ADD 1 to XCOUNT           0032
0004                ELSE
0005             IF X = "B"                     0055
```

Fig 1.7 Sample report generated by UNIX Profiler

A well-known profiler gprof*accounts for the runtime of the routines [9]. The gprof design takes advantage of the fact that the measured procedures are large, hierarchical and well structured.

1.6.1 TYPES OF PROFILING

Profilers can be distinguished in two different categories,

1. Those that presents counts of statement or routine invocations
2. Those who display timing information about the statement and routines.

Profiler can also provide memory usage of the procedures/ routines. Execution count are not necessarily proportional to the amount of time required to execute the routine or statement [9]. Further, the routine execution time will not be the same for all calls on the routine.

1.7 TCL/TK: A PERSPECTIVE

TCL is a high level programming language. It provides object oriented functions like Java and C++. The idea behind the development of TCL is to make is very simple but powerful language. Furthermore it is capable of providing imperative and functional and procedural styles.

As a command line language it is mostly used in EDA and embedded system platforms. Its union with TK is referred as TCL/TK, which helps to create graphical user interface (GUI) inherently in TCL. It was coined in 1988 and the designer was John Ousterhout who is also known for Magic VLSI

*gprof: *a Call Graph Execution Profiler, a performance analysis tool for* UNIX *applications.*

computer aided design. Tcl also supports the powerful regular expressions, which is really helpul while developing the domain specific profilers.

1.8 PROBLEM STATEMENT

The VLSI industry is advancing at a very high rate. It's impossible to be in the race as a semiconductor company, without reducing the expected time to market (TTM). Every semiconductor vendor has their own philosophy and algorithms behind the design process. In absence of better and fast EDA algorithms, a simple IC can take 2 to 3 years to get fabricated and released. This enables research in the field of tool optimization and runtime reduction of the different EDA flows (Synthesis, automatic place & route).

Developing a profiler is the basic requirement in order to reduce design process runtime. Yet, there are profilers like gprof* available to profile procedures in UNIX, there is no profiler present today, which can profile the entire design flow.

Designing is usually done using a Design System, which comprises of methodologies resulting in flows and which is written around design tools. TCL has commonly used language for methodologies and flows. Based on the various scenarios and corner cases, methodologies and flows become heavyweight and eventually buried under various layers of enhancements. This eventually results in huge runtime penalties, due to intricate calling & dependency mechanisms.

This necessitates the requirement for creating profilers which can help in analyzing the various procedures/methods for the design process. Profilers can be dynamic and static depending on the analysis methods and how they are integrated into the design flow. In this work, we are exploring static profilers, which can help in analyzing the design system Environment. The usual metrics which include invocation count, the runtime of the procedures etc. can help in analyzing the runtime hogging procedures. The thesis presents the following contributions from this work:

1. Develop Analyzing scripts which are capable and agnostic to the flow type, like Logical Synthesis, Physical Synthesis, Design Planning etc.,
2. Develop graphical interface for plotting the runtime hoggers based on various mechanisms.
3. Identify the runtime hogging procedures and isolate the critical lines which can reduce the runtime.
4. Develop rule system which can be used to identify critical sections in the procedures.
5. Identify the opportunity for tool enhancements which include newer tool binaries, tool options for better runtime.
6. Identify the dependency on the use of machines which include the CPUs/threads and also the memory.

1.9 ORGANIZATION OF THE THESIS

Chapter I summarizes the problem statement of the work, and provides an overview of physical design process, profiler and different tools used in physical design.

Chapter II has literature survey related to research work.

Chapter III discusses about the development of proposed runtime profiler, its algorithm and statistics.

Chapter IV contains different reports generated by the utility, provides different graphs, and a set of application variables to reduce runtime for a test design.

Chapter V concludes the thesis and provides future scope of the thesis.

1.10 CHAPTER SUMMARY

This chapter explains the basics of VLSI physical design cycle, EDA terminology and application specific Integrated circuits (ASICs). This chapter discusses the profiling of procedures, the language used to create a domain specific profiler for synthesis and APR flow in our thesis. The chapter also provides an overview of design compiler tool used for synthesis of ICs. Problem statement sections shed light on the proposed thesis topic and its structure.

CHAPTER 2
LITERATURE SURVEY

This chapter provides an overview of the literature studied. These literatures vary from VLSI design to program profilers.

Chen YK and Kung SY [4], have proposed different strategies for system designs. The cause-effect relationship between different aspects of SOC design has been proposed. Design strategies like programmable coprocessor, IP reuse, and hardware-software co-design are also discussed in [4], which are the most used design techniques today.

Kriaa L *et al.* [5] proposed parallel programming model for abstracting both hardware and software interface for a heterogeneous multiprocessorSoC(MPSoC). A comparison between MPSoC and classic computers has been done. This paper presents hardware-software architecture for MPSoC with a programming model which necessitates the development of new design flows. A case study of Microccmmodel is also carried out in.

Sherwani NA [6] provided detailed concepts of the physical design cycle. Several methodologies for the physical design were also proposed and were compared against cell size, cell type, and placement. New trends in VLSI design were also discussed in [6].New design and packaging styles of VLSI design were also discussed.

The difference between a 90-nm CMOS field programmable gate array (FPGA) and 90-nm CMOS standard-cell application-specific integrated circuits (ASICs) is presented in terms of speed, logic density, and power consumption. The system engineers are enabled to make choices between FPGA and ASIC, and the shortcomings of FPGAs are targeted to improve FPGAs. It also discusses the advantage of using hard blocks in modern FPGAs, which can reduce the average area gap to below five [7].

Ressia J *et al.* [8] provided the idea of dedicated profiling tools which enables developers to quickly prototype new profilers in domain-specific languages and models. Feasibility of domain-specific profiler was also discussed. They argued that the use of traditional profilers is inadequate as they are concerned with the source code only and demonstrated the drawbacks of them using two test cases. They formulated the requirements of the domain-specific profilers in this work.

Graham SL *et al.* [9] illustrated squeezing of last bits of performance out of UNIX programs. The first ever developed UNIX profiler was discussed. A new profiler gprof was proposed and discussed. The proposed profiler was able to provide different abstraction level for a program. For each level of programme decomposition, the profiler assesses the cost of routine. Without any pre-planning, the

developer can use a profiler and can compile it into the program. The profiler adds an overhead of 5 to 30% to the profiled program.

Tanter E *et al.* [10] depicted preliminary benchmarks and examples to support behavioral reflection. For Java, they described a reflective architecture which offers appropriate interfaces for static and dynamic configuration of partial reflection at different abstraction levels. Reflex which is an open reflective extension for implementing proposed architecture is also described in [10] and it is capable of integrating runtime and load-time behavioral reflection.

Renggli *et al.* [11] advocated the use of dynamic grammar for programming languages. Petitparser was introduced which combines the best qualities of four different parsers, which include ideas from scannerless parsing, parser combinations, parsing expression grammars and packrat parsers to model grammars and parsers as objects. A basic performance analysis has been done which advocates the usage of grammars which is accessible and changeable at runtime.

CHAPTER 3

PROFILER FOR APR AND SYNTHESIS FLOW

This chapter includes motivation behind the developed profiler for the entire design flow. It also provides algorithm to abstract runtime from the input log files. The GUI for the developed profiler is depicted and different sample output reports are presented in this chapter.

3.1 MOTIVATION BEHIND THE PROFILER/ RUNTIME ABSTRACTION UTILITY

Recent advances in domain-specific languages and models reveal a drastic change in the way software is being built. The software engineering community has seen a rapid emergence of domain-specific tools, ranging from tools, to easily build domain-specific languages, to transform models [10], to check source code [11], and to integrate development tools. In the paradigm of generative programming, the generation of programs from specifications forms a key part of the software engineering process. The design of a program is improved by restructuring it. Migration and reverse engineering are the other application of program transformation.

The idea of profiling tool flow in EDA is ideally based on reverse engineering, which can be used to detect the critical procedures used and to restructure them. More specifically, the profiler should be developed in order to get the essential data for optimizing re-optimizing the design process. The developed profiler should be capable of providing following features:

1. The design process is a stepwise process. The designers/ tool developers look into log files for required information, which requires a huge amount of time and manual effort. Thus the profiler should be able to provide three level of abstraction as,
    a. Detailed (stepwise)
    b. Non- detailed (stage wise)
    c. Critical (flow wise)
2. Agnostic to the flow type: Either automatic place or route (APR) or synthesis (SYN), the profiler should be capable of abstracting the required information from the logs.
3. The motive behind the profiler is to find out critical procedures in the flow, thus the profiler should provide tabular data of a routine's runtime or timing information and the counts of routine invocation.

Using the existing design environment which is based on the Synopsys tools, the analyzers are being coded to profile log outputs from the implementation.

3.2 A GUI ANALYZER FOR APR AND SYNTHESIS FLOW

3.2.1 DESIGNED INTERFACE

Figure 3.1 shows the GUI interface developed with TCL/TK graphics. This can be invoked by just sourcing the file created. The utility displays a message after analysis, which is shown in figure 3.2.

How to start: tclsh widgetrun.tcl

Exit: Button is given at the bottom as shown in figure.

The figure shows the GUI after running detailed analysis. A message appears on the successful comparison.

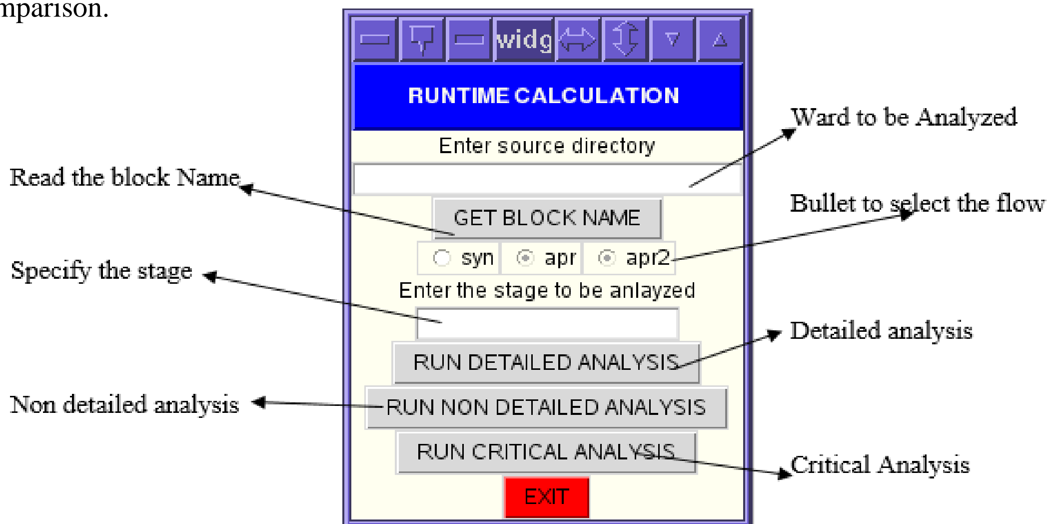

Fig 3.1 Developed utility (Runtime Abstraction) before analysis

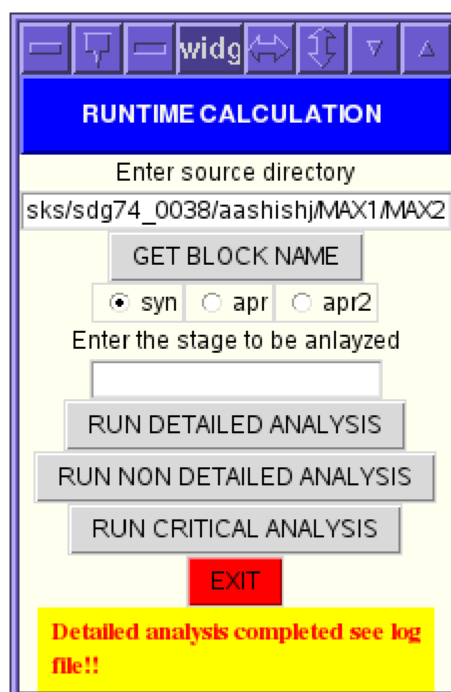

Fig 3.2 Developed utility after analysis

3.2.2 RUNTIME ABSTRACTION UTILITY

ASSUMPTIONS:

1. One stage can have similar steps too.
2. One step can call a procedure multiple times.

Algorithm runtimeAbstraction (log file generated in different steps of physical design)

Output: Clustered procedures for optimization, 3-level abstracted profile Matrices

1 Construct dependency tree after reading the log file

2 Construct profile Matrices using dependency tree

3 Get modulated matrices from above profile matrices combinations

4 Cluster the gathered data

The dependency tree is constructed using constructTree algorithm. A logical representation of such tree is shown in figure, whose tabular form is shown in figure.

Algorithm constructTree (log file)

Output: dependency Tree

1. Read log file
2. Collect all procedure names from the log file using regular expressions.
3. Take out the step information from the log file using regular expressions.
4. For each step

   Grab procedure name and time of execution, make a temporary list with the name and time of execution after converting it to seconds.

   Append list until the step data is parsed fully.
5. For each procedure collected in list

   Create another list with the procedure and the 'time of execution', subtracted from the time of execution of the last procedure, which is the duration of that procedure.
6. Sort the new list according to procedure name, if a procedure is getting called more than once add its duration value with the previous value.
7. Convert the duration into DD:HH:MM:SS.
8. The final list will contain procedure names and their duration in the step, hence the Tree.

A logical representation of created tree is depicted in figure 3.3. A number of smaller modules were also created to provide a suitable code for the utility which includes, two time conversion utilities, a module to append all log files.

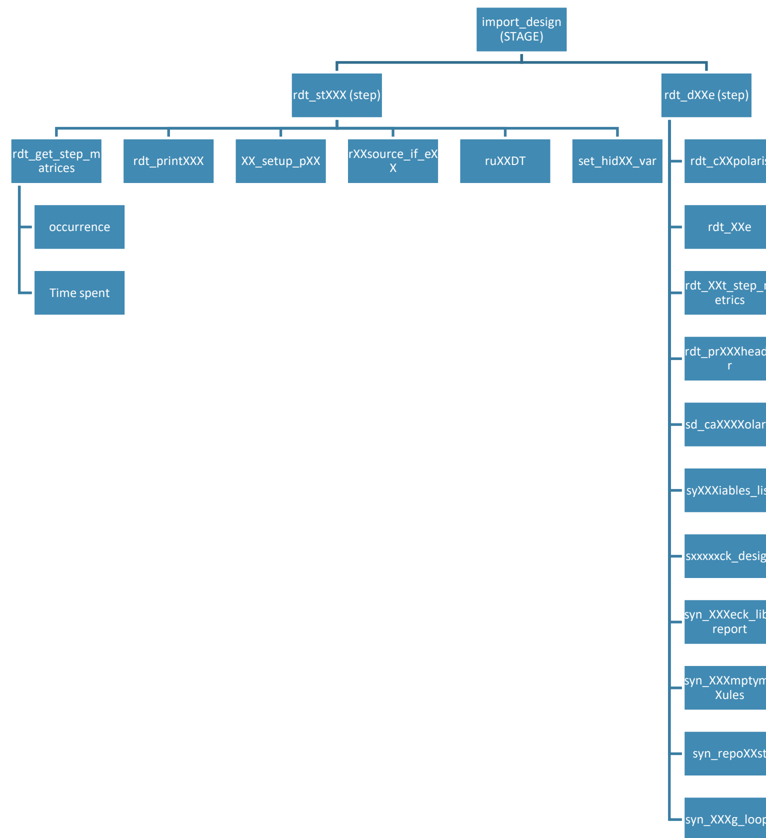

Fig 3.3 Logical tree of execution for import design stage

3.3 SAMPLE REPORTS:

The utility has the following output structure,

1. Detailed Analysis: A step wise approach is applied to catch a runtime hogging procedure. The output file is reported out which provides detailed information of a step which includes total runtime, step runtime, memory occupied and runtime of each procedure used in it. Figure 3.4 shows the output of detailed analysis.
2. Non-Detailed analysis: The log is read stage wise, and a report is generated which carries the information about procedures invoked in the entire stage. Figure 3.5 depicts output log generated after Non-Detailed analysis.
3. Critical analysis: All stage log files are concatenated and parsed in the profiler, this analysis provides a top view approach to the process. It enables users to identify the run time hoggers and provides two discrete reports as,
    a. Procedure report for entire flow
    b. Top 10 run time hoggers

Figure 3.6 shows two different reports generated after running critical analysis.

```
---------------------------------------------------------------------------------
STEP                PROCEDURE                                COUNT     EXECUT_TIME
---------------------------------------------------------------------------------
syn_compile         (PRE.compile.syn_compile.tcl)            10        0 days:0 hrs:32 mins:22 sec
syn_compile         ::POST_compile_syn_compile               2         0 days:0 hrs:0 mins:1 sec
syn_compile         ::rdt_get_step_metrics                   3         0 days:0 hrs:0 mins:0 sec
syn_compile         ::rdt_print_step_header                  2         0 days:0 hrs:0 mins:0 sec
syn_compile         ::rdt_source_if_exists                   13        0 days:0 hrs:0 mins:4 sec
syn_compile         ::runRDT                                 2         0 days:0 hrs:0 mins:11 sec
syn_compile         ::set_hidden_app_var                     2         0 days:0 hrs:0 mins:0 sec
syn_compile         ::special_cells_set_ideal_nets           2         0 days:0 hrs:0 mins:0 sec
syn_compile         ::syn_check_ctech_cells                  3         0 days:0 hrs:0 mins:1 sec
syn_compile         ::syn_compile                            12        0 days:19 hrs:57 mins:58 sec
syn_compile         ::syn_compress_long_design_name          1         0 days:0 hrs:0 mins:0 sec
syn_compile         ::syn_dt_clkbuild_wrappers               2         0 days:0 hrs:0 mins:0 sec
syn_compile         ::syn_handle_boundary_optimization       3407      0 days:0 hrs:0 mins:17 sec
syn_compile         ::syn_handle_boundary_optimization_extract_dfx_controls2        0 days:0 hrs:0 mins:0 sec
syn_compile         ::syn_handle_boundary_optimization_extract_dfx_handler3        0 days:0 hrs:0 mins:2 sec
syn_compile         ::syn_handle_boundary_optimization_extract_dw3        0 days:0 hrs:1 mins:19 sec
syn_compile         ::syn_handle_boundary_optimization_extract_hip3        0 days:0 hrs:0 mins:2 sec
syn_compile         ::syn_handle_boundary_optimization_extract_mco2        0 days:0 hrs:0 mins:17 sec
syn_compile         ::syn_handle_boundary_optimization_extract_upf2        0 days:0 hrs:0 mins:0 sec
syn_compile         ::syn_handle_boundary_optimization_extract_visa2        0 days:0 hrs:0 mins:3 sec
syn_compile         ::syn_idn_cg_cells                       205       0 days:0 hrs:2 mins:36 sec
syn_compile         ::syn_mark_ctech_cells                   1         0 days:0 hrs:0 mins:1 sec
syn_compile         ::syn_outputs_ddc                        1         0 days:0 hrs:5 mins:37 sec
syn_compile         ::syn_remove_rc_scaling_for_opt          1         0 days:0 hrs:0 mins:0 sec
syn_compile         ::syn_swap_wn_to_ln_cells                1         0 days:0 hrs:0 mins:1 sec
---------------------------------------------------------------------------------
Memory: 39.957 MB   Total Memory: 34.386 GB                   Duration: 20:40:51
```

Fig 3.4 report generated after detailed analysis

```
---------------------------------------------------------------------------------
STAGE             PROCEDURE/FILE SOURCED                                 COUNT     EXECUT_TIME
---------------------------------------------------------------------------------
Total Duration: 30:39:41; Duration: 21:01:48
compile           ::syn_compile                                          12        0 days:19 hrs:57 mins:58 sec
compile           (PRE.compile.syn_compile.tcl)                          10        0 days:0 hrs:32 mins:22 sec
compile           ::syn_outputs_ddc                                      2         0 days:0 hrs:6 mins:49 sec
compile           ::rdt_get_qor_data                                     1         0 days:0 hrs:4 mins:1 sec
compile           ::rdt_change_names                                     11        0 days:0 hrs:3 mins:17 sec
compile           ::syn_idn_cg_cells                                     205       0 days:0 hrs:2 mins:36 sec
compile           ::rpt::create_run_summary                              6         0 days:0 hrs:2 mins:17 sec
compile           ::syn_clock_gating                                     6         0 days:0 hrs:1 mins:37 sec
compile           ::syn_handle_boundary_optimization_extract_dw          3         0 days:0 hrs:1 mins:19 sec
compile           ::syn_reports_power_domain_summary                     1         0 days:0 hrs:1 mins:9 sec
compile           ::syn_reports_qor                                      1         0 days:0 hrs:1 mins:1 sec
compile           ::rdt_dont_use                                         5793      0 days:0 hrs:1 mins:0 sec
compile           ::runRDT                                               26        0 days:0 hrs:1 mins:0 sec
compile           ::syn_reports_area                                     1         0 days:0 hrs:0 mins:49 sec
compile           ::rdt_run_checks                                       5         0 days:0 hrs:0 mins:39 sec
compile           ::rdt_print_step_header                                42        0 days:0 hrs:0 mins:33 sec
compile           ::syn_outputs                                          2         0 days:0 hrs:0 mins:21 sec
compile           ::rdt_visa_rtl_check                                   35        0 days:0 hrs:0 mins:18 sec
compile           ::syn_outputs_sdc                                      1         0 days:0 hrs:0 mins:18 sec
compile           ::rdt_call_polaris                                     3         0 days:0 hrs:0 mins:17 sec
compile           ::sd_call_polaris                                      3         0 days:0 hrs:0 mins:17 sec
compile           ::syn_handle_boundary_optimization                     3407      0 days:0 hrs:0 mins:17 sec
compile           ::syn_handle_boundary_optimization_extract_mco         2         0 days:0 hrs:0 mins:17 sec
compile           ::syn_outputs_verilog                                  1         0 days:0 hrs:0 mins:17 sec
compile           ::syn_reports_check_mv_design                          1         0 days:0 hrs:0 mins:16 sec
compile           ::syn_outputs_upf                                      1         0 days:0 hrs:0 mins:11 sec
compile           ::syn_outputs_saif_mapping                             3         0 days:0 hrs:0 mins:10 sec
compile           ::rdt_source_if_exists                                 28        0 days:0 hrs:0 mins:7 sec
compile           (PRE.compile.syn_clock_gating.tcl)                     2         0 days:0 hrs:0 mins:6 sec
compile           ::rdt_done                                             10        0 days:0 hrs:0 mins:5 sec
compile           ::syn_get_power_domain_info                            17        0 days:0 hrs:0 mins:5 sec
compile           ::rdt_setup_pdk                                        1         0 days:0 hrs:0 mins:4 sec
compile           ::syn_reports_check_timing                             1         0 days:0 hrs:0 mins:4 sec
compile           ::syn_reports_resources                                1         0 days:0 hrs:0 mins:4 sec
compile           ::find_nested_PD                                       13        0 days:0 hrs:0 mins:3 sec
compile           ::syn_handle_boundary_optimization_extract_visa        2         0 days:0 hrs:0 mins:3 sec
compile           ::syn_reports_worst_min                                1         0 days:0 hrs:0 mins:3 sec
compile           ::syn_handle_boundary_optimization_extract_dfx_handler 3         0 days:0 hrs:0 mins:2 sec
compile           ::syn_handle_boundary_optimization_extract_hip         3         0 days:0 hrs:0 mins:2 sec
compile           ::POST_compile_syn_compile                             2         0 days:0 hrs:0 mins:1 sec
compile           ::set_hidden_app_var                                   38        0 days:0 hrs:0 mins:1 sec
compile           ::stamp_clock_network                                  2         0 days:0 hrs:0 mins:1 sec
compile           ::start_svf_vsdc                                       2         0 days:0 hrs:0 mins:1 sec
compile           ::stop_svf_vsdc                                        1         0 days:0 hrs:0 mins:1 sec
compile           ::syn_check_ctech_cells                                3         0 days:0 hrs:0 mins:1 sec
compile           ::syn_mark_ctech_cells                                 1         0 days:0 hrs:0 mins:1 sec
compile           ::syn_reports_isolation_cell_details                   1         0 days:0 hrs:0 mins:1 sec
compile           ::syn_reports_threshold_voltage_group                  1         0 days:0 hrs:0 mins:1 sec
compile           ::syn_reports_worst_max                                1         0 days:0 hrs:0 mins:1 sec
compile           ::syn_swap_wn_to_ln_cells                              1         0 days:0 hrs:0 mins:1 sec
```

Fig 3.5 Report generated after non detailed analysis

```
compile
---------------------------------------------------------------------------
STAGE               PROCEDURE/FILE SOURCED                                     COUNT     EXECUT_TIME
---------------------------------------------------------------------------
Total Duration:  30:39:41;  Duration:  21:01:48
compile             ::syn_compile                                              12        0 days:19 hrs:57 mins:58 sec
compile             (PRE.compile.syn_compile.tcl)                              10        0 days:0 hrs:32 mins:22 sec
compile             ::syn_outputs_ddc                                          2         0 days:0 hrs:6 mins:49 sec
compile             ::rdt_get_qor_data                                         1         0 days:0 hrs:4 mins:1 sec
compile             ::rdt_change_names                                         11        0 days:0 hrs:3 mins:17 sec
compile             ::syn_idn_cg_cells                                         205       0 days:0 hrs:2 mins:36 sec
compile             ::rpt::create_run_summary                                  6         0 days:0 hrs:2 mins:17 sec
compile             ::syn_clock_gating                                         6         0 days:0 hrs:1 mins:37 sec
compile             ::syn_handle_boundary_optimization_extract_dw              3         0 days:0 hrs:1 mins:19 sec
compile             ::syn_reports_power_domain_summary                         1         0 days:0 hrs:1 mins:9 sec
compile             ::syn_reports_qor                                          1         0 days:0 hrs:1 mins:1 sec
compile             ::rdt_dont_use                                             5793      0 days:0 hrs:1 mins:0 sec
compile             ::runRDT                                                   26        0 days:0 hrs:1 mins:0 sec
compile             ::syn_reports_area                                         1         0 days:0 hrs:0 mins:49 sec
compile             ::rdt_run_checks                                           5         0 days:0 hrs:0 mins:39 sec
compile             ::rdt_print_step_header                                    42        0 days:0 hrs:0 mins:33 sec
compile             ::syn_outputs                                              2         0 days:0 hrs:0 mins:21 sec
compile             ::rdt_visa_rtl_check                                       35        0 days:0 hrs:0 mins:18 sec
compile             ::syn_outputs_sdc                                          1         0 days:0 hrs:0 mins:18 sec
compile             ::rdt_call_polaris                                         3         0 days:0 hrs:0 mins:17 sec
compile             ::sd_call_polaris                                          3         0 days:0 hrs:0 mins:17 sec
compile             ::syn_handle_boundary_optimization                         3407      0 days:0 hrs:0 mins:17 sec
compile             ::syn_handle_boundary_optimization_extract_mco             2         0 days:0 hrs:0 mins:17 sec
compile             ::syn_outputs_verilog                                      1         0 days:0 hrs:0 mins:17 sec
compile             ::syn_reports_check_mv_design                              1         0 days:0 hrs:0 mins:16 sec
compile             ::syn_outputs_upf                                          1         0 days:0 hrs:0 mins:11 sec
compile             ::syn_outputs_saif_mapping                                 3         0 days:0 hrs:0 mins:10 sec
compile             ::rdt_source_if_exists                                     28        0 days:0 hrs:0 mins:7 sec
compile             (PRE.compile.syn_clock_gating.tcl)                         2         0 days:0 hrs:0 mins:6 sec
compile             ::rdt_done                                                 10        0 days:0 hrs:0 mins:5 sec
compile             ::syn_get_power_domain_info                                17        0 days:0 hrs:0 mins:5 sec
compile             ::rdt_setup_pdk                                            1         0 days:0 hrs:0 mins:4 sec
compile             ::syn_reports_check_timing                                 1         0 days:0 hrs:0 mins:4 sec
compile             ::syn_reports_resources                                    1         0 days:0 hrs:0 mins:4 sec
compile             ::find_nested_PD                                           13        0 days:0 hrs:0 mins:3 sec
compile             ::syn_handle_boundary_optimization_extract_visa            2         0 days:0 hrs:0 mins:3 sec
compile             ::syn_reports_worst_min                                    1         0 days:0 hrs:0 mins:3 sec
compile             ::syn_handle_boundary_optimization_extract_dfx_handler     3         0 days:0 hrs:0 mins:2 sec
compile             ::syn_handle_boundary_optimization_extract_hip             3         0 days:0 hrs:0 mins:2 sec
compile             ::POST_compile_syn_compile                                 2         0 days:0 hrs:0 mins:1 sec
compile             ::set_hidden_app_var                                       38        0 days:0 hrs:0 mins:1 sec
compile             ::stamp_clock_network                                      2         0 days:0 hrs:0 mins:1 sec
compile             ::start_svf_vsdc                                           2         0 days:0 hrs:0 mins:1 sec
compile             ::stop_svf_vsdc                                            1         0 days:0 hrs:0 mins:1 sec
compile             ::syn_check_ctech_cells                                    3         0 days:0 hrs:0 mins:1 sec
compile             ::syn_mark_ctech_cells                                     1         0 days:0 hrs:0 mins:1 sec
compile             ::syn_reports_isolation_cell_details                       1         0 days:0 hrs:0 mins:1 sec
compile             ::syn_reports_threshold_voltage_group                      1         0 days:0 hrs:0 mins:1 sec
compile             ::syn_reports_worst_max                                    1         0 days:0 hrs:0 mins:1 sec
compile             ::syn_swap_wn_to_ln_cells                                  1         0 days:0 hrs:0 mins:1 sec


-------------------------------------------------------------------------------------
                                      Top 10 run time  hoggers
-------------------------------------------------------------------------------------
::syn_compile_incr                                     17                  1 days:0 hrs:4 mins:24 sec
::syn_compile                                          12                  0 days:19 hrs:57 mins:58 sec
(m2upiddp_path_groups.tcl)                             4                   0 days:5 hrs:32 mins:39 sec
::rdt_source_if_exists                                 238                 0 days:3 hrs:40 mins:8 sec
::rpt::create_run_summary                              79                  0 days:1 hrs:7 mins:40 sec
::syn_outputs_ddc                                      14                  0 days:0 hrs:37 mins:58 sec
(PRE.compile.syn_compile.tcl)                          10                  0 days:0 hrs:32 mins:22 sec
(POST.constraints.rdt_timing_constraints.tcl)          2                   0 days:0 hrs:30 mins:37 sec
(PRE.compile_incr.rdt_start.tcl)                       4                   0 days:0 hrs:27 mins:50 sec
::rdt_get_qor_data                                     6                   0 days:0 hrs:25 mins:52 sec
~
~
~
```

Fig 3.6 Snapshot of reports generated after critical analysis

3.4 CHAPTER SUMMARY

This chapter necessitates the usage of runtime abstraction utility and provides the algorithm behind this utility. It also shows sample reports generated by the developed utility and carries critical information of runtime hogging procedures to the next stage.

CHAPTER 4
RESULT & DISCUSSION

This chapter explains different results and reports with several proposals to reduce runtime. These proposal include inclusion of application variables in the compile command of ::syn_compile procedure, using more cores for running compile stage, using newer version of synopsys binaries (SB).

4.1 FINDING THE ACTUAL RUNTIME HOGGER

The critical analysis provided the list of top 10 actual run hoggers which are shown in figure 3.6. Section 4.2, 4.3, 4.4 describe the methodologies to reduce Logic synthesis runtime for the test design.

4.2 RUNNING COMPILE STAGE WITH TWO DIFFERENT SYNOPSYS BINARIES

Following table provides the runtime, QOR (quality of regression) data for the test design.

Table 4.1 Runtime and QoR report of compile stage with two different Synopsys binaries

Parameters	Baseline/ reference design with old SB	Test design with 8 cores new SB
Run Time	21:01:48	22:32:05
Cell Count		
Hierarchical Cell Count:	7120	7157
Hierarchical Port Count:	264088	264244
Leaf Cell Count:	560256	489348
Buf/Inv Cell Count:	181906	126386
Buf Cell Count:	46562	54225
Inv Cell Count:	135344	72161
CT Buf/Inv Cell Count:	1399	1364
Combinational Cell Count:	469653	398895
Sequential Cell Count:	90603	90034
Macro Count:	37	419
Area		
Combinational Area:	57476.7	53655.2
Noncombinational Area:	54846.1	54216.5
Buf/Inv Area:	18662.4	16150.7
Total Buffer Area:	7166	8684.7
Total Inverter Area:	11496.4	7466
Macro/Black Box Area:	53769.6	53829.9
Net Area:	0	0
Net XLength :	3648501.8	3886173.8
Net YLength :	4720498.5	4029830.5

Cell Area:	166092.4	161701.6
Design Area:	166092.4	161701.6
Net Length :	8369000	7916004
Design Rules		
Total Number of Nets:	617501	551928
Nets With Violations:	8959	1850
Max Trans Violations:	8508	1132
Max Cap Violations:	241	622
Max Fanout Violations:	445	204
Compile CPU Statistics		
Resource Sharing:	9437.5	12395.2
Logic Optimization:	3809.4	30611.6
Mapping Optimization:	250826.8	307090.6
Overall Compile Time:	277106.7	368539.4
Overall Compile Wall Clock Time:	70165.2	76886.5
Design		
Number of Violating Paths	17420	7362
TNS	1388714.4	232667.1
WNS	445.1	314.7
Design (Hold)		
WNS	3454.4	3445.8
TNS	6408507	6445084.5
Number of Violating Paths	77627	85840

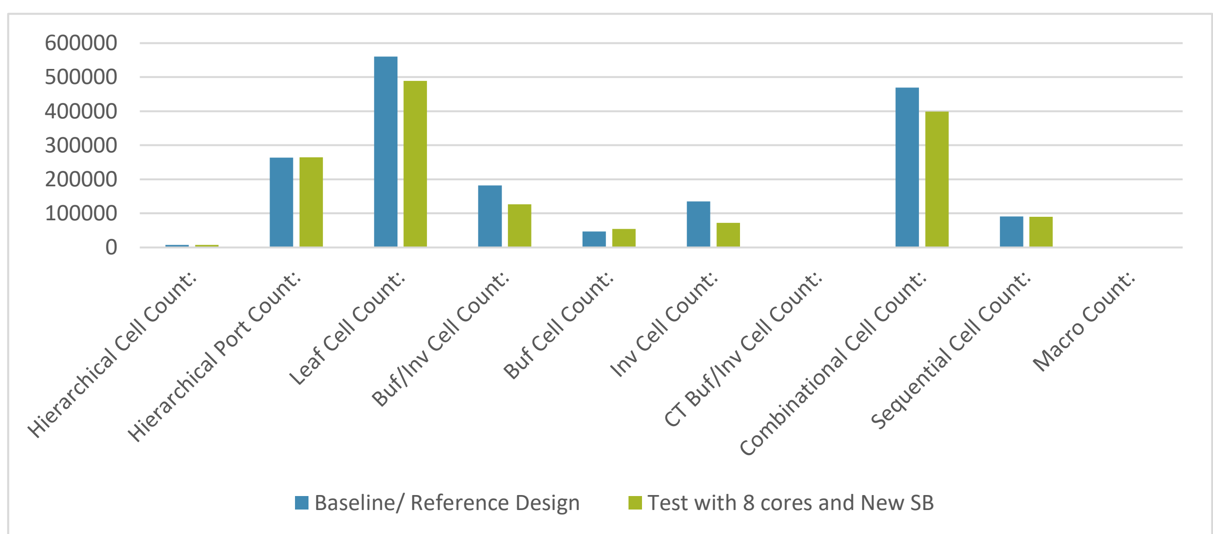

Fig 4.1 Clustered data of cell count of test design with 8 cores and new SB

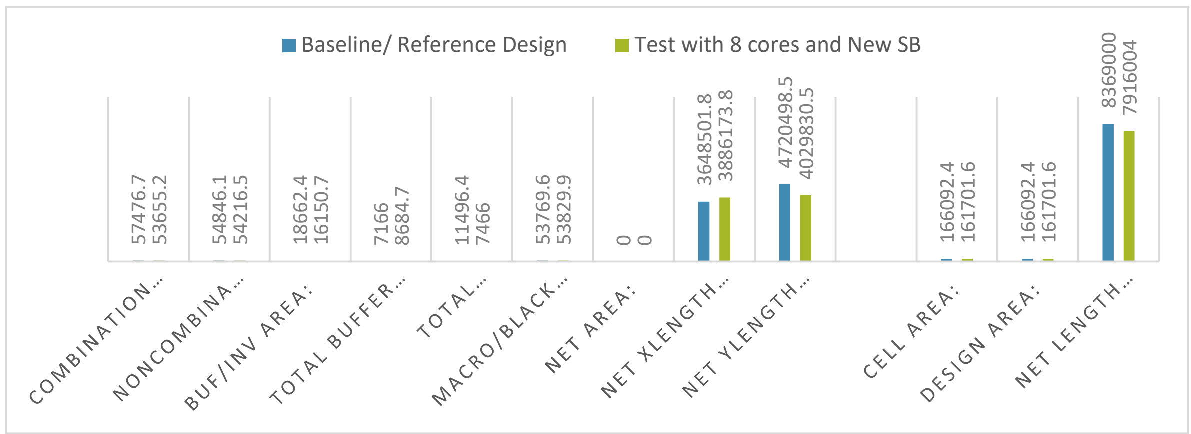

Fig 4.2 Clustered data of area report of test design with 8 cores and new SB

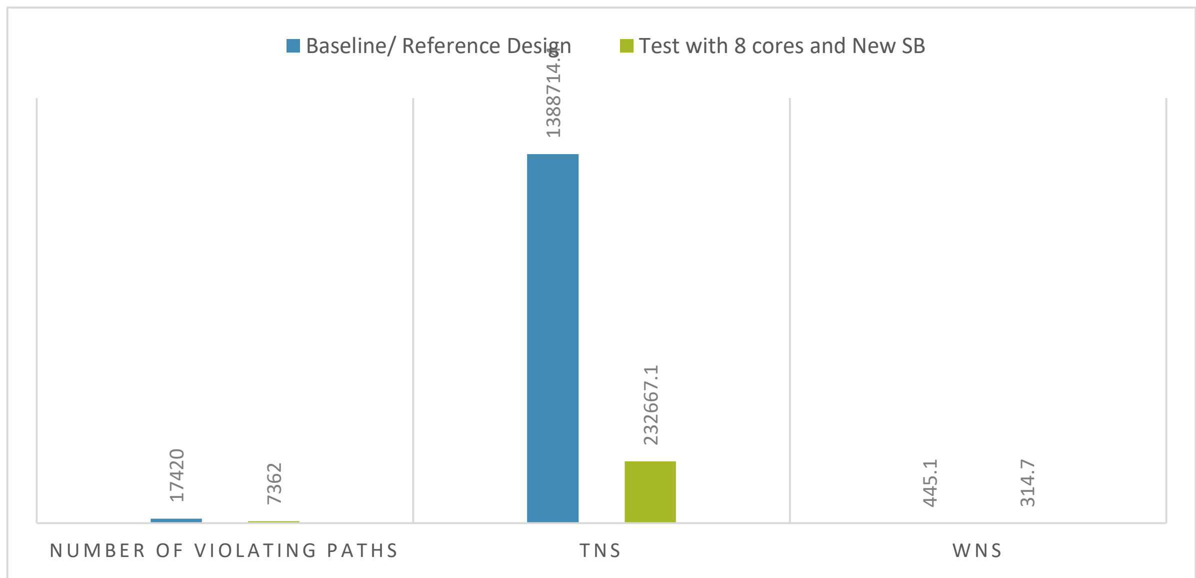

Fig 4.3 Clustered data for Design TNS, WNS and violating paths

Table x can be summarized in the following points

- Even after using 8 cores instead of 4 cores of CPU with new SB, the runtime was not reduced as expected.
- New SB provided better QoR than the baseline
    - The worst negative slack was reduced from 445.1 to 314.7
    - Area was reduced from 57476.7 to 53655.2
    - The net count was decreased from 617501 to 551928

Thus new SB was failed to reduce runtime alone, but it provided other design parameters to be optimized. Figure 4.1, 4.2 and 4.3 represents graphical abstraction of cell count, area information and timing information.

4.3 RUNNING THE COMPILE STEP WITH NEW SB AND PROPOSED APP VARIABLES

As the new SB had opened up the way for a better optimized design, it was still not capable of providing us a better runtime for the test design. Our new methodology involves inclusion of following application variable to the compile step.

App variable	Proposed value
compile_timing_high_effort	FALSE
compile_enable_register_merging	FALSE
spg_enable_via_resistance_support	FALSE
spg_enhanced_timing_model	FALSE
spg_congestion_placement_in_incremental_compile	FALSE
placer_reduce_high_density_regions	FALSE

These app variable with new SB and 6 and 8 CPU cores are able to reduce runtime of compile stage to a new low for the reference design.

Table 4.2 Runtime and QoR report of compile stage with two different Synopsys binaries and 8/6 cores with proposed application variables

Parameters	Baseline/ reference design with old SB	Test design with 8 cores new SB	Test design with 6 cores new SB
Run Time	21:01:48	12:31:11	14:41:17
Cell Count			
Hierarchical Cell Count:	7120	7150	7157
Hierarchical Port Count:	264088	264208	264236
Leaf Cell Count:	560256	473882	472708
Buf/Inv Cell Count:	181906	112092	115367
Buf Cell Count:	46562	50559	51482
Inv Cell Count:	135344	61533	63885
CT Buf/Inv Cell Count:	1399	1361	1360
Combinational Cell Count:	469653	383523	382260
Sequential Cell Count:	90603	89940	90029
Macro Count:	37	419	419
Area			
Combinational Area:	57476.7	50150	50925.4
Noncombinational Area:	54846.1	53559.9	53901.2
Buf/Inv Area:	18662.4	14047.6	14511.9
Total Buffer Area:	7166	7849.5	7923.9

Total Inverter Area:	11496.4	6198	6588
Macro/Black Box Area:	53769.6	53829.9	53829.9
Net Area:	0	0	0
Net XLength :	3648501.8	3281818	3420760.2
Net YLength :	4720498.5	3416733	3570247.8
Cell Area:	166092.4	157539.8	158656.4
Design Area:	166092.4	157539.8	158656.4
Net Length :	8369000	6698551	6991008
Design Rules			
Total Number of Nets:	617501	531795	535280
Nets With Violations:	8959	689	747
Max Trans Violations:	8508	307	348
Max Cap Violations:	241	158	165
Max Fanout Violations:	445	315	315
Compile CPU Statistics			
Resource Sharing:	9437.5	10048.7	9476.7
Logic Optimization:	3809.4	13266.9	9457.1
Mapping Optimization:	250826.8	212445.8	129757.9
Overall Compile Time:	277106.7	252190	161037.9
Overall Compile Wall Clock Time:	70165.2	52321.4	48617.8
Design			
Number of Violating Paths	17420	6113	5608
TNS	1388714.4	212691.4	140668.4
WNS	445.1	267.9	267.7
Design (Hold)			
WNS	3454.4	3456.7	3456.9
TNS	6408507	7047741.5	7040292
Number of Violating Paths	77627	90250	89621

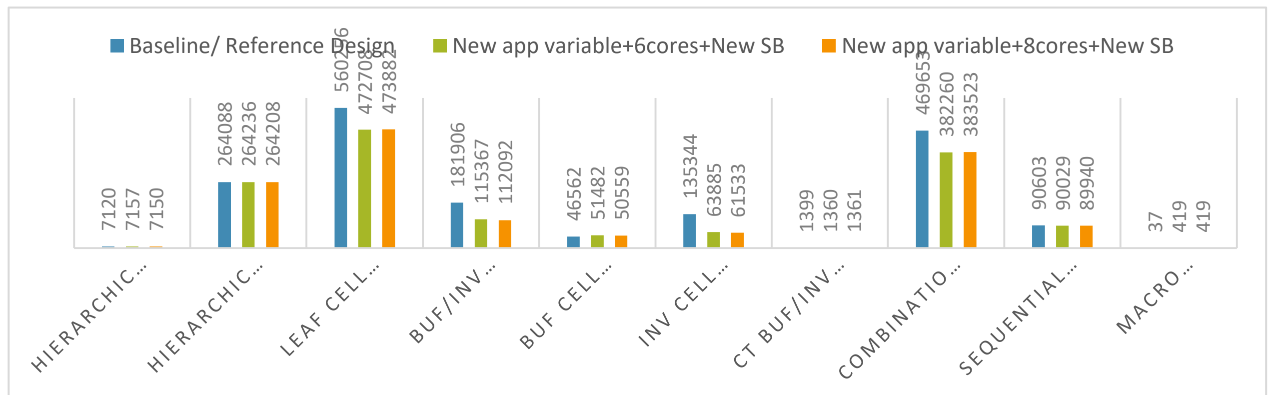

Fig 4.4 Clustered data for cell count with new SB, 6/8 cores and introduced variables

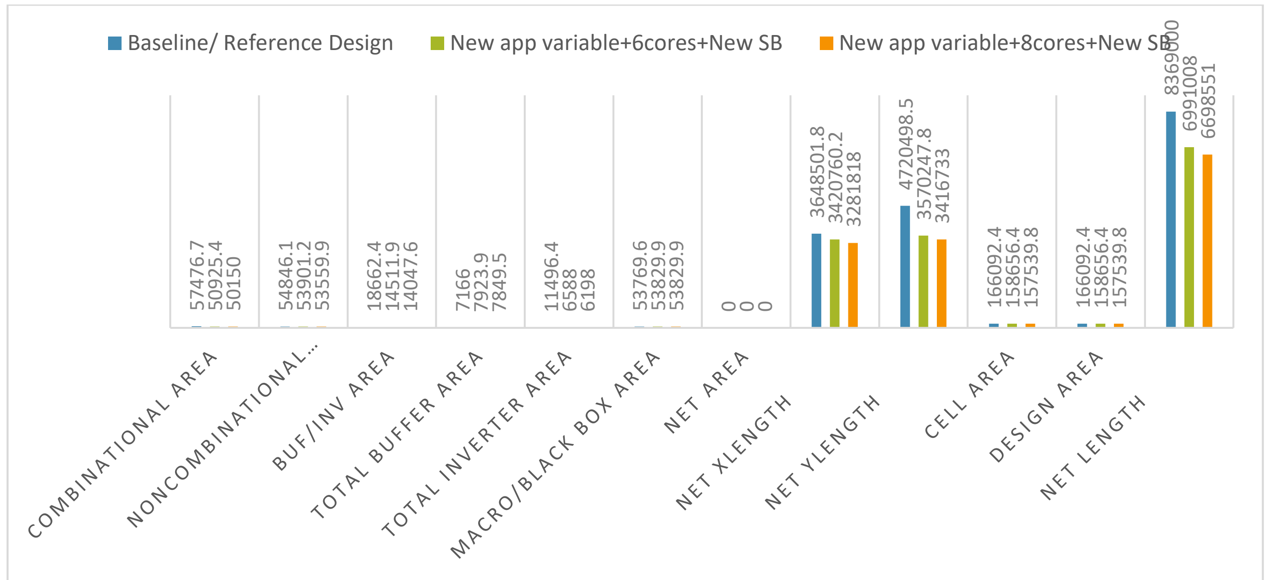

Fig 4.5 Clustered data for area utilization with new SB, 6/8 core and introduced variables

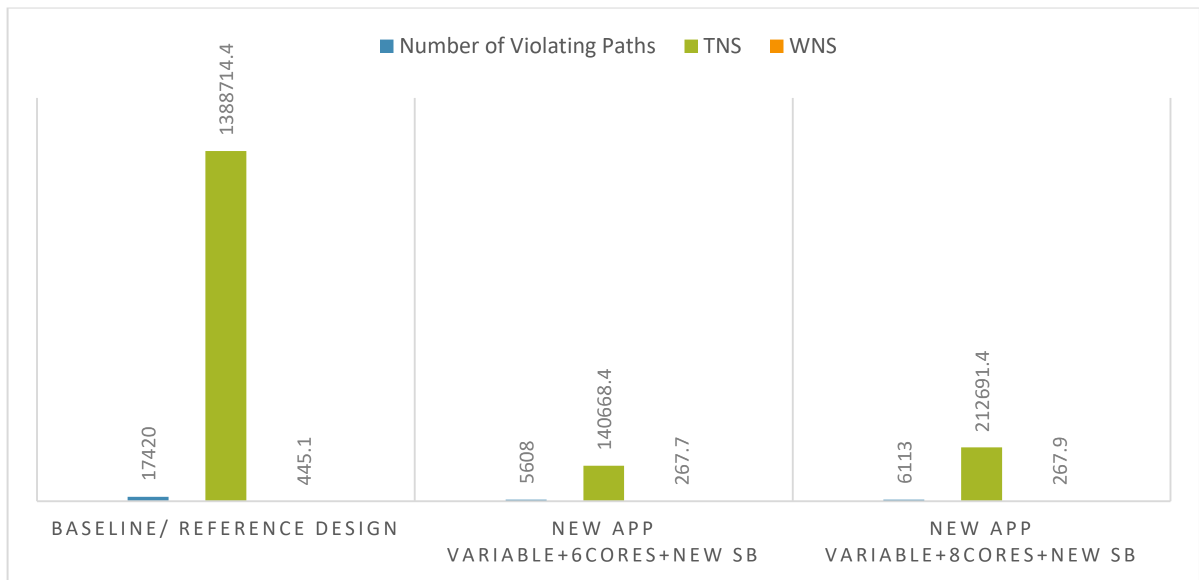

Fig 4.6 Clustered data for TNS, WNS and violating path for 8/6 cores with new SB and proposed application variables

Table can be summarized into following points

- The runtime was reduced from 21:01:48 to 12:31:11 with 8 cores and new SB
- The TNS and WNS are also at their low, 140668.4 and 267.7 respectively for 6 cores and new SB with proposed application variables.
- Though 6 core option is efficient for runtime reduction, 8 core option provides better area optimization.

Thus we carry 8 cores and 6 cores, new SB with modified application variables for the overall runtime reduction for entire synthesis flow. Figure 4.4, 4.5 and 4.6 depicts the graphs for cell count, area utilization and timing information for the test case under proposed methodology respectively.

4.4 ENTIRE SYNTHESIS FLOW WITH NEW APPLICATION VARIABLES, 6/8 CORES WITH NEW SB:

The runtime information along with QoR report is depicted in following tables.

Table 4.3 runtime and QoR data for TNS, WNS and violating path for 8/6 cores with new SB and proposed application variables

PARAMETERS	Baseline/ Reference	6 cores + New SB+ changed app variables	8 cores + New SB+ changed app variables
Overall Time	62:09:18	45:18:23	42:08:51
Cell Count			
Hierarchical Cell Count:	4678	4678	4678
Hierarchical Port Count:	255020	255020	255020
Leaf Cell Count:	526550	460210	458021
Buf/Inv Cell Count:	142077	97315	98046
Buf Cell Count:	45215	39370	39891
Inv Cell Count:	96862	57945	58155
CT Buf/Inv Cell Count:	1412	1360	1359
Combinational Cell Count:	435616	369405	367218
Sequential Cell Count:	90934	90386	90384
Macro Count:	37	419	419
Area			
Combinational Area:	50071.3	43661.7	44245
Noncombinational Area:	53527.9	53325.7	53233.7
Buf/Inv Area:	13945.2	9608.9	9707.6
Total Buffer Area:	5351.1	4199.1	4256.9
Total Inverter Area:	8594.2	5409.8	5450.7
Macro/Black Box Area:	53769.6	53829.9	53829.9
Net Area:	0	0	0
Net XLength:	3556750.8	3237812.8	3181476.2
Net YLength:	4190409.8	3041748.8	3016050.5
Cell Area:	157368.8	150817.3	151308.6
Design Area:	157368.8	150817.3	151308.6
Net Length:	7747160.5	6279561.5	6197527
Design Rules			
Total Number of Nets:	562514	495807	494763

Nets With Violations:	1481	310	281
Max Trans Violations:	765	31	38
Max Cap Violations:	185	169	98
Max Fanout Violations:	581	138	173
Compile CPU Statistics			
Resource Sharing:	0	0	0
Logic Optimization:	0	0	0
Mapping Optimization:	50552.7	54407.4	77781.6
Overall Compile Time:	51891.1	57343	80335
Overall Compile Wall Clock Time:	12750.4	16389.2	15477.9
Design			
WNS:	438.4	268.5	268.7
TNS:	59377.6	43819.5	42560.5
Number of Violating Paths:	1938	3158	3196
Design(Hold)			
WNS:	3463.2	3450.4	3455.1
TNS:	16134612	16257832	16332004
Number of Violating Paths:	126718	129001	128269

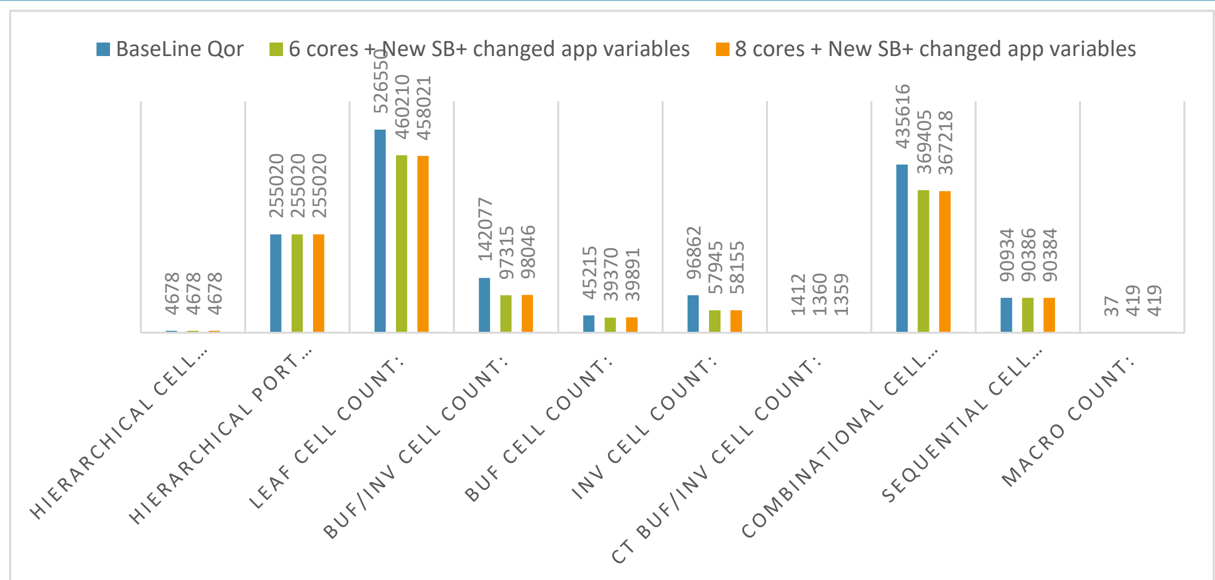

Fig 4.7 Clustered data of cell count for 6/8 cores, new SB and introduced variables for entire synthesis flow

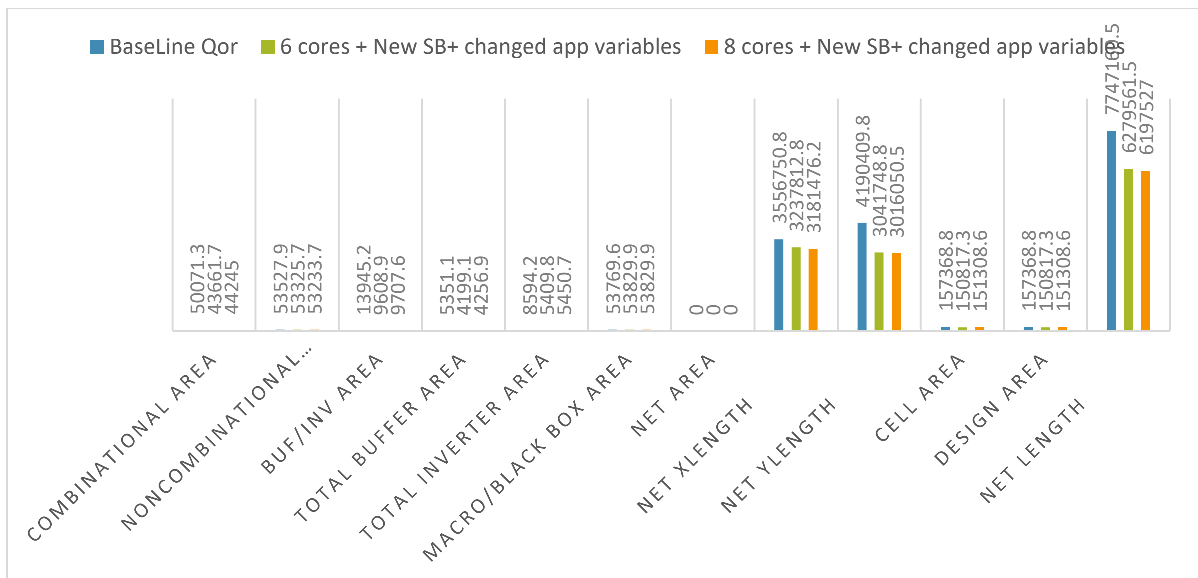

Fig 4.8 Clustered data of area optimization for 6/8 cores, new SB and introduced variable

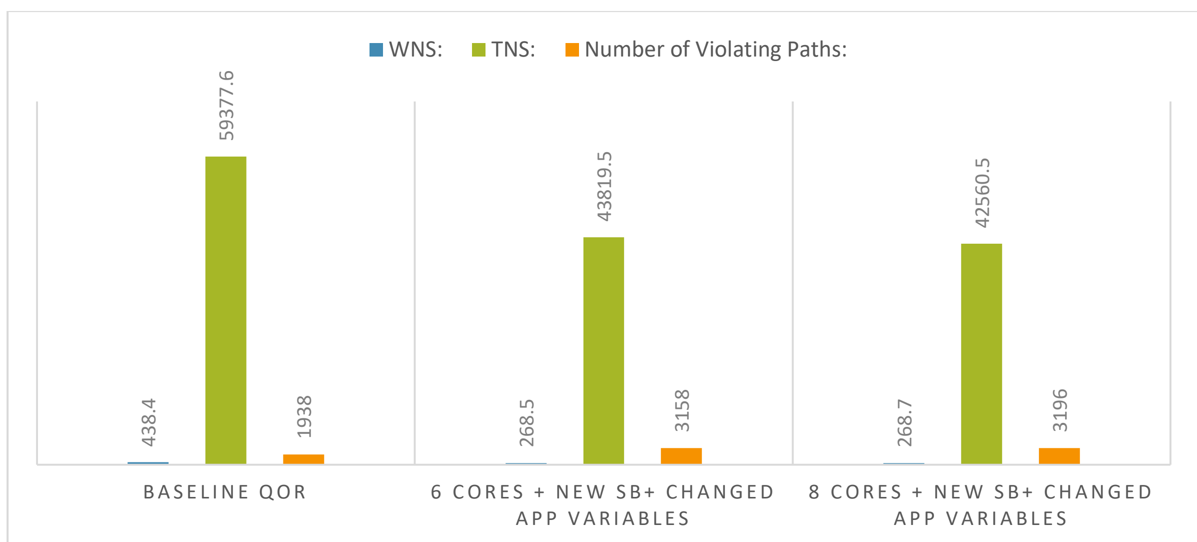

Fig 4.9 Clustered data of TNS, WNS and violating paths for 6/8 cores, new SB and introduced variables

The runtime was reduced with our newly proposed methodology for the test design IP. Following is the Following are the observations after analysis,

- WNS was reduced from 438.4 to 268.5, 268.7 for 6 cores and 8 cores respectively.
- The overall runtime for 8 cores is 42:08:51 as compared to 62:09:18, which is 20 hours less than the baseline.
- There is a tradeoff between number of violating path and WNS.
- Though we were interested in optimizing runtime only, but the proposed methodology provided better QoR w.r.t the baseline designer, which can be used for future designs as a benchmark.

Figure 4.6, 4.7 and 4.7 represents graphs for cell count, area optimization and timing information.

CHAPTER 5
CONCLUSION AND FUTURE SCOPE

This chapter concludes the research work presented as thesis and shed light on the future scope of the presented work.

The exponential increment in the number of transistors on a single chip led us to a very complex design process. Time to market (TTM) for such designs is not only a concern for the tool vendors but also a serious area of concern for the designer which bounds him to use the best design philosophy, and time-efficient modules. In a big organization like Intel, Samsung, Motorola, and Qualcomm have a dedicated engineering team which deals with the development of better-optimized modules to automate the entire design process.

The lack of a decent profiler which is able to identify the most runtime exploiting procedures over the entire physical design flow enabled us to create a dedicated profiler named as runtime abstraction utility. In this thesis work such profiler was proposed and developed with interactive GUI, which eliminates the need of reading into logs for runtime and it is capable of providing information at different abstraction levels (flow/stage/step).

Different designs were parsed through the profiler and analyzed successfully. A design having 62:09:18 (HH:MM:SS) runtime for the combination of synthesis and place & route flow, was selected and used as a test case for further optimization. The profiler provided the list of top 10 runtime hoggers for the subjected design. Out of which a procedure having a runtime of 19:57:58 in the entire flow with 12 invocations was selected for optimization.

With the 8 cores, new SB and setting some specific application variables to false, the procedure depicted a runtime of 12:31:11. Which enabled the designer to opt for a new methodology and the overall runtime got reduced to 42:08:5. This is equal to 32.18% reduction in runtime for the test case.

The future scope of this thesis can be summarized as follows,

1. The current analyzer is static in nature, which can be made dynamic and can run with the flow to provide dynamic information to the designer. Though it sounds good, it will add an overhead to the design flow, thus an utmost care should be taken while developing such profiler as it has to be called every time if a procedure is executed.
2. It is proven that the increase of CPU cores is not enough to reduce the runtime, CAD engineers can look for other solutions to reduce runtime and the current analyzer supports the critical analysis.
3. The developed utility code can also be optimized to get faster analysis.
4. The profiler can also be developed to reduce design time in the front-end of VLSI design.

5. Machine learning can be incorporated to the developed profiler utility to outcast the use of itself (it doesn’t sound good but it is the future) and to provide the least runtime for every flow.

REFERENCES

[1] Golshan K. Physical design essentials: An ASIC design implementation perspective. USA: Springer, 2007.

[2] Bhasker J. and Chadha R. Static timing analysis for nanometer designs. USA: Springer, 2009.

[3] Weste N.H.E. and Harris D.M. CMOS VLSI Design. USA: Pearson Education, 2005.

[4] Chen YK and Kung SY (2008). Trend and Challenges on System-on-a-Chip Designs, *Journal of Signal Processing Systems,* 53(1-2), 217-229.

[5] Kriaa *et al.* (2007). Parallel Programming of Multi-processor SoC: A HW–SW Interface Perspective, *International Journal of Parallel Programming,* 36(1), pp. 68-92.

[6] Sherwani NA. Algorithms for physical design Automation. USA: Kluwer Academic, 1999.

[7] Kuon I and Rose J (2007). Measuring the Gap between FPGAs and ASICs, *IEEE Trans. on CAD*, 26(2), 203-215.

[8] Ressia J *et al.* (2011). Modeling Domain-Specific Profilers, *Journal of Object Technology,* 11(1), 5:1-21.

[9] Graham SL *et al.* gprof: a Call Graph Execution profiler, *Conference on Programming Languages Design and Implementation* [20th: San Diago, CA, USA: 2003], pp. 49-57.

[10] Tanter E *et al.* Partial behavioral reflection: Spatial and temporal selection of reification, *OOPSLA '03 Proceedings of the annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* [18th: Anaheim, California, USA: 2003], pp. 27-46.

[11] Renggli *et al*. Practical Dynamic Grammars for Dynamic Languages, *Workshop on Dynamic Languages and Applications* [4th: Spain:2010], pp. 1-4. Golshan K. Physical design essentials: An ASIC design implementation perspective. USA: Springer, 2007.

DEVELOPMENT OF RUNTIME ABSTRACTION UTILITY AGNOSTIC TO THE FLOW TYPE AND IMPLEMENTATION OF RUNTIME ENHANCED PROCEDURES IN THE MAIN FLOW OF PHYSICAL DESIGN

ORIGINALITY REPORT

18%	%	18%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

	Source	%
1	Andrew B. Kahng. "Introduction", VLSI Physical Design From Graph Partitioning to Timing Closure, 2011 Publication	6%
2	"VLSI Physical Design Automation", Algorithms for VLSI Physical Design Automation, 2002 Publication	4%
3	Alexandre Bergel. "Domain-Specific Profiling", Lecture Notes in Computer Science, 2011 Publication	2%
4	Yen-Kuang Chen. "Trend and Challenge on System-on-a-Chip Designs", Journal of Signal Processing Systems, 11/2008 Publication	1%
5	Algorithms for VLSI Physical Design Automation, 1993. Publication	1%

Scanned by CamScanner