A FRAMEWORK FOR DESIGNING REUSABLE ANALOG CIRCUITS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Dean Liu
December 2003

Abstract

While the practice of design reuse is well established for digital circuits, it is not easily applied to analog circuits. One of the largest problems is that design constraints of analog circuits are sometimes implicit, which makes porting the design to a new environment difficult and prone to failure. This dissertation describes STAR (Schematic Tool for Analog Reuse), a system that captures designer's knowledge as part of the archival circuit representation, and then describes how this system can be used to create portable design modules.

Creating portable analog modules requires the system to capture not only the sized schematic of the circuit but also the objectives that the circuit is trying to achieved. It must also include the constraints on the cell's environment (for proper operation), and how these constraints should scale with technology. Furthermore, the system should help the designer in the current task of creating the design, since it is rare that a designer thinks about creating IP for someone else.

Our solution captures the design knowledge by enabling the circuit designers to annotate their schematics with special comments, called Active Comments. The designers embed predefined functions in the Active Comments to specify the goals and constraints of the circuits. An execution engine turns these comments into simulation runs to measure the circuit parameters and monitors to check the circuit's operating conditions. There are two types of active comments. One ensures the design meets the specifications, given some constraints on the operating conditions. The other checks that these constraints are satisfied for each instance of the circuit. Using the circuit's intrinsic properties to specify the constraints help make the comment portable.

We demonstrate the capability and utility of this system by examining the reuse of a phase-locked loop (PLL). Using the design knowledge captured in STAR, the PLL design is ported to a different process technology and re-optimized. The loop dynamics of the resulting PLL track the operating frequency, with the damping factor varying less than 12% across the frequency range of 500MHz to 1.2GHz. The framework also identified all the potential issues and verified the functionalities of the modified PLL without requiring any expertise of the designer. The ported design successfully operated at 1.2GHz.

Acknowledgments

Someone once told me that the road to earning a Ph.D. degree is long and difficult, and one cannot travel on that road alone. I am very fortunate to have the support and encouragement of a number of people during my time at Stanford, and I would like to express my gratitude to them.

First, I would like to thank my advisor, Prof. Mark Horowitz, for being a great mentor and the best advisor I could ever have. I am very grateful to him for sharing with me his keen insight and expertise, helping me see the greater picture, and having patience to my occasional digressions. It has been a privilege working with him these past years.

I would also like to thank Prof. Kunle Olukotun for being my associate advisor, serving on my orals committee, and reading this thesis. Thanks are also due to Dr. Stefanos Sidiropoulos for his thorough proofreading and the insightful discussions. I am also grateful to Prof. John Gill who served as the chair of my orals committee.

I have benefited greatly from interacting with the senior students in the Horowitz research group. In particular, helping Gu-Yeon Wei with his project introduced me to the area of high-speed links and phase-locked loop design, and discussions with Dan Weilader about the problems in the circuit design flow and in transferring design experiences resulted in the implementation of the CAD framework described in this thesis.

Friends and colleagues at Stanford have helped made my graduate school experience memorable. Specifically, I would like to thank Bob Kunz for taking so many classes with me during my first two years at Stanford and for teaching me the finer details of cache coherence protocols, Jaeha Kim for listening to my crazy ideas and offering his selfless support, and Haechang Lee and Elad Alon for being brave enough to be the first to try the tool and giving me valuable feedback. I am also grateful for Evelina Yeung, Bill Ellersick, Azita Emami-Neyestanak, and Ken Mai for giving me the opportunity to collaborate with them on their chip projects to broaden my circuit design experience. I cherish the friendships I made over the years, especially with David Harris, Bennett Wilburn, David Lee, Vladimir Stojanovic, Ron Ho, Fransçois Labonte, Samuel Palermo, Vicky Wong, Sarah Harris, Paul Hartke, David Barkin, Brucek Khailany, Ujval Kapasi, Ed Lee, and Andrew Chang, to name a few.

This research would not have been possible without the generous support of the C2S2 Marco Center. I would also like to thank the staff of both the Computer Systems Laboratory and the Center for Integrated Systems at Stanford: especially Charlie Orgish and Joe Little for their technical support and Teresa Lynn, Penny Chumley, Taru Fisher, Deborah Harper, Terry West, Darlene Hadding, Pamela Elliot, Ann Guerra, and Claire Ravi.

I am grateful to my family for their support and prayers, especially for my mother's unconditional love and words of wisdom. Even though my grandmother is no longer with us, I give my heartfelt gratitude to her for looking after me since the day I was born until the day she passed away. I also extend my sincere appreciation to my future family-in-law for their constant encouragement. It is impossible to find appropriate words to thank my fiancée, Jennifer Nee, who has been a great companion on this journey. She is an everlasting source of love, encouragement, and happiness to me. I could not have completed this thesis without her, and so I dedicate this work to her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Improvements in process technology have enabled us to integrate more transistors on a chip, leading to more complex circuits, while at the same time market pressures continue to push for shorter design times. To meet these constraints, designers must reuse proven designs and leverage them as building blocks in new designs. This kind of design reuse is well established for digital circuits. However, the same cannot be said about analog circuits. This thesis examines techniques for designing analog circuits to enable greater reuse.

Digital circuits are easier to reuse because their operation is modeled by boolean functions. In digital systems, discrete values are mapped into analog levels such that each signal is quantized with respect to the circuit's logic threshold to be either logical high or low. This quantization provides a large noise margin and allows robust operation. As a result, the circuit performs the correct function and behaves similarly under a wide variation in operating environment and signal amplitude. The noise rejection properties of digital gates, where small voltage errors are attenuated and do not affect functionality, allows one to ignore small differences in the circuit and environment. Static CMOS standard cells with gate inputs work well as reusable components because they have the largest operating tolerance. Even when the environment is very noisy, which is becoming more common in modern fabrication processes, or when the designer uses a less robust circuit family, circuit checking tools have been developed to ensure operating constraints are met [1][2][3]. For example, noise from coupling capacitance must be less than the margin of the gate that the wire drives. Tools now can estimate this noise and flag gates that might malfunction due to this noise. These tools estimate the margin of the receiving gates, and therefore allow less coupling noise when dynamic gates are used [4][5]. Overall, since the operating constraints and performance goals are similar for a wide class of of digital circuits, one set of constraints and checks can be applied to a large number of designs. This reuse of the checking software means that these tools have evolved over time and is one reason these tools are available for digital designs.

Unfortunately analog circuits are often linear, so noise at the input or coupled from the operating environment will directly affect the output. The key design challenge is to keep noise away from critical nodes and to reduce the noise sensitivity of these nodes. But both the specific requirements and the critical nodes are different for each analog circuit. Furthermore, the functionalities, goals and constraints of these analog circuits are often implicit. Essentially, a custom checker is required for each design. This checker needs to ensure that the circuit goals are satisfied and the circuit performs within specification under various operating environments. When we reuse an analog block, we can then apply its custom checker to ensure robust operation.

Without such a custom analog circuit checker, we would like the original designer to help with the reuse process. But most often this is not the case. Instead, a different designer inherits the design database, which consists of a set of transistor level schematics along with some simulation routines that are generally not well documented (i.e., the design knowledge is separated from the implementation). To help make the circuit into a reusable component, we need to capture the designer's knowledge into a custom checker and integrate that into the final design representation, so that the result of a design is both the current implementation and a custom checker for the circuit.

This thesis describes an annotated schematic capture system that focuses on the design reuse problem. It integrates into the schematic capture system a method of specifying the intended circuit functionality and goals as well as constraints on its environment.

Recently there have been a number of tools that attempt to help automate this analog design process, allowing the designer to state some of the optimization objectives at a higher level of abstraction and automatically generate some of the simulation control and measurement files that are needed. Since our system leverages some of these ideas, Chapter 2 describes these systems in more detail. To keep the design information synchronized with the archival representation, we use the idea of integrating the design knowledge directly into the circuit schematics. Finally to enable designers to capture a wide class of circuits, we use a scripting environment to specify the design information.

Chapter 3 describes our analog design system, STAR. This system introduces the notion of “Active Comments”, which are schematic annotations that control simulation files. The active comments are broken into two main forms - measurement comments, which are similar to other integrated systems, and constraint comments, which are like assertions in normal programs. Chapter 3 opens with a description of a phase-locked-loop (PLL) design to provide a concrete example of an analog/mixed-signal design that will be used throughout this thesis.

Having described the functionality of our system, Chapter 4 describes the prototype we implemented. Rather than building a fully integrated system, we opt for implementing an engine that can work with different schematic systems, simulators and analysis tools. The system is designed to be extensible and general to allow the user to capture new circuits, including ones that the tool or the tool designer has not seen before. In a large design group there may be a number of designers accessing the design database. In order to avoid conflicting variable and function names used in the Active Comments, we need to manage the name space to ensure that the correct data is being accessed.

Chapter 5 describes how we use STAR to design and reuse our example PLL and shows the result of using STAR to help port and re-optimize the design. Through reusing the PLL design, we explore the benefits and limitations of our approach and the performance of the prototype.

While not perfect, our approach seems promising. Chapter 6 summaries the advantages of our approach and issues that still need to be resolved in future systems.

Chapter 2

Background

To enable reuse, we want to capture the optimization goals and constraints for each analog block. To do this, we first need to understand the design process for analog circuits to see how the designers create the original design in the first place. Next we look at recent changes to analog design tools that aid the design process, since many have created techniques that we will use. One of the more interesting tools we will explore are analog circuit optimizers. These tools complete the design-validation loop for larger circuit blocks and are attractive components for future analog design systems. Our approach, STAR (Schematic Tool for Analog Reuse) focuses on how to integrate these features to make them attractive to the user. STAR is intentionally very flexible so it can capture a wide class of circuits.

2.1 Analog Design Process

Analog circuit design is usually an interative process alternating between circuit synthesis and design verification. During circuit synthesis, the designer chooses a circuit topology, formulates analytic equations to estimate the critical performance parameters of this circuit, and then adjusts transistor sizes as the result of local optimizations. While trying to meet the design constraints, there may be local subgoals that need to be achieved. For example, one objective of the circuit may be to have good power supply noise rejection. To accomplish this, the designer may want to have high impedance to the supply by adding a current source between the switching circuit and the supply rail. The designer may constrain the transistors that function as current sources to operate within their saturation region. In doing so, the objective of rejecting power supply noise is transformed into checking transistor saturation margin and output impedance. When this kind of reduction is done, a check of the real objective, power supply rejection, is also performed at the end of the design process.

Tightly coupled to the synthesis process is design verification which ensures that under all conditions the circuit meets the performance envelop set by the target application. Typically, the design verification is accomplished by using a circuit simulator like SPICE [6], or one of its derivatives. Running a circuit simulation requires three kinds of input - the circuit to be simulated, which is normally a sized circuit schematic, the simulation control file, which is the description of what inputs should be applied to the circuit, and the measurement file, which is the description of what data should be collected and how it should be analyzed. The algorithms implemented by these pre- and post-processing scripts contain the implicit optimization goals and heuristics that capture some of the designer's thought process and are a critical part of the design record. Often these additional files are lost, or poorly documented, so when another designer attempts to reuse this design, he/she needs to reconstruct the critical design issues and then recreate the control and measurement files. The net result of this loss of design information is that designs get more brittle with time. In order to enable design reuse, we want to capture these pre- and post-processing scripts as part of the design's archival representation.

2.2 Related Tools

Recently, a number of analog design capture and automation tools have been developed to address some of the problems with designing analog circuits. While these tools may not be specifically targeting the design reuse issues, they extend the design representation to include some of the design objectives. These tools can be broadly classified into two approaches: those geared to improve design capture and those geared to improve synthesis.

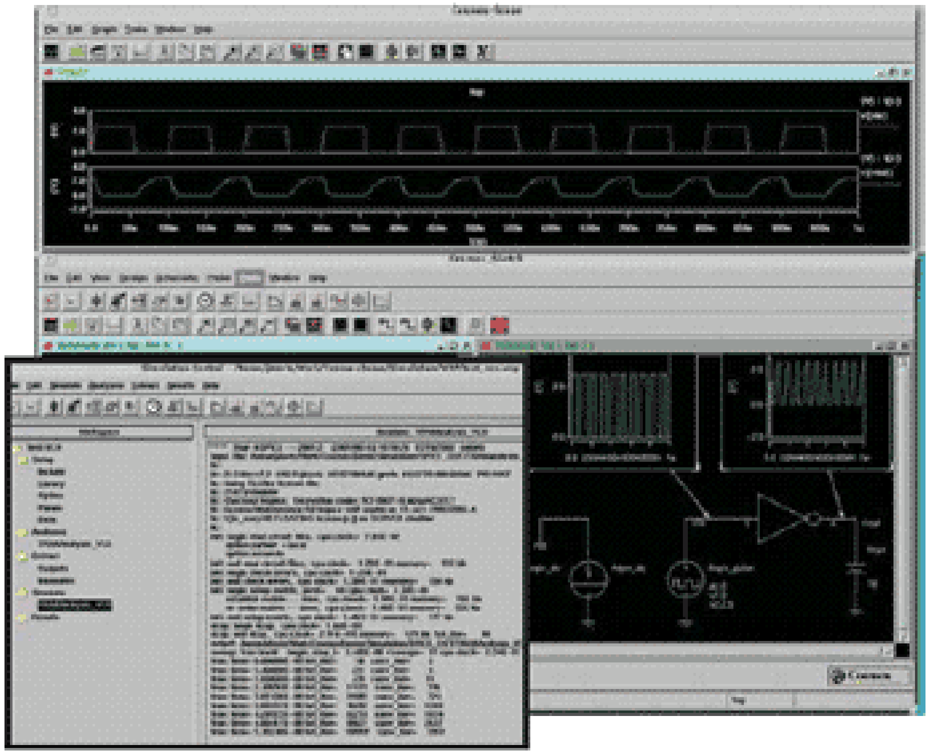


Figure 2.1: Screen Capture of Synopsys CosmosSE System

2.2.1 Design Capture System

The most commonly used design capture tool for analog circuits is schematic entry. With this tool, the user draws the devices in an electronic canvas to create circuit diagrams which provide a visual representation of the circuits. Recent design capture systems extend this approach to enable the designer to specify pre- and post-processing routines and optimization algorithms as part of the design database.

These modern capture systems take a more integrated approach to the design process to preserve some of the designer's knowledge in the archival representation. They merge the schematic entry, the circuit simulator, and the results analyzer into one environment to allow the designers to specify the stimuli and analysis routines directly in the schematics. The Synopsys CosmosSE[7] is an example of one such integrated system, and a screen capture of the tool is shown in Figure 2.1. The user selects some voltage and current sources from a library of stimuli and instantiates them onto the schematic to stimulate the circuits.

The schematic controls the circuit simulation through a series of pull-down menus. The results analyzer and signal waveforms are added to the schematic diagram. For example, to measure the common mode gain of a differential amplifier, the user would connect the inputs of the op-amp to a voltage source and configure it to sweep the common mode level. After the simulation, the user can then direct the analyzer to plot the output voltage versus the input common mode. All these steps are embedded in the schematic. Agilent's ADS[8] extends this integrated approach further by allowing the user to cascade the measurements from different schematics such that the results from one simulation can be accessed and used in a different simulation.

The Cadence Analog Design Environment[9] extends this integrated approach even further by providing a scripting language along with a set of pre-defined analysis functions to make it easier for designers to specify the pre- and post-processing routines and customize the simulation runs. For example, the user can dump the measurement routines from the common mode gain schematic and create a script to specify multiple operating conditions and transistor corners to measure the gain across these variations. The user can build upon the pre-defined functions to create their own routines. It is the ability to create reusable scripts that gives this tool its power, and we use this capability in our system.

These design capture systems make it easier to specify and archive the pre- and post-processing routines by embedding them as part of the design representation. However, when these circuits are instantiated in a higher level of the design hierarchy, only the circuit topology and sizes are propagated to the top level, but not the checks. Consequently, some of the knowledge that we are trying to capture is lost. In the next chapter we will describe how to extend these integrated systems to propagate the circuit's constraints and monitors along the design hierarchy to ensure that all instances of the design are operating within specification.

2.2.2 Automatic Analog Synthesis Tools

The main goal of the synthesis phase is to size the transistors in a given topology to meet the specifications. A number of analog synthesis tools have been developed to automate this process by performing complicated multi-variable optimizations. These approaches can be classified into three categories: knowledge-based, equation-based, and simulation-driven.

The knowledge-based systems [10][11] presented in the 1980s were the first generation of automated analog synthesis systems. In an knowledge-based system, there is usually a library of analog cells where designs can be either flat as in the IDAC system [10] or hierarchical as in OASYS [11]. Each cell in the library has its own hand-crafted design plan. These plans, containing the analytical equations and the manually derived and prearranged design strategies, are encoded in a computer-executable form. The sizing is done by executing a prearranged design plan.

As these analytic equations became more complex, more research was devoted into solving the equations more efficiently. In an equation-based sizing tool, the circuit parameters and specifications are written in the form of analytic design equations. These equations can be either derived and ordered manually as in OPASYN [12] and STAIC [13] or derived automatically using symbolic simulation techniques [14][15]. The tools then apply heuristic approaches such as simulated annealing or genetic algorithms on these equations to optimize the circuit. Recently, it has been shown that certain CMOS analog circuits can be approximated using posynomial equations. These posynomial equations can be solved exactly in seconds using convex optimization techniques. The downside to these powerful tools is that the specification is more complex since it must be formulated to create a convex set of constraints, making it harder for designer/users to add new optimization scripts. For example in convex optimization [16][17][18], the analog circuits' objectives and constraints must be in a specific form for the optimization to work. This task is currently done by experts within the vendor company.

In the simulation-based analog synthesis approach [19][20][21][22][23], these tools do not evaluate any analytic equations. Instead, they leverage numerical simulators to predict the circuit behavior. These tools use heuristic optimization algorithms such as stochastic pattern search, simulated annealing, and/or genetic algorithms to find an optimal solution [24]. However these optimization techniques require a large number of iterations. To reduce the run time, the solution search and optimization are performed in parallel across a pool of workstations [22][23]. In [23], downhill optimization algorithms are used to reduce the number of search steps. While simulation-based optimization can handle more general constraints, care is still required in the problem setup, and the long runtime makes debugging more difficult.

All three synthesis approaches are very powerful optimization engines, and they enable faster exploration of the design space and quicker closure of the design loop. The user specifies the goals and objectives of the analog circuit as the optimization criteria. While these tools focus on the circuit optimization problem, we are interested in finding ways to enable designers to specify their design objects in a more flexible manner to capture a wide class of circuits. To a large extent, our goals complement the synthesis tools since we are capturing and documenting analog designs while the synthesis tools are designed to optimize what we capture. In the future, we wish to incorporate these powerful optimization engines into the our tool to form a complete system for designing analog circuits.

2.3 STAR

Our goal is to be able to design analog circuits and encapsulate the design information as part of the circuit representation such that the circuit becomes a portable module. We built a design capture system, called STAR (Schematic Tool for Analog Reuse), to make it easier for a designer to encode constraints for a new circuit. From the capture tools, we leverage the notion of an integrated design document, but we loosen the requirement of having an integrated tool set. We extended the scripting approach to provide a more general way to specify the pre- and post-processing routines. The system allows the designer to create evaluation scripts at a higher level by providing pre-defined functions that cover common tasks and has mechanisms to enable the designer to create new functions. The user specifies these routines directly on the schematic as comments. With the routines tied to the schematic, it is easy to provide a regression capability to help ensure that the test remain current. In addition to testing the cells in isolation, some of the checks can be passed along with the schematic for each instance of the sub-cell, thus ensuring these constraints hold for each instantiation of the circuit.

We found these schematic-based annotations generally fall under two classes. One is the traditional optimization/measurement comment, and the other represents constraints on operating conditions. We describe our system in the next chapter.

Chapter 3

Active Comments

A widely used approach for annotating an analog design is to add comments to the schematics. Using comments to clarify one's intention is commonly done in many applications. For example, programmers insert comments into their computer programs to give readers some insight into the purpose of the code as well as the programmer's intentions. One problem with writing these comments is that once they are written, they are not always updated to reflect the latest implementation. Our approach to address this problem is to make the comments part of the implementation. In the context of the analog design process, we would like to turn these comments in the schematics into executable code to automate the generation of simulation and data collection scripts. This way, in addition to annotating the design, these comments also aid the designers, thereby offering incentives to the designers to use these comments in the first place. We call these comments Active Comments.

Turning the comments into executables require some constructs to specify the process of simulation and data gathering needed for its implementation. Since it is difficult to create all the possible routines one would ever need for the simulation control and results analysis, our goal was to create a flexible framework which can be extended by user specified routines. These user supplied extensions can then be archived along with the annotated schematic, enabling design reuse.

Having a system that automatically generates the pre- and post-processing routines from user annotation does not mean that the generated routines can be used with a different process technology. Any process specific parameters used in the routines and design constraints would make these comments non-portable. We need to capture these constraints and code the pre- and post-processing functions in a process independent manner in order to help make the analog designs into portable modules.

Section 3.2 describes the basics of Active Comments, followed by a detailed description of the two different types of comments in Section 3.3 and Section 3.4. We examine the issues with making the Active Comments portable in Section 3.5. Since a phase-locked loop (PLL) is used as a concrete analog/mixed-signal circuit example in this and later parts of the thesis, we start this chapter with a brief review of PLL design.

3.1 Phase-Locked Loop Design

Phase-locked loops (PLL) are often found on contemporary digital and mixed-signal VLSI systems. For example, PLLs are used as clock generators in all the microprocessors designed today and in many I/O subsystems. The main goal of a PLL is to create an internal clock with a precise timing relation relative to an external reference. While there are many design objectives for a PLL, the primary ones usually focus on the quality of the generated clock, specifically, its phase offset and jitter. Phase offset is the “DC” error in the timing of the output clock and jitter is the “AC” timing noise. Many circuit parameters affect phase offset and jitter, and we must design the circuits to minimize their effects.

Figure 3.1(a) shows a general structure of PLL, which consists of three main blocks: a phase comparator, a low-pass filter, and a voltage-controlled oscillator (VCO). The VCO generates a periodic signal, ck. This clock signal is fed back to the phase comparator to be compared to an external reference. The comparator measures the phase difference between these two periodic signals and outputs a signal to indicate the error. The low-pass filter converts this phase error to a change in the control voltage which modulates the phase and frequency of the VCO generated clock. If ck lags the reference, the control voltage is adjusted to increase the VCO frequency such that ck advances its phase. Conversely, if ck leads the reference, then its frequency is reduced to retard its phase. The control voltage is adjusted until no phase error is detected. Notice that we are adjusting frequency but comparing phase. Since phase is the integral of frequency, the output phase of the VCO is proportional to the integral of the control voltage and introduces a pole in the frequency

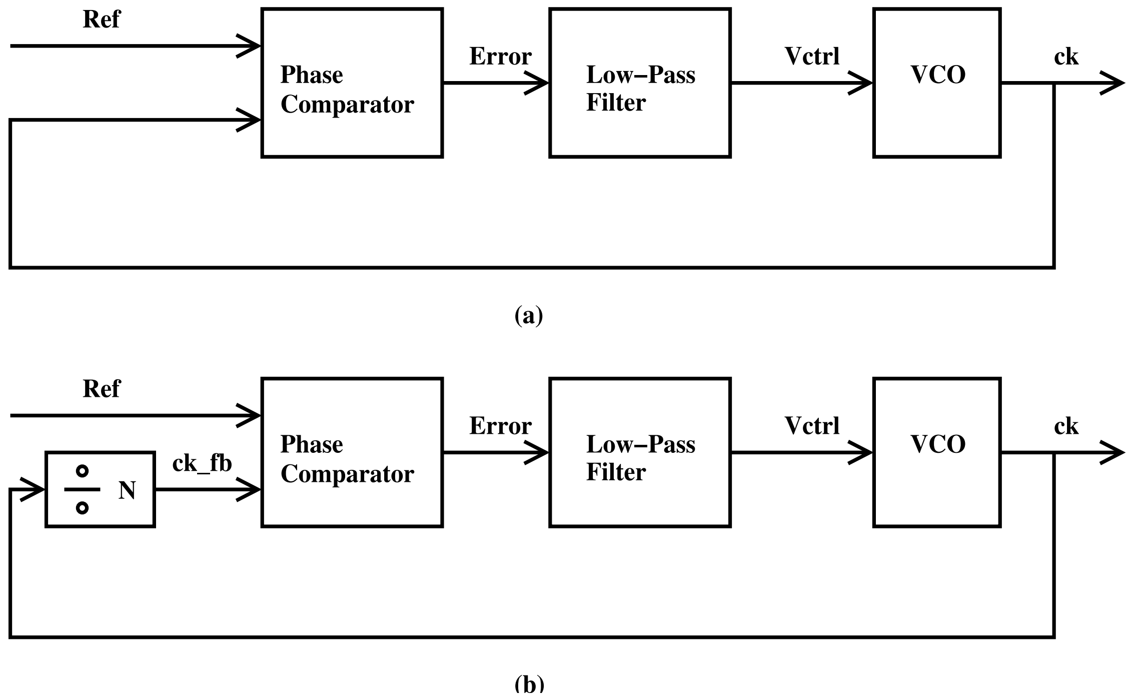


Figure 3.1: General Phase-Locked Loop Structure (a) Basic Blocks (b) with Divider for Frequency Multiplication

domain transfer function. In order to achieve zero phase error, the loop filter introduces another integration into the feedback loop [25]. As a result, the closed loop feedback system is often linearized and modeled as a second order system. The transfer function contains two poles at the origin, and thus requires a zero before the unity gain frequency in order to improve the phase margin and stabilize the loop. This zero is usually implemented within the low-pass filter along with one of the integration poles.

With the phase comparator detecting no error, the generated clock and the reference must have the same frequency. We can extend this architecture to enable frequency multiplication by adding a frequency divider in the feedback path, as shown in Figure 3.1(b). The divider makes the VCO output N times higher in frequency than the reference and feedback inputs, thus allowing the PLL to perform frequency multiplication.

Like most modern integrated circuits, each of the PLL sub-blocks is hierarchical and is composed of one or more analog/mixed-signal components. There are many approaches for building each of these elements. The VCO can be based on LC circuits [26][27], multivibrators [28], or ring structures. For our example we will use a ring based oscillator since it is the most flexible design and can be operated over the widest frequency range. This type of oscillator is composed of identical delay elements cascaded in a ring configuration with inverting feedback between the two elements that close the ring. A ring oscillator can typically generate a wide range of frequencies with a linear relationship between frequency and control voltage. Unfortunately, it is also has higher noise sensitivity than some of the other topologies. Any high frequency noise coupled to the VCO is not corrected by the loop and directly affects the quality of the clock, since the feedback loop has finite bandwidth. Therefore, care is needed to ensure supply noise does not couple into the VCO.

The phase comparator compares the phase of the the generated clock to that of the external reference. The comparator can be implemented using a XOR gate [29], a SR latch [30], or D flip-flops [31][32]. In a PLL application, the phase comparator needs to detect both phase and frequency differences because the feedback clock and the reference can be at different frequencies when the loop starts up and is not in lock with the external reference. In our implementation, we use a D flip-flop based design that detects both phase and frequency errors. This detector is commonly called a phase-frequency detector (PFD). Figure 3.2 shows the block diagram of the PFD with its timing waveform. The PFD compares the rising edges of the reference and feedback clock to output a pair of pulses, up and $down$. The rising edge of the feedback clock asserts the up signal, and the rising edge of the reference clock asserts the $down$ signal. After both outputs have risen, the PFD self-resets to de-assert both outputs which aligns the falling edge of the outputs. If the reference is early, then the up pulse will be wider. On the other hand, if the feedback clock is early, then the $down$ pulse will be wider. Ideally, the difference in the widths of the two output pulses equals the phase error and is zero when the two signals are aligned. To avoid introducing any phase offset, the signal path of the reference clock and that of the feedback clock must be identical with matched input edge-rates.

The low-pass filter converts the phase error detected by the PFD to a change in control voltage to modulate the VCO. This task is generally accomplished using three circuit blocks: a charge-pump, a capacitor, and a resistor. Together the charge-pump and the

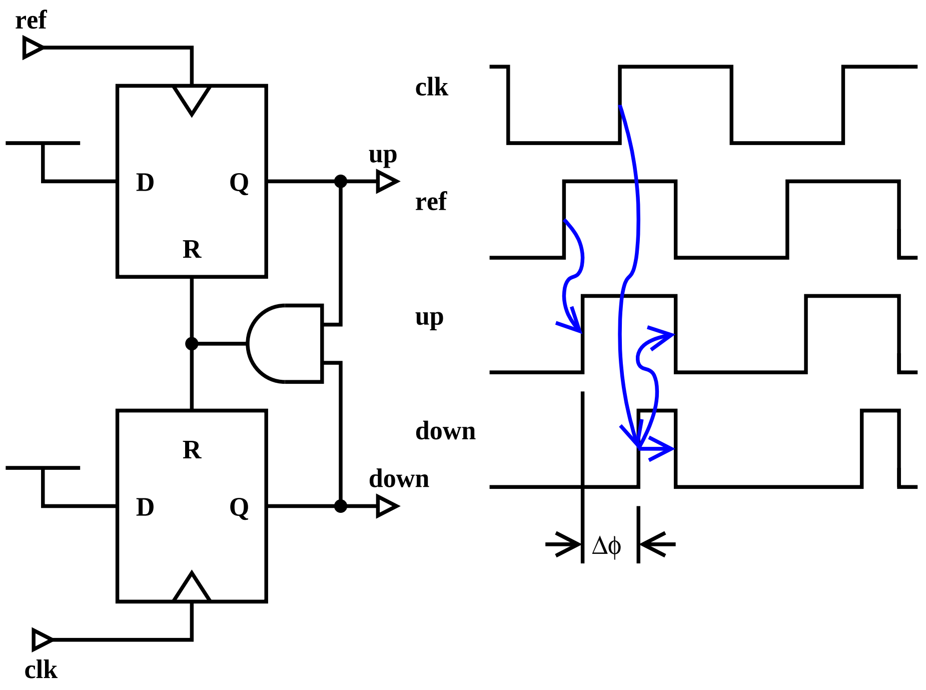


Figure 3.2: PFD Model and Timing Diagram

capacitor achieve the error-to-voltage conversion by adding or subtracting a charge proportional to the phase error onto the capacitor [25]. The resistor forms the zero that stabilizes the PLL. This resistor can be implemented using a passive element in series with the capacitor. In this case, the control voltage is the sum of the instantaneous voltage formed by the current through the resistor and the voltage integrated across the loop filter capacitor. Alternatively, active elements can be used to form an effective resistor in a feed-forward manner [33][34][35][36]. This latter approach adds an extra copy of the charge-pump current directly to the bias current used to control the VCO. By setting the the capacitance and resistance of the RC low-pass filter, the designer determines the frequency of the zero. These values must be carefully chosen to ensure loop stability and a high quality output clock.

The schematics of these analog components are archived in the design database. We call this set of schematics the production schematics because every transistor in this set has a corresponding layout in the cell being created. In addition to these schematics, the database also contains another set of schematics that are used to help characterize and verify

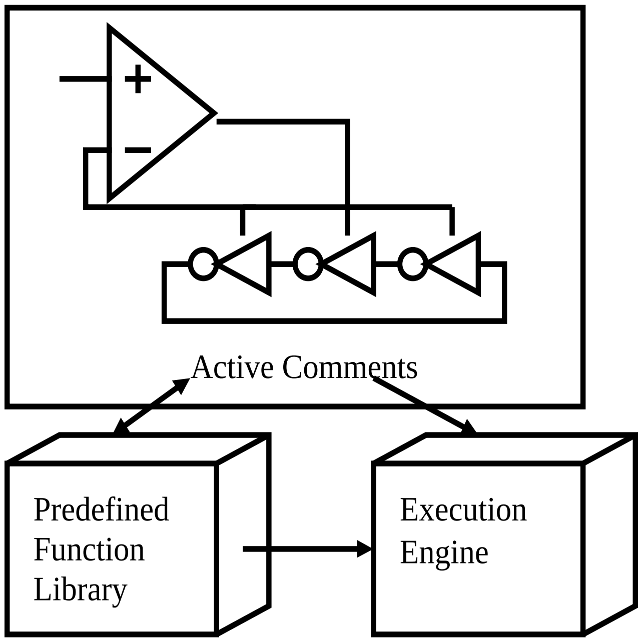


Figure 3.3: Each Active Comment is Composed of Pre-defined Functions and Processed by an Engine

both the components and the overall system. While the designers can measure some of the circuit parameters directly from the production schematics, most often, individual analog cells or groups of cells are used with the addition of some type of test scaffolding. This scaffolding might directly drive some internal nodes, add explicit noise to some nodes in the environment etc. to help characterize the circuit. For example, one cannot simulate the VCO to find how fast it will oscillate after it is integrated into the PLL since the feedback will lock it to the input clock. Instead, it is easier to simulate the VCO in isolation with its bias circuit and the estimated load to measure these parameters. The designer would construct a schematic to include these elements, which is an example of a "test bench" schematic. A complete design database would include many of these test bench schematics along with the set of schematics needed to build the production PLL. How our system deals with these schematics is described next.

3.2 Active Comments

The basic operation of the Active Comments is shown in Figure 3.3. It starts with writing the comments into the circuit schematic. The designer composes these comments out of a set of predefined functions from a library, and an execution engine processes these comments to generate the necessary simulation and analysis routines. We have developed two types of Active Comments: Measurements and Assertions. The Measurement Comment sets up the simulation runs and measures the circuit parameters. It also allows the user to formulate analytical expression and evaluate the expression based on the simulation results. The Assertion Comment monitors the circuit's environment to ensure that the design constraints are not violated.

Where the comments are added and how they are used is based on the how the schematics are used. To characterize a design or measure a circuit parameter, the schematic being used is always the top-level or a test bench schematic. So the Measurement Comments are written into these top level schematics. On the other hand, we need to ensure that all the sub-circuits are functioning correctly whenever they are used, so the Assertion Comments can be placed in the production schematics of any cell that needs to be checked.

The following sections examine these two types of comments in more detail, starting with the Measurement Comment.

3.3 Measurement Comment

In most analog and mixed-signal designs, there is a broad range of parameters that needs to be optimized, such as: voltage, current, delay, pulse width, output impedance, bandwidth, gain, etc. While the schematics clearly show the topology of the solution, most of the parameters that the designer is interested in are the result of the operation of the circuit, and are not directly apparent from just looking at the topology. Measurement Comments allow the designer to explicitly record both the parameters that are important for this circuit, and what tests should be run to measure these parameters. Thus by using Measurement Comments, a designer can write equations and place them in the schematic instead of creating a number of simulation files to extract the parameters of interest.

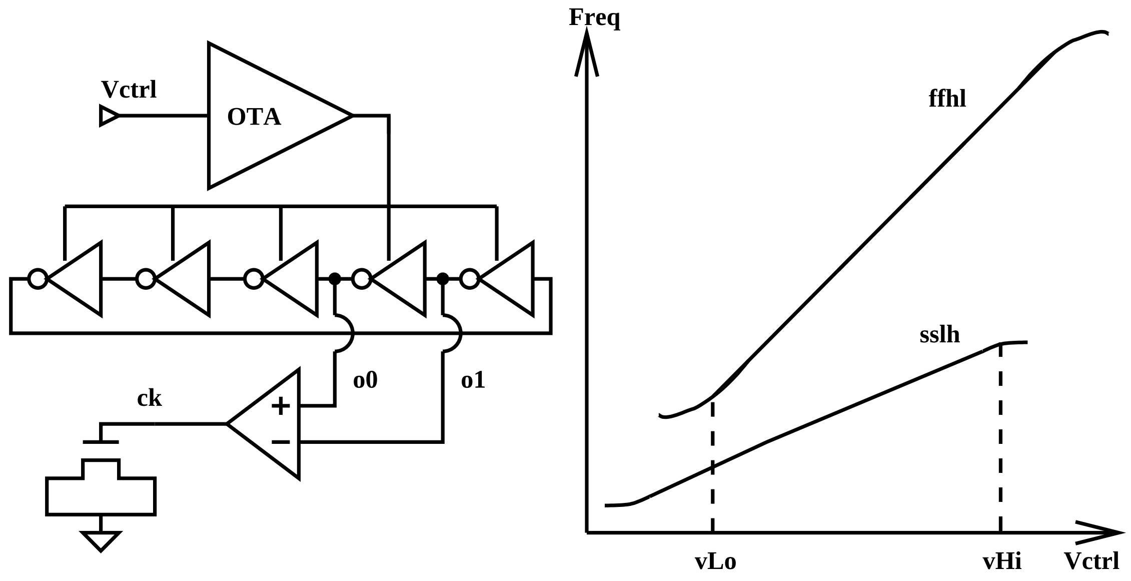


Figure 3.4: VCO Test Bench Circuit and Transfer Curves at Two Extreme Operating Conditions

We use the VCO to demonstrate the use of the Measurement Comments. Figure 3.4(a) shows the test bench of the VCO, which consists of a five-stage inverter ring, an operational transconductance amplifier (OTA) to convert the control voltage $Vctrl$ to a drive current for the oscillator, a level shifter that amplifies the small swing VCO clock to full CMOS level, and an estimated load. As $Vctrl$ increases, the oscillation frequency also increases. Figure 3.4(b) plots the oscillation frequency vs. control voltage transfer curves. Since integrated circuit manufacturing has some variability, a designer needs to test the circuit over a range of manufacturing variations. This leads to simulating the circuit at the extreme points of the manufacturing distribution. These extreme points are usually called corners. Figure 3.4(b) shows simulation at two corners; one when all transistors are as fast as they can be, and the other is at the slow corner

The linearized PLL model assumes a linear relationship between the VCO oscillation frequency and the control voltage. However, the slope of the transfer curve (Figure 3.4(b)) flattens at the extremes of the control voltage because devices in the OTA go out of saturation. Since the VCO frequency is linear with $Vctrl$ only within a range of voltages, the VCO operating range must be limited to the linear portion of the transfer curve to keep the PLL model valid.

To find the frequency (and the corresponding control voltage) limits of the VCO, we need to take the most conservative boundary by simulating the circuit at the extreme corners. We determine the lower frequency limit with the fast transistor model and the high limit with the slow transistor model. Bounding the range of the control voltage this way guarantees that the VCO frequency is linear with respect to $Vctrl$ across all process and operating corners. The upper and lower bounds of the voltage range are used as control parameters in all of our PLL simulations.

Measuring the desired parameters, like the VCO range in the previous paragraphs, can require analyzing the data from a number of simulation runs. In general these simulation runs require different types of information. First the simulation needs to read some global parameters that used in a set of different tests. For example these might be the initial state of the circuit, simulation parameters, and/or simulator settings. Once we have the global information we next need to specify what simulations need to be run, and then what analysis routines we want to run over the direct simulation output. Finally once all the calculations are done, we would like a way to report these processed results either back to the user or to the program to be used in future calculations. To accomplish all these tasks STAR provides four types of Measurement Comments: GLOBAL, DEFINE, CALCULATE, and REPORT. Since most of these tasks are performed in the define comment, the next section starts by explain how this comment works. It then explains the function of the other comments.

3.3.1 Simulation Setup

STAR provides the DEFINE statement to enable users to specify the simulation and analysis routines. The statement consists of three parts: the first part specifies what simulation to run and what operating conditions to use, the second part describes how to analyze the simulation results, and the third part allows one to name some of the results so they can be used for other routines.

For example, to find the upper and lower bounds of the control voltage we can sweep the control voltage across the entire supply range, plot the transfer curve, and then find the inflection points. All this can be done in a DEFINE comment:

```
# DEFINE vLo=LoRange(vctrl,Freq(ck)) w/ SweepV(vctrl,gnd,vdd) @ ffhl
# DEFINE vHi=HiRange(vctrl,Freq(ck)) w/ SweepV(vctrl,gnd,vdd) @ sslh
```

We use keywords, $\# DEFINE$, $w/$, and $@$, to separate the different fields in the comments, and the fields are processed in reversed order, starting from the right end.

The four letter symbols $ffhl$ and $sslh$ explicitly name the type of transistors and the operating conditions under which the simulations are run. Each letter denotes the setting of an item. Together the four letters specify the speed of the PMOS and NMOS transistors, the supply voltage, and the operating temperature, respectively. In general, f means fast, s slow, l low, and h high. The letters $ffhl$ correspond to the operating condition where the transistors are operating the fastest with fast PMOS and NMOS transistors, high supply voltage, and low junction temperature, while $sslh$ specifies the opposite condition. With four letters, $ffhl$ ($sslh$), the comment specifies the fastest (slowest) operating conditions to find the lower (upper) inflection point of the curve. If this field is omitted, then the simulation defaults to the typical condition, $tttt$. Appendix A describes how users can override the default settings.

To the left of the operating condition are the two parts in the comment that specify the simulation stimuli and the analysis procedures. The stimuli are generated in a preprocessing step, and the analysis procedures post-process the simulation results. The function $SweepV()$ is a pre-processor that directs the simulator to sweep the control voltage, $vctrl$, from gnd to vdd. $Freq()$ is another pre-processor, and it uses the simulator's measurement capability to find the frequency of ck. The simulator's measurement command contains a reference name to identify the measured value in the simulation results, and this name is return by $Freq()$. The post-processors, $LoRange()$ and $HiRange()$, use this reference name to access the oscillation frequency in the simulation outputs, find the lower (upper) inflection point of the VCO transfer curve, and return the corresponding voltage.

The post-processing results are the values of the critical parameter that are specified with the $\# DEFINE$ statement. We assign the results to variables so that these values can be used by other routines. In general, these variables can either be scalars or arrays. The datatype of the variables is not declared a priori. Instead, the variables are casted based on the return value of the post-processor. In this example, the post-processors, $LoRange()$

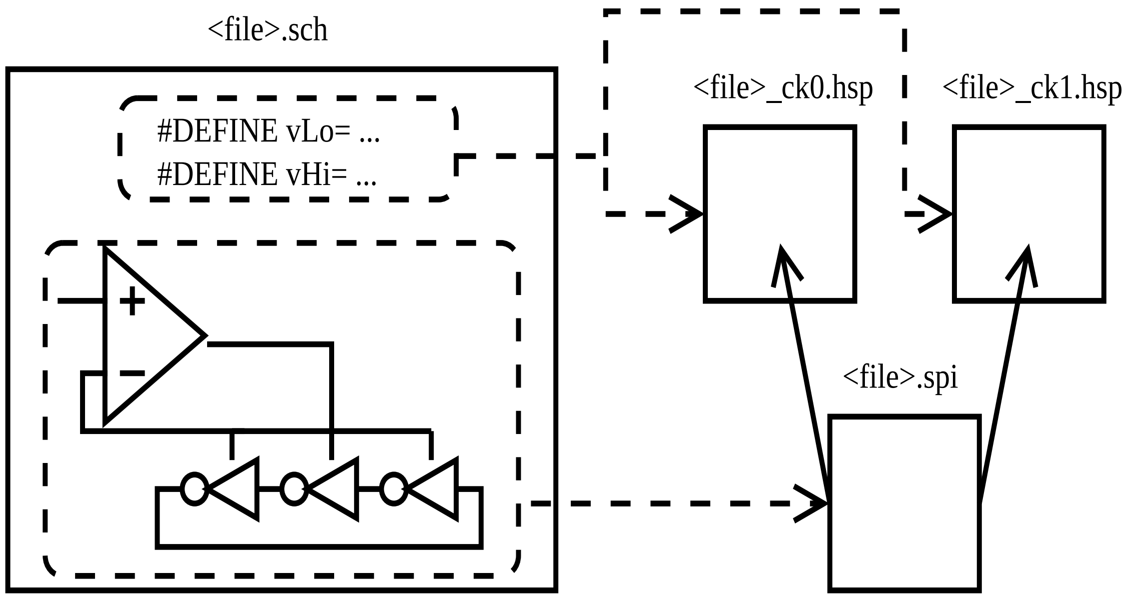


Figure 3.5: The Schematic Produces One Device Netlist (.spi) and Each #DEFINE Generates a Checker File (_ck0/1.hsp)

and $HiRange()$, return the lower and upper bounds of the control voltage, respectively. The return values are assigned as scalars into two variables, vLo and vHi, which are used to control other PLL simulations.

Normally when a user creates a netlist from a schematic, a single file is created which contains information about the devices shown in the schematic. Adding Active Comments increases the number of files produced during this operation. Each $\# DEFINE$ statement in the schematic is really a set of instructions for a simulation run plus a post processing step, so each generates it own control file, as shown Figure 3.5. For the control voltage range measurement explained in the previous paragraphs, the test bench schematic is named “vcoV2,” and the corresponding netlist is named “vcoV2.spi.” From the two $\# DEFINE$ statements, two simulation decks, “vcoV2_ck0.hsp” and “vcoV2_ck1.hsp,” are generated. Figure 3.6 shows the HSpice stimulus deck generate by the comment that defines vLo. The simulation corner specification, $ffhl$, in the comment are turned into the fast-fast transistor corner, 0-degree simulation temperature, and 110% of the nominal supply voltage in lines 4, 6, and 9 in the HSpice deck, respectively. The $SweepV()$ function generates the parameter declaration in line 17 and the sweep command in line 19. The $Freq()$ function

```
* checker simulation deck
***** h e a d e r   b e g i n ********************
.prot
.lib '/home/dliu/lib/spice/tsmc-0.35/mosis.lib' ff
.unprot
.temp 0
.opt post accurate
.option post_version=9007
.param vddval='1.1*3.3'
.param vlow=0
vdd vdd gnd dc vddval
.inc '/home/dliu/tool_test/pll_rc/vcoV2.spi'
.inc 'vcoV2_checker_set.hsp'
***** h e a d e r   e n d ************************

.param swp_vint=0
Vswp_vint        vint    gnd dc swp_vint
.tran 32p 160n uic
+sweep swp_vint 0 3.3 '(3.3-0)/20'

* generate a vdd/2 reference
efreq_ck vfreq_ck gnd vdd gnd 0.5
.meas tran per trig v(ck,vfreq_ck) val=0 rise=5
+              targ v(ck,vfreq_ck) val=0 rise=6
.meas tran freq_ck param='1/per'

.end

```

Figure 3.6: HSpice Stimulus File Generated from Measurement Comment

```
***** Global Parameter File **********************
.ic v(o0)=0
.ic v(o1)=vddval
```

Figure 3.7: HSpice Probe Command Generated from Assertion Comment

produces lines 21 through 25 to make two calls to the simulator's built-in measurement function to find the frequency of ck. The first measurement finds the period of the signal, and the second calculates its reciprocal to convert the period to oscillation frequency. The name of the second measurement, $freq\_ck$, is generated by the $Freq()$ function and passed back as its return value into the STAR system. This enables the post-processing routines, $LoRange()$ and $HiRange()$, to find the measurement results in the simulation output file. All the technology dependent parameters such as the supply voltage, the operating temperatures, and the location of the transistor model are all stored in a global parameter file that STAR accesses. The organization and implementation of the Measurement Comments are described in more detail in Chapter 4.

In addition to including the transistor netlist in line 12, the generated deck also includes a file called "vcoV2_checker_set.hsp" in line 13. This file contains the parameter and initial condition settings that are common to all the measurements and is included by all the decks generated from this schematic. The contents of that file is specified with a different type of statement which is described in the next section.

3.3.2 Shared Parameters

The $\# \; GLOBAL$ statement allows the designers to specify parameters that are shared among different simulations specified within the same schematic. These global parameters include voltage, current, initial values, and simulator parameters. A $\# \; GLOBAL$ statement is a declarative statement that consists of function calls to set the value of the parameter. We use the $\# \; GLOBAL$ statement to initialize the oscillator to a known state to make taking the measurements consistent and repeatable:

```
# GLOBAL InitV(o0,0), InitV(o1,vddval)
```

The $InitV()$ is a pre-defined function that initializes a node to the specified voltage. In this example, nodes $o0$ and $o1$ in the oscillator (see Figure 3.4) are initialized to 0 and Vdd, respectively. Figure 3.7 shows the generated parameter file, "vcoV2_checker_set.hsp," which contains the two initial condition HSpice commands. These global parameters are written into a file which is to be included in all the stimulus decks produced by the $\# DEFINE$ statements on the same schematic. With the $\# GLOBAL$ statement, designers only need to specify these parameters once and the values will be automatically placed in all the generated simulation routines.

Using $\# DEFINE$ and $\# GLOBAL$, the user can specify the simulation procedures to measure circuit parameters. The next section describes how these parameters can be manipulated after they are extracted from simulation results.

3.3.3 Analytic Equations

An important tool for analog circuit designers are the analytic equations used to help predict the circuit behavior. By manipulating the values of the parameters in the equation, the designer can very quickly estimate the performance that a parameter change will have, or conversely, how to set the parameters to achieve the desired performance. These equations offer insights to how the circuit behaves, and also how to optimize the circuit. The user can order these equations using the $\# CALCULATE$ statement which has a similar syntax to a C-style mathematical expressions. The equation contains the critical parameters defined in $\# DEFINE$ statements and the result is assigned to a variable. The statement is written in following form:

# CALCULATE VAR_NAME $=$ EXPRESSION

The $EXPRESSION$ is the user specified analytic equation that combines the parameters measured in $\# DEFINE$ statements. The flexibility and generality of the C math library allow the designer to code any analytic equation. The result of the $EXPRESSION$ is stored in VAR_NAME. Before evaluating the equation, STAR examines the size of each variable used in the $EXPRESSION$. If the critical circuit parameters in $\# DEFINEs$ are vectors, then $EXPRESSION$ is evaluated as a vector and its result is stored as a vector in VAR_NAME.

The $\# CALCULATE$ statements on the schematic are evaluated independently from each other. An $EXPRESSION$ can include parameters defined in the $\# DEFINE$s on the same schematics and other variables evaluated earlier in other schematics, but it cannot use any VAR_NAME that are being defined by the other $\# CALCULATE$ statements on the same schematic. Furthermore, if a VAR_NAME is being defined by more than one $\# CALCULATE$, there will be a name collision. This condition must be avoided by the user since the current version of STAR does not check for name collision errors.

3.3.4 Reporting Results

The variables holding the measured circuit parameters and evaluated expression results are stored internally within the execution engine. If the designer wants to view them in a plot or a report, or to store them in the database for other Active Comments to use, he/she needs to explicitly specify the post-processing functions to manipulate the data. STAR provides the $\# REPORT$ construct to enable designers to specify how the final values of the parameters are used. Similar to the $\# GLOBAL$ statements, the $\# REPORT$ statements are declarative statements consisting of a list of function calls. In the case of the VCO range example, one can use the reporting construct to export the vLo and vHi variables so other Active Comments can access them:

```
# REPORT Export(vLo), Export(vHi)
```

The $Export()$ function writes the value of the variables into a file so that other Active Comments can access them. This enables Measurements from different schematics to be cascaded.

3.3.5 Execution Flow

The Measurement Comments are declarative statements where the ordering between the different types of statements is important. The execution of the Measurement Comment do not follow the order in which the comments were written. Instead, the sequence of execution depends on the type of comment which follows the flow of information outlined in Figure 3.8. Data that is shared among the Measurements on the same schematic must be processed before the simulation decks are generated. The $\# GLOBAL$ statements set the

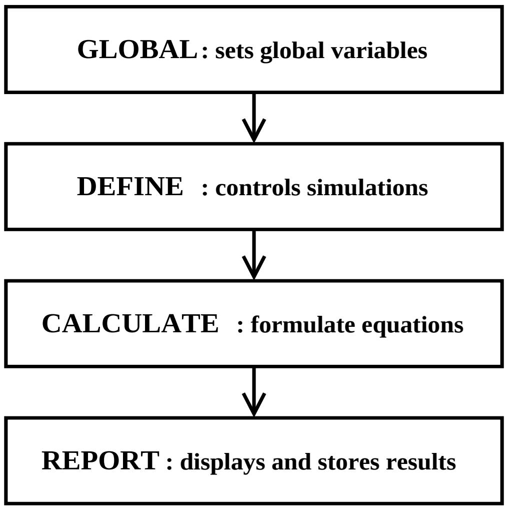

GLOBAL: sets global variables
DEFINE : controls simulations
CALCULATE : formulate equations
REPORT : displays and stores results

Figure 3.8: Execution Flow of Measurement Comments

global parameters used in all the automatically generated pre-processing routines, so they are processed first. Following that are the $\# DEFINE$ statements which generate the simulation decks and specify the analysis routines. Each $\# DEFINE$ statement is split into two parts; the pre-processing portion generates the decks used to control the simulation, while the post-processing portion is executed after the simulation results are available. After the critical circuit parameters are extracted from the simulation runs and stored into the variables declared in the $\# DEFINE$ statements, the analytic equations formulated in the $\# CALCULATE$ statements are evaluated. The, $\# REPORT$ statements are executed last to present the results in graphical or tabular form and to write the final results to the design database.

While the ordering between the different types of statements is important, the statements within each type on the same schematic are independent from each other. So, the sequence in which they are processed can be relaxed . In fact, it may be possible to execute them in parallel as a performance enhancement. On the other hand, Measurement Comments in different schematics may not be independent from each other since STAR enables the designer to use variables defined in one schematic to be used by Measurement Comments in other schematics. Cascading measurements this way naturally creates a causal chain. The ordering of these schematics is determined by where the variables are being defined. If any Measurement Comment attempts to use a variable that has not been evaluated, STAR would search through all the schematics in the database to find where that variable is defined and warn the user to execute that schematic before the current one can be used.

Sometimes Measurement Comments can be written in the production schematic of a cell rather than requiring a separate test bench. Since a Measurement creates an entire simulation environment, including the stimuli to the underlying circuit, when this circuit block is integrated into a higher level, its Measurements are not executed.

3.3.6 Measurement Comment Summary

Using Measurement Comments allows a designer to specify simulation runs and analysis routines in the circuit's schematic representation. Furthermore, it encourages the designer to keep the simulation scripts up-to-date with the circuit implementation since the designer is actively using these scripts. To provide further incentives, regression tests could be run on all circuits as part of the design to find stale files. Finally, circuit schematics convey relevant design information which other design tools might be able to leverage.

In fact analog synthesis tools would provide an additional capability for the Measurement Comments, since they automate the sizing of transistors in a given analog circuit by performing complicated multi-variable optimizations. Thus our plan is to integrate these synthesis tools into our system to create more powerful library operations, such as multi-variable optimization routines, for the designers to use, in additional to the simpler operations we have already implemented. For established blocks, there might be one optimize/measure comment that essentially generates the entire circuit.

3.4 Assertion Comment

In addition to measuring and optimizing the critical circuit parameters, a designer must also check that the circuits' environment always satisfy their operating constraints. For example, transistors acting as current sources need to be in the saturation region or delay matching circuits must maintain certain timing margin with the signal being tracked. Another example might be that when a circuit was tested, the designer assumed that noise on the analog circuit's bias voltage must caused less than a 1% variation on the current it produces.

While designing each component, the circuit designer must guarantee that the component functions correctly under the worst possible operating environment. However, as other designers reuse the circuit and integrate it into different parts of the system, the operating condition may change from what the circuit was originally designed for. To ensure that the circuit operates within its specified environment, we created the Assertion Comments to actively check the constraints placed on the circuit's operations and perform these checks each time the circuit is used. Therefore, these assertion checks must propagate from the component level to the system level. By embedding the Assertion Comments in the design, we tie the operating constraints to the design itself. So, the designer only needs to embed the checks on the properties of interest, and these Assertions will monitor all the properties in all the simulations, across the system hierarchy.

To illustrate how one can embed an Assertion Comment to check the circuit's operation, we use a charge-pump circuit as an example. The charge-pump adds or subtracts from the filter capacitor an amount of charge proportional to the phase error. A model of a charge-pump is shown in Figure 3.9(a), where the output of the charge-pump is connected to two current sources. The upper current source deposits charge to the filter capacitor (not shown) while upb is asserted, and the lower current source withdraws charge while dn is asserted. The amount of charge deposited or withdrew equals the product of the total current multiplied by the duration the switches are asserted. Note, the currents from the current sources are identical such that when both upb and dn are asserted, no net charge is added or subtracted. It is critical that the current level from each source maintains roughly constant during the time the corresponding control signal is asserted. And to avoid introducing any static phase offset or disturbing the control voltage, there must be no current flowing into or out of the charge-pump when both control signals are de-asserted or simultaneously asserted.

Figure 3.9(b) shows the circuit implementation of the charge-pump along with a zero-volt voltage source, $vmeas$, at the output node. This voltage source is used as a current

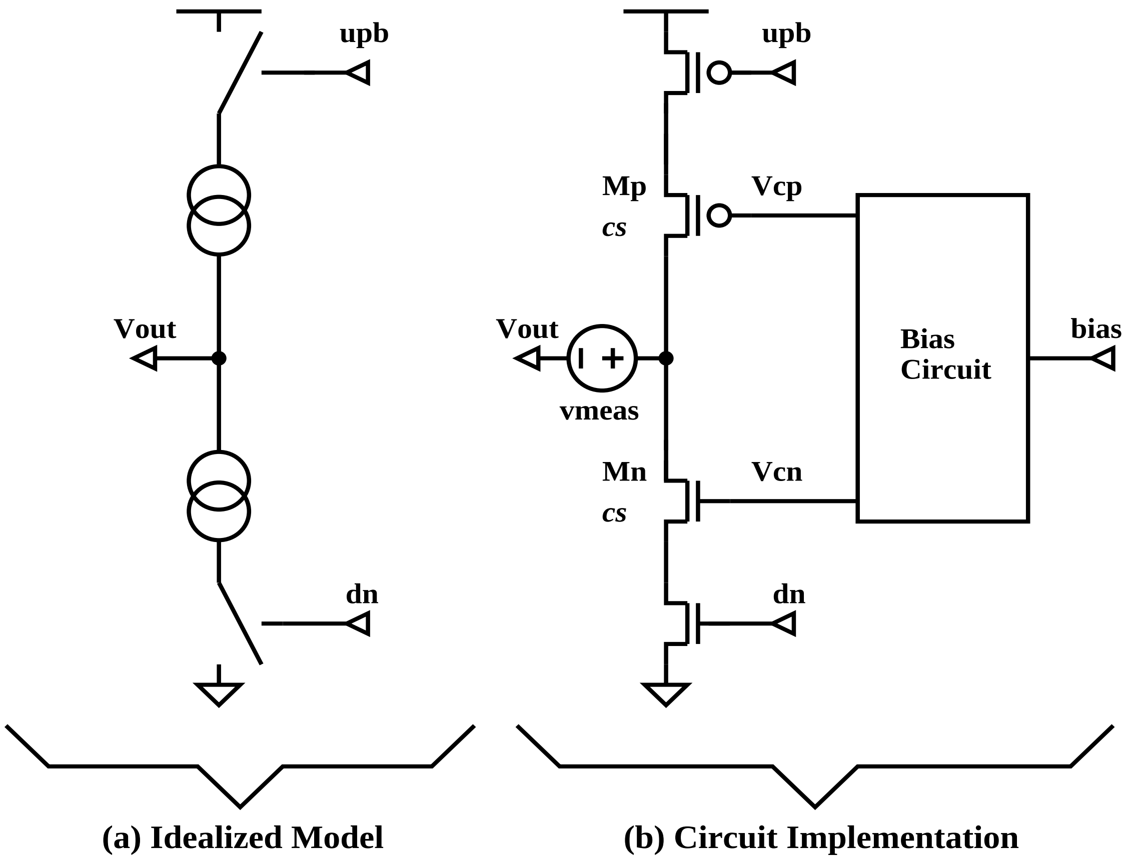


Figure 3.9: Charge-pump Diagram (a) Idealized Model (b) Circuit Implementation with Current Source at the Output

meter to measure the current flowing to and from the circuit. In the following sections we show how to use Assertions to check the circuit's operations to ensure the circuit's critical requirements are always satisfied.

3.4.1 Design Assertions

In Figure 3.9(b), the two inner transistors, Mp and Mn, implement the current sources in the charge-pump. In order for these transistors to function as current sources to produce constant current, they need to be operating in the saturation region under all the operating conditions in all simulation runs.

To check the transistor saturation, we need to measure the Vds and Vdsat of the transistor. For robust operation, the transistor must have a saturation margin, (Vds-Vdsat), of at least 5% of the supply. In this example, we mark the transistors that need to be checked with a label so a program can find the transistor in the netlist. We added a cs label, short for current source, to the transistor property .

The essential features for an Assertion are the specification of a routine to process the simulation data and the constraint that the circuit must hold under all operating conditions. The comment to check the saturation margin of transistors is as follows:

# ASSERT SatMargin(cs) $>=$ 0.05*Vdd

where the keyword $\# \ ASSERT$ denotes an assertion, and $SatMargin()$ is a predefined function that calculates the saturation margin of a transistor. The right hand side of the inequality sets the margin to be 5% of the supply voltage. When the transistor's Vds-Vdsat is less than 0.05*Vdd, STAR prints out a failure message.

As part of the netlisting process, the $SatMargin()$ function first finds all transistors in the sub-circuit that are labeled with the cs property. Then, in a pre-processing step, generates the appropriate HSpice probe commands into a file to find the transistor's instantaneous Vds and Vdsat. Figure 3.10 shows the generated commands. The function finds the two transistors in the sub-circuit with the cs property: Mp and Mn. The HSpice .probe commands call other functions provided by the simulator to find the Vds and Vdsat of the transistors and to calculate the saturation margin. The references to the probes, $vmprb\_0$ and $vmprb\_1$, are unique and need to be passed back to STAR to allow the post-processing routines to access them. For this example, the post-processor simply reads the values of $vmprb\_0$ and $vmprb\_1$ in the simulation results and compares them to the constraint in the Assertion Comments.

If a cell is used multiple times throughout the design hierarchy, STAR will traverse the hierarchy to find all the Assertion Comments in each instance and generate the appropriate probe statements into one file. The circuit designer then includes the generated file when running the simulation and use STAR to post-process and analyze the results.

The Assertion Comment in this simple form is suitable for checking those constraints

```
***** Include File for Assertions ****************
* add .include to the simulation deck
.probe vmprb_0=par('abs(vds(Mp))-abs(vdsat(Mp))')
.probe vmprb_1=par('abs(vds(Mn))-abs(vdsat(Mn))')
```

Figure 3.10: HSpice Probe Command Generated from Assertion Comment

that must always be satisfied. However, not all constraints are structured this way. The next section describes how we augment this simple form to support more complicated constraints.

3.4.2 Conditional Assertions

Sometimes what performance constraints to check are based on the circuit's operation. For example, we may want to delay the start of our saturation margin checks in the charge-pump until the simulation has passed the initial transient to avoid any false assertion violations before the circuit has settled. This creates a condition which must be met to activate the checks. We extend the simple $\# Assertion$ statement to include a validation condition as part of the comment:

# ASSERT SatMargin(cs) IF Time(bias) $>=$ 5ns

This comment contains two Assertions separated by a new keyword IF. The first is the same saturation margin check described in the previous section and the second checks the simulation has progressed longer than 5ns. The $Time()$ function returns the time value at each simulation step to give us a timing reference. The input to this function can be any node in the circuit since all nodes are evaluated at locked step. The IF keyword marks the beginning of the conditional clause. The Assertion following the keyword specifies the conditions under which the first $\# ASSERT$ statement is valid. STAR will first process all the Assertions after the IF keyword. If the results of these assertions make the IF clause true, then STAR will evaluate the first Assertion and report its result.

3.4.3 Assertion Comment Summary

Embedding assertions in every simulation can warn the user when margins are violated and prevent the circuit from operating outside the desired limits. These checks can spot errors that are missed when only functionality is tested. Often when the constraints are violated, the circuit will still function but with reduced robustness. These checks guarantee that all circuits are operating within the conditions for which they are valid. An added benefit of coupling the assertion with the design is that it enables seamless design reuse: when the block is integrated as part of a system by another designer, the same set of checks will be performed automatically.

3.5 Portable Comments

While Active Comments can be used to capture the design knowledge, simply embedding them into analog circuits do not automatically turn these circuits into reusable components. How well a circuit can be reused depends on how portable its design representation is. While Active Comments allow one to attach constraints to the designs, they do not a priori make the constraints portable. For example, constraints that use absolute values such as seconds, volts, and amps may no longer hold true as the target process technology changes, thus rendering these constraints non-portable. One way to improve the portability of the constraints is to transform these absolute values into relative values by using process independent metrics such as gate delays, supply voltages, and reference currents. Using process independent metrics reduces the dependencies on the technology, but does not convey much information about the true issues with the underlying circuit. The best way to constrain the circuit is to use some inherent properties of that circuit (i.e., we should use what the circuit is designed to do) to constrain the circuit. This constraint then describes the circuit at a higher level. Ideally, we want to archive the design information at this level.

We illustrate the concept of portable comments using two examples. In the first example, we transform a non-portable comment into a portable one. In the second example, we show how a constraint that is already scalable with process evolves to capture the designer's knowledge without becoming process dependent.

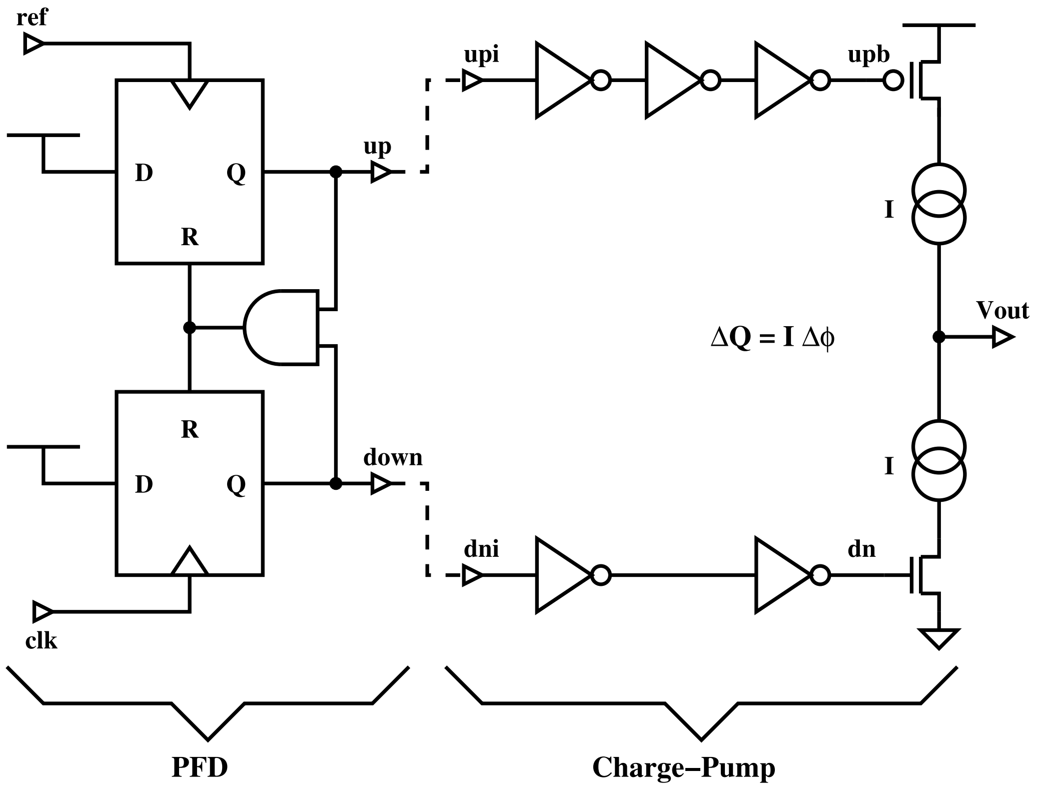


Figure 3.11: Connection of PFD and Charge-Pump Circuit Blocks

In the first example, we applied Active Comments to constrain the interface between the PFD and charge-pump of the PLL circuit and show how these comments evolved and become more robust and scalable. Together, the PFD and charge-pump (Figure 3.11) convert the phase information into voltage. The PFD outputs are pulses where the difference in the up and $down$ pulse widths equals the phase error, $\Delta\phi$. These pulses turn the current sources in the charge-pump on and off such that the net change, ΔQ, from the charge-pump is proportional to $\Delta\phi$. To reduce static phase offset, we want to overlap the up and down currents while minimizing the current pulse widths to reduce the effect of current mismatches. To achieve this, our charge-pump has tuned delay-matching inverter chains to shape the up and $down$ control signals. However, if the control pulses from the PFD are too narrow, they will disappear in the inverter chains because of the limited bandwidth of the inverters. Therefore, we need to set a limit on the minimum pulse width.

From simulation, we found that if the pulse width is less than 500ps, then the pulses will disappear in the inverter chain under some process corners. To ensure the pulses are wide enough, we constrain the pulse width to be at least 500ps as follows:

```
# ASSERT PulseWidth(up) <= 500ps
# ASSERT PulseWidth(down) <= 500ps
```

While the 500ps constraint correctly bounds the pulse width for the up and down control signals for the 0.35μm technology, that value is too wide for technologies with finer geometries. To improve the scalability of this assertion, we normalize the pulse width to the delay of a fanout-of-four (FO4) inverter, and transform the constraint to be 3*FO4:

```
# ASSERT PulseWidth(up) <= 3*FO4
# ASSERT PulseWidth(down) <= 3*FO4
```

Recoding this assertion using gate delays reduces the its dependency on the process technology and thus improves the scalability of the comment. We can update the value of these metrics as part of the technology calibration step. But this is still not quite right.

By examining what the comment is trying to constrain, we can further refine it. In this example, we are interested in making sure that if there is a pulse at the input of the inverter chain, a pulse appears at the output of the chain. Taking advantage of the fact that the PLL is periodic, we can turn the constraint into checking for a pulse at the output when there is a pulse at the input:

```
# ASSERT PulsePropagation(upi,upb)
# ASSERT PulsePropagation(dni,dn)
```

By looking for the existence of pulses, the assertion now checks exactly what the circuit is designed to do – propagate pulses through the inverter chain. Coding the comment this way not only offers more insight to the circuit's intended function, but also make the constraint completely independent of the process technology. Note, one now needs a separate check to check the minimum width of the pulse.

Often a constraint may already be in a portable form. But as the designer transforms the problem or specification into circuit implementation, the constraints evolve to provide more insight on how the circuit should behave. In the second example, we look at the problem of isolating the supply noise. Noise in the power supply network causes uncertainty in the output voltage, current or timing. So, rejecting the power supply noise is a requirement commonly found in analog circuits. Often, the supply rejection is specified as a ratio of the supply noise to output noise. This ratio indicates how well isolated the circuit is from noise. How the specification is written depends on the application. For oscillators, we are interested in the sensitivity of the delay and phase of the output clock to the supply noise. One common figure of merit is %-change-delay/%-change-supply [37][36][38]. We apply Measurement Comments to the VCO critical path schematic (Figure 3.4) to measure the dynamic supply sensitivity of the VCO. We code the comments by mimicking the specification:

```
# DEFINE periodHi=Max(Period(ck)), periodLo=Min(Period(ck) w/ \
    StepV(Vdd,vddval,0.9*vddval,1ns), SweepV(vctrl,vLo,vHi)
# CALCULATE psrr = ((periodHi-periodLo)/periodHi)/0.1
```

Using $\# DEFINE$, we measure the effect of a supply step on the VCO's output frequency across its operating range. We sweep the control voltage from vLo to vHi using the $SweepV()$ function, where vLo and vHi were found in the previous measurement. The supply is modulated by a step using the $StepV$ function. This function takes four inputs: the node to apply the step, the initial voltage, the final voltage, and the time at which the step is applied. The $Period()$ function reads the transient waveforms and continuously measures the clock period. The $Max()$ and $Min()$ functions find the maximum and minimum periods of the VCO at a given control voltage. This captures the delay deviation caused by the supply noise. The variables storing the measurement results, $periodHi$ and $periodLo$, are vectors where each entry in the vector corresponds to a control voltage level.

The value of PSRR is calculated using $\# CALCULATE$. The numerator calculates the percentage change of the VCO period, and the denominator is the 10% change in supply. This ratio is evaluated for each control voltage level. The variable $psrr$ stores the results as

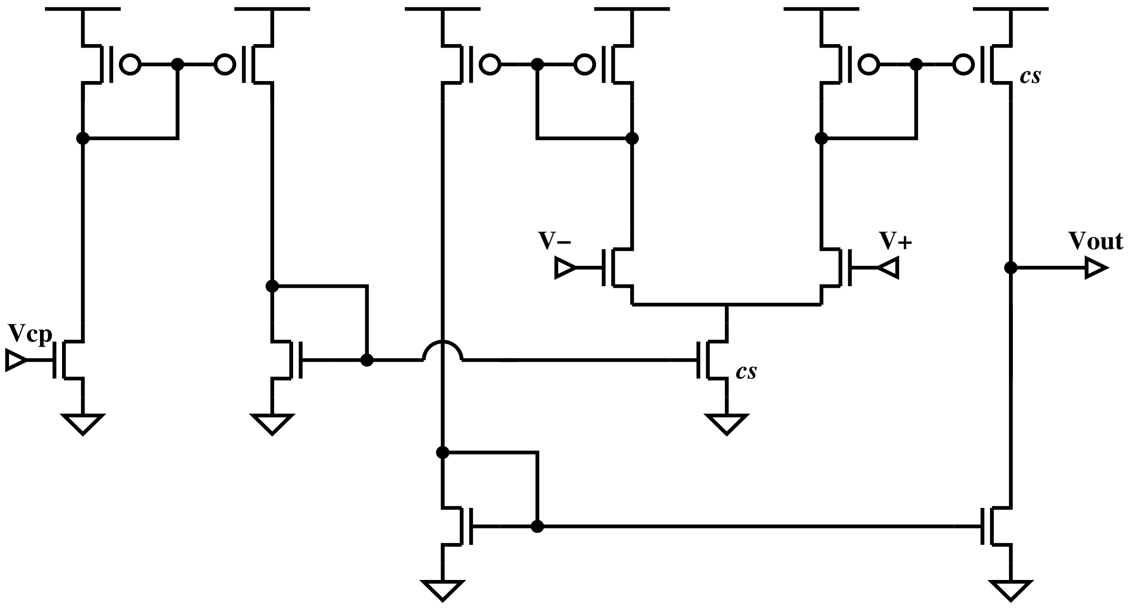


Figure 3.12: Op-Amp Schematic

a vector. The ratio of output distortion to the induced supply noise gives us a measurable quantity of the power supply rejection.

Since PSRR is a ratio of two quantities, this constraint does not depend on the process technology and is therefore portable. However, this specification is also independent of the circuit implementation. Measuring the constraint this way does not offer much insight to how the rejection is achieved. From the test bench schematic, we find that the ring oscillator is shielded from the supply by an op-amp. The supply sensitivity of this amplifier determines the overall PSRR of the VCO. Figure 3.12 shows the circuit of the amplifier. The supply rejection of this circuit is achieved by keeping the output devices and the tail current source in saturation. The saturated devices increase the impedance looking into the supply, thereby isolating the circuit from the supply noise. The circuit implementation transforms the problem of rejecting supply noise into checking the devices' saturation margin and their output impedance. Using Assertions, we can constantly monitor the devices' operating conditions to ensure sufficient saturation margin and output impedance:

```
# ASSERT SatMargin(cs) >= 0.05*Vdd
```

ASSERT OutputResistance(cs) $>=$ 1 Mega

The $OutputResistance()$ function finds the output resistance of the transistors that are labeled with the cs property. To isolate the circuit from supply noise, the current source transistors must have their output impedance in the mega ohms. Checking the devices operation this way captures how the designer intends to have the circuit achieve supply rejection.

The Assertion Comments are very good at monitoring circuit performance, but they can only check those constraints specified by the designer. The Assertions described in the previous paragraph monitor the circuit's operation to ensure the impedance between the supply rails and the switching transistors are sufficiently high, but they do not prevent other causes of failure such as noise feed through from capacitive coupling. Therefore, it is important to use these Assertions in conjunction with the PSRR Measurement described earlier. If the Measurement fails but the Assertions are not violated, then that indicates that mechanisms different from the ones we are checking for are causing the failure, and more Assertions need to be added.

Transforming the constraint from absolute value to relative value usually can be done by normalizing the absolute values to a process independent metric. However, recasting the constraint from a relative value to using circuit properties requires a good understanding of the underlying circuit. It may be unreasonable to expect the designer who designed the original circuit to write all the Active Comments in the scalable form. However with the capability of our tool, we expect other designers to continue to refine the comments as they reuse the circuits. With better understandings of the circuit, the designers can rewrite the comments to be more intrinsic to the underlying circuit, and therefore making the design representation more robust and scalable. To encourage the designers to move along this path, we can enlist the help of other experts to reformulate the constraints as part of the design review process.

3.6 Summary

The Active Comments capture the designers' knowledge as part of the design representation by enabling them to formally specify their intentions. The Measurements describes the critical circuit parameters that the design is being optimized for, and the Assertions ensure that the circuits perform within the specification everywhere they are instantiated. To ensure the portability of the Active Comments, we must avoid process dependent constraints. Instead, we should constrain the circuit using its intrinsic properties. This way, the comments not only offer insight, but also are guaranteed to scale with process.

In the next chapter, we discuss the prototype that we implemented to process the Active Comments and to support the notion of scalable comments.

Chapter 4

Prototype Implementation

Based the functional description of the Active Comments in Chapter 3, we constructed a prototype of STAR to help automate the generation and the verification processes for analog circuits. STAR translates the Active Comments in the schematics into simulation decks and analysis directives. After simulation, the results are combined with the analysis directives and verified automatically by the tool. We use this prototype to demonstrate the utility and feasibility of the proposed design capture system. The prototype is designed to be flexible to allow the user to extend the pre-defined functions used to construct the Active Comments, while at the same time shielding the user from complicated programming effort. Furthermore, the prototype handles the communications between the pre-defined functions used by the Active Comments, thus allowing the user to focus on the implementation of these pre-defined functions and not the interactions between the different functions. Finally, the design database of this prototype integrates the Active Comments into the circuit implementation to ensure that the annotation is coupled and up-to-date with the implementation.

The prototype is composed of four layers, as shown in Fig. 4.1. The top-most layer, the schematic layer, implements the schematic user interface and is written in Tcl/Tk [39] while the middle two layers form the library and the execution engine and are written in Perl [40]. The primitive layer contains C++ code [41] optimized for the calculation intensive data processing routines. We want these layers to be modular so that each is not tied to how the others are implemented.

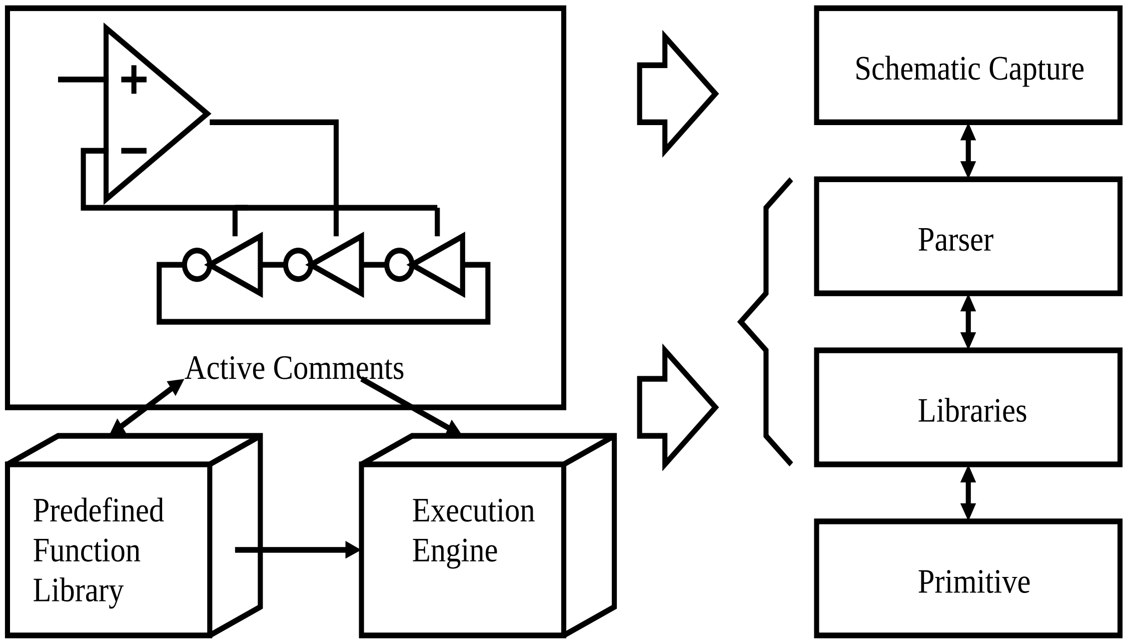
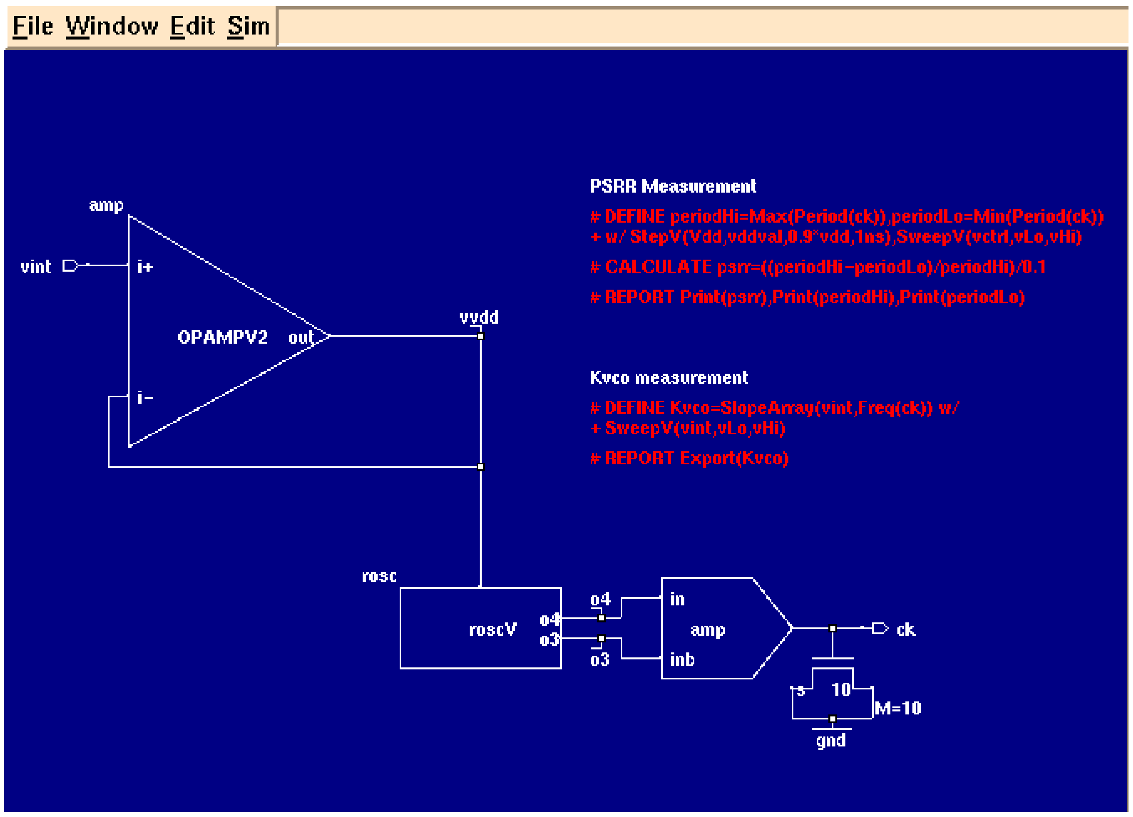


Figure 4.1: Layering of the Prototype Tool

This chapter is organized according to the implementation of the prototype. Each of the first four sections describes one of the layers, followed by a discussion of the implementation complexity in Section 4.5.

4.1 Schematic Layer

For analog circuits, the schematic diagrams are the most commonly used circuit representation. Having the circuit designers interface with the prototype framework through a schematic capture tool is a natural extension to current design practice. So we designed the main interface to our framework to be a schematic capture tool. In our prototype, we leveraged an existing schematic capture tool, SUE [42], as the starting point for the schematic layer. In addition to providing the conventional functions of a graphical design entry system such as drawing circuits and generating circuit netlists, the schematic layer must also provide a way for the circuit designers to enter Active Comments and pass these Comments to the parser layer. To ensure that the Active Comments are tied to the circuit implementation, they should be integrated into the same database as the schematic, such that there is



PSRR Measurement

```
# DEFINE periodHi=Max(Period(ck)),periodLo=Min(Period(ck))
+ w/ StepV(Vdd,vddval,0.9*vdd,1ns),SweepV(vctrl,vLo,vHi)
# CALCULATE psrr=((periodHi-periodLo)/periodHi)/0.1
# REPORT Print(psrr),Print(periodHi),Print(periodLo)
```

Kvco measurement

```
# DEFINE Kvco=SlopeArray(vint,Freq(ck)) w/
+ SweepV(vint,vLo,vHi)
# REPORT Export(Kvco)
```

Figure 4.2: Screen Capture of the VCO Test Bench Schematic in SUE

only one copy of the Active Comments.

In the following subsections, we start by giving a brief overview of SUE, followed by a description of the extensions that allow the circuit designer to add and modify the Active Comment, in Section 4.1.2 and Section 4.1.3.

4.1.1 Schematic Capture Tool

SUE supports hierarchical and parameterized designs, so the user can instantiate the symbol of a device or cell, enter the sizes, and draw the interconnections. The tool is written in Tcl/Tk and the schematic interface is shown in a Tk canvas. Figure 4.2 shows a screen capture of the tool displaying the test bench schematic of the VCO. Each cell in SUE has a schematic view that shows the topology and sizes and an icon view that shows the symbol of the cell. A netlister is included as part of the schematic tool to generate a hierarchical

HSpice netlist.

In order for the back-end engine to process the Active Comments, the schematic layer must pass the Comments into the parser layer. We extended the netlister to produce a comments file that contains the Active Comments in each sub-circuit. Figure 4.3 lists the HSpice netlist (.spi). For clarity, some of the details are omitted. The netlist file contains all the cells used in the design. The definition of each cell starts with $.SUBCKT$ and ends with $.ENDS$. Each cell is defined once and may be used multiple times. For example, the delay cell, $invV$, is defined once between lines 20 and 23 and used five times in the regulated supply oscillator, $roscV$, in lines 26 through 28. Within each sub-circuit block, transistor definitions begin with M and cell instantiations begin with X. In this example, there are five sub-circuits (inv, $lo2hi$, $opAmp$, $invV$, and $roscV$), and one top-level circuit ($vcoV3$).

The corresponding comments file (.cmt) of the VCO circuit is shown in Figure 4.4. The comments file follows the same structure as the netlist and contains the same set of cells as the netlist. In the definition section for each cell, the transistors and cell instantiations are replaced by the Active Comments shown on the schematic of that cell. The '+' sign at the beginning of lines 21, 22 and 23 means the line is a continuation of the preceding line. The connectivity information about where the cells are instantiated can be derived from the transistor netlist.

The Active Comments are integrated into the schematic, as shown in the red text in Figure 4.2. In the first implementation of the interface, we simply wrote the Active Comments directly into the schematic as text. During the netlisting process, the netlister scans the schematic and prints out all the text that begins with the keywords used in the Active Comments. However, we found that while these Active Comments may be easily parsed by a computer program, they require some annotation to clarify their intentions to other circuit designers. Adding detailed description to each Active Comment in the schematic is prohibitive as this approach soon fills the schematic with text. To address this issue, we created a new view, comments view, in addition to the typical schematic and iconic views. This view is edited using a Comments Editor. With it, the user not only codes the Active Comments for the underlying schematic but also enters detailed description about these comments.

```
.SUBCKT inv in out WP=16 LP=2 WN=8 LN=2
M_0 out in Gnd Gnd NMOS W=WN L=LN GEO=1
M_1 out in Vdd Vdd PMOS W=WP L=LP GEO=1
.ENDS

.SUBCKT lo2hi in inb out
Xinv29 net_1 out inv WP=28 LP=2 WN=14 LN=2
Xinv79 pb net_1 inv WP=16 LP=2 WN=8 LN=2
M_0 net_3 net_4 net_2 Vdd PMOS W=16 L=2 GEO=1
    ...
M_9 pb net_3 gnd Gnd NMOS W=8 L=3 GEO=1
.ENDS

.SUBCKT opAmp inM inP out
M_0 cs csM net_1 Vdd PMOS W=64 L=3 GEO=1
    ...
M_7 cs cs gnd Gnd NMOS W=32 L=3 GEO=1
.ENDS

.SUBCKT invV in out vtop WP=16 LP=2 WN=8 LN=2
M_0 out in gnd Gnd NMOS W=WN L=LN GEO=1
M_1 out in vtop vtop PMOS W=WP L=LP GEO=1
.ENDS

.SUBCKT roscV o3 o4 vvdd
XinvV0 o4 o0 net_1 invV WP=32 LP=2 WN=16 LN=2
    ...
XinvV4 o3 o4 net_1 invV WP=32 LP=2 WN=16 LN=2
Vmvvdd vvdd net_1 DC 0
.ENDS

* .SUBCKT vcoV3 ck vint $ start main CELL vcoV3 $
M_0 gnd vvdd gnd Gnd NMOS W=40 L=10 GEO=0 M=25
Xrosc o3 o4 net_1 roscV
Xamp vvdd vint vvdd opAmp
Xoamp60 o4 o3 ck lo2hi
* .ENDS
```

Figure 4.3: Hierarchical HSpice Netlist for VCO Test Bench Schematic

```
.SUBCKT inv
.ENDS

.SUBCKT lo2hi
.ENDS

.SUBCKT opAmp
 ASSERT SatMargin(cs) >= 0.05*vhigh
 ASSERT OutputResistance(cs) >= 1Mega
.ENDS

.SUBCKT invV
.ENDS

.SUBCKT roscV
 ASSERT swing(n4)
.ENDS

* .SUBCKT vcoV3
 DEFINE periodHi=Max(Period(ck)),
+periodLo=Min(Period(ck)) w/
+StepV(Vdd,vddval,0.9*vddval,1ns),SweepV(vctrl,vLo,vHi)
 CALCULATE psrr = ((periodHi-periodLo)/periodHi)/0.1
 REPORT Print(psrr),Print(periodHi),Print(periodLo)
 DEFINE Kvco=SlopeArray(vint,Freq(ck)) w/
+SweepV(vint,vLo_c,vHi_c)
 REPORT Export(Kvco)
* .ENDS
```

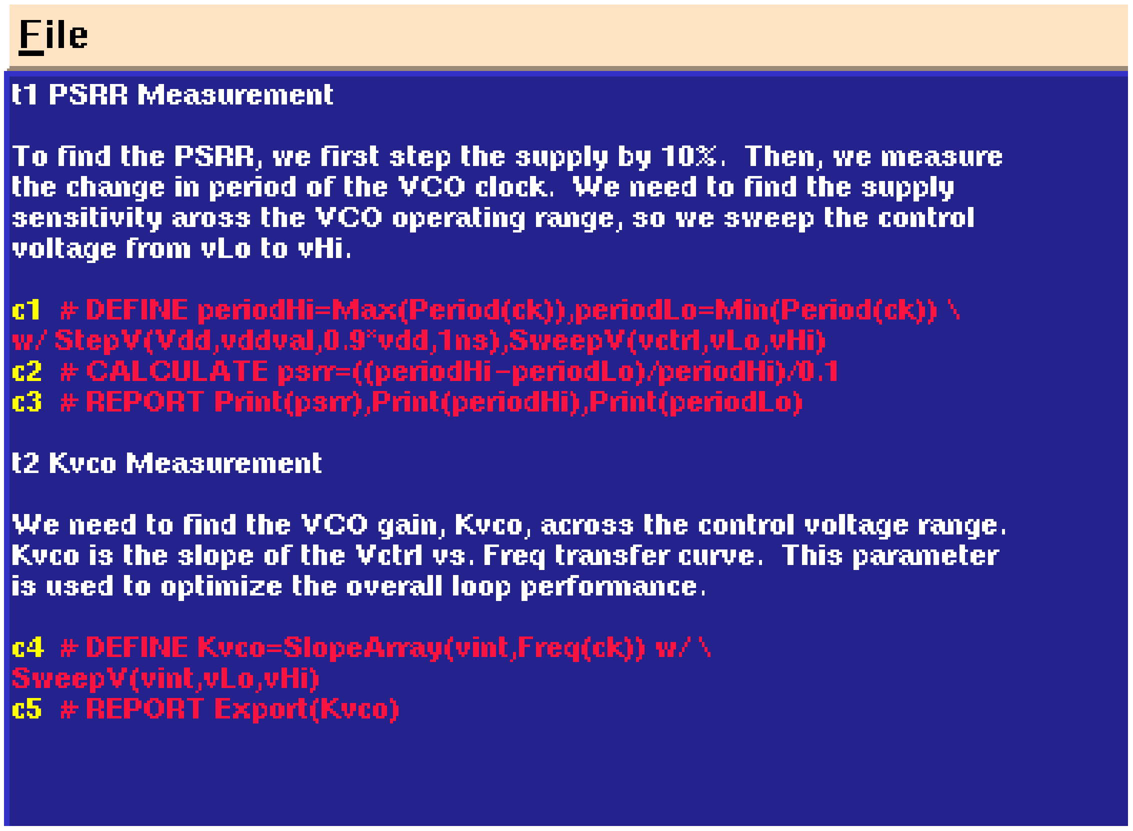
Figure 4.4: Hierarchical Comments File for VCO Test Bench Schematic

Having a separate view of the design enables the user to write detailed description of the circuits and the Active Comments used to capture the circuit. In addition, as the circuit evolves, the user can keep a record of all the checks that were used to constrain the circuit to let later designers know how the circuit has evolved and what checks worked and what did not. However, turning the comments view into a repository means there are many Active Comments in the view, and not all are applicable to the underlying circuit. So, simply writing an Active Comment in this view does not make it executable. We wrote a selector to allow the user to choose which Active Comments are enabled. In the next two subsections, we describe the editor and selector in more detail.

4.1.2 Comments Editor

Figure 4.5 shows the comments view of the VCO test bench. The contents in the editor consist of groups of Active Comments. Each group is divided into three parts: title, description, and Active Comments. The title marks the beginning of a group of related comments and gives an one-line summary of the purpose of that group. In this example, there are two groups: PSRR measurement and Kvco measurement. Each title begins with an identifier, t followed by an integer. The tool uses the identifier to keep track of the different groups. The description section is the notes that describes this group of comments in more detail. Any text that does not begin with an identifier is considered a description. And the last section within a group is the Active Comments. Similar to the title, each Active Comment is identified with a c followed by an integer.

The annotation in the comments view describes what the designer intended the Active Comments to do. However to someone not familiar with the set of pre-defined functions used in these Active Comments, it may still be difficult to decipher the Comments even with the annotation. Therefore, in addition to explaining the Active Comments into a more understandable and readable form, the editor also provides a mechanism to describe what each function was designed to do. This is analogous to using a dictionary to find the meaning of each word in a sentence. The user double clicks on the function in the comments view to bring up a window that displays a description of the function, as shown in Figure 4.6.



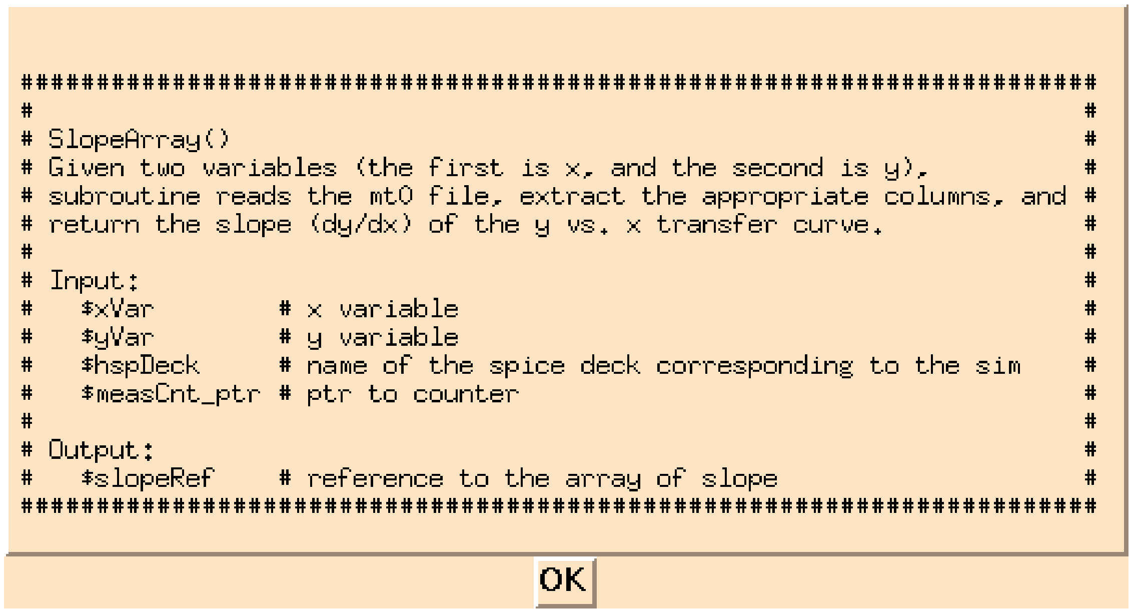
```
t1 PSRR Measurement

To find the PSRR, we first step the supply by 10%.  Then, we measure
the change in period of the VCO clock.  We need to find the supply
sensitivity aross the VCO operating range, so we sweep the control
voltage from vLo to vHi.

c1  # DEFINE periodHi=Max(Period(ck)),periodLo=Min(Period(ck)) \
w/ StepV(Vdd,vddval,0.9*vdd,1ns),SweepV(vctrl,vLo,vHi)
c2  # CALCULATE psrr=((periodHi-periodLo)/periodHi)/0.1
c3  # REPORT Print(psrr),Print(periodHi),Print(periodLo)

t2 Kvco Measurement

We need to find the VCO gain, Kvco, across the control voltage range.
Kvco is the slope of the Vctrl vs. Freq transfer curve.  This parameter
is used to optimize the overall loop performance.

c4  # DEFINE Kvco=SlopeArray(vint,Freq(ck)) w/ \
SweepV(vint,vLo,vHi)
c5  # REPORT Export(Kvco)
```

Figure 4.5: Comments View for the VCO Test Bench Circuit

This description is written by the programmer as part of the pre-defined function. The double clicking action activates a program in the editor to read the library files to dynamically generate a database of the function and its description. In addition to the description, the program also finds the function's input parameter list to inform the user on how each function is called. Extracting these information about the function relies on the programmer's willingness to write the description and to follow a specific coding style. This coding style is described in Section 4.3.2.

4.1.3 Comments Selector

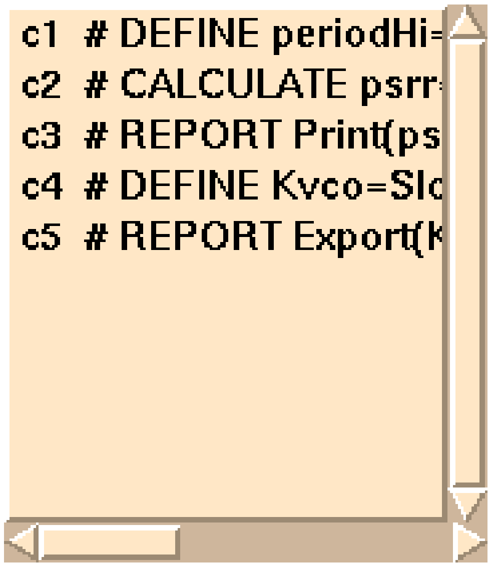
With the comments view being a live document of all the Active Comments that have been applied to the circuit, we need a mechanism to choose which Comments to execute. An



```
########################################################################
#                                                                      #
# SlopeArray()                                                         #
# Given two variables (the first is x, and the second is y),           #
# subroutine reads the mt0 file, extract the appropriate columns, and #
# return the slope (dy/dx) of the y vs. x transfer curve.              #
#                                                                      #
# Input:                                                               #
#    $xVar         # x variable                                        #
#    $yVar         # y variable                                        #
#    $hspDeck      # name of the spice deck corresponding to the sim   #
#    $measCnt_ptr  # ptr to counter                                    #
#                                                                      #
# Output:                                                              #
#    $slopeRef     # reference to the array of slope                   #
########################################################################
```

Figure 4.6: Pop-Up Window Displaying a Description of the Selected Function

Active Comment is activated when the user instantiates it into the schematic using the graphical selector shown in Figure 4.7. The selector shows all the Active Comments in the comments view, and the one chosen by the user is added into the schematic. Along with adding Active Comments to the schematic, we also want to include an one-line description about them to give other designers some idea of these Comments when they review the schematics. So, when instantiating one of the Active Comments from a group for the first time, the group title will be automatically added to the schematic, thus ensuring that there is always a short description associated with the Comments. We guarantee the synchronicity between the design information and the schematic by storing the only copy of the Active Comments in the comments view. Any place that needs to display a Comment must generate it from the source. For example, the Comments in the schematic window in Figure 4.2 are generated from the contents in the comments view and can only be modified using the comments editor.



```
c1  # DEFINE periodHi=
c2  # CALCULATE psrr
c3  # REPORT Print(ps
c4  # DEFINE Kvco=Slc
c5  # REPORT Export(K
```

Figure 4.7: GUI for Comments Selection

4.2 Parser Layer

The parser layer implements the engine of the STAR system to execute the Active Comments. The parser reads the Active Comments, which are declarative statements with variables and nested functions, and translates them into another set of programs to control circuit simulation. After the simulation completes, its results are processed by the analysis functions called by the parser. Many tasks performed by the parser are similar to those done by a compiler, so we can leverage some of these existing techniques when implementing this layer.

Figure 4.8 shows the execution flow of our implementation. The parser takes three inputs – the hierarchical spice netlist of the design (.spi), the comments file (.cmt), and a global parameter file (GlbParam). The global parameter file is part of the tool distribution and contains all the technology specific parameters such as the supply level, FO4 delay, and the location of the device models. This file enables the Active Comments to be written in a process independent manner by coding the comments using these parameters instead of the actual value. For users who work on projects that target different process technologies, the user can set an environment variable to point to the appropriate parameter file.

The execution flow is divided into two phases: generation and verification. In between

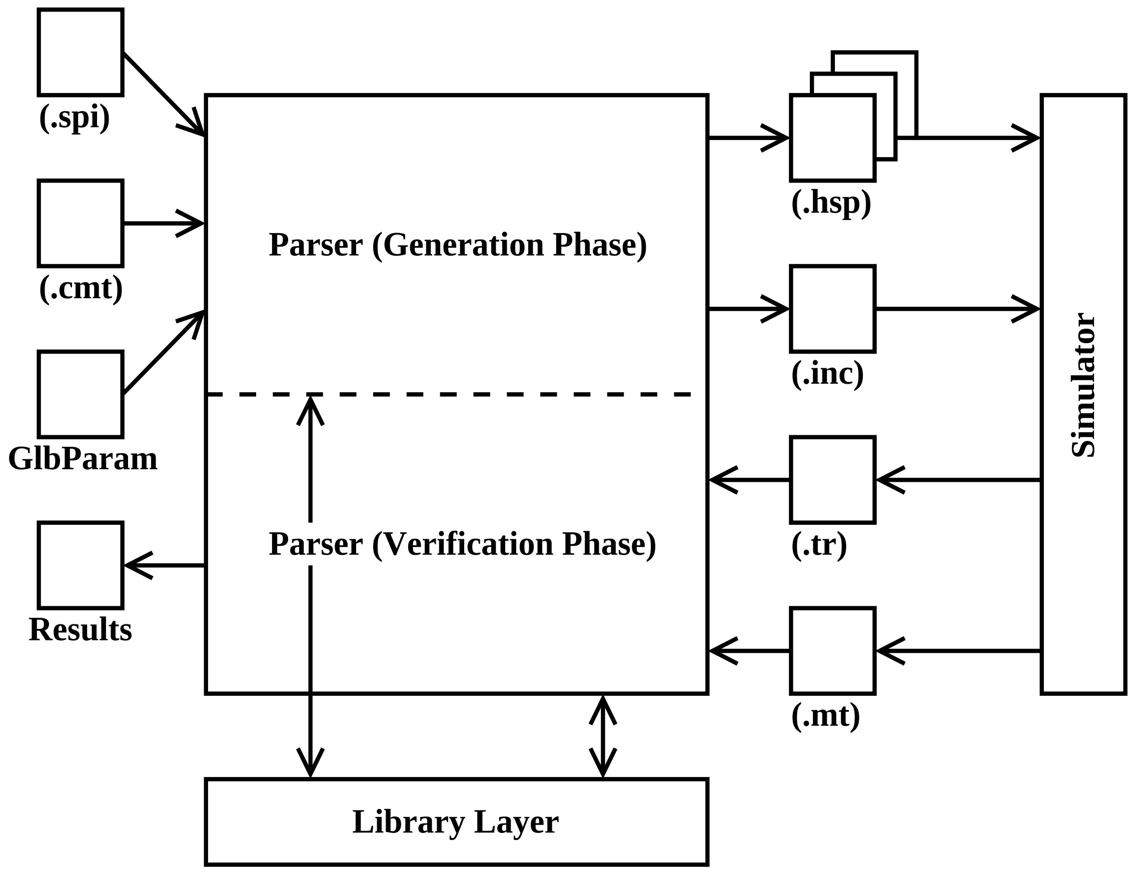


Figure 4.8: Flow of Execution

these two phases is circuit simulation. The generation phase executes all the pre-processing routines used in the Active Comments to generate the simulator stimuli files. After the simulations are completed, the parser enters the verification phase. All the simulation results are read into the system and processed with the appropriate post-processing functions in the library layer. All performance violations are reported in the Results file. While the two types of Active Comments share most of the execution flow, the Assertions have a few more requirements than the Measurements. The next subsection describes how the parser processes the Measurements, followed by a description of the extensions made to process the Assertions.

4.2.1 Measurement

During the generation phase, the parser reads the device netlist (.spi) and comments file (.cmt) produced by the schematic interface. The process ordering of the Measurement Comments follows the sequence outlined in Chapter 3. When implementing the parser to process the Measurements, we had to address two practical issues. One is the handling of variables used in the Active Comments. Since the variable used in both Measurements and Assertions can come from other Measurements, how the parser supports these variables can affect how they are used. The other issue is the handling of nested functions in the Active Comments. Nested functions offer a compact notation but requires the parser to provide a mechanism to enable different functions to pass data between each other.

When the parser encounters a variable in the Active Comment, it substitutes the appropriate value for that variable. This variable can be from one of three sources. The first is from the internal data structure of the current session such as a $\#\, CALCULATE$ statement using a variable defined by a $\#\, DEFINE$ statement on the same schematic. The value of this variable is still in the memory system of the computer, so the parser just need to reference that variable to make the substitution. The second type of variable is a process independent parameter such as Vdd or FO4. The values of these parameters are technology dependent and are stored in the GlbParam file. Finally, the third type of variable is one that is defined by a Measurement in a different schematic. In this case, the parser needs to locate the file that contains the evaluated result of that variable or to inform the user which schematic to run to define that variable. Figure 4.9 shows the flowchart of the search algorithm.

The parser underwent a few rounds of redesign to support this last type of variable. In the first implementation of the parser, there was a flat name space for all the variables. The exported variables from the Measurements are written into the GlbParam file, just like the technology independent variables. Consequently, the variable written by one Measurement may be corrupted by other Measurements. We modified the prototype so the results of the Measurements are exported to an unique file, one file per schematic. In the Comments, the user can specify the file that stores the variable by specifying the variable as <filename>.<variable>. If the filename is omitted, then the parser falls back to the search

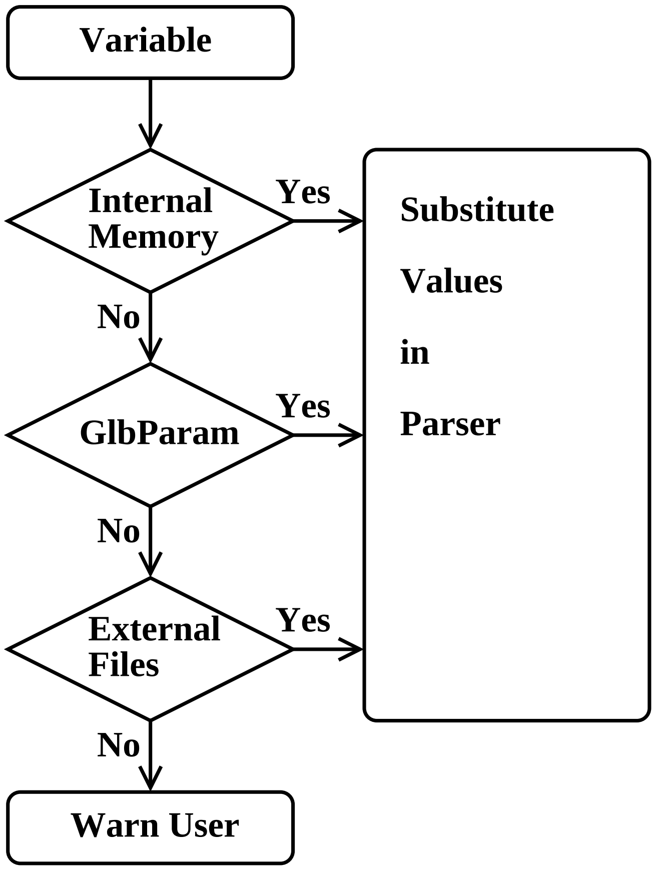


Figure 4.9: Parameter Search Flowchart

scheme described in the previous paragraph. We keep the three tier search algorithm for convenience and backward compatibility. By prepending the filename to the variables, we get around the flat name space problem.

The other practical issue with writing the parser is the transfer of data from one function to another. For example, many post-processing analyzer need multiple processing steps. These complicated analysis are often specified as a set of nested functions and require the results of one function to be passed to another. Returning to the PSRR measurement example in Chapter 3, finding the maximum and minimum period is a complex routine with nested functions:

```
# DEFINE periodHi=Max(Period(ck)), periodLo=Min(Period(ck)) w/ \
```

StepV(Vdd,vddval,0.9*vddval,1ns), SweepV(vctrl,vLo,vHi)

Implicit in the nested function notation is passing the results of the $Period()$ function to the $Max()$ and $Min()$ functions. We looked at how compilers support passing data between nested functions since most computer languages use this notation. One common approach to do this is to use temporary variables to hold the results of the inner functions and then pass the temporary variable into the outer function [43]. We use a similar approach to pass data between the analysis functions. First, we created a global hash table to store and keep track of these temporary results. Then, each analysis function allocates a new array to store its results. At the completion of the function, it generates an unique reference name and add the results array into the hash table using the reference as the key. This reference is the return value of the function so that other functions can access the data by using the reference as the index into the global hash table. Unlike most compilers, the parser in our system does not allocate the temporary variable because it does not know the data type of the analysis result. Instead, each function generates its own temporary variable, and the parser just passes the references between the nested functions.

4.2.2 Assertion

The Assertions differ from the Measurements in two ways. First, while only the Measurement Comments in the top level schematic are executed, all the Assertions across the hierarchy must be processed. This requires the parser to find all the instances of the Assertions. The second difference is that unlike the Measurements, the Assertions are not associated with any circuit stimulus. They ensure the circuit performs within specification regardless of it is excited.

In order to propagate the Assertion Comments across the design hierarchy, the parser needs to find all the instances of the cells that contain Assertions. The transistor-level spice netlist, like most VLSI connectivity files, is a tree with the root being the top-most circuit and the edges pointing from the parents to their children. Figure 4.10 shows the instantiation tree of the VCO test bench where each arc from the parent to the child represent one

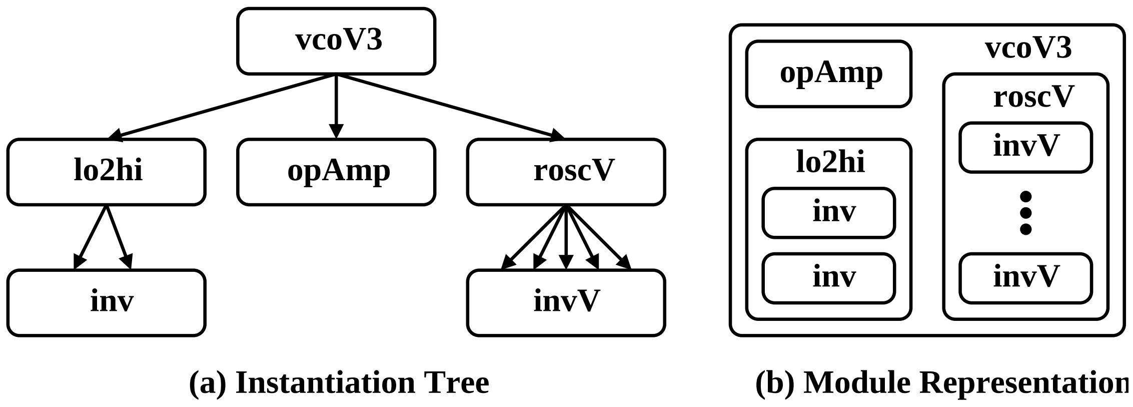


Figure 4.10: Different Representation of the VCO Test Bench Schematic (a) Instantiation Tree (b) Module Representation

instance of that child. Making each instance of a cell unique requires flattening of the hierarchy. This can be accomplished in a pre-processing step by searching breadth first or depth first from the root. Alternatively, each instance can be found by performinng a depth first search from the cell towards the root. For historical reasons, the prototype implemented the latter approach. While the search from the leaf node to the root can be executed by recursively traversing the netlist, it is not very efficient since the connectivity information is only from the parents to the cells. Instead, the parser creates a tree with double edges pointing in both direction to enable efficient traversal of the graph.

The Measurement Comments completely specify the pre- and post-processing procedures of a simulation, so the generation, simulation, and verification steps are all encapsulated in one session. All the data can be passed internally between the two generation and verification phases. However, the Assertion Comments are intended to be used even when the stimuli is not generated by STAR. As the analog block is integrated into the complete system, the simulation routines may be generated from other sources such as verilog or manual coding by other designers. To support this model, the parser produces an include file (.inc) which contains the simulation monitors from all the Assertion Comments, such as the .probe command from the $SatMargin()$ function. The designers and verification engineers include this file as part of their simulation runs and use STAR to ensure all levels of the circuit performed within the specification.

Each simulation monitor in the include file has an unique reference name that post-processing functions use to identify the monitor in the simulation results. There needs to be a mechanism to associate the reference to the original Assertion Comment so that the Comment can be properly executed when the simulation is completed. While linking the reference and the Assertion can be done easily when the generation, simulation, and verification loop is encapsulated one STAR session, since Assertions supports the notion of having the simulation directives be generated by other sources, the loop is broken, and it is possible that the schematics, Active Comments, and/or the pre-defined functions used during the generation phase have changed while the design is in simulation. We cannot simply parse the database again to reproduce the simulation monitors and references.

There are a number of ways to solve this problem. One approach is to write out the analysis procedures as scripts during the generation phase. These scripts would then contain all the reference names, and the user can just execute these scripts at the end of the simulation run. The problem with this approach is that it will be difficult to modify these generated script to perform a different analysis on the simulation results. Another approach is to save the netlist and comments file and take a snapshot of the pre-defined functions in the library. This way, we are guaranteed to be able to recreate the references to the probes and measurements. With the references available, the user can modify the post-processing routines in the library to perform different analysis on the simulation results, if necessary.

Since we are not going to change the pre-processing routines after the simulation, it is not necessary to save a snapshot of the library. Actually, we do not need to recreate the references. Instead, we only need to preserve the relationship between the reference name with its Assertion Comment. Our approach is to write the parse tree into a file. This file represents the state of the parser at the end of the generation phase. At the beginning of the verification phase, the designer loads the parse tree file to restore the state and then proceed with the verification of the Assertions. This way, we are still able to modify the analysis routines, if necessary. To ensure that we load in the correct state, a time stamp is added to the parse tree file and the include file. This approach preserves the needed information in the prototype engine to analyze the simulation results. However if there are assertion failures, then we would need the original schematics to debug the design. While not implemented in the current prototype, the schematics should be placed under revision control and be tagged with the same time stamp.

After loading the parse tree and checking its time stamp against the one in the include file, the parser reads the simulation results. The Assertion Comments monitor the design at each time step of the simulation. In general, there are two ways to process the Assertions. One way is to traverse the simulation result once and evaluate all the Assertions at each time step. The other way is to evaluate one Assertion after another and traverse through the entire simulation results during each processing of the comment. While the second approach is less efficient given the overhead of opening and reading the simulation files, we choose this approach because some analysis routines require multiple passes at the data set.

While the parser layer implements the execution platform, the actual processing functions are stored the library layer. In the next section, we describe the organization and implementation of the library.

4.3 Library Layer

In addition to the schematic layer, the designers also interface with the library layer. They use the pre-defined functions stored in this layer to compose the Active Comments. This layer is organized into pre-simulation and post-simulation libraries. Each library is a file containing the definition of the pre-define functions. When STAR is executed, the contents of these files are linked into the tool to form one executable. Dividing the libraries this way allows the same function be split across circuit simulation, so that the stimuli generation portion can be placed in the pre-simulation library and the verification portion can be placed in the post-simulation library. There are four libraries, two for the pre-processing functions and two for the post processing functions. The pre-processing functions for the Measurement and Assertions are stored in MeasGenerate.pm and AsrtGenerate.pm, respectively. Similarly, MeasAnalysis.pm and AsrtAnalysis.pm hold the post-processing function. Table 4.1 summarizes the libraries.

Having an extensible library means that some of the functions in the library will be written by circuit designers instead of CAD developers. The implementation of library layer should be geared towards reducing the programming effort for the designers when they create their own functions. Judging from the HSpice simulation decks and the Active Comments, we anticipate the functions to perform text processing and regular expression matching. The datatype of the parameters in the Active Comments can be either scalar or vector, so the underlying language must support at least these two datatypes. To reduce the effort of using the variables when writing the pre-defined functions, the library implementation should shield the user from managing memory usage such as allocating and freeing memory.

Perl seems like a good fit for this application. The language is designed to support text processing with regular expression. Perl also provides three basic datatypes – scalar, array, and hash – and pointers to variables. By using a combination of arrays and hashes, the user can create any data structure [44]. The runtime system includes a memory manager which supports dynamic memory allocation to allow user to freely increase the size of the data structure and automatic garbage collection to free unused memory. An added benefit of using an interpreted language is that the code is easily accessible to the circuit designers.

The functions used to capture a design depend on the class of circuits. In Section 4.3.1, we describe the set of default functions that we include as part of the tool distribution. In Section 4.3.2, we describe how to extend the library and the issues associated with having an extensible library.

4.3.1 Default Functions

For the type of circuits used to build a phase-locked loop, we were able to construct all the functions needed for our Active Comments based on the functions listed in Table 4.2. To manipulate the inputs, we created some functions to set the voltage to a particular value, ramp the voltage for some pseudo dc simulations, and sweep the voltage. A similar set of functions are created to manipulate the current. We found that the simulator can give us better access to the inner operations of the transistors. So, we leverage the simulator

Table 4.1: Matrix of Library Files

	Measurement	Assertion
Generation Functions	MeasGenerate.pm	AsrtGenerate.pm
Analysis Functions	MeasAnalysis.pm	AsrtAnalysis.pm

Table 4.2: Summary of Functions

Name	Description
SetV, SetI, SweepV, SweepI, RampV, RampI	Input stimuli to set, sweep, or ramp (for pseudo DC simulations) a voltage
MeasRout, MeasRch, SatMargin	Use simulator to measure transistor's operating condition
V, I	Access raw simulation output
FindDelay, FindSlope, RunningAvg	Process output waveforms to find the delay, slope, or running average

to find the transistor's channel resistance, gate capacitance, and saturation margin. To analyze the simulation results, we need to access the raw voltage and current waveforms. We also need to process the output before doing the analysis, and we found three functions to be very useful. With the $FindDelay()$ function, we can find the time when the signal transitions or crosses a preset threshold. The $FindSlope()$ function is useful in finding the first derivative or slope of the curve. Finally, the $RunningAvg()$ function help smoothes the signal waveform. Below we briefly describe each of the functions. Appendix A contains a more detailed description of the implementation of the functions.

SetV, SetI

These two functions set the voltage at a node or current through a node to a DC value by adding a voltage or current source to the specified node. The functions are in the following form:

```
SetV(node,value)
SetI(node,value)
```

where $node$ is the name of the node in the circuit and $value$ is the DC voltage or current. These functions return the name of the voltage or current source added to the circuit.

SweepV, SweepI

These two functions sweep the voltage or current across a range of values. Using these two functions result in multiple simulation runs where each run corresponds to one value of the voltage or current. The functions are in the following form:

SweepV(node,value1,value2)
SweepI(node,value1,value2)

where $node$ is the name of the node in the circuit and $value1$ and $value2$ are the limits of the sweep. These functions first create a parameter in the stimulus deck, then sweep that parameter across from $value1$ to $value2$. The number of points in the sweep is a parameter in the GlbParam file. The instantiated voltage and current sources are grounded such that the voltage values are relative to ground and the injected current is sourced from ground. These functions return the name of the parameter they create.

RampV, RampI

These two functions ramp the voltage or current across a range of values. These functions are used for pseudo DC simulations. Unlike $SweepV()$ and $SweepI()$, there is only one simulation run when using these two functions. The functions are in the following form:

RampV(node,value1,value2)
RampI(node,value1,value2)

where $node$ is the name of the node in the circuit and $value1$ and $value2$ are the limits of the ramp. These two functions ramp the voltage or current by adding a piece-wise-linear voltage or current source to the specified node. Similar to $SweepV()$ and $SweepI()$, the voltage or current source are also grounded sources. These functions return the name of the voltage or current source added to the circuit.

MeasRout, MeasRch, SatMargin

These three functions uses the simulator's built-in functions to find the transistor's properties or operating conditions. These include the device's output impedance, channel resistance, and saturation margin. The functions are in the following form:

MeasRout(tr_prop)
MeasRch(tr_prop)
SatMargin(tr_prop)

where tr_propt is the name of the transistor property assigned by the designer. These functions add the appropriate simulation directives and probes to extract the information from the simulator. The return values of these functions are the names of the probes added to the simulation deck. During the verification phase, the parser uses these names to access the probe waveforms in the simulation results. The datatype of these waveforms are arrays of ($time$,$value$) where $time$ is the time at each simulation step and $value$ is the value of the probe at that time instant.

All the functions described up to this point are used during the generation phase of execution. They create simulation directives or add measurements and probes to monitor the circuit performance. The following functions are called during the verification phase. They are used to read and manipulate the simulation data.

V, I

These two functions read simulation transient results and return the raw values of the waveforms. The functions are in the following form:

V(node)
I(volt_src)

where $node$ is the name of the node in the circuit and $volt\_src$ is the name of the voltage source (or current meter). The return value of the function is the reference to an array of

($time$,$value$) ordered pair, where $value$ is the node voltage or current through the $volt\_src$.

FindDelay

This function finds the timing relationship between two signals. The function is written to be as general as possible which results in a long argument list. The function is in the following form:

FindDelay(signal1,threshold1,direction1,signal2,threshold2,direction2,numOfTran)

where $signals1/2$ are the name of two signals, $threshold1/2$ are the voltage levels to initiate the measurements, $direction1/2$ are the direction of transition (rise or fall), and $numOfTran$ refers to the number of transitions the second signal waits after the first signal reaches its threshold. The first signal/threshold is sometimes called the triggering event and the second signal/threshold the target event. If the same signal name is used for both $signal1$ and $signal2$, then the function finds the timing properties of that signal. Because of the long argument list, the function is not used in this form. Instead, wrappers were written to handle the different timing measurements we made in the PLL design. Chapter 5 describes what wrappers are used, and Appendix A describes these wrapper in detail. The return value of the function is the reference to an array of ($time$,$value$) ordered pair, where $time$ is the time instant when the triggering event happened and $value$ is the delay between the trigger and target events.

FindSlope

This function finds the instantaneous slope (or first derivative) of a waveform by calculating the change in the horizontal and vertical directions between each time step of the waveform. The function is in the following form:

FindSlope(signal)

where $signal$ is the name of the signal waveform. The return value of the function is the reference to an array that contains the slope of the input waveform at every time step.

RunningAvg

This function finds the running average of a waveform by taking the average within a timing window and sliding that window across the waveform. This effectively smoothes out the any noise on the signal. The function is in the following form:

RunningAvg(signal, size)

where $signal$ is the name of the signal waveform and $size$ is the width of the window over which the function takes the average. The return value of the function is the reference to an array that contains the running average of the input waveform.

By writing wrappers to nest the functions or combine them with simple arithmetic operators, we constructed the functions we used in our Active Comments. For example, the noise on the bias current can be found by taking the ratio of the instantaneous current through a current meter and the running average at that instant as follows:

NoiseI(t) = (I(t)-RunningAvg(I(t),30*FO4) / RunningAvg(I(t),30*FO4)

The $RunningAvg()$ function smoothes the signal using a window that is 30 FO4 delays wide to remove the noise, then the difference is the actual noise.

4.3.2 Extending Library

The library layer is designed to be flexible and extensible. Consequently, the layer is designed to enable the users to add their own functions to their private library in the local work area to extend or override the default functions provided by the tool distribution. During execution, the parser first uses the libraries in the directory pointed by the environment variable, $\$STAR\_USER\_LIB$ before using the ones in the distribution directory.

There are a number of practical issues with having each designer creating his/her own function library. One problem is that different designers may name some of their functions the same, causing name collisions. In order to avoid colliding function names, we need to manage the name space of the user provided functions similar to how the schematic names are managed (e.g., prepending unique project and module names to the function name). Another problem with the user-provided functions is that the design knowledge is split between the schematic and the local function library. Therefore, the designers must check-in both the circuit schematics and the supporting function library as part of the archival process. The third problem is when the design database is passed along to other users, we want to use the user interface in Figure 4.6 to provide some information about these user defined functions. Ideally, we would like to extract the functionality from the code. However, this is very difficult. So instead, we would like to enforce some coding style and annotation guidelines to help the tool to extract some information from the function.

We use $SatMargin()$ as an example on how to write a pre-defined function with the coding style that would enable the tool to pass back information about the function. The $SatMargin()$ function is used in Assertion Comments to find the saturation margin of transistors. This function takes the name of the transistor property as its argument and generates the appropriate HSpice .probe command to calculate the saturation margin of the transistors that has the specified property.

Figure 4.11 lists the Perl code implementing the function. The definition of the function begins with a block of comments describing the what the code does. Lines 8 through 11 are the inputs to the function. These inputs are passed into the function by position, and the $shift$ function (built-in in Perl) assigns each argument to the internal variables. While Perl provides a default array variable to hold the argument list passed into a function, STAR imposes a coding style that requires these inputs be assigned to scalars or arrays and be commented so that the utility program in the comments editor can identify them.

Recall that in the Assertion Comment $SatMargin()$ only takes one argument – the name of the transistor property. However, there are four input variables in the function implementation because the parser automatically adds three additional arguments during the function call. These three arguments are provided by the system to simplify the programming tasks and are always appended to the end of the argument list for any function called in the generation phase. Of these three additional variables, the first variable is the name of the sub-circuit where the function is used, the second is the name of the top-most circuit,

```perl
####################################################
#                                                  #
# Adds HSpice statements to check the saturation   #
# margin of a current source transistor            #
#                                                  #
####################################################
sub SatMargin{
  my $pattern      = shift;    # transistor name
  my $cktName      = shift;    # subckt name
  my $topCktName   = shift;    # top level ckt
  my $$measCnt_ptr = shift;    # ptr to counter

  my $CK_HSP_DIR;    # checker hspice directory
  my $incFile;       # include file for ASSERTION
  my $txPath_ptr;    # ptr to the transistor path
  my $tx;            # temp variable
  my $tmpStr;        # temp string
  my @varName;       # name of the measure variable

  $txPath_ptr = &FindTx($pattern,
                        $topCktName,
                        $cktName);

  $incFile  = &GetIncFile();
  open (INC, ">>$incFile");
  foreach $tx (@$txPath_ptr) {
    $tmpStr = "vmprb_".$measCnt++;
    push(@varname,$tmpStr);
    printf INC (".probe $tmpStr=par(";
    printf INC ("’abs(vds($tx))-abs(vdsat($tx))’)\n");
  }
  close(INC);
  return (\@varName);
}

```

Figure 4.11: Perl Code of $SatMargin()$

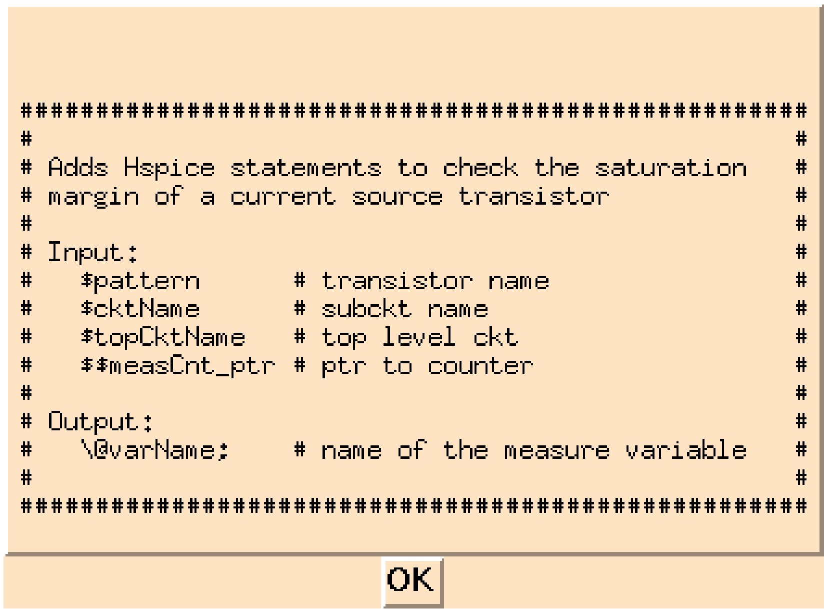
and the last is a pointer to a counter. The first two variables are necessary for searching and identifying all the instances of the sub-circuit, while the counter enables the function to generate an unique name for a spice measurement or probe. The user creates an unique name by incrementing the count by one and appending the new value to the name. Since this count is passed into every function as a pointer, we can guarantee that each function receives a different count value.

In lines 20 through 22, the function calls $FindTransistorProperty()$ to find all the transistors whose property contains the input pattern. The search function is very general and can search from any part of the graph to any other part of the graph. So we pass in the name of the top most circuit and the name of the sub-circuit that contains the Active Comment as the limits of the search space. The search function returns an array containing the full path of those transistors that match the search pattern.

For the rest of the function, we process this array to create a simulation probe for each transistor. In line 24, we locate the file name of the include file that our HSpice commands are to be written in. This file is opened in append mode because there may be other Assertion functions that will write to it. We write a probe statement for each transistor in the array using the for loop in lines 26 through 31. Each probe statement has a unique reference name which is stored in another array. The memory address of this array is passed back to the parser as the return value of this function.

Previously in Section 4.1.2 we described an utility in the comments editor that produces a dialog window to display information about the function. The auto-generated help window for this example function is shownd in Figure 4.12. The content of the window includes the comment in the “#” text box in lines 1-6, the input variables, which are those defined with the Perl $shift$ function call in lines 8-11, and the return variable, which is marked by the Perl $return()$ funcion line line 33.

By restricting the coding style and having a guideline for adding comments, a tool can easily recognize the inputs and outputs of a function without carefully analyzing the code, extract the comments from the library, check for the existence of comments as part of the archiving process, and auto-generate a template to help get the user started on a new function. While it still rely on the programmer to enter the correct comments, this approach provides a mechanism to relay some information about the functions to the user.



```
##################################################
#                                                #
# Adds Hspice statements to check the saturation #
# margin of a current source transistor          #
#                                                #
# Input:                                         #
#   $pattern      # transistor name              #
#   $cktName      # subckt name                  #
#   $topCktName   # top level ckt                #
#   $$measCnt_ptr # ptr to counter               #
#                                                #
# Output:                                        #
#   \@varName;    # name of the measure variable #
#                                                #
##################################################
```

Figure 4.12: Pop-Up Window Displaying a Description of the SatMargin Function

4.4 Primitive Layer

Having the analysis function written in an interpreted language can lead to long runtime. To improve the performance, we re-implemented some of the common tasks in C++. These tasks are associated with reading and processing the transient waveforms when verifying the simulation results, and they are very IO and computation intensive. The two C++ functions are $FindWave()$ and $FindTime()$.

To process the results, we first need to load them into the system. The $FindWave()$ takes the signal name as its argument, reads the transient results file, and returns the voltage and current waveforms. For example:

FindWave(clk)

returns an array of $(time, value)$ ordered pair for the clk signal. This primitive is called by the pre-defined functions $V()$ and $I()$ in the library layer to find the voltage and current waveforms. It is important to note that since the the primitive layer interfaces with the library not the parser, the primitives do not have access to the global hashtable used by

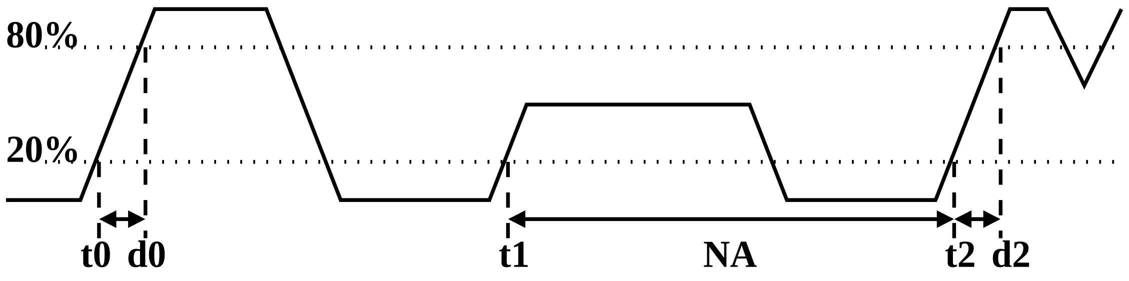


Figure 4.13: Using $FindTime()$ to Find Rise Time of a Signal: The function returns [(t0,d0),(t1,NA),(t2,d2),...]

the parser. Therefore, unlike the analysis functions in the library, the return value of the primitives are the actual arrays of ordered pair not the $reference$ to an array stored in the global hashtable.

For mixed-signal circuits, designers often need to extract timing information from the waveforms such as rise and fall times, delay between two signals, and period of a signal. We wrote the $FindTime()$ primitive to process these waveform. The primitive has the same argument list as the $FindDelay()$ function in the library layer:

```
FindTime(signal1,threshold1,direction1,signal2,threshold2,direction2,numOfTran)
```

where $signals1$ and $threshold1$ are the signal name and threshold voltage of the triggering event, $signal2$ and $threshold2$ are the signal name and threshold voltage of the target event, $direction1/2$ are the direction of transition (rise or fall), and $numOfTran$ refers to the number of transitions the second signal waits after the trigger. The return value of the primitive is an array of ($time$,$value$) ordered pair, where $time$ is the time instant when the triggering event happened and $value$ is the delay between the trigger and target events. Since the delay is measured from trigger to target, it is always positive. This process of measuring the delay repeats for the entire simulation results. For example, to measure the rise-time of a signal sig, the function call will be:

```
FindTime(sig,0.2*vdd,rising,sig,0.8*vdd,rising,1)
```

We measure the elapse time between when sig rises to 20% of vdd and the time the signal rises to 80% of the supply. The operation is illustrated in Figure 4.13. If the signal glitches such that there are two transitions to 80% of the supply after the trigger event, as in the case of the third pulse, the function returns the delay to the closest target as specified in the function call. On the other hand, if the signal fails to reach the second threshold by the time the first threshold is met again, as in the case of the second pulse, then the function returns “NA” as the second value of the order pair. Depending on how the primitive is being used, the wrapper calling the function can choose to ignore the “NA” or to interpret it as a failure. For example, if the primitive is used to find the delay between two signals, then the “NA” may be ignored since the absence of the target may be because that logically the second signal is not suppose to transition. However, if the primitive is used to check the existence of a pulse of a transition, then an “NA” must be interpreted as a violation.

Since the simulator evaluates the circuit at discrete time steps, the simulation result may not contain a data point at exactly the time when the waveform crosses the threshold. In this case, the function interpolates linearly between adjacent points. This primitive is used by the $FindDelay$ function in the library to process the timing information of the waveforms.

4.5 Implementation Complexity

The first version of the prototype contained about 4000 lines of Perl code and was built in three weeks. It implemented only the basic functionality of the parser and a subset of the pre-defined functions stored in the current version of the library. The prototype was refined and repartitioned over the following year as issues related to multiple projects, constraint protability, and analyzer performance were discovered. The C++ code was added in the form of primitives to improve the run time. Finally, user interface and documentation were addressed by creating the comments editor which took a few days to implement once we understood the required features. The final prototype contains about 1100 lines of Tcl/Tk code implementing the comments editor and other modifications to SUE, 5100 lines of Perl code implementing the parser layer, 2000 lines of Perl code in the library layer, and 4000 lines of C++ code to implement the two primitive functions. The Tcl/Tk and Perl code are platform independent and the C++ code is compiled for solaris and linux.

The prototype contains about 1100 lines of Tcl/Tk code to implement the comments editor and other modifications to SUE, 5100 lines of Perl code in the parser layer and 2000 lines of Perl code in the library layer, and 4000 lines of C++ code to implement the two primitive functions. The Tcl/Tk and Perl code are platform independent and the C++ code is compiled for solaris and linux. The first implementation was built in three weeks and contained most of the features described in this chapter, except the comments editor which took a few days to implement once we understood the required features.

During the development, most of the time was spent on coding the parser layer which resembled a mini compiler project. The implementation effort was split between processing parameters in the Comments and supporting nested functions. Since in the STAR system, there is no concept of datatype and name space, the parser needs to determine whether the variable is a vector or scalar and to search a large number of files to find the value of the variable for parameter substitution. We also tried a number of different approaches to passing data between the different functions that are nested before settling down to the current implementation described in the previous sections.

One problem that we did not address in this prototype is the debugging of simulation results. Currently, every time the circuit violates an assertion, STAR prints an error message stating the violation and the time it happened. It is up to the user to bring up the waveform and trace the signal path. Finding the cause of the violation and creating a dependency chain is a difficult problem that should be addressed in the future.

4.6 Summary

We implemented a prototype of STAR to execute the Active Comments described in Chapter 3. The system is designed to be flexible to enable the users to extend it to capturing new circuits. The prototype is composed of four layers: schematic, parser, library, and primitive. Partitioning the implementation into four layers enabled us to study what features are necessary to support the concept of Active Comments and encourage its adaptation by the designers.

In the schematic interface, we created a comments view to enable user to document both the Active Comments and the design. This way, the Comments continue to evolve with the circuit as it is transferred and reused.

The parser layer executes the Active Comments. One problem with the Active Comments is its flat name space which results in Measurement Comments from different schematics overriding each other's results. While the parser circumvents the problem by prepending a filename when reading and writing variables, a more robust approach may be to create different name spaces for the different type of parameters used in the Comments.

The key to the flexibility of our system is the extensible library. The users can create their own pre-defined functions and use them in the Active Comments. STAR imposes some coding style and annotation guidelines to help encourage the users to document their extended functions. Having description and comments for the functions can provide information to others using the system, but we may need to use the review process to encourage the designers to annotate their code.

Chapter 5

Phase-Locked Loop Design

In the previous chapters we described the process of designing custom circuits and a framework that simplifies the design tasks while capturing design knowledge. In this chapter, we demonstrate how to use STAR to specify the critical circuit parameters and the performance constraints a process independent manner. We use our proposed system to design a PLL in one process, then reuse it as a building block after porting it to a different process.

Our example PLL is composed of a number of analog and mixed-signal blocks. In Section 5.1 we describe each circuit block in detail along with the Active Comments used to capture the design. This PLL was later used as a module in a different testchip. The new testchip imposed a number of new requirements and resulted in a few changes to the original PLL. The VCO was replaced to one that generates multi-phase clock, and the loop filter was simplify to reduce power. We discuss the porting process in Section 5.2. STAR is used to guide the re-optimization of the PLL and ensure robust performance across the design hierarchy. Section 5.3 describes the results of the optimization and explores the benefits and the limitations of the prototype system in the context of designing and porting the PLL.

5.1 Phase-Locked Loop Design

We designed the PLL to be used as clock generator in processor cores and I/O subsystems. Our primary design objective was the quality of the generated clock, specifically, phase

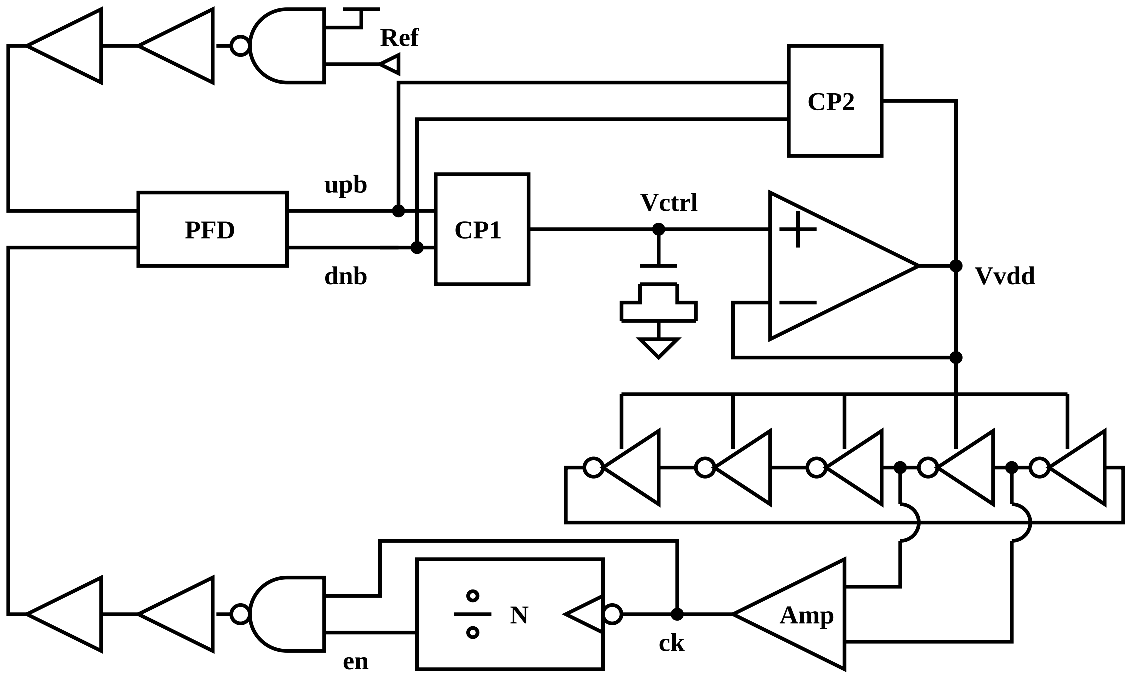


Figure 5.1: PLL Block Diagram

offset and jitter. Static phase offset is usually the result of asymmetry in the design and transistor mismatches, while jitter is commonly caused by noise, particularly supply noise, coupled into the circuits. We also must design the PLL to have a well damped loop response and the circuits to have good supply noise immunity.

Figure 5.1 shows the block diagram of the example PLL which was implemented in a 0.35μm technology [36]. Since jitter is often dominated by power supply coupling into the VCO, this design uses a regulator to both control the oscillation frequency and to isolate power supply noise. The VCO is composed of a five-stage inverter ring to produce a single-ended clock. Following the VCO is a two-stage amplifier that converts the low-swing VCO output to full CMOS levels. The programmable divider outputs a pulse that fully overlaps the high phase of the VCO clock to qualify the clock when the number of rising clock edges reaches the divide value. The clock buffers after the qualifying NAND gate complete the clock feedback path. The phase-frequency detector (PFD) compares the feedback clock and the reference clock and controls the charge-pump to dump either positive or negative charge onto the capacitor. The matching of delay between the clock buffers and buffers in the feedback path is critical for low phase offset, as is the design of the PFD. The output of the first charge-pump is integrated onto the filter capacitor to produce the integral control voltage, $Vctrl$. The control voltage at the filter capacitor is replicated using an unity-gain amplifier whose output impedance along with the output of the second pump provides the proportional control to the loop. To achieve a wide operating range, all the analog components are biased using $Vctrl$ [34].

The closed-loop transfer function of the linear second-order PLL is shown to be [25]:

$$H(s) = \frac{1 + 2 * \zeta * s/\omega_n}{1 + 2 * \zeta * s/\omega_n + (s/\omega_n)^2} \tag{5.1}$$

where the loop damping factor ζ and bandwidth ω_n are given by:

$$\zeta = 0.5 * R * \sqrt{I_{CP} * K_{VCO} * C_{CP}/N} \tag{5.2}$$

$$\omega_n = 2 * \zeta/(R * C_{CP}) \tag{5.3}$$

where I_{CP} is the charge-pump current, K_{VCO} is the VCO gain, C_{CP} is the capacitance of the loop filter, R is the resistance of the loop filter, and N is the frequency multiplication factor.

Regrouping the building blocks in the design, we can map the components into the simplified PLL model presented in Chapter 3. The PFD implements the phase comparator block. The low-pass filter is implemented by the two charge-pumps, the NMOS capacitor, and the op-amp. The ring oscillator together with the level converter form the VCO block. The divider block includes the actual divider and the qualifying NAND gate. Figure 5.2 shows the mapping.

As part of the PLL design representation, Active Comments are embedded in all levels of the design hierarchy. We use Measurement Comments to measure critical circuit parameters such as the VCO gain, the charge-pump current, and the loop filter capacitance. Assertion Comments are used to monitor the circuit performance as well to set the constraints at the interfaces between different sub-blocks. When we first captured the design, not all of the Active Comments where written in the scalable form. In order for the Active Comments to be applicable when the design is reused, all the absolute constraints were

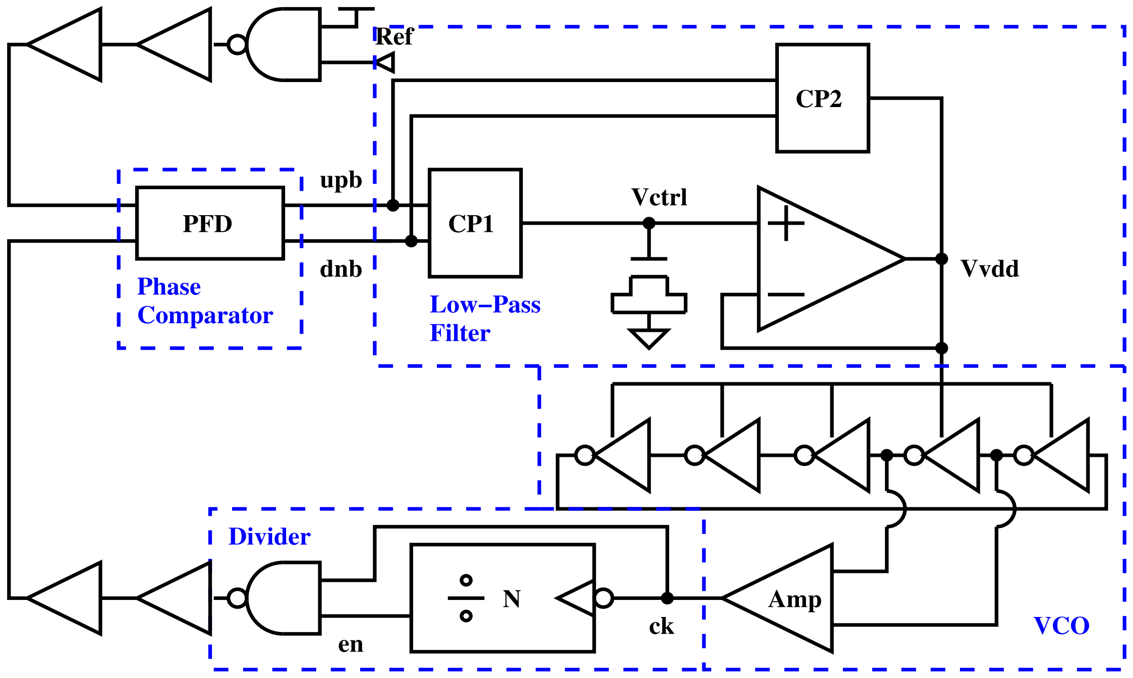


Figure 5.2: PLL Block Diagram with Abstraction

recoded to be more scalable. In the following subsections, we describe each circuit component in detail and explain how we use Active Comments to capture this PLL design. We start from the input to the loop, the phase-frequency detector.

5.1.1 Phase-Frequency Detector

The PFD schematic is shown in Figure 5.3(a). In high frequency linear PLL design, the Phase-Frequency Detector (PFD) capture range sets the frequency limit of the reference clock and affects the lock acquisition time [45]. Ideally, the sequential PFD implemented by our circuit has a capture range of +/-2π [46]. However, in practice the capture range is often affected by the operating frequency. Figure 5.4 plots the capture range of the PFD at two input frequencies and shows the capture range reduces as the frequency increases. The capture range is reduced because the delay to assert and de-assert the PFD output becomes a larger fraction of the cycle time as the operating frequency increases. To illustrate the different components of this delay, we plot the PFD timing diagram in Figure 5.3(b). This fixed delay is composed of the delay from clock to output, the output pulse, and some time

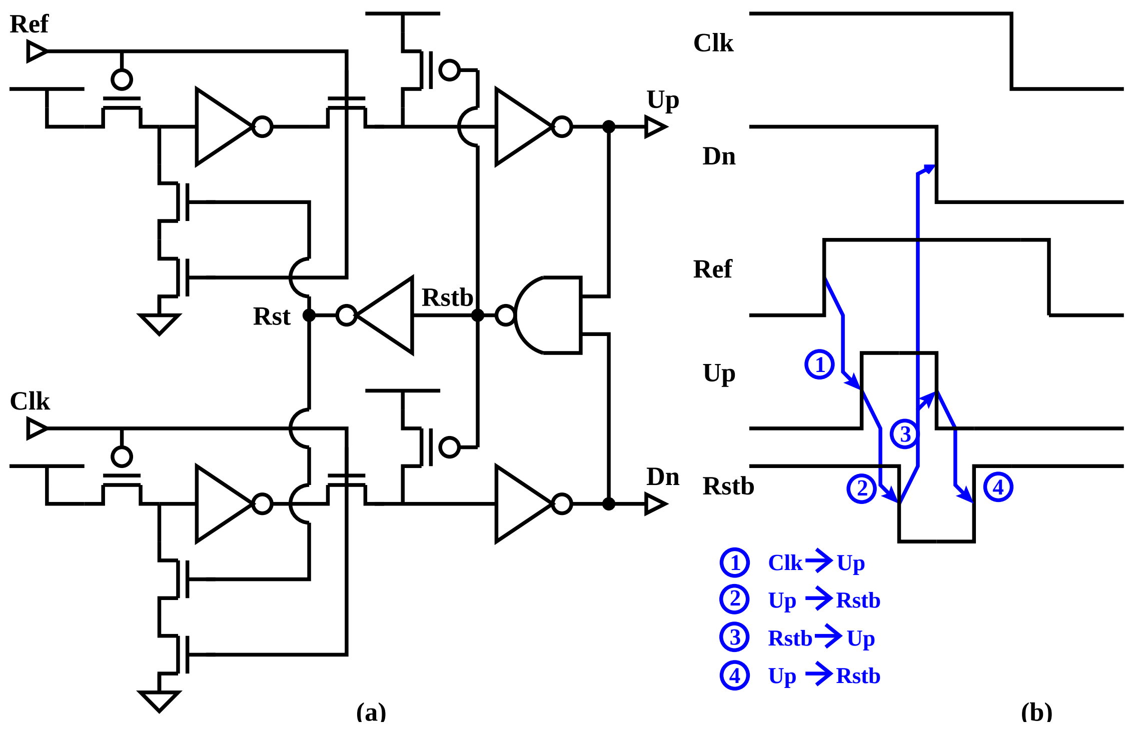


Figure 5.3: PFD Schematic and Timing Waveforms

to de-assert the self-reset so the next clock edge can assert the output. We can bound the capture range of the PFD by:

$$T_{fix} = T_{clk->up/dn} + 2 * T_{up/dn->rstb} + T_{rstb->up/dn} \tag{5.4}$$

$$Range = [-2\pi * \frac{(T_{period} - T_{fix})}{T_{period}}, +2\pi * \frac{(T_{period} - T_{fix})}{T_{period}})] \tag{5.5}$$

Using the node names in the figure, $T_{clk->up/dn}$ is the delay between input rising to output asserting. The pulsewidth is expressed as $T_{up/dn->rstb}$+$T_{rstb->up/dn}$. And an additional $T_{up/dn->rstb}$ delay is need to de-assert the reset. T_{fix} is the minimum delay between consecutive input rising edges or equivalently the maximum operating frequency of the PFD.

To achieve fast phase acquisition, we want the capture range of our PFD to be at least +/-π at the high reference frequency. We use the Measurement Comments to find the PFD capture range:

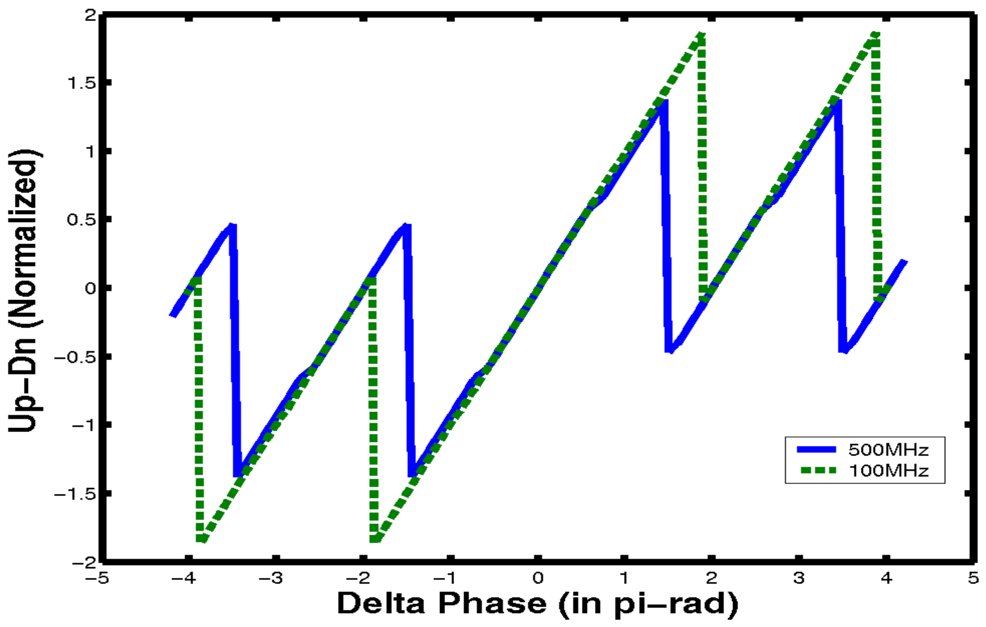


Figure 5.4: PFD Characteristic at 500MHz and 100MHz

```
# DEFINE tLo=LoRange(Offset(ref,ck),PulseDiff(up,dn)), \
    tHi=HiRange(Offset(ref,ck),PulseDiff(up,dn)) w/ SweepClk(ref,ck,10*FO4)
```

The $LoRange()$ and $HiRange()$ functions find the lower and higher inflection points, respectively. The $SweepClk()$ sweeps the two input clocks, ref and ck, from -2π to 2π. The delay is based on the operating frequency, which is set to 1/(10*FO4) in this example. The $PulseDiff()$ adds HSpice measurements to find the difference in the output pulse widths. And the $Offset()$ adds HSpice measurement to find the offset between ref and ck.

5.1.2 Low-Pass Filter

Following the PFD is the low-pass filter which converts the phase error to a change in control voltage. The low-pass filter in our PLL design is composed of two charge-pump, a NMOS capacitor, and an op-amp functioning as a voltage buffer.

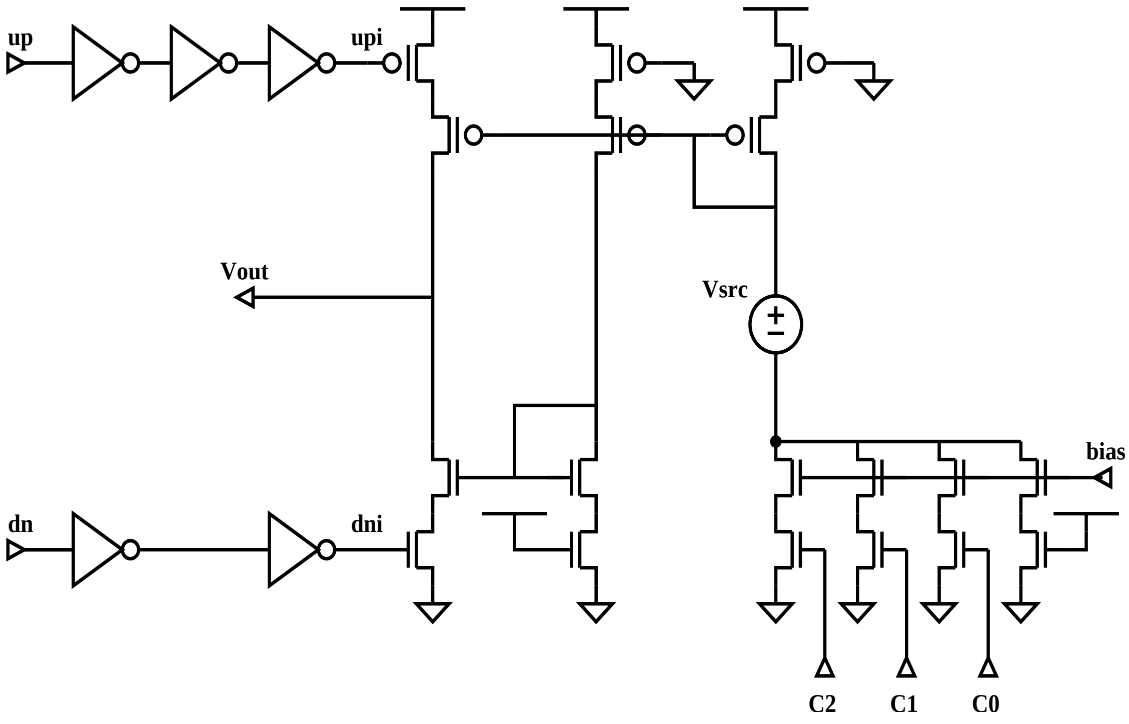


Figure 5.5: Charge-Pump Schematic

Charge-Pump

The two charge-pumps are controlled by the PFD. The charge from the first pump, CP1, is integrated onto the filter capacitor while the output of the second pump, CP2, is summed with that of the op-amp to drive the VCO. While the two charge-pumps are sized independently, they share the same circuit topology. Figure 5.5 shows the transistor level schematic of the push-pull type charge-pump used in this PLL design. The charge-pump is biased through a current mirror based DAC whose bias is established by the loop control voltage $Vctrl$. The charge-pump current scales as the square of the change in operating frequency thus tracks the loop dynamics according to Equation 5.2. The DAC control words for the two charge-pumps can be configured independently and are programmed to compensate for varying frequency multiplication factor N to enable the loop to always achieve close to optimal characteristics.

The most important circuit parameter associated with the charge-pump is the current, Icp, that it sinks from or sources onto the filter capacitor. To measure the pump current, we add a voltage source at the charge-pump output. This voltage source is ramped from vLo to vHi, to measure the output current across the operating range of the PLL. We bias the circuit with the filter voltage using a voltage-controlled voltage source (vcvs) controlled by the charge-pump output. The PMOS current source is turned on and current flows into the voltage source at the output. This test setup is specified with the following Measurement Comment:

```
# DEFINE Icp1=I(RampV(Vout,vLo,vHi)) w/ SetE(bias,gnd,Vout,gnd,1), \
    SetV(up,vdd),SetV(dn,gnd)
```

The $RampV()$ function adds a voltage source to $Vout$ and ramps the voltage from vLo to vHi. The function returns the name of the voltage source which is passed into $I()$ to monitor the output current. The $SetE()$ function adds a vcvs, or e-element in HSpice terminology, between $bias$ and gnd and sets the voltage potential between these two nodes to be the same as that between $Vout$ and gnd. The second charge-pump CP2 similarly defines Icp2. Instead of drawing a separate test bench schematic to include these extra voltage sources, we created the simulation environment by adding the Measurement Comment directly into the production schematic of the charge-pump. This helps reduce the number of test benches in the archival database. Although the Measurement is part of the production schematic, it is not executed when the charge-pump is integrated into the PLL.

Measuring just the pump current is not sufficient to capture the design. There are two practical issues with implementing the circuit. One is to match the up and down currents to minimize the static phase offset, and the other is to limit the noise on the bias voltage in order to keep the current variation small.

When PLL is locked, the up and down currents are overlapped. If the currents do not match, then the residual charge is integrate on the filter capacitor and causes a static phase error. We use a Measurement Comment to find the value of the mismatched current by turning on both current sources and monitor the current into the voltage source at the output:

```
# DEFINE Idiff=I(RampV(Vout,vLo,vHi)) w/ SetE(bias,gnd,Vout,gnd,1), \
SetV(up,vdd),SetV(dn,vdd)
```

Excessive mismatches in the currents not only causes phase error but also affects the loop dynamics since the linear model assumes only one current value. We use the above Comment to measure the static current offset and minimize its value during circuit optimization.

Even if the up and down currents are matched statically, they can still vary during operation. Any noise coupled into the bias voltage distribution affects the output current. We must ensure that the noise on the bias voltage is kept small such that the change in the bias current is less than 1%. An Assertion is added to check the bias current:

# ASSERT NoiseI(i(vsrc)) $<=$ 1%

As described in Chapter 4, $NoiseI()$ is a wrapper function that uses $RunningAvg()$ and $I()$ to find the instantaneous noise on the current through the zero-voltage source, $vsrc$.

Filter Capacitor

The capacitor integrates the phase error to generate the PLL control voltage. This voltage is also used to bias all the analog components to enable the loop dynamics to scale across process variations and operating frequencies. The capacitance value is an important design parameter in determining the loop dynamics. We choose a NMOS capacitor because all the circuits are referenced to ground for better supply noise isolation. The source, drain, and bulk terminals of the transistor are connected to ground, and the gate terminal is connected to the output of the charge-pump. The transistor is operating in inversion mode, and the capacitance is mostly determined by the oxide thickness and the gate area [47]. We use a Measurement Comment to find the gate capacitance of the NMOS transistor:

```
# DEFINE Ccp=MeasureCap(MCcp)
```

The $MeasureCap()$ function is a wrapper that calls uses pre-defined functions to setup the simulation stimuli and measure the capacitance. The function first initializes the gate terminal to vLo, using $InitV()$, then charges the gate with a constant current I, with $SetI()$.

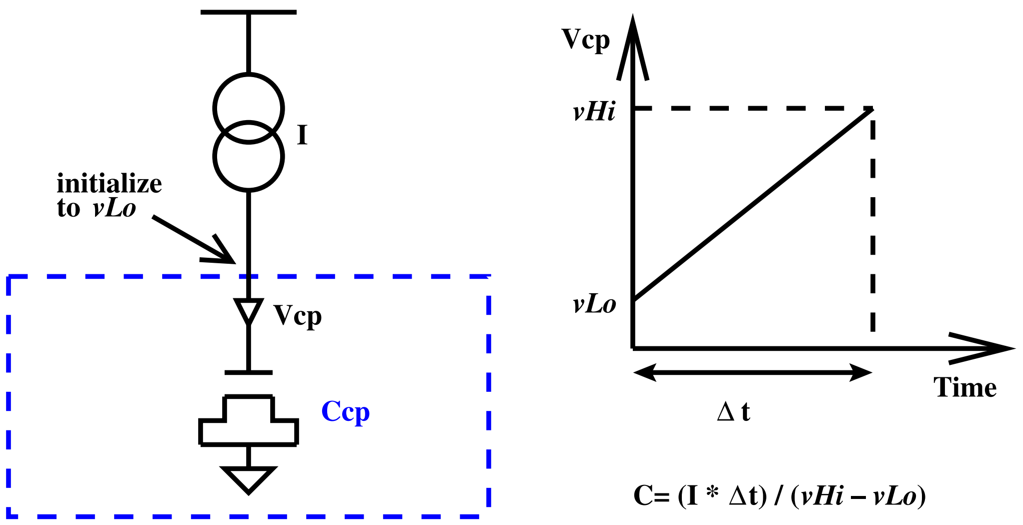


Figure 5.6: MOS Gate Capacitance Measurement Model and Waveform

After the simulation, $MeasureCap()$ finds the time it takes to charge the gate voltage to vHi. This Measurement Comment is applied directly to the capacitor without a test bench. Figure 5.6 shows a visual representation of the Comment and the simulation waveform. Knowing the current, I, the change in voltage, ΔV, and the time to charge the capacitor, ΔT, we calculate the capacitance as:

$$Ccp = I * \Delta T / \Delta V \tag{5.6}$$

Equation 5.6 assumes a linear capacitor such that the change in voltage is a ramp. This assumption is valid within the VCO's operating range bounded by vLo and vHi [47].

Connecting the loop filter directly to the source terminal of the PMOS devices in the VCO will deplete the charge stored on the capacitor. Instead, the loop filter must be connected to a high impedance node. We use an op-amp to buffer the analog voltage. The amplifier is described next.

Voltage Buffer

At the heart of this PLL architecture is the voltage buffer which is implemented using the op-amp which shown in Figure 5.7. This buffer servers four functions: suppressing the

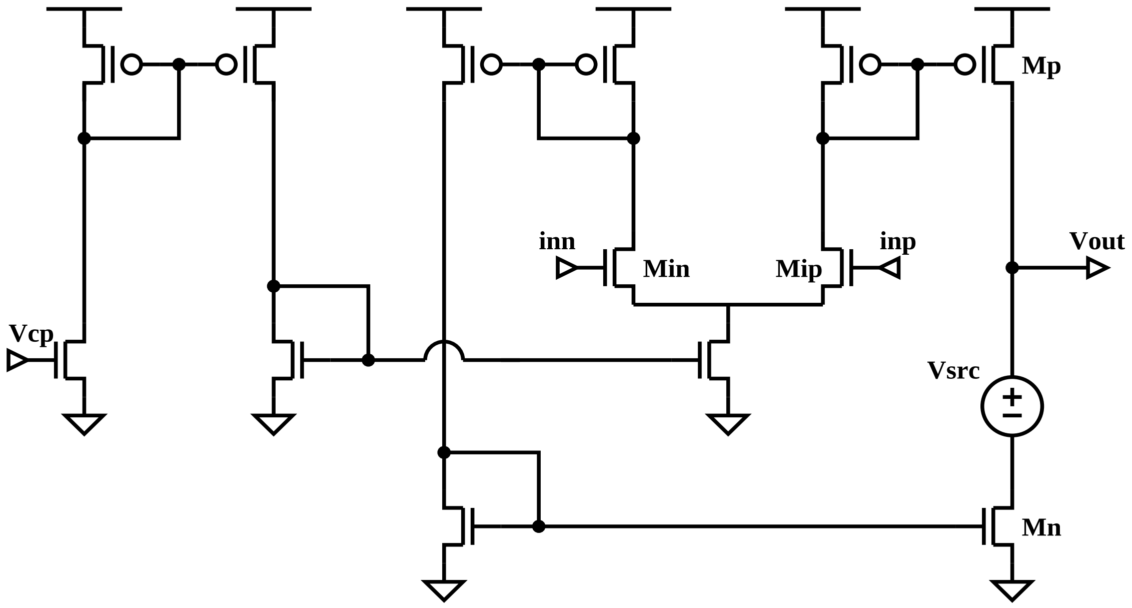


Figure 5.7: Voltage Regulator

effects of supply noise, replicating the filter voltage, providing the loop stabilizing zero, and providing drive current to the VCO.

We use Active Comments to help ensure that these functionalities are properly carried out. The op-amp rejects the supply noise by increasing the impedance looking into the supply. In Chapter 3 we described how to use Active Comments to measure the PSRR and monitor the circuit's saturation margin and output impedance. So this section focuses on the three remaining functions of the amplifier.

i. Closed-Loop Voltage Gain

Arranged in the unity-gain configuration, the op-amp buffers the filter voltage. Ideally, the output of this voltage buffer should equal its input, or equivalently the op-amp should have a closed loop gain of 1. Since the output of the op-amp is within the larger PLL feedback loop, we could relax the voltage offset requirement between the input and output. However, this offset should still be bounded as a measure of how well the op-amp tracks the input in the presence of high frequency noise coupled into the circuit. The following Assertion Comment is added to check that the voltage offset between the output and input is less than

1%:

# ASSERT abs(v(inp)-v(Vout))/v(inp) $<=$ 1%

We defined the offset as a ratio of the voltage difference between the input and the output to the voltage at the positive input terminal. Using a ratio instead of an absolute value allows this constraint to scale across technology. It is important to note that this constraint is assuming perfect transistors without any mismatches.

ii. Loop Compensation Zero

The transfer function of the PLL contains two poles at the origin. In order to stabilize the loop, this PLL architecture provides a feed-forward zero with the output impedance of the amplifier. Implementing a PLL stabilizing resistor through active components has been originally proposed in [33] and adopted by the self-biased differential PLL in [34]. As illustrated in Figure 5.8(a) the control voltage of a conventional PLL at any point in time is determined by the aggregate charge stored on the loop capacitor plus the instantaneous voltage across the filter resistor. This configuration is equivalent to the one depicted in Figure 5.8(b) where the integral voltage across Ccp is first buffered by the unity gain amplifier and then augmented by the instantaneous voltage formed by the second charge pump and the amplifier output impedance. We can calculate Vcp as follows:

$$Vcp(t) = \frac{\int_0^t Icp(\tau)\,d\tau}{Ccp} + Icp(t) * R = \frac{\int_0^t Icp1(\tau)\,d\tau}{Ccp} + \frac{Icp2(t)}{g_{mOP}} \tag{5.7}$$

The amplifier output impedance is the impedance of the input transistor (1/g_m) divided by the inter-stage mirroring ratio RM which is set to 3 in our implementation. Since RM is kept low, the amplifier is virtually a single pole system and does not require stabilizing compensation. To measure the output impedance of the amplifier, we constructed a test bench to put the op-amp in unity gain configuration and add the following Measurement Comment:

# DEFINE roamp=MeasRout(Mip) w/ RampV(Vcp,vLo,vHi)

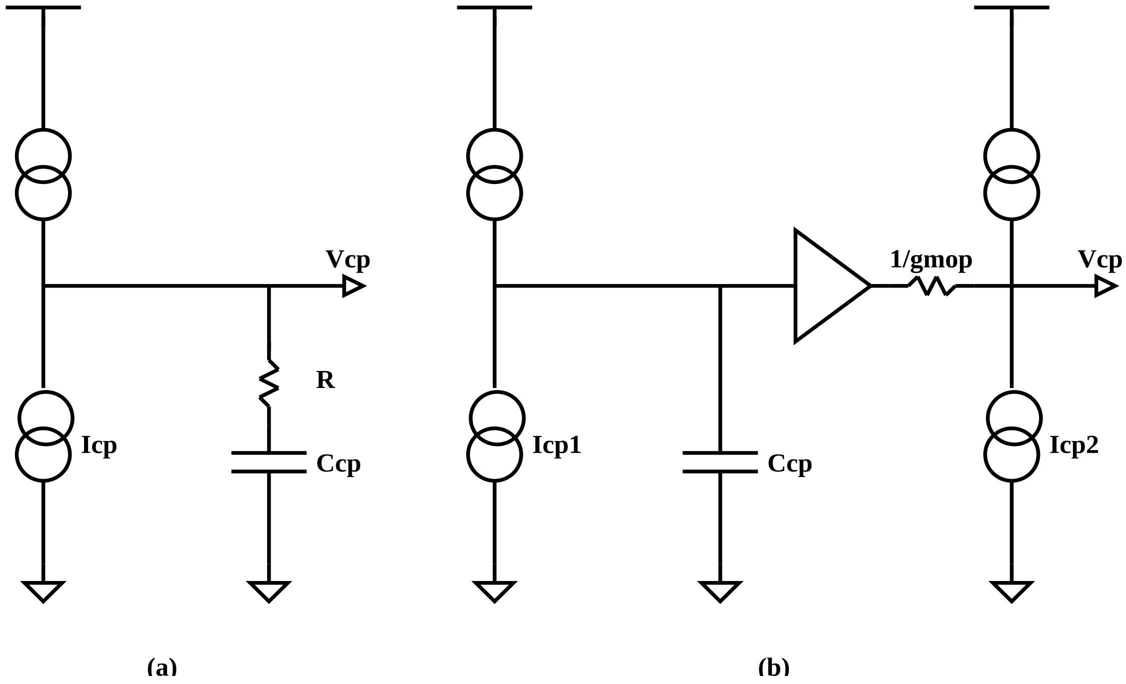


Figure 5.8: Implementing the PLL Stabilizing Zero

# CALCULATE R=Icp2/Icp1*roamp/3

Using $MeasRout()$ function, we measure the output impedance of the input transistor, Mip (Figure 5.7) for the entire operating range of the PLL. The impedance is multiplied by the ratio of the charge-pump currents to determine the effective resistance R. The resistance value is optimized based on Equations 5.2 and 5.3. This Measurement Comment not only automates the measurement of the resistance, but also documents the $Rout$ of Mip as an important design parameter for this PLL architecture and that any modification to the op-amp must take this into consideration.

iii. Amplifier Output Current

The op-amp buffers the voltage and in turn provides the current to drive the oscillator. The inverter-based ring generates noise on its control voltage while switching. To suppress this noise, the current into the NMOS transistor, Mn, must be at least twice the load current into the VCO. The VCO will cease to oscillate if the VCO requires more current than what the amplifier can provide. To guarantee that the current requirement is satisfied when the two cells are used together, we placed two zero-volt voltage sources as current probes. One voltage source, $vsrc$ (xopamp.vsrc), is in series to Mn (Figure 5.7) and the other (xvco.vsrc) is at the input of the VCO. We add the following Assertion into the parent cell that instantiated the buffer and the VCO to check the current requirement:

# ASSERT i(xvco.vsrc)/i(xopamp.vsrc) $<=$ 1/2

Using Assertion to check the current ratio ensures that the constraint is satisfied at the interface of these two circuit blocks, independent of the operating frequency and process technology. The buffer drives the VCO which is described next.

5.1.3 Voltage-Controlled Oscillator

The voltage-controlled oscillator (VCO) generates the internal clock in the PLL. In our PLL architecture, the VCO is comprised of five CMOS inverters arranged in a ring configuration followed by a level shifter.

Ring Oscillator

Figure 5.9 shows the five-stage ring oscillator. The oscillation frequency is controlled by modulating the supply voltage. In order to keep the PLL predictable by a linear model, we limit the range of its oscillation frequency. In Chapter 3, we used a pair of Measurement Comments to find the control voltage range and store low and high limits in two variables: vLo and vHi. We will use these two variables to control the circuit parameter measurements for all the circuit blocks in our PLL.

The oscillation frequency at a particular control voltage is determined by the Ron of the inverter at that voltage and the load capacitance driven by the delay element. The quantity of how much the VCO frequency changes given a change in the control voltage is called the VCO gain or $Kvco$. To find this parameter, we return to the VCO test bench schematic and regenerate the VCO transfer curve. The $Kvco$ is the slope of the curve and can be measured by:

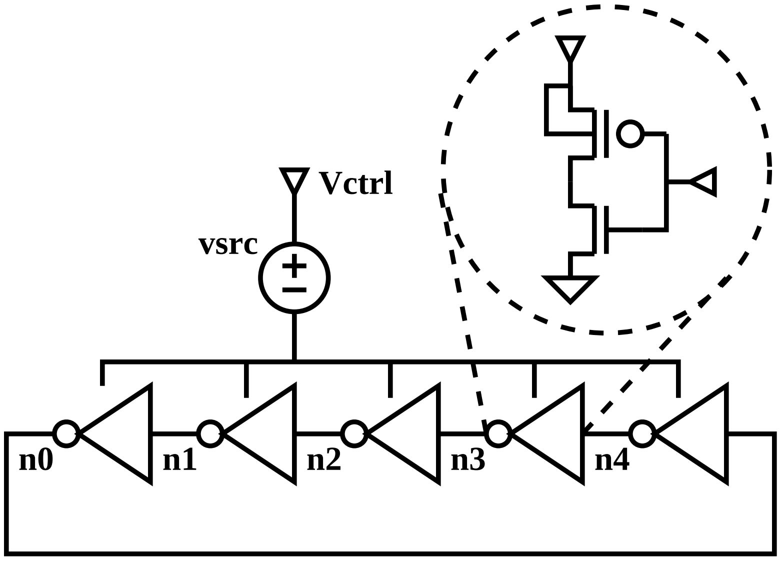


Figure 5.9: CMOS Inverter-Based VCO

# DEFINE Kvco=Slope(vctrl,Freq(ck)) w/ SweepV(vctrl,vLo,vHi)

We measure the slope across the range of the control voltage from vLo to vHi and store the values as a vector. $Kvco$ is a design parameter that we adjust when optimizing the PLL's performance.

One problem with the ring-type oscillator is when the frequency is too high, the internal edges do not swing to rail. This leads to an increased sensitivity to supply noise and distortion of the duty cycle. One must prevent the oscillator from operating at those high frequencies. The following Assertion Comments monitor the voltage swing of the VCO:

# ASSERT Swing(n4)

The $Swing()$ function is a wrapper around the $FindTime()$ primitive. The wrapper uses $FindTime()$ to find the 2-98% risetime of $n4$, where the 100% voltage level is the instantaneous voltage at the source terminal of the PMOS transistor. If there is a rising transition, but the signal does not reach 98%, then the Assertion fails. Checking whether the signal

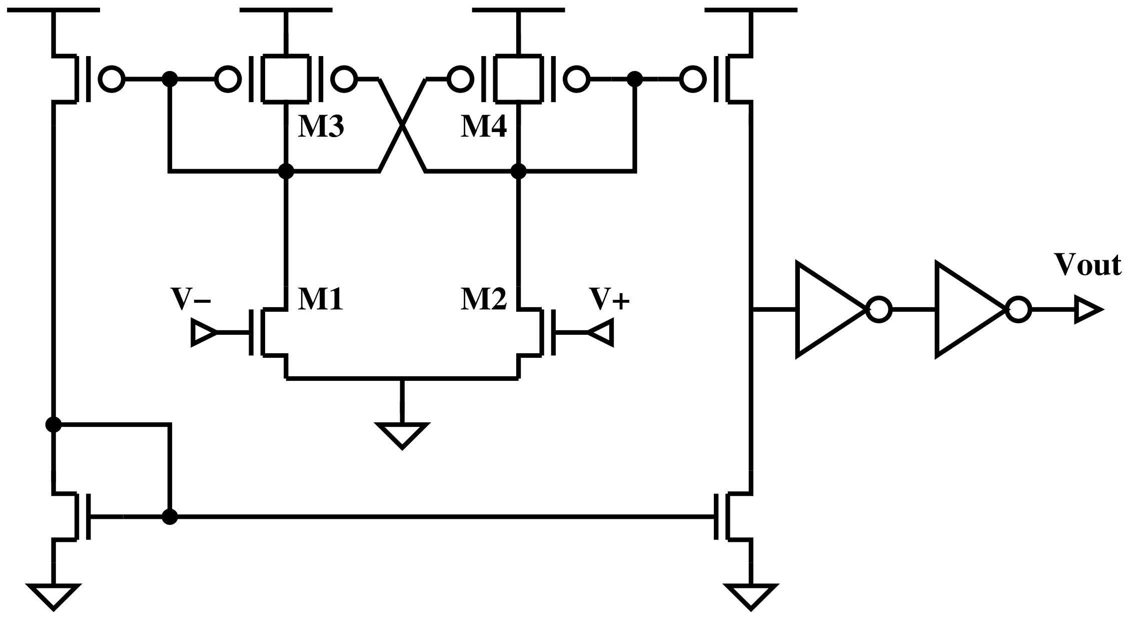


Figure 5.10: Level Shifter Schematic

swings to at least 98% of the source voltage makes the Assertion independent of the process technology and the operating frequency. Since the maximum swing of the oscillator is limited to the control voltage, its output must be converted to the full supply level before driving any CMOS logic.

Level Shifter

The level shifter, depicted in Figure 5.10, is a simple two-stage current-mirror based amplifier that converts the partial swing signals generated by the ring oscillator to full CMOS level. Similar to the VCO, the output of the level shifter must also reach 98% of its supply, which in this case is the full CMOS level. The finite gain-bandwidth product of the level shifter and the larger required swing mean that the circuit may fail to propagate the clock at high frequencies before the oscillator fails its swing assertion. To guarantee the output of the level shifter completes its transition, we added the following Assertion:

```
# ASSERT Swing(Vout)
```

In the Assertion call, we only need to specify the node to check since the $Swing()$ function automatically finds the correct swing level by tracing the PMOS transistor connected to the node.

Shifting voltage levels distorts the duty cycle of the input. The two inverters following the second stage are sized to offset the distortion introduced in the amplifier. We add two Assertions to check the duty cycle of the final output:

```
# ASSERT PulseWidth(Vout)/Period(Vout) >= 48%
# ASSERT PulseWidth(Vout)/Period(Vout) <= 52%
```

The $PulseWidth()$ and $Period()$ are wrappers which use $FindDelay()$ to find the high phase and period of the $Vout$ signal, respectively. Since the PFD used in this design only compare the rising edges, we relax the duty cycle constraint to allow a +/-2% variation.

To enable frequency multiplication, the output of the VCO is divided down in frequency before comparing to the reference by the PFD. The frequency division is accomplished using a divider.

5.1.4 Divider

The divider in the feedback path divides the internal clock to enable the VCO to oscillate at N times the reference clock frequency. In this design, we implemented the divider as a binary counter using static CMOS logic. The counter can be programmed to count to 1, 2, 4, or 8. When the number of clock edges reached the pre-programmed divide value, the counter output is asserted for one VCO cycle. This way, the period of the counter output equals to that of the reference clock, with the high phase of the signal being one VCO cycle long.

As with most digital blocks, and the two most important design objectives of the divider are correct functionality and meeting cycle time. By coding the logic of each state bit into the conditional clause of an Assertion, we could check the signal transition at the input of each flop. However, there are a number of commercial tools available to help analyze the functionality and delay of the circuit [48][49], and the designers should use them to

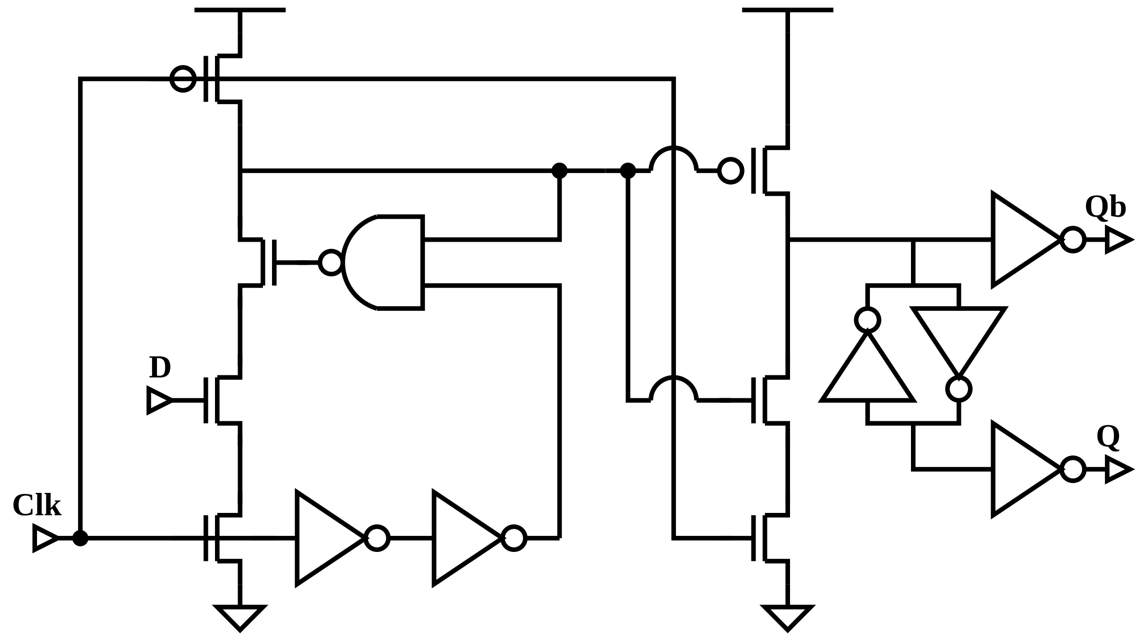


Figure 5.11: Semi-Dynamic Flip-Flop

guarantee design robustness. Our goal is not to replace those tools, but to capture the design constraints so that we can monitor the circuit during the system level mixed-signal simulation. In this design, we verify the logic implementation with a functional simulator Verilog-XL [50] and use Active Comments to validate the timing.

Since the goal is to check the timing of the circuit, constraints are added to the clocked element in this design. To support the fast cycle time, we used the semi-dynamic flip-flop [51], shown in Figure 5.11. Intuitively, we would bound the data arrival time of the flop to ensure that it meets the setup time. We specify the timing constraint as follows:

# ASSERT delayRR(D,ck) $>=$ 0.5*FO4
# ASSERT delayFR(D,ck) $>=$ 0.5*FO4

where $delayRR()$ finds the delay between every rising edge of ck to rising edge of Q, and $delayRF()$ is similarly defined. If the combination logic results in the data input, D, transitioning multiple times before the clock rises, the $FindTime()$ primitive will generate “NA”s for all but the last data transition since D is the trigger signal. However, the

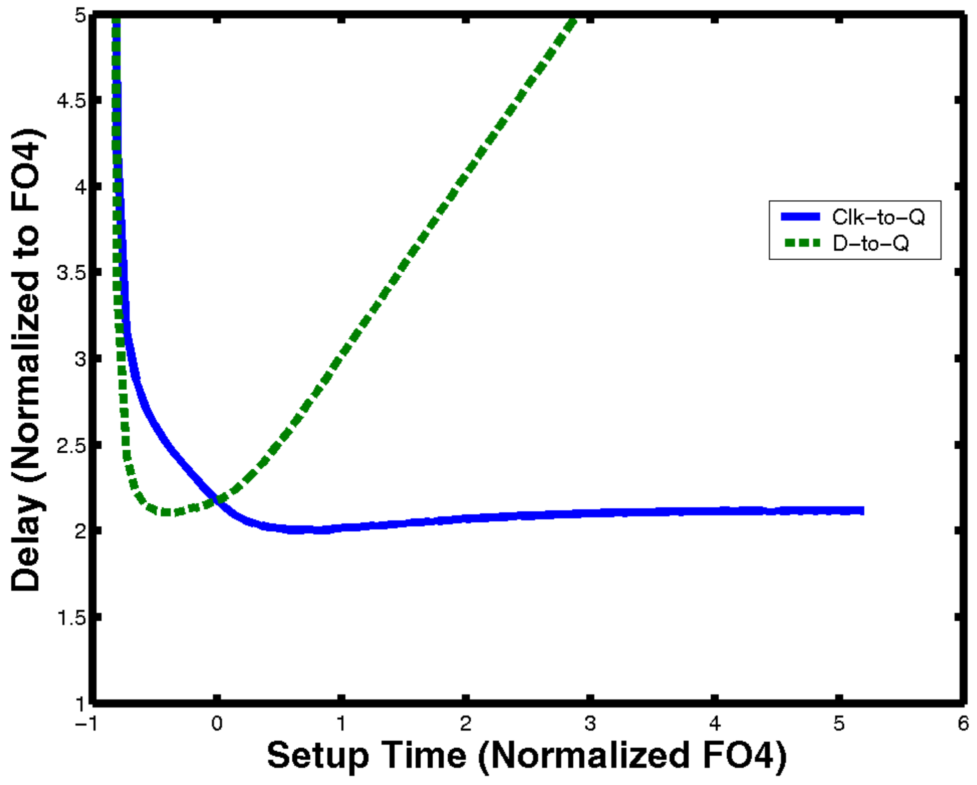


Figure 5.12: Normalized Clk→Q/D→Q Vs. Setup Time

wrapper function safely ignores these “NA”s since only the last data transition needs to be constrained. The Assertions ensure that the last data transition occurs half an inverter delay before the clock rises.

While this set of constraints correctly ensures the proper design margin, we may be able to relax the constraints. Figure 5.12 shows the clk→Q and D→Q of the flop with respect to its setup time, normalized to the FO4 delay. The clk→Q of the flop increases as the data arrives later relative to the clock (more negative setup time). Note that the output of the flop still transitions even with some negative setup time; in fact the minimum D→Q occurs when the data arrives about half an inverter delay *after* the clock. It is reasonable to recast the setup time problem into bounding the clk→Q delay. This way, we can relax the constraint on the data arrival time as long as the the clk→Q delay is not pushed out to cause failure in downstream logic. Adding the following Assertion Comments to the flip-flop bounds the output delay:

# ASSERT delayRR(ck,Q) $<=$ 2.5*FO4
# ASSERT delayRF(ck,Q) $<=$ 2.5*FO4

Using the above Assertions ensures that the clk→Q is never more than 2.5 times the FO4 delay and indirectly ensures sufficient setup time for the flop to capture the data. We chose the constraint to be $2.5 * FO4$ because it is the clk→Q at the minimum D→Q.

Divider-VCO

Although the divider output has the same frequency as the the reference, we should not compare it to the reference clock directly because this signal has a static phase offset equal to the Clk→Q delay of the flop. Instead of comparing the output pulse to the reference, we use it as an enable signal to qualify the feedback clock. This imposes a set of timing constraints between the enable signal and the feedback clock to ensure correct operation of the PLL. Specifically, the enable signal must completely overlap the high phase of the VCO clock. These edges can be constrained with two Assertions.

To check the assertion edge of the enable signal, we must make sure that the rising edge of en comes before that of ck, as shown in Figure 5.13(a). Constructing the assertion as follows captures this behavior:

# ASSERT delayRF(en,ck)-delayRR(en,ck) $>=$ 0

Checking the rising edge of the enable signal against both the rising and the falling edges of the feedback clock ensures that the enable's rising edge is placed during the low phase of the clock. If this set of timing relationship is not observed, the loop may still lock to the correct frequency, but there will be an inherent static phase offset between the clock distribution output and the reference clock. This phase offset could be difficult to detect because the inputs of the PFD may be perfectly in phase.

However, checking only the setup of the signal is not sufficient. If the enable pulse is too short such that it does not overlap the rising edge of the feedback clock, then there

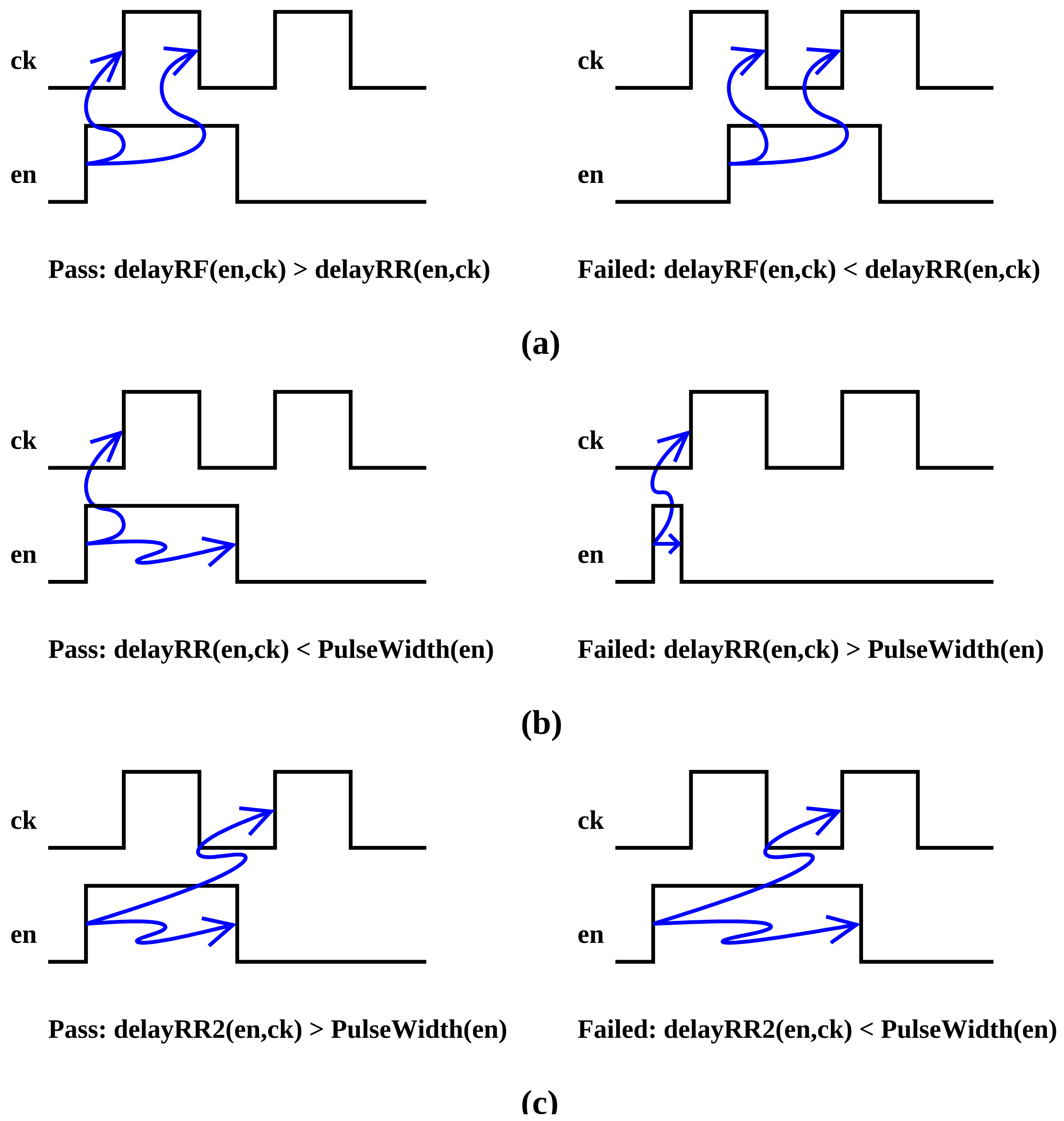

ck
en
Pass: delayRF(en,ck) > delayRR(en,ck)
ck
en
Failed: delayRF(en,ck) < delayRR(en,ck)
(a)
ck
en
Pass: delayRR(en,ck) < PulseWidth(en)
ck
en
Failed: delayRR(en,ck) > PulseWidth(en)
(b)
ck
en
Pass: delayRR2(en,ck) > PulseWidth(en)
ck
en
Failed: delayRR2(en,ck) < PulseWidth(en)
(c)

Figure 5.13: Timing Diagram of Enable and Clock

would not be any signal at the input of the PFD. Figure 5.13(b) shows that the enable signal must fall after the rising edge of the feedback clock in order to qualify it. On the other hand, if the pulse is too wide such that it overlaps with the high phase of the next feedback clock, then the PFD will incorrectly sense a higher frequency clock. Figure 5.13(c) shows that the falling edge of the enable signal must occur during the next low phase of the feedback clock. These two timing constraints can be enforced by adding the following Assertions:

```
# ASSERT delayRR(en,ck)-PulseWidth(en) <= 0
# ASSERT delayRR2(en,ck)-PulseWidth(en) >= 0
```

where $delayRR2()$ finds the delay of en rising to the second rising edge of ck, and $Pulse-Width()$ measures the pulse width, or the delay between the rising and falling edges, of the signal. Both of these functions are wrappers around $FindDelay()$. Checking the relative timing of these signals instead of imposing a specific delay value allows the constraints to automatically scale with the VCO frequency and the divide ratio.

5.2 Design Reuse

Often when reusing a design to target a new application, we need to modify the circuits beyond simply scaling the device sizes. While we would like to leverage the entire proven design to reuse it as-is, new applications and/or new process technology may force us to modify parts of the circuit in order to meet the new performance envelope.

The example PLL described in the previous section was fabricated and proven to work in the 0.35μm technology. We intend to reuse this PLL as a component in a different system, using the Active Comments already embedded in the design database along with the prototype framework to help with the reuse process. The database of the original PLL was transferred to a different designer who ported and reused the circuits in a new application. In the target application, the PLL is to be used as a plug-in module in an optical transceiver testchip. The testchip will be fabricated in the 0.25μm process targeting an I/O bit-rate of 5Gb/s/link while achieving low power consumption to enable large-scale integration. The cycle time of the core logic is limited by the minimum clock waveform that can be distributed by the inverters which is typically 6 to 8 FO4 delays [52]. In the 0.25μm technology, the FO4 delay is about 125ps which sets the maximum frequency to 1GHz. Data rates higher than the core frequency can be achieved by multiplexing the I/O where a set of parallel transmitters and receivers operating at a lower frequency rotate their duties in time [53][54]. The timing of the active period of each transmitter and receiver is controlled by equally-spaced, multiple phases of the lower frequency clock.

Our example PLL is used to generate the multi-phase clock to control the timing of the transmitter and receiver. To achieve the 5Gb/s I/O bandwidth in a 0.25μm technology requires the transceiver to operate at five times the internal core frequency. To control the sequencing of the 5:1 multiplexing and de-multiplexing operations, the PLL needs to generate ten evenly spaced clock phases. However, the single-ended VCO in our example PLL only generates five of the ten needed clock phases. To produce the needed clock phases, we couple two of these five-stage ring oscillators as proposed in [55].

Using the differential oscillator results in a much higher power consumption than originally anticipated. With two rings and the coupling inverters, the number of gates in the oscillator is quadrupled. The differential VCO required four times more current than the single-ended version and thus failed the current ratio Assertion at its interface to the voltage buffer. Keeping a low inter-stage multiplication factor in the op-amp forces us to up-size all the devices in the amplifier by a factor of four to provide the current needed for the VCO to oscillate. In addition to driving the VCO, the voltage buffer also provides the stabilizing zero of the PLL. Up-sizing the circuit compromises the loop stability. In order to maintain the same loop dynamics, the sizes of both charge-pumps must also be quadrupled. As a result of changing the oscillator, all the analog cells are increased by a factor of four. With the larger voltage buffer and charge-pumps, the differential PLL meets the frequency specification at the cost of quadrupling the power consumption.

We would like to modify some of the circuits, possibly changing the circuit topology, to reduce the power consumption. Since the voltage buffer consumes the most power, we replaced it with a more power efficient design [56][57]. Instead of using the load current as in the original design, the modified topology uses a capacitive load at the output to filter out the high frequency noise. To further reduce power, we removed the proportional charge-pump CP2 in Figure 5.1, and replaced the feed forward zero with a more conventional

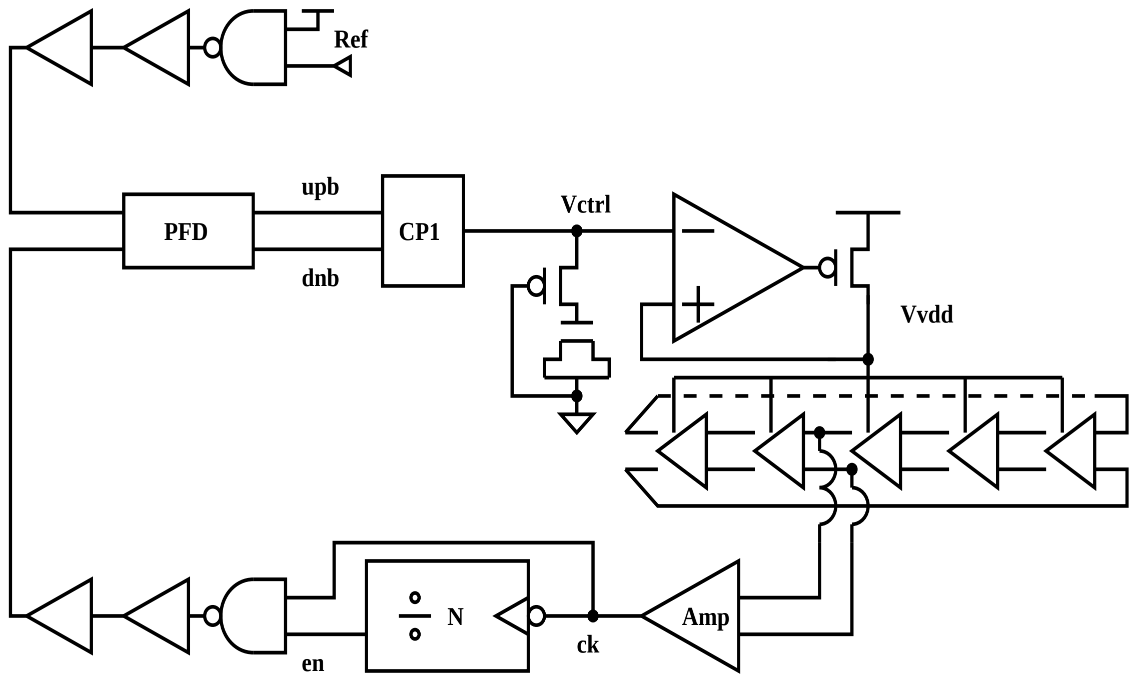


Figure 5.14: PLL Block Diagram

series RC loop filter [25]. Finally, decoupling the buffer from the stability zero allows the remaining charge-pump to be sized down for more power savings.

The RC filter is implemented with a PMOS transistor in series with the NMOS loop capacitor. To achieve a wide operating range, the PMOS resistor is biased with Vctrl to compensate for the PVT variations and to allow the loop dynamics to track across a wide operating frequency [38]. With Vds=0, the PMOS transistor is operating in the triode region and the resistance is the channel resistance, which is different from the "effective" resistance provided by the $1/g_m$ of the saturated transistor. As a result, the resistance measurement (using $MeasRout()$) was removed from the op-amp schematic (but kept in the comments view for record) and a new Measurement Comment using $MeasRch()$ was added to find the series resistance in the RC filter.

The block diagram of the final PLL is shown in Figure 5.14. For clarity the figure only shows one VCO buffer driving a level shifter in the feedback path. In the actual implementation, the other four buffers are also driving identical level shifters to produce the multiple clock phases. Originally, the PLL modification was limited to only replace the single-ended VCO with a differential design in order to generate the multi-phase clock needed by the new application. However, the goal of minimizing power consumption led us to modify other circuit blocks. Even though we changed the circuits, most of the Active Comments in the modified blocks were carried over from the original design. The only change is that we measure the channel resistance of the PMOS transistor in the RC loop filter instead of the output impedance of the amplifier to reflect the simplified filter topology.

5.3 Results

5.3.1 PLL Porting Results

Since the testchip was to be fabricated in a 0.25μm process, we need to port the design and resize the devices. All the analog circuits need to be re-optimized, and we focus our design objective on reducing jitter and static phase offset. To have small jitter, the PLL needs to have an over-damped response to minimize noise amplification, and a high loop bandwidth to minimize noise accumulation. In terms of the standard loop parameters, this means that the damping factor ζ should be greater than 0.7 and the loop bandwidth ω_n should be close to its theoretical maximum of 10% of the reference frequency.

Since each term in the loop dynamics equations is implemented by a different sub-block in the PLL, we used the Measurement Comments embedded in each block to find the circuit parameters in the sub-circuit, then evaluated the damping factor and loop bandwidth equations at the system level to help guide our optimization decisions. We can express the loop dynamic equations (Equations 5.2 and 5.3) with Measurement Comments:

```
# CALCULATE zeta=0.5*R*sqrt(Icp*Kvco*Ccp/N)
# CALCULATE wn=sqrt(Icp*Kvco*Ccp/N)/Ccp
```

Figure 5.15 shows the resulting loop dynamics tracking across the operating range with ζ always being above 0.7. The ω_n is pushed lower to about a quarter of the theoretical maximum of 10% of the reference frequency to account for the delay in the loop and to filter out some low frequency noise in the reference clock.

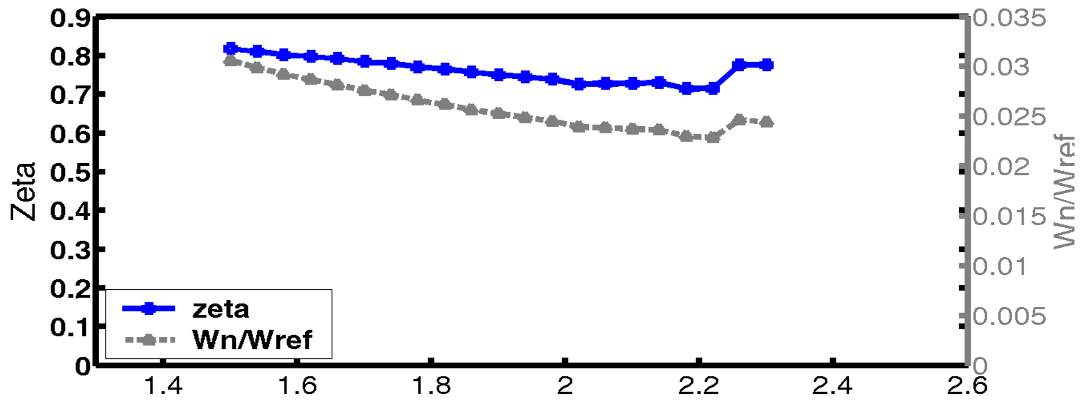


Figure 5.15: Tracking of PLL Loop Dynamics

The PLL was manually re-optimized based on the values of ζ and ω_n. This may be done more efficiently with the help of some of the modern analog synthesis tools [58][59][60].

5.3.2 Hidden Errors

In addition to automatically measuring and evaluating the critical circuit parameters, STAR is also used to monitors the circuit performance in the background. Often, designers pay close attention to the circuits they designed or modified and assume that the other parts are working correctly. Unfortunately, this assumption is not always valid. The new applications or process may introduce errors in the unmodified circuits, but these errors may be masked by top level signals. STAR uncovered a hidden error in the lower hierarchy that causes a static phase error between the internal clock and the external reference.

The top level simulation results in Figure 5.16 show the the rising edges of the feedback clock aligned with those of the reference clock at the PFD inputs. This implies that the loop is running at the target frequency with negligible phase offsets between the two clocks. From these waveforms, one would expect that the PLL is working properly, and all the sub-circuits are performing within the specification. However, this is not the case.

While a designer may not be able to check every signal in the simulation results, STAR automatically processes all the Assertion Comments embedded in the design. Even though the clocks are aligned at the PFD inputs, the tool reported a number of assertion failures at

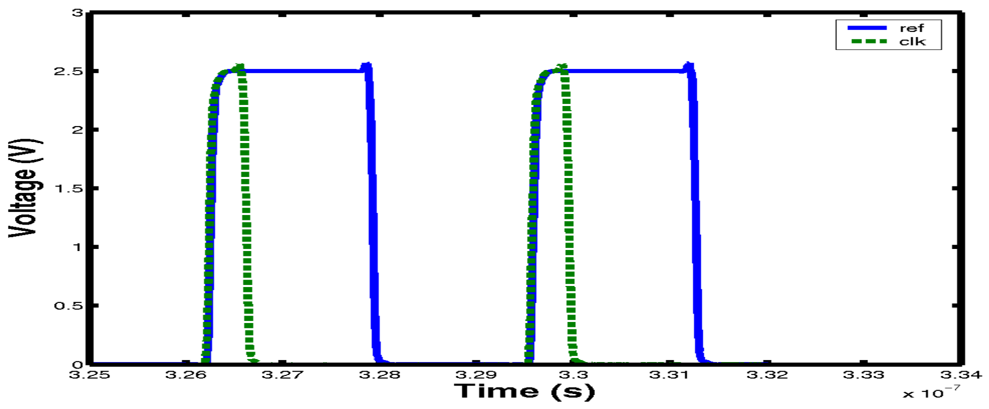


Figure 5.16: Reference and Feedback Clock Aligned at the Inputs of PFD

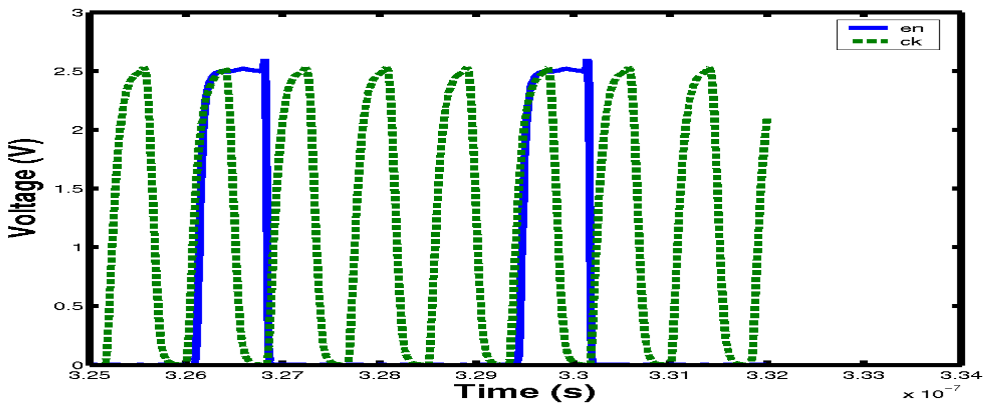


Figure 5.17: Enable Signal Fails Setup Time to Clock

the sub-block level. Specifically, the failed assertions are the timing constraints associated with the flip-flops in the divider and the qualifying NAND gate in Figure 5.14. The clk→Q of the flop is pushed out and causes the clock qualifying signal, en (Figure 5.14), to rise after the clock has risen, as shown in Figure 5.17. This results in a significant phase offset between the generated clock and the reference clock, even though the inputs to the PFD are aligned. Although the PLL is still functional and locks in simulation despite this error, the design margin is nevertheless reduced. The circuit is operating near the edge of failure. Uncovering this error made the ported design more robust.

Using STAR, we are able to successfully port and reuse this PLL design. Simulation

Table 5.1: PLL Performance Summary (Simulated at 800MHz)

Operating Range:	125-1200MHz
Active Area:	0.098mm^2
Phase Offset:	$<$ 20ps
Jitter:	$<$ 80ps (10% Vdd noise)
Power Dissipation:	72mW @ 2.5V
Technology:	National 0.25μm CMOS 1poly-5metal

results indicate that the operating range of the final PLL is 125-1200MHz. When simulating at 800MHz VCO frequency and N=4, the final design achieves a period jitter of 20ps and peak to peak jitter of $<$ 80ps with 10% step on the power supply. Table 5.1 summarizes the simulated PLL performance.

5.3.3 Prototype Tool Performance

We use the run time of the PLL porting experiment to study the performance of the prototype and evaluate the design decisions we made when implementing the the prototype. For our PLL design, the types of simulations and results analysis range from short simulations to measure specific circuit parameters to very long loop transient simulations that explore the loop dynamics. This large variation in the size of the data allows us to test the prototype under different circumstances.

Table 5.2 shows the total times the prototype took to simulate and extract all the parameters needed to evaluate the damping factor expression. The simulation time for most of the parameters was about 5 seconds each, except for measuring $Kvco$ which took about 17 minutes. The longer run time for the $Kvco$ measurement was because the simulation swept the control voltage to find the corresponding frequencies, and the simulation time for

Table 5.2: PLL evaluating zeta expression

Simulation Type:	Kvco	Icp	R	Ccp
Simulation Time:	17min	5sec	5sec	5sec
Simulation Results:	1.3K	19K	19K	0.15K
Reading Results:	1sec	1sec	1sec	1sec
Processing Measurements:	1 Sec			

Table 5.3: PLL transient simulation checking

Simulation Time:	$>$ 1 day
Simulation Results:	267M (248 variables, 280707 data points/var)
Reading Results:	49sec
Process Assertions:	50min 37sec (40 assertions total)

each run in the sweep was determined by the lowest frequency. Reading and processing the measurement results took on the order of seconds for each parameter. On an absolute time scale, the overhead of using the tool to process the results of these short simulations is very small.

In contrast to the short simulations for the measurements, the loop transient simulation took over a day to complete. The long simulation time for the PLL loop transient is determined by two time constants associated with the circuit. One time constant is the loop bandwidth which dictates the simulation time, and the other time constant is the frequency of the signals which dictates the time step. The simulation produced a 267MB transient waveform (.tr0) file which contains 280,707 data points for each of the 248 variables. The prototype took 49 seconds to load the transient file into the internal data structure and another 50min 37sec to process the 46 Assertion Comments embedded throughout the design hierarchy. Table 5.3 summarizes the performance.

From this experiment, we learn that simulation time dominates the design turn-around time for parameter measurements and loop transient simulations. We also found that in the verification phase, reading simulations results (I/O) is not the bottleneck. Instead, the system performance is limited by the processing time. Some of the performance degradation is the result of our choice of using Perl for its easy of implementation and flexibility. Another reason for the long processing time is that the current implementation processes one assertion at a time. Some of the performance can be gained back by processing the assertions in parallel since the Assertion Comments are independent from one another. We manually coded a three pass verification scheme where in the first pass we read the simulation results and load all the signal waveforms that are needed by the Assertions in the entire design. We pre-calculate all the signals and data that are used in the multi-pass Assertions in the second pass. In the final pass, we execute the Assertions in sequence using the pre-compute data. Using this manually coded system, the same set of Comments took

15 minutes to complete.

5.3.4 Design Reuse Experience

The porting and reuse of the circuit was performed by a graduate student who was not familiar with the framework and was designing a PLL for the first time. Using the embeded Measurement Comments to automatically generate the simulation runs and analysis routines, he was able to assess the loop performance on the first day of inheriting the database. The notes in the comments view for each schematic not only helped explain the Active Comments, but also served as a teaching tool to identify the key design criteria for each analog cell. The framework's capability of cascading measurements and checking dependencies between critical circuit parameters ensures that the simulations are executed in the correct order. The PLL was ported and optimized in about three man-weeks which included the time to learn both to use the framework and to design a PLL.

The feedback from applying the framework to help reuse the design focused on the learning curve of using Active Comments to create new simulation runs, which is different from the traditional paradigm of writing spice decks taught in the analog design classes today. While the user liked using spice decks generated from the existing Measurement Comments, he found that creating new Measurements required more understanding of both the framework and the pre-defined functions provided in the tool distribution. This initially demanded more time to create a new simulation run when compared to writing a new spice deck. However, once the user was familiar with the set of available functions and was comfortable with using Measurement Comments to control the simulations, he liked the ability to cascade measurements and reuse circuit parameters in the same framework without writing each line of the spice deck and creating his own scripts to process the results. Compared to the Measurements, the concept of analog assertions was very well received from the beginning. The user found that the Assertions greatly improved the robustness of the design because the critical performance constraints were automatically checked without requiring him to have extended knowledge of all the components in the reused design.

5.4 Summary

We used STAR to capture a PLL design and then used the PLL as a building block in an optical testchip fabricated in a different process technology. During the reuse process, we modified the VCO, voltage buffer, and loop filter of the PLL to meet the performance requirement set by the new application. We used the Measurement Comments in STAR to guide the manual re-optimization of the loop and the Assertion Comments to check and validate the system integration. STAR uncovered an hidden error in the lower level of the design which caused a significant static phase error between the internal clock and external reference.

The design turn-around time during the reuse process is dominated by the simulation run time. STAR adds about 5% to the total turn-around time for the long loop verification transient simulations. Most of the run time is spent in the Perl analysis routines not the I/O since we already optimized the reading of long simulation results with C. We can further decrease the STAR overhead by processing the Analysis routines in parallel.

Chapter 6

Conclusion

Reuse of analog circuits requires the archival representation to be more than just the circuit topology and sizes. The archive must include some design knowledge about how the circuit was designed and what constraints were assumed when the device was simulated. We need to check these constraints not only when designing the circuit in isolation, but also for each instance of the circuit where it is used.

We created a system that addresses many of these issues using Active Comments. This system is flexible enough to allow the designers to easily extend it to capture the constraints of new designs. We were able to capture a PLL design as a reusable analog module using a combination of ten base functions.

To enable designers to create their own functions to capture the constraints and assumptions, two programming language interfaces are provided. One is a scripting language which enables rapid prototyping, and the other is a compiled language which provides higher performance at the cost of higher programming effort. By using these two interfaces, the designer can trade-off between development time and processing speed. With a mixture of Perl functions and two compiled C routines, our prototype system incurred a verification time overhead of less than 5%.

The key additional requirement for making reusable blocks is that Active Comments metrics should be coded in a process independent way, preferably by using the inherent properties of the underlying circuit. While currently this approach requires some additional effort, we hope if systems like this become more common, designers will become more comfortable with it, and start coding this way naturally.

6.1 Future Work

The work presented in this dissertation can be extended in several ways. While designers can use the current version of STAR to specify the circuit parameters and performance constraints, they still need to manually optimize the design. A natural extension to STAR is to integrate the recently available multi-variable optimization engines into our framework to form a complete system for designing and reusing analog modules.

Work on designing and reusing analog circuits has primarily focused on constraint specification and circuit optimization. One area that has been ignored by the CAD community is the debugging of circuits performing outside the specification. Being able to find the causes of the problem is key to design reusability since the persons instantiating these analog modules are often not the original designer and therefore not familiar with the design. Performance degradations are caught in the form of assertion failures. Often each assertion failure is the result of other failed assertions. By creating a dependency graph between the assertions, we can trace the graph to find the possible causes when a violation is encountered.

In addition to improving the debugging process for the back-end verification, we can also extend our design capture front-end to embed more design information into the circuit representation. In the examples used throughout this dissertation, we alternate between production and test bench schematics to measure circuit parameters. However these two sets of schematics are two representations of the design and may become out of sync over time. Since the test benches are modeled after the intended operating environment of the module, one should be able to derive them from the top level production schematic. It may be possible to combine the test bench and the production schematics into one unified design representation. User can include additional devices and group the different components on the schematic to indicate the different test benches. From this unified representation, the design system can generate the user specified view on the fly. This way, any changes to the production design can be automatically reflected in the test benches.

We anticipate that in the future, analog circuits will continue to be designed by experienced designers but with more help from CAD tools to ensure robustness and reusability. The framework described in this dissertation can be used to capture design goals and check performance of both manually designed and automatically generated circuits. Using this type of design tools should help expedite the adaption of analog synthesis tools as they become commercially available.

Appendix A

STAR User Manual

The STAR design framework provides a set of pre-defined functions that enable designers to capture performance goals and operating constraints of analog and mixed-signal circuits. The functions used to capture an example PLL design are included as part of the tool distribution. The details of these functions are presented in this appendix along with a brief description of the datatypes and global parameters supported in the prototype.

This appendix is organized as a manual to using the STAR system. The syntax of the Active Comment enables the user to define variables which are type casted at run time. Section A.1 describes the available datatypes. Section A.2 describes the global parameters provided by the STAR framework to support technology independent specifications and constraints and to facilitate data passing between functions. All the functions created to capture the example PLL design are listed alphabetically in Section A.3 while the two primitives used to improve the performance of the tool can be found in Section A.4.

A.1 Datatypes

Variables used in the Active Comments can take on one of three datatypes: $scalar$, $vector$, and $wave$. A $scalar$ is a single data value which can be a string, an integer, or a float, while a $vector$ is an one dimensional array of $scalars$. The $wave$ datatype is an array of ordered pairs such as ($time$,$value$). The ordered pair identifies a point in a two dimensional plot, and together all the pairs in the array form a curve or signal waveform from a transient

Table A.1: Supported Datatypes in STAR

Datatype	Description
$scalar$	integer, float, or string
$vector$	array of scalar
$wave$	array of ($time$,$value$) ordered pair

analysis. The $scalar$ and $vector$ datatypes are used for circuit parameters defined by the Measurement Comments, and $wave$ is the datatype used for transient signal waveforms in Assertions. These three datatypes are summarized in Table A.1.

These three types of data are represented in STAR using native Perl datatypes. The $scalar$ and $vector$ types map to the Perl scalar and array structures. While there is no native $wave$-like data structure in Perl, a $wave$ data is represented by an array of strings, where the value of the string is "$time$,val". Concatenating the ordered pair into a string allows the $wave$ to be represented in a simple array instead of using more complicated structures such as an array of arrays or an array of hashes. This simplifies the designer's coding effort when creating new functions in the library.

A.2 Global Parameters

In STAR, the Active Comments and library functions use technology independent parameters to improve the portability of the design. These parameters are stored in the GlbParam.prm file, and the values of the parameters are updated for each technology process and design project. Figure A.1 displays the contents of the GlbParam.prm file. The parameters in this file include $hspLib$, $vsup$, $FO4Delay$, $FO4RiseTime$, $defaultCorner$, and $vectorSize$. The parameter names are used as keys to a hash table, and their values are the corresponding entries. Using a hash table separates the name space of these global parameter from the variables used in the engine and simplifies the process of loading these parameter into memory and searching their values during run time.

The parameter $hspLib$ holds the location of the device model file, $vsup$ is the supply voltage, $FO4Delay$ and $FO4RiseTime$ are the process dependent gate delay metrics, $defaultCorner$ sets the simulation corner when it is omitted in a $\#\ DEFINE$ statement

```
$glbList->{"hspLib"} = "\$ENV{PROJ}/lib/fets.lib";
$glbList->{"vsup"} = "3.3";
$glbList->{"FO4Delay"} = 180e-12;
$glbList->{"FO4RiseTime"} = 180e-12;
$glbList->{"defaultCorner"} = "tttt";
$glbList->{"vectorSize"} = 20;
```

Figure A.1: Contents of GlbParam.prm

, and $vectorSize$ sets the size of the vectors for the variables defined by the Measurement Comments. Defining the size of the vector as a global variable guarantees that all the vectors defined by the Measurements are the same size and can therefore be operated on together. The pre-processing functions in the MeasGenerate.pm use this parameter when creating sweeps and ramps.

In STAR, the parser layer provides a global hash table to store the results computed by the analysis functions. Each function passes the reference (or key) to the entry in the hash table as its return value such that the passing of parameters between functions is done using these references. The global hash table is called %ResultTable, where the % sign denotes a built-in hash table datatype in Perl syntax. In addition to storing the analysis results, the hash table also contains additional information about the comments used by the parser. Consequently, each entry in the table is referenced by a combination of two keys: the actual text string of the Active Comment that instantiated the function ($cmtName) and a reference name generated by the function ($refName). The first key is used to identify the comment, while the second key uniquely identifies the data. For example:

$ResultTable{$cmtName}{$refName}[$i]

accesses the ith element in the array entry referenced by $cmtName and $refName. The values of $cmtName and $refName are passed into the parser, and the details are described in the next section.

Table A.2: Additional Arguments Passed into Functions

Generate Functions		Analysis Functions	
$cktName	sub-circuit name	$cktName	sub-circuit name
$topCktName	top-most circuit	$topCktName	top-most circuit
$$masCnt_ptr	pointer to counter	$$masCnt_ptr	pointer to counter
		$cmtName	Active Comment

A.3 Functions

In order for the functions in the library layer to access the hash table and to create unique reference names, the parser passes a few extra arguments into the function at runtime. The argument list is different for the generation functions and the analysis functions. For the generation functions (especially for the Assertions), we need to know which sub-circuit the Assertion was embedded in order to find all the instances of that sub-circuit. The search routine that traverses the netlist also requires the name of the top level circuit. Therefore, these two values, $cktName and $topCktName, are the first two parameters in the list of extra arguments. The last parameter in the list, $$measCnt_ptr, is a pointer to a counter variable that can be used to generate an unique name to reference the measurement, probe, and voltage or current sources produced by these functions. In addition to this set of extra arguments, the parser passes one more parameter into the analysis function. This parameter, $cmtName, is the Active Comment that instantiated the function. When a cell is instantiated more than once, the Assertion embedded in that cell will be expanded to contain the full path to the cell. This way, $cmtName is guaranteed to be unique. Table A.2 summarizes the different parameters which are passed into the functions.

When creating new functions, the programmer must account for these extra arguments in the code. However, from the tool user's perspective, these parameters are hidden and are not part of the function's interface. For clarity, the function description below only includes arguments a designer would enter in the comments editor when creating Active Comments.

Abs

Abs($signal$) => $abs\_\$measCnt$

Description

This function finds the absolute value of the $signal$. The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ is the time step of the simulation and $value$ is the delay.

Arguments

$signal$: name of signal

Value Returned

$abs\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

Abs(I(vsrc)) => “abs_i_0”

Effect: @{$ResultTable{$cmtName}{“abs_i_0”}} contains the absolute value of the current entering the voltage source vsrc

FindDelay

FindDelay($signal1$,$thres1$,$dir1$,$signal2$,$thres2$,$dir2$,$numOfTran$) => $dly\_\$measCnt$

Description

This function finds the delay from the time instant when $signal1$ reaches $thres1$ (trigger event) to the time instant when $signal2$ reaches $thres2$ for $numOfTran$ times (target event). The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ is the time of the trigger and $value$ is the delay. The delay is measured once per trigger.

Arguments

$signal1$: name of trigger signal
$thres1$: threshold level to initiate the trigger
$dir1$: direction of the transition (rise or fall)
$signal2$: name of the target signal
$thres2$: threshold level to initiate the target
$dir2$: direction of the transition (rise or fall)
$numOfTran$: the number of transitions the target signal waits after the trigger event

Value Returned

$dly\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

FindDelay(clk,0.2*vdd,rise,0.8*vdd,rise,1) => “dly_1”

Effect: @{$ResultTable{$cmtName}{“dly_1”}} contains the rise time of the node clk

DelayRR,DelayRF,DelayFF,DelayFR

DelayRR($signal1$,$signal2$) $=>$ $dlyrr\_\$measCnt$
DelayRF($signal1$,$signal2$) $=>$ $dlyrf\_\$measCnt$
DelayFF($signal1$,$signal2$) $=>$ $dlyff\_\$measCnt$
DelayFR($signal1$,$signal2$) $=>$ $dlyfr\_\$measCnt$

Description

This function finds the delay from the $signal1$ rising/falling to 50% of vdd to $signal2$ rising/falling to 50% of vdd. The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ is the time of the trigger and $value$ is the delay. The delay is measured once per trigger.

Arguments

$signal1$: name of trigger signal
$signal2$: name of the target signal

Value Returned

$dlyrr\_\$measCnt$: name of the key to entry in the global hash table where the data is stored
$dlyrf\_\$measCnt$
$dlyff\_\$measCnt$
$dlyfr\_\$measCnt$

Example

DelayRR(clk,clk) $=>$ “dlyrr_2”
Effect: @{$ResultTable{$cmtName}{“dlyrr_2”}} contains the period of the node clk

DelayRR2,DelayRF2,DelayFF2,DelayFR2

DelayRR2($signal1$,$signal2$) => $dlyrr2\_\$measCnt$
DelayRF2($signal1$,$signal2$) => $dlyrf2\_\$measCnt$
DelayFF2($signal1$,$signal2$) => $dlyff2\_\$measCnt$
DelayFR2($signal1$,$signal2$) => $dlyfr2\_\$measCnt$

Description

This function finds the delay from the $signal1$ rising/falling to 50% of vdd to the second time $signal2$ rising/falling to 50% of vdd. The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ is the time of the trigger and $value$ is the delay.

Arguments

$signal1$: name of trigger signal
$signal2$: name of the target signal

Value Returned

$dlyrr2\_\$measCnt$: name of the key to entry in the global hash table where the data is stored
$dlyrf2\_\$measCnt$
$dlyff2\_\$measCnt$
$dlyfr2\_\$measCnt$

Example

DelayRR2(en,clk) => “dlyrr2_3”

Effect: @{$ResultTable{$cmtName}{“dlyrr2_3”}} contains the delay from en rising to the second time clk rises

FindNode

FindNode($pattern$,$topCktName$,$cktName$) => *@MatchedNode*

Description

This is a helper function used by other functions and not directly called by the user. The function traverses through the netlist to find all the nodes within the subckt, $cktName$, that match the regular expression $pattern$.

Arguments

$pattern$: regular expression for the node to be found
$topCktName$: name of the top-most circuit in the netlist
$cktName$: name of the sub-circuit where the nodes are to be found

Value Returned

@MatchedNode: array of full path to the nodes that match $pattern$

Example

n0 is in subckt oamp and the top-most circuit is called PLL
FindNode(n0,PLL,oamp) => (xoamp.n0)

FindTx

FindTx(tx_prop,$topCktName$,$cktName$) => *@MatchedTx*

Description

This is a helper function used by other functions and not directly called by the user. The function traverses through the netlist to find all the transistors within the subckt, $cktName$, that contain the property, tr_prop.

Arguments

tr_prop: name of transistor property added to the transistor of interest
$topCktName$: name of the top-most circuit in the netlist
$cktName$: name of the sub-circuit where the transistors are to be found

Value Returned

@MatchedTx: array of full paths to the transistors that have the tr_prop

Example

m0 and m1 in subckt oamp are tagged with cs property
FindTx(cs) => (xoamp.m0, xoamp.m1)

Freq

Freq($node$) => $freq\_\$measCnt$

Description

This function adds spice .meas commands to calculate the frequency of the signal at node $node$.

Arguments

$node$: node in the schematic

Value Returned

$freq\_\$measCnt$: reference name of the measurement command added to the spice deck, used to access simulation result

Example

Freq(clk) => “freq_4”

Effect: two .meas statements are added to the spice deck to find the frequency of clk

HiRange

HiRange($xVar$,$yVar$) $=>$ hi_x

Description

This function finds the upper inflection of a curve. An inflection point is defined as a more than 20% change in the slope of the curve.

Arguments

$xVar$: name of measured parameter that forms the independent (x) axis

$yVar$: name of the measured parameter that forms the dependent (y) axis

Value Returned

hi_x: value of the xVar at the upper inflection point

Example

vctrl is the swept parameter, freq_ck is measured frequency, upper inflection point occurs at vctrl=2.7V

HiRange(vctrl,freq_ck) $=>$ 2.7

I

I($vsrc$) => $i\_\$measCnt$

Description

This function reads the specified current waveform from the simulation transient results into the memory of the executing prototype. The data is stored in the global hash table. The datatype of the waveform is an array of ($time$,$value$) ordered pairs.

Arguments

$vsrc$: name of the voltage source that the current being measured is flowing into

Value Returned

$i\_\$measCnt$: name of the key to enter into in the global hash table where the data is stored

Example

I(vsrc0) => “i_5”

Effect: @{$ResultTable{$cmtName}{“i_5”}} contains the current waveform

InitV

InitV($node$,$value$) => $node$

Description

This functions initializes node to a voltage value.

Arguments

$node$: name of node in the schematic

$value$: voltage to be initialized at the node

Value Returned

$node$: name of the node to be initialized

Example

InitV(bias,100m) => “bias”

Effect: the node $bias$ is set to 100mV DC relative to ground

LoRange

LoRange($xVar$,$yVar$) $=>$ lo_x

Description

This function finds the lower inflection of a curve.

Arguments

$xVar$: name of the measured parameter that forms the independent (x) axis

$yVar$: name of the measured parameter that forms the dependent (y) axis

Value Returned

lo_x: value of the xVar at the lower inflection point

Example

vctrl is the swept parameter, freq_ck is the measured frequency, lower inflection point occurs at vctrl=1.2V

LoRange(vctrl,freq_ck) $=>$ 1.2

HiRange

Max(var) $=>$ max_var

Description

This function finds the maximum value in a vector.

Arguments

var: name of measured parameter that forms the vector

Value Returned

max_var: maximum value in the given vector

Example

per_ck is a measured parameter with values [1 2 3 4 3 2 1]

Max(per_ck) $=>$ 4

MeasCap

MeasCap($node$) => $mc\_\$measCnt$

Description

This function creates a complete spice run routine to measure the effective capacitance at $node$.

Arguments

$node$: name of the node in the schematic

Value Returned

$mc\_\$measCnt$: name of the spice measurement created by the function, used to find the measured capacitance in the simulation results

Example

node0 has 20pF

MeasCap(node0) => “mc_6”

MeasRch

MeasRch(tr_prop) => \@*probe*

Description

This function adds spice .probe commands to find the channel resistance of transistors marked with tr_prop in the schematic.

Arguments

tr_prop: name of transistor property added to the transistors of interest

Value Returned

\@*probe*: pointer to an array of probe names added to the spice deck

Example

m0 and m1 are tagged with cs property

MeasRch(cs) => pointer to (prb_m0, prb_m1)

Effect: two probes, prb_m0 and prb_m1, are added to the spice deck

MeasRout

MeasRout(tr_prop) => $\backslash$@*probe*

Description

This function adds spice .probe commands to find the output impedance of the transistors that are marked with tr_prop in the schematic.

Arguments

tr_prop: name of transistor property added to the transistor of interest

Value Returned

$\backslash$@*probe*: pointer to an array of probe names added to the spice deck

Example

m0 and m1 are tagged with cs property

MeasRout(cs) => pointer to (prb_m0, prb_m1)

Effect: two probes, prb_m0 and prb_m1, are added to the spice deck

HiRange

Min(var) => min_var

Description

This function finds the minimum value in a vector.

Arguments

var: name of measured parameter that forms the vector

Value Returned

min_var: minimum value in the given vector

Example

per_ck is a measured parameter with values [1 2 3 4 3 2 1]

Min(per_ck) => 1

Mt0Reader

Mt0Reader($mt0File$,var) => \@*measResult*

Description

This function reads the mt0 file which contains the results of all .meas commands in the spice simulation run.

Arguments

$mt0File$: name of mt0 file

var: name of the measured parameter to be read in

Value Returned

\@*measResult*: pointer to array of measured values in the mt0 file

Example

vctrl is the swept parameter and freq_ck is the measured frequency

mt0Reader(vcoV3.mt0,freq_ck) => pointer to (freq, ...)

Noise

Noise($signal$) $=>$ $noise\_\$measCnt$

Description

This function finds the noise on the $signal$ waveform (usually a DC signal). The result is stored in the global hash table. This function leverages the $RunningAvg$ function and the averaging window is set to 30 FO4 delays. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ is the time step of the simulation and $value$ is the delay.

Arguments

$signal$: name of signal

Value Returned

$noise\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

Noise(i(vsrc)) $=>$ "noise_7"

Effect: @{$ResultTable{$cmtName}{"noise_i_6_7"}} contains the instantaneous noise on the current through the voltage source vsrc

OutputResistance

OutputResistance(tr_prop) => \@*probe*

Description

This function adds spice .probe commands to find the output resistance of the transistors that are marked with tr_prop in the schematic. This function is used in Assertions where MeasRout is used in Measurements.

Arguments

tr_prop: name of transistor property added to the transistor of interest

Value Returned

\@*probe*: pointer to an array of probe names added to the spice deck

Example

m0 and m1 are tagged with cs property

OutputResistance(cs) => pointer to (prb_m0, prb_m1)

Effect: two probes, prb_m0 and prb_m1, are added to the spice deck

Period

Period($node$) => $per\_\$measCnt$

Description

This function adds spice .meas commands to find the period of the signal at node $node$.

Arguments

$node$: name of signal waveform

Value Returned

$per\_\$measCnt$: reference name of the measurement command added to the spice deck, used to access simulation result

Example

Period(clk) => “per_8”

Effect: one .meas is added to the spice deck to find the period of the signal at node clk

PulseWidth

PulseWidth($signal$) $=>$ $pw\_\$measCnt$

Description

This function finds the pulse width of the $signal$. The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ is the time of the trigger and $value$ is the delay.

Arguments

$signal$: name of signal waveform

Value Returned

$pw\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

PulseWidth(clk) $=>$ “pw_9”

Effect: @{$ResultTable{$cmtName}{“pw_9”}} contains the pulse width of clk

PulsePropagation

PulsePropagation($signal\_in$, $signal\_out$) $=>$ $pp\_\$measCnt$

Description

This function checks the propagation of the signal from $signal\_in$ to $signal\_out$ by finding the delay from $signal\_in$ to $signal\_out$. The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ is the time of the trigger and $value$ is the delay. If $signal\_out$ has not transitioned by the second time $signal\_in$ has switched, then the $value$ is set to “failure”.

Arguments

$signal\_in$: name of input signal
$signal\_out$: name of output signal

Value Returned

$pp\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

PulsePropagation(upi,upo) => pp_10
Effect: @{$ResultTable{$cmtName}{“pp_10”}} contains the delay from upi to upo

RampI

RampI($node$,$value1$,$value2$) $=>$ $i\_\$measCnt$

Description

This function adds a current source into $node$ and ramps the current from $value1$ to $value2$. The ramp rate is set to the risetime of a FO4 inverter. This function is used in pseudo DC simulation.

Arguments

$node$: name of node in the schematic
$value1$: initial current value
$value2$: final current value

Value Returned

$i\_\$measCnt$: name of the current source added to the spice deck

Example

RampI(bias,100m,200m) $=>$ i_11
Effect: a current ramping linearly from 100mA to 200mA is injected into the node $bias$

RampV

RampI($node$,$value1$,$value2$) $=>$ $v\_\$measCnt$

Description

This function adds a grounded voltage source to $node$ and ramps the voltage from $value1$ to $value2$. The ramp rate is set to the risetime of a FO4 inverter. This function is used in pseudo DC simulation.

Arguments

$node$: name of node in the schematic

$value1$: initial voltage value

$value2$: final voltage value

Value Returned

$v\_\$measCnt$: name of the grounded voltage source added to the spice deck

Example

RampV(bias,100m,200m) $=>$ “v_12”

Effect: the voltage at node $bias$ ramps linearly from 100mV to 200mV

RunningAvg

RunningAvg($signal$,$size$) => $ravg\_\$measCnt$

Description

This function finds the running average of a waveform by taking the average within a timing window and sliding that window across the waveform. The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pairs, where $time$ is the time of the simulation step and $value$ is the running average.

Arguments

$signal$: name of signal

$size$: size of window in absolute time over which the values are averaged

Value Returned

$ravg\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

RunningAvg(bias,30*FO4) => “ravg_13”

Effect: @{$ResultTable{$cmtName}{“ravg_13”}} contains the running average of the signal bias

SatMargin

SatMargin(tr_prop) => \\@*probe*

Description

This function adds spice .probe commands to find the saturation margin, (Vds-Vdsat), of transistors marked with tr_prop in the schematic.

Arguments

tr_prop: name of transistor property added to the transistor of interest

Value Returned

\\@*probe*: pointer to an array of probe names added to the spice deck

Example

m0 and m1 are tagged with cs property

SatMargin(cs) => pointer to (prb_m0, prb_m1)

Effect: two probes, prb_m0 and prb_m1, are added to the spice deck

SetE

SetE($node0$, $node1$, $node2$, $node3$ $value$) $=>$ $e\_\$measCnt$

Description

This functions adds a voltage-controlled voltage source (VCVS) between $node1$ and $node2$. The voltage between these two nodes is the voltage between $node3$ and $node4$ multiplied by $value$

Arguments

$node1$: name of higher controlled-voltage node in the schematic
$node2$: name of lower controlled-voltage node in the schematic
$node3$: name of higher controlling-voltage node in the schematic
$node4$: name of lower controlling-voltage node in the schematic
$value$: multiplication factor of the controlling voltage

Value Returned

$e\_\$measCnt$: name of the VCVS added to the spice deck

Example

SetE(bias,gnd,Vout,gnd,1) $=>$ “v_14”

Effect: the node $bias$ is set to the same voltage as $Vout$ and a VCVS is added to the spice deck

SetI

SetI($node$,$value$) $=>$ $i\_\$measCnt$

Description

This functions injects a DC current into the specified node.

Arguments

$node$: name of node in the schematic

$value$: current to be injected into the node

Value Returned

$i\_\$measCnt$: name of the current source added to the spice deck

Example

SetI(bias,100m) $=>$ “i_15”

Effect: 100mA is injected into the node $bias$

SetV

SetV($node$,$value$) $=>$ $v\_\$measCnt$

Description

This functions sets the specified node to a constant voltage with a voltage source.

Arguments

$node$: name of node in the schematic

$value$: voltage to be set at the node

Value Returned

$v\_\$measCnt$: name of the grounded voltage source added to the spice deck

Example

SetV(bias,100m) $=>$ “v_16”

Effect: the node $bias$ is set to 100mV DC relative to ground

Slope

Slope($xVar$,$yVar$) $=>$ $\backslash$@*slope*

Description

This function finds the slope of each data point of a curve by calculating ΔY/ΔX between adjacent data points.

Arguments

$xVar$: name of measured parameter that forms the independent (x) axis

$yVar$: name of the measured parameter that forms the dependent (y) axis

Value Returned

$\backslash$@*slope*: pointer to an array of scalars where the scalar value is the slope of the curve

Example

vctrl is the swept parameter and freq_ck is the measured frequency

vctrl=(x_1, x_2, x_3,....) and freq_ck=(y_1, y_2, y_3,....)

Slope(vctrl, freq_clk) $=>$ pointer to ($\frac{y_2-y_1}{x_2-x_1}$, $\frac{y_3-y_2}{x_3-x_2}$, ...)

StepV

StepV($node$,$valueOld$,$valueNew$,$time$) $=>$ $v\_\$measCnt$

Description

This functions steps the node voltage from $valueOld$ to $valueNew$ at time $time$. The transition time is set to 1/10 of the FO4 risetime. This function is usually applied to the supply node.

Arguments

$node$: name of node in the schematic
$valueOld$: voltage to be set at the node until the time of the step
$valueNew$: voltage at the node after the step
$time$: the time at which to apply the step

Value Returned

$v\_\$measCnt$: name of the grounded voltage source added to the spice deck

Example

StepV(Vdd,vddval,0.9*vddval,1ns) $=>$ “v_16”

Effect: the node Vdd is set to vddval from time 0 until 1ns, at which time it becomes 0.9*vddval

SweepClk

SweepClk($node1$,$node2$,$period$) $=>$ $swpc\_\$measCnt$

Description

This function generates two 50% duty-cycle clocks, one at $node1$ and another at $node2$. The period of the clocks is $period$ and the delay/phase between these two clocks is swept from -$2pi$ to +$2pi$. This is a specialized function to characterize the phase-frequency-detector.

Arguments

$node1$: name of the first clock node
$node2$: name of the second clock node
$period$: period of the generated clocks
($vectorSize$): default number of points in sweep, set in GlbParam

Value Returned

$swpc\_\$measCnt$: name of the sweep parameter for the phase of the second clock

Example

SweepClk(ref,ck,10*FO4) $=>$ "swpc_17"

Effect: a clock signal with the period of $10 * FO4$ is added to ref and ck, and the delay of ck to ref is swept from -$10 * FO4$ to +$10 * FO4$.

SweepI

SweepI($node$,$value1$,$value2$) => $swpi\_\$measCnt$

Description

This function adds a current source to $node$ and sweeps the current from $value1$ to $value2$.

Arguments

$node$: name of node in the schematic

$value1$: initial current value

$value2$: final current value

($vectorSize$): default number of points in sweep, set in GlbParam

Value Returned

$swpi\_\$measCnt$: name of the sweep parameter added to the spice deck

Example

SweepI(bias,100m,200m) => "swpi_18"

Effect: a DC current sweeping from 100mA to 200mA is injected into the node $bias$

SweepV

SweepV($node$,$value1$,$value2$) $=>$ $swpv\_\$measCnt$

Description

This function adds a grounded DC voltage source to $node$ and sweeps the voltage from $value1$ to $value2$.

Arguments

$node$: name of node in the schematic

$value1$: initial voltage value

$value2$: final voltage value

($vectorSize$): default number of points in sweep, set in GlbParam

Value Returned

$swpv\_\$measCnt$: name of the sweep parameter added to the spice deck

Example

SweepV(bias,100m,200m) $=>$ "swpv_19"

Effect: voltage at node $bias$ is swept from 100mV to 200mV

Swing

Swing($signal$) => $sw\_\$measCnt$

Description

This function finds the rise time of the $signal$ from 2% to 98% of a reference. The reference is set to the source voltage of the PMOS transistor connected to the signal node. The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ is the time of the trigger and $value$ is the delay. If $signal$ does not reach 98%, then the $value$ is set to "failure".

Arguments

$signal$: name of signal waveform

Value Returned

$sw\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

Swing(clk) => "sw_20"

Effect: @{$ResultTable{$cmtName}{"sw_20"}} contains the rise time of clk

Time

Time($signal$) => $time\_\$measCnt$

Description

This function finds the the time value at each simulation step to give us a timing reference. The result is stored in the global hash table. The datatype of the result is an array of ($time$,$value$) ordered pair, where $time$ and $value$ are both the time value at each simulation step.

Arguments

$signal$: name of signal waveform

Value Returned

$time\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

Time(clk) => "time_21"

Effect: @{$ResultTable{$cmtName}{"time_21"}} contains the time value at each simulation step

V

V($signal$) => $v\_\$measCnt$

Description

This function reads the specified voltage waveform from the simulation transient results into memory. The data is stored in the global hash table. The datatype of the waveform is an array of ($time$,$value$) ordered pair.

Arguments

$signal$: name of the signal to be read in

Value Returned

$v\_\$measCnt$: name of the key to entry in the global hash table where the data is stored

Example

V(clk) => “v_22”

Effect: @{$ResultTable{$cmtName}{“v_22”}} contains the voltage waveform of the node clk

Write

Write($parameter$) $=>$ $parameter$

Description

This function writes the value of the parameter defined by the $\# DEFINE$ and $\# CALCULATE$ statements to $<$design$>$.prm where $<$design$>$ is the basename of the design being checked by STAR.

Arguments

$parameter$: name of the parameter

Value Returned

$paramter$: name of the parameter, not expected to be used

Example

vLo is defined by $\# DEFINE$ in vcoV3.sue

Write(vLo) $=>$ “vLo”

Effect: The value of vLo is written into vcoV3.prm

A.4 Primitives

FindTime

FindTime($sig1$,$thres1$,$dir1$,$sig2$,$thres2$,$dir2$,$numTran$) => \@*delayWaveform*

Description

The primitive finds the delay from the time instant when $sig1$ reaches $thres1$ (trigger event) to the time instant when $sig2$ reaches $thres2$ for the $numTran$ times (target event). The datatype the result is an array of ($time$,$value$) ordered pair, where $time$ is the time of the trigger and $value$ is the delay.

Arguments

$sig1$: name of trigger signal
$thres1$: threshold level to initiate the trigger
$dir1$: direction of the transition (rise or fall)
$sig2$: name of the target signal
$thres1$: threshold level to initiate the target
$dir2$: direction of the transition (rise or fall)
$numOfTran$: the number of transitions the target signal waits after the trigger event

Value Returned

\@*delayWaveform*: pointer to the array of ($time$,$value$) ordered pair, where $time$ is the time of the trigger and $value$ is the delay.

Example

FindDelay(clk,0.2*vdd,rise,0.8*vdd,rise,1) => pointer to (t1,d1 t2,d2 ...)

FindWave

FindWave($signal$) => \\@*signalWaveform*

Description

This function reads the waveform of the specified signal frmo the simulation transient results into memory. The signal can be a node voltage or current through a voltage source. The waveform is an array of ($time$,$value$) ordered pair.

Arguments

$signal$: name of signal to read; signal can be a node voltage or current through a voltage source

Value Returned

\\@*signalWaveform*: pointer to the array of ($time$,$value$) ordered pair, where $time$ is the time at the simulation step and $value$ is the value of the signal

Example

FindWave(clk) => pointer to (t1,v1 t2,v2 ...)

Bibliography

[1] Ron Ho. *Private communication on proprietary circuit checkers at Intel*. 2003.

[2] Kathirgamar Aingaran. *Noise Tool: Noise Analyzer for High-Performance CMOS Circuits*. proprietary tool at Sun Microsystem, 1997.

[3] R. Arunachalam, K. Rajagopal, and L. T. Pileggi. TACO: Timing Analysis with Coupling. In *Proc. Design Automation Conference*, pages 266–269. IEEE/ACM, June 2000.

[4] Patrik Larsson and Christer Svensson. Noise in Digital Dynamic CMOS Circuits. *IEEE Journal of Solid-State Circuits*, 29(6):655–662, June 1994.

[5] Ki-Wook Kim and Sung-Mo Kang. Crosstalk Noise Minimization in Domino Logic Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1091–1100, September 2001.

[6] Laurence W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, University of California at Berkeley, May 1975.

[7] *CosmosSE User's Guide*. Synopsys, Inc., Mountain View, CA, 2002.

[8] *Advance Design System User's Guide*. Agilent Technology, 2002.

[9] *Analog Design Environment*. Cadence, 2002.

[10] M Degrauwe. IDAC: An interactive Design Tool For Analog CMOS Circuits. *IEEE Journal of Solid-State Circuits*, 22:1106–1115, December 1987.

[11] R Harjani, R Rutenbar, and L.R. Carley. OASYS: A Framework for Analog Circuit Synthesis. *IEEE Trans. Computer-Aided Design*, 8:1247–1265, December 1989.

[12] H Koh, C Sequin, and Paul Gray. OPASYN: A Comiler for CMOS Operational Amplifier. *IEEE Trans. Computer-Aided Design*, 9:124–125, Febuary 1990.

[13] J Harvey, M Elmasry, and B Leung. STAIC: An Interactive Framework for Synthesing CMOS and BiCMOS Analog Circuits. *IEEE Trans. Computer-Aided Design*, 11:1402–1416, November 1992.

[14] G Gielen and W Sansen. *Symbolic Analysis for Automated Design of Analog Integrated Circuits*. Kluwer, Norwell, MA, 1991.

[15] Walter Daems, Georges Gielen, and Willy Sansen. Simulation-Based Automatic Generation of Signomial and Posynomial Performance Models for Analog Integrated Circuit Sizing. In *Proc. International Conference on Computer Aided Design*, pages 70–74. IEEE/ACM, November 2001.

[16] Maira del Mar Hershenson, Stephen P. Boyd, and Thomas H. Lee. GPCAD: A Tool for CMOS Op-Amp Synthesis. In *Proc. International Conference on Computer Aided Design*, pages 296–303. IEEE/ACM, November 1998.

[17] Maira del Mar Hershenson, Stephen P. Boyd, and Thomas H. Lee. Optimal Design of a CMOS Op-Amp via Geometric Programming. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(1):1–21, January 2002.

[18] Pradip Mandal and V. Visvanathan. CMOS Op-Amp Sizing Using a Geometric Programming Formulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(1):22–38, January 2001.

[19] F. Medeiro, F. V. Fernandez-Fernandez, R. Dominguez-Castro, and A. Rodriguez-Vazquez. A Statistical Optimization-Based Approach for Automated Sizing of Analog Cells. In *Proc. Design Automation Conference*, pages 594–597. IEEE/ACM, June 1994.

[20] E. Ochotta, Rob A. Rutenbar, and L. Richard Carley. Synthesis of High-Performance Analog Circuits in ASTRX/OBLX. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16:273–294, March 1996.

[21] Rodney Phelps, Michael Krasnicki, Rob A. Rutenbar, L. Richard Carley, and James R. Hellums. Anaconda: Simulation-Based Synthesis of Analog Circuits via Stocastic Pattern Search. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(6):703–717, June 2000.

[22] Michael Kransnicki, Rodney Phelps, Rob A. Rutenbar, and L. Richard Carley. MAELSTROM: Efficient Simulation-Based Synthesis for Custom Analog Cells. In *Proc. Design Automation Conference*, pages 945–950. IEEE/ACM, June 1999.

[23] Michael Kransnicki, Rodney Phelps, James R. Hellums, Mark McClung, Rob A. Rutenbar, and L. Richard Carley. ASF: A Practical Simulation-Based Methodology for the Synthesis of Custom Analog Circuits. In *Proc. International Conference on Computer Aided Design*, pages 350–357. IEEE/ACM, November 2001.

[24] George G.E. Gielen and Rob A. Rutenbar. Computer-Aided Design of Analog and Mixed-Signal Integrated Circuits. *Computer-Aided Deisgn of Analog Integrated Circuits and Systems*, pages 3–30, 2002.

[25] F. Gardner. Charge-Pump Phase-Locked Loops. *IEEE Trans. Come.*, 28(11), November 1980.

[26] M. E. Frerking. *Crystal Osillator Design and Temperature Compensation*. Van Nostrand Reinhold, New York, 1978.

[27] J. K. Clapp. Frequency Stable LC Oscillators. In *Proc. IRE*, volume 42, pages 1295–1300, August 1954.

[28] A. B. Grebene. The Monolithic Phase-Locked Loop – A Versatile Building Block. *IEEE Spectrum*, 8:38–49, March 1971.

[29] Roland E. Best. *Phase-Locked Loops: Design, Simulation, and Applications*. New York: McGraw-Hill, 1997.

[30] Stefanos Sidiropoulos and Mark Horowitz. A Semidigital Dual Delay-Locked Loop. *IEEE journal of Solid-State Circuits*, 32(11):1083–1092, November 1997.

[31] J. I. Brown. A Digital Phase and Frequency-Sensitive Detector. *Proc. of IEEE*, pages 717–718, April 1971.

[32] J. D. Alexandar. Clock Recovery from Random Binary signals. *Electronics Letters*, 19:541–542, October 1975.

[33] Ilya I. Novof, John Austin, Ram Kelkar, Don Strayer, and Steve Wyatt. Fully Integrated CMOS Phase-Locked Loop with 15 to 240MHz Locking Range and +/- 50ps Jitter. *IEEE Journal of Solid-State Circuits*, 30(11):1259–1266, November 1995.

[34] John Maneatis. Low-Jitter Process Independent DLL and PLL Based on Self-Biased Techniques. *IEEE Journal of Solid-State Circuits*, 31(11), November 1996.

[35] R Baghwan and A Rogers. A 1GHz Dual-Loop Microprocessor PLL with Instant Frequency Shifting. In *Proc. International Conference on Solid-State Circuits*, pages 336–337. IEEE, Feburary 1997.

[36] Stefanos Sidiropoulos, Dean Liu, Jaeha Kim, Gu-Yeon Wei, and Mark Horowitz. Adaptive Bandwidth DLLs and PLLs using Regulated Supply CMOS Buffers. In *Proc. Symposium on VLSI Circuits*, pages 124–127. IEEE, June 2000.

[37] Anatha Chandrakasan, William Bowhill, and Frank Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2000.

[38] Mozhgan Mansuri and Chih-Kong Ken Yang. A Low-Power Low-Jitter Adaptive-Bandwidth PLL and Clock Buffer. In *ISSCC Digest of Technical Papers*, pages 430–431. IEEE, Febuary 2003.

[39] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, Reading, MA, 1994.

[40] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

[41] Bjarne Stroustrup. *The C++ Programming Language, Third Editition*. Addison Wesley, Reading, MA, 1997.

[42] Lee Tavrow. *Schematic User Environment*. Micro-Magic Inc., Sunnyvale, CA, 1994.

[43] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison Wesley, Reading, MA, 1988.

[44] Jon Orwant, Jarkko Hietaniemi, and John Macdonald. *Mastering Algorithms with Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1999.

[45] Mozhgan Mansuri, Dean Liu, and Chih-Kong Ken Yang. Fast Frequency Acquisition Phase-Frequency Detector for GSamples/s Phase-Locked Loops. *IEEE Journal of Solid-State Circuits*, 37(10):1331–1334, October 2002.

[46] W. Dally and J. Poulton. *Digital Systems Engineering*. New York: Cambridge University Press, 1998.

[47] robert F. Pierret. *Semiconductor Device Fundamentals*. Addison Wesley, Reading, MA, 1996.

[48] *PathMill User Manual*. Synopsys, Inc., Mountain View, CA, 1996.

[49] *PrimeTime UserGuide*. Synopsys, Inc., Mountain View, CA, 2002.

[50] *Verlog-XL User Maual*. Cadence, 1996.

[51] F. Klass. Semi-Dynamic and Dynamic Flip-Flop with Embedded Logic. In *Proc. Symposium on VLSI Circuits*, pages 108–109. IEEE, June 1998.

[52] Chih-Kong Ken Yang. *Design of High-Speed Serial Link in CMOS*. PhD thesis, Stanford University, December 1998.

[53] Sungjoon Kim, Kyeongho Lee, Deog-Kyoon Jeong, Davod D. Lee, and Andreas G. Nowatzyk. An 800Mbps Multi-Channel CMOS Serial Link with 3x Oversampling. In *Proc. Custom Integrated Circuits Conference*, pages 451–452. IEEE, 1995.

[54] Chih-Kong Ken Yang, Ramin Farjad-Rad, and Mark Horowitz. A 0.6$mu$m CMOS 4Gb/s Transceiver with Data Recovery using Oversampling. In *Proc. Symposium on VLSI Circuits*, pages 71–72. IEEE, June 1997.

[55] Jaeha Kim and Mark Horowitz. Adaptive Supply Serial Links with Sub-1V Operation and Per-Pin Clock Recovery. In *ISSCC Digest of Technical Papers*, pages 268–269. IEEE, Febuary 2002.

[56] Ming-Ju Edward Lee, William J. Dally, and Patrick Chiang. Low-Power Area-Efficient High-Speed I/O Circuit Techniques. *IEEE Journal of Solid-State Circuits*, 35(11):1591–1599, November 2000.

[57] Jaeha Kim and Mark Horowitz. Adaptive Supply Serial Link with Sub-1-V Operation and Per-Pin Clock Recovery. *IEEE Journal of Solid-State Circuits*, 37(11):1403–1413, November 2002.

[58] Neolinear. NeoCircuit. `http://www.neolinear.com/sections/products_solutions/neocircuit_brochure.pdf`.

[59] Analog Design Automation. Creative Genius. `http://www.analogsynthesis.com/products/creative_genius.php`.

[60] Barcelona Design. Prado Synthesis Platform. `http://www.barcelonadesign.com/products/prado.asp`.