Phase Noise Simulation and Modeling of ADPLL by SystemVerilog

Tingjun Wen

Integrated Device Technology, Canada
Ottawa, Ontario, Canada K2K 3K2
tingjun.wen@idt.com

Tad Kwasniewski

Department of Electronics, Carleton University
Ottawa, Ontario, Canada K1S 5B6
tak@doe.carleton.ca

***Abstract*—Event driven phase noise simulation and modeling of an ADPLL by SystemVerilog is presented in this paper. It uses the simple Stochastic Voss-McCartney algorithm to generate the pink noise so that the $1/f$ phase noise effect can be easily modeled. Since the event driven simulation is extremely fast compared to the circuit level simulation, it allows circuit designers to explore different ADPLL architectures at the early stage without going through the time-consuming circuit level simulation. Pure SystemVerilog implementation also makes it possible to simulate the phase noise effect of the ADPLL efficiently in a large SOC system.**

I. Introduction

It is a well known issue that the phase locked loop (PLL) simulation by Spice-like simulator is time consuming. In order to predict the noise in a circuit, Spice needs a quiescent operating point which is not always the case in a PLL circuit [1]. This means that it is impractical to use Spice simulator to explore the PLL architecture due to its simulation speed and sometimes it is even impossible due to its inability to simulate the noise in a complex system. In order to solve the speed problem of the Spice simulation, phase-domain simulation technique was used. However, the phase domain simulation is only useful for predicting the loop dynamics and gives little intuitive insight into the circuit itself especially for the beginners. Moreover, the phase domain simulation can not be used to evaluate the phase noise performance which is the most important feature in a PLL system.

With the increased interest in the all digital phase locked loop (ADPLL), researchers began to simulate the ADPLL by the event driven technique. Reference [2] explores Matlab's event driven programming technique to simulate the digital PLL at the system level. Because the Matlab simulation code uses the event driven functions, the simulation speed is fast. It is easy to add the phase noise simulation in the Matlab code. However, it is not an easy task to integrate the Matlab simulation into a system-on-a-chip (SOC) simulation environment. Reference [1] is a comprehensive tutorial paper that uses the mixed signal simulator VerilogA to simulate a charge pump PLL (CPPLL). The best part of this paper is its jitter and phase noise simulation technique. Compared to the Matlab or other general purpose programming languages such as C, the hardware description languages Verilog or VHDL are intrinsically event driven and programmers do not need to worry about how event driven functions work. Adding a $\Sigma\Delta$ modulator into the simulation code will make it possible to study the $\Sigma\Delta$ modulator phase noise effect. Reference [3] does this by a different mixed signal simulator SpectreVerilog from Cadence. With the advance of the CMOS process, more and more circuit components can be integrated into a single chip and most SOC designs have one or more PLL blocks. From the architectural point of view, it is desirable to use as many as possible digital parts in the PLL design. This results in the proliferation of the ADPLL in the SOC applications. Due to ADPLL's digital nature, it is very easy to incorporate the ADPLL simulation into the SOC simulation environment. It is a difficult task to effectively simulate the phase noise in a pure digital environment without slowing down the whole simulation because phase noise simulation requires complex noise generation algorithms. Reference [4] simulates the phase noise within the SOC environment with the pure digital VHDL language.

The key component in the phase noise simulation is the time domain noise generator. Once the noise in the digital controlled oscillator (DCO) is generated correctly, the closed loop dynamics with the noise effect can be simulated. The most important noises in a DCO is the $-20dB/dec$ frequency modulated thermal noise, the $-10dB/dec$ flicker noise, and the flat white Gaussian noise. Noises generated in blocks that are not part of the PLL loop can be modeled as the additive white noise and they are not cumulative so they can be inferred into the DCO's output white noise.

This paper first briefly explains a new ADPLL architecture that the authors are currently working on in section II. SystemVerilog is used to model this ADPLL building blocks since SystemVerilog's direct programing interface (DPI) makes it extremely easy to call external noise generation functions written in C. The noise generation algorithms are thoroughly described in section III. The highly respected Mersenne-Twister random number generator [5] is used to generate the white noise with uniform distribution. The white noise with Gaussian distribution is then generated from this uniform white noise by the Box-Muller algorithm. The 1/f noise is generated by a simple Stochastic Voss-McCartney algorithm [6] from the music DSP cummunity. The higher order noises are generated by integration over the previously mentioned

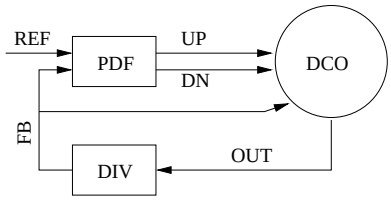

Fig. 1. ADPLL

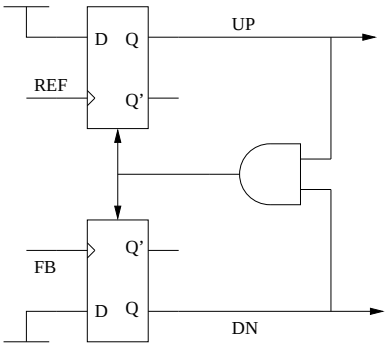

Fig. 2. PDF

lower order noise sources. The phase noise simulation results of the new ADPLL architecture are presented in section IV. Finally, this paper is concluded by section V with comparisons and discussions of the simulation results.

II. ADPLL Architecture

The block diagram of the ADPLL under consideration is depicted in Fig. 1. The low speed reference signal REF is fed into the phase frequency detector (PFD). The PFD compares this REF with the feedback signal FB from the frequency divider (DIV) and produces two signals UP and DN. The UP and DN signals are then fed to the digitally controlled oscillator DCO. Let us denote the intrinsic output period of the DCO as T_0 when both UP and DN are tied to ground. The instantaneous output period of the DCO T is then determined by T_0 and the state value stored in the varactor array. The high speed output signal OUT with period T is divided down to FB with the period of $T * N$ by the frequency divider DIV where N is the divider ratio.

Compared to most of the traditional PLL configurations, this ADPLL does not have an explicit low pass filter LPF. The LPF function in this ADPLL is tightly integrated inside the DCO. The DCO frequency is controlled by an array of switched capacitors which can also be used to perform the addition operations. Integrating the LPF function inside the DCO has the benefit of eliminating the complex adders needed by the traditional LPF module. The detailed configuration of the DCO and its operation will be described in section II-C.

A. Phase frequency detector

The phase and frequency detector used in this simulation is the traditional 3-state PFD as depicted in Fig. 2. When the reference clock REF is ahead of the divide down signal FB, the UP output is high; otherwise, the DN signal is high. When the reference clock REF and the divide down signal FB have the same frequency and they are phase synchronized, both UP and DN are low. Due to the AND gate delay, there is a very short time when both UP and DN are high. Listing 1 shows the PFD SystemVerilog code. The file adpll.vh contains the design parameters and noise parameters. They can be found in Listing 4 and Listing 5.

Listing 1. PFD

```
1  `include "adpll.vh"
2  module pfd (fref, fdiv, up, dn, rst_n);
3    output up, dn;
4    input fref, fdiv, rst_n;
5    reg up, dn;
6    wire pfd_rst;
7
8    // fref faster => up
9    always @(posedge fref or posedge pfd_rst
         ) begin
10     if (pfd_rst) up <= 0; else up <= 1;
11   end
12
13   // fdiv faster => dn
14   always @(posedge fdiv or posedge pfd_rst
         ) begin
15     if (pfd_rst) dn <= 0; else dn <= 1;
16   end
17
18   assign pfd_rst = ~rst_n | (up & dn);
19 endmodule
```

B. Frequency divider

The frequency divider in a real circuit implementation may be the dual modulus divider. However, for the behavioral simulation purpose, we simply use a counter. Initially, the counter is reset to 0. Whenever the counter is greater than or equal to the programmed divider ratio, the output is flipped and the the counter is reset to 0 again. Listing 2 shows the frequency divider SystemVerilog code.

Listing 2. FDIV

```
1  `include "adpll.vh"
2  module fdiv (clk, M, fdiv, reset);
3    parameter div_width = 8;
4    input clk, reset;
5    input [div_width-1:0] M;
6    output fdiv;
7    wire fdiv;
8    reg q;
9    integer i;
10
11   always @(negedge reset) begin
12     i = 0;
13     q = 0;
14   end
15
16   always @(clk) begin
17     if (~reset) begin
18       i = i + 1;
19       if (i >= M) begin
20         q = ~q;
21         i = 0;
```

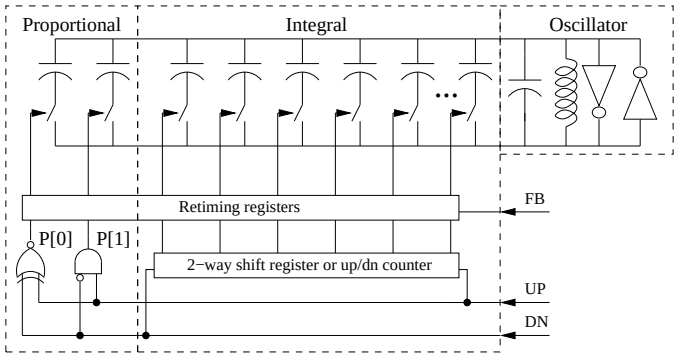

Fig. 3. DCO

```
22          end
23        end
24      end
25
26      assign fdiv = q & ~reset;
27  endmodule
```

C. Digitally controlled oscillator

Traditionally, a standalone LPF is needed on the DCO control input. Complex adders and multipliers are needed in this type of LPF. If two capacitors are connected in parallel, the total capacitance is simply the addition of two individual capacitances. This property makes it possible to use the varactor array itself to perform the addition operation. The purpose of the multiplier is to control the ratio of the proportional coefficient and the integral coefficient. If we use extra control signals to control how many varactor units are used for the proportional path and how many varactor units are used for the integral path, we do not need multipliers in the LPF either.

Fig. 3 shows the proposed DCO circuit block with the integrated LPF. The oscillator sub-module defines the intrisic frequency T_0. The capacitor in the oscillator sub-module also includes the lump-sum off-values of varactors in the proportional and integral paths. The input control signals are directly connected to the PFD output UP and DN.

The proportional path simply includes an XNOR gate and an AND gate. The truth table for the proportional path is shown in Table I. The last column of this table shows that when UP and DN are both 0s or 1s, the proportional path value (P-1) remains 0; when UP is 0 and DN is 1, the proportional path value (P-1) is -1. Minus 1 can be considered as part of the integral path capacitance or as the parasitic capacitance of the oscillator loop. This means that the simple circuit does perform the proportional path functionality.

The integral path can be implemented as either a 2-way shift register or an up/down counter. There are three different methods to realize the integral path: (A) If it is implemented as an up/down counter, then the varactors need to be sized as the binary encoding $1C_0, 2C_0, 4C_0, 8C_0, ...$ where C_0 is the minimal achievable capacitance with the minimal feature size. (B) If it is implemented as a 2-way shift register with its lower

TABLE I
PROPORTIONAL PATH TRUTH TABLE

UP	DN	P[1]	P[0]	P-1
0	0	0	1	0
0	1	0	0	-1
1	0	1	0	1
1	1	0	1	0

TABLE II
INTEGRAL PATH CODING METHODS AND EXAMPLES

	Varactor Coding	Ctrl Logic	Ctrl Word	Varactor Change	Granularity
A	Binary	up/dn counter	01010	$10C_0$	Coarse
B	Unary	2-way shift reg	01111	$4C_0$	Medium
C	1-hot	2-way shift reg	01000	$3C_\Delta$	Fine

significant bits as high, then the varators can be sized equally as C_0. (C) If it is implemented as a 2-way shift register with only one bit high at any time (1-hot coding), then the varactors need to be sized as $C_0+0C_\Delta, C_0+1C_\Delta, C_0+2C_\Delta, ...$ where C_Δ is the minimal achievable capacitance increment which is smaller than C_0. Method C was employed in [7] to achieve high frequency resolution of the DCO. These methods and their examples are summarized in table II.

Method (A) is simple and compact but may not have good matching property. Method (B) has very regular circuit structure and its matching property can be very good. Method (C) can have very smaller increment C_Δ than C_0 of a minimal feature sized varactor. The actual circuit implementation and its encoding will depend on the minimal capacitance achievable by the technology used and the ADPLL requirements. Three different methods can also be combined together to do the coarse tuning, medium tuning, and fine tuning to shorten the settling time and to lower the phase noise.

Listing 3 shows the DCO SystemVerilog code. Only the up/down counter method is shown in this listing for simplicity purpose. The noise generation related code will be explained in section III.

Listing 3. DCO

```
1   `include "adpll.vh"
2   module bbdco (up, dn, fout, fref, fdiv);
3     input up, dn;
4     output fout;
5     reg fout;
6     input fref, fdiv;
7     reg [1:0] P;
8     integer I;
9     real Ctrl, period;
10    real jitter_stddev, flicker_stddev,
          wander_stddev, saunter_stddev;
11    real pjitter,       pflicker,
          pwander,        psaunter;
12
13    initial begin
14      fout = 1'b1;
15      P = 0;
16      I = 0;
```

```
17      Ctrl = 0.0;
18      pjitter  = 0.0;
19      pflicker = 0.0;
20      pwander  = 0.0;
21      psaunter = 0.0;
22      jitter_stddev  = white2stddev(`f0dco,
                        `white_pn);
23      flicker_stddev = pink2stddev (`f0dco,
            `pink_corner, `pink_pn);
24      wander_stddev  = brown2stddev(`f0dco,
            `brown_corner,`brown_pn);
25      saunter_stddev = red2stddev  (`f0dco,
            `red_corner,  `red_pn);
26    end
27
28    always @(up or dn) begin : lpf
29      case ({up, dn})
30        2'b01: I = I - 1;
31        2'b10: I = I + 1;
32        default: ;
33      endcase
34      P[0] = up ^~ dn;
35      P[1] = up & ~dn;
36    end
37
38    always @(posedge fdiv) begin : retiming
39      Ctrl <= (`Kp * P) + (`Ki * I);
40    end
41
42    always begin
43      period = 1.0 / (`f0dco + `Kdco * Ctrl)
            ;
44      period = period + jitter (
            jitter_stddev,  pjitter);
45      period = period + flicker(
            flicker_stddev, pflicker);
46      period = period + wander (
            wander_stddev,  pwander);
47      period = period + saunter(
            saunter_stddev, psaunter);
48      if (period <= 0) $stop;
49      else begin
50        #(period/1e-12/2) fout = ~fout;
51        #(period/1e-12/2) fout = ~fout;
52      end
53    end
54  endmodule
```

III. Noise Simulation

The same noise terminologies from different publications can sometimes mean different types of noises. In order to have consistent noise terminologies, we use the terms white, pink, red, and infrared to designate the frequency domain phase noises with different slopes in this paper. White phase noise is flat; pink phase noise has $-10dB/dec$ slope; red phase noise has $-20dB/dec$ slop; infrared phase noise has $-30dB/dec$ slope. The corresponding time domain noise terms for the above phase noises are jitter, flicker, wander, and saunter respectively as shown in the parenthesis in Fig. 4.

Please note that the term infrared phase noise and the saunter are newly introduced in this paper to refer to the integrated pink noise and the flicker noise in the frequency domain and in the time domain respectively.

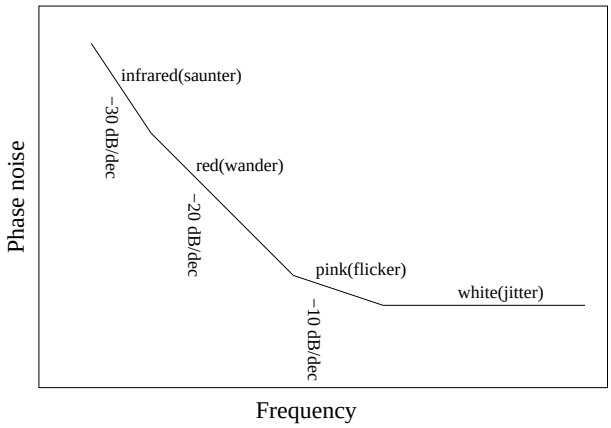

Fig. 4. Phase noise spectral density

Given the white phase noise $\mathcal{L}_w$ in dBc and the oscillator frequency f_0, the jitter noise σ_j can be calculated by the following equation [4]:

$$\sigma_j = \frac{1}{2\pi}\sqrt{\frac{10^{\mathcal{L}_w/10}}{f_0}} \tag{1}$$

Given the pink noise $\mathcal{L}_p$ in dBc and frequency offset Δf, the flicker noise σ_f can be calculated by the following empirical equation introduced in this paper:

$$\sigma_f = \frac{\Delta f}{f_0}\sqrt{\frac{10^{\mathcal{L}_p/10}}{2\pi f_0}} \tag{2}$$

Given the red noise $\mathcal{L}_r$ in dBc and frequency offset Δf, the wander noise σ_w can be calculated by the following equation [4]:

$$\sigma_w = \frac{\Delta f}{f_0}\sqrt{\frac{10^{\mathcal{L}_r/10}}{f_0}} \tag{3}$$

Given the infrared noise $\mathcal{L}_i$ in dBc and frequency offset Δf, the saunter noise σ_s can be calculated by the following empirical equation introduced in this paper:

$$\sigma_s = \frac{\Delta f}{2\pi^2 f_0}\sqrt{\frac{10^{\mathcal{L}_i/10}}{2\pi f_0}} \tag{4}$$

Noise simulation in the behavioral simulation involves the generation of the white, pink, red, and the infrared noise. The simplest noise is the white noise. Theoretically, the white noise can be generated by any uniform random number generator readily available from any software packages. However, a random uniform random number generator may not have the expected power spectral density value in the ADPLL simulation. The normal random number generator should be used in the simulation. In this paper, the normal random number generator is implemented by the Box-Muller algorithm and the uniform random number generator used by the Box-Muller algorithm is implemented by the highly respected Mersenne-Twister random number generator from [5].

Designers who are new to the noise simulation are often confused by the power spectral density and the random number probability distribution. Actually, these two properties do not have any direct relationship. Both the uniform random number generator and the normal random number generator can generate the time sequences that have flat spectral density. Noise with different power spectral density can be generated by integration or differentiation of the white noise. The integration and the differentiation serve the purpose of generating poles or zeros so that we can get the desired power spectral density graphs. This is called the filtering method or fitting method.

Like the white noise, the pink noise is also found to be universal in all kinds of applications such as electronics, economics, and music. However, unlike the white noise that is easy to generate, the flicker noise is the most difficult one to generate. It remains one of the most interesting topic of research in the art, science, and engineering fields. Reference [4] uses the filtering method to generate the pink noise. The filtering method is straight forward but its running simulation efficiency is not the best. Reference [6] proposed an improved Voss-McCartney algorithm in the music DSP community to generate the pink noise. This paper will use this improved Voss-McCartney algorithm in the ADPLL phase noise simulation.

The red and the infrared noise generators can be easily constructed with the white noise and the pink noise generators. The red noise generator is simply the integration of the white noise and the infrared noise generator is simply the integration of the pink noise.

The noise generation functions are written in C language as the SystemVerilog direct programming interface (DPI). DPI allows SystemVerilog code to call any user defined functions or operating system provided C libraries so it is extremely easy to integrate any external noise generators written in C into the simulation model written in RTL.

IV. Simulation Results

The SystemVerilog test bench writes the time domain periods into a file. After the simulation is done, a Matlab script is used to plot the time domain periods graph and to calculate the phase noise. The Matlab script is partially based on the script provided by [1]. The difference is that the Matlab script use by [1] does the noise integration by the Matlab and the Matlab script used by this paper does not do the noise integration and it is done by the PLL simulation itself. The noise integration is the transfer function of the DCO so it should be handled by the simulation. The phase noise is calculated by the Welch algorithm available in Matlab with the input period time sequence.

Listing 4. ADPLL design parameters

```
1 `define f0ref      24.0e6
2 `define f0dco      2200e6
3 `define Kdco       100e3
4 `define Kp         10.0
5 `define Ki         1.0
6 `define div_ratio  100
```

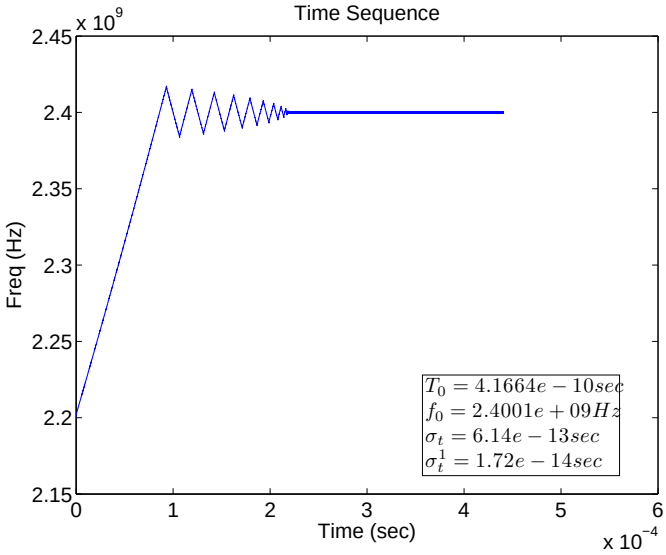

Fig. 5. Frequency response without noise

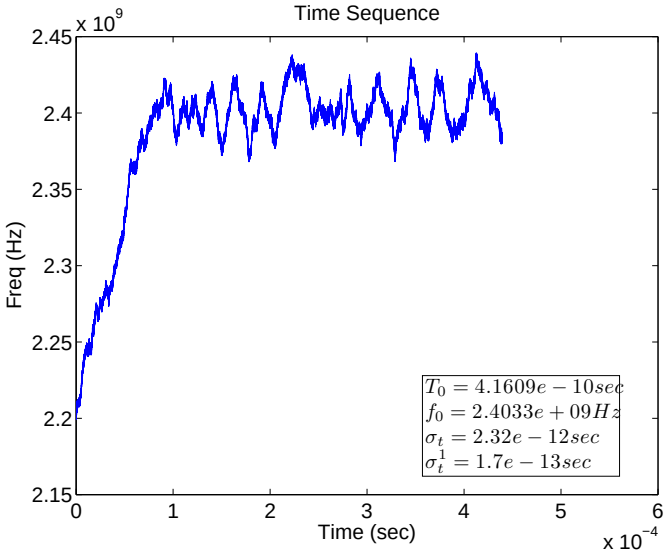

Fig. 6. Frequency response with noise

Listing 5. ADPLL noise parameters

```
1 `define ref_white_pn    -130
2 `define white_pn        -150
3 `define pink_corner      1e8
4 `define pink_pn         -150
5 `define red_corner       1e7
6 `define red_pn          -140
7 `define infrared_corner  1e6
8 `define infrared_pn     -120
```

Fig. 5 shows the time domain step response without any noise with the design parameters in Listing 4. The locking and the settling behavior can be clearly seen from this diagram. With the realistic noise parameters from Listing 5, however, the locking in and the settling behavior is not so obvious from the step response diagram as show in Fig. 6. The time domain

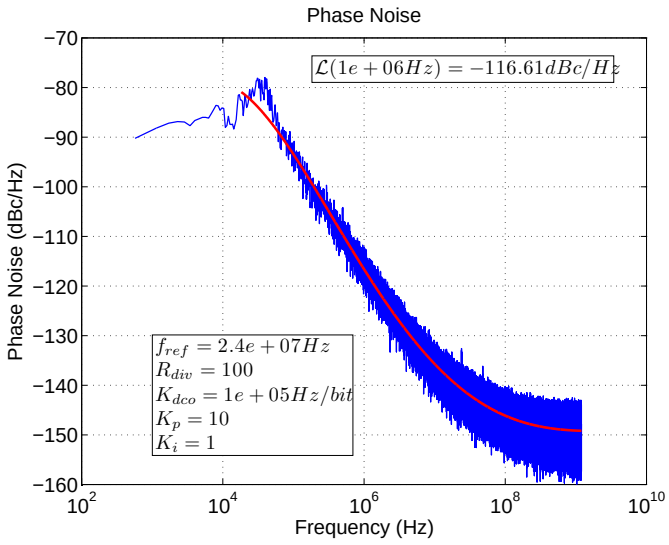

Fig. 7. Phase noise

noise σ_t can be calculated from the period time sequence used to draw the step response diagram. However, this time domain noise increases with the the simulation time if the phase noise is not pure white. A better measure of the time domain noise will be the differentiated σ_t^1 which is calculated by $\partial\sigma_t/\partial t$.

Fig. 7 shows the phase noise of the simulated ADPLL architecture. The phase noise values and shape are close to the calculated ones. For example, the white phase noise caused by peripheral circuits is expected to be $-150dBc/Hz$ and phase noise at $10MHz$ is expected to be $-120dBc/Hz + 3dBc = -117dBc/Hz$ mainly determined by the red noise corner and the infrared noise corner and the simulated result is $-116.6dBc/Hz$. We can see from this diagram that the ADPLL effectively suppresses the close-in phase noise. The loop bandwidth is estimated to be $30kHz$ from this diagram.

The program takes about 5 seconds to finish a 436 μs simulation without any noise and it takes 46 seconds to finish the same simulation with all four different noises on a Sun-Fire-V440 machine. This amounts to about $10\mu s$ simulation time per second.

V. Conclusion

Table III summaries different PLL simulation techniques described in the references. The simulation languages used in the references include Matlab, VerilogA, and VHDL. This paper chose SystemVerilog because the DPI interface provided by SystemVerilog makes it extremely simple to integrate the noise generators written in C. The noise performance is one of the most important parameters in any PLL design. Reference [2] does not include the noise simulation. Reference [1] and [3] only include the white noise and the red noise simulation. Reference [4] and this paper both include the $1/f$ noise simulation but with different algorithms. Reference [4] uses the IIR filter method to generate the $1/f$ noise while this paper uses the simple and efficient Voss-McCartney algorithm to generate the $1/f$ noise. This paper also introduces a new infrared noise (saunter in the time domain) in the simulation to model the $-30dB/dec$ effect in addition to the white, pink, and red noises. Because not all the papers include all four noise sources and the simulation environments are different, the simulation speed values listed in table III are for references only.

TABLE III
PLL behavioral simulation comparison

	Language	**Noise**	$1/f$ **Noise**	**Speed**
[1]	VerilogA	Yes	No	N/A
[2]	Matlab	No	No	N/A
[3]	VerilogA	Yes	No	$7\mu S/Sec$
[4]	VHDL	Yes	IIR Filters	$15\mu S/Sec$
This work	SystemVerilog	Yes	Voss-McCartney	$10\mu S/Sec$

Note: The speed values are for references only due to machine difference

The pure digital language SystemVerilog is successfully used in this paper to simulate the step response and the locking behavior in the ADPLL. Since SystemVerilog is intrinsically event-driven, the simulation speed is fast enough to allow a SOC design to include the detailed ADPLL simulation.

The noise generation algorithms in C language are used in the SystemVerilog model by DPI interface. The algorithms include the Mersenne-Twister random number generator, Box-Muller algorithm, and the Voss-McCartney algorithm. The noise simulation techniques can be used to investigate the phase noise behavior with any ADPLL system architecture at the very early design stage.

The proposed ADPLL architecture with a unique proportional path and a simplified integral path integrated in the DCO is verified to have good phase noise performance. These design parameters obtained from the simulation will be used in the transistor level simulation and layout.

References

[1] K. Kundert. (2006, Aug.) Predicting the phase noise and jitter of pll-based frequency synthesizers. [Online]. Available: http://www.designers-guide.org/Analysis/PLLnoise+jitter.pdf

[2] J. Zhuang, Q. Du, and T. Kwasniewski, "Event-driven modeling and simulation of an digital PLL," *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, pp. 67–72, Sept. 2006.

[3] S. Huang, H. Ma, and Z. Wang, "Modeling and simulation to the design of $\Sigma\Delta$ fractional-n frequency synthesizer," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*. San Jose, CA, USA: EDA Consortium, 2007, pp. 291–296.

[4] R. Staszewski, C. Fernando, and P. Balsara, "Event-driven simulation and modeling of phase noise of an rf oscillator," *Circuits and Systems I: Regular Papers, IEEE Transactions on [Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]*, vol. 52, no. 4, pp. 723–733, April 2005.

[5] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[6] L. Tammell. (2006, Jan.) Improvements in the correlated pink noise generator evaluation. [Online]. Available: http://home.earthlink.net/%7Eltrammell/tech/newpink.htm

[7] J. Zhuang, Q. Du, and T. Kwasniewski, "A 3.3 ghz lc-based digitally controlled oscillator with 5khz frequency resolution," *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian*, pp. 428–431, Nov. 2007.