

Digital Phase Locked Loop Interface Design

*Thesis report submitted towards the partial fulfillment of
Requirements for the award of the degree of*

Master of Technology (VLSI Design & CAD)

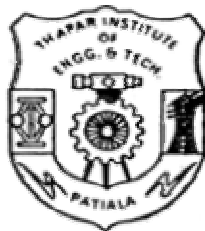
Submitted by

PRAVESH KUMAR SAINI
Roll No. 6040412

Under the Guidance of

Mr. Sanjay Kumar Wadhwa
Design Engineer, Freescale Semi.
Patiala

Dr. Ravinder Agarwal
Asstt. Prof. & Head USIC, TIET-



Department Of Electronics and Communication Engineering
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY,
(Deemed University), PATIALA – 147004, INDIA

June, 2006

DECLARATION

I hereby declare that the seminar report entitled “**Digital Phase Locked Loop Interface Design**” is an authentic record of my own work carried out as requirements for the award of degree of M.Tech. VLSI Design & CAD at Freescale Semiconductor India Pvt. Ltd., Noida, under the guidance of Mr. Sanjay Kumar Wadhwa, Design Engineer, Freescale Semiconductor India Pvt. Ltd., Noida, Dr. Ravinder Agarwal, Asstt. Prof. & Head USIC, TIET-Patiala during January’06 to June’06.

Date: _____

Pravesh Kumar Saini

Roll No. 6040412

Certified that the above statement made by the student is correct to the best of my knowledge and belief.

(Mr. Sanjay Kumar Wadhwa)
Design Engineer, Freescale Semi.

(Dr. Ravinder Agarwal)
Asstt. Prof. & Head USIC, TIET-Patiala

Countersigned by

H.O.D., E.C.E.D, TIET-Patiala

Dean of Academic Affairs, TIET-Patiala

ABSTRACT

The report presents the implementation of Digital Phase Locked Loop (DPLL) Interface Design to configure the DPLL. The ASIC design methodology is used to implement this interfacing module. The flow can improve the design performance and still preserve the efficiency of the standard ASIC design flow. A typical ASIC design flow is composed of two phases. A first phase is called the front end, in which the synthesis from a hardware description language and/or schematic capture is performed, followed by time and area optimization. This is the logic design step, and it is made as much technology independent as possible. The second phase, called the back end, is the physical design of the circuit; it consists of the placement of all of the components on the core, followed by the routing of the power and signal wires. Obviously, this phase is strongly technology dependent. The hand-crafted fully-custom application-specific integrated circuit (ASIC) design methodology usually delivers chips with lower power consumption, 5 to 8 times higher speed and more compacted chip layouts than conventional standard ASIC design method. On the other hand, automatic standard ASIC design flow enables more efficient design process. Based on pre-characterized and silicon verified standard cells, the emerging of design automation technology enables standard ASIC have many advantages over full custom designs, such as fast-turn-around time, less design resource requirement, especially human resources, easy for technology migration, high productivity, and thus low price.

ACKNOWLEDGMENTS

I express my sincere gratitude to my supervisors Mr. Sanjay Kumar Wadhwa, Ms. Deeya Muhury, Mr. Amit Srivastva as well as all other members of our Analog Group Design team at Freescale Semiconductors India ltd. and especially Dr. Ravinder Agarwal and Mrs. Alpana Agarwal under whose inspiration, encouragement and guidance I had the opportunity to do the work for my thesis. I am also thankful to Mr. Pradeep Mishra and Mr. Kulbhushan Misri to give me opportunity to work with them. They all allowed me to work on my thesis in complete freedom while strongly supporting my academic endeavors, no matter where they took me. I would like to thank them for introducing me to the problem and providing invaluable advice throughout the course of my thesis.

It was a pleasure working at India Design Center (IDC), Freescale Semiconductors India Pvt. Ltd., Noida and T.I.E.T., Patiala. This is mostly due to the wonderful people who have sojourned there over the past years.

This work could never have been accomplished without the inspiration, guidance and support of my innumerable friends, colleagues and my family members.

Pravesh Kumar Saini
Regn. No. 6040412

LIST OF FIGURES

Figure Number	Page Number
Fig 1.1 ASIC Design Flow.....	2
Fig 1.2 Test Bench Model.....	5
Fig. 1.3 Design Compiler Input and Output.....	6
Fig. 2.1 Standard negative-feedback control system models	30
Fig. 2.2 Basic phase-locked-loop model.....	31
Fig. 2.3 VCO transfer function.....	33
Fig 2.4 Block diagram of a PLL implemented with a TCXO reference.....	35
Fig. 3.1 DPLL Interface Architecture.....	40
Fig. 4.1 Pre layout Simulation .fsdb file's waveform in Debussy.....	49
Fig. 4.2 Post layout Simulation .fsdb file's waveform in Debussy.....	52

LIST OF TABLES

Table	Page Number
Table 3.1 Shift Register bit allocation	41
Table 3.2 Programming Model for data_in.....	44
Table 3.3 Programming Model for data_out.....	45
Table 3.4 Input Constraints.....	47

ABBREVIATIONS

VCO	Voltage Controlled oscillator
PD	Phase Detector
CP	Charge Pump
PLL	Phase Locked Loop
LPLL	Linear Phase Locked Loop
ADPLL	All Digital PLL
SPLL	Software PLL
DSP	Digital Signal Processing
CLG	Closed Loop Gain
TCXO	Temperature Compensated Crystal
Oscillator	
RTL	Register Transfer Logic
ASIC	Application Specific Integrated Circuit
DSM	Deep Sub Micron
DID	DPLL Interface Design
DPLL	Digital Phase Locked Loop
MSB	Most Significant Bit
LSB	Least Significant Bit
HDL	Hardware Description Language
DFT	Design for Testability
BIST	Built in self Test
DC	Design Compiler
PC	Physical Compiler
STA	Static Timing Analysis
CT	Clock Tree
LVS	Layout versus Schematic
DRC	Design Rule Check
VHDL	Very High Speed IC HDL
IC	Integrated Circuit
ECO	Engineering Change Order

PT	Prime Time
SDF	Standard Delay Format
PDEF	Parasitic Delay Extraction Format
IPO	In Place Optimization
VCS	Verilog Code Simulator
DEF	Design Exchange Format
ICFB	Integrated Circuit Front end & Backend
RCXT	RC Extractor

TABLE OF CONTENTS

DECLARATION.....	i
ABSTRACT.....	ii
ACKNOWLEDGMENTS.....	iii
LIST OF FIGURES.....	iv
LIST OF TABLES.....	v
ABBREVIATIONS.....	vi
CHAPTER 1 ASIC DESIGN FLOW.....	1
1.1 Introduction.....	1
1.2 Specification and RTL Coding.....	3
1.3 Dynamic Simulation.....	4
1.4 Constraints, Synthesis and Scan Insertion.....	5
1.5 Formal Verification.....	7
1.6 Static Timing Analysis using PrimeTime.....	9
1.7 Placement, Routing and Verification.....	10
1.8 Physical Synthesis.....	12
1.9 Initial Setup.....	13
1.10 Traditional flow.....	14
1.10.1 Pre-Layout Steps.....	15
1.10.1.1 Synthesis.....	15
1.10.1.2 SDF Generation.....	18
1.10.2 Floorplanning and Routing.....	19
1.10.3 Post-Layout Steps.....	23
1.10.3.1 Post-Layout Static Timing Analysis using PrimeTime.....	24
1.10.3.2 Post-Layout Optimization.....	25
1.10.4 Hold-Time Fixes.....	26
1.11 Engineering Change Order.....	28
1.12 Objective of the Thesis.....	29
1.13 Organization of the Thesis.....	29

CHAPTER 2 PLL ARCHITECTURE	30
2.1 Introduction.....	30
2.2 Frequency Synthesis.....	34
2.3 Digital PLL (DPLL) Synthesis.....	34
2.4 Integer-N PLL.....	34
2.5 Fractional-N PLL.....	36
CHAPTER 3 DPLL INTERFACE DESIGN	39
3.1 Introduction to Architecture	39
3.2 Block Diagram.....	40
3.3 Programming Model.....	41
3.4 Shift in to Registers.....	42
3.5 Shift out of Registers.....	43
3.6 Writing into the Shift Registers.....	46
3.7 Reading the Shift Registers.....	46
3.6 Input Constraints.....	47
CHAPTER 4 RESULTS AND DISCUSSION	48
4.1 Introduction.....	48
4.2 Prelayout Waveforms.....	48
4.3 Timing Check.....	49
4.4 Read Parasitic Report.....	50
4.5 Report Physical Optimization.....	50
4.6 Post Layout Waveforms and Results.....	52
CHAPTER 5 CONCLUSION AND FUTURE SCOPE	54
REFERENCES	57

CHAPTER-1 ASIC DESIGN FLOW

1.1 Introduction

The complexity of today's largest IC designs is over ten million gates. Looking ahead it is prudent to prepare a methodology for the 50M gate ASIC [1]. Three forces are at work, which makes designing chips at the edge of the capability of the fabrication technology increasingly difficult. First, the size of today's 10M gate designs taxes even the largest and fastest computers. Second, the deep sub micron (DSM) effects are breaking existing design flows. The third force is the shrinking market window. The capacity and complexity problems impact the productivity causing a product to miss its market window. This leads to a requirement for a fast, high capacity and scalable technology that provides early estimates of post-layout performance and identifies many issues that would typically have been found only after detailed place & route in a conventional flow. This saves numerous time-consuming iterations and enhances the productivity to enable fast time-to-market. An additional advantage of such a technology is that engineers can explore design architectures and implementation alternatives and have a high level of confidence that performance goals can be achieved.

Alok Mehrotra, Lukas van Ginneken and Yatin Trivedi in [2] and M. Cote and P. Hurat. in [3] proposed automatic design flows focusing on post-layout cell resizing. Both approaches showed performance improvements on the ASIC designs. The drawback of these approaches lies in their capabilities of handling large-scale designs. And the design time had been significantly sacrificed.

The traditional ASIC design flow contains the steps outlined below. Fig. 1.1 illustrates the flow chart relating to the design flow described below.

1. Architectural and electrical specification.
2. RTL coding in HDL.

3. DFT memory BIST insertion, for designs containing memory elements.
4. Exhaustive dynamic simulation of the design, in order to verify the functionality of the design.
5. Design environment setting. This includes the technology library to be used, along with other environmental attributes.
6. Constraining and synthesizing the design with scan insertion using Design Compiler.
7. Block level static timing analysis, using Design Compiler's built-in static timing analysis engine.
8. Formal verification of the design. RTL compared against the synthesized netlist, using Formality.
9. Pre-layout static timing analysis on the full design through PrimeTime.
10. Forward annotation of timing constraints to the layout tool.

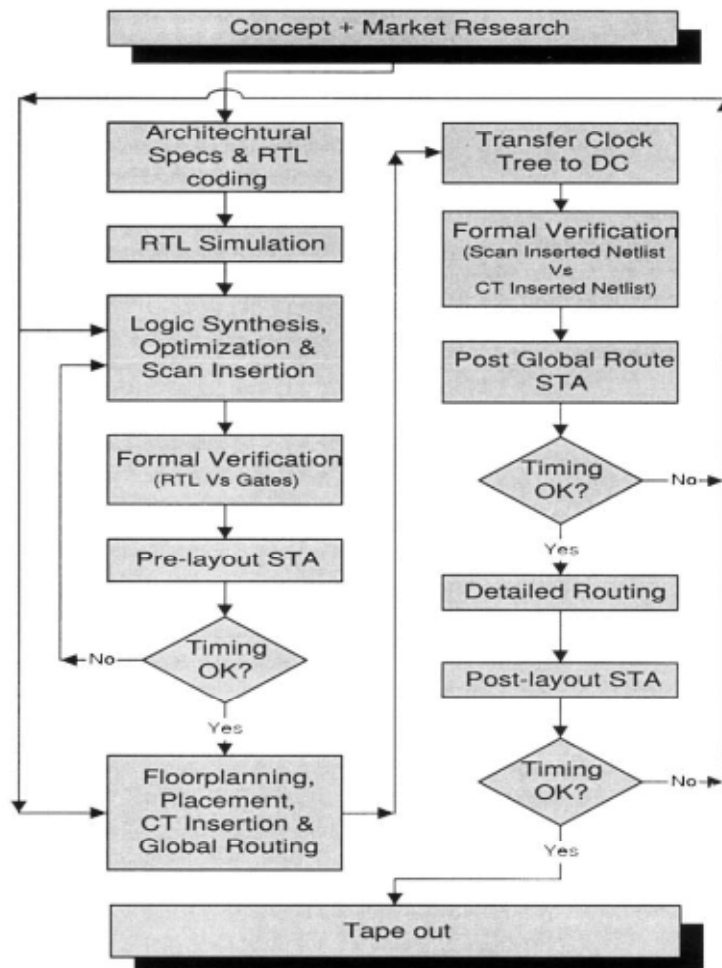


Fig 1.1 ASIC Design Flow

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

Fig. 1-1, graphically illustrates the typical ASIC design flow discussed above. The acronyms STA and CT represent static timing analysis and clock tree respectively. DC represents Design Compiler.

1.2 Specifications and RTL Coding

Chip design commences with the conception of an idea dictated by the market. These ideas are then translated into architectural and electrical specifications. The architectural specifications define the functionality and partitioning of the chip into several manageable blocks, while the electrical specifications define the relationship between the blocks in terms of timing

information.

The next phase involves the implementation of these specifications. In the past this was achieved by manually drawing the schematics, utilizing the components found in a cell library. This process was time consuming and was impractical for design reuse. To overcome this problem, hardware description languages (HDL) were developed. As the name suggests, the functionality of the design is coded using the HDL. There are two main HDLs in use today, Verilog and VHDL. Both languages perform the same function, each having their own advantages and disadvantages.

There are three levels of abstraction that may be used to represent the design; Behavioral, RTL (Register Transfer Level) and Structural. The Behavioral level code is at a higher level of abstraction. It is used primarily for translating the architectural specification, to a code that can be simulated. Behavioral coding is initially performed to explore the authenticity and feasibility of the chosen implementation for the design. Conversely, the RTL coding actually describes and infers the structural components and their connections. This type of coding is used to describe the functionality of the design and is synthesizable to form a structural netlist. This netlist comprises of the components from a target library and their respective connections; very similar to the schematic based approach.

The design is coded using the RTL style, in either Verilog or VHDL, or both. It can also be partitioned if necessary, into a number of smaller blocks to form a hierarchy, with a top-level block connecting all lower level blocks.

1.3 Dynamic Simulation

The next step is to check the functionality of the design by simulating the RTL code. All currently available simulators are capable of simulating the behavior level as well as RTL level coding styles. In addition, they are also used to simulate the mapped gate-level design.

Fig. 1.2 illustrates a partitioned design surrounded by a test bench ready for simulation. This test bench is normally written in behavior HDL while the actual design is coded in RTL.

Usually the simulators are language dependent (either Verilog or VHDL), although there are a few simulators in the market, capable of simulating a mixed HDL design.

The purpose of the test bench is to provide necessary stimuli to the design. It is important to note that the coverage of the design is totally dependent on the number of tests performed and the quality of the test bench.

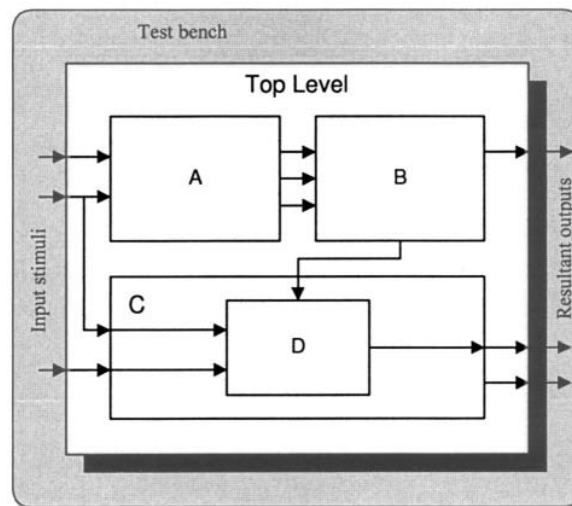


Fig 1.2 Test Bench Model

This is the reason why a sound test bench is extremely critical to the design. During the simulation of the RTL, the component (or gate) timing is not considered. Therefore, to minimize the difference between the RTL simulation and the synthesized gate-level simulation at a later stage, the delays are usually coded within the RTL source, usually for sequential elements.

1.4 Constraints, Synthesis and Scan Insertion

For a long time, the HDLs were used for logic verification. Designers would manually

translate the HDL into schematics and draw the interconnections between the components to produce a gate-level netlist. With the advent of synthesis tools, this manual task has been rendered obsolete. The tool has taken over and performs the task of reducing the RTL to the gate-level netlist. This process is termed as synthesis.

Synopsys's Design Compiler (from now on termed as DC) is the de-facto standard and by far the most popular synthesis tool in the ASIC industry today.

Logic synthesis is the process that takes a logic description of a circuit in an abstract form (such as Boolean equations) and a technology library and obtains a realization of the circuit in terms of gates from the library. Optimization techniques can be classified as technology independent optimization [5-6] and technology dependent optimization [7-10]. A key component of any synthesis architecture is an incremental static timing engine [11-12], also called a timer. The function synthesizing a design is an iterative process and begins with defining timing constraints for each block of the design. These timing constraints define the relationship of each signal with respect to the clock input for a particular block. In addition to the constraints, a file defining the synthesis environment is also needed. The environment file specifies the technology cell libraries and other relevant information that DC uses during synthesis.

DC reads the RTL code of the design and using the timing constraints, synthesizes the code to structural level, thereby producing a mapped gate-level netlist. This concept is shown in Fig. 1-3.

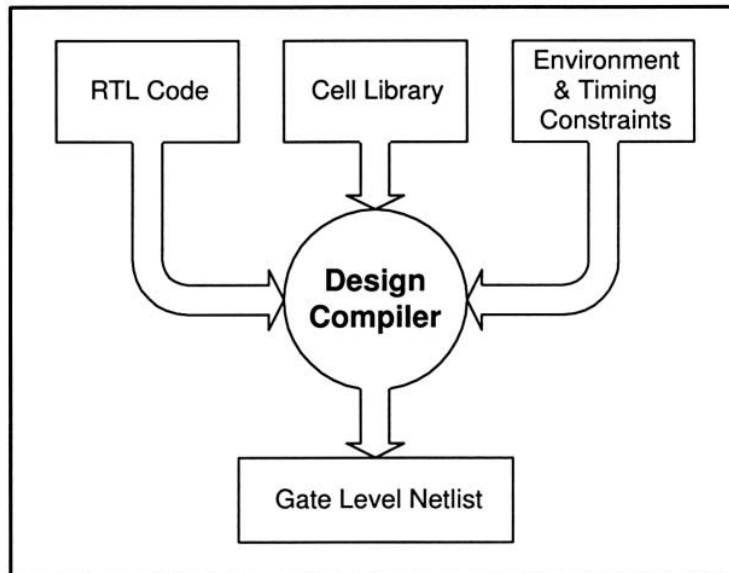


Fig. 1.3 Design Compiler Input and Output

Usually, for small blocks of a design, DC's internal static timing analysis is used for reporting the timing information of the synthesized design. DC tries to optimize the design to meet the specified timing constraints. Further steps may be necessary if timing requirements are not met.

Most designs today, incorporate design-for-test (DFT) logic to test their functionality, after the chip is fabricated. The DFT consists of logic and memory BIST (built-in-self-test), scan logic and Boundary Scan logic etc. Boundary Scan Test can be integrated into an existing ASIC design flow [4].

The logic and memory BIST comprises of synthesizable RTL that is based upon controller logic and is incorporated in the design before synthesis. There are tools available in the market that may be used to generate the BIST controller and surrounding logic. Unfortunately, Synopsys does not provide this capability.

The scan insertion may be performed using the test ready compile feature of DC. This procedure maps the RTL directly to scan-flops, before linking them in a scan-chain. An advantage of using this feature is its ability to enable DC to take the scan-flop timing

into account while synthesizing. This technique is important since the scan-flops generally have different delays associated with them as compared to their non-scan equivalent flops (or normal flops).

JTAG or boundary scan is primarily used for testing the board connections, without unplugging the chip from the board. The JTAG controller and surrounding logic may also be generated directly by DC.

1.5 Formal Verification

The concept of formal verification is fairly new to the ASIC design community. Formal verification techniques perform validation of a design using mathematical methods without the need for technological considerations, such as timing and physical effects. They check for logical functions of a design by comparing it against the reference design.

A number of EDA tool vendors have developed the formal verification tools. However, recently, Synopsys also introduced to the market its own formal verification tool called Formality.

The main difference between formal methods and dynamic simulation is that former technique verifies the design by proving that the structure and functionality of two designs are logically equivalent. Dynamic simulation methods can only probe certain paths of the design that are sensitized, thus may not catch a problem present elsewhere. In addition, formal methods consume negligible amount of time as compared to dynamic simulation.

The purpose of the formal verification in the design flow is to validate the RTL against RTL, gate-level netlist against the RTL code, or the comparison between gate-level-to-gate-level netlists.

The RTL to RTL verification is used to validate the new RTL against the old functionally correct RTL. This is usually performed for designs that are subject to frequent

changes in order to accommodate additional features. When these features are added to the source RTL, there is always a risk of breaking the old functionally correct feature. To prevent this, formal verification may be performed between the old RTL and the new RTL to check the validity of the old functionality.

The RTL to gate-level verification is used to ascertain that the logic has been synthesized accurately by DC. Since the RTL is dynamically simulated to be functionally correct, the formal verification of the design between the RTL and the scan inserted gate level netlist assures us that the gate-level also has the same functionality. In this instance if dynamic simulation method were used to verify the gate-level, it would have taken a long time (days and weeks, depending on the size of the design) to verify the design. In comparison, the formal method would take a few hours to perform a similar verification.

The last part involves verifying the gate-level netlist against the gate-level netlist. This too is a significant step for the verification process, since it is mainly used to verify what has gone into the layout versus what has come out of the layout. What comes out of the layout is obviously the clock tree inserted netlist (flat or hierarchical). This means that the original netlist that goes into the layout tool is modified. The formal technique is used to verify the logic equivalency of the modified netlist against the original netlist.

1.6 Static Timing Analysis using PrimeTime

As previously mentioned, the block level static timing analysis is done using DC. Although, the chip-level static timing can be performed using the above approach, it is recommended that PrimeTime, be used instead. PrimeTime is the Synopsys stand-alone sign-off quality static timing analysis tool that is capable of performing extremely fast static timing analysis on full chip-level designs. It provides a Tcl interface that provides a powerful environment for analysis and debugging of designs.

The static timing analysis, to some extent, is the most important step in the whole ASIC design process. This analysis allows the user to exhaustively analyze all critical paths of the

design and express it in an orderly report. Furthermore, the report can also contain other debugging information like the fanout or capacitive loading of each net.

The static timing is performed both for the pre and post-layout gate-level netlist. In the pre-layout mode, PrimeTime uses the wire load models specified in the library to estimate the net delays. During this, the same timing constraints that were fed to DC previously are also fed to PrimeTime, specifying the relationship between the primary I/O signals and the clock. If the timing for all critical paths is acceptable, then a constraints file may be written out from PrimeTime or DC for the purpose of forward annotation to the layout tool. This constraint file in SDF format specifies the timing between each group of logic that the layout tool uses, in order to perform the timing driven placement of cells.

In the post-layout mode, the actual extracted delays are back annotated to PrimeTime to provide realistic delay calculation. These delays consist of the net capacitances and interconnect RC delays.

Similar to synthesis, static timing analysis is also an iterative process. It is closely linked with the placement and routing of the chip. This operation is usually performed a number of times until the timing requirements are satisfied.

1.7 Placements, Routing and Verification

As the name suggest, the layout tool performs the placement and routing. There are a number of methods in which this step could be performed. However, only issues related to synthesis are discussed in this section.

The quality of floor plan and placement is more critical than the actual routing. Optimal cell placement location, not only speeds up the final routing, but also produces superior results in terms of timing and reduced congestion. As explained previously, the constraint file is used to perform timing driven placement. The timing drive-in placement method forces the layout tool to place the cells according to the

criticality of the timing between the cells.

Traditional “rough” placement techniques can roughly classified into three categories based on the approach to solve the problem: graph partition- being and clustering [13-19], simulated annealing [20- 22], and mathematical programming.

After the placement of cells, the clock tree is inserted in the design by the layout tool. The clock tree insertion is optional and depends solely on the design and user’s preference. Users may opt to use more traditional methods of routing the clock network, for example, using fishbone/spine structure for the clocks in order to reduce the total delay and skew of the clock [23-24]. As technologies shrink, the spine approach is getting more difficult to implement due to the increase in resistance (thus, RC delays) of the interconnect wires.

At this stage an additional step is necessary to complete the clock tree insertion. As mentioned above, the layout tool inserted the clock tree in the design after the placement of cells. Therefore, the original netlist that was generated from DC (and fed to the layout tool), lacks the clock tree information (essentially the whole clock tree network, including buffers and nets). Therefore, the clock tree must be re-inserted in the original netlist and formally verified. Some layout tools provide direct interface to DC to perform this step. For the sake of simplicity, let’s assume that the clock tree insertion to the original netlist has been performed.

The layout tool generally performs routing in two phases – global routing and detailed routing. After placement, the design is globally routed to determine the quality of placement, and to provide estimated delays approximating the real delay values of the post-routed (after detailed routing) design. If the cell placement is not optimal, the global routing will take a longer time to complete, as compared to placing the cells. Bad placement also affects the overall timing of the design. Therefore, to minimize the number of synthesis-layout iterations and improve placement quality, the timing information is extracted from the layout, after the global routing phase. Although, these delay numbers are not as accurate as the numbers extracted after detailed routing, they do provide a fair idea of the post-routed

timing. The estimated delays are back annotated to PrimeTime for analysis, and only when the timing is considered satisfactory, the remaining process is allowed to proceed.

Detailed routing is the final step that is performed by the layout tool. After detailed route is complete, the real timing delays of the chip are extracted, and plugged into PrimeTime for analysis.

These steps are iterative and depend on the timing margins of the design. If the design fails timing requirements, post-layout optimization is performed on the design before undergoing another iteration of layout. If the design passes static timing analysis, it is ready to undergo LVS (layout versus schematic) and DRC (design rule checking) before tape-out.

It must be noted that all steps discussed above can also be applied for hierarchical place and route. In other words, one can repeat these steps for each sub-block of the design before placing the sub-blocks together in the final layout and routing between the sub-blocks.

1.8 Physical Synthesis

Traditionally synthesis methods are based on using the wire-load models. The basic nature of the wire-load models is such that they are fanout based. In other words, the delay computation of cells is performed based on the number of fanouts a cell drives. While this method was ideal for larger geometries ($>0.35\mu\text{m}$), it is not suitable for smaller geometries. The resistance of wires is dominating the cell delays causing the fanout based delay computation to be unreliable and totally unpredictable.

The concept of physical synthesis was recently introduced by Synopsys in the form of Physical Compiler (henceforth, called PhyC) as a solution to the above problem. The previous capability of DC is retained and the PhyC enhancements have been

added on top of DC thus making PhyC a superset of DC.

PhyC does not use the wire-load models; instead the delay computation is based on the placement rather than fanout. In other words, the synthesis and optimization is based on the placement of cells. By incorporating scan chain re-ordering capability in the current release of PhyC (2000.11), it indeed makes it an extremely powerful and useful tool.

Fig. 2.1 illustrates this approach in a very generic form. PhyC can be used in two modes: RTL-to-placed-gates (rtl2pg) or Gates-to-placed-gates (g2pg). For the former mode, the input to PhyC is the RTL, the floorplan information, along with the necessary setup to include logical and physical libraries. The output produced by PhyC is a structural netlist and the placed gates information in PDEF3.0 format. The second mode of g2pg is provided that can be used for optimizing an existing gate level netlist based on the floorplan information. In this case, instead of the RTL, the input to PhyC is the gate level netlist. The rest of the setup and I/O files remain the same.

One important point to note is that in the current release of PhyC (version 2000.11), does not have the capability of synthesizing clock trees in the design. Synopsys has recently announced the availability of the Clock Tree Compiler tool that is an add-on to PhyC. Users who do not have access to this tool have no option but to use their layout tool to insert clock tree in the design database.

The best way to start this topic is to go through the whole process as an example design. A tap controller design, coded in Verilog HDL and consisting of one level of hierarchy as shown below is chosen for this purpose:

```
tap_controller.v tap_bypass.v tap_instruction.v tap_state.v.
```

The top level of the design is called tap controller which instantiate three modules called tap_bypass, tap instruction and tap_state. This design contains a single 30 MHz clock called “tck” and a reset called “trst”. Timing specifications for this

design dictate that the setup-time needed for all input signals with respect to “tck” is 10ns, while the hold-time is 0ns. Furthermore, all output signals must be delayed by 10ns with respect to the clock.

The process technology targeted for this design is 0.25 micron. In order to achieve greater accuracy due of variance in process, two Synopsys standard cell technology libraries, characterized for the worst-case and the best-case process parameters are used. The libraries are called ex25_worst.db and ex25_best.db, with a corresponding symbol library containing schematic representations, called ex25.sdb. The name of the operating conditions defined in the ex25_worst.db library is WORST, while the name of the operating conditions in the ex25_best.db library is BEST.

It is assumed that the functionality of the design has been verified by dynamically simulating it at the RTL level.

1.9 Initial Setup

The next step is to synthesize the design, i.e., map the design to the gates belonging to the specified technology library. Before the design begins synthesis, several setup files must be created as follows:

- a) .synopsys_dc.setup files for DC & PhyC.
- b) .synopsys_pt.setup files for PT.

The first file is the setup file for DC & PhyC and is used for logic synthesis as well as physical synthesis, while the second file is associated with PT and defines the required setup to be used for static timing analysis.

Create both of these files with the following contents, assuming that the libraries are kept in the directory –

/usr/golden/library/std_cells/DC & PhyC .synopsys_dc.setup file

```
set search_path      [list /usr/golden/library/std_cells]
set target_library   [list ex25_worst.db]
set link_library     [list {*} ex25_worst.db ex25_best.db]
set symbol_library   [list ex25.sdb]
set physical_library [list ex25_worst.pdb]
define_name_rules BORG -allowed {A-Za-z0-9_} -first_restricted "_" -last_restricted "_"
-max_length 30 -map {{"*cell*"}, {"mycell"}}, {"*-return"}, {"myreturn"}}
set bus_naming_style %s[%d]
set verilogout_no_tri true
set verilogout_show_unconnected_pins true
set test_default_scan_style multiplexed_flip_flop
```

PT .synopsys_pt.setup file

```
set search_path [list ./usr/golden/library/std_cells]
set link_library [list {*} ex25_worst.db ex25_best.db]
```

1.10 Traditional Flow

The following steps outline the traditional flow. Here DC is used for logic synthesis while the layout tool handles the rest of the back-end that includes placement and routing.

1.10.1 Pre-Layout Steps

The following sub-sections illustrate the steps involved during the pre-layout phase. This includes one-pass logic synthesis with scan insertion, static timing analysis, SDF generation to perform functional gate-level simulation, and finally formal verification between the source RTL and synthesized netlist.

1.10.1.1 Synthesis

The pre-layout logic synthesis involves optimizing the design for maximum setup-time, utilizing the statistical wire-load models and the worst-case operating conditions from the ex25_worst.db technology library. In order to maximize the setup-time, you may constrain the design by defining clock uncertainty for the setup-time. In general, a 10% over-constrain is usually sufficient, in order to minimize the synthesis-layout iterations.

After initial synthesis if gross hold-time violations are detected, they should be fixed at the pre-layout level. This also helps in reducing the synthesis-layout iterations. However, it is preferable to fix minor hold-time violations after the layout, with real delays back annotated.

Here it is assumed that minor hold-time violations exist, therefore these violations will be fixed during the post-layout optimization. Fixing hold-time violations involves back annotation of the extracted delays from the layout to DC. In addition, hold-time fixes require usage of the best-case operating conditions from the ex25_best.db library.

Generic synthesis script for sub-modules

```
set active_design tap_bypass
analyze -format verilog $active_design.v elaborate $active_design
current_design $active_design link
uniquify
set_wire_load_model -name SMALL
set_wire_load_mode top
set_operating_conditions WORST
create_clock -period 33 -waveform [list 0 16.5] tck
set_clock_latency 2.0 [get_clocks tck]
set_clock_uncertainty -setup 3.0 [get_clocks tck]
set_clock_transition 0.1 [get_clocks tck] set_dont_touch_network [list tck trst]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
```

```

set_drive 0 [list tck trst]
set_input_delay 20.0 -clock tck -max [all_inputs]
set_output_delay 10.0 -clock tck -max [all_outputs]
set_max_area 0
set_fix_multiple_port_nets -buffer_constants -all_compile -scan
check_test
remove_unconnected_ports [find -hierarchy cell {"*"}]
change_names -h -rules BORG
set_dont_touch current_design
write -hierarchy -output $active_design.db
write -format verilog -hierarchy -output $active_design sv

```

The above script contains a user-defined variable called `active_design` that defines the name of the module to be synthesized. This variable is used throughout the script, thus making the rest of the script generic. By re-defining the value of `active_design` to other sub-modules (`tap_instruction` and `tap_state`), the same script may be used to synthesize the sub-modules. Users can apply the same concept to clock names, clock periods, etc. in order to parameterize the scripts.

Let's assume that you have successfully synthesized three sub-blocks, namely `tap_bypass`, `tap_instruction` and `tap_state`. The one-pass synthesis was done with no scan chain stitching. Only the flops were directly mapped to scan flops. One can apply the same synthesis script to synthesize the top level; with the exception that the design have to include the mapped "db" files for the sub-blocks, before reading the `tap_controller.v` file. In addition, this time the design have to perform scan insertion also in order to stitch the scan chains. Also, the wire-load mode may need to be changed to enclosed for proper modeling of the interconnect wires. Since the sub-modules contain the `dont_touch` attribute, the top-level synthesis will not optimize across boundaries, and may violate the design rule constraints. To remove these violations, you must re-synthesize/optimize the design with the `dont_touch` attribute removed from the sub-blocks.

DFT scan insertion at the top-level is another reason for removing the `dont_touch`

attribute from the sub-blocks. This is due to the fact that the DFT scan insertion cannot be implemented at the top-level, if the sub-blocks contain the dont_touch attribute. The following script exemplifies this process by performing initial synthesis with scan enabled, before re-compiling (compile -only_design_rule) the design with dont_touch attribute removed from all the sub-blocks.

Synthesis Script for the top-level

```
set active_design tap_controller
set sub_modules {tap_bypass tap_instruction tap_state}
for each module $sub_modules { set syn_db $module.db read_db syn_db
}
analyze -format verilog $active_design.v elaborate $active_design
current_design $active_design link
uniquify
set_wire_load_model -name LARGE
set_wire_load_mode enclosed
set_operating_conditions WORST
create_clock -period 33 -waveform [list 0 16.5] tck
set_clock_latency 2.0 [get_clocks tck]
set_clock_uncertainty -setup 3.0 [get_clocks tck] set_clock_transition 0.1 [get_clocks tck]
set_dont_touch_network [list tck trst]
set_driving_cell -cell BUFF1X -pinZ [all_inputs]
set_drive 0 [list tck trst]
set_input_delay 20.0-clock tck-max [all_inputs]
set_output_delay 10.0-clock tck-max [all_outputs]
set_max_area 0
set_fix_multiple_port_nets -all -buffer_constants
compile -scan
remove_attribute [find -hierarchy design {"*"}] dont_touch
current_design $active_design uniquify
check_test
create_test_patterns -sample 10
```

```
preview_scan insert_scan check_test
compile -only_design_rule
remove_unconnected_ports [find -hierarchy cell {"*" }]
change_names -hierarchy -rules BORG
set_dont_touch current_design
write -hierarchy -output $active_design.db
write-format verilog -hierarchy -output $active_design.sv
```

1.10.1.2 SDF Generation

To perform timing simulation, you will need the SDF file for back annotation. The static timing was performed using PT; therefore it is prudent that the SDF file be generated from PT itself as shown in the previous scripts. However, some designers feel comfortable in using DC to generate the SDF file. The design will therefore use DC to generate the SDF in this section.

The following script may be used to generate the pre-layout SDF for the tap_controller design. This SDF file is targeted for simulating the design dynamically with timing. In addition, the script also generates the timing constraints file. Though this file is also in SDF format, it is solely used for forward annotating the timing information to the layout tool in order to perform timing driven layout using the traditional approach.

DC script for pre-layout SDF generation

```
set active_design tap_controller
read_db $active_design.db
current_design $active_designlink
set_wire_load_model LARGE
set_wire_load_mode top
set_operating_conditions WORST
create_clock -period 33 -waveform [list 0 16.5] tck
```

```
set_clock_latency 2.0 [get_clocks tck]
set_clock_transition 0.2 [get_clocks tck]
set_clock_uncertainty 3.0 -setup [get_clocks tck]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
set_drive 0 [list tck trst]
set_load 50 [all_outputs]
set_input_delay 20.0-clock tck-max [all_inputs]
set_output_delay 10.0-clock tck-max [all_outputs]
write_sdf -output $active_design.sdf
write_constraints -format sdf -cover_design -output constraints.sdf
```

1.10.2 Floorplanning and Routing

The floorplanning step involves physical placement of cells and clock tree synthesis. Both these steps are performed within the layout tool. The placement step may include timing driven placement of the cells, which is performed by annotating the constraints.sdf file (generated by DC) to the layout tool. This file consists of path delays that include the cell-to-cell timing information. This information is used by the layout tool to place cells with timing as the main criterion i.e., the layout tool will place timing critical cells closer to each other in order to minimize the path delay.

Let's assume that the design has been floorplanned. Also, the clock tree has been inserted in the design by the layout tool. The clock tree insertion modifies the existing structure of the design. In other words, the netlist in the layout tool is different from the original netlist present in DC. This is because of the fact that the design present in the layout tool contains the clock tree, whereas the original design in DC does not contain this information. Therefore, the clock tree information should somehow be transferred to the design residing in DC or PT. The new netlist (containing the clock tree information) should be formally verified against the original netlist to ensure that the transfer of clock tree did not break the functionality of the original logic. For the sake of simplicity, let us assume that the clock tree information is present in the tap_controller design.

The design is now ready for routing. In a broad sense, routing is performed in two phases – global route and detailed route. During global route, the router divides the layout surface into separate regions and performs a point-to-point “loose” routing without actually placing the geometric wires. The final routing is performed by the detailed router, which physically places the geometric wires and routes within the regions. Lets assume that the design has been global routed.

The next step involves extracting the estimated parasitic capacitances, and RC delays from the global routed design. This step reduces the synthesis- layout iteration time, especially since cell placement and global routing may take much less time than detailed routing the entire chip. However, if the cells are placed optimally with minimal congestion, detailed routing is also very fast. In any case, extraction of delays after the global route phase (albeit estimates) provides a faster method of getting closer to the real delay values that are extracted from the layout database after the detailed routing phase.

Back annotate the estimates to the design in PT for setup and hold-time static timing analysis, using the following scripts.

PT script for setup-time analysis, using estimated delays

```
set active_design tap_controller
read_db -netlist_only $active_design.db
current_design $active_design
set_operating_conditions WORST
set_load 50.0 [all_outputs]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
source capacitance.pt #estimated parasitic capacitances
read_sdf rc_delays.sdf #estimated RC delays
create_clock -period 33 -waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty 0.5 -setup [get_clocks tck]
```

```
set_input_delay 20.0-clock tck [all_inputs]
set_output_delay 10.0-clock tck [all_outputs]
report_Constraint -all_violators
report_timing -to [all_registers -data_pins]
report_timing -to [all_outputs]
```

PT script for hold-time analysis, using estimated delays

```
set_active_design tap_controller
read_db -netlist_only $active_design.db
current_design $active_design
set_operating_conditions BEST
set_load 20.0 [all_outputs]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
source capacitance.p # estimated parasitic capacitances
read_sdf rc_delays.sdf # estimated RC delays
create_clock -period 33 -waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty 0.05 -hold [get_clocks tck]
set_input_delay 0.0-clock tck [all_inputs]
set_output_delay 0.0-clock tck [all_outputs]
report_constraint -all_violators
report_timing -to [all_registers -data_pins] -delay_type min
report_timing -to [all_outputs] -delay_type min
```

The above script back annotates capacitance.pt and rc_delays.sdf file. The capacitance.pt file contains the capacitive loading per net of the design in set_load format, while the rc_delays.sdf file contains point-to-point interconnect RC delays of individual nets. DC (and PT) performs the calculation of cell delay, based upon the output net loading and input slope of each cell in the design.

If the design fails setup-time requirements, you may re-synthesize the design with adjusted constraints or re-floorplan the design. If the design is failing hold-time

requirements, then depending on the degree of violation you may decide to proceed to the final step of detailed routing the design, or re-optimize the design with adjusted constraints.

If re-synthesis is desired then the floorplan (placement) information consisting of the physical clusters and cell locations, should be back annotated to DC. This step is desired because up till now, DC did not know the physical placement information of cells. By annotating the placement information to DC, the post-layout optimization of the design within DC is vastly improved. The layout tool generates the physical information in PDEF format that can be read by DC, using the following command:

```
read_clusters <file name in PDEF format>
```

The script to perform this is similar to the initial synthesis script, with the exception of the annotated data and incremental compilation of the design, as illustrated below:

Script for incremental synthesis of the design

```
set active_design tap_controller
read_db $active_design.db
current_design $active_design link
source capacitance.dc /* estimated parasitic capacitances */
read_timing -f sdf rc_delays.sdf /* estimated RC delays */
read_clusters clusters.pdef /* physical information */

create_wire_load -hierarchy -percentile 80 -output cwl.m.txt
create_clock -period 33 -waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]
set_clock_transition 0.2 [get_clocks tck]
set_clock_uncertainty 3.0 -setup [get_clocks tck]
set_dont_touch_network [list tck trst]
set_driving_cell -cell BUFF1X -pin Z [list all_inputs]
```

```
set_drive 0 [list tck trst]
set_input_delay 20.0-clock tck -max [all_inputs]
set_output_delay 10.0-clock tck -max [all_outputs]
set_max_area 0
set_fix_multiple_port_nets -all -buffer_constants
reoptimize_design -in_place
write -hierarchy -output $active_design.db
write -format verilog -hierarchy -output $active_design.sv
```

The `create_wire_load` command used in the above script creates a custom wire-load model for the `tap_controller` design. The initial synthesis run used the wire-load models present in the technology library that are not design specific. Therefore, in order to achieve better accuracy for the next synthesis iteration, the custom wire-load models specific to the design should be used.

The following command may be used to update the technology library present in DC's memory to reflect the new custom wire-load models. For example:

```
dc_shell-t> update_lib ex25_worst.db cwlm.txt
```

Let's assume that the design has been re-analyzed and is now passing both setup and hold-time requirements. The next step is to detail route the design. This is a layout dependent feature, therefore will not be discussed here.

1.10.3 Post-Layout Steps

The post-layout steps involve, verifying the design for timing with actual delays back annotated; functional simulation of the design; and lastly, performing LVS and DRC.

Let us just presume that the design has been fully routed with minimal congestion and area. The finished layout surface must then be extracted to get the actual parasitic capacitances and interconnect RC delays. Depending upon the layout tool

and the type of extraction, the extracted values are generally written out in the SDF format for the interconnect RC delays, while the parasitic information is generated as a string of set_load commands for each net in the design. In addition, if a hierarchical place and route has been performed, the physical placement location of cells in the PDEF format should also be generated.

1.10.3.1 Post-Layout Static Timing Analysis using PrimeTime

The first step after layout is to perform static timing on the design, using the actual delays. Similar to post-placement, the post-route timing analysis uses the same commands, except that this time the actual delays are back annotated to the design.

Predominantly, the timing of the design is dependent upon clock latency and skew. It is therefore prudent to perform the clock skew analysis before attempting to analyze the whole design. A useful Tcl script is provided by Synopsys through their on-line support on the web, called SolvNET. You may download this script and run the analysis before proceeding. Let us assume that the clock latency and skew is within limits. The next step is to perform the static timing on the design, to check the setup and hold-time violations (if any) using the following scripts:

PT script for setup-time analysis, using actual delays

```
set active_design tap_controller
read_db -netlist_only $active_design.db
current_design $active_design set_operating_conditions WORST set_load 50.0
[all_outputs]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
source capacitance.pt # actual parasitic capacitances
read_sdf rc_delays.sdf # actual RC delays
read_parasitics clock_info_wrst.spf # for clocks etc.
create_clock -period 33 -waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty 0.5 -setup [get_clocks tck]
```

```
set_input_delay 20.0-clock tck [all_inputs]
set_output_delay 10.0-clock tck [all_outputs]
report_constraint -all_violators
report_timing -to [all_registers -data_pins]
report_timing -to [all_outputs]
```

PT script for hold-time analysis, using actual delays

```
set active_design tap_controller
read_db -netlist_only $active_design.db
current_design $active_design
set_operating_conditions BEST
source capacitance.pt # actual parasitic capacitances
read_sdf rc_delays.sdf # actual RC delays
read_parasitics clock_info_best.spf # for clocks etc.
set_load 50.0 [all_outputs]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
create_clock -period 33 -waveform [0 16.5] tck
set_propagated_clock [get_clocks tck]

set_clock_uncertainty 0.05 -hold [get_clocks tck]
set_input_delay 0.0-clock tck [all_inputs]
set_output_delay 0.0-clock tck [all_outputs]
report_constraint -all_violators
report_timing -to [all_registers -data_pins] -delay_type min
report_timing -to [all_outputs] -delay_type min
```

1.10.3.2 Post-Layout Optimization

The post-layout optimization or PLO may be performed on the design to improve or fix the timing requirements. DC provides several methods of fixing timing violations, through the in place optimization (or IPO) feature. As before, DC also

makes use of the physical placement information to perform location based optimization (LBO). In this example, the design will use the cell resizing and buffer insertion feature of the IPO to fix the hold-time violations.

1.10.4 Hold-Time Fixes

The design was synthesized for maximum setup-time requirements. Timing was verified at each step (after synthesis and then, after the global route phase), therefore in all probability the routed design will pass the setup-time requirements. However, some parts of the design may fail hold-time requirements at various endpoints.

If the design fails the hold-time requirements then you should fix the violations by adding buffers to delay the arrival time of the failing signals, with respect to the clock. Let's assume that the design is failing hold-time requirements at multiple endpoints.

There are various approaches to fix the hold-time violations. In this example, the design will utilize the `dc_shell-t` commands to fix the hold time violations, as illustrated below:

DC Script to fix the hold-time violations

```
set active_design tap_controller
read_db $active_design.db
current_design $active_design link
source capacitance.dc /*actual parasitic capacitances */
read_timing -f sdf rc_delays.sdf /*actual RC delays */
read_clusters clusters.pdef /*physical hierarchy info */
create_clock -period 33 -waveform {0 16.5} tck
set_propagated_clock [get_clocks tck]
```

```

set_clock_uneertainty -hold 0.05 tck
set_dont_touch_network [list tck trst]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
set_drive 0 [list tck trst]
set_input_delay _min 0.0 -clock tck-max [all_inputs]
set_output_delay -min 0.0 -clock tck-max [all_outputs]
set_fix_hold tck /* fix hold-time violations w.r.t. tck */
reoptimize_design -in_place
write -hierarchy -output $active_design.db
write -format verilog -hierarchy -output $active_design.sv

```

In the above script, the `set_fix_hold` command instructs DC to fix hold-time violations with respect to the clock `tck`. The `-in_place` argument of the `reoptimize_design` command is the IPO command, which is regulated by various variables. Making use of these variables, DC inserts or resizes the gates to fix the hold time violations. The LBO variables are helpful in inserting the buffers at the correct location, so as to minimize its impact on some other logic path, leading off from the violating path.

After IPO, the design should again be analyzed through PT to ensure that the violations have been fixed using the post-layout PT script illustrated before.

Once the design passes all timing requirements, the post-layout SDF may be generated (from PT or DC) for simulation purposes, if needed. The design will use DC to generate the worst-case post-layout SDF using the script provided below. A similar script may be used to generate the best-case SDF. Obviously, you need to back annotate the best-case extracted numbers from the layout tool to generate the best-case SDF from DC. This solely depends on the layout tool and the methodology being used.

DC script for worst-case post-layout SDF generation

```

set_active_design tap_controller
read_db $active_design.db

```

```

current_design $active_design link
set_operating_conditions WORST
source capacitance.dc /* actual parasitic capacitances */
read_timing rc_delays.sdf /* actual RC delays */
create_clock-period 33 -waveform {0 16.5} tck
set_propagated_clock [get_clocks tck]
set_clock_uncertainty -setup 0.5 [get_clocks tck]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
set_drive 0 [list tck trst]
set_load 50 [all_outputs]
set_input_delay 20.0-clock tck-max [all_inputs]
set_output_delay 10.0-clock tck-max [all_outputs]
write_sdf -output $active_design.sdf

```

It is recommended that formal [23-24] verification be performed again between the source RTL and the final netlist, to check for any errors that may have been unintentionally introduced during the whole process. This is the final step; the design is now ready for LVS and DRC checks, before tape-out.

1.11 Engineering Change Order

This step is an exception to the normal design flow and should not be confused with the regular design cycle. Many designers regard engineering change order (ECO) as the change required in the netlist at the very last stage of the ASIC design flow. For instance, ECO is performed when there is a hardware bug encountered in the design at the very last stage (say, after tape-out), and it is necessary to perform a metal mask change by re-routing a small portion of the design.

As a result ECO is performed on a small portion of the chip to prevent disturbing the placement and routing of the rest of the chip, thereby preserving the rest of the chip's timing. Only the part that is affected is modified. This can be achieved, either by targeting the spare gates incorporated in the chip, or by routing only some of the metal layers. This process is termed as metal mask change.

Normally, this procedure is executed for changes that require less than 10% modification of the whole chip (or a block, if doing hierarchical place and route). If the bug fix requires more than 10% change then it is best to repeat the whole procedure and re-route the chip (or the block).

The latest version of DC incorporates the ECO compiler. It makes use of the mathematical algorithms (also used by the formal verification techniques), to automatically implement the required changes. Making use of the ECO compiler provides designers an alternative to the tedium of manually inserting the required changes in the netlist, thus minimizing the turn-around time of the chip. Some layout tools have incorporated the ECO algorithm within their tool. The layout tool has a built-in advantage that it does not suffer from the limitation of crossing the hierarchical boundaries associated with a design. Also, the layout tool benefits from knowing the placement location of the spare cells (normally included by the designers in the design), thus can target the nearest location of spare cells in order to implement the required ECO changes and achieve minimized routing.

1.12 Objective of the Thesis

This objective of this thesis is the implementation of Digital Phase Locked Loop (DPLL) Interface Design to configure the DPLL. The implementation is done using the ASIC design flow. This DPLL interface will provide the necessary signal to DPLL in both functionality and testing mode.

1.13 Organization of the Thesis

The thesis is divided in to five chapters. The represent deals with the survey of the literature and ASIC design flow that is used in the implementation of this design. Chapter-2 introduces the PLL architecture, DPLL architecture and the need of this interface. Chapter-3 represents the architecture of DPLL Interface and various registers programming. Chapter-4 shows the prelayout and postlayout netlist

simulation waveform. Chapter-5 gives represents the conclusion of this design and finally references are given. Convention used is standard one.

CHAPTER 2 PLL ARCHITECTURE

2.1 Introduction

This chapter introduces the PLL architecture and basics related to PLL design. The Digital Phase Locked Loop Interface (DPLL) need will be explained on the basis of this architecture. A phase-locked loop is a feedback system combining a voltage-controlled oscillator and a phase comparator so connected that the oscillator frequency (or phase) accurately tracks that of an applied frequency- or phase-modulated signal. Phase-locked loops can be used, for example, to generate stable output frequency signals from a fixed low-frequency signal. The first phase-locked loops were implemented in the early 1930s by a French engineer, de Bellescize. However, they only found broad acceptance in the marketplace when integrated PLLs became available as relatively low-cost components in the mid-1960s. The phase locked loop can be analyzed in general as a negative feedback system with a forward gain term and a feedback term. A simple block diagram of a voltage-based negative-feedback system is shown in Fig. 2.1.

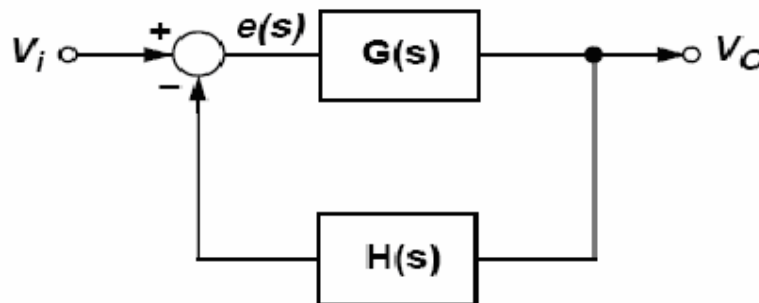


Fig. 2.1 Standard negative-feedback control system models

In a phase-locked loop, the error signal from the phase comparator is the difference between the input frequency or phase and that of the signal fed back. The system will force the frequency or phase error signal to zero in the steady state. The usual equations for a negative-feedback system apply.

$$\text{Forward Gain} = G(s), [s = j\omega = j2\pi f]$$

$$\text{Loop Gain} = G(s) * H(s)$$

$$\text{Closed Loop Gain} = G(s) / \{1 + G(s) * H(s)\}$$

Because of the integration in the loop, at low frequencies the steady state gain, $G(s)$, is high and

$$V_o/V_i, \text{ closed loop Gain} = 1/H$$

The components of a PLL that contribute to the loop gain include:

1. The phase detector (PD) and charge pump (CP).
2. The loop filter, with a transfer function of $Z(s)$
3. The voltage-controlled oscillator (VCO), with a sensitivity of K_V/s
4. The feedback divider, $1/N$.

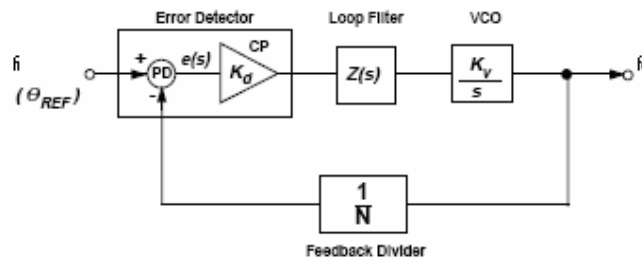


Fig. 2.2 Basic phase-locked-loop model

If a linear element like a four-quadrant multiplier is used as the phase detector, and the loop filter and VCO are also analog elements, this is called an analog or linear PLL (LPLL).

If a digital phase detector (EXOR gate or J-K flip flop) is used, and everything else stays the same, the system is called a digital PLL (DPLL).

If the PLL is built exclusively from digital blocks, without any passive components or linear elements, it becomes an all-digital PLL (ADPLL).

Finally, with information in digital form, and the availability of sufficiently fast processing, it is also possible to develop PLLs in the software domain. The PLL function is performed by software and runs on a DSP. This is called software PLL (SPLL).

In Fig. 2.2, a system for using a PLL to generate higher frequencies than the input, the VCO oscillates at an angular frequency of $\Delta\omega$. A portion of this frequency/phase signal is fed back to the error detector, via a frequency divider with a ratio $1/N$. This divided-down frequency is fed to one input of the error detector. The other input in this example is a fixed reference frequency/phase. The error detector compares the signals at both inputs. When the two signal inputs are equal in phase and frequency, the error will be zero and the loop is said to be in a “locked” condition. If one simply look at the error signal, the following equations may be developed.

In commercial PLLs, the phase detector and charge pump together form the error detector block. When $\omega_o = N \omega_i$, the error detector will output source/sink current pulses to the low pass loop filter. This smoothes the current pulses into a voltage which in turn drives the VCO.

$$e(s) = \omega_i - \omega_o / N$$

$$\text{When, } e(s) = 0, \omega_o / N = \omega_i$$

$$\text{Thus, } \omega_o = N * \omega_i$$

The VCO frequency will then increase or decrease as necessary, by $KV \Delta V$, where KV is the VCO sensitivity in MHz/Volt and ΔV is the change in VCO input voltage. This will continue until $e(s)$ is zero and the loop is locked. The charge pump and VCO thus serves as an integrator, seeking to increase or decrease its output frequency to the value required so as to restore its input (from the phase detector) to zero.

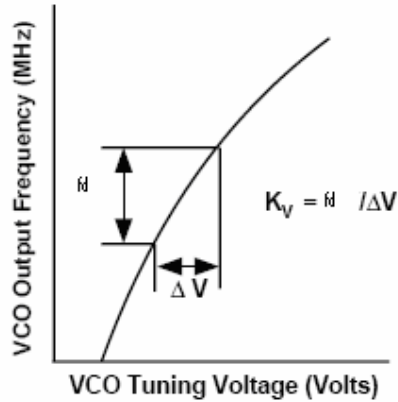


Fig. 2.3 VCO transfer function.

The overall transfer function (CLG or Closed-Loop Gain) of the PLL can be expressed simply by using the CLG expression for a negative feedback system as given above.

$$f_o / f_i = \text{Forward Gain} / (1 + \text{Loop Gain})$$

$$\text{Forward Gain, } G = K_D * K_V * Z(s)/s$$

$$\text{Loop Gain, } GH = K_D * K_V * Z(s)/(N*s)$$

When GH is much greater than 1, one can say that the closed loop transfer function for the PLL system is N and so $f_o = N * \Delta f_i$. The loop filter is a low-pass type, typically with one pole and one zero. The transient response of the loop depends on:

- The magnitude of the pole/zero,
- The charge pump magnitude,
- The VCO sensitivity,
- The feedback factor, N.

All of the above must be taken into account when designing the loop filter. In addition, the filter must be designed to be stable (usually a phase margin of $\Pi/4$ is recommended). The 3-dB cutoff frequency of the response is usually called the loop

bandwidth, BW. Large loop bandwidths result in very fast transient response. Since there is a tradeoff between fast transient response and reference spur attenuation.

2.2 Frequency Synthesis

Frequency Synthesis is the engineering discipline dealing with the generation of multiple signal frequencies, all derived from a common reference or time base. The time base used is typically a Temperature Compensated Crystal Oscillator (TCXO). The TCXO provides a reference frequency to the synthesizer circuit so that it may accurately produce a wide range of signals that are stable and relatively low in phase noise.

2.3 Digital PLL (DPLL) Synthesis

Among the many different frequency synthesis techniques, the dominant method used in the wireless communications industry is the digital PLL circuit. While there are some benefits to using other synthesis techniques, they are outside the scope of my thesis and will not be discussed here.

2.4 Integer-N PLL

Compared to the analog techniques used in the infancy of frequency synthesis, the modern PLL is now a mostly digital circuit. Fig. 2.4 shows a typical block diagram of a PLL implemented with a TCXO reference.

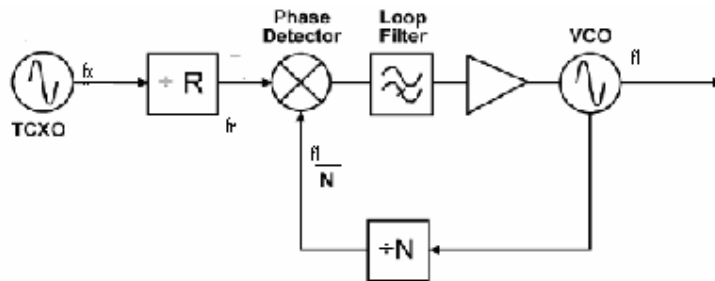


Fig 2.4 Block diagram of a PLL implemented with a TCXO reference

This traditional digital PLL implementation will be termed “integer-N” to avoid confusion due to the addition of fractional-N technology. The PLL circuit performs frequency multiplication, via a negative feedback mechanism, to generate the output frequency, f_1 , in terms of the phase detector comparison frequency, f_r .

$$f_1 = N \cdot f_r \quad \dots\dots\dots 2.1$$

To accomplish this, a reference frequency must be provided to the phase detector. Typically, the TCXO frequency (f_x), is divided down (by R) “on-board” the PLL IC. The phase detector utilizes this signal as a reference to tune the VCO and, in a “locked state,” it must be equal to the desired output frequency, f_1 , divided by N.

$$f_1 / N = f_x / R = f_r \quad \dots\dots\dots 2.2$$

Thus, reprogramming the divider N to a new value can change the output frequency that the synthesizer generates, f_1 . By changing the value N, the VCO can be tuned across the frequency band of interest. The only constraint to the frequency output of the system is that the minimum frequency resolution, or minimum channel spacing, is equal to f_r .

$$\text{Channel spacing} = f_1 / N = f_r \quad \dots\dots\dots 2.3$$

When the PLL is in unlocked state (such as during initial power up or immediately after reprogramming a new value for N) the phase detector will create an error voltage based on the phase difference of the two input signals. This error voltage will change the output frequency of the VCO so that it satisfies Eq. 2. As long as the system is in a locked condition the VCO will have the same frequency accuracy as the TCXO reference. If the crystal accuracy is 1 part-per-million (ppm), the output frequency of the synthesizer will also be accurate to 1 ppm.

Specifically, if $f_r = 30$ kHz and $N = 32000$, the only way for this circuit to be in a stable state (locked) is when $f_1 = 960$ MHz. If N were changed to 32001, a frequency

and phase error will develop at the input of the phase detector that will, in turn, retune the VCO frequency until a locked state has been reached. The locked state will be reached when $f_1 = 960.03$ MHz and, if the TCXO has an accuracy of 1ppm, the output of the VCO will be accurate to $\sim \pm 960$ Hz.

2.5 Fractional-N PLL

An unavoidable occurrence in digital PLL synthesis is that frequency multiplication (by N), raises the signal's phase noise by $20\text{Log}(N)$ dB. The main source of this noise is the noise characteristics of the phase detector's active circuitry. Because the phase detectors are typically the dominant source of close-in phase noise, N becomes a limiting factor when determining the lowest possible phase noise performance of the output signal. A multiplication factor of $N = 30,000$ will add about 90 dB to the phase detector noise floor.

30,000 is a typical N value used by an integer PLL synthesizer for a cellular transceiver with 30 kHz channel spacing. It would seem that one could radically reduce the close-in phase noise of our system by reducing the value of N but unfortunately the channel spacing of an integer- N synthesizer is dependent on the value of N (Eq. 3). Due to this dependence, the phase detectors typically operate at a frequency equal to the channel spacing of the communication system.

A phase detector is a digital circuit that generates high levels of transient noise at its frequency of operation; f_r . This noise is superimposed on the control voltage to the VCO

and modulates the VCO RF output accordingly. This interference can be seen as spurious signals at offsets of $\pm f_r$ (and its harmonics) around f_1 . To prevent this unwanted spurious noise, a filter at the output of the charge pumps (called the loop filter) must be present and appropriately narrow in bandwidth. Unfortunately, as the loop filter bandwidth decreases, the time required for the synthesizer to switch between channels increases.

For a 2nd order loop with natural frequency (loop bandwidth) ω_n and damping factor ξ , the switching speed (T_{sw}) is proportional to the inverse of their product.

$$T_{sw} \propto 1 / \omega_n \xi, \dots\dots\dots 2.4$$

If N could be made much smaller, frequency would increase and the loop filter bandwidth required to attenuate the reference spurs could be made large enough so that it does not impact the required switching speed of our system. Once again, the upper limit of frequency is bound by our channel spacing requirements. This illustrates how our desires to optimize both switching speed and spur suppression directly conflict with each other. A newly emerging PLL technology has made it possible to alter the relationship between N, frequency, and the channel spacing of the synthesizer. It is now possible to achieve frequency resolution that is a fractional portion of the phase detector frequency. This is accomplished by adding internal circuitry that enables the value of N to change dynamically during the locked state. If the value of the divider is “switched” between N and N+1 in the correct proportion, an average division ratio can be realized that is N plus

some arbitrary fraction, K/F. This allows the phase detectors to run at a frequency that is higher than the synthesizer channel spacing.

$$f_l = f_r (N + K / f_r) \dots\dots\dots 2.5$$

N, K & fr are integers. Where fr is the fractional modulus of the circuit (i.e. 8 would indicate a 1/8th fractional resolution.), K is the fractional channel of operation.

From this architecture it’s clear that to operate the DPLL in Integer division or fractional division the design have to program it and insert some signal that will tell us about the mode of signal. The division factor in both forward and feedback path is also programmable; the division factor varies depending upon its mode and input frequency. So for that the design needs an interface that will decide all these things.

CHAPTER 3 DPLL INTERFACE DESIGN

3.1 Introduction to Architecture of DPLL Interface Design (DID)

DPLL Interface is an interface that gives necessary signal to the DPLL. These signals control the whole functionality of DPLL.

This architecture contains a total of seven registers. Each register is used to hold the different set of bits, which are used to program the DPLL. These registers receive bits serially and give the bits to DPLL in parallel form. The bits to be store in the registers are feed serially at each positive clock edge of the clock signal, `clkin`, by putting them on `data_in` input pin. The `reset_b` is a reset input pin which reset the all registers when takes transition from high to low in asynchronous with clock. `data_in_sel` input pin is used to put the register in a mode where they will receive the data from `data_in` pin. `data_out_sel` input pin is used to program the register in output mode. The register to be programmed is selected by `reg_sel` input pin. This architecture also contains seven flip-flops and a mux. The flip-flop holds the MSB of the corresponding registers to which they are connected. When the design have to check whether data has been correctly written or not or data remain in the registers after reading them one can check the registers by comparing their stored bits with bits that one has written. Changing the `reg_sel` bit and enabling the registers in data out mode by setting high the `data_out_sel` pin select all registers one by one. All bits are scanned out in a cyclic manner from these registers and hold by flip-flop, which is further applied to mux. All these are readout one by one at each positive edge of clock signal and compared with the actual values, which one has written.

3.2 Block Diagram

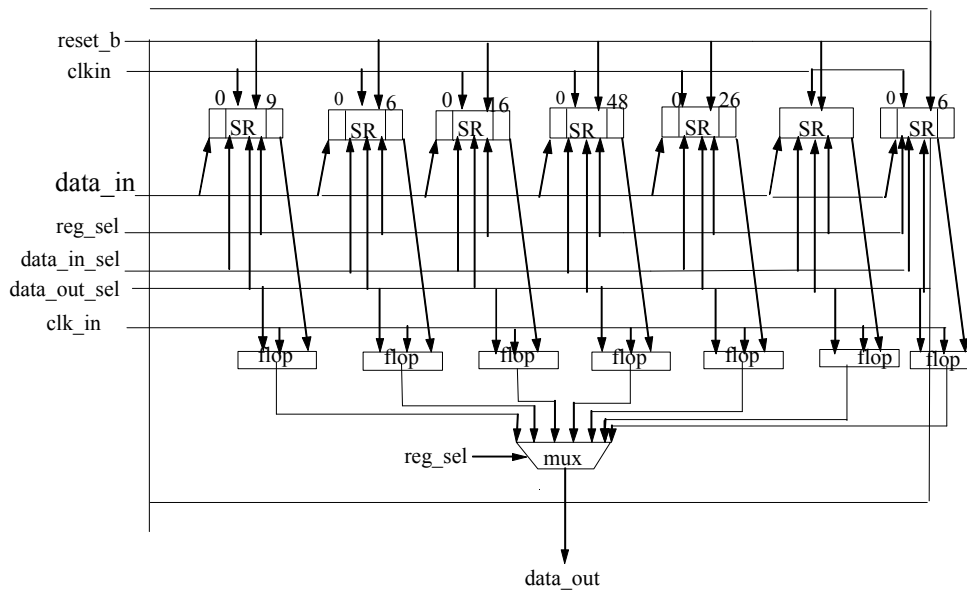


Fig 3.1 DPLL Interface Architecture

Fig. 3.1 shows the PLL interface block diagram inside the test chip. This block contains seven shift registers, gen_reg (10bits wide), dfd_reg (8bits wide), dfm_reg (17bits wide), dpll_reg (49bits wide), dpll_mfn_reg (27bits wide), clkmon_reg (1 bit wide), lvds_opt_reg (7bits wide). Data is serially left shifted in these registers. Data (MSB) is serially scanned out of the shifted in at the LSB (cyclic shifter). All bits of these registers are shifted in a cyclic manner so that after shifting of all bits they got restore in their corresponding registers.

The shift registers, once filled, acts as configuration registers to PLL. The various bits of these registers are as follows:

All these bits are of read-write (RW) types.

Table 3.1 Shift Register bit allocation

BIT POSITION	NAME OF BIT	RESET VALUE
gen_reg (9:8)	clk_in_mux_sel (1:0)	2'b11
gen_reg (7:6)	pll_clk_in_mux_sel (1:0)	2'b0
gen_reg (5:3)	dfd_dfm_clk_out_ready_mux_sel(2:0)	3'b0
gen_reg(2)	load_ack_clkmon_noclk_mux_sel	1'b0
gen_reg(1:0)	dppll_lrf_mux_sel (1:0)	2'b0
dfd_reg (7:5)	dfd_clk_mux_sel (2:0)	3'b0
dfd_reg (4:0)	dfd_fc (4:0)	5'b10011
dfm_reg (16:15)	dfm_clk_mux_sel (1:0)	2'b0
dfm_reg (14:10)	dfm_fc (4:0)	5'b10011
dfm_reg (9:0)	dfm_mf (9:0)	10'b0
dppll_reg (48:38)	cbrmo(10:0)	11'b1
dppll_reg (37)	cdcken	1'b1
dppll_reg (36:10)	cmfd (26:0)	5'b0
dppll_reg (9:6)	cmfi (3:0)	4'b0
dppll_reg (5:2)	cpd (3:0)	4'b0
dppll_reg (1)	cplm	1'b1
dppll_reg (0)	crep	1'b0
dppll_mfn_reg (26:0)	cmfn (26:0)	27'b0
clkmon_reg	freq_sel	1'b0
lvds_opt_reg (6:5)	tres (1:0)	2'b0
lvds_opt_reg (4:0)	lvds_opt (4:0)	5'b0

3.3 PROGRAMMING MODEL

Only one register can be selected at a time for serial data in or data out based upon the setting of reg_sel bits. If:

reg_sel = 3'b000, gen_reg is selected

reg_sel = 3'b001, dfd_reg is selected
reg_sel = 3'b010, dfm_reg is selected
reg_sel = 3'b011, dpll_reg is selected
reg_sel = 3'b100, dpll_mfn_reg is selected
reg_sel = 3'b101, clkmon_reg is selected
reg_sel = 3'b110, lvds_opt_reg is selected

3.4 Shift in to Registers

Registers can be programmed either for shifting in or shifting out. It can not be programmed for both shifting in and shifting out.

Asserting data_in_sel input signal enables shifting in.

data_in_sel=1'b1, shift in is enabled.

data_in_sel =1'b0, shift in is disabled.

gen_reg will take a total of 10 cycles of clk_{in} (posedge) after the assertion of data_in_sel and deassertion of reg_sel to completely program the register.

To write gen_reg:

data_in_sel = 1'b1 and reg_sel = 1'b000.

dfd_reg will take total of 8 cycles of clk_{in} (posedge) after the assertion of data_in_sel and deassertion of reg_sel to completely program the register.

To write dfd_reg:

data_in_sel = 1'b1 and reg_sel = 3'b001

dfm_reg will take total of 17 cycles of clk_{in} (posedge) after the assertion of data_in_sel and deassertion of reg_sel to completely program the register.

To write dfm_reg:

data_in_sel = 1'b1 and reg_sel = 3'b010

dpll_reg will take total of 49 cycles of clk_{in} (posedge) after the assertion of data_in_sel and deassertion of reg_sel to completely program the register.

To write dpll_reg:

data_in_sel = 1'b1 and reg_sel = 3'b011

dpll_mfn_reg will take total of 27 cycles of clkin (posedge) after the assertion of data_in_sel and deassertion of reg_sel to completely program the register.

To write dpll_mfn_reg:

data_in_sel = 1'b1 and reg_sel = 3'b100

clkmon_reg will take total of 1 cycle of clkin (posedge) after the assertion of data_in_sel and deassertion of reg_sel to completely program the register.

To write clkmon_reg:

data_in_sel = 1'b1 and reg_sel = 3'b101

lvds_opt_reg will take total of 7 cycles of clkin (posedge) after the assertion of data_in_sel and deassertion of reg_sel to completely program the register.

To write lvds_opt_reg:

data_in_sel = 1'b1 and reg_sel = 3'b110.

Note: data_out_sel should not be asserted till the registers are programmed. data_out will remain deasserted during shifting in. MSB has to be shifted first in all gen_reg to lvds_opt_reg.

3.5 Shift out of Registers

Asserting data_out_sel input signal enables shift out. Shifted data is visible on data_out output port.

data_out_sel = 1'b1, shift out is enabled.

data_out_sel = 1'b0, shift out is disabled.

gen_reg will take total of 10 cycles of clkin (posedge) after the assertion of data_out_sel and deassertion of reg_sel to completely shift out the register value.

After 10 cycles, the register will regain its programmed value.

Table 3.2 Programming Model for data_in

reg_sel	Description
000	gen_reg is serially shifted with data_in
001	dfd_reg is serially shifted with data_in
010	dfm_reg is serially shifted with data_in
011	dppll_reg is serially shifted with data_in
100	dppll_mfn_reg is serially shifted with data_in
101	clkmon_reg is serially shifted with data_in
110	lvds_opt_reg is serially shifted with data_in

Table 3.2 is for data_out_sel=1'b0 and data_in_sel= 1'b1.

To shift out gen_reg:

data_out_sel = 1'b1 and reg_sel = 1'b000.

gen_reg will take total of 10 cycles of clkin (posedge) after the assertion of data_out_sel and deassertion of reg_sel to completely shift out the register value. After 10 cycles, the register will regain its programmed value.

To shift out dfd_reg:

data_out_sel = 1'b1 and reg_sel = 1'b001.

dfd_reg will take total of 8 cycles of clkin (posedge) after the assertion of data_out_sel and deassertion of reg_sel to completely shift out the register value. After 8 cycles, the register will regain its programmed value.

To shift out dfm_reg:

data_out_sel = 1'b1 and reg_sel = 1'b010.

dfm_reg will take total of 17 cycles of clkin (posedge) after the assertion of data_out_sel and deassertion of reg_sel to completely shift out the register value. After 17cycles, the register will regain its programmed value.

To shift out dppll_reg:

data_out_sel = 1'b1 and reg_sel = 1'b011.

dp11_reg will take total of 49 cycles of clkin (posedge) after the assertion of data_out_sel and deassertion of reg_sel to completely shift out the register value. After 49cycles, the register will regain its programmed value.

To shift out dp11_mfn_reg:

data_out_sel = 1'b1 and reg_sel = 1'b100.

dp11_mfn_reg will take total of 27 cycles of clkin (posedge) after the assertion of data_out_sel and deassertion of reg_sel to completely shift out the register value.

After 27cycles, the register will regain its programmed value.

To shift out clkmon_reg:

data_out_sel = 1'b1 and reg_sel = 1'b101.

clkmon_reg will take total of 1 cycle of clkin (posedge) after the assertion of data_out_sel and deassertion of reg_sel to completely shift out the register value.

After 1cycle, the register will regain its programmed value.

To shift out lvds_opt_reg:

data_out_sel = 1'b1 and reg_sel = 1'b110.

lvds_opt_reg will take total of 7 cycles of clkin (posedge) after the assertion of data_out_sel and deassertion of reg_sel to completely shift out the register value.

After 7cycles, the register will regain its programmed value. Table 3.3 is for data_out_sel=1'b1 and data_in_sel= 1'b0.

Table 3.3 Programming Model for data_out

reg_sel	Description
000	gen_reg (MSB) is serially shifted with data_out.
001	dfd_reg (MSB) is serially shifted with data_out.
010	dfm_reg (MSB) is serially shifted with data_out.
011	dp11_reg (MSB) is serially shifted with data_out.
100	dp11_mfn_reg (MSB) is serially shifted with data_out.
101	clkmon_reg (MSB) is serially shifted with data_out.
110	lvds_opt_reg (MSB) is serially shifted with data_out.

3.6 Writing into the Shift Registers

The programming of the shift registers to configure the DPLL in a particular mode should be done according the following protocol:

1. ASSERT the reset signals of PLL interface and the DPLL i.e. 'reset_b' and 'cpres' respectively.
2. Apply 'reg_sel' = 3'b000, to select shift register gen_reg.
3. Apply 'data_in_sel=1'b1 & data_out_sel = 1'b0 to enable shifting 'data_in' into the registers and to disable shifting out from the shift registers.
4. Apply clk_in and DEASSERT 'reset_b'.
5. Apply data_in according to table1.1 for 8 consecutive cycles of clk_in to program gen_reg completely.
6. Apply 'reg_sel' = 3'b001, to select shift register dfd_reg.
7. Apply data_in according to table 1.1 for 6 consecutive cycles of clk_in to program dfd_reg completely.
8. Repeat step 6 & 7 for dfm_reg to lvds_opt_reg to program the dfm_reg to lvds_opt_reg completely.
9. Now DEASSERT 'cpres' and hold the register values for at least 3 cycles of 'crckd' – the reference clock for DPLL, so that the new configuration is stored in the DPLL.

The shift registers, once filled, act as configuration register to the PLL.

3.7 Reading the Shift Registers

There is a provision to read data serially from the shift registers:

1. Apply 'reg_sel' = 3'b000 to 3'b110 to select shift register reg-1 to lvds_opt_reg respectively.

2. After the registers have been programmed apply 'data_in_sel' =1b'0 & 'data_out_sel'= 1b'1 to enable shifting data out of the registers and to disable shifting.
3. 'data_in' into registers. The serial data shifted out is obtained on the port 'data_out'.
4. gen_reg takes a total of 8 cycles of clk_in to completely shift out the register value. After 8 cycles the register will regain its programmed value.
5. Similarly, other registers will take a total of cycles equal to the number of bits stored in them to completely shift out the register value and to regain their programmed value.

3.8 Input Constraints

This DPLL module was designed for a frequency of 100MHz. The various constraints in this design are as follows:

Table 3.4 Input Constraints

S.N.	Constraint	Value
1	Duty cycle	50%
2	Clock uncertainty	1ns
3	Input delay	2.5ns
4	Output delay	7ns
5	Maximum input transition	0.3ns
6	Maximum output transition	0.7ns
7	Load at all outputs	0.3pF
8	Clock latency	0.50ns

CHAPTER 4 RESULTS AND DISCUSSION

4.1 Introduction

This chapter shows the pre layout and post layout simulation of the design. Both pre layout and post layout waveforms are shown in Fig. 4.1 and Fig. 4.2 respectively using the .fsdb file.

4.2 Prelayout Waveform

The prelayout simulation results are shown in the Fig. 4.1. Debussy tool is used to view the waveforms. It reads the .fsdb file generated by VCS simulator and displays the waveforms.

In the prelayout simulation, the prelayout gate level netlist is tested through a test bench written in Verilog language. The test vectors are applied in the module of test bench. This test module generates the .fsdb file, which is read into Debussy as shown in the fig. 4.1. The data is read or write at the positive edge of clkin clock signal. The data_in signal allows the data to be written in the registers when the data_in_sel input is high. The reg_sel register selects the particular register to be read or written. Read is enabled when data_out_sel is high. As all these registers are programmed in cyclic mode, they will restore the data after completely reading the data. The reset_b signal causes the reset values to be stored in all registers.

In this waveform reg_sel =0, selects the gen_reg, data_in_sel=1 causes the data to be written in this registers. The data at the data_in pin is written at each positive edge of clkin signal.

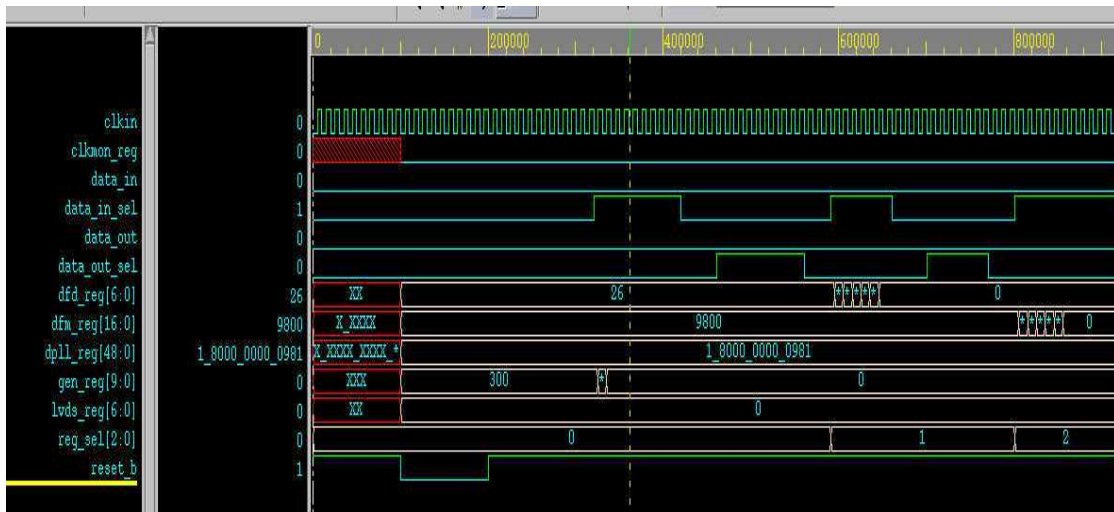


Fig. 4.1 Prelayout Simulation .fsdb file's waveform in Debussy

4.3 Timing Check

In timing check, there are eight ports with parasitic but with no driving cell. These ports are input ports, so these warning are ignored. This check was performed to check either the design meets input constraints in terms of setup and hold time violation. A slack of 0.0 is found at each true path. This means that design has no timing violation. Both setup and hold time check were performed separately because for both libraries used are different. Both check gives same warning. The input ports, which show warning, are as follows:

Ports

clk_in

data_in

data_in_sel

data_out_sel

reg_sel [0]

reg_sel [1]

reg_sel [2]

reset_b

4.4 Read Parasitic Report

To perform the setup time and hold time check, the parasitic information is necessary. This parasitic information is obtained by generating the .sdf file. This .sdf file contains the delay of each path in terms of resistance and capacitance. In this .sdf file derived library resistance unit is 1.000000 Kohm, time unit is 1 ns, and capacitance unit is 1.000000 pF. The report of parasitic reading is as follows:

```
0 error(s)
Format is SPEF
Annotated nets: 615
Annotated capacitances: 34489
Annotated resistances: 33874
Reduced coupling capacitances: 224414
Annotated PI models: 0
Annotated Elmore delays: 0
```

1

The .sdf file has been successfully read with 0 error. This also shows the number of annotated nets, annotated capacitance, annotated resistance etc. The ‘ 1 ’ at the end of report indicate successfully reading of the .sdf file.

4.5 Report Physical Optimization

The physical optimization is performed after the physical synthesis. This optimization is done to further optimize the design. The physical compiler is used for physical optimization of the design. The report of physical optimization is as follows:

```
Cap for layer M1: 0.00026 0.00026
Res for layer M1: 0.0017 0.0017
Cap for layer M2: 0.00029 0.00029
Res for layer M2: 0.0011 0.0011
Cap for layer M3: 0.00028 0.00027
```

Res for layer M3: 0.0011 0.0011
 Cap for layer M4: 0.00028 0.00028
 Res for layer M4: 0.0011 0.0011
 Cap for layer M5: 0.00028 0.00028
 Res for layer M5: 0.0011 0.0011
 Cap for layer M6: 0.00027 0.00027
 Res for layer M6: 0.00015 0.00015
 Res for layer M7: 4.5e-05 4.5e-05
 Horizontal Cap: 0.00029 0.00029
 Horizontal Res: 0.00098 0.00098
 Vertical Cap: 0.00028 0.00028
 Vertical Res: 0.00076 0.00076
 Std cell utilization: 57.74% (7013/(12880-735)) (Non-fixed + Fixed)
 Std cell utilization: 57.74% (7013/(12880-735)) (Non-fixed only)
 Chip area: 12880 sites, (0.60 0.60 37.40 126.60) um
 Std cell area: 7013 sites, (non-fixed: 7013 fixed:0)
 Macro cell area: 0 sites 0 cells
 Placement blockages: 735 sites, (excluding fixed std cells)
 735 sites, (include fixed std cells & chimney area)
 0 sites, (complete net blockages)
 Routing blockages: 0 sites, (partial net blockages)
 0 sites, (routing blockages and signal pre-route)
 Avg. std cell width: 2.00 um
 Physical DB scale: 1000 db_unit = 1 um
 Number of Violating Paths: 0
 Total moveable cell area: 2524.7
 This report shows the result of physical optimization. In this report the capacitances and resistances associated with each metal layer is given after the physical optimization. Other information like standard cell utilization, blockage area and movable area are annotated.

4.6 Post Layout Waveform and Results

The post layout netlist is also tested in the same way as the prelayout netlist through a test bench written in Verilog language. Fig. 4.2 shows the prelayout waveforms. The test vectors are applied through this test bench.

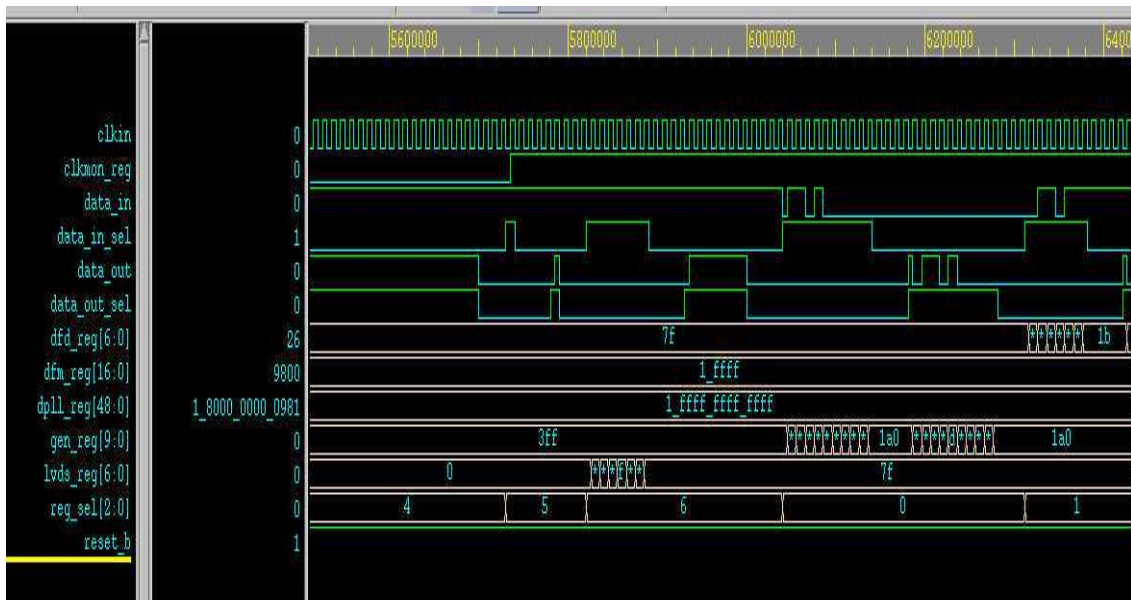


Fig. 4.2 Post Layout Simulation .fsdb file's waveform in Debussy

The total number of cells, area etc. final result of design are as follows:

Total no. of combinational cells: 1947

Total no. of sequential cells: 138

Total no. of cells: 2085

Number of Violating Paths: 0

Total moveable cell area: 2524.7

Total fixed cell area: 0.0

Core area: 37.4 μm * 126 μm

Combinational area: 1263.599243

Noncombinational area: 1355.040771

Total cell area: 2618.640137

These results meet the desired area (38 μm * 127 μm) and operating frequency (100 MHz) of the design. The prelayout simulation and post layout simulation both meet the functionality of design specified in the RTL written in Verilog language.

Finally, this postlayout netlist is used to stream in to the ICFB for layout. The prelayout gate-level netlist is used to stream in to the ICFB for schematic. The layout passes both DRC and LVS check against prelayout netlist generated gate-level schematic. Hence it is now ready for tape out of the design.

CHAPTER 5 CONCLUSION AND FUTURE SCOPE

The aggressive time to market and the short product life span in the computer segment require a design methodology that can offer the best trade-off between this designed fast cycle time as addressed by classic ASIC methodology and the performance and die-size benefits of full custom design. To meet this challenge a unique design flow was used that allow this designed for the best mix of multivendor CAD and in-house tools.

To improve the performance of the clock tree, an interaction strategy between this designed logic and physical ASIC design phases was used. The logic clock tree synthesis can drive Place & Route tools in the physical implementation of the clock network, targeting skew and reducing clock wire congestion. Simulated results confirm the validity of both the logic clock tree synthesis, and the layout driving technique, which keeps together the components driven by a single driver of the clock network. Moreover, the CPU time of the clock network generation is acceptable, being a very small percentage of the synthesis and optimization time, whereas Place & Route execution time does not increase at all.

Finally, to accurately evaluate both clock skews and phase delays inside the logic design environment, a dedicated timing analysis of the clock tree was used by post layout back annotation into the synthesis circuit database.

Further developments are possible by refining the connectivity analysis, evaluating path delays more accurately. Other improvements could be achieved by making the interaction between this design logic and physical design phases stronger. A possibility in this direction is the execution of a preliminary floorplanning, before the clock tree synthesis. In this way, during partitioning of the clocked cells, the position of the synchronized elements on the core could also be taken into account, together with the connectivity information, to cluster together the neighboring cells, and, on

the other hand, not to connect to the same driver two cells placed too far one from the other.

Typical metrics used during rough placement include estimates of wire lengths, wire congestion, timing criticality of wires, communication between this design partitions etc., to mention a few. In the past, various approaches to timing driven placement have been proposed. These approaches can be classified into two techniques. The best uses static timing analysis to determine budgets for net lengths based on slack computations. These budgets are periodically updated during the placement iterations. This approach is a heuristic but has the advantage of not impacting the run time of the placement algorithm adversely. The second approach incorporates timing analysis into the placement formulation as a constraint to the placement problem. Although a superior technique, this method is limited in the size of the circuits that it can handle in a reasonable amount of time. In this design focus was on the best approach for capacity and efficiency. For ASIC designs, there is a fundamental law with using only slack to drive the placement. Gates in ASIC libraries are characterized for delay using table lookup models, which are two-dimensional tables.

This DPLL Interface Design is designed for specific application of DPLL configuration. This DPLL Interface design is an improvement over the existing DPLL interface design that has only two registers for data storage. The previous DPLL Interface was not optimized at RTL level. This architecture removes the redundant logic. This architecture can be extended as per the requirement and constraints of design. This ASIC design methodology using PC with time driven placement is a time saving and gives better placement of cells that can give the tight constraint to be meet.

This design uses the library of CMOS 65 nm technology cells for its implementation. This design has the clock of 100 MHz so design constraint was not so much tight but even Physical Compiler (PC) was used for design synthesis and optimization. PC allows integrating both synthesis and placement thus avoid delay computation based on the wire load model. The wire load model is not efficient at 65 nm design because it does not include the interconnect parasitic .The interconnect delay is comparable

with the cell delay. So, design with PC allows us faster implementation and meeting the targeted constraints.

The floorplan was also done on Encounter that allows us to change the pin location as per the requirements. This makes better top-level integration of module. The design also includes ECO and provides a margin for that. The effective utilization of core area is fixed at 60%. So, if there are any changes due to top level integration or any bug is found in at the time of tape out, they can be corrected at that time. Filler cells are used to fill the rest of core area.

REFERENCES

- [1] Alok Mehrotra, Lukas van Ginneken, Yatin Trivedi, “*Design Flow and Methodology for 50M gate ASIC*”, Proc. IEEE Custom Integrated Circuits Conf., pp. 640-647, 21-24, Jan. 2003.
- [2] M. Cote and P. Hurat. “*Better cells for better designs in closing gap between full-custom and ASIC*”. IEEE Transaction on Computer Aided Design, vol. 18, no. 9, pp. 1305-1315, Sept. 1999.
- [3] M. Vujkovic and C. Sechen. “Optimized power-delay curve generation for standard cell ICs”. Design Automation Conference, pp. 387-394, 10-14, Nov. 2002.
- [4] Math Muris, “*Integrating Boundary Scan Test into an ASIC Design Flow*”, IEEE International Test Conference, pp.124-127.10-14, Sep. 1990.
- [5] S. Devadas, A. Ghosh, and K. Keutzer. “*Logic Synthesis*”, McGraw-Hill, 1994.
- [6] R. Rudell. “*Logic Synthesis for VLSI Design*”. PhD thesis, University of California, Berkeley, 1989.
- [7] K. Keutzer. “*DAGON: Technology Binding and Local Optimization by DAG Matching*”. Proceedings of the Design Automation Conference, ACM/IEEE, pp. 341-347, 28-1, June1987.
- [8] O. Coudert, R. Haddad, and S. Manne. “*New Algorithms for Gate Sizing: A Comparative Study*”. Proceedings of the Design Automation Conference, ACM/IEEE, pp. 734-739, 3-7, June1996.
- [9] C. L. Berman, J. L. Carter, and K. F. Day. “*The Fanout Problem: From Theory to Practice*”. Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference, pp. 69-99, 1989
- [10] K. Kodandapani, J. Grodstein, A. Domic, and H. Touati. “*A Simple Algorithm for Fanout Optimization using High-Performance Boer Libraries*”. Proceeding IEEE on Digest of the International Conference on Computer-Aided Design, pp. 466-471, 1993.

- [11] R. B. Hitchcock, G. L. Smith, and D. D. Cheng. “*Timing Analysis of Computer Hardware*”. Technical report, IBM, 1982.
- [12] R. B. Hitchcock. “*Timing Verification and Timing Analysis Program*”. 25 Years of 19th Electronic Design Automation. IEEE/ACM, pp. 325-329, 14-16, June 1988.
- [13] B. Kernighan and S. Lin. “*An efficient heuristic procedure for partitioning graphs*”. Bell Sys. Tech. J., pp. 291-308, 1970.
- [14] C. M. Fiduccia and Mattheyses. “*A Linear Time Heuristic for Improving Network Partitions*”. Proceedings IEEE of Design Automation Conference, pp. 175-181, 14-16, June 1982.
- [15] C. J. Alpert and A. B. Kahng. “*Recent Directions in Netlist Partitioning: A Survey*”. Integration, the VLSI Journal, pp. 432-436, 1995.
- [16] C. J. Alpert, J.-H. Huang, and A. B. Kahng. “*Multilevel Circuit Partitioning*”. Proceedings of the Design Automation Conference, pp. 530-533, 1997.
- [17] M. A. Breuer. “*Min-Cut Placement*”. J. Design Automation and Fault Tolerant Computing, pp. 124-127, 1977.
- [18] M. Burstein and S. J. Hong. “*Hierarchical VLSI Layout*”. Proceedings of VLSI, 2003.
- [19] U. Lauther. “*A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graph representation*”. Proceedings of the Design Automation Conference, pp. 1-10, 25-27, June 1979.
- [20] S. Kirkpatrick, C. D. Gelatt Jr., and M.P. Vecchi. “*Optimization by Simulated Annealing*”. Science, vol. 220, pp. 671-680, 2003.
- [21] F. Romeo and A. Sangiovanni-Vincentelli. “*Convergence and finite time behavior of simulated annealing*”. Proceedings of the 24th Conference on Decision and Control, pp. 761-767, 1985.
- [22] C. Sechen. “*Chip planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing*”. Proceedings of the Design Automation Conference, pp. 73-80, 12-15, June 2002.

- [23] Shiva Embanatha and Ramkrishnan Venkata, “*Exceptional ASIC: Through Automatic Timing Exception Generation (ATEG)*”. 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06), pp. 503-506, 3-7, June 2006.
- [24] Viktor Owall, Hugo Hedberg, Methias Kamuf and J.N. Rodrigues, “*A Manual on ASIC front to Back End Design Flow*”, IEEE International Conference on Microelectronic Systems Education (MSE'05), pp, 75-76, 12-14, June 2005