

Functional Verification in the Presence of Linear Analog Circuits

Thomas J. Sheffler
Rambus Inc.
Los Altos, CA
sheffler@rambus.com,

Abstract— Mixed-signal architectures, in which digital control is used to adjust the electrical characteristics of analog circuits, are an important part of modern PHY designs. Verification of these types of mixed-signal systems remains difficult, partly because they are not easily modeled in the languages used for digital system interchange. This paper illustrates this point by exploring the microarchitecture of a PHY output driver and proposes an effective modeling and verification technique to ensure its correctness.

I. INTRODUCTION

Increasingly, logic designs are responsible for the control and regulation of non-digital circuits. This is especially true in the PHY arena. Whereas 10 years ago, a Rambus PHY had one configuration bus and one programmable current source, today's XDR PHY has over 500 registers and literally dozens of programmable DACs adjusting on-chip voltages, currents and impedances. Over the same 10 years, verification technology has evolved considerably in its expressiveness and ability to scale with increasing complexity. However, little literature has been presented regarding the verification of PHY designs until last year [1,2], and the verification challenges presented by these types of designs persist.

This paper updates our problem statement of last year by presenting a case study of the verification issues surrounding a differential current-steering output driver with equalization. The driver is a carefully designed high-speed component comprising logic and circuits. Integration of this component into the full PHY design requires dissecting it into primary sub-components and reassembling them in the final PHY. It is at this step that problems occur, and it is common for digital control of analog blocks to become inverted, transposed, or otherwise corrupted.

For this circuit, we show an appropriate mathematical formula specifying its equalization behavior in the voltage domain. We propose an appropriate modeling abstraction for use in this type of integration verification. This model updates the work of [3] by extending Verilog into the voltage and current domain through a PLI application. The PLI

application allows the modeling of variable current sources (\$csrc), variable voltage sources (\$vsrc), variable resistors (\$resistor) and one non-linear element, a digitally controlled switch (\$switch). We then put the pieces together, using a Constrained-Random Testing (CRT) pseudo-random testbench to verify the correct integration of the transmitter in the context of an industrial-sized PHY design. We show how integration errors are caught by this technique, and also present performance data on the runtime overhead of our PLI application and models.

II. DIFFERENTIAL CURRENT-STEERING OUTPUT DRIVER WITH EQUALIZATION

Equalization (or pre-emphasis) is a technique used in an output driver to compensate for data losses and reflections on a noisy channel [4]. Such reflections cause interference between adjacent bit-times, which is called inter-symbol interference (ISI). In the frequency domain, equalization reduces the low-frequency components of the transmitted signal as appropriate for the transmission medium.

A Differential Current-Steering Output Driver with Equalization (DCSODwE) modulates the magnitudes of the bits transmitted on the channel to manage the frequency components of the signal. Rather than transmit one of only two voltage levels, representing a logical "0" and a logical "1", the DCSODwE implements a recurrence equation, where the voltage value produced at each bit time is a weighted sum of the data bits *before* (and possibly *after*) the current bit, x_i . (Bit x_i is called the cursor.) Equation 1 defines such a recurrence.

$$\text{(Eq 1)} \quad V_i = a_0 x_i + a_1 x_{i-1} + a_2 x_{i-2}$$

In use, the coefficients a_0 , a_1 and a_2 are chosen to minimize the effective ISI in the channel environment. These coefficients are determined by simulation, by lab experiments, or possibly by a calibration procedure. At the microarchitecture level, the design and implementation of such a driver involves assembling digital and analog components to implement this recurrence. The question of how to ensure that this assembly is correct in the context of its system is the subject of this paper.

A. Differential Current-Steering Output Driver with Programmable Swing

The output driver with equalization builds on a simpler structure: a differential output driver with programmable swing. This section describes the simple driver. A following section uses the components of the simple driver to assemble the output driver with equalization.

The architecture of a non-equalizing differential current-steering output driver (DCSOD) (including its termination components) is shown in Figure 1. The current source produces a current called i_{SINK} that passes through one of two switches. The bit stream (value "bit") and its complement (value "!bit") open and close the switches, which are implemented as appropriately biased transistors. The termination voltage, v_{TERM} and the termination pullup resistors reside outside of the driver, on the far end of the channel. Because the current is always flowing through one of the legs, this architecture is called "current-steering."

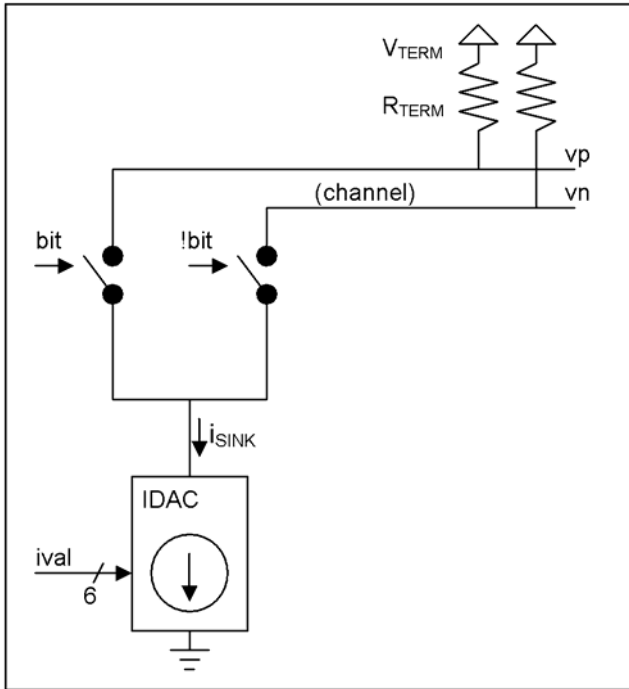


Figure 1. Differential Output Driver

When the driver is transmitting a logical "1", the switch controlled by "bit" is closed and v_p is pulled down to

$$V_{\text{LOW}} = V_{\text{TERM}} - (R_{\text{TERM}} * i_{\text{SINK}}).$$

The switch controlled by the complement of "bit" remains open, and v_n floats high to

$$V_{\text{HIGH}} = V_{\text{TERM}}.$$

Conversely, if the driver is transmitting a logical "0", then v_n is pulled down and v_p floats high.

(In use, the signals v_p and v_n are routed as differential signals to provide noise immunity in the electrical environment in which the PHY is operating.)

Many modern PHYs provide a means to tune the magnitude of the current, i_{SINK} . To do this digitally, a component called an IDAC is used. The IDAC is a form of digital-to-analog converter that produces a current determined by a digital control value. The transistor-level architecture of an IDAC is shown in Figure 2.

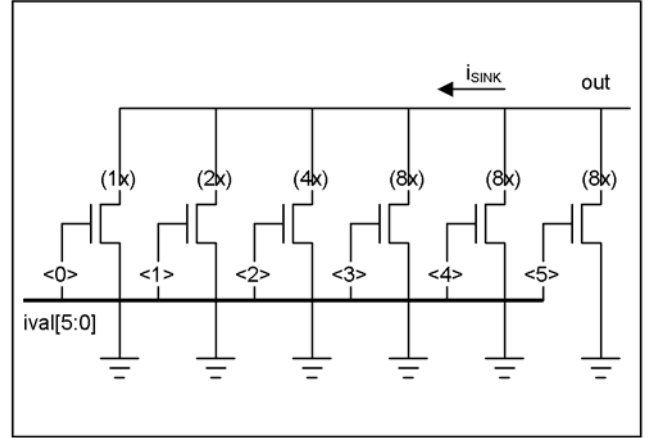


Figure 2. Simplified Schematic of an IDAC

To implement programmable current, the sizes of the transistors are modulated in powers-of-two. Thus the leg selected by "idac[0]" sinks one unit of current, and the leg selected by "idac[3]" sinks eight units of current. The high-order legs break the power-of-two pattern and are selected in an additive fashion to sink the desired current. The encoding of the high-order bits is called "thermometer" encoding, and the fact that the low-order bits are binary makes this a "segmented" DAC architecture. The circuit design of the overall system ensures that the transistors of the IDAC are operating in the saturated region.

B. Output Driver with Equalization

The addition of the equalization capability adds multiple stages to the output driver. The architecture of a three-stage DCSODwE (Differential Current-Steering Output Driver with Equalization) is shown in Figure 3. Here, banks of the simple output driver are ganged together, each contributing to the pulldown swing of the differential outputs. The flip-flops are used to produce the bit stream x_i , x_{i-1} and x_{i-2} ; these are normal digital components. The current source magnitude of each stage is programmable through the values of $ival_0$, $ival_1$ and $ival_2$. This architecture implements the following mixed-signal recurrences.

$$(Eq 2) \quad v_{p_i} = V_{\text{TERM}} - (ival_0 x_i + ival_1 x_{i-1} + ival_2 x_{i-2})$$

and

$$(Eq 3) \quad v_{n_i} = V_{\text{TERM}} - (ival_0 !x_i + ival_1 !x_{i-1} + ival_2 !x_{i-2})$$

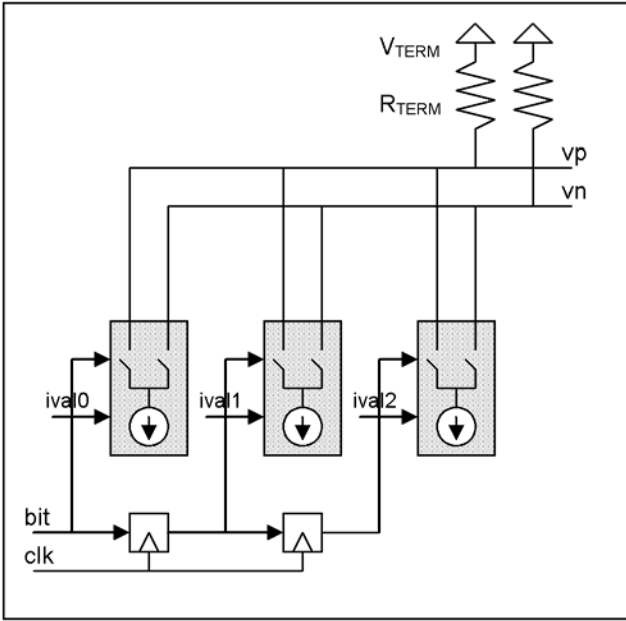


Figure 3. Three-Stage Output Driver with Equalization

These two equations are the mathematical specifications of the implementation of the circuit in the voltage domain, for discrete time. The circuit implements the operation of addition through current summing and multiplication as a switch, combining Boolean and real values. The two equations represent the mixed-signal specification of the transfer function of the output driver.

C. Microarchitecture Verification in a System Context

In a normal design flow, the microarchitecture of the output driver, as shown in Figure 3, is defined early in the system design process. Digital and analog design then both begin concurrently. On the circuit side, the component blocks of the output driver (switches, IDACs, channel and termination) are analyzed using a Spice variant. The blocks (switches, IDACs, shift-register components) are dissected and reassembled in the context of the PHY system.

Concurrent with circuit design, digital design proceeds on the containing system. During this process, representative models for the components of the output driver are used. With regard to the example here, the output drivers are a small, but critical, part of the system. The sub-components of the output driver are divided into digital and analog parts, and are buried deep in the hierarchy of the design. Figure 4 shows how the output driver components fit into such a system.

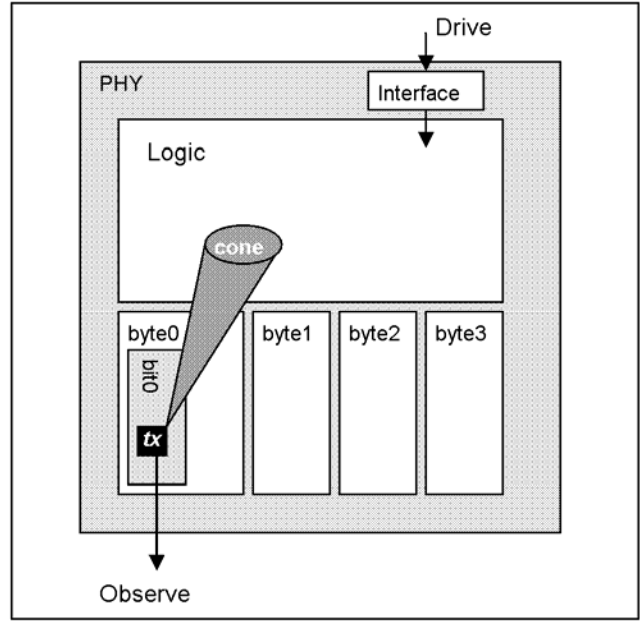


Figure 4. The TX Block in a System Context

Even though the analog components represent only a small part of the system, they are the key elements, and the verification process must ensure that the logic cone driving these components is correct. We desire representative models complete enough to verify that the effect of logic driving the output drivers can be observed at the pins. These representative models should also be lightweight enough so that computation time is not spent computing details not necessary for the system verification task.

By verifying that the implementation matches the specification in the context of its system, we can gain high confidence that the components of the microarchitecture are assembled correctly and that the digital logic driving the output driver is providing the correct voltage value at each bit time.

III. COMMON VERIFICATION PROBLEMS

It is important that behavioral models for analog blocks of the type of the IDAC reflect the effect of their inputs on the value of their outputs. In the case of the IDAC, when the magnitude of the digital input is not reflected in the output of the block, a number of errors can be missed by functional verification. These can include the following mistakes.

- **Bus Swaps:** A control bus intended for one IDAC delivered to another one.
- **Inversion Errors:** Some control bits are inverted. This happens when additional buffering is introduced after signal integrity analysis shows it is needed and the resulting logic is not appropriately verified.

- Encoding Errors: This problem is characterized by the logic designer assuming one encoding for the control bus (unsigned integer, for example) and the circuit designer assuming something else (perhaps a thermometer encoding).
- Overflow, Underflow: At the boundary of the IDAC range, the digital counter may overflow or underflow. If the counter is supposed to saturate instead, this error can be difficult to detect.

We have found that using plain Verilog for behavioral modeling of the IDAC makes it difficult to catch these errors in a routine way. It is crucial that the representative model of the IDAC used for logic verification present its transfer function from the digital to the current domain. The next section describes one way to do this.

IV. MODELING

For the purpose of analog block integration into a largely digital design, a model that provides a reference analog transfer function as a function of digital and analog inputs is necessary. The Verilog language does not have the notion of an "analog wire," however, so behavioral modeling of this type is not straightforward.

A number of groups have proposed work-arounds for this limitation of Verilog. One technique is to transform each such "analog wire" into a bus carrying a real value [5]. Another promising approach extends Verilog by adding PLI calls to send analog values over wires in the netlist database [3]. Both of these methods avoid the execution overhead of Verilog-AMS simulation, which integrates differential equations over continuous time [6]. In contrast, these methods promote computing a new analog value only at each required clock cycle.

In essence, these methods advocate event-driven, discrete time, piecewise-constant models for some analog blocks. This model has the benefit of avoiding numerical integration over continuous time. Because values produced by these models are constant over one or more cycles, it is also possible to code exact-value checkers for them as well.

A. A PLI Library for Linear Systems

We have extended the techniques of [3,5] and implemented a PLI library for modeling networks of variable linear circuit elements. Our implementation uses the VPI programming interface [7]. The first level of the library defines system functions for modeling resistors, current sources, voltage sources and switches. Verilog registers of real and integer types assign values to these elements, and their values may be updated dynamically.

In this model, an electrical node is named by a Verilog wire. A wire describes one endpoint of a multi-terminal net in Verilog. Using PLI functions to trace connectivity, we define all of the aliases of the multi-terminal net to refer to the same electrical node. In this manner, wiring hierarchy in

the Verilog database connects electrical nodes in the expected way. This is the same technique used by [3].

A variable resistor is instantiated with the executable call to "\$resistor." An example appears in Figure 5. Wires w1 and w2 identify two electrical nodes, and the call to "\$resistor" creates an electrical circuit element across them. The value of the resistor is dynamically controlled by "rval," which is called the "control register" of the resistor. In the example, the initial value of "rval" is set before instantiating the resistor. Subsequent assignments to the value of "rval" during simulation change the value of the resistor, which changes the steady-state of its electrical network.

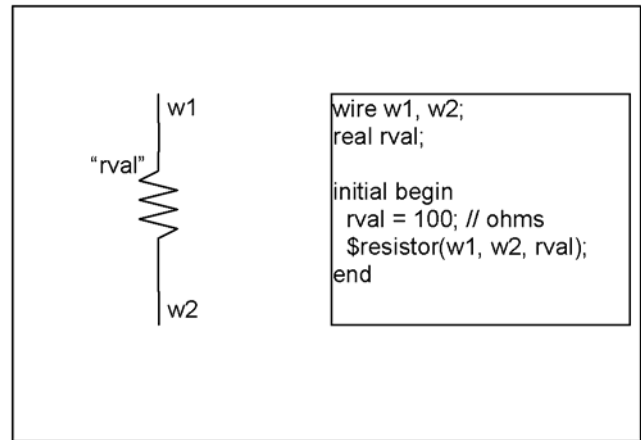


Figure 5. Resistor Instantiation

A variable current source is instantiated as shown in Figure 6. In this example, the Verilog register "cval" is the control register of the current source. The amount of current flowing through the current source may be dynamically changed by re-assigning the value of register "cval."

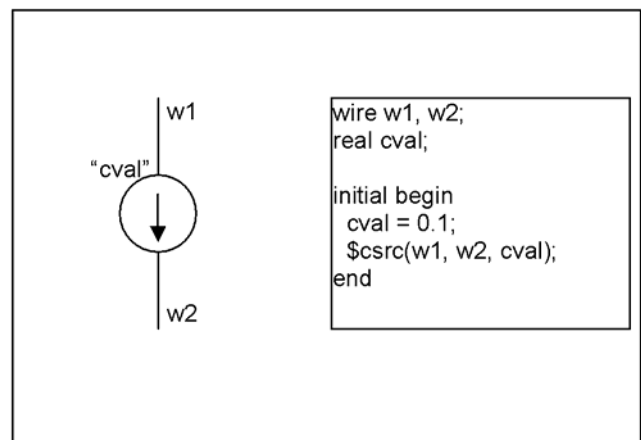


Figure 6. Current Source Instantiation

A variable voltage source may likewise be instantiated as shown in Figure 7. A voltage source has one control register

and also has a "result register" which is used to report the amount of current flowing through the voltage source in the steady-state solution of the subnetwork. In the example, the control register is "vval" and the result register is "ires."

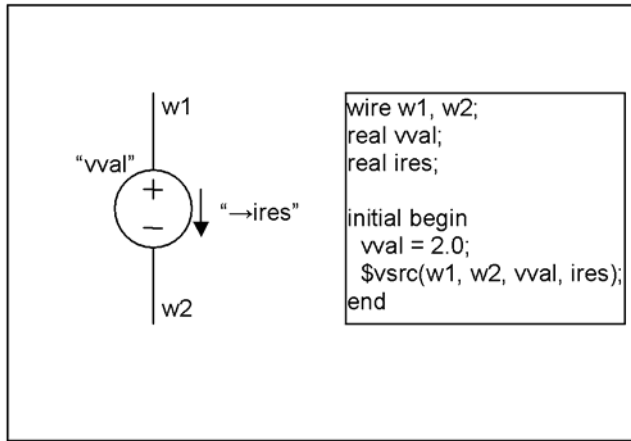


Figure 7. Voltage Source Instantiation

An ideal electrical switch is instantiated as illustrated in Figure 8. The control register of a switch is a Boolean value, and selects whether the switch is ON (1) or OFF (0). An ideal switch has zero resistance (a short circuit) when ON and zero admittance (an open circuit) when OFF [8]. Our model implements these definitions directly.

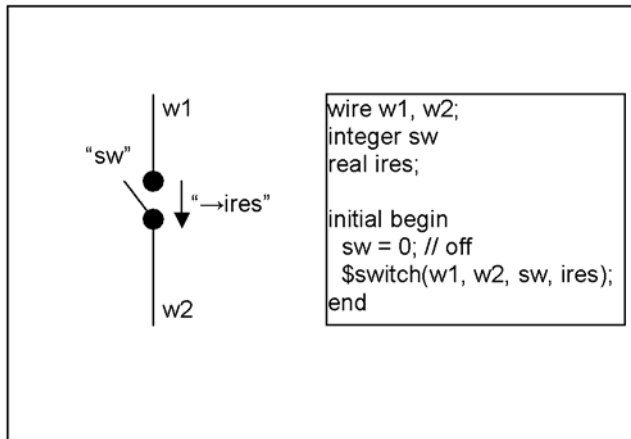


Figure 8. Switch Instantiation

A primitive is also provided for observing a voltage on an electrical node and reporting it in a Verilog register. An example of the use of a voltage probe is shown in Figure 9. Recall that our system does not extend the data types of Verilog. Because we are using wires only as a naming device, the "\$vprobe" system call is the way in which node names (wires) are related to state objects (registers).

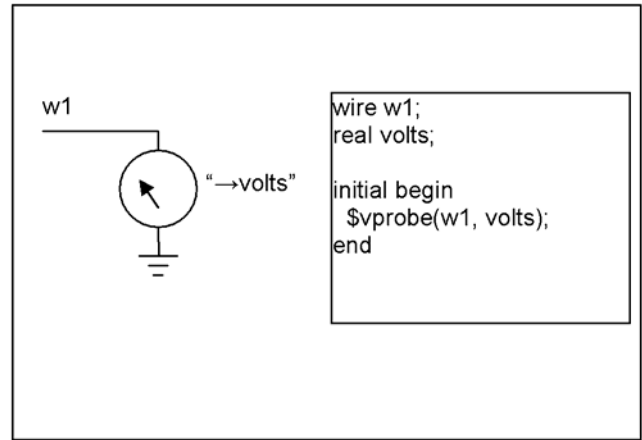


Figure 9. Voltage Probe Instantiation

In this fragment, "volts" is a result register. Whenever the steady-state operating point of the voltage on the electrical node named by "w1" changes due to a change in the value of an electrical element in its subnetwork, the value of "volts" will be changed to reflect the new value. (Note: The function "\$vprobe" shown here is very similar to the function "\$ana_voltage" of [3].)

B. The Simulation Algorithm

A sketch of the event-driven simulation model is as follows. Before simulation starts, each electrically-connected subnetwork (ECS) is identified. This is analogous to the means in which channel-connected components (CCC) are identified for the switch-level elements of Verilog [9,10]. Here, the two terminals of the elements replace the source and drain terminals in the CCC algorithm.

A partitioning of the electrical elements into subnetworks follows from the two definitions below.

- An electrical element can be in only one subnetwork.
- Two electrical elements are in the same subnetwork if they share a multi-terminal net.

At the start of simulation, a listener callback is registered for each control register of a circuit element. When the listener is awoken, the corresponding subnetwork is scheduled for evaluation at the end of the current time step. The new steady-state of the linear time-invariant network is computed using a nodal analysis algorithm [11]. Each result register attached to the subnetwork is then set with a zero-delay update.

These primitives are the building blocks by which digital-to-analog converters of various types may be modeled. An example Verilog definition for the IDAC of the output driver appears in Figure 10. The call to the PLI function "\$scsrc" instantiates the variable circuit element. The body of the "always" block implements the simulation

behavior of the IDAC, changing the value of the current in response to a change in a digital input.

```

module idac (inout wire out,
            input [5:0] ival);

    real isink;

    initial begin
        isink = 0.0; // initial current
        $csrc(out, GND, isink);
    end

    always @(ival)
        case (ival) // decode value
            5'b00000: isink = 0.1;
            5'b00001: isink = 0.15;
            ...
            5'b11111: isink = 1.65
        endcase
    endmodule

```

Figure 10. IDAC Behavioral Model Code Listing

The "case" block has the important task of recording the encoding contract between digital designer *using* the IDAC, and the circuit designer *implementing* the IDAC. Using the behavioral model shown here, the system verifier can observe the effect of the digital signals at the pins of the device. A block-level verification step is also required to rigorously show the desired correspondence between the behavioral model and the circuit.

V. EXPERIMENTAL RESULTS

We modeled the output drivers of a testchip PHY using the electrical elements shown here, and modified an existing purely-digital testbench to one that checked the voltage-based recurrence values of Equation 2 on pin "vp". Our testbench wrote to the registers controlling the coefficient values (idac0, idac1 and idac2) and generated a data stream. By allowing pseudo-random values across the permitted range of coefficients, our testbench could explore and check that the received sequence of voltages matched that expected for the given data stream and coefficient values.

Figure 11 shows the resulting piecewise-constant waveforms from such a system. The signal at the top shows the digital-only output of the pre-existing testbench. The second signal is the transmitted "analog" value, and the third is the analog signal as sampled by a receive clock. The fourth signal is the "expect" value as given by the definition recurrence. The fifth signal is the numerical difference between the sampled and the expect value, which remains zero (within a small floating point tolerance) through the course of the simulation, indicating no error.

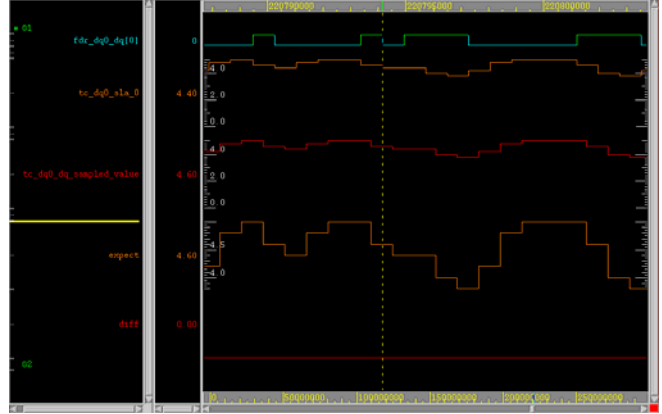


Figure 11. Waveform Capture from Good System

Using digital-only behavioral modeling, we were unable to detect certain classes of errors, namely, those described in Section III. To demonstrate detection of these types of errors, we crossed the control busses between idac0 and idac1. For programmed coefficients where idac0 and idac1 differ, there should be observable errors. Figure 12 shows part of a trace from such a case. As expected, the difference term is non-zero. Interestingly, for certain sequences of bits in this case, there are periods of time where the difference is zero. This shows that the detection of errors is data-dependent, and that at least a pseudo-random approach is necessary.

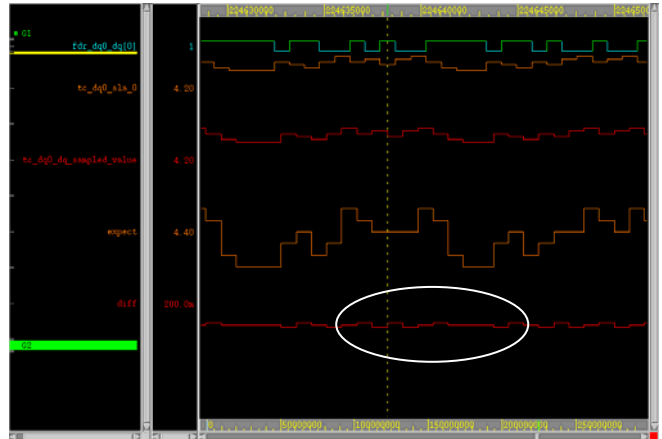


Figure 12. Waveform Capture of Erroneous System

A. Performance

We also measured the computational overhead of our approach. Each electrical subnetwork should require roughly one matrix inversion to evaluate its steady state, and we were interested in how much CPU time this would cost. The other main use of time is PLI (VPI) overhead.

As a baseline, the PHY system without any analog modeling required 251 CPU seconds to run a complete test. The full system has a three stage output driver, but we also wanted to observe the performance of systems with simpler output drivers and compare the results. We created systems

with one, two, and three-stage output drivers and measured the CPU time used using both dense and sparse matrix solvers. Our dense matrix solver is a straightforward LU decomposition with full pivoting. For our sparse matrix solver we used Meschach [12].

Tables 1 and 2 show the simulation time data collected for the three configurations. Figure 13 presents this same data as a graph, with the sparse and dense solvers compared side by side for each configuration. We measured the total simulation time, and the CPU time used by the numerical matrix inversion. The rest of the time (minus the baseline) was lumped into a category called "Other." Significant uses of time in this category are PLI overhead, and for the sparse case, the lookup of matrix elements by a search function on their indices.

Table 3 gives information about these four configurations, including the number of electrical devices, the order of the matrices generated, the number of digital events (toggles) and the number of calls to the matrix solve routines.

For the full three-stage output driver, using the sparse solver, the total simulation time rose to 448 seconds. It is interesting that only 64 seconds of this time was spent inverting the matrix. The other 133 seconds is dedicated to PLI interfacing, and searching the matrix for entries. For this same configuration, the dense solver devoted 86 seconds to overhead; this amount is all PLI interfacing time, since the matrix entry time update is insignificant.

It is interesting to compare the dense implementation for the extremely small subnetwork resulting from a single stage output driver. For this case, the total simulation time is less than that for the sparse case. Notably the "other" time is much smaller, because the matrix elements are located using direct index arithmetic. Our dense LU algorithm is not very well optimized, actually, and we think we could make it better. This suggests that for very small subnetworks a dense matrix solver might be preferred over a sparse one.

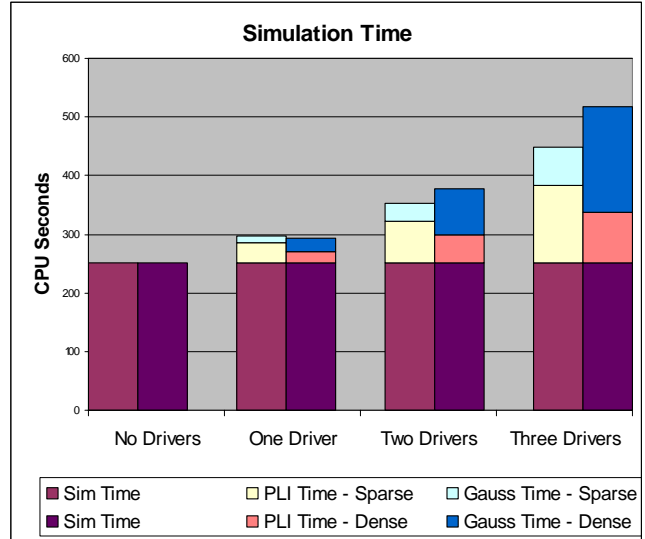


Figure 13. Simulation Times

TABLE I. SIMULATION TIME WITH SPARSE SOLVER

	Sim Time	Sparse	Other (PLI)
No drivers	251	-	-
1 driver	297	11	35
2 drivers	352	30	71
3 drivers	448	64	133

TABLE II. SIMULATION TIME WITH DENSE SOLVER

	Sim Time	Dense	Other (PLI)
No drivers	251	-	-
1 driver	294	24	19
2 drivers	377	78	48
3 drivers	517	180	86

TABLE III. TEST CONFIGURATIONS

	No drivers	1 driver	2 drivers	3 drivers
Devices	0	216	345	432
Order	0	7 (x36)	10 (x36)	13 (x36)
Switch toggles	0	1.1M	2.2M	3.4M
Solves	0	573K	852K	998K

VI. CONCLUSION

This paper has shown an example mixed-signal subsystem and described how its integration into a predominantly digital system is not straightforward using existing modeling capabilities. To address this deficit, we have proposed an event-driven analog framework and shown how it can be used with a mathematical description of the sequence of voltages observed to detect wiring and logic errors. Using simulation performance data, we showed that the computational overhead is reasonable for the output drivers shown.

ACKNOWLEDGEMENTS

I would like to thank my colleagues Kevin Jones, Kathryn Mossawir, Victor Konrad and Jaeha Kim for their feedback and comments as this work has proceeded.

REFERENCES

- [1] Mixed-Signal Cosimulation Methodology. Jeff McNeal and David A. Yokoyama-Martin. In Proceedings DVCon, 2007.
- [2] PHY Verification - What's Missing? Thomas Sheffler, Kathryn Mossawir and Kevin Jones. In Proceedings DVCon, 2007.
- [3] Sending Analog Values Along Digital Wires. Chris S. Jones, Jeff McNeal and Ross Segelken. In Proceedings DVCon, 2007.
- [4] Equalization and Clock Recovery for a 2.5-10-Gb/s 2-PAM/4-PAM Backplane Transceiver Cell. Jared L. Zerbe, Carl W. Werner, Vladimir Stojanovic et. al. IEEE Journal of Solid-State Circuits. Vol 38, No. 12. 2003.
- [5] Modeling, Simulation, and Design of a Multi-Mode 2-10Gb/s Fully Adaptive Serial Link System. Carl Warner, Claus Hoyer, Andrew Ho et. al. In Proceedings 2005 Custom Integrated Circuits Conference.
- [6] The Designer's Guide to Verilog-AMS by Ken Kundert, Olaf Zinke Copyright 2004 Kluwer Academic Publishers Norwell, MA 02061 ISBN: 1-4020-8044-1.
- [7] The Verilog PLI Handbook. Stuart Sutherland. Springer. 2002.
- [8] Time-Domain Analysis of Networks with Internally Controlled Switches. David Bedrosian and Jiri Vlach. IEEE Transactions on Circuits and Systems. Vol 39. No 3. 1992.
- [9] Boolean Analysis of MOS Circuits. Randal E. Bryant. IEEE TCAD, 6(4), pp. 634-649, Jul. 1987.
- [10] Verilog-XL User Guide. Product Version 5.1. Sept 2003.
- [11] Electronic Circuit and System Simulation Methods. Lawrence T. Pillage, Ronald A. Rohrer and Chandramouli Visweswariah. McGraw-Hill, TX. December 1, 1994.
- [12] <http://www.math.uiowa.edu/~dstewart/meschach/>