Cycle Domain Simulator for Phase-Locked Loops

by

Norman Karl James, B.S.E.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 1999

Cycle Domain Simulator for Phase-Locked Loops

Approved by
Supervising Committee:

_____

_____

Acknowledgements

I acknowledge my colleagues at IBM for their continuing help and support throughout my career.

November 16, 1999

Abstract

Cycle Domain Simulator for Phase-Locked Loops

Norman Karl James, M.S.E.

The University of Texas at Austin, 1999

Supervisor: Brian L. Evans

As computers become faster and more complex, clock synthesis becomes critical. Due to the relatively slower bus clocks compared to the processor, it is necessary to use phase-locked loops (PLL) for multiplication and phase aligning of the clocks.

A PLL is composed of both digital and analog components and is not modeled well in a design environment for digital systems. There are design tools available that are more adept for doing PLL simulations; however, they can be very costly and are still not suitable for the way PLL's are used in computer systems. The goal of this report is to discuss current ways of simulating PLL's, then introduce a new simulator that is specifically designed for simulating PLL's used in computer systems.

Table of Contents

List of Tables

List of Figures

INTRODUCTION

Chapter 1: Introduction

Phase-locked loops (PLL's) are used in synchronous computer systems throughout the clock distribution to multiply and phase align the primary clocks. They can be arranged in series or parallel and can be on a separate chip or integrated into another chip. The computer as a synchronous system implies that each replicated (synthesized) clock has a known phase and frequency relationship to its reference. This makes the phase relationship (phase error) an important parameter in chip-to-chip communication. Also since chips have a upper frequency limit to which they operate, knowing the frequency distribution is imperative. These phase and frequency errors are known as jitter. Jitter is the main parameter that any simulator should be able to predict.

There are several different simulators used in industry for PLL simulations, each one having its advantages and disadvantages. Traditionally, PLL's implemented on a chip (microprocessor, digital signal processor, ASIC, clock chip, etc.) are modeled with general purpose simulators such as SPICE. Other simulators include EESof, Matlab, and analytical/behavioral models. The main aspects that make a PLL difficult to simulate are as follows:

- Two sources of feedback – main feedback ("loop" in phase-locked loop) and oscillator feedback
- High frequency clocks in conjunction with low frequency time constants

- Picosecond accuracy
- Digital components mixed with analog components (mixed-mode)
    - Digital – phase detector, frequency divider
    - Analog – voltage-controlled oscillator (VCO), charge pump, loop filter

A PLL simulator should:

- Run fast
- Model CMOS processes
- Predict jitter (clock stability)
- Predict lock range (frequency range over which the PLL will lock)

1.1 SPICE

SPICE is a well-known general purpose time-domain circuit simulator [1][2]. When simulating PLL's, a high frequency component such as the clock must be simulated in conjunction with low frequency components, such as the lock time [1]. The clock period can be on the order of 1 ns and the time constants associated with the lock time (time required for PLL to phase/frequency lock on its reference) could be on the order of 1 ms.

Sampling a 1 GHz square wave requires a sample roughly every 50 ps for 1 ms, which becomes 20 million time points [2]. This is just for one node of the schematic. This in conjunction with solving a matrix equation for each time step for thousands of transistors is not feasible in a reasonable amount of time.

Although SPICE, given accurate models, can perform an accurate simulation, it can be very time consuming. The utilization of a voltage-controlled

tuned circuit as the VCO that does not require feedback to produce an oscillating output is a method that can be used to reduce the simulation time [3]. Despite the reduce simulation time, the number of data points and calculations is still immense.

The advantage of SPICE is that the models that accurately define a CMOS process are readily available. SPICE will typically be used for determining the VCO frequency range and doing independent component tuning because of the superior model accuracy.

1.2 MATLAB

Matlab is a general purpose analysis tool by the MathWorks. It has toolboxes available for analysis of signal processing, communications, and control systems. Simulink is the graphical interface for building simulation models.

Included with the communications toolbox are charge-pumped PLL models. These models run fairly fast but still generate large amounts of data. The Matlab models would have to be significantly modified to simulate a PLL built in a real CMOS process. Matlab still has the notion of time steps which leads to some of the same problems with SPICE, but does not have to make transistor-level calculations. Clearly accuracy would suffer if the analytical model did not mimic the behavior in silicon. Efforts have been made to increase the accuracy of Matlab simulations by better modeling of the PLL blocks to reflect the CMOS process [4][5].

1.3 ADVANCED DESIGN SYSTEM

The Advanced Design System is an extensive software tool published by HP EESof. It has many different modules to it including microwave, RF, high speed interconnect, and device modeling and extraction.

For PLL simulations, EESof uses a technique called circuit envelope simulation [1]. It accepts the input stimulus as RF carriers with time-varying complex envelopes (i.e. amplitude and phase modulations). The output solution is represented as a sum of the RF carriers and their harmonics, each having a time-varying complex envelope. Circuit Envelope has a fundamental advantage over time-domain simulators in that the time step-size need only be small enough to capture the bandwidth of the modulation envelope (e.g. 30 kHz), instead of the RF carrier (e.g. 1 GHz) [1].

The Circuit Envelope technique works well with the continuous time aspects of the PLL, but for components such as the digital phase detector it is more awkward. PLL's in communication applications use sinusoidal phase detectors which is more easily used with EESof.

1.4 ANALYTICAL/BEHAVIORAL MODELS

Traditionally, the textbook *s*-domain PLL model has been used to model the behavior of the loop in the locked state [9]. The input and output waveforms are assumed to be sinusoidal and the phase detector is modeled as a linear analog

multiplier with an inherent ideal lowpass filter [6]. Many charge-pumped CMOS PLL's do not behave exactly like the model represents.

Z-domain models for discrete-time PLL's have been developed as well. The *z*-domain model takes into account the sampled nature of the digital phase detector and accurately predicts the overall loop performance [6]. Although useful for predicting the input to output jitter relationship of the loop, the model is not a simulator. It would be extremely difficult to map it into the continuous time-domain and use for spread spectrum and noise effects.

Other behavioral models have been developed using Analog Hardware Definition Language (AHDL). SprectreRF is a software design tool published by Cadence that uses AHDL to describe analog circuits in terms of their behavior [7][8]. Equations, state machines, or discrete circuit elements can be used to describe the PLL components in AHDL. The simulations are done in the time-domain similar to SPICE, so there is still the data explosion problem. Similar to AHDL, several custom simulators have been developed using C and other programming languages [10][11].

The main problem associated with all of the behavioral models mentioned is the fact that they work in a domain (time-domain, frequency-domain, *s*-domain, *z*-domain) that is not optimum for PLL's. Either the domain is not robust enough or not efficient enough. Ideally, the simulation should relate back to the time-domain, since that is the world in which we live. It is the goal of this report to introduce a new domain that is well-suited to PLL's (and clocks in general) and a simulator that uses this domain to perform robust and efficient simulations. The

simulator is called CycleSim. In order to understand CycleSim and its benefits, it is necessary to give a summary of PLL's and their important design parameters.

1.5 Overview of Report

In general, the goal of this report is to provide a way to accurately and efficiently determine the stability of a PLL. Because the PLL sources clocks throughout the system, it is critical to the system and chip timings. The computer industry is driven by obtaining higher clock frequencies which puts emphasis on the PLL's performance.

Chapter 2 gives an overview of PLL's including applications, the individual building blocks, and the system as a whole. Chapter 3 goes beyond the basics and covers important parameters associated with a PLL's performance and impacts from noise. Finally, Chapter 4 introduces a new simulator, shows how it works, and presents results and comparisons to hardware.

PLL PRIMER

Chapter 2: The Basics

The PLL is typically a misunderstood circuit that is treated as a block box to many system designers. Although books have been written on the design of PLL's, this chapter is only intended to give an overview of its basic operation.

2.1 PLL APPLICATIONS

A PLL has many applications ranging from communications to clock synthesis in computer system. This report mainly focuses on uses in a computer system; however, in this chapter other usages are discussed for a more broad understanding.

2.1.1 Communications

In the field of communications, a PLL is typically used for modulation and/or demodulation. In the common FM stereo receiver, a PLL can be used to demodulate the music off of a carrier wave.

2.1.2 Computers

In computers, it is very common to have multiple PLL's in the clock distribution. Each microprocessor, memory controller, I/O controller, etc., will contain a PLL to perform frequency multiplication if it is desired to run the core of the chip at a higher frequency than the bus. A PLL within a chip may also be used to phase align the clock so that all chips have a common reference in time.

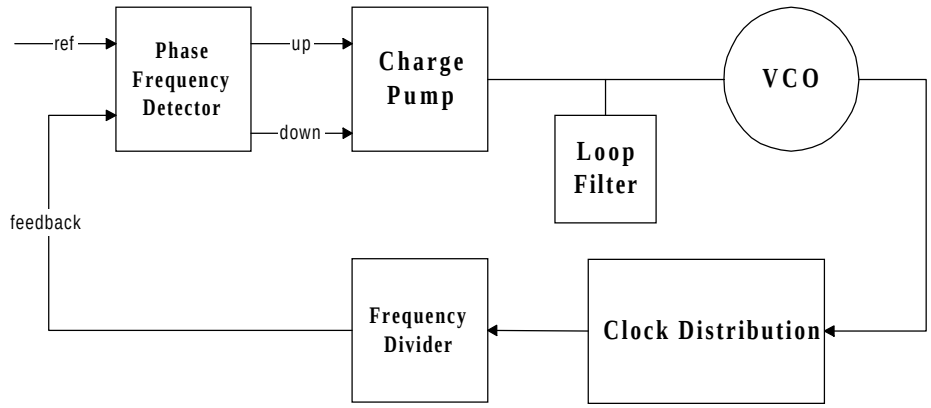

Figure 2.1: Phase-Locked Loop Block Diagram

This allows for synchronous data transfers between chips. The rest of the report will focus on PLL's used in this context as opposed to communications.

2.2 BASIC OPERATION

In general, a PLL used in computer systems is called a charge-pumped PLL and is made up of the following components: phase-frequency detector (PFD), charge pump, loop filter, voltage-controlled oscillator (VCO), and frequency divider. The basic block diagram is shown in Figure 2.1. Essentially, a reference clock is driven into one input of the PFD and a feedback clock is fed into the other input. The PFD outputs an up or down pulse depending on whether the feedback is leading or lagging the reference. The up and down pulses are proportional to the difference in phase of the two clocks. The pulses are translated into current by the charge pump, which either forces current into or out of the loop filter. A basic loop filter integrates the current and generates a voltage

which is the control for the VCO. If the PFD is generating up pulses, the control voltage is "pumped" up causing the frequency of the VCO to increase. The frequency divider, divides the VCO output by the desired bus to chip multiplication factor.

2.2.1 Components

A classical PLL is made up of both digital and analog components. The PFD and dividers are considered digital and the charge pump, filter, and VCO are analog. The following sections give a brief summary of each component along with its desirable features.

2.2.1.1 Phase-Frequency Detector (PFD)

A PFD may be implemented as two flip-flops with additional logic to reset the latches. A simplistic PFD is shown in Figure 2.2. The PFD can be viewed as a state machine whose state is U (up) when the ref is leading the feedback, D (down) when the feedback is leading the ref, and N (null) when neither is true. Figure 2.3 shows a SPICE simulation of a PFD.

The most desirable feature of a PFD is to have zero dead-zone. Dead-zone occurs when the PFD detects 0 phase error when phase error is present. Zero dead-zone implies that the PFD can detect any amount of phase error. With basic PFD's, the dead zone can be on the order of 100 ps. Well-designed PFD's can obtain zero dead-zone.

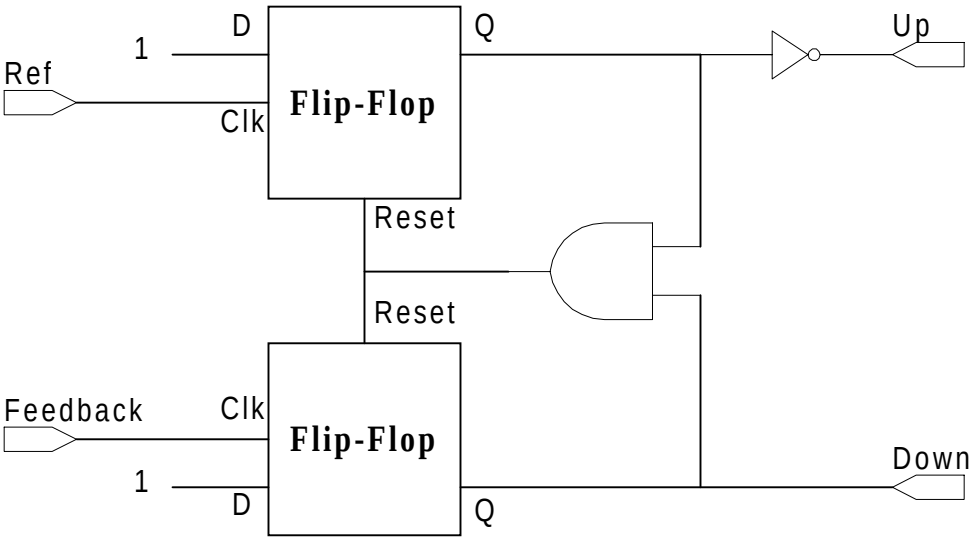

Figure 2.2: Phase-Frequency Detector diagram

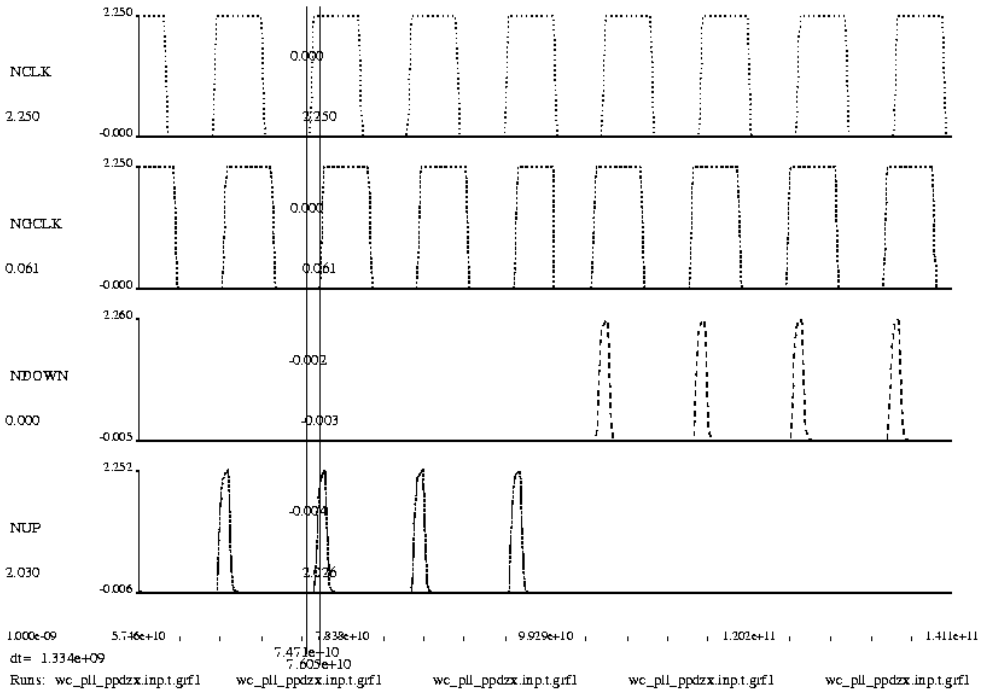

Figure 2.3: Phase-Frequency Detector SPICE output showing up and down pulses

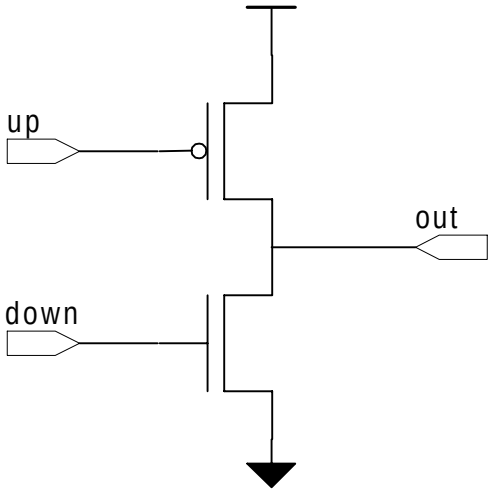

Figure 2.4: Charge Pump diagram

2.2.1.2 Charge Pump

A charge pump delivers a pump current of $\pm I_p$ to the loop filter whenever the PFD logic produces an up or down [2]. Duration of an active state is determined by the magnitude of the phase error [2]. Typically the charge pump is tuned in conjunction with the PFD to achieve zero dead zone.

2.2.1.3 Loop Filter

In classical PLL's, the loop filter is an RC network with one pole and one zero. Some loop filters also have a gain factor included. The new generation of PLL's use a capacitor only and have an additional current port as an input to the VCO to create the zero of the filter [3]. Two charge pumps are required now, one to apply charge to the capacitor which is connected to the standard voltage port of

the VCO. The other pumps current directly into the VCO which allows instantaneous changes in phase. This approach yields greater flexibility in tuning the characteristics of the PLL. It is also easier to implement in silicon.

2.2.1.4 VCO

There are several types of VCO's including current-starved ring, delay interpolating, and LC tank oscillators. A current-starved ring oscillator is typically a ring of inverters with some sort of current control which is the frequency control. These types of VCO's typically have extremely high gains and wide frequency ranges.

The delay interpolating VCO's consist of two delay chains, one having half the delay of the other, and an analog mixer. The mixer is controlled by the control voltage. If the control voltage is at its maximum, then the mixer selects the shortest delay path which yields the highest frequency. If the control voltage is at its minimum, then the mixer selects the longest delay path which yields the lowest frequency. If the control voltage is somewhere in between, then the mixer mixes the two delay chains appropriately. Advantages of delay interpolating VCO's are known for their low gain and easily tunable ranges.

LC tanks are difficult to implement in silicon, so they are not used extensively in microprocessors. They consist of a LC network whose resonant frequency causes the oscillation. LC tanks are extremely stable and are insensitive to power supply fluctuations, but do not have a wide tunable range.

2.2.1.5 Clock Distribution

The clock distribution is not typically contained in the PLL block, but it is an important part of the PLL's operating characteristics. The clock distribution can be treated as a variable delay for analysis. Since the clock distribution can add jitter to the system, it could be a noise source to the loop.

2.2.1.6 Divider

A simple divider can be implemented as an edge-triggered flip-flop. The output toggles with every input rising edge causing a divide by 2. Flip-flops in series can perform binary divides. To perform non-binary divides, counters can be used to "pick off" certain edges to send to the output. The number of edges skipped determine the divide ratio.

2.2.2 Putting It All Together

The theories describing PLL behavior become very important when trying to simulate the behavior. Rather than deriving the equations, I will simply give the important formulas that are used in PLL design. When a PLL initially starts up and tries to "lock" on the reference frequency, it behaves like a dampened sine wave. In other words, if the control voltage into the VCO (or the period of the output clock from the VCO) where plotted versus time, it would look like a dampened sine wave. The equations that govern this behavior are as follows:

$$\omega_n = \sqrt{\frac{KI_p}{2\pi C}} \quad [12]$$

ω_n is the natural frequency
K is the VCO gain in Hz/volt
I_p is the charge pump current in amps
C is the filter capacitor value

$$\zeta = \frac{RKI_p}{2\omega_n} \quad [12]$$

ζ is the dampening factor
R is the filter resistor value

The natural frequency and dampening factor are the primary design parameters for a PLL. These depend on the loop filter, VCO gain, and charge pump current. The natural frequency and dampening factor essentially relate to how fast the PLL tracks the reference signal.

2.3 CONCLUSIONS

This chapter is intended to give the reader a high level understanding of the PLL as a system of individual components. Chapter 3 goes into more detail of the interaction of the components and the important parameters associated with a PLL's performance.

Chapter 3: Beyond the Basics

Based on the basic operation of a PLL described in Chapter 2, this chapter describes the characteristics of a PLL that are important in design. Since the objective of a PLL in a microprocessor system is to obtain a near ideal clock no matter how much multiplication is desired, this chapter focuses on key measurements that make a clock perfect and sources that make a clock imperfect. Section 3.1 discusses the impact of jitter on PLL performance. Jitter tells the designer how close to ideal the clock is. Section 3.2 discusses the impact of noise on PLL performance. Section 3.3 summarizes other key PLL parameters. Section 3.4 concludes the chapter.

3.1 JITTER

In a computer system, it is ideal to have perfect clocks whose period and phase alignment do not vary. This variability is commonly known as jitter. There is no standard definition for how to measure and interpret jitter. Jitter occurs randomly. If the jitter were deterministic, then the PLL is not behaving properly or external noise is causing jitter modulation. Any model used for PLL simulation must accurately predict jitter. The next few sections attempt to give the most common definitions for jitter.

3.1.1 Period Jitter

Period jitter is simply a measure of the clock period over time. It is commonly specified peak-to-peak which is simply the maximum period measured minus the minimum period measured. Period jitter is important in microprocessors because the critical paths are highly dependent on the clock period. If the clock period becomes too short, then a timing violation could occur causing the system to hang or even worse, cause the system to have its data integrity compromised.

3.1.2 Cycle-to-Cycle Jitter

Cycle-to-cycle jitter is how much the period can vary from one cycle to the next. Cycle-to-cycle jitter can also be referred to as “cycle deviation” and is a type of short-term jitter. This type of jitter is important in microprocessors because two cycle paths depend on the previous clock to latch data.

3.1.3 Long-Term Jitter

Long-term jitter is even more misunderstood than short-term jitter. Overall, long-term jitter defines how much the clock phase can deviate from where an ideal clock would be. Long-term jitter is also known as accumulated phase error and tracking skew. Table 3.1 illustrates the effects long-term jitter with respect to period jitter. The clock period shown in columns 1 and 3 are the absolute measured clock periods for 5 cycles.

The period jitter is again Period(max)-Period(min). For the ideal clock this calculation is 2-2 which is 0. So as expected the ideal clock has no jitter. For the real clock, the calculation is 2.1-2 or 0.1. For the long-term jitter, the real

Ideal clock period	Ideal clock position	Real clock period	Real clock position
2	2	2	2
2	4	2.1	4.1
2	6	2.1	6.2
2	8	2.1	8.3
2	10	2.1	10.4

Table 3.1: Jitter table

clock deviates from the ideal clock a maximum of 0.4 (10.4-10), so the long-term jitter for this clock is 0.4. This deviation from the ideal clock will be called phase error for this report. Typically phase error is in terms of radians, but for this report it will be in terms of time.

3.2 NOISE

Since the magnitude of the jitter is the most critical aspect of PLL's for computer systems, it is important to understand what are the causes of it. The most common contributors to jitter in a computer system are power supply noise on the VCO and noise on the clock distribution. Any model used for simulation should include these effects.

3.2.1 VCO

A high gain VCO is extremely sensitive to noise on the power supply. Changes in voltage directly cause changes in the output frequency. A VCO's

power supply rejection can be specified in terms of picoseconds/millivolt (ps/mV). Clearly the designer should be confident that the power supplied the VCO is quiet. Just recently, PLL chip designers are using on chip voltage regulators to achieve quiet power [13].

3.2.2 Clock Distribution

The clock distribution can introduce a significant amount of delay. Anytime there is delay caused by transistors, that delay becomes sensitive to the voltage. The amount of jitter introduced on the clock is directly proportional to the amount of delay and the amount of voltage fluctuation on the power plane.

3.3 Other Important PLL parameters

3.3.1 Lock Range

Lock range is the minimum and maximum frequency of the reference on which the PLL can lock. This is primarily depends on the VCO frequency range and the divider settings.

3.3.2 Loop Bandwidth

The loop bandwidth is closely related to the natural frequency and simply means how fast the control loop tracks the reference. This becomes extremely important for spread spectrum clocking. Spread spectrum clocking means that the reference clock is slowly being frequency modulated at a carrier frequency of 30 kHz. The 30 kHz is picked because it is above the audio band, yet it is small enough so as not to affect the system timing. The loop bandwidth should be at

least one order of magnitude higher than the modulation frequency [14]. Spread spectrum is used to reduce electromagnetic interference in computer systems. It essentially keeps the magnitude of the emissions away from one particular frequency.

3.4 CONCLUSION

Jitter, noise, lock range, and loop bandwidth must either be modeled properly or predicted for the simulator to be of any value. Chapter 4 introduces a simulator that addresses these issues and provides an efficient and accurate means of modeling or predicting the parameters.

SIMULATION

Chapter 4: Cycle Domain Simulator (CycleSim)

Based on the brief overview of PLL's and key PLL design parameters given in Chapter 2 and 3, respectively, this chapter covers a new method of simulation. The purpose of CycleSim is not to perform basic loop simulations (although it could), because the previously mentioned tools are capable of this task. Instead the purpose is to perform simulations with sufficient speed and accuracy to give the designer the flexibility to finely tune a PLL for low jitter, or try different techniques or ideas that are not a part of the standard solutions. CycleSim is a simulator in the true sense of the word. In other words, it is not simply solving equations and plotting the answer. The code makes adjustments, one iteration at a time and has no idea where it will end up. This is essentially the same behavior as a PLL.

Section 4.1 is an overview of the CycleSim and introduces the concepts associated with the simulator. Section 4.2 covers the implementation. Section 4.3 outlines the results along with corralation to hardware. Finally, Section 4.4 describes how CycleSim would be used in practice.

4.1 OVERVIEW

CycleSim is written in ANSI C++ and has been compiled under Windows NT and AIX. The simulator is neither a time-domain or frequency-domain

simulator. It is a cycle-domain simulator. The basic premise is that if a PLL were considered a black box with one input and one output, both input and output would be clocks and the only significant aspect of these clocks is the period and phase. The amplitude aspects are of no importance since we are primarily concerned with the phase and frequency accuracy of the clock.

The simulator stores and works on rising edges only. This is equivalent to taking a sampled clock (from a simulator or hardware) and extracting the zero-crossing points which are essentially the points in time at which the waveform rises through some threshold (typically the midpoint). These points in time are referred to as time tags for the purposes of this report. To illustrate the nature of the time tags, Figure 4.1 is a plot of Simulation Iterations vs. Time Tags for clocks where one clock is three times faster than the other clock.

CycleSim has no notion of a time step size. For each iteration, the time tags for the multiple nodes are adjusted. For example, a simple simulation would contain three time tagged nodes, the reference clock, the VCO output, and a divider output. Since the reference frequency is fixed, the time tags simply increase each iteration by the period. The VCO is more complicated in that its period is constantly being adjusted by power supply noise, the control voltage, and a feedfoward current port. The divider is simple because it just changes the period of the incoming time tags by the desired divider setting. The only circuit issues the simulator cares about are ones that adjust the time tags and that the time tags can easily be mapped into the time domain.

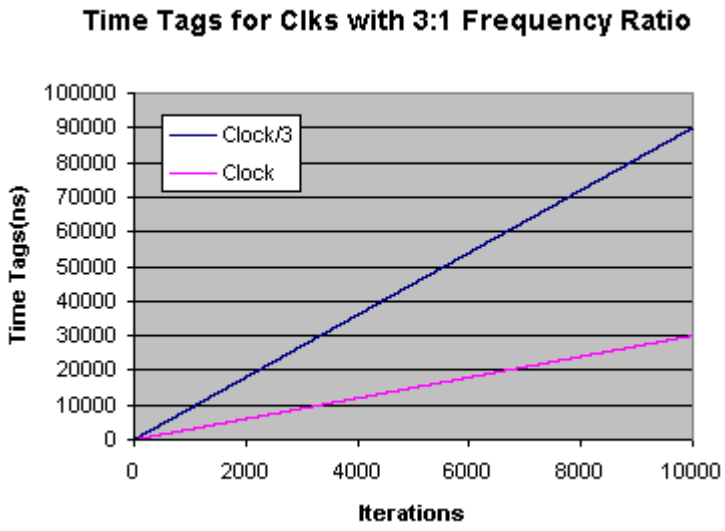

Figure 4.1: Time Tags illustration showing two clocks with a 3:1 frequency ratio

4.2 IMPLEMENTATION

The main simulation object is *PLLSim.* The *PLLSim* object initializes the various components then iterates for the desired number of steps. Figure 4.2 is the source code for the main simulation loop.

4.2.1 Pulse Generator

The pulse generator object generates the time tags with constant period for the reference clock into the PLL. It can also modulate the clock through a lookup table. Using a table can be used to generate spread spectrum clocks, frequency steps and phase steps. All of these can be used to characterize the behavior of the PLL. Figure 4.3 is the *step* method which the main simulator instance calls for each iteration.

```
void PLLSim::go(int num_steps,char* simfile)
{
     loadSimFile(simfile);

     for(int i=0;i<num_steps;i++)
     {
          stepRef();
          stepFeedback();
          cntl_i=0.0;
          ph=0.0;
          if (i%div.div==0)
          {
               ph=pfd.step(this);
               cntl_i=cp.step(ph);
               cntl_v=filt.step(ph,cntl_i);
          }
          // Print any desired outputs here
     }
}
```

Figure 4.2: Code for main simulator loop

```
double PulseGen::step()
{
     // Does a table lookup for a period
     // modifier percentage.
     // The getY() function interpolates in the table.
     if (bSSC)
     {
          double xmax=oSSC.xx[oSSC.size-1];
          double xnorm=dTime-int(dTime/xmax)*xmax;
          dPeriod=dPeriodNom*oSSC.getY(xnorm);
     }
     // Performs a frequency step
     if (dTime>dRate && dDeltaPeriod!=0)
     {
          dPeriod=dPeriod+dDeltaPeriod;
          dDeltaPeriod=0.0;
     }
     // Performs a phase step
     if (dTime>dRate && dDeltaPhase!=0)
     {
```

```
            dTime=dTime+dDeltaPhase;
            dDeltaPhase=0.0;
        }
        dTime=dTime+dPeriod;
        return dTime;
}
```

Figure 4.3: Code snippet of Pulse Generator “step” method

4.2.2 Phase-Frequency Detector (PFD)

The PFD object was perhaps the most difficult component to implement. At first glance, it seems as if the phase detector could be implemented by just subtracting the two input time tags. This does give a phase difference, but if the two signals are not frequency locked, then multiple edges of one input could occur, before one edge occurs on the other input. Now the edges would be out of sync for each iteration. For this reason, the PFD needs to have control of the pulse generator and VCO. Control means that the PFD can call the *step* method which essentially generates the next time tag.

The variables *e1* and *e2* keep track of the state and edges of the PFD. At time 0, the first rising edge is stored in *e1* and then it waits for the first rising edge on the other input before it outputs the phase. If there is no rising edge on the other input, then it must step the correct frequency source until a rising edge occurs. This is the purpose of the *while* loops. Figure 4.4 shows a state machine that illustrates the logic. Figure 4.5 shows the code that performs these functions.

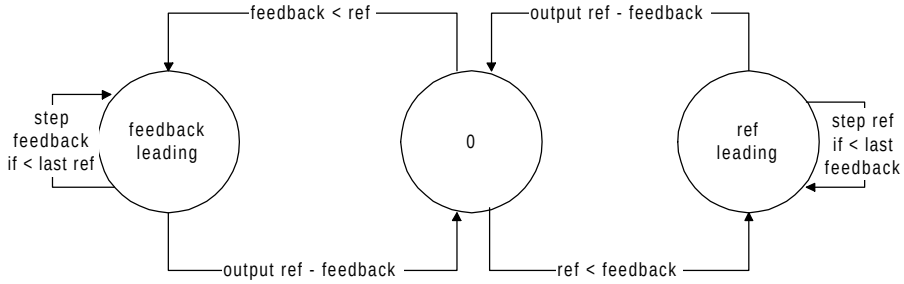

Figure 4.4: State machine describing Phase-Frequency Detector

```
double PFD::step(PLLSim* pll)
{
     double ph=pll->feedback-pll->ref;
     int s=0;


     if (ph>=0) //positive means ref is leading
     {
          s=1;
          //Iterates VCO
          while (pll->ref<=e2)
               pll->stepRef();

          e2=pll->feedback;
          e1=pll->ref;
     }
     else
     {
          s=-1;
          //Iterates VCO
          while (pll->feedback<=e2)
               pll->stepFeedback();


          e1=pll->feedback;
          e2=pll->ref;
     }
     ph=s*(e2-e1);
     return ph;
}
```

Figure 4.5: Code snippet of Phase-Frequency Detector “step” method

4.2.3 Charge Pump

The charge pump object *step* method takes the phase as its input and outputs the corresponding current. The translation is done through a lookup table.

4.2.4 Filter

The filter object behaves like a capacitor, in fact a capacitor value is a parameter for it. The *step* method takes a current and the phase error as its input. It integrates the current over time (the phase error) to compute a voltage which is then applied to the capacitor. The voltage on the capacitor is limited to above 0 and below the power rail (vdd is parameter in the simulator).

4.2.5 VCO

The basic VCO *step* method gets the nominal period for a given voltage through a table lookup. Then it adjusts it by a current gain. The current port is the zero for the loop filter, since the loop filter in this implementation does not have any resistors. The equation for the equivalent resistor value is as follows:

$$R_{eq} = \frac{K_j I_f}{K_0 I_p} \quad [13]$$

where:
K_j is the VCO current gain
I_f is the feedforward current
K_0 is the VCO voltage gain
I_p is the charge pump current

```
double VCO::step(double v,double i)
{
     // Table lookup
     dPeriodNom=1/oRange.getY(v);
     dPeriod=dPeriodNom-(i/dCurrentGain);
     dTime=dTime+dPeriod;
     return dTime;
}
```

Figure 4.6: Basic Voltage-Controlled Oscillator “step” method

More detailed implementations contain a power port so noise can be injected into the VCO. Figure 4.6 is the *step* method for the VCO object.

4.3 SIMULATION RESULTS

This section covers the results of the various tests run using CycleSim. The plots generated are direct output from the simulator. These plots are in the cycle domain (time vs. cycle (time)). Many jitter analysis packages for hardware characterization also display the data in this manner.

4.3.1 Simulation Speed

The simulations are several orders of magnitude faster than time-domain simulators such as SPICE. For 25,000 cycles, the simulator takes < 5 seconds to complete on a Pentium 133. Even scaled to slowest processors, the performance is still reasonable. Also, the number of data points created is the number of cycles times the number of nodes. Table 4.1 is a comparison of SPICE and CycleSim for a PLL lock simulation.

	SPICE	CycleSim
Simulation time (hours)	120	4[1]
Number of data points (output node)	10,000,000	25,000
Accuracy (ps)	10	.03[2]

[1] Includes the time to perform SPICE simulations on individual components
[2] Accuracy of SPICE simulation on individual components

Table 4.1: Simulation comparison for a Phase-locked loop lock test

4.3.2 Basics

The first test was to make sure the PLL would lock phase and frequency. Figure 4.7 shows the first 400 cycles of a PLL simulation on a plot of period vs. cycle. This is a typical dampened sine wave response. The parameters could be adjusted to give various natural frequencies and dampening factors. The "jaggedness" of the line in the *y* direction is jitter once the loop is locked.

4.3.3 Existing Model Correlation

Another way to validate the simulator's behavior is to make sure it is obeying the *s*-domain equations relatively well. It will never match exactly due to the "digital" nature of the PFD [9]. Figure 4.8 shows a plot of phase error vs. cycle. The frequency of the oscillation can be measured to calculate the natural frequency.

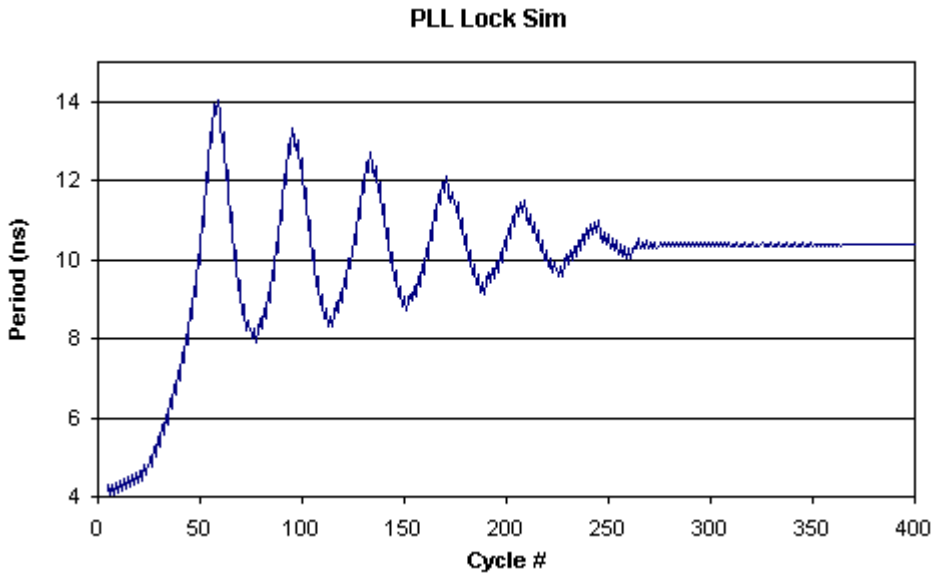

Figure 4.7: Lock simulation result

The equations are as follows:

$$\omega_n = \frac{2\pi}{T * \sqrt{1-\zeta^2}}$$

$$A = \ln\left(\frac{A_1}{A_2}\right) \qquad [15]$$

$$\zeta = \frac{A}{\sqrt{\pi^2 + A^2}}$$

where:
T is the period of the oscillation
A_1 is the first largest amplitude
A_2 is the second largest amplitude

for the plot below:
T = (72 cycles)*10.9ns = 785ns
A_1 = 5.165 V
A_2 = 3.777 V

$\zeta = 0.1$
$\omega_n = 1.286$ MHz * 2π

solving the s-domain equations gives:
$\zeta = 0.04$
$\omega_n = 1.281$ MHz * 2π
using $C = 100$ pF, $K_0 = 425$ MHz/V, $I_p = 60$ μA, $R_{eq} = 27\Omega$

The conclusion is that the results show close correlation to the analytical approach. It shows that the simulator is on the right track.

4.3.4 Hardware Correlation

A simulator is not much value unless it predicts the behavior of working hardware. Figure 4.9 is a period vs. cycle graph (also known as a cyclegraph) taken using jitter measurement hardware. The graph is showing the response of the PLL output during a step change on the power of the VCO. Figure 4.10 shows the results of the simulator for the same event. The idea is to match the frequency of the oscillation and the amplitude.

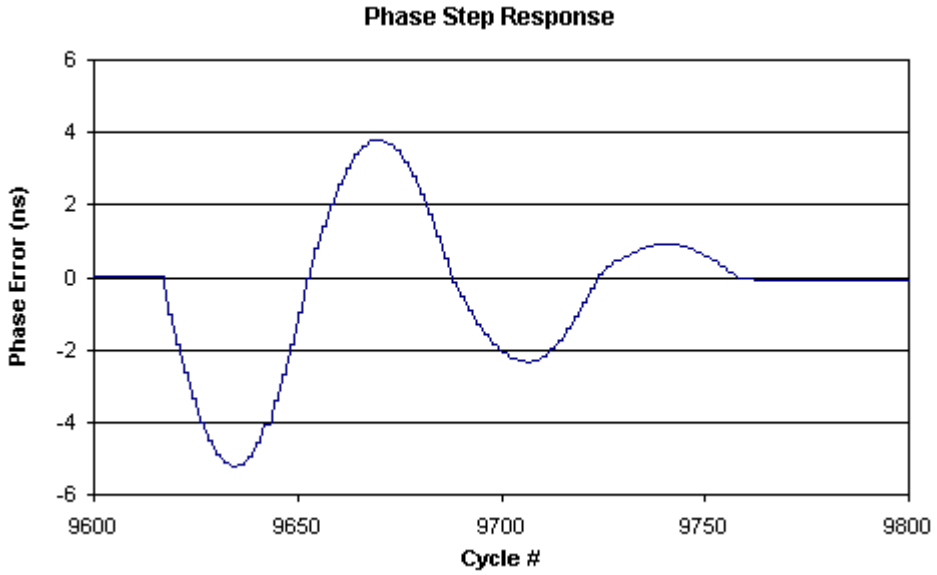

Figure 4.8: Phase step simulation result

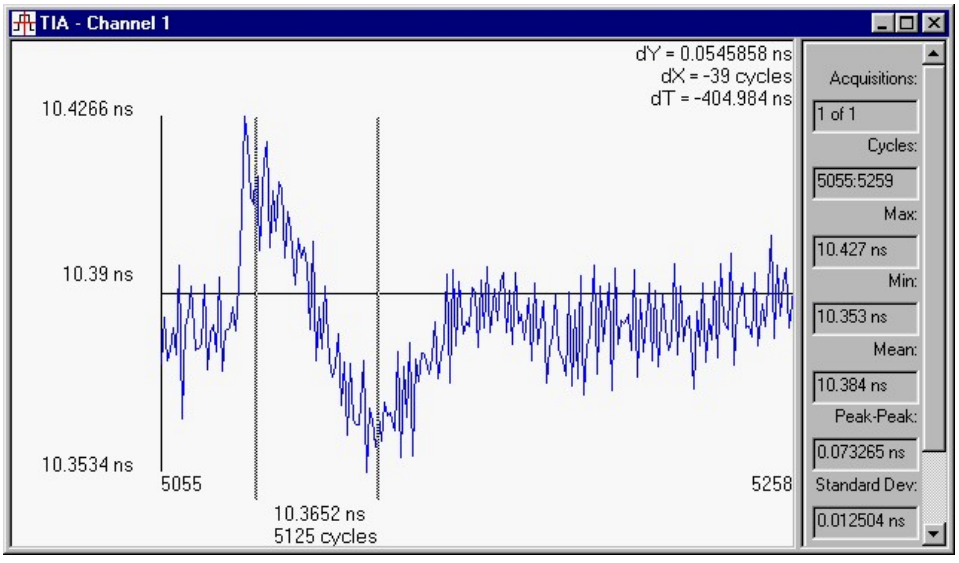

Figure 4.9: Step change of power on oscillator (Hardware measurement)

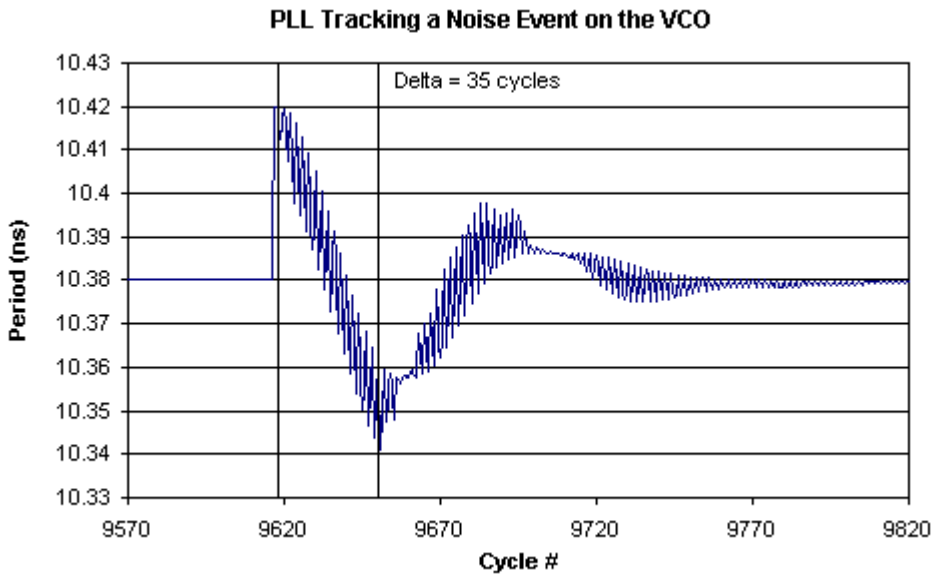

Figure 4.10: Step change in power on oscillator (CycleSim)

4.3.5 Advanced Applications

Now that the basic comparisons have been presented, we consider more complex examples. Spread spectrum clocking is becoming more popular in computer systems due to the higher frequencies and power. The challenging aspect to performing spread spectrum simulations is that the simulations have to be run for thousands of cycles because of the low modulation frequency. Also, spread spectrum clocks contain many frequency components due to the shape of the waveform. Figure 4.11 shows the cyclegraph for a typical spread spectrum clock.

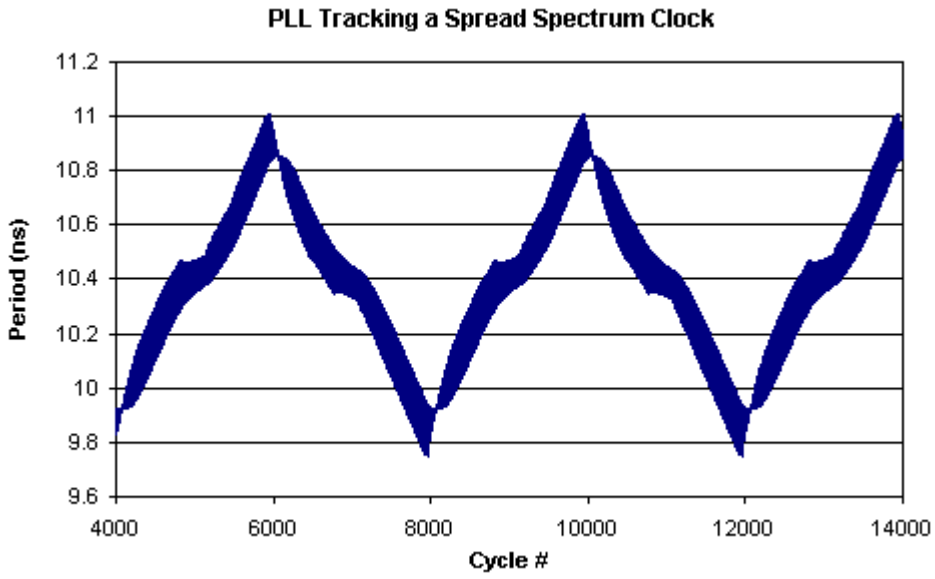

Figure 4.11: Spread spectrum clock cyclegraph

When analyzing a PLL whose reference is a spread spectrum clock, it is important to make sure the PLL's loop bandwidth is sufficient to track the modulation. Figure 4.12 shows the phase error of a PLL with a spread spectrum reference. Notice the deviation is ± 0.150 ns. Figure 4.13 shows another design with slightly higher bandwidth and its deviation is ± 0.06 ns.

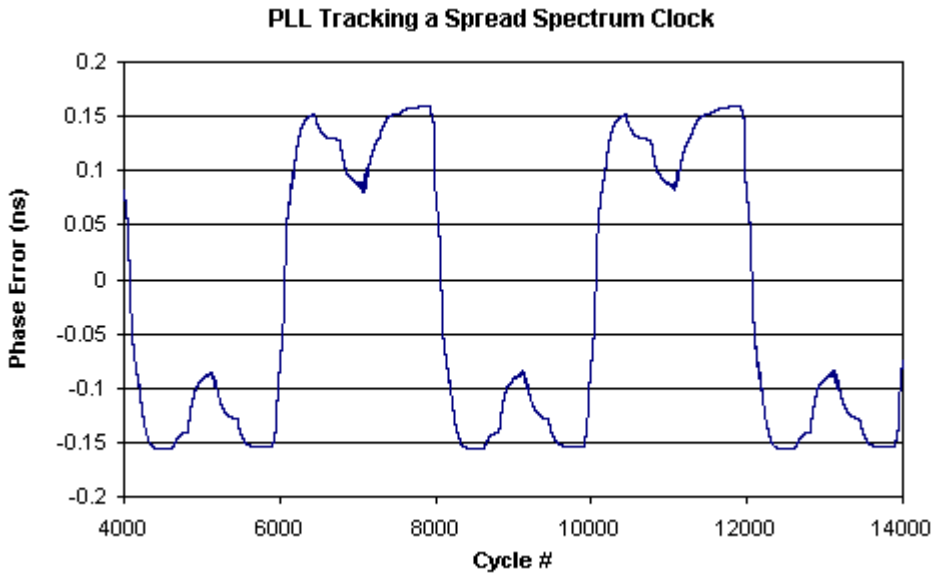

Figure 4.12: Phase error for low bandwidth phase-locked loop

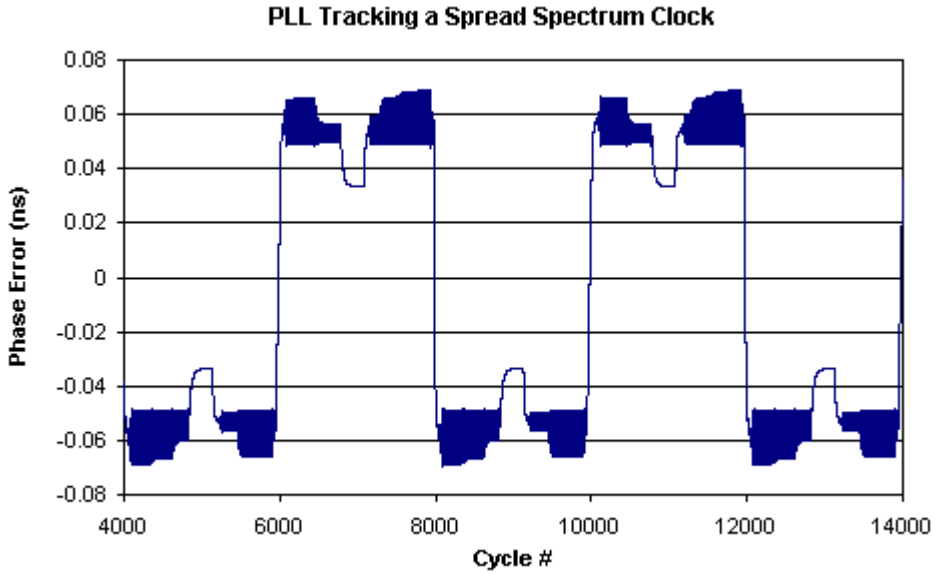

Figure 4.13: Phase error for high bandwidth phase-locked loop

4.3.6 MODEL USAGE FOR DESIGN

Now that some examples of the simulator's performance have been presented, we describe how the simulator could be actually used in the design process. The idea is to use SPICE to simulate the individual components, since all of the proper CMOS models are available. An automated extraction from the SPICE simulation results could be used to create the tables required for the cycle-domain simulator. SPICE could also be used for some of the required parameters, such as VCO current gain.

Now the designer will have a model that mimics the actual silicon, yet simulates extremely fast. Currently the model supports only tables for the phase error vs. charge pump current and control voltage vs. VCO frequency, but this could easily be expanded.

4.4 CONCLUSION

No simulator is perfect, but hopefully CycleSim makes the best compromises for PLL's. Using this simulator in conjunction with SPICE could be a very powerful tool for PLL designers. The fast simulation speed allows for new PLL topologies to be explored, along with taking the existing circuits to their limits. As the computers get faster, the accuracy of the clock becomes much more critical. The industry is starting to focus more on jitter and noise issues that affect PLL's because of the smaller margins allowed in the system. Hopefully CycleSim can be used to design a sub-picosecond PLL that is impervious to noise.

Appendix A – Complete Source Code

```
// PLLSim.h: interface for the PLLSim class.
˜
//
˜
//////////////////////////////////////////////////////////////////////
/////
˜

˜
#if
!defined(AFX_PLLSIM_H__D13DAEE2_769D_11D3_AD03_0080C8876A90__INCL
UDED_)
˜
#define
AFX_PLLSIM_H__D13DAEE2_769D_11D3_AD03_0080C8876A90__INCLUDED_

#include <fstream.h>

class InterpolateArray
{
public:
      InterpolateArray() { array=NULL; xx=NULL; yy=NULL; size=0;
}
      ~InterpolateArray() { if (array!=NULL) delete [] array;
            if (xx!=NULL) delete [] xx; if (yy!=NULL) delete []
yy; }

      void load(char* filename,double s,double mx);
      void load(char* filename);
      double getY(double x);
      void print();

      double* array;
      double* xx;
      double* yy;
      double step,max,min;
      int size;
};

class Divider
{
public:
      Divider();
      Divider(int d);
      ~Divider();
```

```
      void setup(int d);
      double step(double e);
      void print() { printf("Divider: div_ratio = %2d\n",div); }

      double e_last;
      int div;
};

class ChargePump
{
public:
      ChargePump();
      ~ChargePump();

      void setup(char* filename);
      double step(double p);
      void print() { printf("Charge Pump Table (seconds vs.
Amps):\n"); oI.print(); }

      InterpolateArray oI;
};

class Filter
{
public:
      Filter();
      ~Filter();

      void setup(double vv,double v,double c);
      double step(double p,double i);
      void print() { printf("Filter: Cap=%.6g pF\n",dCap*1e12); }

      double dVolts,dCap,vdd;
};

class VCO
{
public:
      VCO();
      ~VCO();

      void setup(char* filename);
      double step(double v,double i);
      void print() { printf("VCO Range Table (volts vs. Hz):\n");
oRange.print();
                  printf("VCO: current_gain=%.6g
Hz/Amp\n",dCurrentGain); }
      int iCycle;
      double dTime,dPeriod,dPeriodNom,dCurrentGain;
```

```
	InterpolateArray oRange;
};

class PulseGen
{
public:
	PulseGen();
	~PulseGen();

	void setup(double period);
	double step();
	void print();
	void doModulation(char* modfile);

	double
dPeriod,dPeriodNom,dTime,dDeltaPeriod,dRate,dDeltaPhase;
	int iCycle,bSSC;
	InterpolateArray oSSC;

};

class PLLSim;
class PFD
{
public:
	PFD();
	~PFD();

	double step(PLLSim* pll);

	double e1,e2;
};

class PLLSim
{
public:
	PLLSim();
	virtual ~PLLSim();

	void go(int num_steps,char *simfile,char* outfile);
	void stepRef();
	void stepFeedback();

	double ref,feedback,cntl_v,cntl_i,vco_out;
	int iIterations,iStartOutput;

	VCO vco;
	PulseGen pg;
	PFD pfd;
```

```
      ChargePump cp;
      Filter filt;
      Divider div;

};

#endif


// PLLSim.cc: implementation of the PLLSim class.
//
//////////////////////////////////////////////////////////////////////
#include "main.h"
#include "PLLSim.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
void main(int i,char *simfile[])
{
      PLLSim sim;
      if (i>2)
            sim.go(15000,simfile[1],simfile[2]);
      exit(0);
}

static void loadSimFile(char* filename,PLLSim* pll)
{
      fstream f;
      f.open(filename,ios::in|ios::nocreate);
      char buf[255];
      char* out;
      char* value;
      char seps[]="= \t\n";

      //defaults
      double vdd=1.8;
      double cap=100e-12;
      double vco_current_gain=350000;
      while (f.good())
      {
            f.getline(buf,254);
            out = strtok( buf, seps );
            if (out==NULL)
```

```
                continue;
            value=strtok( NULL, seps );

            if (!strcmp(out,"vco_file"))
                pll->vco.setup(value);
            if (!strcmp(out,"ref_period"))
                pll->pg.setup(atof(value));
            if (!strcmp(out,"div_ratio"))
                pll->div.setup(atoi(value));
            if (!strcmp(out,"vdd"))
                vdd=atof(value);
            if (!strcmp(out,"cp_file"))
                pll->cp.setup(value);
            if (!strcmp(out,"cap"))
                cap=atof(value);
            if (!strcmp(out,"sim_cycles"))
                pll->iIterations=atoi(value);
            if (!strcmp(out,"vco_current_gain"))
                pll->vco.dCurrentGain=atof(value);
            if (!strcmp(out,"modulation_file"))
                pll->pg.doModulation(value);
            if (!strcmp(out,"start_output"))
                pll->iStartOutput=atoi(value);
            if (!strcmp(out,"delta_period"))
                pll->pg.dDeltaPeriod=atof(value);
            if (!strcmp(out,"delta_phase"))
                pll->pg.dDeltaPhase=atof(value);
            if (!strcmp(out,"modulation_rate"))
                pll->pg.dRate=atof(value);
        }
        pll->filt.setup(vdd,vdd,cap);
        f.close();
}

PLLSim::PLLSim()
{
        cntl_v=filt.dVolts;
        cntl_i=feedback=vco_out=ref=0.0;
        iIterations=1000;
        iStartOutput=0;
}

PLLSim::~PLLSim()
{

}

void PLLSim::stepRef()
{
```

```
	ref=pg.step();
}

void PLLSim::stepFeedback()
{
	vco_out=vco.step(cntl_v,cntl_i);
	feedback=feedback+div.step(vco_out);
}

void PLLSim::go(int num_steps,char *simfile,char* outfile)
{
	double ph=0.0;

	fstream f;
	f.open(outfile,ios::out);
	  f.precision(6);

	loadSimFile(simfile,this);

	//Print out Sim parameters to screen
	printf("\nPLL Sim Config\n");
	printf("---------------------------------------\n");
	pg.print();
	printf("---------------------------------------\n");
	div.print();
	printf("---------------------------------------\n");
	vco.print();
	printf("---------------------------------------\n");
	filt.print();
	printf("---------------------------------------\n");
	cp.print();
	printf("---------------------------------------\n");


	for(int i=0;i<iIterations;i++)
	{
		stepRef();
		stepFeedback();
		cntl_i=0.0;
		if (i%div.div==0)
		{
			ph=pfd.step(this);
			cntl_i=cp.step(ph);
			cntl_v=filt.step(ph,cntl_i);
		}

		if (i>=iStartOutput) // Print desired outputs here
			f << i << " " << vco.dPeriod*2e9 << " " <<
ph*1e9 << "\n";
```

```
	}
}

Divider::Divider()
{
	div=4;
	e_last=0;
}

Divider::Divider(int d)
{
	div=d;
	e_last=0;
}

Divider::~Divider()
{

}

void Divider::setup(int d)
{
	div=d;
}

double Divider::step(double e)
{
	double r=div*(e-e_last);
	e_last=e;
	return r;
}

ChargePump::ChargePump()
{
}

ChargePump::~ChargePump()
{

}

void ChargePump::setup(char* filename)
{
	oI.load(filename);
}

double ChargePump::step(double p)
{
	return oI.getY(p);
```

```
}

Filter::Filter()
{
	dVolts=0.0;
	dCap=200e-12;
}

Filter::~Filter()
{

}

void Filter::setup(double vv,double v,double c)
{
	dVolts=v;
	dCap=c;
	vdd=vv;
}

double Filter::step(double p,double i)
{
	if (i<0)
		i=i*-1;
	dVolts=dVolts+p*i/dCap;
	if (dVolts>vdd)
		dVolts=vdd;
	if (dVolts<0)
		dVolts=0.0;
	return dVolts;
}

VCO::VCO()
{
	iCycle=0;
	dTime=0;
	dPeriod=0;
	dPeriodNom=0;
	dCurrentGain=350000;
}

VCO::~VCO()
{

}

void VCO::setup(char* filename)
{
	oRange.load(filename);
```

```
}

double VCO::step(double v,double i)
{
	dPeriodNom=1/oRange.getY(v);
	dPeriod=dPeriodNom-(i/dCurrentGain);
	dTime=dTime+dPeriod;
	iCycle++;
	return dTime;
}

PulseGen::PulseGen()
{
	dPeriod=1e-9;
	dTime=0;
	dDeltaPeriod=0;
	dDeltaPhase=0;
	dRate=200e-6;
	iCycle=0;
	bSSC=0;
}

PulseGen::~PulseGen()
{

}

void PulseGen::print()
{
	printf("Pulse Gen: period=%.6g ns; delta_period=%.6g ns;
delta_phase=%.6g ns; mod_rate=%.6g s\n",
		dPeriod*1e9,dDeltaPeriod*1e9,dDeltaPhase*1e9,dRate);
	if (bSSC)
	{
		printf("Modulation Table:\n");
		oSSC.print();
	}
	else
		printf("No Modulation Table defined.\n");
}
void PulseGen::setup(double period)
{
	dPeriod=period;
	dPeriodNom=period;
}

void PulseGen::doModulation(char* modfile)
{
	oSSC.load(modfile);
```

```
	bSSC=1;
}

double PulseGen::step()
{
	  if (bSSC)
	  {
	      double xmax=oSSC.xx[oSSC.size-1];
	      double xnorm=dTime-int(dTime/xmax)*xmax;
	      dPeriod=dPeriodNom*oSSC.getY(xnorm);
	}
	if (dTime>dRate && dDeltaPeriod!=0)
	{
		dPeriod=dPeriod+dDeltaPeriod;
		dDeltaPeriod=0;
	}

	if (dTime>dRate && dDeltaPhase!=0)
	{
		dTime=dTime+dDeltaPhase;
		dDeltaPhase=0;
	}
	iCycle++;
	dTime=dTime+dPeriod;
	return dTime;
}

PFD::PFD()
{
	e1=e2=0;
}

PFD::~PFD()
{

}

double PFD::step(PLLSim* pll)
{
	double ph=pll->feedback-pll->ref;  //positive means ref is
leading
	int s=0;
	if (ph>=0)
	{
		s=1;
		while (pll->ref<=e2)
		{
			pll->stepRef();
		}
```

```
            e2=pll->feedback;
            e1=pll->ref;
      }
      else
      {
            s=-1;
            while (pll->feedback<=e2)
            {
                  pll->stepFeedback();
            }
            e1=pll->feedback;
            e2=pll->ref;
      }
      ph=s*(e2-e1);
      return ph;
}

void InterpolateArray::print()
{
      for (int a=0;a<size;a++)
            printf("%10d  %12.6g %12.6g\n",a,xx[a],yy[a]);

}

double InterpolateArray::getY(double x)
{
      int s=1;
      if (x<0.0)
      {
            s=-1;
            x=-x;
      }

      int i=-1;
      if (x<=xx[0])
            i=0;
      if (x>=xx[size-1])
            i=size-2;
      if (i==-1)
      {
            i++;
            while (x>xx[i])
                  i++;
            i--;
      }
      double m=(yy[i]-yy[i+1])/(xx[i]-xx[i+1]);
      return s*(m*(x-xx[i])+yy[i]);
}
void InterpolateArray::load(char* filename)
```

```
{
	fstream f;
	f.open(filename,ios::in|ios::nocreate);
	char buf[255];
	char* out;
	char seps[]=" \t\n";

	size=0;
	while (f.good())
	{
		f.getline(buf,254);
		out = strtok( buf, seps );
		if (out!=NULL)
			size++;
	}
	f.close();
	xx=new double[size];
	yy=new double[size];

	f.open(filename,ios::in|ios::nocreate);

	int i=0;
	while (f.good())
	{
		f.getline(buf,254);
		out = strtok( buf, seps );
		if (out==NULL)
			continue;
		xx[i]=atof(out);
		out=strtok( NULL, seps );
		if (out==NULL)
		{
			yy[i]=0.0;
			continue;
		}
		else
			yy[i]=atof(out);
		i++;
	}
	f.close();
}
```

Appendix B – Example Main Input File

[pll.txt]

```
vco_file=vco.txt
cp_file=cp.txt
ref_period=10.38e-9
div_ratio=2
vdd=1.8
cap=100e-12
vco_current_gain=132000
sim_cycles=25000
modulation_rate=100e-6
#modulation_file=ssc.txt
delta_period=0.5e-9
#delta_phase=0.02e-9
start_output=0
```

Appendix C – Example Input Definition Files

[cp.txt – charge current definition]

```
0.0     0e-6
50e-12  4e-6
100e-12 30e-6
150e-12 60e-6
200e-12 60e-6
300e-12 60e-6
```

[vco.txt – VCO definition]

```
0 10e6
.4  25e6
1.4 450e6
1.8 500e6
```

[ssc.txt – spread spectrum clock definition]

```
0       0.95
3.8e-6  0.974
6.47e-6 0.99
8.53e-6 1
11.76e-6 1.005
15.16e-6 1.02
20e-6   1.05
20.5e-6 1.052
21e-6   1.05
25.84e-6 1.02
29.24e-6 1.005
32.47e-6 1
34.53e-6 0.99
37.2e-6  0.974
41e-6  0.95
41.5e-6   0.948
```

References

[1] E. L. Acuna, J.P. Dervenis, A.J. Pagones, F. L. Yang, and R.A. Saleh, "Simulation Techniques for Mixed Analog/Digital Circuits," IEEE J. of Solid-State Circuits, vol. 25, no. 2, Apr. 1990, pp. 353-362.

[2] H. S. Yap, "Designing to Digital Wireless Specifications Using Circuit Envelope Simulation," HP EESof Application Note, 1995.

[3] W. E. Thain and J. A. Connelly, "Simulation and Modeling – an Improved VCO Macromodel," IEEE Circuits and Devices Magazine, vol. 84, July 1992, pp. 8-16.

[4] K. W. Current, J. F. Parker, and W. J. Hardaker, "On Behavioral Modeling of Analog and Mixed-Signal Circuits", Signals, Systems, and Computers, vol. 1, 1994, pp. 264-268.

[5] A. Albiol, N. Cardona, and J. Marti, "An Interactive Program for PLL Simulations," Frontiers in Education, 1992, p. 835.

[6] J. P. Hein and J. W. Scott, "z-Domain Model for Discrete-Time PLL's," IEEE Transactions on Circuits and Systems, vol. 35, no. 3, Nov. 1988, pp. 1393-1397.

[7] A. Phanse, R. Shirani, R. Rasmussen, R. Mendal, J. S. Yuan, "Behavioral modeling of a phase locked loop," Southcon/96, 1996, pp. 400-404.

[8] M. Takahashi, K. Ogawa, and K. S. Kundert, "VCO Jitter Simulation and it Comparison with Measurement," Proc. ACM/IEEE Design Automation Conference, 1999, vol. 1, pp. 85-88.

[9] D. Armaroli, V. Liberali, and C. Vacchi, "Behavioral Analysis of Charge-Pump PLL's," Proc. IEEE Circuits and Systems 1995, vol 2., 1996, pp. 893-896.

[10] B. A. A. Antao, F. M. El-Turky, R. H. Leonowich, "Mixed-Mode Simulation of Phase-Locked Loops," Proc. IEEE Custom Integrated Circuits Conference, 1993, pp. 8.4.1-8.4.4.

[11] P. Larssom, “A Simulator Core for Charge-Pump PLLs,” IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing, vol. 45, no. 9, Sept 1998, pp. 1323-1326.

[12] F. M. Gardner, “Charge-Pump Phase-Lock Loops,” IEEE Transactions on Communications, vol. COM-28, no. 11, Nov. 1980, pp. 1849-1855.

[13] D. W. Boerstler, “High Frequency Phase-Lock Loop Circuit Having Reduced Jitter,” U.S. Patent, no. 5870003, Feb 9, 1999.

[14] M. T. Zhang, “Notes on SSC and Its Timing Impacts,” Intel Application Note, rev. 1.0, Feb. 1998.

[15] R. E. Best, Phase-Locked Loops Theory, Design, and Applications, 2d ed., McGraw-Hill, Inc., 1993.

[16] G. T. Roh, Y. H. Lee, and B. Kim, “Optimum Phase-Acquistion Technique for Charge-Pump PLL,” IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing, vol. 49, Sept. 1997, pp. 729-740.

Vita

Norman Karl James was born in Dallas, Texas on October 12, 1972, to parents Karl and Eugenia James. He attended primary school in Winnsboro, Texas, and was awarded a Bachelor's of Science in Electrical Engineering at Texas A&M University in College Station, Texas. His primary focus in school was analog circuit design, which led him to a career at IBM doing PLL design for microprocessors. He is currently pursuing a Master's of Science in Engineering at the University of Texas in Austin, Texas.

Permanent address: 16824 Village Oak Loop
Austin, TX 78717

This report was typed by Norman K. James.