

# Behavioral Simulator of Analog-to-Digital Converters for Telecommunication Applications

Grzegorz Zareba, Olgierd A. Palusinski

University of Arizona  
1230 E Speedway Blvd  
Tucson, AZ, 85721  
(1) - 520 626 70 78

zgrzes@ece.arizona.edu, palusinski@ece.arizona.edu

## ABSTRACT

This paper presents a new Behavioral Simulator capable of modeling various types of Analog-to-Digital Converters (ADCs). Behavior of Basic Building Modules (BBMs) of ADCs, such as for example sample-and-hold, and comparators is encapsulated in Dynamic Linked Libraries (DLLs). Predefined behavioral models of BBMs and a netlist describing connections between them are used to form a behavioral representation of simulated ADC. As an example of BBM, a behavioral model of comparator module is presented. An 8-bit multi-stage ADC is used to demonstrate simulation process.

## 1. INTRODUCTION

Analog to Digital Converters (ADCs) [1] built in CMOS technology are essential elements of wireless communication systems (for example in zero-IF receivers [2]). In communication applications many features of data converters are important: high speed, wide dynamic range, and low power consumption (for portable devices).

To verify the performance of designed ADC several behavioral level simulations have to be performed. Many different circuit structures are used for ADC's implementation [1] (flash, n-stage pipeline, folding and interpolating technique) such that behavioral simulation of ADCs requires a simulator, which is dedicated to a given converter structure [3] or uses a standard mixed-mode simulator [4]. Dedicated simulators have very limited utilization and due to excessive programming effort needed for implementation of converter model in such simulators they are very rarely used. The second method of modeling ADCs is based on a specialized simulation language (Mast, VHDL-A, or Verilog-A), is error-prone, and extremely expensive in terms of computer time. Another restriction in using standard simulators (for example Simulink) is that in many cases it is very difficult or even not possible to properly build behavioral models [4][5] and simulate some important imperfections of converters for example clock jitter and glitch energy in analog switches.

A new approach in behavioral modeling of ADCs presented in this paper is based on utilization of Dynamic Linked Libraries (DLLs) to encapsulate behavior of basic blocks of ADCs. Basic Building Modules (BBMs) of ADCs such as comparators, folding circuits, analog switches, binary encoders and many others are used to form a behavioral model of various types of ADCs.

The main advantage of this approach is that all the pre-defined BBMs are independent from the simulator core, and because DLL modules are executable files simulation time is significantly reduced (no translation or interpretation of simulation language commands is needed).

Presented Behavioral Simulator has been developed in VC++ environment with some external components developed in C# language.

## 2. DESCRIPTION OF THE BEHAVIORAL SIMULATOR

The Behavioral Simulator consists of three main parts: library of BBMs, configuration files, and simulator core (Figure 2-1).

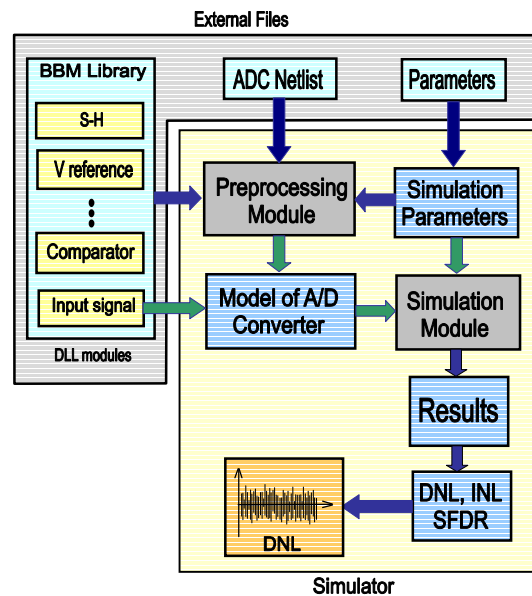


Figure 2-1 Block diagram of the Behavioral Simulator.

The library of BBMs includes all DLL modules, which are used to build a behavioral representation of simulated ADC. A set of configuration files allows description of particular architecture of an ADC using a *Net-list*, and a *Parameters* files. A simulation session is described by simulation parameters kept in a *Configuration* file.

The simulation core is divided into two separate modules: *Preprocessing Module* and *Simulation Module*. The *Preprocessing Module* is responsible for preparing a behavioral model of an ADC in the form of a dynamic multilevel list (static structure is not suitable because simulator is to be applied for simulation of various converter types). The *Simulation Module* performs behavioral simulation of given ADC and allows post-processing of data obtained during simulation (including calculation of converter's performance metrics).

Simulation framework, presented in Figure 2-1, handles various converter structures, major circuit imperfections, and it is much simpler to use than existing simulators.

## 2.1 Preprocessing Module

The *Preprocessing Module* creates a dynamic multilevel structure, which fully describes architecture of simulated A/D converter (Figure 2-2).

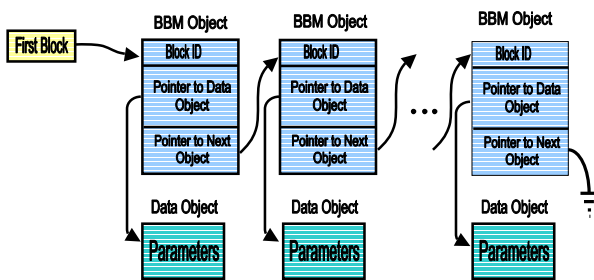


Figure 2-2 Dynamic list used to define internal structure of ADCs.

The dynamic multilevel structure presented in Figure 2-2 is created using two text files: *Net-list* and *Parameters* file. The first file describes connectivity between BBMs used to form internal structure of an ADC. The second file provides parameters associated with all used BBMs. Separation of behavioral modules (BBMs) from their parameters allows for easy and simple reusing of the same converter's structure for multiple simulations with different parameters of BBMs. In HDL based simulators all parameters of behavioral models are kept inside these models [6]. Because of this it is necessary to implement and store several identical designs differing only by the parameter values in order to investigate behavioral response of the system with various sets of parameters.

Figure 2-2 presents a simplified structure of the multilevel list describing simulated ADC. The multilevel list consists of *BBM Objects*, which corresponds to a set of BBMs used to create behavioral representation of simulated ADC. *BBM Objects* are used to link the behavioral description of the block (kept in DLL files) with the simulator. Each *BBM Object* has a pointer to a *Data Object*. The *Data Object* is used to keep a set of parameters of the particular BBM, for example value of droop rate in the case of a sample-and-hold module. All parameters of BBMs are kept in the *Parameters* file, and are copied into the *Data Objects* during creation of the ADC model.

The *Preprocessing Module* provides a convenient tool for modifying parameters of BBMs. A screenshot of a dialog used for editing parameters of BBMs is presented in Figure 2-3.

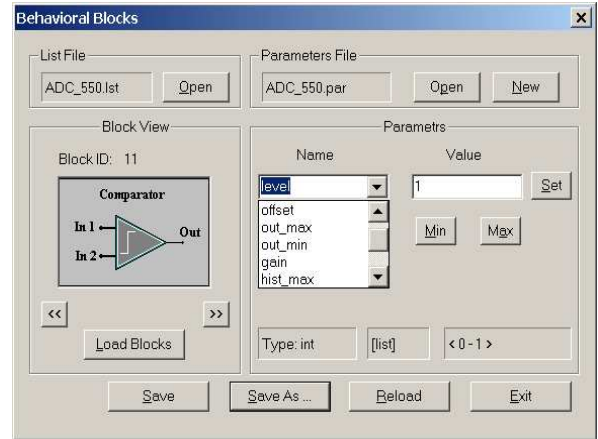


Figure 2-3 Preprocessing Module – dialog for setting up parameters of comparator module.

## 2.2 Dynamic Linked Libraries

DLLs are similar to the standard library in that they both contain sets of functionality that have been packaged for use by applications [7]. The difference is visible when the application links to the library. In the case of a standard library module (LIB) the application is linked to the library during the compile and build process. In the case of a DLL the application links to the library when the application is run. The DLL file remains a separate file that is referenced and called by the application.

The main reason for using DLLs in developed simulator is that DLL files can be updated and modified without having to update the application executable. Another advantage of using DLLs is that any programming language or environment (VB, VC++, or C#) capable of generating DLL modules can be used to create a behavioral description of BBMs.

Figure 2-4 illustrates the way in which a DLL module communicates with the Simulation Module.

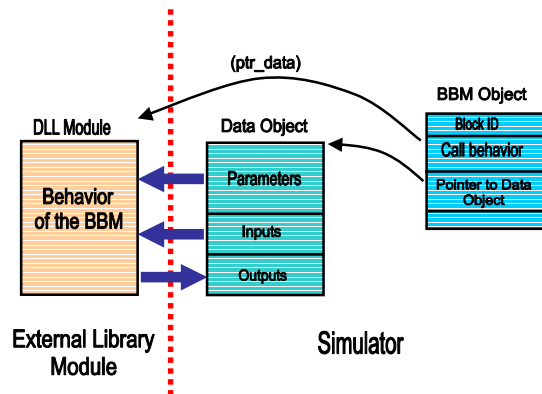


Figure 2-4 Communication interface between the simulator and a DLL module.

The main interface between a DLL module and the simulator is a *Data Object* (Figure 2-2 and Figure 2-4). Each *Data Object* is directly “connected” to a BBM represented by *BBM Object*. The *Data Object* is divided into three separate sections: parameters, inputs, and outputs section. The Parameters section keeps all

parameters of the block, which are read from the *Parameters* file during initialization of the *BBM Object*. This section is used by the DLL module to determine current parameters of the BBM, and can be modified only by the simulator core; the DLL module can only read these parameters. The Input section holds current values of the input terminals and is read by the DLL module before new output values are calculated. When the BBM module is activated it calls *RunBlock(BBlock \*)* method from the DLL module and waits until the DLL module updates the Output section. The *RunBlock(BBlock \*)* method implements the behavior of the BBM, the *BBlock* data structure is the communication interface between the DLL module and the simulator. Behavior of a BBM can be described using for example, mathematical formulas, look-up tables, or logical statements.

### 2.3 Simulation Module

The *Simulation Module* is partially based on an event driven simulation technique [8] and a data flow technique [9]. The data flow technique is similar to the technique employed in data flow computers e.g. an action is taken only if all inputs of the object have valid data, and then results of performed operation is distributed among all objects connected to the output of the activated object. Combination of these two techniques results in a significant reduction of the simulation time.

Another important feature of developed simulation algorithm is reduction of generated events due to introduction a new group of BBMs called passive modules. In a standard event-driven simulator all objects, which are used during simulation can generate events. In the case of large systems, the number of events, which are to be processed during simulation, can slow down the simulation. To overcome this problem in addition to a standard group of BBM, which generate events, a new group of passive BBMs have been designed. Passive BBMs cannot generate events; their activation depends on other modules: both active and passive BBMs. Proper interaction between all BBMs is achieved by utilization of the data flow technique.

Before the simulation starts the *Preprocessing Module* initializes the event queue. During initialization each active module posts its first event and sets the time when the second event will be posted (the time of next event is kept inside the *BBM Object*). When an active module is called it automatically posts next event, and then activates all passive modules connected to its output or control terminal. Each activated module (*BBM Object* in Figure 2-2) checks its input(s) to determine if the input values are valid. If the input values are valid the DLL module is called to calculate the new output value. When the new output value is known the module distributes it among all modules connected to its output. If at least one input value is not valid the passive module calls the module connected to its input. This process is repeated until all inputs of the activated passive module are valid, and then the DLL module is called to calculate the new output value. Any passive module can call its DLL module only if all its input terminals have valid values, which is similar to the data flow concept.

Before each simulation session a user can set simulation parameters such as: simulation time, simulation mode, sampling frequency and type of the input signal. A simulation setup dialog is presented in Figure 2-5.

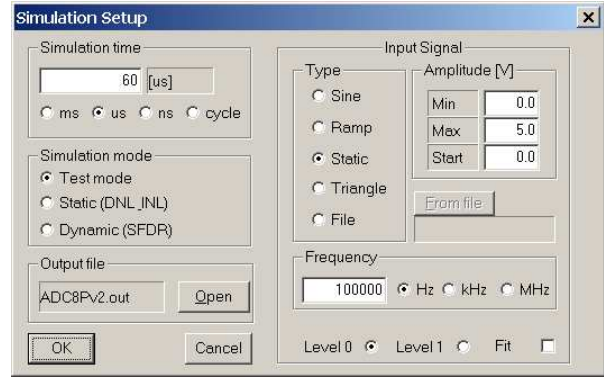


Figure 2-5 Simulation setup dialog.

## 3. BASIC BUILDING MODULES

Basic Building Modules (BBMs) have been designed as separate modules from which several architectures of ADCs, such as flash, pipelined and folding, can be built and simulated. The interaction between the Simulator Module and BBMs during simulation process is limited to control and management operations. The Simulator Module is primarily responsible for distributing data between all BBMs and checking for new events in the event queue.

The behavior of a BBM is entirely encapsulated inside the DLL module; therefore simulation results depend on interaction between BBMs (DLLs) and imperfections included in the behavioral description of BBMs.

Each BBM is defined by three main elements: a *BBlock* data structure, a set of terminals, and a DLL file (Figure 3-1).

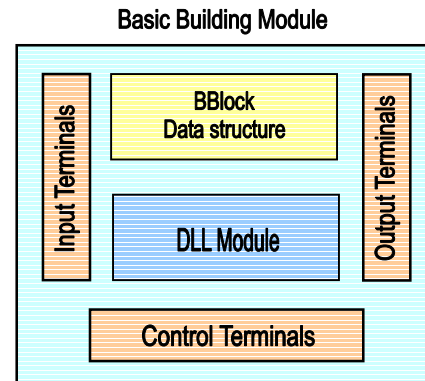


Figure 3-1 Structure of BBM.

The *BBlock* data structure defines parameters and variables used by BBMs during simulation. The set of terminals includes: *Input*, *Output* and *Control* lines, which define the interface between particular BBM and the rest of the circuit.

The functionality of the BBM is defined inside the DLL module. The DLL determines the behavior of BBM, how inputs are used and how outputs are generated.

Utilization of DLL modules gives very good flexibility in defining internal structure (behavior) of a BBM. There are no restrictions in programming language, which can be used to prepare the DLL module. Any programming environment, which supports generation of DLL modules, can be used to create behavior of a

BBM (for example VC++,C# or Visual Basic). There are also no restriction in terms of possible data structures and commands used to describe behavior of BBMs. Advanced mathematical formulas and look-up tables can be easily used. In the classical simulation languages such Verilog and VHDL [10] only limited data structures and commands can be used to create behavioral models of simulated circuits, which sometimes complicates creation of models and additionally extends simulation time.

Several BBMs have been developed to support simulation of flash, multi-stage, pipelined, and folding ADCs:

- Input Signal Module
- Clock Source Module and Clock Delay Module
- Comparator
- Register, Shift Register and Digital Parallel Register
- Sample and Hold
- Folding circuit
- Voltage Reference with Resistor Ladder
- Sub-ADC and Sub-DAC – for pipeline architecture
- Digital Correction – for pipeline architecture
- Binary encoder
- Analog Switch

The main advantage of encapsulation of BBMs in DLL files is that the behavior of a particular Module is kept away from the behavior of other Modules and from the Simulation Module. In many cases restrictions imposed on standard behavioral or mixed-mode simulators led to necessity to combine two or more features or imperfections of functionally separated BBMs. For example, including variations of reference voltage in the model of comparator [5] will hide the real behavior of the Reference Voltage module and will result in less accurate simulation results.

### 3.1 Example of BBM – model of comparator

Behavior of a comparator circuit is approximated using five main regions of operation (Figure 3-2).

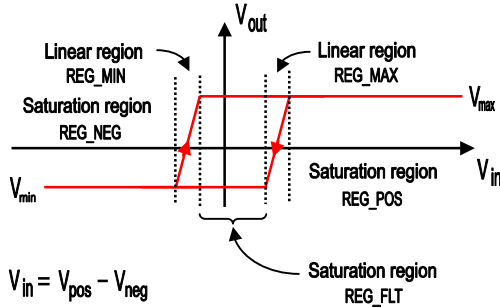


Figure 3-2 Regions of operation of comparator module.

These five regions include: two regions, in which the comparator works as an amplifier (finite gain), and three regions, in which the comparator is saturated. Two regions in which the comparator is saturated lie on the right and left side of the hysteresis points. The third region of saturation is located between the hysteresis points. In this case the output of comparator can have two different values: min and max. To determine the proper output value, the slope of the input signal has to be known.

Following equations are used to determine the output value in described regions of operation of Comparator module.

Saturation regions: REG\_NEG + REG\_FLT

$$V_{out} = V_{min}$$

Linear region: REG\_MIN

$$V_{out} = \frac{H_{min} - \frac{A \cdot (V_{max} - V_{min})}{2} - V_{in}}{A}$$

where

$H_{min}$  - negative hysteresis point (when the input signal increases),

$A$  - finite gain,

$V_{in}$  - input voltage.

Linear region: REG\_MAX

$$V_{out} = \frac{H_{max} - \frac{A \cdot (V_{max} - V_{min})}{2} - V_{in}}{A}$$

where

$H_{max}$  - positive hysteresis point (when the input signal decreases).

Saturation regions: REG\_FLT + REG\_POS:

$$V_{out} = V_{max}$$

Presented behavioral model of Comparator Module can be used to investigate following imperfections: finite gain, input offset voltage, slew rate, and hysteresis. All listed imperfections can be selectively turned on and off.

## 4. SIMULATION SETUP

Simulation of ADCs with utilization of developed Behavioral Simulator consists of several steps. In the first step, a design of ADC built using defined BBMs has to be prepared. The simulator requires two text files to create behavioral representation of the ADC: *Net-list* and *Parameters* files. It would be very inconveniently, although possible, to create these two files by hand.

Two options were considered in terms of creating *Net-list* files: separate graphical module attached to the simulator framework and use of existing graphical interface from available simulation tools. The first option would involve extensive programming; therefore the second option was chosen. Among many available simulation tools, PSpice appeared to be very suitable for this task.

PSpice Schematics is an original interface of PSpice simulator used to create schematics of electronic systems. Because developed simulator uses different basic elements than those included in PSpice Schematics libraries, a new library with all designed BBMs was created. Once the schematic is completed, the net-list can be generated. The net-list generated by PSpice Schematics is not compatible with the net-list required by the Behavioral Simulator. A special translation application, Net2Net.exe, was developed (using C# language [11]) to convert PSpice net-list into the Behavioral Simulator's *Net-list*. The Net2Net.exe application generates also *Parameters* file. Initially,

all parameters of used in the design BBMs are set to their default values. Later, these default values can be changed using the *Preprocessing Module*.

### 4.1 Simulation Modes

Three main simulation options are available in the simulator (Figure 2-5): test, static, and dynamic simulation. The first option is used for testing new BBMs and evaluating their performance. The static and dynamic simulations are used to obtain static and dynamic performance metrics of simulated converters.

In the static simulation mode offset and gain error are calculated as well as Differential Nonlinearity (DNL) and Integral Nonlinearity (INL) errors. The dynamic simulation mode is used to achieve information about dynamic performance of the converter, mainly SFDR.

The static and dynamic simulation differs only in the shape of the input signal and some post processing procedures used to determine output data format, which will be used to calculate performance metrics. For a static simulation a ramp signal is generated by the Input Module, for the dynamic simulation a sinusoidal signal is required.

### 4.2 Simulation Results

A special BBM called Register is used to collect simulation results. A Register module captures data at certain moments in time according to the control signal. Functioning of Registers is similar to functioning of Current/Voltage Markers in PSpice. A Register module can be placed at any point of the circuit. Captured data are stored in a text file. Registers can have from 1 to 16 input lines, and therefore can capture a vector with maximum 16 input values plus a time value at which the input vector was captured. To illustrate the use of Registers an example of simple circuit is presented in Figure 4-1.

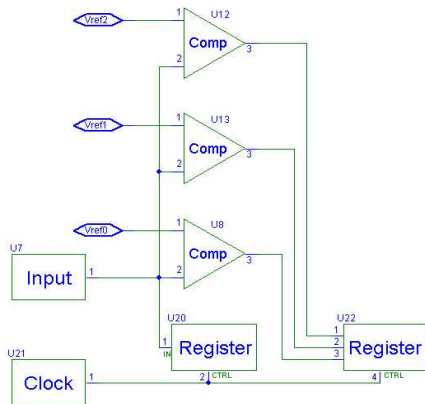


Figure 4-1 Example circuit with Register modules.

Register U20 is used to capture the input signal according to the clock signal generated by Clock Module U21. Register U22 is used to capture output values of comparators U12, U13 and U8. Data collected by Register U22 can be seen as the output values generated by a 2-bit flash ADC.

It is necessary to place one Register module at the output of the simulated converter in order to collect output values generated during simulation. A text file created by the 'output' Register can be used to calculate performance metrics of the converter (INL, DNL, or SFDR) or just to examine the output data.

Registers are ideal modules therefore they do not affect the functioning of the simulated converter and can be placed at any output terminal of any BBM.

## 5. SIMULATIONS OF ADCs

A simple 8-bit multi-stage ADC was chosen to demonstrate the simulation process. The simulated ADC comprises 17 comparators, 17 analog switches, 3 binary encoders, one voltage reference with resistor ladder, and several Registers (in total 46 BBMs). The schematic of the converter is presented in Figure 5-1.

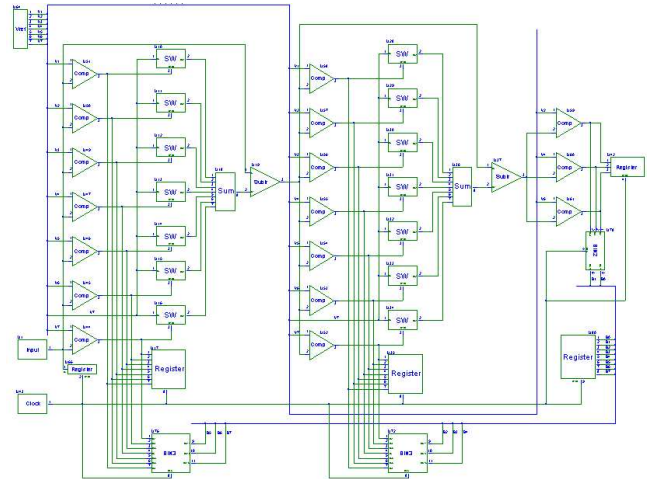


Figure 5-1 Schematic of 8-bit multistage ADC.

Several simulations were performed, starting from the simulation, where all BBMs were ideal. The main purpose of using ideal modules is to confirm proper functioning of the design. When the design is validated, some imperfections can be included.

Presented circuit was simulated with the following imperfections:

Reference voltage with resistor ladder:

- stability 99.5%
- resistor ladder mismatch 0.5 %

Comparators:

- input offset voltage: random, +/- 3 mV

Simulation time in the static simulation mode is less than 10 seconds (PIII 733 MHz, 256 MB RAM). Simulation results presented in Figure 5-2 include: offset and gain error, DNL and INL errors before and after gain correction.

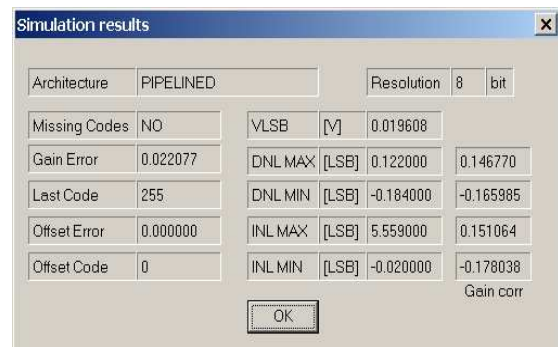


Figure 5-2 Dialog with simulation results.

The graphical illustration of the DNL and INL errors is presented in Figure 5-3 and Figure 5-4.

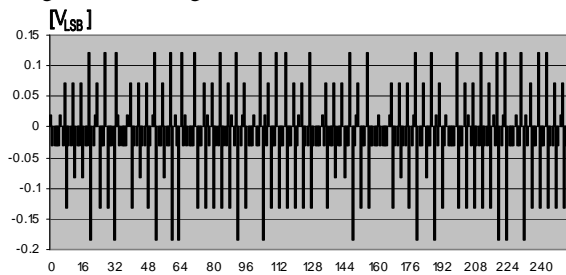


Figure 5-3 DNL error of simulated ADC.

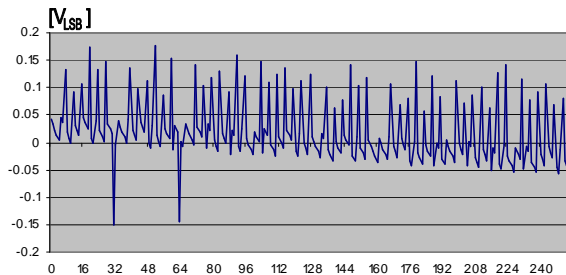


Figure 5-4 INL error of simulated ADC.

## 6. Conclusions

Presented Behavioral Simulator allows modeling various converter structures including flash, multi-stage, pipelined, and folding ADCs. Imperfections of Basic Building Modules (BBMs) are part of the behavioral description of BBMs and not of the simulator. This approach simplifies modeling and reduces simulation time as well as gives more realistic simulation results.

Imperfections of BBMs can be selectively turned on and off, therefore it is possible to investigate and estimate an influence of a chosen limitation of a single BBM on the converter performance. The Behavioral Simulator can evaluate static and dynamic responses of the ADC such as offset and gain error, DNL, INL, and SFDR.

Two examples show time efficiency of developed simulator. First, an ideal model of 8-bit multistage ADC described in previous section has been built using Simulink. Time needed to complete simulation in Simulink has exceeded 6 minutes, while the developed simulator has completed simulation in less than 10 seconds. The results of both simulations were identical. Adding any imperfections in the model developed in Simulink, would extend preparation time of the circuit, and would significantly extend the simulation time, (because the number of blocks used to implement these imperfections would be increased as well). Another inconvenience in the case of Simulink is access to parameters, which further extends time needed to obtain final simulation results. Second example is a BBM developed to support simulations of switching activities in a binary decoder used in current steering Digital-to-Analog converters (DACs). This BBM is needed for estimation of clock jitter in such DACs.

Model of the BBM developed in Matlab allowed for simulations up to 8-bit binary encoders, with simulation time about 20 minutes. The same block converted into a DLL module allows for simulations up to 16-bit binary encoders with simulation time less than 30 seconds (10 seconds for 8-bit encoder). Simulations of more than 8-bit binary encoders in the case of using Matlab are not practical due to enormous time needed to obtain simulation results.

## 7. ACKNOWLEDGMENTS

This work was partially carried out at the Center for Low Power Electronics and Connection One Research Center.

## 8. REFERENCES

- [1] David F. Hoeschele. *Analog-to-Digital and Digital-to-Analog Conversion techniques*. John Wiley & Sons, New York, 1994.
- [2] Da-You Sun, Jian Xu, *Zero-IF topology*, Electronics Letters, Volume: 36, Issue: 12, 8 June 2000, Pages: 1009-1010.
- [3] Venkata K. Navin et al. *A Simulation Environment for Pipelined Analog-to-Digital Converters*. IEEE International Symposium on Circuits and Systems, June 9-12, 1997, Hong-Kong.
- [4] Gashing Ruan. *A Behavioral Model of A/D Converters Using a Mixed-Mode Simulator*. Custom Integrated Circuits Conference, 1990, Proceedings of the IEEE 1990.
- [5] F. Maloberti, P. Estrada, P. Malcovati and A. Valero. *Behavioral Modeling and Simulations of Data Converters*. Proceedings of International Workshop on ADC Modeling and Testing (IWADC '00), Vienna, Austria, pp.229-236, September 2000.
- [6] E. Christen, K. Bakalar. *VHDL-AMS – A Hardware Description Language for Analog and Mixed-Signal Applications*. IEEE Transactions on Circuits and Systems, Analog and Digital Signal Processing, Vol. 46, No. 10, October 1999.
- [7] D. Chapman, J. Heaton. *Visual C++ in 21 Days; Professional Reference Edition*. Sams Publishing, A Division of Macmillan Computer Publishing, 1999.
- [8] R. A. Saleh, A. R. Newton. *Mixed-Mode Simulation*, Kluwer Academic Publishers, Boston, 1990.
- [9] K. Hwang, F. A. Briggs. *Computer Architecture and Parallel Processing*, McGraw Hill Inc., 1985.
- [10] S. Ghosh. *Hardware Description Languages – Concepts and Principles*. IEEE Press Series on Microelectronic System, New York, 2000
- [11] B. Wagner. *C# Core Language Little Black Book*, The Coriolis Group, LLC, 2002.