Abstract

MINEO, CHRISTOPHER ALEXANDER. Clock Tree Insertion and Verification for 3D Integrated Circuits. (Under the direction of Dr. W. Rhett Davis)

The use of three dimensional chip fabrication technologies has emerged as a solution to the difficulties involved with the continued scaling of bulk silicon devices. While the technology exists, it is undervalued and underutilized largely due to the design and verification challenges a complex 3D design presents. This work presents a clock tree insertion and timing verification methodology for three dimensional integrated circuits (3DIC). It has been designed in the context of and incorporated into the 3DIC design methodology also developed within our research group. The 3DIC verification methodology serves as an efficient means to perform all setup and hold timing checks harnessing the power of existing commercial chip design and verification tools. A novel approach is presented in which the multi-die design is temporarily transformed to appear as a traditional 2D design to the commercial tools for verification purposes. Various parasitic extraction algorithms are examined, and we present a method for performing accurate 3D parasitic extraction for timing purposes. We offer theoretical insight into the optimization of a 3D clock tree for power savings and coupling-induced delay minimization. A practical example of the 3DIC design and verification flow is detailed through the explanation of our research group's test chip, a nearly 140,000 cell 3D fast Fourier transform chip currently awaiting fabrication at MIT's Lincoln Labs.

CLOCK TREE INSERTION AND VERIFICATION FOR 3D INTEGRATED CIRCUITS

by

CHRISTOPHER ALEXANDER MINEO

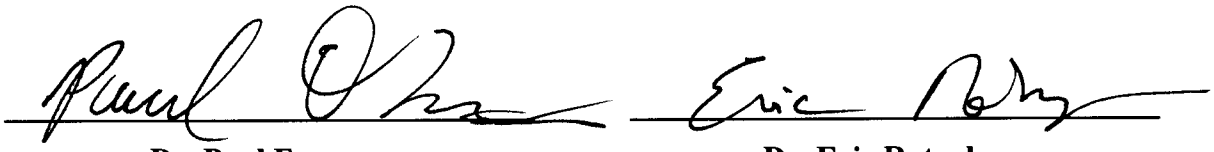
A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER ENGINEERING

Raleigh

2005

APPROVED BY:



Dr. Paul Franzon **Dr. Eric Rotenberg**

Dr. W. Rhett Davis

Chair of Advisory Committee

Biography

Christopher Alexander Mineo was born on September 19, 1981 in Manhasset, NY. For most of his life he has lived in northern New Jersey with loving parents Ron and Paula, and wonderful sister Kim. Chris received the Bachelor of Engineering degree in Electrical and Computer Engineering with a VLSI concentration from Rutgers, The State University of New Jersey—New Brunswick in 2003. In the fall of 2003 he began the Master of Science in Computer Engineering program at North Carolina State University in Raleigh. Shortly thereafter, he joined the Methodologies for User-friendly System-on-a-chip Experimentation research group under the advisement of Dr. Rhett Davis.

Since graduating from Rutgers University, Chris has worked at BAE Systems North America in Wayne, NJ. There he primarily wrote VHDL to program FPGA's for communication systems. He also works at IBM in Research Triangle Park, NC. At IBM Chris works with both the custom digital circuits VLSI teams and the timing teams on the next generation PowerPC processor.

Chris currently works as a research assistant at NC State University on the 3DIC project funded by DARPA. This project involves primarily the design of complex digital systems, design tool modification and development, and design and verification methodology development.

Acknowledgements

I would first like to thank my parents and sister for the love and support they continually give both on this project and everything else I am involved with. I surely could never be where I am now without their encouragement. It is wonderful to know I can always count on them no matter what is going on.

I certainly owe many thanks to Dr. Davis for the insight and motivation that has got me to finish this project. He is an excellent advisor with genuine interest in all aspects of chip design and enthusiasm that cannot help but rub off on those around him. He always seems to be able to make himself available and takes a real interest in each of his students to make sure they succeed.

I also thank Dr. Franzon for his interest in the areas of this work, for serving on my thesis committee, and for being an advisor on the 3DIC research project. I thank Dr. Rotenberg for also serving on my committee and looking at my work from a computer architecture point of view. Thanks to Dr. Steer for also being a faculty advisor on the 3DIC project and helping with the background knowledge about parasitic extraction.

I appreciate the help Hao Hua and Ambarish Sule, other members of my research group, were always willing to give. They are also working on the 3DIC project; Hao helped me a lot with the specifics of the 3DIC design flow and with making some of the flow charts. Ambarish designed the FFT used as a test chip to demonstrate all of our work, and helped me out also with the figures below and the specifics of the architecture of the test chip that I discuss.

I also definitely thank my friends for the support (and distraction) they provide. First, my girlfriend Lana is always there to relax with, work with, and always seems to be proud of me. My buddies from home are always ready to stop what they are doing when I come home and are only a phone call away, thanks so much to Ryan, Mike, Kris, and the rest of you guys. Thanks to Brandon, Kory, Dave, Samson and my friends around here for helping me to keep my sanity on and off campus.

Table of Contents

List of Tables vi
List of Figures vii
Chapter 1: Introduction 1
Chapter 2: 3DIC Design Process 7
2.1 3DIC Technology 7
2.2 3DIC Design and Verification Flow 9
2.2.1 Inputs to the 3DIC Design and Verification Flow 9
2.2.2 Partitioning 10
2.2.3 Via Insertion 11
2.2.4 Floorplanning and Physical Design 12
2.2.5 Verification 14
2.3 Clock Tree Insertion 18
Chapter 3: 3DIC Timing Verification Methodology 29
3.1 Inputs to the 3DIC Timing Verification Flow 29
3.2 SPEF File Merging 30
3.3 Circuit Simulation of Clock Tree 34
3.4 Static Timing Verification 43
3.5 Timing Closure 48
Chapter 4: Parasitic Extraction 50
4.1 RC Extraction 50
4.2 Parasitic Extraction Algorithms 51
4.3 Electromagnetic Field Solvers and Capacitance Matrices 52
4.4 3D Extraction versus 2.5D Extraction 55
Chapter 5: Results and Analysis 63
5.1 FFT Architecture 63
5.2 Static Timing versus Circuit Simulation 68
Chapter 6: Conclusion 73
Bibliography 76

List of Tables

Table 5-1: Spice Simulation Timing Results from FFT (with Coupling Capacitors)..... 69
Table 5-2: Static Timing Analysis Timing Results from FFT (with Coupling Capacitors)... 69
Table 5-3: Spice Simulation Timing Results from FFT (without Coupling Capacitors) 69
Table 5-4: Static Timing Analysis Timing Results from FFT (without Coupling Capacitors) 69
Table 5-5: Spice Simulation Timing Results for Data Paths 71
Table 5-6: Static Timing Analysis Timing Results for Data Paths..... 71

List of Figures

Figure 1-1: The Four Valid Timing Arcs of an AND Gate 3
Figure 2-1: Cross Section of a 3D Process 7
Figure 2-2: 3DIC Design and Verification Flow 9
Figure 2-3: 3D Chain of Three Inverters 11
Figure 2-4: Cross Section of a 3D Via 12
Figure 2-5: 3DIC Verification Flow 14
Figure 2-6: 3DIC Timing Verification Flow 17
Figure 2-7: Ideal Zero-Skew 8 Level H-Clock Tree with Buffers 19
Figure 2-8: Pi Model of Interconnect with Driver and Load 20
Figure 2-9: Pi Model of Interconnect with Coupling Capacitors 23
Figure 2-10: FFT Timing Uncertainty Plot 26
Figure 2-11: FFT Clock Power Consumption Plot 27
Figure 3-1: Simulation Based Timing Verification Flow 34
Figure 3-2: Static Timing Based Timing Verification Flow 43
Figure 4-1: Capacitance Matrix Formulation [6] 53
Figure 4-2: Example Net 1 in First Encounter 55
Figure 4-3: Example Net 1 in Q3D 56
Figure 4-4: Example Net 2 in First Encounter 57
Figure 4-5: Example Net 2 in Q3D 58
Figure 4-6: Inter-tier Via Modeled in Q3D 59
Figure 4-7: Shielded Inter-tier Via Modeled in Q3D 61
Figure 4-8: Winograd FFT Architecture 62
Figure 5-1: FFT Tier A Clock Tree 64
Figure 5-2: FFT Tier B Clock Tree 66
Figure 5-3: FFT Tier C Clock Tree 68

Chapter 1: Introduction

We are rapidly approaching the 35 nm technology mark where one billion transistors will physically fit on a single traditional bulk silicon chip. The current trend of device scaling in bulk silicon CMOS processes has been progressing for 30 years, but in order to continue building high performance circuits, threshold voltages would have to shrink proportionately. This is beginning to cause a number of other significant problems previously unseen, including leakage current, noise susceptibility, and yield due to manufacturing defects. Experts believe that if the trend of increasing chip complexity known as Moore's Law is to continue, the industry will undoubtedly turn to alternatives to classic bulk silicon technology. Well known possibilities include double gate fully-depleted silicon-on-insulator (SOI) devices, fin-FET transistors, and the topic of this paper, three-dimensional SOI processes [1]. The 3D SOI processes are particularly appealing because they allow for chips of higher complexity while also addressing the problem of increased wire delays in today's chips.

In modern technologies, interconnect in circuits has overtaken gate delay as the dominant source of path delay in complex chips. Multiple dies (also known as tiers) in a single integrated circuit (IC) package not only allow designers to fit more transistors—hence additional functionality—in a chip, but also drastically reduce average interconnect length between circuits. Shorter wires, and in turn smaller wiring capacitances, offer both performance and power saving advantages over traditional two-dimensional processes. Our research group tested these claims by fabricating a chip in such a 3D process, overcoming multiple challenges in the process. One important challenge, on which this work is focused, is the insertion of a clock tree and the timing verification of a 3D chip. We needed a design

and verification methodology for three dimensional integrated circuits (3DIC) immediately with the available tools. However, today's design tools are based on 2D chip design. An entire suite of design software is needed to bring a new chip to fruition in a competitive amount of time; and we, like industrial chip designers, do not have the time to wait for commercial tool vendors to address our needs.

Aside from the lack of 3D design features in tools, every chip designer is aware of the shortcomings of modern design tools with how fast the industry is evolving. Many modern day tools are the result of large companies acquiring a number of smaller companies and doing their best to integrate all the tools together. As technologies evolve, the requirements of design tools change and the electronic design automation (EDA) industry is often reluctant to make the enormous time and financial commitment involved with redesigning a tool from the ground up. Chip designers that have strict deadlines to meet certainly cannot rely on EDA companies to release new tools or fixes for every difficulty they encounter. The result is designers and in-house support teams regularly incorporating their own design methodologies and custom EDA tool fixes into a standard design flow. Such is the case with the recent introduction of three-dimensional IC manufacturing processes. In a 3DIC process, separate dies are combined and interconnected within a single package. The dies are fabricated on different wafers, quite possibly in different foundries. The wafers are brought together, ground very thin, aligned, and stacked. Inter-tier vias are created, which much like traditional vias, are vertical segments that connect metal layers. The inter-tier vias, however, connect a metal layer in one die with a metal layer in another die and are significantly larger than traditional vias. These inter-tier vias will be discussed in detail in chapter 2 and are shown in figures 2-3 and 2-4. Design tools commercially available today have no notion of

any of this, especially inter-tier connections. The problem that will be addressed here is how to use commercially available EDA tools to reach timing closure on a complex synthesized 3DIC. A solution is presented in the form of a primarily manual methodology with complex steps being automated in a variety of ways.

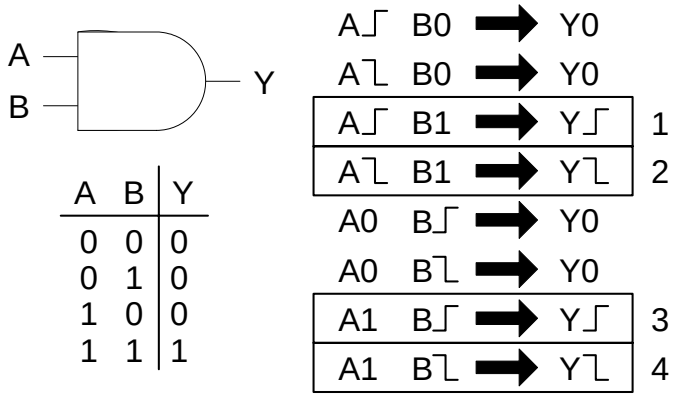


Figure 1-1: The Four Valid Timing Arcs of an AND Gate

We consider a chip to have met timing closure after examining each chip-level timing arc in the entire chip, and ensuring that each arc passes all setup and all hold timing tests. The simplest set of timing arcs is seen in figure 1-1. Any two-input, one-output gate has the potential to have 8 timing arcs as shown. A timing arc is a specific path of signal propagation from a transitioning input to a transitioning output while all other inputs are set to constant values. An AND gate, for example, has just four valid timing arcs, since the ones not boxed do not have transitioning outputs. In this chip-level work, the timing arc concept is abstracted to the chip level, where each path can contain multiple gates. In this case the starting point of a timing arc can be either the primary input to a chip or the output of a register, and the ending point of a timing arc can be either the input of a downstream register or the primary output of a chip. Therefore, in order to close timing we must check each primary input to register, register to register, and register to primary output path for timing

violations. This process will account for clock skew and assume a certain frequency of operation.

The timing verification methodology for a 2D chip is well known and available EDA tools are well-equipped to handle such a task. For single clock designs with no clock-gating, cycle stealing or other timing tricks, the verification process is nearly pushbutton and is mostly handled under the covers of complex tools. In essence, RTL for a design is often coded in a hardware development language, synthesized into a gate-level netlist, then placed and routed using a physical design tool. Once the design is placed and routed, we can extract parasitic resistive and capacitive (and at times inductive) values for all interconnections between gates. By combining this information with standard cell library files, a static timing analysis program can efficiently determine delays for each timing arc and determine if it meets timing. Timing tools can efficiently be used in a variety of ways and can even be run in batches to make a large number of calculations through the use of custom scripts. The amount of timing data that can be efficiently obtained using static timing tools is enormous when compared to circuit simulation based tools such as Spice, at the expense of a small degree of accuracy.

There are a number of difficulties that a designer faces in this timing verification process when beginning to use a new technology, especially a 3D process. First of all, designing the chip is more complex. Our research group has formalized the 3DIC design flow; the major difficulty being that 3DIC's have not yet been addressed by vendor tools. We have created a design and verification methodology that resembles a 2D design flow to such an extent that existing tools can be used. My contribution has been the clock tree insertion and timing verification methodology that will be presented here. There are some

very important modifications to a 2D flow, mostly revolving around the fact that all tools have to treat each tier as an unrelated chip. This creates complications in timing verification, because many timing arcs travel between tiers and we need some way to time the design as a whole. That is the main issue being addressed here, is how to reach timing closure in the presence of a 3DIC design flow developed around traditional 2DIC design tools. A novel approach is presented to accomplish this by operating on intermediate files to make a 3D stack of interconnected dies appear as if it is a normal 2D chip to the design tools. Timing is analyzed using two independent approaches, circuit simulation and static timing.

First we will discuss the 3DIC technology and examine the 3DIC design methodology and 3DIC timing verification methodology on a high level, from a user's perspective. For simplicity we will discuss the design flow mainly as it pertains to the inputs to the timing verification flow, and present a novel approach to optimizing the clock tree. In designing the clock tree, we will examine how a specific 3D configuration can optimize the clock tree for power consumption and delay uncertainty. Next we will dive into the timing verification methodology again, but in deeper detail; all along the differences and similarities between the proposed flow and the standard 2D flow will be addressed. Chapter 4 then serves to validate the flow's accuracy; there will be a discussion of RC parasitics in general and how they were computed in this study. Finally, the results section will provide the reader with data showing how closely the simulation versus static timing data match, and see how this methodology is applied in a practical example. The 3DIC design flow and the 3DIC timing verification flow work were used in the April 2005 submission of our research group's FFT chip using the MIT Lincoln Labs 3D FDSOI technology. The reasons for success will be investigated and why the FFT is a good demonstration of the design flow. In the conclusion the important

points of the 3DIC clock tree insertion and timing verification methodology will be summarized and we will suggest future work to improve the efficiency of 3DIC design.

Chapter 2: 3DIC Design Process

2.1 3DIC Technology

It is important to go into some detail about a typical 3DIC fabrication process. The design methodologies in the following discussions were created such that they would be portable to any 3D process available with little or no modification. However, they were designed with the MIT Lincoln Labs 180 nm FDSOI 3D process in mind because that is currently the only set of 3DIC design rules available to universities, which we used to fabricate the test chip to be discussed later. The essentials of this process are outlined by a cross-sectional diagram in figure 2-1.

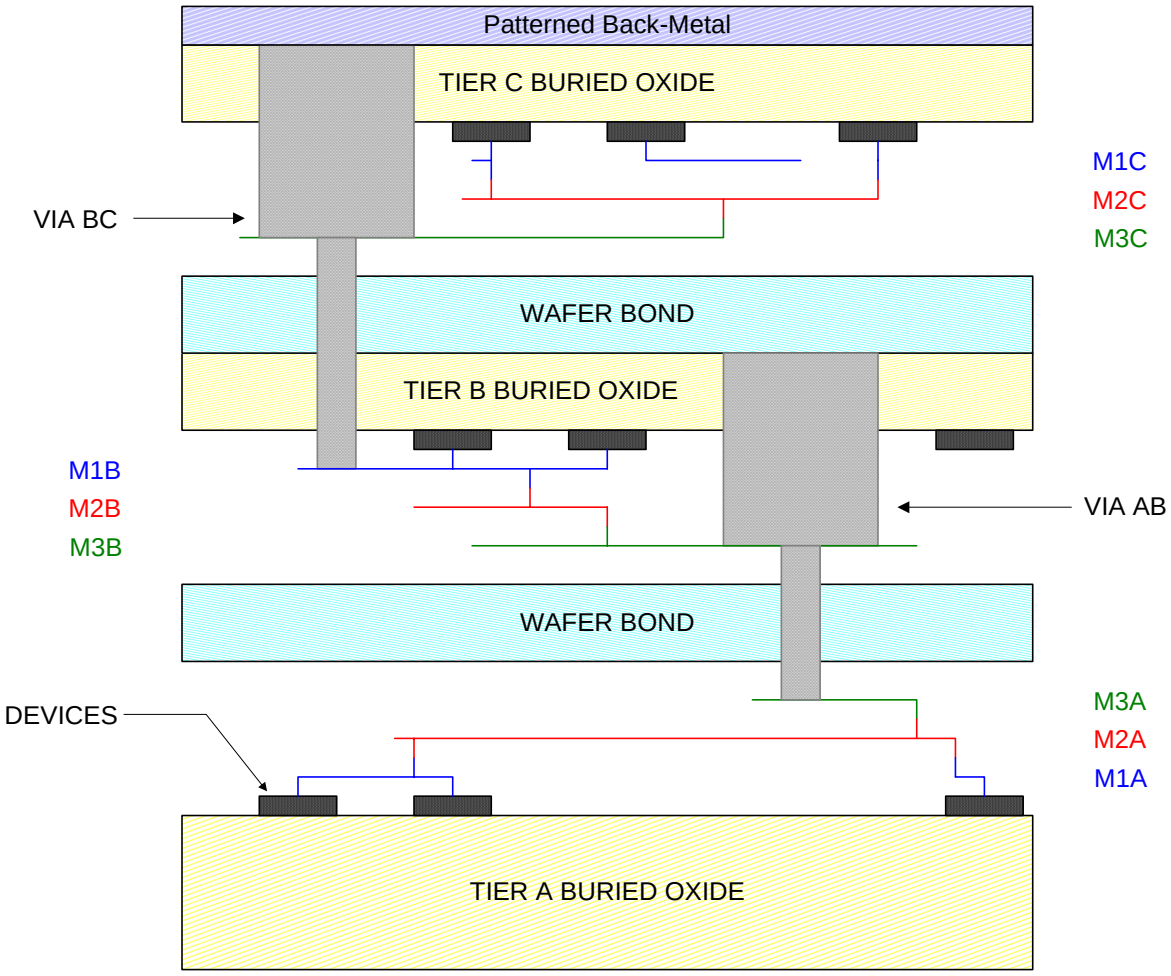


Figure 2-1: Cross Section of a 3D Process

The MITLL process uses three tiers, with three metal layers on each. It uses two different types of inter-tier vias, a VIA AB and a VIA BC. VIA AB electrically connects metal layer 3 in tier A to metal layer 3 in tier B. VIA BC electrically connects metal layer 3 in tier C with metal layer 1 in tier B. After all tiers have been fabricated, first the bottom tier, tier A, is laid down. The next tier, tier B is then inverted so the devices are face down. This is a SOI process, so all transistors exist on islands of silicon in a buried oxide. The buried oxide is then ground thin, essentially to transparency. Using fabrication equipment the tier B wafer is aligned and bonded on top of the tier A wafer. Then the locations for the instances of VIA AB are identified, etched, and filled. After that, the same process is performed for tier C; it is inverted, ground, aligned, bonded, and all VIA BC's are created. On the back of the highest tier is a back-metal layer. This serves primarily to contact ground vias for heat dissipation, but that is beyond the scope of this paper [2].

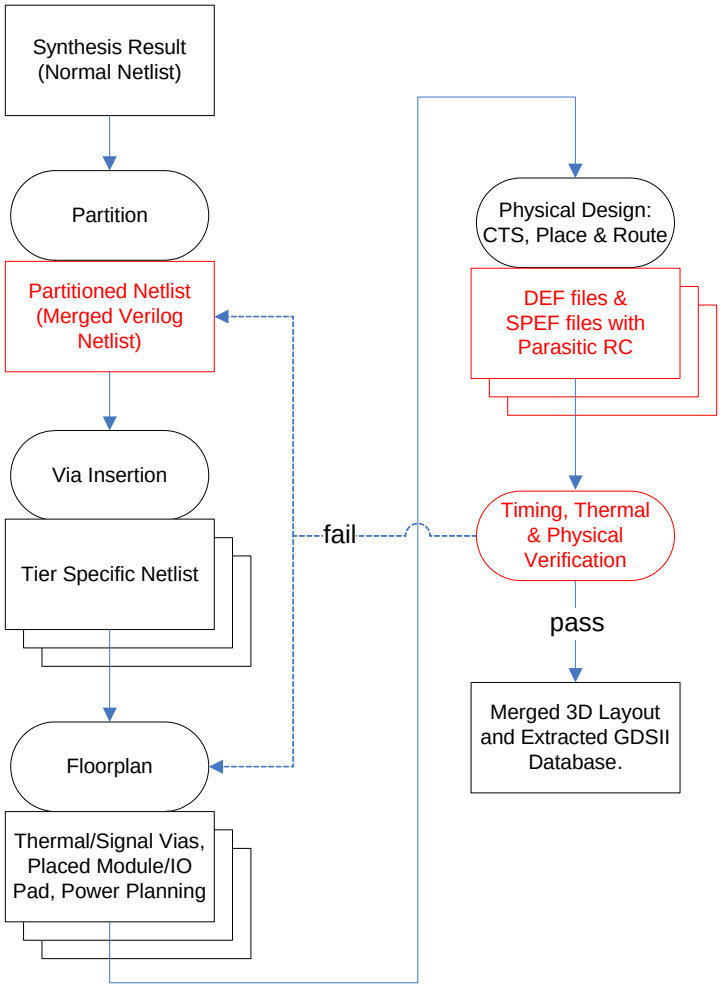


Figure 2-2: 3DIC Design and Verification Flow

2.2 3DIC Design and Verification Flow

2.2.1 Inputs to the 3DIC Design and Verification Flow

The 3DIC design and verification flow is a complex methodology, so at this point we will outline the important details to understand the complications that a 3D technology causes, specifically making note of the information we have to work with when verifying that the chip meets timing. The overall 3DIC physical design flow developed by our research group is shown at the highest level in figure 2-2. The timing verification methodology was designed to prototype 3DIC's in the context of this design flow, so we can start by looking at the 3DIC timing and verification flow at a high level. Throughout this document, a number

of flow charts will be referred to for clarity. Steps shown in red are of particular importance and will be discussed again in detail as we continue. Steps shown in blue are complex steps that have been automated through a variety of scripting languages. We represent user action with an oval and important files as rectangles. Figure 2-2 shows the timing verification steps in red. This methodology begins with completed and logically verified RTL Verilog code that has been synthesized successfully into a Verilog netlist, just as would be done when creating a 2D chip. This Verilog netlist is the "Synthesis Result" shown as the starting point in the figure.

2.2.2 Partitioning

Now that we have a netlist for the entire design, we must decide which modules and which gates will be placed on which tier. This process is shown as the "Partition" step. The design is partitioned into tiers such that the average wire length across the design is minimized. The quantity of vias is also a concern, because the inter-tier vias are rather large and using too many of them results in a waste of chip area. We can see this in figure 2-3, a 3D rendering to scale of a chain of three CMOS inverters. The NMOS are shown in green and the PMOS are shown in brown. The large orange and yellow structures contacted by the grey metal 3 strips are the inter-tier vias. The input to the chain is on the bottom tier, and the output on the highest tier. The partitioning step is largely automated with python scripts, which write out an important file hereafter referred to as the *merged Verilog netlist*. This file is shown in red in figure 2-2 because we use it as an input to the timing verification flow. This file is written out for both verification purposes and use in other design steps. It describes the entire design, but is broken up into hierarchy showing which gates are on which tier. The *merged Verilog netlist* is used in the next stage, "Via Insertion."

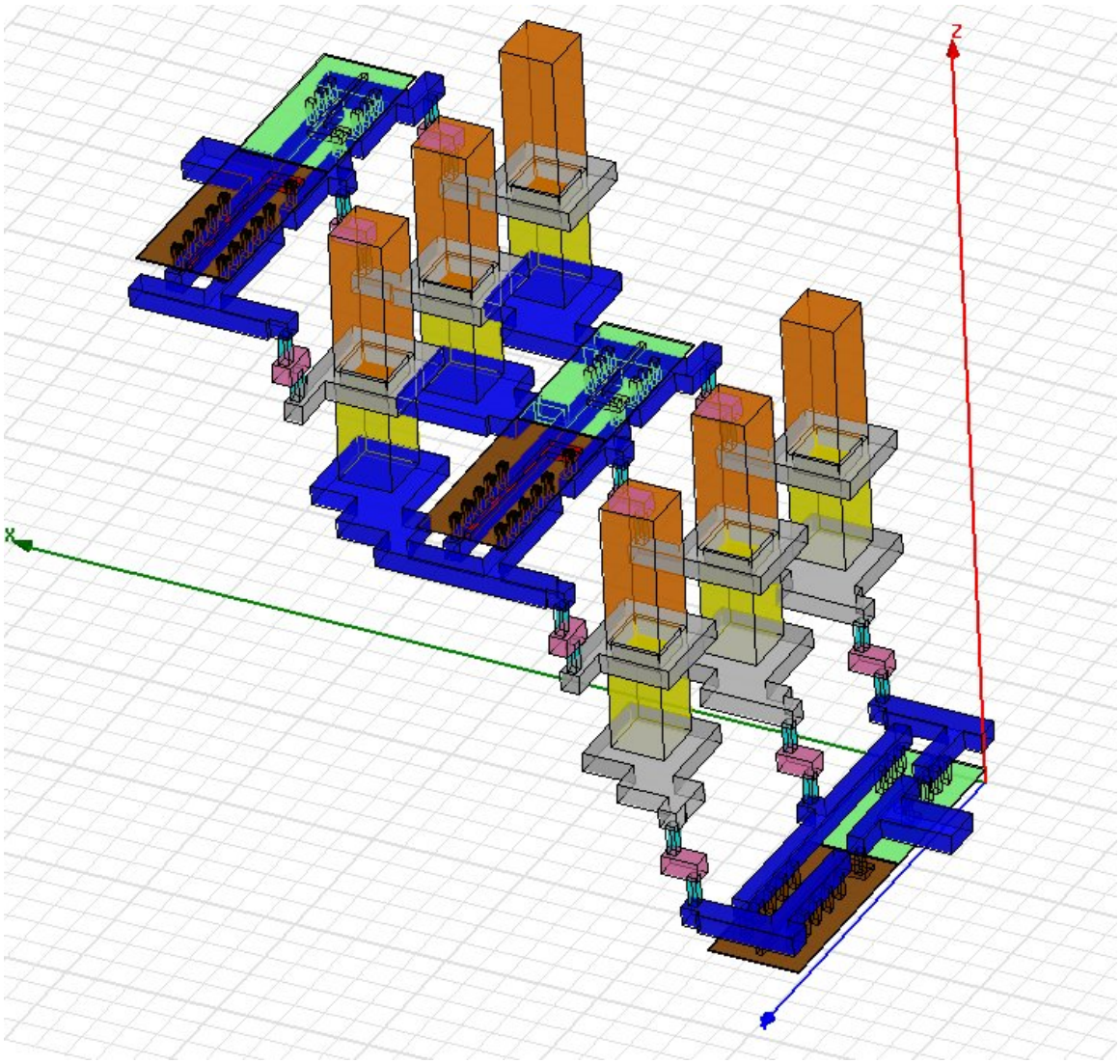


Figure 2-3: 3D Chain of Three Inverters

2.2.3 Via Insertion

In the "Via Insertion" stage, inter-tier via cells are instantiated in the design wherever there is a connection between any two tiers designated by the newly added hierarchy. Simultaneously, the single netlist is broken up into separate netlists, one file for each tier of the design. Figure 2-4 shows the details of an inter-tier via and what is involved with inserting one. The fabrication information for each tier must be submitted to the foundry separately, which means information for any particular inter-tier via must be in two places. An inter-tier via is comprised of an instance of the VIA Cut cell on the upper tier, and the

VIA Land cell on the lower tier. A VIA Cut consists of a via layer surrounded by and partially overlapping a ring of metal. The electrical connection is made by the via going through the metal ring. The VIA Land consists of a landing pad of metal and a via layer coming down to meet it; each has associated design rules. The specifics of inter-tier via creation are irrelevant here, but it is important to recognize that in order to fabricate one 3D via there must exist an instance of VIA Cut in the layout for one tier, and in the tier immediately below it, in the same location, an instance of VIA Land. This means that in the "Via Insertion" step, we must split the merged Verilog netlist into separate files, and in order to instantiate a via, one VIA Cut must be instantiated in one file and a corresponding VIA Land in another file.

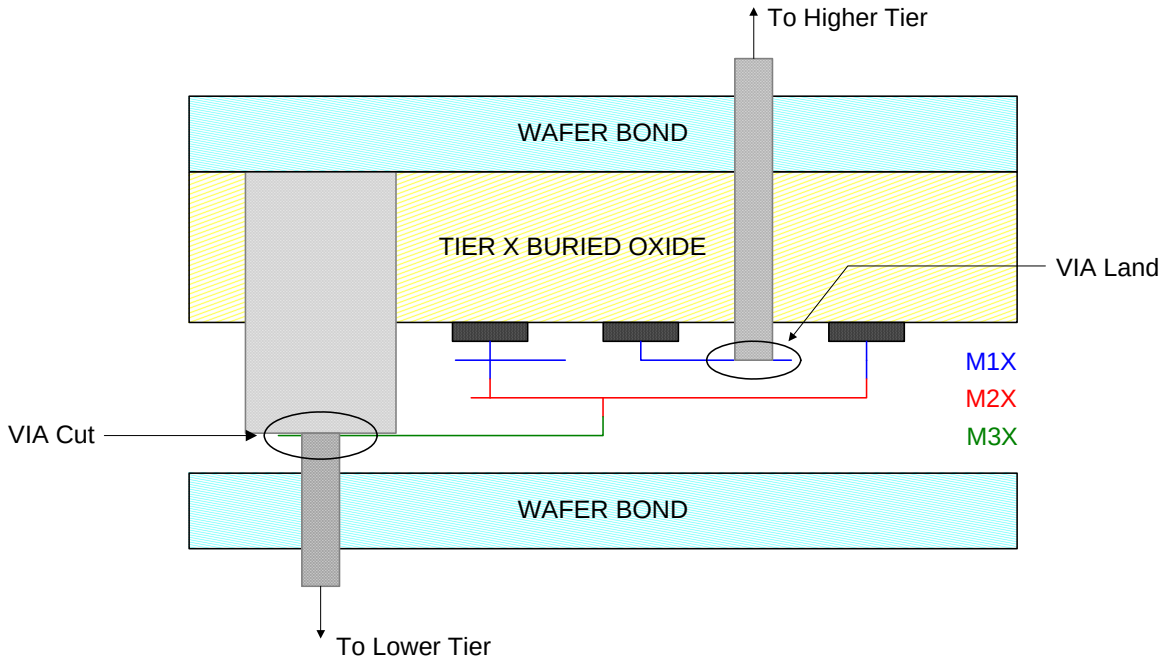


Figure 2-4: Cross Section of a 3D Via

2.2.4 Floorplanning and Physical Design

Referring back to figure 2-2, we can continue with the "Floorplan" stage. This step is mainly manual and has similar goals to 2D floorplanning, except that it is obviously more

complicated because in three dimensions there are more options. We also have the added task of inserting inter-tier vias that exist for the specific purpose of channeling heat out of this stack of insulators. After the "Floorplan" step we are ready to let the EDA tools work with the design more. There now exists a separate netlist for each tier of the chip, complete with floorplanning decisions. Even the VIA Land and VIA Cut cells that have been instantiated are treated by the 2D design tools the same as any other instantiated cell with ports and abstracted circuit elements inside. Therefore, a physical design tool such as Cadence's First Encounter can handle the exact placement and routing of the cells in each tier-specific netlist individually. In each tier we first place the cells, and then use First Encounter's built in clock tree synthesis tool to insert clock tree buffers into the layout. First Encounter then routes each tier individually, and we can manually connect the clock tree from each tier together. Upon completion, First Encounter writes out a set of important files for each tier. For each tier a design exchange format (DEF) file and a standard parasitics exchange format (SPEF) file is created. The DEF file is a type of netlist that includes connectivity and floorplanning information; it completely describes the layout of the tier. The SPEF file is also a netlist, but describes each segment of interconnect in detail. It gives all resistive and capacitive values associated with each net. A complete description of a net in a SPEF file lists each pin on each standard cell to which the net is connected. Then, it breaks each straight jog of metal in the net into an RC Pi model. Each time a path turns at 90° a new segment with a Pi model is listed, whether the path is changing directions in the same metal layer or turning out of its plane through a via. There is an option in First Encounter to have the SPEF file include coupling capacitors separately in the parasitics, which is recommended at times as will be discussed later. At this point the actual design

work is complete, and what remains are several verification flows and intricacies to prepare the design for submission to the foundry. We will next explore the verification step, which includes the 3DIC timing verification methodology.

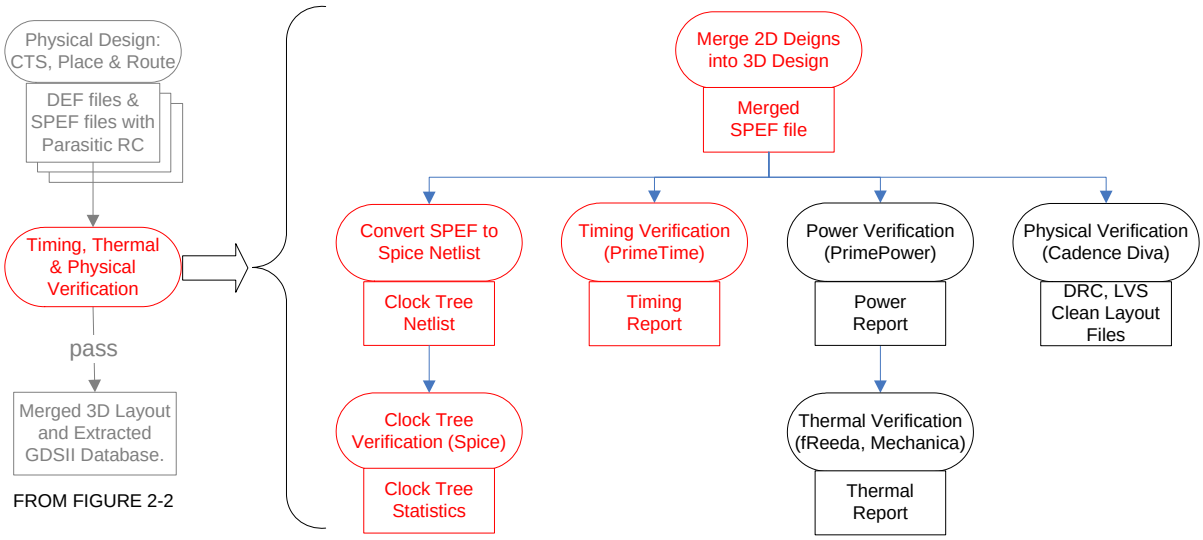


Figure 2-5: 3DIC Verification Flow

2.2.5 Verification

Figure 2-5 gives an overview of the verification methodology for a 3DIC. We can start with the physical verification steps. Design rule checking (DRC) is fairly similar to that of a 2D chip; when the DRC deck of design rules is being written it must include all design rules having to do with the insertion of inter-tier vias, in addition to those having to do with device construction and wiring for the specific process. Layout versus schematic (LVS) checking is slightly more complicated because we need to physically compare the preliminary schematic of the design with the completed layout to make sure there are no short circuits or other mistakes that would result in the layout and schematic not having all identical devices and connections. This means that we have to make the 3D chip appear as a 2D design to the LVS tool. This is done by again using the *merged Verilog netlist* from figure 2-2. This netlist that uses hierarchy to represent the partitioning of the 3D design in a

2D netlist is critical to both the LVS step mentioned here and as a starting point for the timing verification flow. The common idea behind the physical verification methodology is that we have to analyze the design as a whole, not the tiers individually. In order to do this we must make the complete 3D design appear as a 2D design to the EDA tools. We apply this idea to the timing verification as well.

In figure 2-5, we can now look at the parts shown in red, which specifically represent the timing verification flow. The power and thermal verification are handled by separately using other Synopsys tools and tools developed within the university, which is outside the scope of this paper. The timing verification is performed in two somewhat independent ways in order to validate each other, as can be seen in figure 2-6. One way is by obtaining clock tree data such as skew, slew rate, and insertion delay through Spice circuit simulations. The same tasks that were performed by the simulations are then performed using static timing analysis with Synopsys' PrimeTime tool. We then use PrimeTime to time the design as a whole to check for timing violations; paths of particular interest can then be reanalyzed using Spice at the user's discretion. Looking again at figure 2-6, we start with the SPEF files that we have extracted for each tier. Through the use of a custom Perl script, *spefmerger*, we can combine these files into one SPEF file such that it can be used to annotate the *merged Verilog netlist* with parasitics. We use this *merged SPEF file* to perform a number of tasks. First, a custom Perl script, *spef2spice*, has been written specifically to convert a SPEF file into a Spice netlist. This way we can perform circuit simulations of interesting or important parts of a design. Having a hierarchical Verilog netlist and a compatible SPEF file allows us to use static timing to time however much of the design we desire. In this verification flow, we use it to find vital clock tree performance statistics and eventually to verify timing across

the whole design. It is the steps involved with this timing verification on which we focus in chapter 3, by explaining figures 3-1 and 3-2 in detail.

The objective is to design a 3D chip using commercially available 2D EDA tools as much as possible, and deviating from the 2D complex chip design flow as little as possible. The idea is to produce the simplest method possible to successfully create a 3DIC immediately. The design and verification processes for a 3D chip has now been summarized, but before proceeding to the verification portion it is necessary to review how timing closure is met for a 2D chip. The process is normally totally completed using a static timing analysis tool. One complete Verilog netlist exists, and after placement and routing, there exists a SPEF file with which the Verilog netlist can be back-annotated to include parasitics. The clock tree is inserted using a physical design tool; some timing constraints are given to the tool and it does its best to produce a clock tree that will meet the specification. The static timer is fed some technology information about metal layers and parasitics, and files that describe the timing properties of each cell in the standard cell library. All of this information can be included in files adhering to the Synopsys Liberty (LIB) and Open Library Exchange Format (LEF) standards. A member of a timing team then reads into the timing tool a LIB file and a LEF file, a Verilog netlist, and the corresponding SPEF file. After that, more timing constraints will need to be set, the complexity of which often varies with the complexity of the design being timed. The user would typically specify the periods and duty cycles of all clocks, arrival times of valid signals at primary inputs, and any special situations such as false paths at the very least. In general the number of clocks in the design and how often they interact with each other dictates how much effort setting up these assertions requires. For most designs, min and max timing reports can then be generated for the design;

they give the specified number of worst slack setup and worst slack hold timing arcs. If the worst slack is positive, then timing is generally considered closed. Timing tools usually have the option to receive their commands in the context of a language such as TCL (tool command language), so if any more specific or other data besides worst slack paths is desired it can be calculated and post-processed. The largest complication between this method and what is available in a 3D flow is that this 2D flow needs to be modified to accommodate the fact that the only available parasitic information is in three separate files. The next chapter will now go into the details of the 3D timing verification flow.

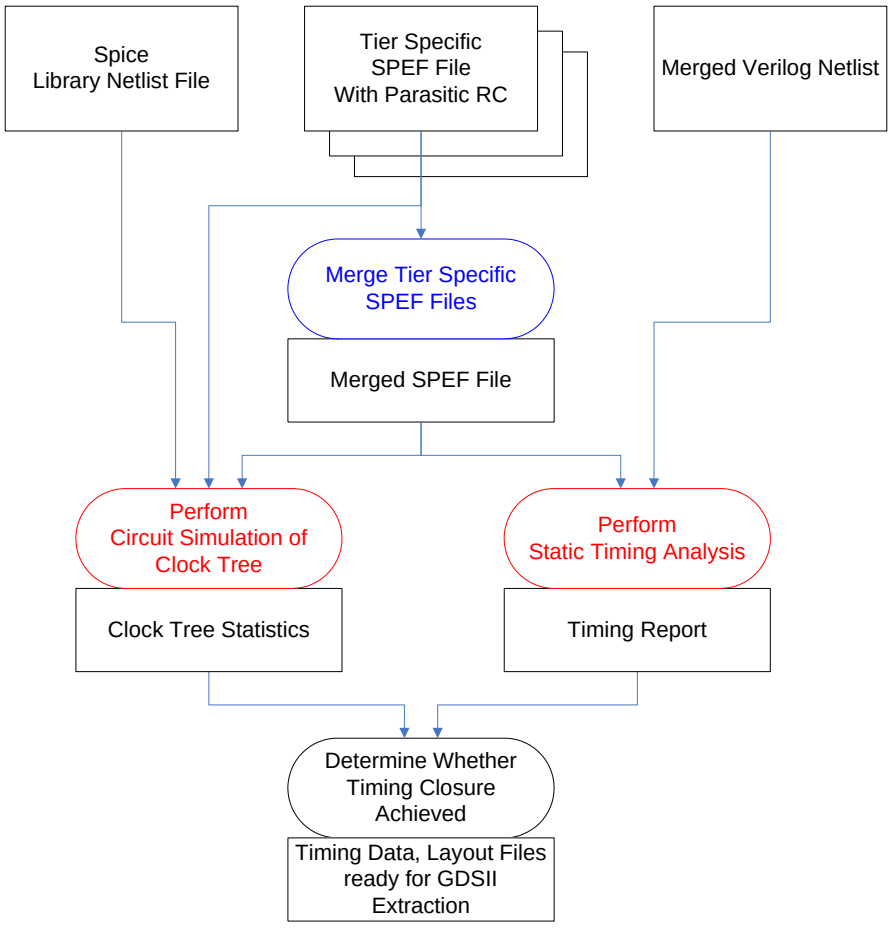


Figure 2-6: 3DIC Timing Verification Flow

2.3 Clock Tree Insertion

We have outlined how to verify a clock tree that has been routed to multiple tiers, and it is important to discuss how to build the optimal clock tree. Depending on application, we will consider optimal to be either the clock tree with the lowest timing uncertainty, or the clock tree which consumes the least power. The FFT clock tree was designed using First Encounter's clock tree synthesis tool almost entirely, but if it was necessary some custom design work could have improved its performance; it makes a good example for the following analysis. For the FFT a timing constraint file imposed some requirements on the tool which it attempted to adhere to while inserting clock buffers and routing the clock tree. The clock tree insertion tool automatically routes the clock tree to a pad on the perimeter of the chip, so by inserting a clock tree automatically on each tier and forcing the clock pad to be in the same spot on each tier, we can easily manually connect the signal on the tiers and have a satisfactory clock tree. This method requires little designer effort, and as will be shown, is not a bad one.

Many of the issues a designer faces when creating a 3D clock tree are the same as the ones faced when inserting a 2D clock tree. For this reason, we will make some assumptions that force us to concentrate on the optimizations that apply just to the 3D aspect of clock tree synthesis. We will use the number of clock sinks in the FFT design and the dimensions of each tier; we will assume the clock sinks are distributed uniformly across three tiers, and are evenly spaced within each tier. Modern clock tree synthesis tools generally use some form of an H-tree clock tree, so we will also assume an H-tree in order to analyze a tool-generated clock tree. To distribute the clock we will assume an ideal zero-skew H-tree of eight levels; an eight level H-tree in the described situation means that each last stage buffer has a fan-out of 5 - 6. This is adequate, although an additional level would sometimes be desired because

optimal designs generally have a fan-out of 4. This clock tree is shown in figure 2-7, and we will assume that it is replicated in each of the tiers. The different levels of the clock tree are shown in different colors, starting in dark green and fading to dark blue then black. This leaves the question, of when we should branch into the third dimension. At this time we will discuss both how to buffer the clock tree and where to place the inter-tier vias based on the most critical objective.

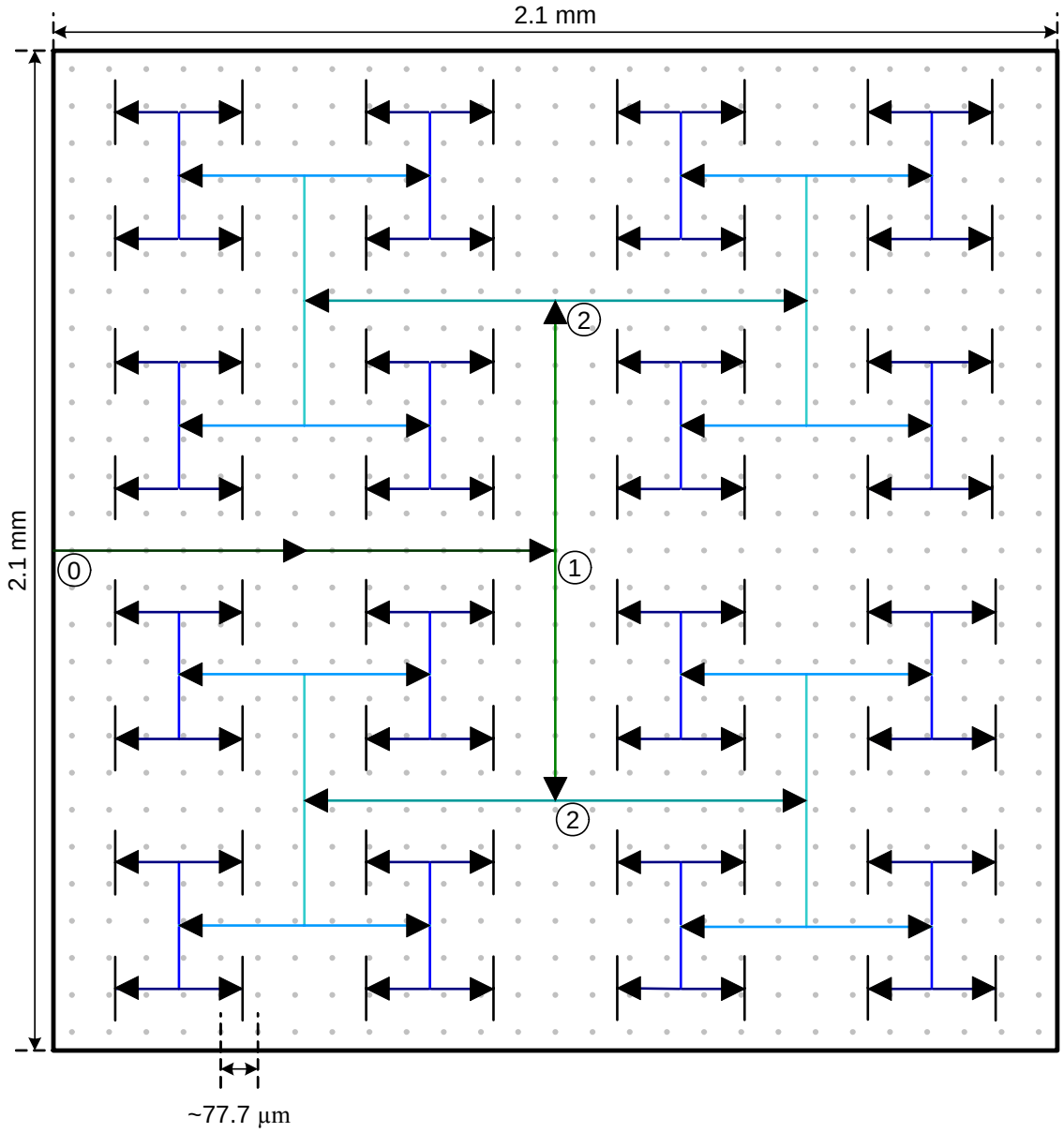


Figure 2-7: Ideal Zero-Skew 8 Level H-Clock Tree with Buffers

We will attempt to optimize the clock tree based on timing uncertainty and power consumption by placing the inter-tier vias in an optimal location. For this analysis we can say that we have the choice of branching to 3D at point 0 (at the perimeter), point 1 (after level 1—between the dark green and green routes), point 2 (after level 2—between the green and turquoise routes), and so on, as labeled in figure 2-7. In this clock tree, we have 9 choices of where to branch, points 0-8 although only 0-2 are labeled for simplicity. In this decision there will be trade-offs because different branch points require different lengths of wire and different quantities of inter-tier vias. For example, if we choose to branch at point 2 then we say levels 1 and 2 (the dark green and green routes) will only exist in the center tier but all other routes exist in all tiers. However, if this was the case then we need 4 inter-tier vias to distribute the clock, one going up and one going down at each point labeled 2. Similarly, if we branch at point 0 then the entire clock tree is replicated on each tier but we need only 2 inter-tier vias. However, first we will address another issue concerning wire length, how to insert the buffers shown in figure 2-7. Too many buffers will result in a drastic waste of power, but too few results in long insertion delays and large slew rates.

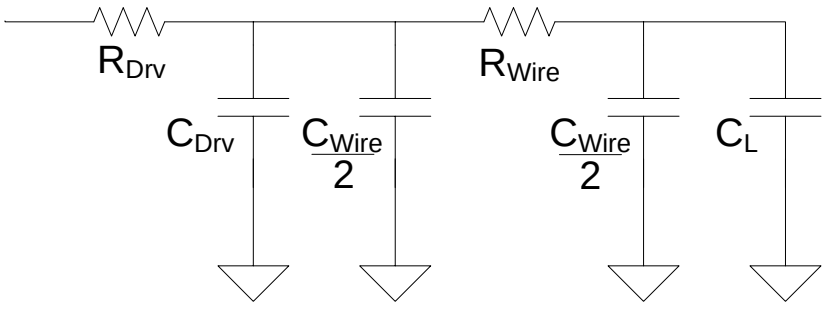


Figure 2-8: Pi Model of Interconnect with Driver and Load

In general we can determine the optimal length of interconnect between repeaters starting with an analysis that uses the Elmore Delay to model a segment of interconnect. The

line is represented as a Pi model with capacitance C_{Wire} and resistance R_{Wire}. The line is driven by a gate with output resistance R_{Drv} and output capacitance C_{Drv}. The line is loaded with another gate, the input capacitance of which is seen by the line as C_L. According to the Elmore Delay, the propagation delay along this RC network is $t_d = 0.69 \cdot \tau$, where τ is the sum of RC time constants along the path of interest as dictated by this timing approximation. This model is shown in figure 2-8, and the calculation of τ for this network is:

$$\tau = R_{Drv}\left(C_{Drv} + \frac{C_{Wire}}{2}\right) + \left(R_{Drv} + R_{Wire}\right)\left(\frac{C_{Wire}}{2} + C_L\right) \quad (1)$$

Certain substitutions can be made so that τ is a function of more general process parameters. The substitutions include:

$R_{Drv} = \frac{r_d}{s}$	r_d = driver resistance (Ω·μm)
$C_{Drv} = 3c_o s$	c_o = driver output capacitance (fF/μm)
$C_L = 3c_g s$	c_g = load gate capacitance (fF/μm)
$R_{Wire} = r_w L$	r_w = wire resistance (Ω/μm)
$C_{Wire} = c_w L$	c_w = wire capacitance (fF/μm)

The driver resistance is for a micron of transistor width, assuming the resistance is due to the one device in the gate that is on during a given transition. The *s* is microns of transistor width, assuming minimum length, so R_{Drv} is inversely proportional to device width. The driver resistance can be found via a circuit simulation of an inverter. From the V_{DS} vs. I_D curve we can find the instantaneous output resistance when $V_{DS} = V_{DD}$ and when $V_{DS} = V_{DD}/2$ and average the two. The driver output capacitance is directly proportional to device width, and is dependent on the drains of all devices. We will assume that PMOS devices will be sized twice as large as NMOS, which is where the coefficient of 3 comes from. The output capacitance is also derived from circuit simulation; a single device (we will say NMOS) can be connected with a gate voltage to bias it in the correct region of operation and

an ideal current source forcing current into the drain while the source is grounded. The simulation can measure V_{DS} and subtract I_{DS} from the ideal current source. This gives enough information to use $I = C\frac{dV}{dt}$ to find the capacitance. The gate capacitance is found in a similar way to the driver output capacitance, except by grounding the source and drain to keep the device in cutoff while forcing current into the gate. The parasitic resistance and capacitance of the wire vary with *L*, the length of wire being considered. The values are generally obtained from the foundry-provided information about the technology. The resistance of each metal layer is expressed as a sheet resistance in Ω/□. The capacitance can be found by using the provided area and fringing capacitances. We assume minimum width wires. Making the described substitutions into (1) and performing some expansion yields (2):

$$\tau = k\left[\frac{r_d}{s}\left(3c_o s + \frac{1}{2}c_w\frac{L}{k}\right) + \frac{r_d}{s}\left(\frac{1}{2}c_w\frac{L}{k} + 3c_g s\right) + r_w\frac{L}{k}\left(\frac{1}{2}c_w\frac{L}{k} + 3c_g s\right)\right] \quad (2)$$

The only difference, however, is that we substitute in $\frac{L}{k}$ instead of just *L*, where k is the number of repeaters that exist along the length of wire, *L*. This is because we really have a collection of smaller wire segments separated by *k* buffers. We also multiply the entire expression by *k*, which is the equivalent of adding up all of the τ's for the entire line. Minimizing τ will effectively minimize the delay along the line, and *k* is the parameter that we are looking to vary to perform this optimization. Therefore, in (3) we take the partial derivative of τ with respect to *k*, and set the result equal to zero to find the minimum τ.

$$\frac{\partial\tau}{\partial k} = 3r_d\left(c_o + c_g\right) - \frac{1}{2}r_w c_w\frac{L^2}{k^2} = 0 \quad (3)$$

In (4) the expression is rearranged into the form that gives the optimal number of repeaters to insert in a length of wire, *L*.

$$\left(\frac{L}{k}\right)_{optimal} = \sqrt{\frac{6r_d\left(c_o + c_g\right)}{r_w c_w}} \qquad (4)$$

It is important to note that in taking the derivative, *s* dropped out of the relationship. The repeaters inserted will have to be sized considering delay, power, and slew rate, yet the number of buffers that should be inserted is dependent upon process parameters and the length of wire being buffered. This value should be taken as a guide when inserting buffers, but cannot be followed literally first because the number of buffers to insert is very discrete, but also because nets have different branches and fan-outs. Nevertheless, this provides a good starting point, one that was used in figure 2-7.

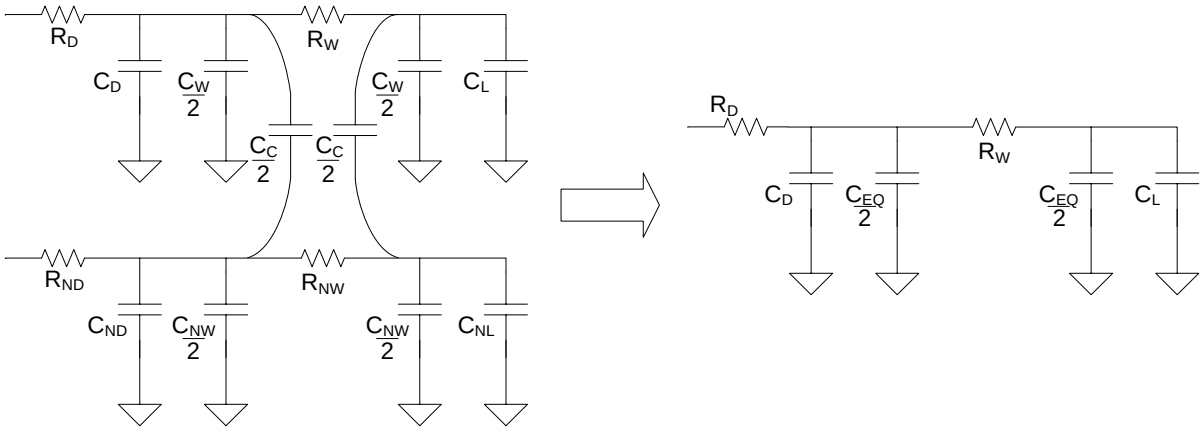


Figure 2-9: Pi Model of Interconnect with Coupling Capacitors

The simple model we have used so far, however, is inadequate to proceed with. For the reasons to be described in the RC extraction chapter, we need to consider coupling capacitances to decide when to split to 3D. When a wire has a close neighbor, a coupling capacitance exists between them. The activity of the neighbor line can either speed up or slow down the line of interest, based on the slew rate of the neighboring line and whether it is

switching with or against the line of interest. To model this we can convert the coupling capacitance and the wire ground capacitance to one equivalent ground capacitance.

The left side of figure 2-9 shows a segment of interconnect similar to figure 2-8, but with the coupling capacitances connecting it to a neighboring net with parasitics marked with an 'N.' The wire capacitance and the coupling capacitance from the figure are combined into C_{EQ} according to (5).

$$C_{EQ} = C_W + (1-\beta)C_C, \text{ and } \beta = \frac{\Delta V_N}{V_{DD}/2} \qquad (5)$$

In (5) $\Delta V_N = V_{N2} - V_{N1}$ if V_{N1} is the voltage of the neighboring net when the net of interest begins to transition and V_{N2} is the voltage on the neighboring net when the net of interest reaches a switching threshold of $V_{DD}/2$. In this case, the $-1 \le (1-\beta) \le 3$, so when $1-\beta = 3$ C_{EQ} will be largest and the propagation delay will be longest. When $1-\beta = -1$ C_{EQ} will be smallest and the propagation delay will be the fastest [3]. Using (5) we can convert the circuit on the left side of figure 2-9 to the one on the right side for easier analysis. We will write the Elmore Delay equation for the circuit on the right side of figure 2-9 two times, once for when C_{EQ} is at a maximum and once for when it is at a minimum. Subtracting these two values and diving by two will give us the timing uncertainty in a segment of interconnect due to coupling capacitances, assuming we have no knowledge of the activity on the neighboring line.

Equations (6) and (7) show the maximum and minimum propagation delays that can result from the circuit on the right side of figure 2-9.

$$t_d(\max) = 0.69\left[R_D\left(C_D + \frac{C_W + 3C_C}{2}\right) + (R_D + R_W)\left(\frac{C_W + 3C_C}{2} + C_L\right)\right] \qquad (6)$$

$$t_d(\min) = 0.69\left[R_D\left(C_D + \frac{C_W - C_C}{2}\right) + \left(R_D + R_W\right)\left(\frac{C_W - C_C}{2} + C_L\right)\right] \quad (7)$$

All of the variables in (6) and (7) are calculated as described earlier, with the exception of C_C which was determined using an electromagnetic fields solver and assumed to vary directly with wire length. However, that would be assuming that there is a neighboring net coupling to the net of interest for its entire length. This is overly pessimistic so steps can be taken to calculate the routing congestion of the design. The routing congestion calculated as an percentage of occupied g-cell tracks after detail routing should be used as a multiplier to the simulated coupling capacitance. From this information we can calculate the timing uncertainty due to coupling capacitance in (8).

$$t_{uncertainty} = \pm\frac{t_d(\max) - t_d(\min)}{2} \quad (8)$$

The parasitic data for a 3D via is available from the simulations to be discussed in chapter 4. Using this we can model a 3D via the same was as we modeled the interconnect in figure 2-9, and also calculate the timing uncertainty introduced by each inter-tier via using this method. Because splitting the clock tree and distributing it to 3D at different points results in different wire lengths and different numbers of inter-tier vias, splitting to 3D at different points results in different amounts of total timing uncertainty in the clock tree. We can add together all of the uncertainties to find a total timing uncertainty, and this would be a good value to optimize, but we cannot take this value too literally. First of all, accurate values of r_d, c_o, c_g, r_w, c_w, and c_c are difficult to obtain. Next, there is not necessarily a register-register data path between each set of storage elements. Furthermore, even if there is a register-register path coupling capacitance is likely to affect it, but it is an extreme upper bound to consider all of the uncertainties to be additive. However, from gathering this data and finding the total

timing uncertainty when branching to 3D at each possible point, we can draw some important conclusions.

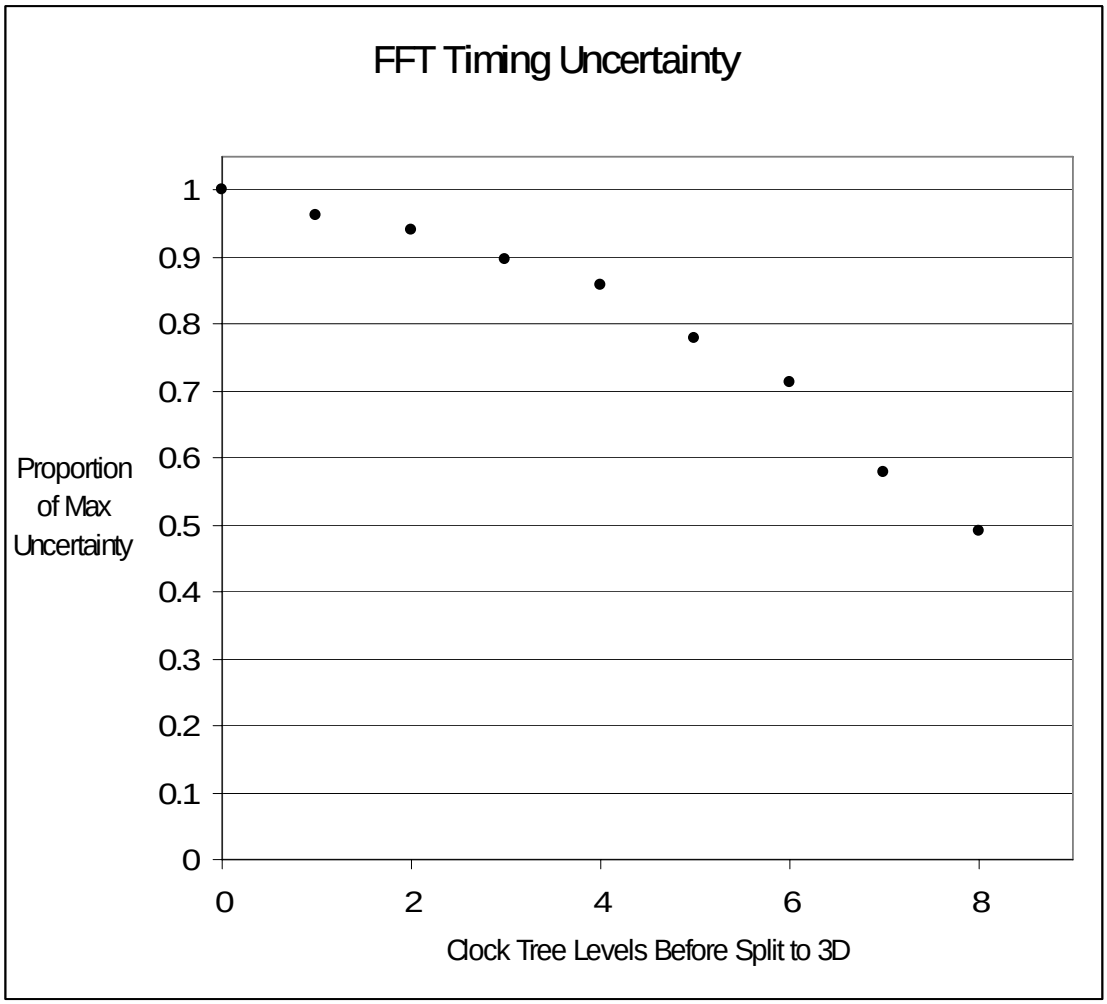


Figure 2-10: FFT Timing Uncertainty Plot

Using the described method to calculate timing uncertainty due to coupling capacitance, we can obtain the data in figure 2-10. Since the timing uncertainty due to coupling capacitance on wiring is much greater than that due to coupling capacitance on 3D vias, the graph shows that in our case it is most beneficial to split to three dimensions as close to the clock sinks as possible. The most useful information to draw from figure 2-10 is that there is no real benefit to wait just one or two levels before splitting, doing so would

reduce the uncertainty by less than 10%. If the clock tree is split at any place other than at the perimeter, point 0, significant designer effort will be required. This analysis shows that there is significant benefit to customizing the clock tree, but a lot of designer effort would likely be required to achieve this benefit. In addition to the designer effort, the landing pad for a 3D via is 25 μm^2, so there could be very significant area increases; splitting at point 8 adds 256 3D vias to the design. The decision of where to split to 3D depends on the amount of slack in the timing and area budgets, and the amount of designer effort that can be afforded. We will now also examine how the power consumption varies with the split point.

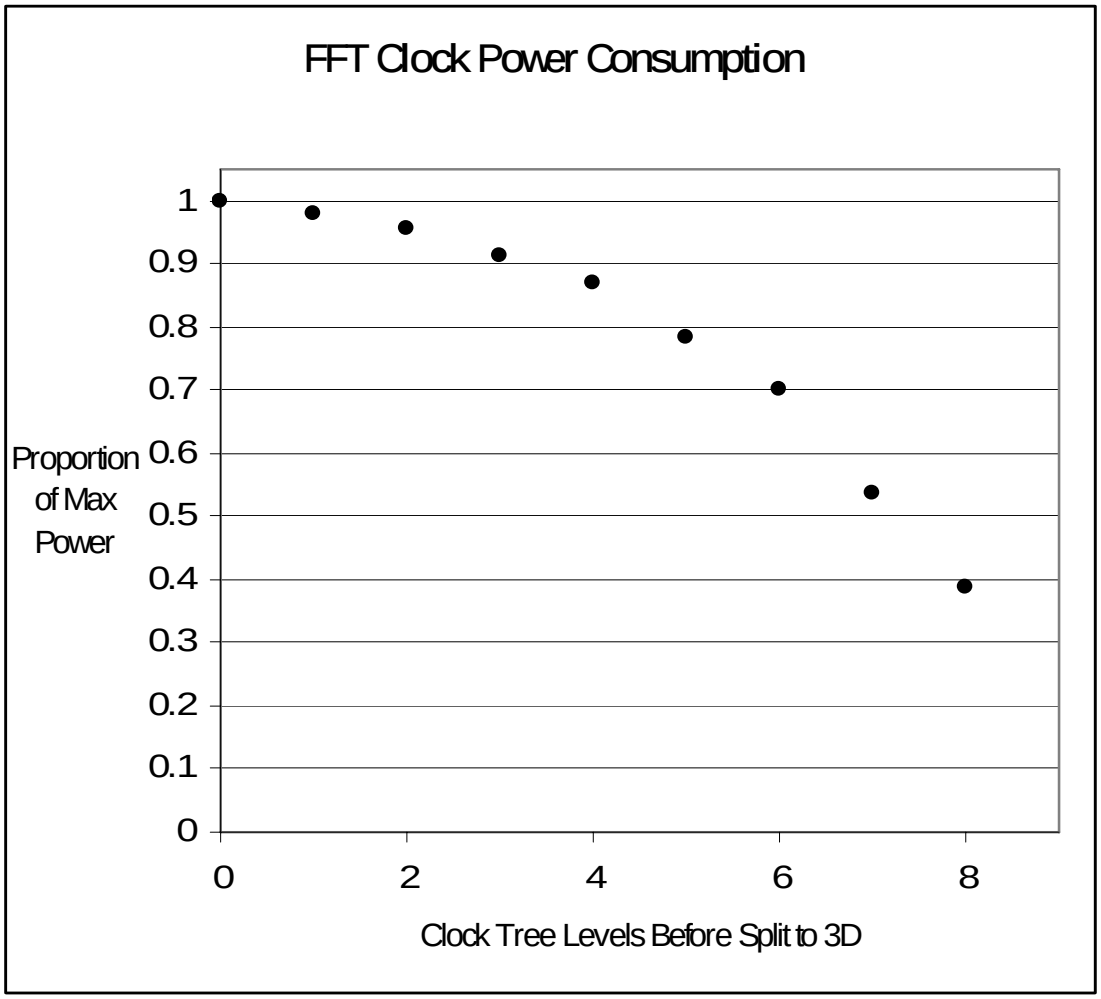


Figure 2-11: FFT Clock Power Consumption Plot

The clock net is one of the most power hungry nets in a chip, because it is typically very large and switches on every cycle. The equation to calculate the power consumed by a net is $P = \frac{1}{2} V_{DD}^{\ 2} f C \alpha_{0\to1}$. If we are trying to optimize where to split to 3D, we cannot control the supply voltage, operating frequency, or probability of a 0→1 transition. Therefore, we control the capacitance on the net only, so the power consumed is directly proportional to the total capacitance on the net. By the same rationale as the timing uncertainty calculations, we can calculate the total power consumed by the clock tree for each of the possible locations we could choose to split to 3D. Such a plot is shown in figure 2-11. We see that branching into three dimensions close to the clock sinks also helps to conserve power, again because the 3D vias have less capacitance than the interconnect. There is also a similar trend regarding how much designer effort and added area is needed to reach the power reduction benefit.

Chapter 3: 3DIC Timing Verification Methodology

The design flow laid out so far proves to have been a success because as far as the tools are concerned, we are designing several independent 2D chips. It is through user interaction and processing steps in between EDA tool steps that the seemingly independent chips are made compatible with one another and can be interconnected to form a 3DIC. Like the physical design DRC / LVS verification, we intend to verify timing by treating the design as a whole and making the multi-tier design appear as a 2D chip.

3.1 Inputs to the 3DIC Timing Verification Flow

This section begins the detailed explanation of the timing verification flow shown in figure 2-6. The figure shows the input data and output data, along with the necessary tool executions. Besides the theoretical clock tree analysis, this was my main contribution to the 3DIC design and verification methodology. To begin timing verification, there are five essential inputs that must be prepared. These are shown on the top row of figure 2-6, the high level diagram of the timing verification flow. From the design flow, we need the *tier-specific SPEF files* (there are three in this case), the *merged Verilog netlist*, and the *Spice library netlist file*. The *Spice library netlist file* is normally used only by those working on the standard cell library specifically, and contains a Spice netlist for each cell that is in the design. Continuing with figure 2-6, the first actual step in the timing verification process is to merge the *tier-specific SPEF files*. This step is in blue because it has been automated and will be discussed shortly. With the aforementioned inputs and the *merged SPEF file*, we can then calculate timing data via two independent paths, the static timing analysis path, and a circuit simulation path. We will use the similarity in results between these two methods to validate one another. These two paths are shown in red, again, because we will dive into

each of them in greater detail. After all timing reports and such have been generated, we can determine whether the chip meets timing, or if we need to iterate back through part of the design process.

3.2 SPEF File Merging

Merging the *tier-specific SPEF files* into a single SPEF file is an important task but has been automated with the *spefmerger*. The script can take up to one hour for a design with 140,000 cells, but heavily depends on the complexity of the design and what parasitics exactly are being extracted. To run the code, one must execute the script with four command line arguments. A path to the SPEF file for each tier (three in our case) and a name for the design as a whole must be supplied. When completed, and used to back-annotate the *merged Verilog netlist* with parasitic information, we can once again make a 3D design appear as a 2D design so that we can operate on it with EDA tools. Theoretically, the *spefmerger* is fairly simple. We first take the SPEF file from each tier and concatenate them together section by section into one file. We must take care to change all net and instance names such that there is no name aliasing, and then remove the 3D vias and insert propagate segments in their place to maintain the electrical connection. However, there are a number of complications. For the reasons to be discussed, the *spefmerger* is specific to the MITLL 3D FDSOI technology and the 3DIC design flow developed by our research group. The timing verification flow, however, is structured such that the *spefmerger* can easily be modified or replaced to accommodate a slightly different 3D process or different design flow without upsetting the surrounding steps.

Primarily, the *spefmerger* must follow every single convention assumed by the code already in place that creates the *merged Verilog netlist*. That includes how all names of nets

and instances are changed during floorplanning and merging, because of tier placement, position in a re-powering tree, if connected to a pad cell, etc. Also, there are naming consistency problems that arise between Verilog and SPEF files which are normally taken care of under the covers of other tools but become our responsibility here. There are certain characters that sometimes must be escaped in each type of file because they have special meanings, and hierarchy delimiting characters in one type of file can create an illegal net name in the other. There is quite a bit of translation in the names alone to associate the parasitics with the correct net.

We also must be selective about which nets from the individual tiers we include in the *merged SPEF file* for fear of duplicating nets, such as those connected to pins that were instantiated for the sole purpose of creating re-powering trees. Also having to do with name mapping and aliasing problems is the way that the SPEF file refers to nets and instances. At the top of the file there is the "NAME_MAP" section. Here every instance and net in the design is given a number; from that point on the net or instance is referred to by that number preceded by an asterisk (*). Therefore, we need to also avoid net/instance number aliasing when merging the files. Also, when the parasitics of a net are described, that one net number is broken down into many subnets. For example, net *1 may be broken down into *1:1, *1:2, *1:3, etc., all of which are connected by different resistances to maintain the same electrical connections as *1, and all of which could have different capacitances. If an inter-tier via connects net *1 in tier A with, for example, net *5 in tier B, we would have first offset *5 in tier B so that the net is referred to as *(5+x) where x is the total number of nets and instances in tier A so that *5 in tier B could not alias with *5 in tier A. Then we would have found the inter-tier connection, and would have had to change *(5+x) to *1 because in actuality they

are the same net. Then we must offset all of the subnet assignments in the tier B portion of the net so that they do not share any subnet numbers with the tier A portion of the net, or else we would be shorting together parts of the net and lose parasitics that we should be accounting for.

After the naming conventions are settled, there is the task of actually removing the 3D via and joining the nets that were joined by the 3D via. First we need to identify a 3D via, which we cannot do solely using net names. If we were to identify which nets in one tier were connected to a net in another tier only by some sort of suffix added to the net name, we would not know which specific subnet to make the connection at. Instead we need to use net name suffixes and standard cell library names. In our research group's standard cell library, all 3D via cells (and only 3D via cells) had names starting with 'V' so that was used to identify a 3D via cell and the specific net subnets it was connected to. Then we use the net names to know which net in another tier is connected to the current net. Unfortunately this is one reason that this code is specific to a standard cell library, albeit an easily modifiable one.

Perhaps the most obvious problem is that we need to insert a parasitic resistance and capacitance for the 3D via. Seeing as current EDA tools have no notion of 3D vias we certainly cannot extract the parasitics of one using design tools. Instead, we needed to perform electromagnetic field simulations to determine the parasitics of an inter-tier via. This was as interesting and accurate approach, which we will discuss in the RC extraction chapter later. Assuming we can now replace the 3D via with an interconnect Pi model, how we connect the nets from each tier, i.e. if we need to add subnets or modify existing ones, depends on how the RC network is connected to the 3D via cell and whether there was wiring capacitance on the 3D via cell's pins. Another worthwhile aspect of the code is its

ability to handle coupling capacitors. For future integration of a crosstalk analysis tool into the 3DIC verification methodology, we would need to supply a merged SPEF file that includes coupling capacitors. While experiments have shown timing runs to be accurate with lumped ground capacitances, this also gives the user the option to include coupling effects in timing calculations.

The last complication with the *spefmerger* to be discussed here has to do with run time. Based on the fact that it is not uncommon for SPEF files to be two million lines long or longer, the code dealing with them is difficult to write. It must not load too much into memory and must avoid traversing the entire file more than once or else run time would grow too high. The code is also writing out a file that contains all the information in each *tier-specific SPEF file*, which was normally between five and six million lines in our case. To avoid the problems that were just explained and others, there were a number of complications that result in a lot of bookkeeping during program execution. This, combined with the fact that it was written in Perl which is an interpreted, as opposed to compiled, language already forces the sacrifice of performance for ease of writing the code.

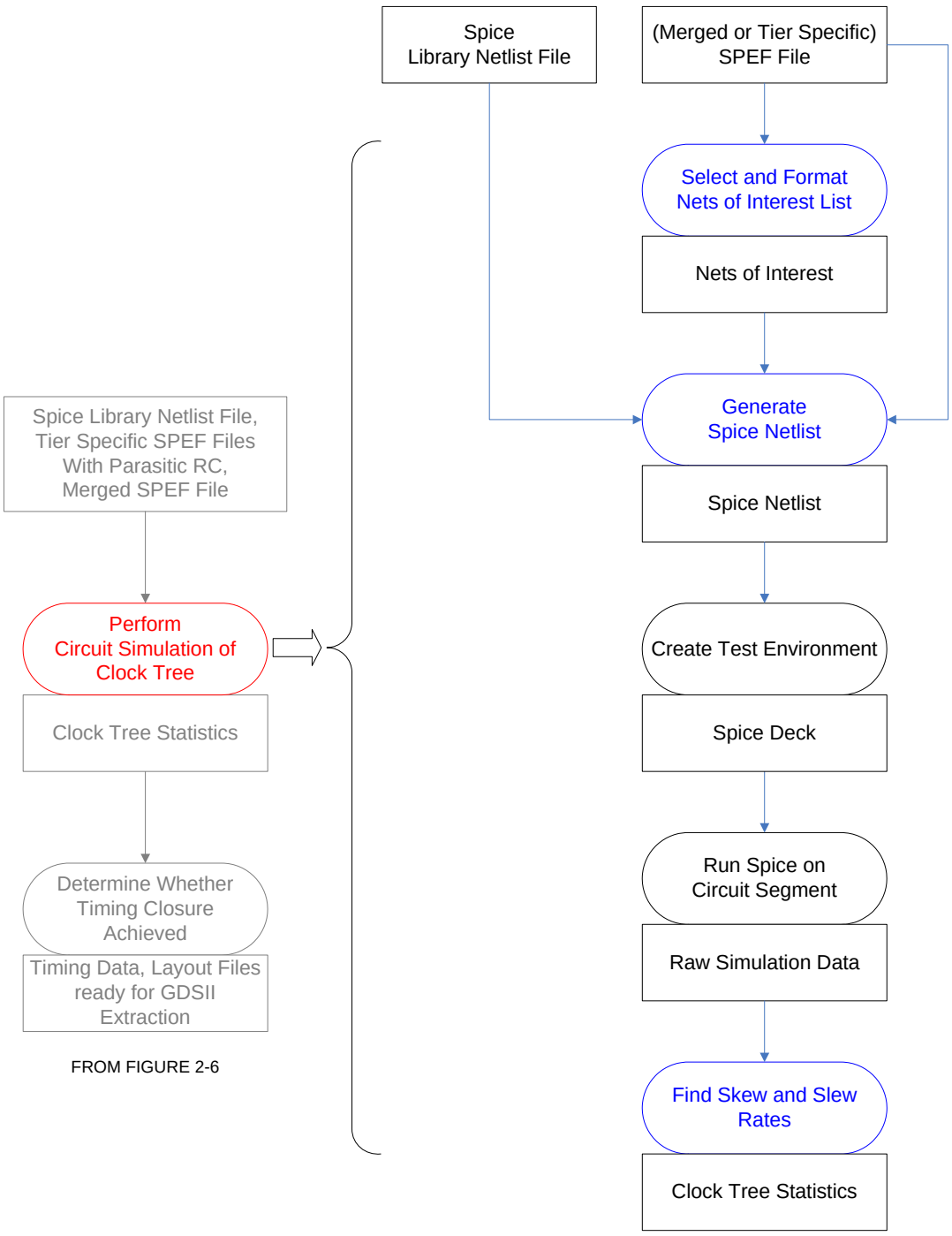


Figure 3-1: Simulation Based Timing Verification Flow

3.3 Circuit Simulation of Clock Tree

Figure 2-6 shows that at this point we can "Perform Circuit Simulations of the Clock Tree." Figure 3-1 dives into this step in greater detail. The overall goal of this step is to

determine the specific performance of a placed and routed clock tree. Using a circuit simulator we will find the skew of the clock tree, the minimum and maximum slew rates at the clock sinks, and the insertion delay. It is first important to note that this figure makes no mention of a clock tree in particular and that it does not specify whether we are working with one tier individually or the design as a whole. The extraction and simulation methodology shown in figure 3-1 will work for any circuit or segment of a circuit. It is based on *spef2spice*, which when written was intended for use with clock trees but can also be used on any other part of a circuit. It can be used to easily extract a portion of a parasitics file and translate that to a Spice netlist for circuit simulation of any path that is of particular interest. Furthermore, it is recommended that this flow actually be performed on the clock tree several times independently, once for each tier and then once for the design as a whole. According to the design methodology, the clock tree was created on each tier independently, so we should analyze them individually first. That way we can determine whether a specific tier's clock tree is performing worse than the others, or if one tier's clock tree has a particularly long or short insertion delay that ruins the overall performance of the clock tree. Afterwards, of course, it is necessary to analyze the clock tree as a whole. The flow will be described as if we were running the design as a whole, but the process is the same for an individual tier.

After obtaining the *merged SPEF file*, we must determine which nets are to be included in the simulation. In the case of the clock tree, the user must just create a list of the nets that make up the clock tree. This is slightly more cumbersome for a simulation of a data path, but it is generally easiest to create a list of the nets in a clock tree using unix commands. A single command such as:

```
grep clk design_name.spef | grep -v "I[0-9]" > clocknets_merged
```

would very quickly write to a file called *clocknets_merged* each clock net listed in the SPEF file. The second "grep" command in the line removes all of the clock buffer and inverter instances before writing to the file; it is assumed that in the instance name instances are differentiated from one another using I1, I2, I10, etc. Some formatting changes using a text editor may still be necessary because later stages expect a file containing net names, one per line, with nothing but the net name on the line.

Next we are ready to use the *Spice library netlist file* discussed earlier, the *merged SPEF file*, and our list of *nets of interest* to create a Spice netlist. This step is automated by the *spef2spice* Perl script mentioned earlier. Unlike the *spefmerger*, this code is not specific to a particular technology or design methodology. A valid SPEF file contains all parasitics of cell to cell interconnect and tells which subnet is connected to which pin of all standard cells the net is connected to. Therefore, between the SPEF file and the *Spice library netlist file*, we have enough information to construct a netlist. The script generates a Spice netlist that includes each net listed in the *nets of interest* list. It also instantiates every standard cell with at least one output and one input pin connected to nets in the nets of interest list, i.e. every standard cell instance that exists between any two nets of interest. *The Spice library netlist* is included with an .include statement in the Spice deck, so whenever the script needs to instantiate a standard cell it can simply instantiate it using a .subckt Spice statement.

In order for a standard cell library to be useful, it must be characterized. This normally consists of a large number of circuit simulations (such as Spice as in our case) on each cell individually to find the delay of each timing arc in it over a broad range of temperature, capacitive loading, input slew rate, etc. conditions. This information is then normally used to create large lookup tables, to which a timing tool would refer when

calculating timing data for a large design made up of these cells. The *Spice library netlist file* is normally created as an intermediate step in library characterization and would be obtained from library developers. However, we can use it because it contains separate netlists of every cell in the library. So long as we can reference this file from our Spice deck we can easily instantiate cells in our Spice netlist.

Using the *Spice library netlist* works well for simple clock trees where the only cells that need to be instantiated are inverters and clock buffers. In more complex trees with clock gating, the data phase input to the clock gate may not be listed as a net of interest, in which case that input to the gate would be labeled as a floating node and a warning message would be printed to the screen. Such also is the case when this code is used on a data path, and inputs to cells along the data path are nets not listed in the *nets of interest* list. Those inputs also would be wired to a new net labeled as a floating node. It would then be the user's responsibility to edit the netlist by hand to drive those floating nodes with stimuli such that the desired transitions along the path would result. After the netlist is created the user is prompted with the question:

```
Do you want to measure skew to a pin that is common to several cells?
```

The user can reply "y" or "n", the latter of which causes the run to terminate. Replying "y" will prompt the user for the reference node. The reference node must be one of the nets listed in the *nets of interest*; this net will be considered the source of the signal of which the skew is being measured. To measure clock skew, we would enter "clk" or whichever node/pin the voltage source driving the clock is connected to. After that, the user is prompted for the standard cell pin to measure to. This is the name of the pin in the standard cell .subckt statement of the sink for the signal whose skew we are measuring. For example, if measuring the clock skew using the 3DIC standard cell library we would enter "CLK."

Measure statements are then inserted to measure from the net assigned to be the reference node, to each sink. Assuming we are analyzing a clock tree, this is effectively finding the insertion delay. Hard coded into the script are statements that will measure rising to rising and falling to falling transitions. Minor modifications would allow the script to work with inverting logic paths, but this way we keep Spice from trying to measure transitions that do not exist. When measuring insertion delay we measure from the 50% mark of the source's transition to the 50% mark of the sink's transition. Measure statements are also written to measure the rising and falling slew at each clock sink. For these inverting versus non-inverting logic is irrelevant because the signal is measured against itself. The .measure statements record all data over the first clock period, so long simulation times are generally not needed.

Depending on the application, the user may choose to deviate from the default settings in *spef2spice*. At the top of the code, there are three parameters the user can modify, but they are generally not changed often so are not command line options. There exists a variable called $use_pin_caps. The LIB file lists pin capacitances for each cell that are included in the SPEF file separately from the wiring capacitances. Normally this variable would be set to a 1 so that this pin capacitances from the LIB file would be included, but when they can be ignored by setting $use_pin_caps to a 0. Like the *spefmerger*, this code also can accommodate coupling capacitances. However, this poses a similar problem to instantiating the cells. At times when coupling capacitors are used in the SPEF file a capacitor appears between a net you intend to extract and one you do not. The net you do not extract would not be in the simulation, and thus we would have a floating node. This capacitor is set to ground, under the assumption that the net that is not extracted is not

switching during the simulation. This case is handled by the $use_worst_case_switching_mode variable; setting this switch to a 1 doubles the capacitance values, modeling the situation where the nets are switching in opposite directions. The default value for $use_worst_case_switching_mode is 0. The last parameter to be adjusted has to do with the slew rate .measure statements. The $slew_high variable is set to 0.8 by default and the $slew_low variable is set to 0.2 by default. This will measure a 20%-80% of V_{DD} rise time and an 80%-20% of V_{DD} fall time. Changing these variables allows you to measure the transition time differently. After setting these variables, all other information is provided at run time on the command line. An example of a command to run the code is:

```
perl spef2spice.pl [SPEF file] [Spice library netlist file] [-nets [nets
      to extract] | -file [file with nets to extract]]
```

The first two arguments, the SPEF file and the *Spice library netlist file* have been discussed; paths to these two files need to be supplied here. The third argument gives the nets of interest, or list of nets to extract. There are three options to give this information, but so far we have only mentioned listing them in a file. If the third argument is "-file," then that must be followed by the path to a file listing the *nets of interest* as we discussed earlier. However, in the event that we are doing a very simple simulation, e.g. just one net is being simulated, there is the option to make the third argument "-nets," in which case the nets to extract are listed right on the command line, a single space between each. Either the "-nets" or "-file" option can be used, but not both. If the third argument is left off, then the entire SPEF file is automatically converted to a Spice netlist. This is recommended only for very simple circuits, otherwise long *spef2spice* runtimes and even long Spice simulation runtimes would result. Assuming that upon completion there were no floating node warning messages

generated, a complete netlist has been produced. The Spice deck requires a little more modification.

The next step in figure 3-1, "Create Test Environment", requires the user to manually complete the Spice deck. The *spef2spice* script creates a file that contains a comment about how and when the Spice file was created, it inserts the V_{DD} rail (if the process uses a V_{DD} not equal to 1.5 V or multiple power rails then either the Spice deck or Perl code will have to be modified) and inserts a hardcoded .lib statement, which is a path to the transistor model being used. It then writes out the entire netlist, including standard cell instances and the .include statement to which the .subckt statements refer. The default operating temperature is 25° and standard .OP, .save, and .OPTION lines are included in addition to any .measure statements that were requested. However, the Spice deck is missing other analysis statements and voltage sources to provide test stimulus. To measure clock skew normally we would insert a .tran statement with step size and simulation end times to perform a transient analysis. Also, we would need to insert a pulse voltage source to drive the clock. This specifies the clock low and high voltages, delay time, transition times, pulse width, and period. This is normally a simple task, and once these lines have been added, then the Spice deck is complete and ready for simulation.

After the "Create Test Environment" step, the figure shows that we are ready to simulate. This is as simple as running HSpice from the command line on the Spice deck that we have just completed. Even for a large clock tree, the run time generally is not unreasonable, less than thirty minutes for our entire FFT test chip of 2,061 clock sinks. This simulation produces binary data files that can be viewed with a waveform viewer if the user so desires. More importantly, an mt0 file is generated by all of the .measure statements.

This file contains all of the numerical data that the .measure statements recorded; for each clock sink there is a rising insertion delay, falling insertion delay, rising transition time, and a falling transition time. This mt0 file is represented as "Raw Simulation Data" in figure 3-1. Important clock tree performance numbers can then be extracted from the "Raw Simulation Data". This is performed in the next step of the flow, the "Find Skew and Slew Rates" step. This step is also shown in blue because it is automated by the supplied *find_skew* Perl script. This code parses the mt0 file and records the maximum and minimum rising and falling insertion delay. From this, the rising and falling skews are calculated. Normally we would only care about one of these values unless there are both rising and falling edge triggered storage elements in the same design, or latches and flip-flops in the same design. The script also finds the maximum rising and falling slew. All of this information is printed to the standard output, but can be redirected to a file. It is recommended that the user run this circuit simulation methodology once for each tier and once for the design as a whole also as a check. The performance statistics for the whole design should make sense in the context of the results of the tiers individually, for example by looking at the maximum and minimum insertion delays of the individual tier runs, we should be able to roughly predict the skew of the design as a whole. This information is the goal of the clock tree circuit simulations from figure 2-6; we will find the same information and more from a static timing tool next, and check that the methods agree with one another.

We can now explore the static timing analysis approach, shown in red on the right side of figure 2-6. The static timing verification flow requires just two input files, the *merged Verilog netlist*, and the *merged SPEF file*. The sequence of steps to verify timing using Synopsys' static timing analysis tool PrimeTime are shown in figure 3-2. This flow

needs to be performed just once for the overall design. This is where the ideas of making a 3D design appear as a 2D design really come into play. By the end of this part of the timing verification flow we will have generated timing reports that show the worst slack timing arcs for both setup and hold tests, which will show whether the chip meets timing. This part of the flow can easily work with any technology or design methodology, it is as easy to adapt as any other major EDA 2D design tool.

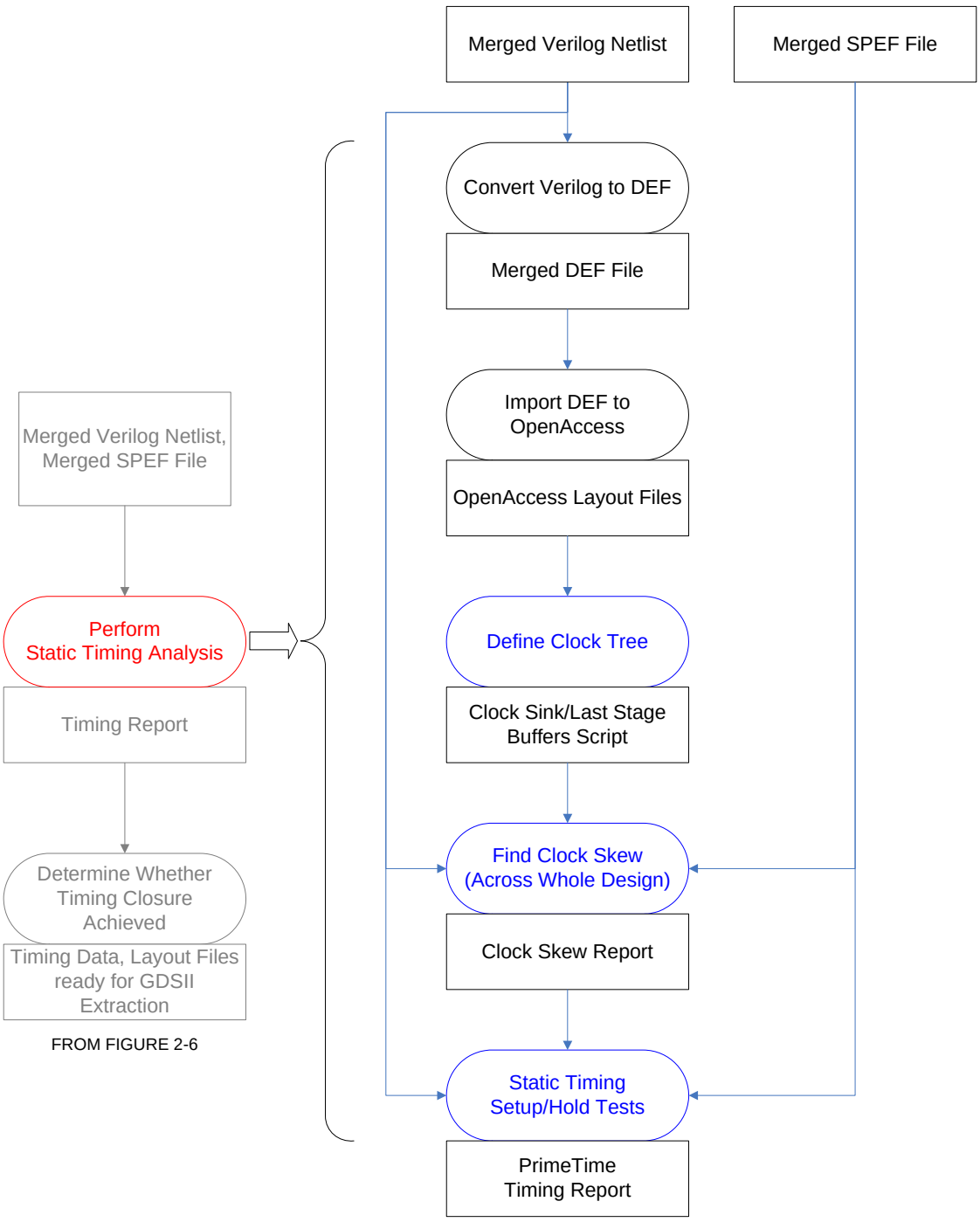


Figure 3-2: Static Timing Based Timing Verification Flow

3.4 Static Timing Verification

The first step is to create a DEF netlist of the design. A DEF file normally contains detailed physical design floorplanning and layout information, but that is not necessary here.

There is a sequence of scripts we will be using that read in a netlist as it would appear in a DEF file; all we need is the connectivity information. This is fortunate because placing and routing a large design generally takes design tools long time and still is likely to require some user interaction and modification. However, since we just need a DEF netlist we can use Cadence's First Encounter physical design tool to read in the *merged Verilog netlist*. After the design is loaded into memory we can immediately write out a DEF file. Figure 3-2 refers to this DEF file as the *merged DEF file*, and it is used in the next step which involves importing a design to an OpenAccess database and operating on it.

OpenAccess is available as a C++ or Python application protocol interface (API), but here Python is used for coding ease. The intention of OpenAccess is to provide a database in which a design can be stored and accessed by a variety of design tools and methodologies independent of file formats and such [4]. The available version of OpenAccess does not support SPEF files, which would make the verification task considerably easier. After OpenAccess is installed and available on the system, first the standard cell library used to create the design needs to be imported. For this the user needs the LEF file that was mentioned earlier. The LEF file contains some process information, but the LEF file for a standard cell library is analogous to a DEF file of a design. Running a single OpenAccess function, lef2oa, reads a LEF file into the OpenAccess database. Upon completion, the standard OpenAccess function def2oa is used to read in the *merged DEF* file that we created using First Encounter in the previous step. With the design in the database, we can execute a Python script that identifies and defines the clock tree. This Python script called *gensinklist* makes extensive use of OpenAccess commands; it takes in only a list of the standard cells used as buffers and inverters in the clock tree and the OpenAccess layout file generated in the

previous step where we imported the design into the database. The user tells the script which net to analyze, in our case the clock, and it iteratively finds each instance connected to that net. If the instance is in the list of clock tree buffers defined earlier, then the script recursively goes through that cell and analyzes the net on its output. Execution continues until each path the script traverses down hits the input to a cell that is not in the buffer list. The script records these input ports and considers them the clock sinks; it also records the last cell instance that it traversed through before hitting a clock sink, and considers that instance a last stage buffer. The output of *gensinklist* is TCL code that defines and populates two list variables, one for clock sinks and one for last stage buffers. This script, referred to in figure 3-2 as the "Clock Sink/Last Stage Buffers Script," that is generated is sourced later. At this point we are ready to begin timing the design in PrimeTime; first we will find the clock skew and other clock tree performance metrics across the entire design and compare this with the data obtained from the Spice simulations.

According to figure 3-2 we must next find the clock skew across the design using static timing, via an automated step. This is done with the *clockskew* TCL script. The *clockskew* script is the first of two essential TCL scripts that are invoked from within the PT Shell (PrimeTime command line) environment. They both are based on PrimeTime TCL commands within TCL language loops and constructs. The *clockskew* script begins by sourcing the script generated in the last step to define the clock sinks and last stage buffers. After loading in the design and setting up the clock and a few other parameters, it generates some preliminary reports. One report is generated by the clockskew procedure within this script, which uses PrimeTime to identify each timing path from the source of the clock signal to each clock sink listed in the clock sinks list. The delay along this path is then calculated;

as we iterate through each clock source to clock sink path we keep track of the maximum and minimum insertion delay. Subtracting those two in the end gives us the skew of the clock tree. Similarly, we examine the output pin of each buffer in the last stage buffers list and find the transition time of a rising edge at that point. This way, this script will calculate for us the skew and max slew rate of the clock tree. The clockskew script calculates the skew for the clock tree both before and after it is back annotated with parasitics, for comparison, and writes this information to the clock skew report. At this point we have determined the insertion delay for each clock sink both using circuit simulation and static timing analysis, and from that have arrived at the clock skew two different ways. We would expect to see concurring results, if that is the case then we can move on and continue to use static timing analysis to verify that the timing arcs meet setup and hold timing. However, in comparing the slew values we can expect some discrepancy, because the Spice simulations and static timing calculations measure it at different places. The Spice simulations measure the slew at the clock input of the storage element, which is where it is really of interest. The static timing calculations measure the transition rate at the output of the last cell before the storage element. This is because PrimeTime only allows you to measure slew at the output pin of a cell, but the impact should be small because the difference is attributed only to the additional degradation introduced by the parasitics on the (hopefully short) wire connecting the last stage buffer to the storage element.

The setup and hold timing checks are also performed by a TCL script, the *timing_cd* script. This is the final step of the static timing flow shown in figure 3-2, the PrimeTime timing report indicates whether or not we have met timing. The code for *timing_cd* is slightly simpler than the *clockskew* code, because we mostly use commands built into the

static timing tool to check for setup and hold violations. However, we do need to refer to the clockskew report to find the maximum transition time and maximum insertion delay values for use here. Before loading in the design we set the CLK_SAFETY_MARGIN parameter to the maximum transition time and the OFFCHIP_INPUT_DELAY equal to the sum of the maximum insertion delay and synthesis input delay. First, we add in a clock safety margin because we can not be certain when exactly during the clock edge transition the flip flop senses the transition. Therefore, adding in the entire transition time is likely to be a little pessimistic but will ensure that we are not relying on the flip flop to see the clock transition at a certain point, e.g. exactly at the 50% mark. The off-chip input delay is used to set when valid data appears at the primary input pins; this value is in reference to the clock so it is set to the clock insertion time plus the estimated arrival time with which the logic was synthesized. After loading the design and the associated *merged SPEF parasitics file*, we set up the clock, set up the driving cell, and a few other parameters before executing the report_timing command. This command actually triggers timing within the tool; it begins to calculate the delay for each timing arc to see if it violates any setup constraints. The default mode for report_timing is to time for maximum path delay, so essentially the generated timing report will tell us the maximum operating frequency of the current design. The timing report generated by this command tells us the single path with the worst setup slack. It will tell us what two points the path is between, i.e. which registers or primary I/O, and the exact path through the logic with the breakdown of propagation delays along that path. When defining the clocks earlier in this script we specified the period of the clock with which to run, and because clock cycle time fits into the setup equation, by subtracting the slack of the displayed path (whether negative or positive) from the period we can determine a clock speed

at which the design will operate correctly. We can modify this command to report as many worst slack paths as we wish. Next it is clearly necessary to run the report_timing command with the –delay min option. This tells the timer to check each timing arc for the minimum path delay, and run a hold test with this value. Once again, the worst slack hold path is reported, but only modification to the hardware will fix hold violations. If the timing reports generated by this script show only positive slacks, the clock tree performance statistics from static timing and simulation agree, and we are pleased with the clock tree performance, then timing can be considered closed for the chip. If not, either the circuit or the constraints under which the circuit was timed may need to be modified. We must be sure that all paths that violate timing are not false paths and that timing assertions are realistic before making changes to the circuit, which would necessitate rerunning this timing verification flow.

3.5 Timing Closure

At this point we have reached the final step in the timing verification flow depicted in figure 2-6. By examining the clock tree reports, setup timing reports, and hold timing reports we are able to determine whether timing closure has been achieved. Beginning with tier-specific SPEF parasitic files, a Verilog netlist of the entire design, and a Spice netlist of the standard cell library, we can generate all other data and files needed to verify the timing of a 3DIC. This design flow presented here closely resembles that of 2D timing verification through the use of certain merging scripts and OpenAccess code to make our multi-tier 3D design appear as a normal 2D design to the EDA design tools currently available. This allows us to use PrimeTime, and its useful built-in timing functions [5]. This is the simplest approach until the EDA tools industry can catch up with the technology available. This flow was intended for use with the described 3DIC design flow, but for the most part is not

specific to it. In the following chapter we will go into more detail on the parasitic extraction and describe some examples that help to verify its accuracy.

Chapter 4: Parasitic Extraction

There is no commercial tool that promises to accurately extract a layout with inter-tier vias, an essential task in verifying clock tree's performance. Furthermore, we have stated how important parasitic extraction of metal is, in today's technologies where wiring delay dominates gate delay. This trend will certainly continue, as feature size shrinks and chips get more complicated. It follows that as chips become more complicated accurate parasitic extraction of interconnect becomes more difficult and computationally intensive, it becomes more important. In a number of ways, 3DIC's present solutions to some of the problems with interconnect by shortening average wire length in a chip. Wiring in today's chips is very susceptible to cross process and even cross chip manufacturing variation. To model this variation accurately, tools must treat each metal layer independently since each is fabricated with a different mask during its own process step. Parasitics of metal layers within a chip vary completely independently of each other, which is not typically addressed at all by today's design tools. This compounded with the often inadequate wire load models being used today, produce a great deal of uncertainty in timing measurements. Smaller wires can lessen the impact of such timing uncertainty and lead to more accurate predictions of an actual chip's behavior.

4.1 RC Extraction

So far, we have only discussed resistive and capacitive parasitics, but at times we must treat long interconnect as a transmission line and also include inductive parasitics; in this case we would be performing RLC extraction. This certainly would add a great deal of complexity to every single delay calculation, so we should make sure that it is absolutely necessary. A common rule of thumb when modeling interconnect today is that if the time of

flight is at least 2.5 times larger than the rise time of a digital signal, then transmission line modeling must be used and inductance cannot be ignored. In this case the signal will have increased by 40% over its initial value and we can no longer say that the level of the signal is constant along the entire line. We say it is safe to use a lumped RC model when the rise time is greater than 5 times the time of flight. This is typically the case with interconnect in digital circuits that are not abnormally long, so in this work we use only RC extraction [6].

4.2 Parasitic Extraction Algorithms

There are many methods and algorithms in use today to calculate the parasitics of an interconnect network, capacitance in particular, but in general they fall into one of three categories. The fastest and simplest algorithms perform 2D extraction, that is, all 3D aspects are ignored. Capacitance and resistance are only measured per unit of length; it assumes that the width of segments does not vary. Overlapping lengths are counted additionally. This method uses resistance per square values to accurately calculate resistance, but this method sacrifices accuracy in the capacitance calculations. Here we are able to consider the system as a set of 2D shapes. Requiring more computation and providing more accuracy are the 2.5D or quasi-3D extraction algorithms. These perform two separate 2D passes to analyze the system. The system is modeled accurately, but this is considered flat extraction because the heights of individual layers are not considered. In 2.5D algorithms signal propagation can be considered in three dimensions, but only one at a time. A signal is considered to travel in one direction, that segment's electromagnetic field is analyzed, and it determines the effect of that segment on neighboring conductors using Green's Algorithm. The accuracy of the 2.5D extractors lies somewhere between that of the 2D extractors and that of fully 3D extractors. The 3D extraction tools are generally referred to as electromagnetic field solvers.

These achieve the highest level of accuracy by modeling everything as a 3D object. Signal propagation no longer is assumed to travel in one direction only. The electromagnetic fields that exist are analyzed and capacitance values are calculated using the concept of capacitance matrices [7].

To determine parasitic information for timing calculations for our design and verification flow, we have chosen to use the 2.5D parasitic extractor in Cadence's First Encounter. This quickly performs full chip RC extraction and writes out a SPEF file directly from the tool we used to perform placement and routing [8]. This gives us a number of options, most notably the ability to extract coupling capacitances separately. We have shown this method even with the default settings to be accurate by comparing extraction results from examples of circuit segments between First Encounter and Ansoft's Q3D tool, an electromagnetic field solver. Q3D works like the 3D extraction tools mentioned, by solving capacitance matrices.

4.3 Electromagnetic Field Solvers and Capacitance Matrices

Typically in a structure there are multiple objects, and we want to know all of the capacitances that exist between each of them individually. However, these tools work by determining one capacitance at a time for different possible configurations of the interconnect being connected to one another. If we examine the case of three pieces of interconnect some distance above a conductive ground plane we would discover that there are seven such possible configurations of the interconnect being connected to each other, these are shown in figure 4-1. In each case the conductors are broken into a white group and a black group. Q3D then applies a zero volt potential to one group and a one volt potential to the other and solves for the total charge on each. It can then determine the total capacitance

that exists between the two groups. This total capacitance is the sum of a set of smaller capacitances, as shown in the figure, which are the ones we are actually interested in. Through simulation, Q3D finds the capacitances labeled in figure 4-1 as C_A-C_G. We are not interested in these total capacitances, but are in the smaller ones, those capacitances between the conductors individually. Once we know the total capacitances, however, and which of the capacitances between the conductors comprise each total capacitance, we can solve a system of equations that will tell us the capacitance between each two objects in the system. This system of equations is most easily represented and solved in matrix form; the tool would solve such a capacitance matrix in order to find the individual capacitances we are looking for.

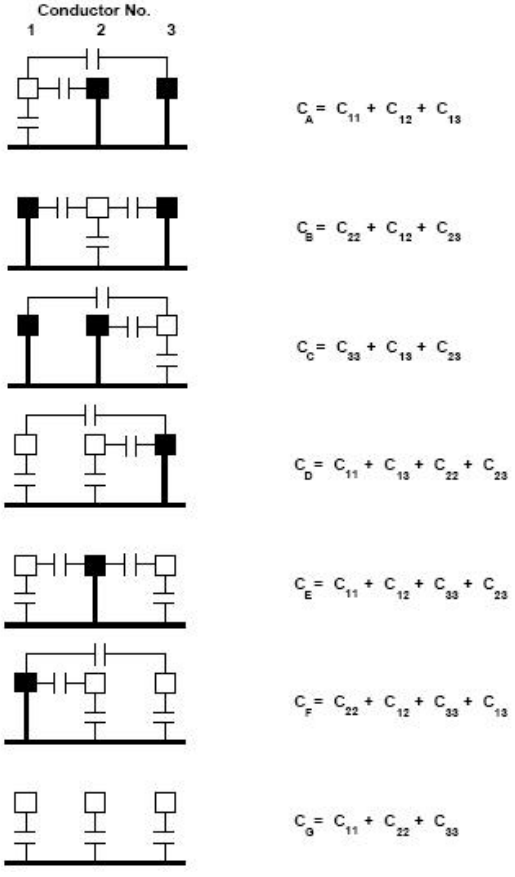


Figure 4-1: Capacitance Matrix Formulation [6]

Q3D creates these capacitance matrices and solves them internally; for ease of use it only gives the user values for each of the capacitances that are between two unconnected objects, the values above labeled as C_{11}, C_{12}, etc. Q3D performs this analysis several times, each analyzing the objects in 3D with finer granularity until finally it arrives at results multiple times in a row that are all within a user-defined threshold of one another. At this point it is satisfied that it has converged on a solution. Here we consider this to be the most accurate analysis available and judge the accuracy of other parasitic extraction methods based on their proximity to Q3D's results [9].

There is just one important shortcoming of using Q3D; since it works by converging on a solution we must worry about situations where it cannot converge. As mentioned briefly earlier and shown in figure 2-1, there exists a layer of metal that is often grounded on the top of the stack of dies. We do not model the complete 3DIC chip with all of its layers at one time because it is so complex and because many of the layers that span an entire die are not conductors. However, it is reasonable to model an object of interest in the presence of the large silicon substrate on the bottom of the stack and the back-metal. Doing this presents a problem because the back-metal that covers nearly the whole chip and the bulk silicon substrate that is as large as the chip are normally both much larger than the object of interest, which is usually a single net or inter-tier via. In fact, they are so much larger that they appear as infinite planes to the object of interest. Introducing two infinite planes into the problem removes the bounds and makes the electromagnetic field that we are trying to solve divergent. Q3D can no longer converge on a solution, and just stops running after a user-defined number of iterations. Therefore, for the purpose of our analysis, we have the choice of ignoring the back-metal or using Q3D to approximate the solution.

4.4 3D Extraction versus 2.5D Extraction

Next we will look at two instances where a net from a placed and routed design was examined and its parasitics extracted. Each time we will first find the parasitics using the 2.5D extractor in First Encounter, then recreate the net and neighboring nets in 3D for analysis in Q3D. This was part of the analysis used to determine how reliable First Encounter's results were for use in timing verification. For simplicity in the Q3D simulation, we assume all nets that surround the net of interest are at a fixed voltage so that we can effectively combine all of those and set them to ground. This way we can just compare the total capacitance calculated for each net, using each method. Figure 4-2 shows a segment of a layout from First Encounter, the net we will analyze is pointed out by the red arrow. Metals 1, 2, and 3 are represented by the colors blue, red, and green, respectively. It is worthwhile to note that that the net is in an area of moderate routing density; the metal 1 segment has a minimum distance metal 1 strip running along side it

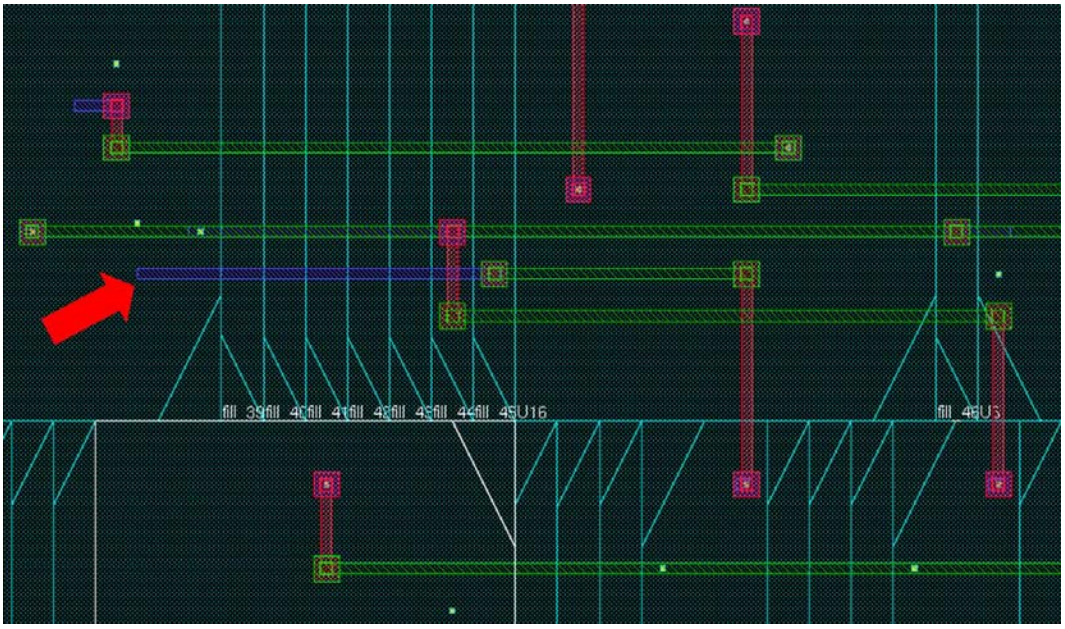


Figure 4-2: Example Net 1 in First Encounter

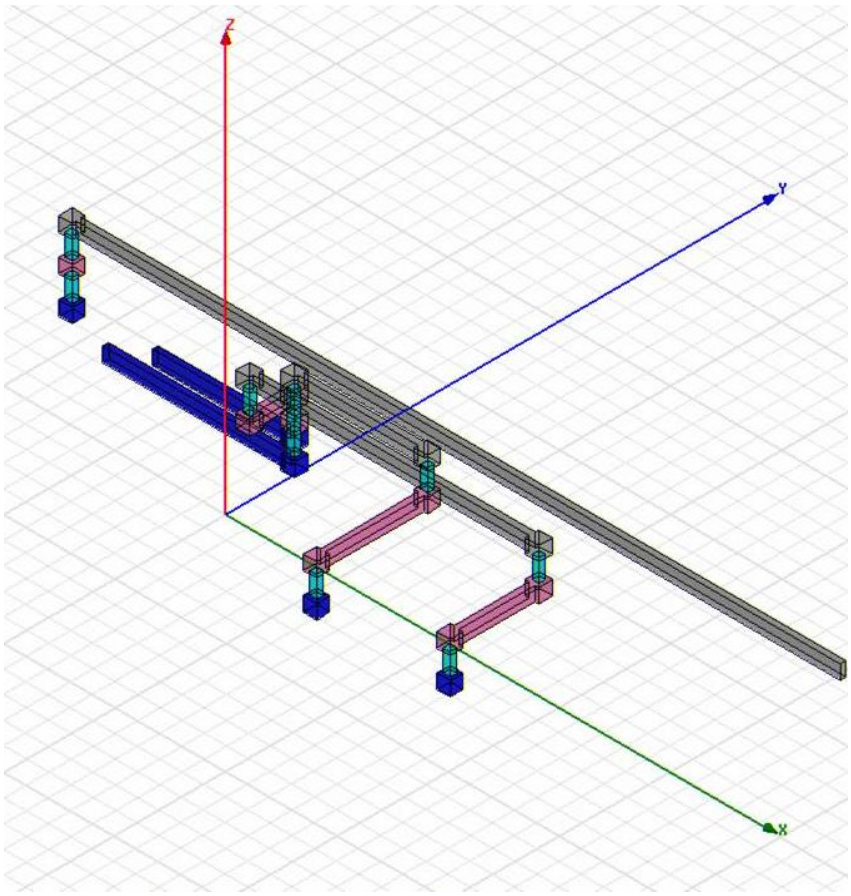


Figure 4-3: Example Net 1 in Q3D

and the metal 3 segment has minimum distance metal 3 strips running on either side of it. The metal 2 run has a perpendicular crossing with a piece of metal 3. This setup has been duplicated in the Q3D modeling environment and shown in figure 4-3, the net of interest and two closest nets have been included. When running the two parasitic analyses on this net, First Encounter finds 2.52 fF on the net while Q3D calculates 2.67 fF. This shows agreement within 6% between the two methods, but to be sure we will perform the same analysis on a different net.

Figure 4-4 shows a portion of a First Encounter layout with the next net to be examined. This one is not secluded, but does not really have any neighboring nets with significant minimum distance parallel segments. This net, like the last, was also modeled in

Q3D and can be seen in figure 4-5. First Encounter finds the capacitance on this net to be 2.10 fF, while Q3D calculates 2.05 fF which is about a 2% difference. This too is acceptable, even better than the first net. It seems that this may suggest the impact of including coupling capacitance in our analysis. From this data we conclude that for speed and ease we can ignore coupling capacitance and rely on First Encounter's 2.5D parasitic extraction. However, we do have cause to make the verification flow compatible with coupling capacitors for use with other current or future technologies and tools.

It is important to recognize that as the feature size on chips shrinks, so does the minimum wire width. Current trends show that the resistance is increasing at a higher rate than is capacitance decreasing for wires. In order to account for this and attempt to reduce wire delays, metal routing has taken on a tall aspect ratio. The metal routing height to width aspect ratio (assuming minimum width wires) is 2.52:1 in the MITLL 0.18 µm SOI process, which is standard for today's technologies. This trend could lead

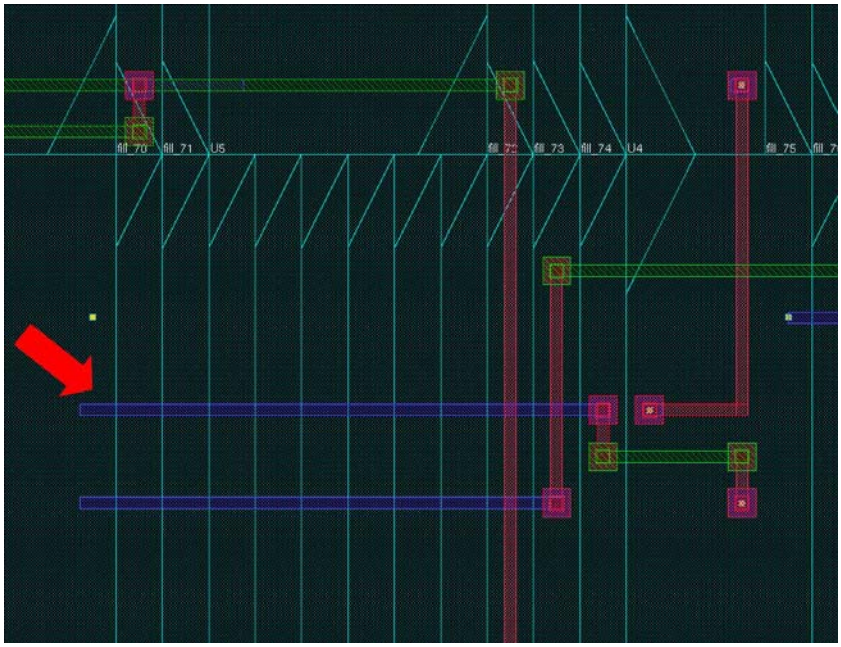


Figure 4-4: Example Net 2 in First Encounter

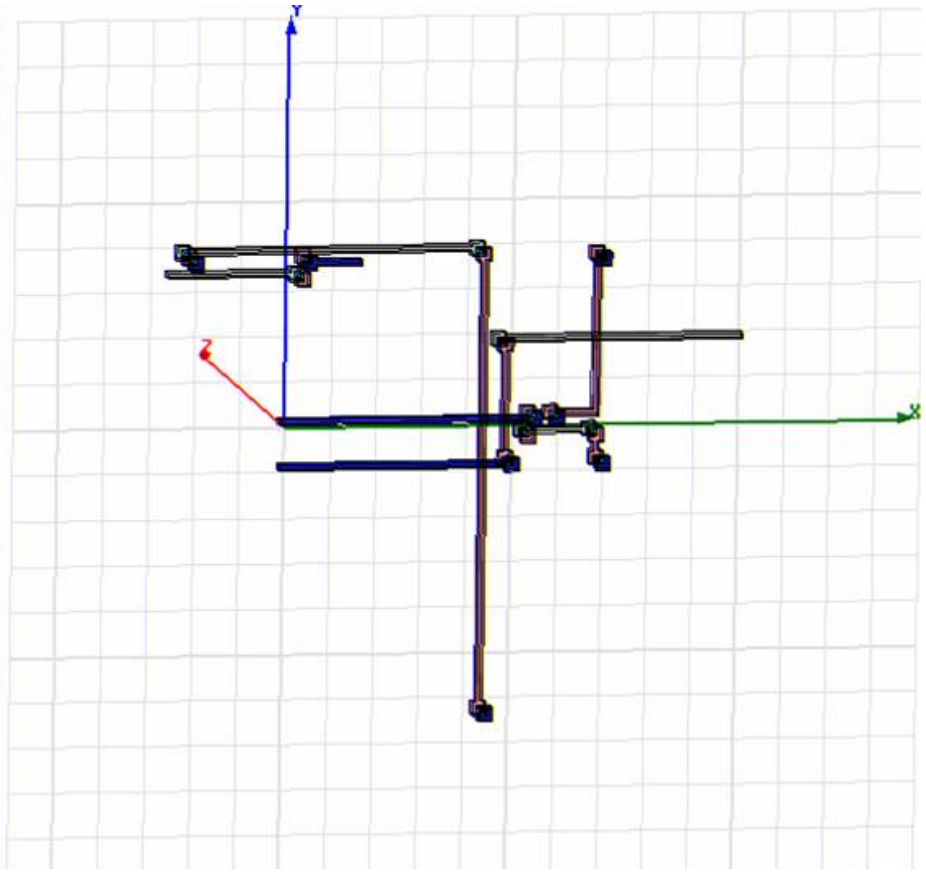


Figure 4-5: Example Net 2 in Q3D

to coupling capacitance soon beginning to have a larger affect on timing, since parallel wires now have large flat surfaces facing one another. For this reason also, the 3DIC verification flow, *spefmerger* and *spef2spice* in particular, work with coupling capacitors.

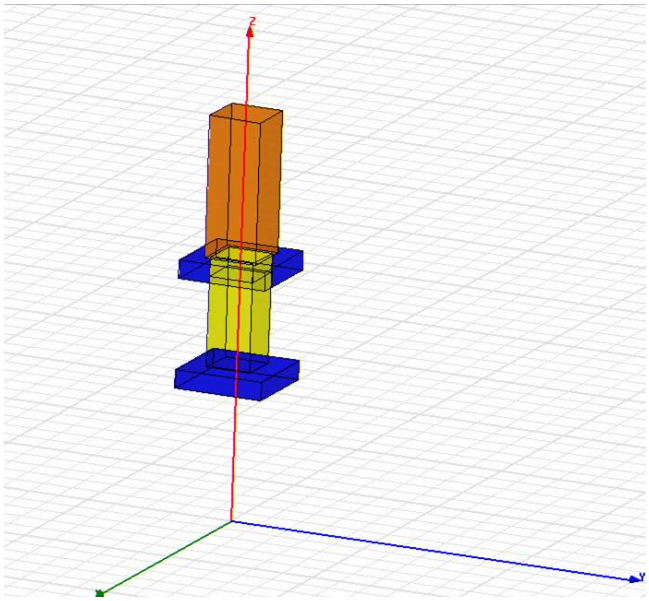


Figure 4-6: Inter-tier Via Modeled in Q3D

Another major use of Q3D in this work was determining the parasitic capacitances of an inter-tier via. Unfortunately, the chip foundry is not yet able to provide us with parasitic data for inter-tier vias. As mentioned earlier, throughout the 3DIC timing verification flow we hide the concept of inter-tier vias from the EDA tools. In the *spefmerger* script, for example, we identify the 3D vias and replace them with capacitors and a resistor in Pi model formation to imitate their electrical behavior. The only option left was to determine the resistive and capacitive properties of the inter-tier vias by Q3D simulations. As can be seen in figure 4-6, a 3D via between the bottom two tiers can be modeled. The dimensions are as per the foundry specification, as are the materials of which the via consists. For the simulation, this via exists alone over a large substrate; we measure the capacitance between these two objects. In this case Q3D shows the via to have a capacitance of 0.82 fF and a resistance of 0.115 Ω. Moving the via higher above the substrate to the position of a 3D via between the next two tiers, tiers B and C in the MITLL process, causes only a minor change in capacitance to 0.89 fF. However, this value is not realistic because in application the via is

likely to be surrounded by metal routing. To account for this, another simulation was performed and is shown here in figure 4-7. The via was shielded on all sides by all possible metal layers that could be a minimum distance away. Once again assuming that all conductors other than the via are at a constant potential, we add these capacitances together to find a total of 4.34 fF on the via. In the simplest accurate model, we would consider this to be the parasitic capacitance on all 3D vias in a design. In a 2D process it is common practice to place two or more vias in parallel when critical signals are traveling between metal layers. This can drastically reduce propagation delay along a line because local vias typically have high resistance, 4 Ω in this MITLL SOI process; the resistance normally dominates the parasitic capacitance on the via. Therefore, putting vias in parallel would greatly reduce the large resistance with a relatively small increase in capacitance. In the case of an inter-tier via, however, the capacitance dominates the resistance. This is not surprising, seeing as at its narrowest point an inter-tier via is 1.75 μm by 1.75 μm. We can conclude, and have verified through Q3D simulations, that there is nothing to gain other than perhaps reliability in placing multiple inter-tier vias in parallel. In fact, the RC product for two inter-tier vias in parallel is actually higher than that for a single inter-tier via. To put this value in perspective, an inter-tier via has approximately the same parasitic capacitance as does a 20 μm run of metal 2 that is shielded at minimum distance on all sides in the same technology.

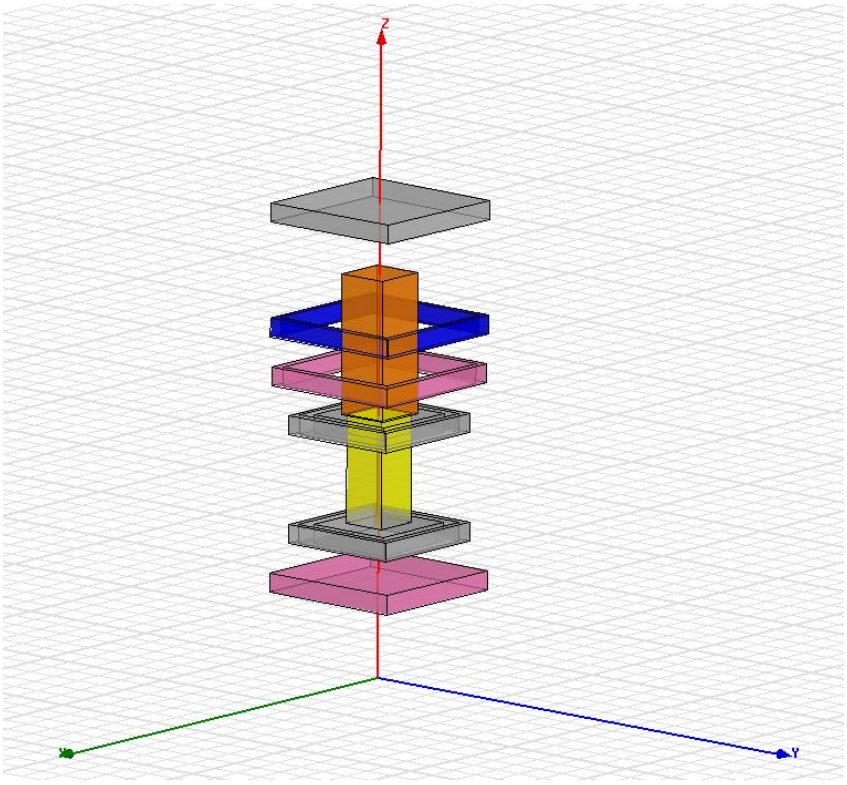


Figure 4-7: Shielded Inter-tier Via Modeled in Q3D

Ideally we would be able to use an electromagnetic fields solver such as Q3D to extract all parasitics for a chip, and have that generate our SPEF file for verification use. However, among other reasons, it is impractical to use a fully 3D parasitic extractor because of the way the capacitance matrix problem blows up. In a system with n conductors, we have $\frac{n(n-3)}{2}+n$ variables to solve for. That means that if one conductor was added to a system of n conductors, then the number of unknowns that we need to solve for increases by n. Each conductor is a net in the design, so this makes a much larger matrix that can be efficiently solved. Other than an electromagnetic field solver, there are detailed parasitic extraction tools such as Cadence's Fire & Ice which claims accuracy within 10% when compared to measured data from silicon [10]. Tools such as these behave as a 3D extractor but can run

much faster because they require extremely detailed techfiles that contain foundry specific data. Such techfiles are difficult to compile and require data that the foundries themselves often to not have readily accessible. In our design environment, a First Encounter's 2.5D parasitic extractor uses a complex and trusted algorithm that has been shown to be accurate through test cases. It is the best available option for use with the verification methodology.

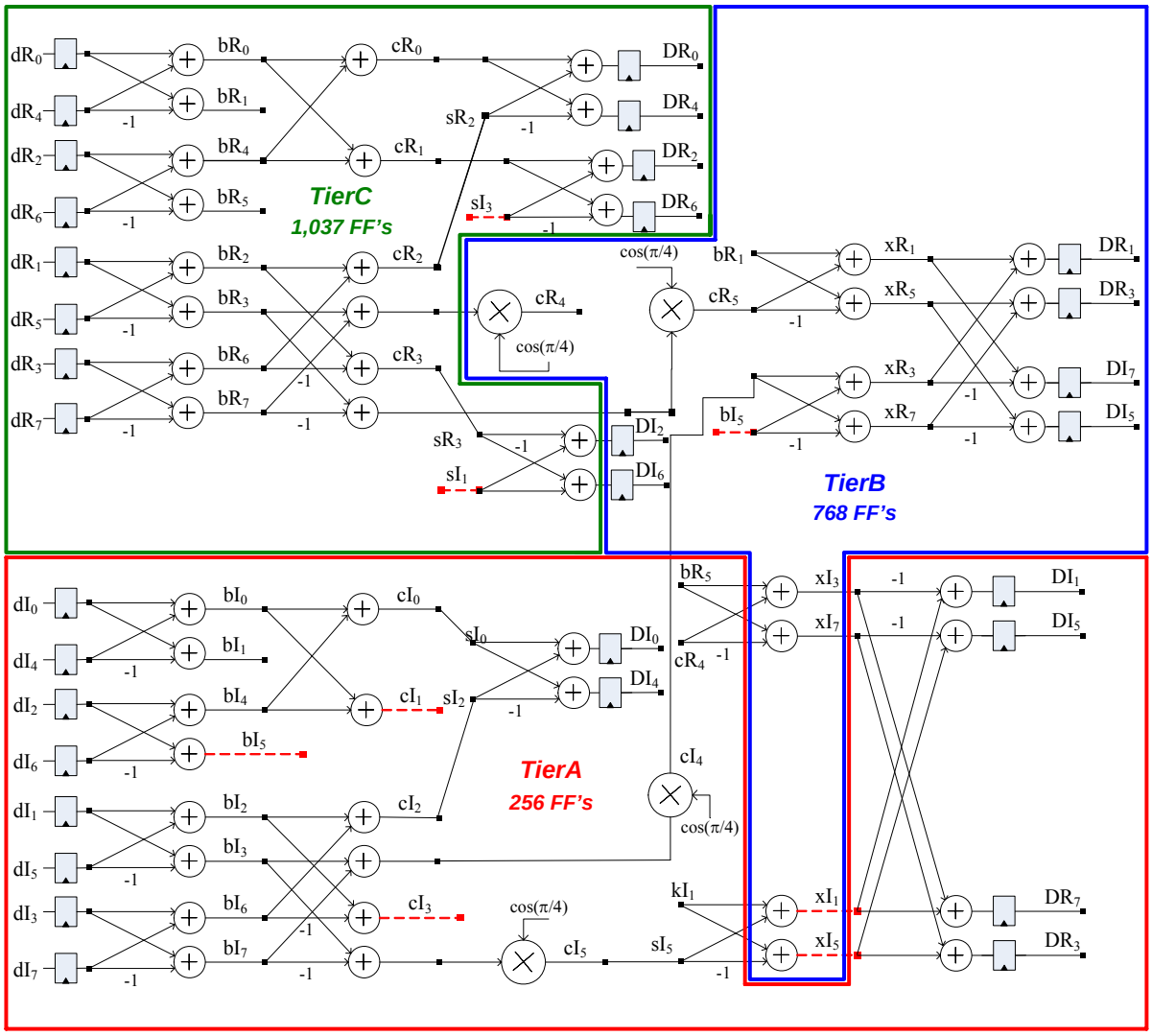


Figure 4-8: Winograd FFT Architecture

Chapter 5: Results and Analysis

5.1 FFT Architecture

After our research group had devised a design and verification methodology for a 3DIC, we needed to test it. To demonstrate the methodology, we then designed for fabrication a chip utilizing nearly 140,000 cells. We have synthesized a complex fast Fourier transform (FFT) design as the test vehicle. It is being fabricated using the MITLL 0.18 μm 3D FDSOI three tier process described in the beginning of chapter 2. The design is an 8 point FFT, 8 samples are simultaneously processed, each sample being 64 bits. The design is a complex FFT where each sample is 32 real bits and 32 imaginary bits. It is a floating point FFT designed using the Winograd architecture which optimizes to use the fewest number of multipliers, which are the most power hungry component of the design [11]. The FFT is such that with minimal difficulty it can be used as a building block to create a 64 or 512 point FFT. Figure 4-8 shows the structure of the FFT, as well as how it was partitioned into tiers. We can see that its design requires four multipliers; two exist on tier A and two on tier B. There are a number of reasons that this FFT was a particularly good choice for testing the 3DIC methodology. First of all, an FFT chip performs a very well known and important function in digital signal processing. It is a common macro to place on a larger chip or to be placed on its own chip. It also is mainly made up of a series of adders and multipliers, two of the most common circuits to appear on digital chips today. Furthermore, it is a large and complicated circuit that is commonly both designed custom and synthesized depending on the application.

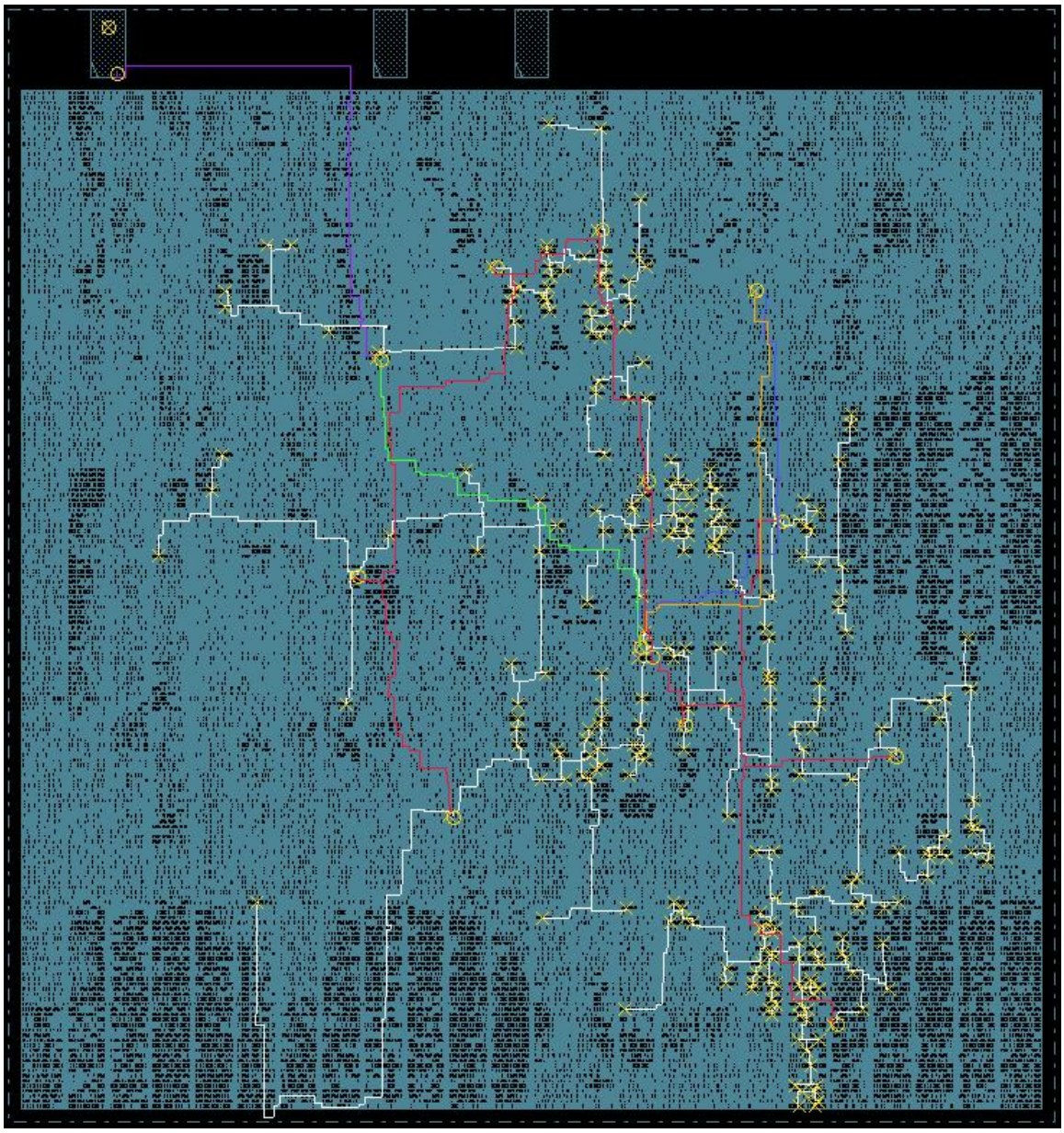


Figure 5-1: FFT Tier A Clock Tree

The aspect of this design important to this work has to do with the number of and location of its storage elements, all positive edge triggered scan D-flip flops (FF's). There are 2,061 FF's in the design, and are accounted for as follows: There are 8 inputs and 8 outputs, each with a 32 bit imaginary and a 32 bit real component which come to 1,024 FF's. Also on the chip but not pictured in figure 4-8 are linear feedback shift registers that feed

input data into the FFT and check its outputs against known correct output data. The chip is for academic purposes and the I/O count is limited, so a built in test vector approach is the best way to test the chip's operation. These LFSR's of 512 FF's, combined with a large hash register of 512 FF's account for an additional 1,024 FF's, which bring the total to 2,048. There is also a 10 bit counter that is used and 3 FF's for outputs to the chip for that we monitor to ensure that the chip is functioning properly.

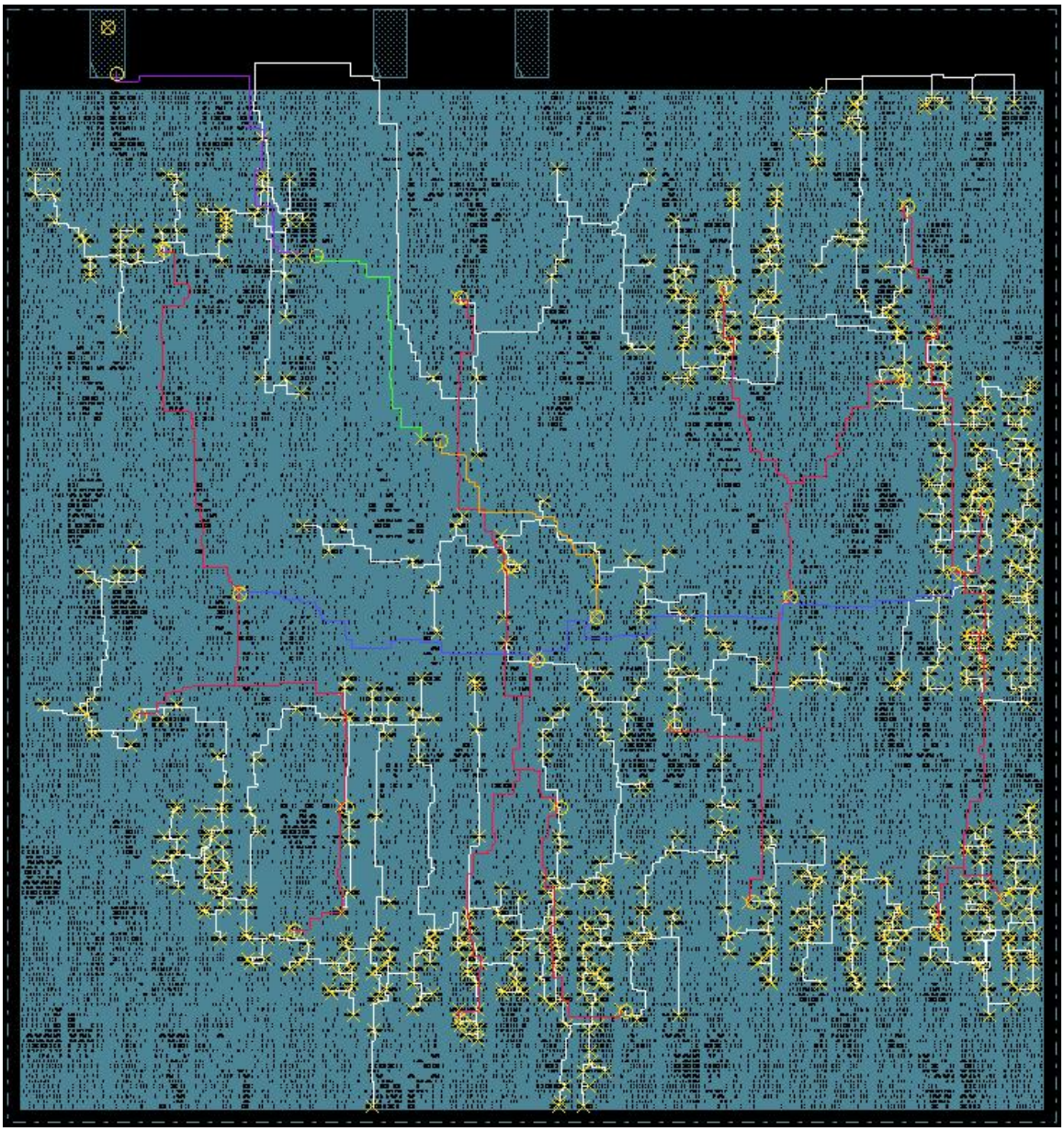


Figure 5-2: FFT Tier B Clock Tree

Figure 4-8 shows the number of FF's on each tier, and shows that there are FF's at each input and output. However, for simplicity the tier divisions are only accurate for the multipliers and dividers and those I/O FF's may not be placed exactly as shown. Figures 5-1 to 5-3 show the layouts of tiers A-C, respectively. All of the routing except the clock tree was removed so that we can examine the clock tree. The yellow marks represent clock tree buffer

cells and clock sinks. On tiers A and B there are 6 level clock trees, and on tier C there is a 5 level clock tree. We see that on each tier the signal is brought in from the pad on the top left corner, level 1 is shown in purple. The signal is then buffered and level 2 is shown in green. The signal continues through buffers and branching out each time, each level is shown in a different color. Starting with the signal coming in from the pad on level 1, the levels 1-6 are represented by purple, green, orange, blue, red, and white, respectively. We can easily see that there are the most clock sinks on tier C, then tier B, and the fewest on tier A. Tier B most clearly shows the H-tree structure, as the clock was routed to the center of the chip then distributed symmetrically.

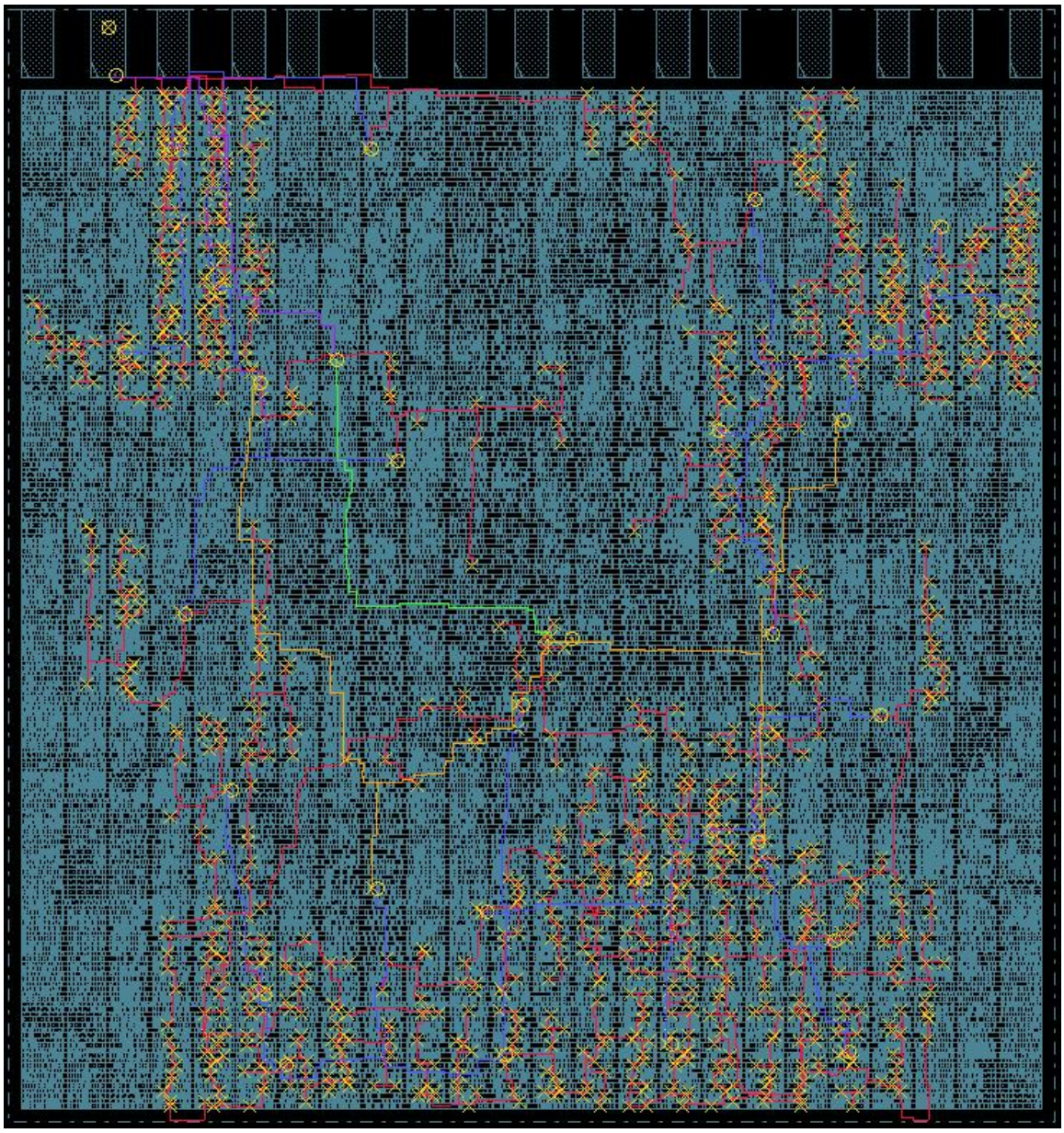


Figure 5-3: FFT Tier C Clock Tree

5.2 Static Timing versus Circuit Simulation

At this point we can examine what results we hope to draw from the FFT about the verification flow. First, we should perform the circuit simulations of the clock tree by going through the steps described earlier for each tier and for the design as a whole. Doing this for the FFT yields the results shown in table 5-1; these simulations are first done with coupling

capacitors. Before going further, we should compare the tiers individually with the entire design. Across the three tiers, we see the largest maximum insertion delay appear in tier A with 2.031 ns. When we analyze the entire design, we find the maximum insertion delay to be 2.084 ns which agree within 3%. Looking at the minimum insertion delay, we see that 1.286 ns on tier C differs from

Table 5-1: Spice Simulation Timing Results from FFT (with Coupling Capacitors)

	Tier A	Tier B	Tier C	Combined
Skew	0.673 ns	0.452 ns	0.428 ns	**0.743 ns**
Max Insertion Delay	2.031 ns	1.812 ns	1.714 ns	2.084 ns
Min Insertion Delay	1.358 ns	1.360 ns	1.286 ns	1.341 ns
Max Slew (20%-80%)	0.371 ns	0.724 ns	0.417 ns	0.743 ns

Table 5-2: Static Timing Analysis Timing Results from FFT (with Coupling Capacitors)

	Combined	
Skew	**0.736 ns**	(1%)
Max Insertion Delay	2.018 ns	(3%)
Min Insertion Delay	1.282 ns	(5%)
Max Slew (20%-80%)	0.348 ns	

Table 5-3: Spice Simulation Timing Results from FFT (without Coupling Capacitors)

	Tier A	Tier B	Tier C	Combined
Skew	0.464 ns	0.306 ns	0.317 ns	**0.502 ns**
Max Insertion Delay	1.592 ns	1.472 ns	1.406 ns	1.626 ns
Min Insertion Delay	1.128 ns	1.166 ns	1.089 ns	1.124 ns
Max Slew (20%-80%)	0.261 ns	0.497 ns	0.328 ns	0.497 ns

Table 5-4: Static Timing Analysis Timing Results from FFT (without Coupling Capacitors)

	Combined	
Skew	**0.446 ns**	(11%)
Max Insertion Delay	1.554 ns	(4%)
Min Insertion Delay	1.108 ns	(1%)
Max Slew (20%-80%)	0.250 ns	

the multi-tier analysis value of 1.341 ns by about 4%. Judging by the tier specific simulations, we should expect to see a clock skew near 0.745 ns (2.031-1.286). This value agrees with the combined tier analysis of 0.743 ns within 1%. We can check the slew rate in a similar way, 0.724 ns and 0.743 ns differ by less than 3%. Seeing as the merged tier

analysis makes sense in the context of the tier specific simulations, we can continue with the timing verification flow assuming we have accurate data. After performing the circuit simulations, we can move along to the static timing portion of the flow. Using the scripts described in detail in chapter 3, we can arrive at the clock tree performance numbers shown in table 5-2. We see that these numbers also agree with the simulation data, according to the percent difference values at the right, so we can proceed to viewing the timing reports to see if the chip meets timing. Regardless of the outcome of the setup and hold timing tests, we should note that having the timing data for the tiers individually gives us a relatively easy way to optimize the clock tree. Reexamining table 5-1, the first thing we notice is that tier B seems to have a clock sink with an abnormally large slew rate. The difference could be partially attributed to the fact that the slew rates are measured at slightly different points, but it would make sense to try and identify which clock sinks are causing this. If they are all on the same last stage clock buffer, perhaps that group should be split in two or that buffer should be resized. It is also clear that there is a bit of a discrepancy since either PrimeTime did not pick up on this skew, or it was erroneously found by Spice. The large slew can also be seen to a less drastic extent in the Spice analysis that ignores coupling capacitors. This is one of the reasons that performing the timing analysis once with simulation and once with static timing is recommended, to catch such problems. It also catches problems related to running static timing at a point with a parameter that is out of range. The standard cell data a static timing tool reads is characterized only for a certain range of conditions, and if a parameters steps outside of this range then the results are no longer accurate. Spice models are also not valid for all conditions, but typically are more accurate for a much broader range of conditions than are characterized libraries. Also, perhaps we can reduce the overall skew

of the design by looking back at the tier A clock tree. We can go back to the raw clock tree data file (mt0 file generated by HSpice) for tier A and see if the sinks that have an insertion delay over 2 ns can also be driven by stronger buffers. Finally, since tier C has both the fastest minimum and fastest maximum insertion delays perhaps we should add a buffer to the clock path that goes to tier C to bring its insertion delays more in line with those of the other tiers. If we want to tweak the clock tree in any way, we certainly need all of the information that we took the time to calculate. We can also examine tables 5-3 and 5-4 to find that without the coupling capacitors the circuit seems to run a little faster, but we still have adequate correlation between static timing and simulation data and see the same trends as we did when examining the coupling capacitor data. The scripts used here calculate slew by finding the absolute maximum insertion delay of all clock tree paths, then finding the minimum delay, and subtracting the two. However, while this is a good value to optimize and is indicative of the skew, it is an upper bound. There is likely to not be a register to register data path between every single pair of registers in the design, however we are essentially finding the maximum skew as if there were. Therefore, if the skew we find is intolerable, before redesigning the circuit it is necessary to see if there actually is a path between the two registers for which the skew was reported.

Table 5-5: Spice Simulation Timing Results for Data Paths

	Coupled SPEF file	Decoupled SPEF file	Default SPEF file
Long Path	2939 ps	2930 ps	2331 ps
Short Path	432 ps	433 ps	414 ps

Table 5-6: Static Timing Analysis Timing Results for Data Paths

	Coupled SPEF file	Decoupled SPEF file	Default SPEF file
Long Path	2860 ps	2860 ps	2220 ps
Short Path	380 ps	380 ps	370 ps

In tables 5-5 and 5-6 above two analyses of data paths are presented. There are actually many settings when extracting capacitances using First Encounter, but three

important ones are analyzed here. A coupled SPEF file is the most detailed, it extracts and reports coupling capacitances. A decoupled SPEF file extracts coupling capacitances, but then sets those capacitances to ground instead of between two signals. This usually yields results close to a coupled SPEF file; it is easier and faster to read in but cannot be used with tools that examine crosstalk. This file is often used for static timing analysis, seeing as most static timing tools set coupling capacitances to ground anyway. When extracting parasitics in the default mode, coupling capacitances are ignored. The long path analyzed in the table is completely contained within tier C. It consists, in order, of a NOR, AND, NAND, NOR, NAND, NOR, NAND, NOR, NAND, AOI, OAI, all feeding each other. The short path is simply a chain of 8 low power buffers. These results also show reasonable agreement between the simulation and static timing tools. We can draw a few conclusions from the results presented in this chapter, the first being that we have a valid method of performing 3D timing verification. This also provides some valuable insight to designing the optimal 3D clock tree. The clock tree created for the FFT was simple and required minimal designer effort, but because of the parasitic information we have, and the ability this verification flow gives the user to see into the design, we can optimize the clock tree's performance. Some ideas for this are presented as conclusions of this work in the following chapter.

Chapter 6: Conclusion

Reasons have been presented that the scaling of device feature size that we have enjoyed for roughly 30 years will not be able to continue forever. We have mentioned a few alternatives and briefly explained why a shift to 3DIC's is particularly favorable. Then we went over the limitations of commercially available chip design tools and how using them to design 2D chips presents some serious challenges. The EDA industry is comparatively slow to respond to new fabrication technology that becomes available, so our research group was to devise a method by which one could efficiently design and verify a 3DIC immediately, largely using design and verification tools already on the market. Thus far we have reviewed the normal chip design and verification process for a 2D chip, and stated that the intention was to model the 3DIC design and verification flow after this existing process as much as possible. We described the 3DIC design and verification methodology that our research group has developed, and concentrated on a way to design an optimal 3D clock tree. We looked at clock tree synthesis from a bit more of a theoretical standpoint, what makes 3D clock tree synthesis different from 2D, and some tradeoffs to consider. Following the explanation of this methodology, we dove into the 3DIC timing verification methodology in detail, which was the contribution and focus of this paper. After the chapter on the details of the flow, there was a section that helped to validate the flow by first explaining that parasitic extraction is fundamental to this work, then by explaining the details of how our parasitic extraction is performed and why it is accurate. We also went into how our design and verification flow was tested using the Winograd FFT. The architecture of the FFT was briefly described, as well as why it makes for a good test design.

Stacked IC's open up a whole new world of decisions for both the custom VLSI circuit designer and the logic designer. Using a 3D process for system on a chip (SoC) applications can drastically improve performance through clever floor planning techniques, such as allowing a multiprocessor core to utilize even larger memories without suffering larger wire delay penalties by placing a cache directly above or beneath the multiprocessor. I would go so far as to say that the potential for additional complexity on a single chip brought by 3DIC technology, is tantamount to the improvements superscalar and parallel processing bring to performance. There is a world of work to be done in the area of merging physical design and architecture techniques; other aspects of this same research project investigate the use of 3DIC's in the efficiency of crossbar and CAMRAM circuits.

There is still more work to be done in using this timing verification flow to determine the best clock tree structure for a 3D chip. It was because of time constraints that we used First Encounter to create a clock tree individually on each tier, connecting them at what was referred to as point 0 (at the perimeter). We did not have the time to write a clock tree synthesis tool from the ground up, nor did we have the time my theoretical analysis suggested it would take to see performance benefits from optimizing the clock tree by hand. There are a variety of clock tree structures that we could test using the existing FFT design, and compare them on basis of insertion delay, skew, and slew rate. Sometimes clock trees are designed using a grid structure, which we have not yet investigated. It would be interesting to actually insert various clock trees into the FFT design and compare performance when they have been placed and routed. There are also a number of different algorithms that have been published for creating clock trees, but many of these will be difficult to test in an actual design. Perhaps we can devise some sort of benchtest using the

3DIC timing verification flow to evaluate the results of a different clock tree insertion algorithms. The design and verification methodology described in detail here has been used in its entirety on one occasion here at NC State, and is currently in use in whole or in part at a number of other universities across the country.

Bibliography

[1] R. Joshi, and K. Roy. "Design of deep sub-micron CMOS circuits." *Proceedings of the 16th International Conference on VLSI Design.* Jan. 4-8, 2003, 15-16.

[2] V. Suntharalingam, R. Berger, J.A. Burns, C.K. Chen, C.L. Keast, J.M. Knecht, R.D. Lambert, K.L. Newcomb, D.M. O'Mara, D.D. Rathman, D.C. Shaver, A.M. Soares, C.N. Stevenson, B.M. Tyrrell, K. Warner, B.D. Wheeler, D.W. Yost, D.J. Young. "Megapixel CMOS image sensor fabricated in three-dimensional integrated circuit technology." *IEEE International Solid-State Circuits Conference.* Feb. 6-10, 2005, 356-357.

[3] Zhaoran Yan, Xiaojun Guo, Huazhong Yang, Rong Luo, Hui Wang. "An iterative delay model for interconnects with coupling effects." *Proceedings of the 5th International Conference on ASIC.* Oct. 21-24, 2003, Vol. 1, 311-314.

[4] OpenAccess Documentation and Source Code, available from Silicon Integration Initiative at http://www.si2.org/

[5] Synopsys PrimeTime product datasheet, available from Synopsys at http://www.synopsys.com/products/analysis/primetime_ds.html

[6] T.C. Edwards and M.B. Steer. Foundations of Interconnect and Microstrip Design. West Sussex, England: John Wiley and Sons (May 2000).

[7] Wayne W.-M. Dai. "Chip Parasitic Extraction and Signal Integrity Verification." *Proceedings of the 34th Design Automation Conference.* Jun. 9-13, 1997, 717-719.

[8] Cadence SoC Encounter product datasheet, available from Cadence at http://www.cadence.com/datasheets/socencounter_ds.pdf

[9] Ansoft Q3D Extractor product information, available from Ansoft at http://www.ansoft.com

[10] Cadence Fire & Ice QXC product datasheet, available from Cadence at http://www.cadence.com/datasheets/fireandice.pdf

[11] S. Winograd. "A New Method for Computing DFT." *IEEE International Conference on Acoustics, Speech, and Signal Processing.* May, 1977, Vol. 2, 366-368.