Asynchronous Tracking ADC for digital DC-DC converters

Frederico Pratas, (fcpp@inesc-id.pt) and Marcelino Santos, (marcelino.santos@ist.utl.pt)

INESC-ID
Avenida Alves Redol, 9
1000-029 Lisboa, Portugal

Abstract—An analog-to-digital (A/D) conversion technique compatible with standard CMOS is described. We propose a fully asynchronous modular tracking analog-to-digital converter (ADC) designed for DC-DC converters with digital control. The main objective of this work is to reduce power consumption in this type of converters by reducing the activity needed by a typical synchronous ADC. Moreover, the solution proposed herein is fully modular and can be easily adapted to different DC-DC control laws by changing the number of output bits and/or by adjusting the quantization levels of the feedback digital-to-analog converter (DAC).

1 INTRODUCTION

Analog-to-Digital Converters (ADCs) are useful building blocks in many applications because they represent the interface between the real world analog signal and the digital signal processors [1], [2]. The typical architecture of an ADC is presented in Figure 1. There are a lot of implementations that have been reported in the literature in order to obtain high-speed ADCs [3], [4]. Flash architectures [5], [6] have received a great interest since they offer a high sampling frequency and a high conversion speed, because of its fully parallel architecture. Their main disadvantage is the power consumption which increases exponentially with the resolution. Successive approximation ADCs [7], [8] have also been reported in order to minimize the complexity and the power consumption. Nevertheless, the need of multiple clock cycles to implement the conversion algorithm due to their logarithmic dependence on resolution still represents a huge bottleneck in performance and power consumption. Solutions such as [9], [10], [11], [12] try to overcome the performance limitations by parallelizing the architecture and performing time interleaving, but do not completely solve the power consumption problem. Tracking ADCs [13] have also received a great interest since they provide a single cycle conversion and their complexity is independent of the resolution.

Another characteristic of ADCs that has been studied in [14], [15] is the possibility to have an asynchronous control logic instead of a synchronous one. Although this can significantly increase the complexity of the system, it can also improve its power efficiency by performing each conversion only when a significant variation in the input is observed.

In this work, we present a one-hot encoded tracking ADC with asynchronous control. Our main concern in this work is to make the design suitable for DC-DC converters with digital control (see Figure 2) such as the ones proposed in [16], [17], which are used mainly in embedded systems of SOC (System on Chip) powered by batteries. We believe this type of ADC is best suited to be used in such converters because of: i) the natural tracking characteristics of these systems, since the goal of the controller is to maintain a constant output voltage; ii) its ability to sustain a fast conversion response; iii) the number of conversions are maintained to the minimum, thus reducing the so critical power consumption and increasing the autonomy of the systems; iv) its modularity and simplicity.

The rest of this document is organized as follows. Section 2 describes the proposed architecture and gives some considerations about the synchronous mode, the asynchronous mode, and the feedback DAC implementation. Section 3 provides the implementation details of the different digital modules used in this work, and Section 4 presents an analysis of the timing constraints. Section 5 shows the details about the simulations and the results obtained and finally Section 6 concludes the work.

2 PROPOSED ARCHITECTURE

Typically tracking ADCs such as the ones presented in [13], [14] are based on two comparators and a up/down counter as shown in Figure 3. This architecture is able to follow a certain input voltage by tracking the quantization interval where the input voltage is located at conversion time. In order to do that both comparators are feed by the input voltage, one comparing it to the voltage corresponding to the actual quantization level.
(-LSB), and the second comparing it with the next quantization level (+LSB). While both comparators are at '0', it means the input voltage is between the two levels, and the converter keeps the same output value. When the input voltage crosses one of the quantization levels, the comparators detect it, and the counter is updated. If the -LSB comparator is at '1', the input crossed the lower quantization level and the counter is decremented. If the +LSB counter is at '1', the upper level was crossed and the counter is incremented. It is also worth to note that under reset, the shifting module can be restarted in a position that is more likely to be correct, or that is nearer to the correct value. Usually it is restarted at the middle of the scale.

While the architecture presented herein is similar to the ones described before, we propose some modifications to improve DC-DC converters with digital controllers. In this work instead of a up/down counter we opted by a shifting module (see the control block in Figure 4). This module has as output one line per each quantization level. There are two main reasons for this change: i) firstly it allows us to design a simpler DAC, which can be easily adapted to match the amplitude quantization that best fits the DC-DC controller law as will be explained in Section 2.2; and ii) the shifter is faster than the up/down counter, thus allowing us to reduce the total delay of the conversion loop as will be explained in Section 3.

### 2.1 Regular vs. Irregular Sampling

Typical synchronous ADCs perform regular sampling, i.e., each A/D conversion is controlled by an external clock signal with a fixed period $T_{\text{sample}}$ (see Figure 5). This type of sampling where samples are regularly spaced out in time, is also referred to as Nyquist sampling. In this technique samples are taken by a Sample and Hold, respecting the Shannon theorem, and then quantized by the ADC logic. The sampling time instants are perfectly known and the only imprecision is due to the quantization noise added during the A/D operation, which is characterized by the signal-to-noise ratio (SNR). The SNR only depends on the number of bits and is defined by:

$$SNR\,dB(N) = 6.02 \times N + 1.76$$ (1)

On the other hand, asynchronous ADCs apply irregular sampling, i.e., in an asynchronous ADC each sample is triggered only when the input signal crosses a certain reference level (see Figure 6). In this case the amplitude range of the output signal is divided into quantization levels and a sample is performed every time the input signal crosses one of these levels. Contrary to regular
2.1 Sampling considerations

Sampling, in this case the samples do not depend on the clock signal but on the total delay of the conversion loop (i.e., the minimum time needed for one sample $\delta t$), and on the input signal variations (i.e., the sharper the signal, the closer the samples). In other words, the slope of the input signal must respect the following condition:

$$ \left| \frac{d(V_{in}(t))}{dt} \right| \leq \frac{q_{\text{min}}}{\delta t} \quad (2) $$

where $q_{\text{min}}$ is the smallest quantization level.

While the synchronous regular solution performs one conversion per each clock cycle, even if there is no variation at the input line, the asynchronous one performs a conversion only when a significant variation of the input analog signal is detected. Thus, the second case is more power efficient, as the number of conversions performed depends on the input signal slope, i.e., the circuit only performs as much conversions as necessary. The overall design of the Asynchronous Tracking ADC architecture is depicted in Figure 7.

2.2 DAC considerations

In this work we consider a simple resistive divisor such as the one presented in Figure 8(a). This structure is directly fed with the output of a one-hot coded shifter, i.e., a shifter which only has one active bit line at a time. Each bit line $n$ of the shifting block is directly connected to two DAC switches $\text{Inp}[n]$ and $\text{Inp}[n+1]$. Since only the bit corresponding to the actual quantization level will be at '1', the correct $V_n$ and $V_{n+1}$ will be driven to the inputs of the comparators, -LSB and +LSB, respectively. Other solutions can be easily implemented, such as a thermometer-coded current DAC, which instead of a one-hot coded shifter would be connected to a thermometer-scale shifter module (see Section 3).

In the resistive divisor case, each quantization level $V_n$ depends on the function $f(n)$, and is defined according to:

$$ V_n = \frac{V_{\text{ref}}}{R_{\text{total}}} \times f(n), \quad n \in [0, N-1] \quad (3) $$

where $N$ is the total number of quantization levels.

Therefore, if the designer uses a simple uniform linear function such as $f(n) = n - 1/2$, with $N = 8$, the result is a resistive divisor as the one depicted in Figure 6 and a linear quantization like the one represented in Figure 8(b).

At the first sight it may seem that this kind of solution is not very efficient because the number of resistances and switches increases linearly with precision. However, with this type of design the quantization intervals can be easily customized. This is particularly interesting for the DC-DC digital control, because the control needs to be more accurate only near the target voltage. Therefore, in practice the designer can reduce the number of quantization levels while maintaining the necessary precision. For example, the designer can apply a certain function $f(n)$ which results in a quantization distribution as the one depicted in Figure 9.

3 Digital Modules Implementation

As shown in Figure 7 the proposed ADC circuitry has a digital part (Control and Shift logic) and an analog part (DAC and comparators). The digital part has two main modules: i) the shifting module, which is responsible for actually generating the output sequence of bits, and ii) the asynchronous control module, which detects and performs a new conversion. In this section we will describe the implementation details of these two modules.
3.1 Shifting Module

The proposed one-hot coded shifter module is built using a chain of token cells. A token cell is a modified flip-flop implemented as shown in Figure 10 (the respective Verilog code can be found in Appendix A.6). Each token cell requires 16 logic gates, 6 NAND, 2 OR, 5 AND, and two inverters. Moreover, it has five inputs (Inc, Dec, D0, D1, and Reset/Set) and one output (Q), which behave according to Table 1. The first two inputs (Inc and Dec) are control variables. When Inc transits from ‘0’ to ‘1’ and Dec is at ‘0’, D0 is copied to Q, when the opposite occurs, i.e., Dec transits from ‘0’ to ‘1’ and Inc is at ‘0’, D1 is copied to Q. In any other conditions the state of the cell is maintained. To guarantee the correct behavior of the token cells we implemented the XOR gate represented in red in Figure 10 using some extra logic as shown in Figure 11. This adjustment was necessary because node ‘Y’ must be stable when a rising edge arrives to node ‘X’ (“setup time”). Moreover, we have also connected the reset signal to the extra logic in order to guarantee that there are no unwanted transitions during the reset phase.

By putting this cells together in a chain as presented in Figure 12, we are able to create the expected behavior, i.e., a sequence of bits which according to the input signals “Inc” and “Dec” are shifted towards the LSB or the MSB (in this document we also use the terminologies “left” and “right” when referring to “towards the LSB” and “towards the MSB”, respectively). In detail, the design proposed in Figure 12 starts with the middle bit at high and all the other bits at low, thus creating a sequence with only one bit active at a time. This bit is then shifted back and forth according to the control. The OR gates introduced at the border cells are necessary to maintain the last bit active when the system saturates.

In Figure 13 we show an alternative thermometer-scale design. In this case all the bits in the left part of the sequence are active. Upon initialization the left half of the output word starts at high and the right half at low. When a shift occurs, either the first ‘0’ in the sequence is activated (Inc) or the rightmost ‘1’ in the sequence is

<table>
<thead>
<tr>
<th>Inc</th>
<th>Dec</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>(01)</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>0</td>
<td>(01)</td>
<td>D1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>Q (Keep)</td>
</tr>
</tbody>
</table>
deactivated (Dec) (see Table 2). The Verilog code with both implementations can be found in Appendix A.4. It is worth to note that the number of logic gates needed to implement the shift module increases linearly with the number of bits \( N \), and is given by \( \text{NumGates}(N) = 16N \).

**Fan-out analysis**

One of the issues identified in the shift module for both proposed implementations is the fan-out of the “Inc” and “Dec” signals coming from the controller. It is easy to understand from the previous description that the number of logic gates connected to these signals increases linearly with \( N \). Namely the “Inc” signal increases according to the function \( \text{NumGates}_{\text{Inc}}(N) = 3N \) and the “Dec” signal according to \( \text{NumGates}_{\text{Dec}}(N) = 2N \). In order to reduce the fan-out we have decided to implement busses before the token cells as shown by the “b” gates in Figures 12 and 13. In detail the busses are implemented in a tree structure where each bus only drives a fixed number of cells \( n_c \), except for the first and last token cells which are driven by only one buffer. Herein we have used \( n_c = 4 \) as shown in Figure 14. The total construction is fully modular and the size of the tree is directly calculated from Equations 4 and 5.

\[
\begin{align*}
\text{Levels}(N, n_c) &= \lceil \log_{n_c}(N + 2(n_c - 1)) \rceil \\
\text{Groups}(N, n_c) &= \left\lfloor \frac{N - 2}{n_c} \right\rfloor + 2
\end{align*}
\] (4) (5)

**TABLE 2**

<table>
<thead>
<tr>
<th>Inc</th>
<th>Dec</th>
<th>Tracking</th>
<th>Termometer-scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00010000</td>
<td>11110000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001000</td>
<td>11111000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000100</td>
<td>11111100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00000100</td>
<td>11111100</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00001000</td>
<td>11111000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00010000</td>
<td>11110000</td>
</tr>
</tbody>
</table>

Fig. 12. Shift module with tracking topology

Fig. 13. Shift module with thermometer-scale topology

Fig. 14. Bus tree topology
3.2 Asynchronous Control Logic

The second module asynchronously detects and controls each new conversion. In other words, when the comparators detect a significant change in the input signal, the circuit triggers a new conversion. The implementation of this module is based on typical token arbiters such as the ones used to implement petri nets \([18], [19]\), i.e., a request-acknowledge transition state machine where only one state can be active at a time. Overall, it behaves according to the transition and state diagrams presented in Figure 16. Each conversion is triggered by the input signals LSBP and LSBN. When a new conversion starts a sequence of transitions is triggered and the system goes through a group of states till it reaches a new stable state. There are two possible sequences for one conversion, either \(S_1 \rightarrow S_3 \rightarrow S_4\) or \(S_2 \rightarrow S_3 \rightarrow S_4\), according to the input signal that triggered the conversion. In particular, when a change in LSBP or LSBN is sensed by either \(S_1\) or \(S_2\), the “detection states”, and the system finished any previous conversion, a new conversion starts. At this point we have two concurrent actions, the signals Inc and Dec are sent to the shift module, and a signal is sent to transit to the next state \(S_3\), the “processing state”. During the third state the system resets states \(S_1\) and \(S_2\) and waits for the conversion to finish, i.e., waits for a new valid input from the comparators. It is important to note at this point that the waiting time is defined by a delay block as it will be explained further down. Finally, the system transits to the fourth state \(S_4\), the “waiting state”. At this point the system is again in a stable condition and a new conversion can start at any moment.

The control module logic implementation is depicted in Figure 15 and the respective Verilog code can be found in Appendix A.2. It is divided mainly in three parts: 1) the logic responsible for the transitions between control states, 2) the logic responsible for the conversion between the comparators output and the controller, and 3) a block of logic that detects when the shifter saturates.

Part “1” implements the function depicted in the diagrams in Figure 16. It requires four “C” elements or Arbiters such as the one shown in Figure 17, one per each state. Each Arbiter requires 6 logic gates, 4
NAND and 2 inverters. The Arbiter behaves according to Table 3 (the respective Verilog code can be found in Appendix A.5), i.e., when it receives a request (Req) it becomes active, when it receives an acknowledge (Ack) it becomes inactive, in any other situation it keeps its previous state. Besides the Arbiters, there is a “Delay” block which is used to adjust the time between successive conversions. This module is implemented outside of the control itself, and it can be done using an RC circuit. The function of this block will be explained in more detail in Section 4. Moreover, there is a group of buffers between the two first Arbiters and the OR gate, which are introduced just for timing purposes. The remaining glue logic implements the transitions depicted in Figure 16-b). Overall, the control module requires 35 constant logic gates plus the translation module, which depends on the comparators used. The case represented in Figure 15 requires exactly 39 logic gates.

The module corresponding to the second part was introduced in this work to detach in a simple way the analog part from the digital part. Namely, what this module does is a simple translation between the output of the comparators and the input of the control logic. This modular structure allows the comparator’s designer to transparently adapt the controller to his design without knowing how the controller is actually implemented. The designer must only be aware and respect the behavior of the controller, which is presented in Table 4.

Finally, the third part of the circuit represents an optimization which was introduced to save power by reducing the number of unnecessary signal transitions when the output bit sequence is saturated. This logic detects when the sequence is saturated to the right and to the left and disables the “Inc” and “Dec” signals accordingly, leaving only the alternative option active (i.e., “Dec” and “Inc”, respectively).

### 4 Timing Considerations

Till this point we have focused on the functional and logic description of the several modules proposed in this work. In this section we provide a detailed analysis of the timing constraints of these modules, including how to calculate the delay block and maximum frequency of an input signal.

As explained in Section 3.2 each conversion performs a sequence of state transitions. The time taken since a conversion starts till the system is again stable dictates the conversion duration $\delta t$, as shown in the diagram of Figure 18. $\delta t$ is then calculated according to:

$$
\delta t = t_{S3} + t_{S4} + t_{S1/S2} + t_e
$$

where $t_{S1/S2}$ and $t'_{S1/S2}$ represent the delays of states S1 and S2 in the best case and the worst case. The difference between the two cases is that, in the first it is considered that the whole system was in a steady state when the conversion started and in the second case it is considered that a previous conversion has just finished and some of the signals are still propagating (see Figure 22 in Appendix D).

Let us consider the times $t_{S3}$, $t_{S4}$, $t_{S1/S2}$, and $t'_{S1/S2}$ independently (see the detailed time analysis in Figure 22, Appendix D). $t_{S3}$ depends on the delay module, and is given by the following equation:

$$
t_{S3} = \max \left\{ t_{delay} + t_{NAND} + t_{NAND3} \right\}
$$

where $t_{\alpha}$ is given by:

$$
t_{\alpha} = \max \left\{ 2 \cdot t_{NAND} - (t_{NOT} + 2 \cdot t_{NAND}) \right\}
$$

The second case, $t_{S4}$, is given by the following expression:

$$
t_{S4} = 4 \cdot t_{NAND} + t_{NAND3} + t_{OR} + t_{NOT}
$$
Finally $t_{S1/S2}$ and $t_{S1/S2}'$ are given by Equation 10 and 11, respectively.

$$t_{S1/S2} = t_{NOT} + 2 \cdot t_{NAND} + t_{buffer} + t_{OR} + 2 \cdot t_{NAND} + t_{NAND3}$$ (10)

$$t_{S1/S2}' = t_{NOT} + 2 \cdot t_{NAND} + t_{buffer} + t_{OR} + 2 \cdot t_{NAND} + t_{NAND3}$$ (11)

and $t_\beta$ is given by:

$$t_\beta = t_{delay} - (t_{OR} + t_{NOT} + 2 \cdot t_{NAND})$$ (12)

According to Figure 18, to guarantee that a new conversion does not start before the actualization of the DAC, the comparators and the conversion and saturation logic, the designer must adjust the delay time. According to Equations 7, 9, and 10, the delay block must be configured such that the following equation is respected:

$$\min(t_{S1/S2}, t_{S1/S2}') + t_{S3} + t_{S4} > t_{shift} + t_{DAC} + t_{Comp} + t_{Conv.&Sat}$$ (13)

where $t_{Conv.}$ and $t_{Sat.}$ are the delays of the controller blocks' 2 and 3, respectively. The delay of block 3 is given by $t_{Sat.} = t_{NAND} + t_{Comp} + t_{DAC}$, and $t_{Comp}$ depend on the implementation. As for $t_{shift}$, is given by Equation 14 (see Figure 22 in Appendix D).

$$t_{shift} = t_{buffer} + t_{NOT} + 2 \cdot t_{AND} + t_{OR} + 2 \cdot t_{NAND3} + t_{NAND}$$ (14)

Thus, the designer should adjust the delay block according to the following:

$$\text{if } t_{DAC} + t_{Comp} + t_{Conv.} < t_{\theta} + 2 \cdot t_{OR} + 2 \cdot t_{NAND3} + 2 \cdot t_{NOT} + 10 \cdot t_{NAND} + t_{buffer} - 3 \cdot t_{AND}$$

then $t_{delay} = 0$

else $t_{delay} = t_{DAC} + t_{Comp} + t_{Conv.} + 3 \cdot t_{AND} - (t_{OR} + t_{NOT} + 8 \cdot t_{NAND} + t_{NAND3})$

end

Input Timing Constraints

To calculate the maximum supported frequency of a given input signal, i.e., the maximum frequency for which the ADC is able to follow the signal without any delay, one must find if the condition in Equation (2) is respected. Let us consider for example an input sine wave with frequency “f” and peak-to-peak amplitude “A”:

$$V_{in}(t) = \frac{A}{2} \times \sin(2\pi f \times t)$$ (15)

we can calculate the maximum frequency “$f_{max}$”, for which the ADC is able to follow the sine wave, according to Equation (16). As one can understand from the equation, optimizing $t_{delay}$ is of utmost importance, because while on the one hand we need to increase the delay, and consequently $\delta t$, in order to have a valid result between successive conversions, on the other hand, increasing $\delta t$ reduces the tracking speed of the circuit. This approach is valid for different input signals.

$$A \times 2\pi f \times \cos(2\pi f \times t) \leq q_{min}$$

$$\max(cos(2\pi f \times t)) \leq \frac{q_{min}}{A\pi f \times \delta t}$$ (16)

$$f \leq \frac{q_{min}}{A\pi f \times \delta t}$$

$$f_{max} = \frac{q_{min}}{A\pi f \times \delta t}$$

5 Simulations

In order to evaluate the architecture proposed in this work we have used the iVerilog v0.10.0 simulation and synthesis tool [20], and to obtain the waves we have used both the GTKWave v3.3.15 [21] and MatLab R2010b [22]. We have performed two types of simulations: i) an open loop simulation, i.e., a simulation without the DAC and the Comparators feedback loop, where we manually defined the LSBP and LSBN signals; and ii) a closed loop simulation, i.e., simulating the whole feedback system, for which we defined an input sine wave with different frequencies. For the second case we had to use the PLI-Verilog [23], which allowed us to program a group of functions to simulate the analog part of the circuit. The test benches used for both types of simulations are presented in Appendix B, and the PLI functions created to simulate the analog part of the circuit are presented in Appendix C. The time delays of the logic gates used in this work where obtained from Faraday Technology Corp (FTC) Verilog Simulation Library of standard cells revision 1.1. Although we have performed simulations with both the tracking and the thermometer-scaled shifting modules, herein we only present the simulations obtained with the second case. For the simulations presented here we have also considered ideal DAC and Comparators, and a delay block configured with $t_{delay} = 0$, i.e., $t_{S1/S2} = t_{S1/S2}'$. Moreover, a peak-to-peak voltage of 20mv for both $V_{in}$ and $V_{ref}$, and 16 regular quantization levels, i.e., each quantization level with $q_{min} = 1.25$mv.

The open loop simulation is shown in Figure 19. One can see the variations obtained in the output signal “O” when the input signals “LSBP” and “LSBN” change. Namely, one can see that the output remains the same when the inputs have the same logic level, i.e., {LSBN,LSBP} are at “00” or at “11”, the second case being an anomaly. Therefore, the circuit is stable even when the inputs are in a forbidden state. When the increment and decrement functions are activated the output results in a staircase effect. As explained before the delay between different steps is given by Equation 6, and with the logic gates used in this work we obtained a
maximum delay of 731ps, which means that the circuit considered is able to track a signal with a maximum slope of 1.7MV/s, or in other words, we get \( \approx 1368 \text{MS/s} \) in comparison to the 15MS/s obtained in [15]. In practice the circuit can follow for example a triangular wave with peak-to-peak amplitude of 20mV and frequency of \( \approx 85.5 \text{MHz} \).

Moreover, according to the implementation conditions described before and Equation 16, the maximum frequency of the sine wave that the circuit is able to follow is \( f_{\text{max}} \approx 27.2 \text{MHz} \). Figures 20 shows the results obtained with the closed loop simulation, using Verilog and C. The figures show simulations with input sine waves of 20MHz, 27.2MHz, 30MHz, 40MHz, 50MHz, and 100MHz, respectively. One can notice a slight delay at 40MHz, mainly when the input wave is rising, this happens because the shifting module is slightly faster when tracking down than when rising. For 50MHz it is already possible to see the delay affecting both tracking directions. Finally for 100 MHz one can observe already a filtering effect where the ADC is not anymore able to follow the input wave.

6 CONCLUSIONS

In this report we propose a fully asynchronous modular tracking analog-to-digital converter (ADC) designed for DC-DC converters with digital control. The architecture proposed had the huge advantage of only performing a conversion when the input signal crosses a given quantization level, and not every clock cycle as in typical ADC architectures, thus reducing its power consumption. Moreover, we propose two different topologies for the ADC, one tracking and one thermometer-scaled. Both topologies may be useful depending on the design of the feedback DAC. The design of the digital part of the ADC is presented in detail, including a full time analysis of the circuitry. The obtained results show that the proposed circuit is able to achieve in the best case scenario 1368MS/s in comparison to the 15MS/s obtained in previous works.

In the future we plan to integrate the methodology proposed in [13], this will improve the response of the circuit even further by increasing the number of quantization levels it may shift in one conversion.

REFERENCES

Fig. 20. Closed loop simulation with sine wave


APPENDIX

A  VERILOG MODULES

A.1  Digital Overall Logic Block

`timescale 10ps/1ps
//
// Module digital_block
//
// Created by Frederico Pratas on 18/Jan/11.
// Copyright 2011 IST/INESC-ID. All rights reserved.
//
module digital_block(O,LSBP,LSBN,Reset,DelayOut,DelayIn);
    // parameter to define the number of bits of the shift module
    parameter N=8;
    // parameter to define the number of levels of the buffers depending on the fan-out
    output [N-1:0] O;
    input LSBP;
    input LSBN;
    input Reset;
    output DelayOut;
    input DelayIn;
    wire Inc;
    wire Dec;
    controller #(N) control(Inc,Dec,LSBP,LSBN,Reset,DelayOut,DelayIn,{O[N-1],O[1:0]});
    shift #(N) tokenring(O,Inc,Dec,Reset);

initial begin
    $display("N=%d, Groups=%d, levels=%d
",N,((N-2)%4?(N-2)/4+1:(N-2)/4)+2,
        ($clog2(N+6)%2)?$clog2(N+6)/2+1:$clog2(N+6)/2));
end
endmodule

A.2  Asynchronous ADC Controller Block

`timescale 10ps/1ps
//
// Module controller
//
// Created by Frederico Pratas on 16/Jan/11.
// Copyright 2011 IST/INESC-ID. All rights reserved.
//
module controller(Inc,Dec,LSBP,LSBN,Reset,DelayOut,DelayIn,Sense);
    parameter N=8;
    // parameter to define the number of levels of the buffers depending on the fan-out
    parameter levels=($clog2(N+6)%2)?$clog2(N+6)/2+1:$clog2(N+6)/2);
    output Inc;
    output Dec;
    input LSBP;
    input LSBN;
    input Reset;
    output DelayOut;
    input DelayIn;
    input [2:0] Sense;
    genvar i;
    wire winc;
    wire wdec;
    wire delay2;
    wire [levels:0] wreq13;
    wire [levels:0] wreq23;
    wire wreq1;
    wire wreq34;
    wire wreq04, wreq4, wreq41, wreq42;
    wire wack31, wack32;
    //saturation wires
    wire wNsense1, wNsense2, wSense1;
    wire wUpper, wLower;
    assign Inc = wreq13[0];
    assign Dec = wreq23[0];
module that performs the translation of the LSBP/LSBN to the Inc/Dec signals
input_coding transmodule(winc,wdec,LSBP,LSBN);

// logic to detect the saturation of the shift module
// sense the upper limit
INVCHD sat_not0(wNsense2,Sense[2]);
AN2CHD sat_and0(wUpper,wNsense2,winc);
// sense the lower limit
INVCHD sat_not1(wNsense1,Sense[1]);
ND2CHD sat_nand0(wSense1,wNsense1,Sense[0]);
AN2CHD sat_and1(wLower,wSense1,wdec);

AN2CHD and0(wreq41,wreq4,wUpper);
AN2CHD and1(wreq42,wreq4,wLower);
OR2CHD or0(wack31,wreq34,wreq23[0]);
OR2CHD or1(wack32,wreq34,wreq23[0]);

// first state - perform shift
C_element_R element01(wreq13[0],,wreq41, wack31, Reset);
C_element_R element02(wreq23[0],,wreq42, wack32, Reset);

// These buffers are introduced for equalization of the fan-out delay in the shift module
generate
  for(i=0; i<levels; i=i+1)begin:
    BUFFER U0 (wreq13[i+1],wreq13[i]);
    BUFFER U1 (wreq23[i+1],wreq23[i]);
  end
endgenerate

OR2CHD or2(wreq3,wreq13[levels],wreq23[levels]);

// second state - wait to update DAC
C_element_R element03(wreq34,, wreq3, wreq04, Reset);

// the delay block is exterior for an easier adjustment
assign DelayOut = wreq34;
assign delay2 = DelayIn;

// third state - wait for comparators
C_element_S element04(wreq04,,delay2, wreq3, Reset);
AN2CHD and2(wreq4,wreq04,Reset);

endmodule

A.3 Encoding block
’timescale 10ps/1ps

// Module input_coding
//
// Created by Frederico Pratas on 18/Jan/11.
// Copyright 2011 IST/INESC-ID. All rights reserved.
//
// This module implements the input logic of the controller.
//
// The logic should be such that it translates the comparators’ output LSBP and LSBN
// to the Inc and Dec signals according to the following table
//
// Inc | Dec | Function
// 0 | 0  | Keep State / wait for next change
// 0 | 1  | Shift to the lower quantization level
// 1 | 0  | Shift to the upper quantization level
// 1 | 1  | Forbidden state - Undefined behaviour
//
// This module should be modified according to the comparators used

module input_coding(Inc,Dec,LSBP,LSBN);

output Inc;
output Dec;
input LSBP;
input LSBN;
wire wNinc;
wire wNdec;

// The logic implemented at this point by default is a
// simple mutual excusion to avoid the forbidden state
//
// LSBN LSBP | Inc | Dec
// 0 0  | 0  | 0
// 0 1  | 0  | 1
// 1 0 1 0
// 1 1 0 0
// INVCHD not0(wNinc,LSBP);
INVCHD not1(wNdec,LSBN);
AN2CHD andP(Inc,LSBP,wNdec);
AN2CHD andN(Dec,LSBN,wNinc);
endmodule

A.4 Tracking Block

// define TRACKING
	timescale 10ps/1ps

// Module shift

// Created by Frederico Pratas on 16/Jan/11.
// Copyright 2011 IST/INESC-ID. All rights reserved.

module shift(O,Inc,Dec,Reset);
    parameter N=8; //default
    parameter levels=function($clog2(N+6)%2?$clog2(N+6)/2+1:$clog2(N+6)/2); // default
    parameter groups=(N-2)%4?(N-2)/4+1:(N-2)/4+2;

    output [N-1:0] O;
    input Inc;
    input Dec;
    input Reset;

    wire [N-1:0] wQ;
    wire wfirst;
    wire wlast;

    wire [levels*groups:0] wDec;
    wire [levels*groups:0] wInc;

    genvar i, j, aux;

    assign O = wQ;

    // buffer composition, max fan-out = 4 (bi-quad tree)
    // first and last elements are independent
    // but are also connected to the tree for symmetry

    BUFFER buffer_dec0(wDec[0],Dec);
    BUFFER buffer_inc0(wInc[0],Inc);

    generate
        for(i=1; i<levels; i=i+1)begin:BiquadTree
            for(j=0; j<((groups$((levels-1-i)>074**(levels-1-i):1))?groups/((levels-1-i)>074**(levels-1-i):1):1)); j=j+1 begin:BufferArray
                BUFFER Udec (wDec[i*groups+j],wDec[(i-1)*groups+j/4]);
                BUFFER Uinc (wInc[i*groups+j],wInc[(i-1)*groups+j/4]);
            end
        end
    endgenerate

ifdef TRACKING
    //Tracking topology

    OR2CHD or0(wfirst,wQ[0],wQ[1]);
    token_R elementFirst(wQ[0],,1'b0,wfirst,wInc[(levels-1)*groups],wDec[(levels-1)*groups],Reset);

    generate
        for(i=1; i<N/2; i=i+1)begin:TokenRingFirstHalf
            token_R U (wQ[i],,wQ[i-1],wQ[i+1],wInc[(levels-1)*groups+(i+3)/4],wDec[(levels-1)*groups+(i+3)/4],Reset);
        end
    endgenerate

    token_S elementMiddle(wQ[N/2],,wQ[N/2-1],wQ[N/2+1],wInc[(levels-1)*groups+(N/2+3)/4],wDec[(levels-1)*groups+(N/2+3)/4],Reset);

    generate
        for(i=N/2+1; i<N-1; i=i+1)begin:TokenRingSecondHalf
            token_R U (wQ[i],,wQ[i-1],wQ[i+1],wInc[(levels-1)*groups+(i+3)/4],wDec[(levels-1)*groups+(i+3)/4],Reset);
        end
    endgenerate
endif


endgenerate

//first element (0)
token_S elementFirst(wQ[0],,1'b1,1'b1,wInc[(levels-1)*groups],wDec[(levels-1)*groups], Reset);

//elements 1 to N/2
generate
    for(i=1; i<=N/2; i=i+1)begin:TokenRingFirstHalf
        token_S U (wQ[i],,wQ[i-1],wQ[i+1],wInc[(levels-1)*groups+(i+3)/4],wDec[(levels-1)*groups+(i+3)/4], Reset);
    end
endgenerate

//elements N/2+1 to N-2
generate
    for(i=N/2+1; i<N-1; i=i+1)begin:TokenRingSecondHalf
       token_R U (wQ[i],,wQ[i-1],wQ[i+1],wInc[(levels-1)*groups+(i+3)/4],wDec[(levels-1)*groups+(i+3)/4], Reset);
    end
endgenerate

//last element (N-1)
token_R elementLast(wQ[N-1],,wQ[N-2],1'b0,wInc[levels*groups-1],wDec[levels*groups-1], Reset);
'endif
endmodule

A.5 Control element (arbiter)
	' timescale 10ps/1ps
	// Module c_element_R
	//
	// Created by Frederico Pratas on 16/Jan/11.
	// Copyright 2011 IST/INESC-ID. All rights reserved.
	//
	// Truth table
	//
	// Reset
	// 0 -> Q = 0
	// 1 -> Normal behavior
	//
	// A B Q
	//-----------------
	// 0 0 Keep (Q)
	// 0 1 Set (1)
	// 1 0 Reset (0)
	// 1 1 Reset (0)
	//

token_R(Q, nQ, Req, Ack, Reset);
th module c_element_R(Q, nQ, Req, Ack, Reset);
    output Q;
    output nQ;
    input Req;
    input Ack;
    input Reset;

    wire wset;
    wire wreset;
    wire wnReq;
    wire wnAck;
    wire wnot0;
    wire wnot1;

    INVCHD not0(wnot0, wnAck, Ack);
    INVCHD not1(wnot1, wnReq, Req);

    ND2CHD NAND0(wset, Req, wnot0);
    ND2CHD NAND1(wreset, wnReq, wnot1);

    SR_NAND_latch latch1(Q, nQ, wset, wreset, Reset);
endmodule

A.6 Token Cell
	' timescale 10ps/1ps
	// Module token_R
	//
	// Created by Frederico Pratas on 17/Jan/11.
// Copyright 2011 IST/INESC-ID. All rights reserved.
//
module token_R(Q,nQ,D0,D1,A,B,Reset);
  output Q;
  output nQ;
  input D0;
  input D1;
  input A;
  input B;
  input Reset;

  wire wset;
  wire wreset;
  wire wenable;
  wire wenable0;
  wire wenable1;
  wire wbridge;
  wire wD;
  wire wNA;
  wire wNB;
  wire wAND0;
  wire wAND1;
  INVCHD not0(wNA,A);
  INVCHD not1(wNB,B);

  //generate enable (enable=xor(A,B))
  //xor #(globals.xorDelay) xor0(wenable,A,B);
  //the logic is modified to delay the edge sensitive entry
  AN2CHD and0(wenable0,A,wNB);
  AN2CHD and1(wenable1,B,wNA);
  OR2CHD or0(wenable2,wenable0,wenable1);
  AN2CHD and2(wenable,wenable2,Reset);

  //generate D
  AN2CHD and3(wAND0,A,D0);
  AN2CHD and4(wAND1,wNA,D1);
  OR2CHD or1(wD,wAND0,wAND1);

  //top latch
  SR_NAND_latch latch1(wset,wbridge,Reset,wenable);

  //bottom latch
  SR_NAND_latch latch2(wbridge,wreset,wD,wset,wenable);

  //storing latch
  SR_NAND_latch latch0(Q,nQ,wset,wreset,Reset);
endmodule

A.7 \( SR \) Latch

`timescale 10ps/1ps
//
// Module SR_NAND_latch
//
// Created by Frederico Pratas on 15/Jan/11.
// Copyright 2011 IST/INESC-ID. All rights reserved.
//
// Truth table
//
// Reset
// 0 -> Q = 0
// 1 -> Normal behavior
//
// S R Q
//----------------
// 0 0 Restricted
// 0 1 Set (1)
// 1 0 Reset (0)
// 1 1 Keep (Q)
//
module SR_NAND_latch(Q,nQ,S,R,Reset);
  output Q;
  output nQ;
  input S;
  input R;
  input Reset;

  wire feedback0;
  wire feedback1;

  assign Q = feedback0;
  assign nQ = feedback1;
A.8 Logic Gates

ND2CHD nandS(feedback0, R, feedback1);
ND3CHD nandR(feedback1, Reset, R, feedback0);
endmodule

A.8 Logic Gates

////////////////////////////////////
// Fan-out BUFFER - should be adapted to the technology
////////////////////////////////////
'timescale 10ps/1ps
'celldefine
module BUFFER(O, I);
output O;
input I;
wire a0;
not #(1) a(a0, I);
not #(1) b(O, a0);
endmodule
'endcelldefine

FLICTC standard cell revision 1.1 --//
/////////////////////////////////////////////////////////////////////////
// NOT
/////////////////////////////////////////////////////////////////////////
'timescale 10ps/1ps
'celldefine
module INVCHD(O, I);
output O;
input I;
not #(1.40:2.15:3.59, 0.76:1.16:2.14) g1(O, I);
endmodule
'endcelldefine

// NAND
/////////////////////////////////////////////////////////////////////////
'timescale 10ps/1ps
'celldefine
module ND2CHD(O, I1, I2);
output O;
input I1, I2;
//nand g1(O, I1, I2);
not #(3.14:4.81:8.42, 3.69:5.62:9.90) not0(tmp0,I1);
not #(3.17:4.89:8.62, 4.04:6.20:10.95) not1(tmp1,I2);
or or0(O,tmp0,tmp1);
endmodule
'endcelldefine

// NAND 3
/////////////////////////////////////////////////////////////////////////
'timescale 10ps/1ps
'celldefine
module ND3CHD(O, I1, I2, I3);
output O;
input I1, I2, I3;
//nand g1(O, I1, I2, I3);
not #(1.87:2.85:4.71, 1.21:1.94:3.74) not0(tmp0,I1);
not #(2.29:3.47:5.70, 1.37:2.21:4.33) not1(tmp1,I2);
not #(2.66:4.02:6.53, 1.47:2.37:4.64) not2(tmp2,I3);
or or0(O,tmp0,tmp1,tmp2);
endmodule
'endcelldefine

// AND
/////////////////////////////////////////////////////////////////////////
'timescale 10ps/1ps
'celldefine
module AN2CHD(O, I1, I2);
output O;
input I1, I2;
//and g1(O, I1,I2);
not #(3.14:4.81:8.42, 3.69:5.62:9.90) not0(tmp0,I1);
not #(3.17:4.89:8.62, 4.04:6.20:10.95) not1(tmp1,I2);
or nor0(O,tmp0,tmp1);
endmodule
module OR2CHD(O, I1, I2);
    output O;
    input I1, I2;
    // or g1(O, I1, I2);
    not #(3.23:4.82:8.19, 4.08:6.36:11.25) not0(tmp0,I1);
    not #(2.94:4.38:7.42, 3.66:5.66:10.02) not1(tmp1,I2);
    nand nand0(O,tmp0,tmp1);
endmodule
B  TEST BENCHES

B.1 Digital block test bench

`timescale 10ps/1ps

// Module main

// Created by Frederico Pratas on 17/Jan/11.
// Copyright 2011 IST/INESC-ID. All rights reserved.

module main;

parameter NumBits = 16;

// delay of the DAC+Comparator
parameter DACdelay = 0;

reg A, LSBN, LSBP, Reset;
wire DelayOut, DelayIn, delay1;

digital_block #(NumBits) Async_ADC(,LSBP,LSBN,Reset,DelayOut,DelayIn);

// this two gates are just representative of the DAC+conversion delay, this will be implemented with an RC
not not3(delay1,DelayOut);
not #(DACdelay) not4(DelayIn,delay1);

initial begin

$dumpfile("./test.vcd");
$dumpvars(0,Async_ADC);

Reset = 1;
LSBP = 0;
LSBN = 0;
A=0;

#10
Reset = 0;

#50
Reset = 1;

#30
LSBP = 0;
LSBN = 1;

#40
LSBP = 1;
LSBN = 0;

#50
LSBP = 1;
LSBN = 1;

#200
LSBP = 0;
LSBN = 0;

#400
LSBP = 1;
LSBN = 0;

#800
LSBP = 0;
LSBN = 1;

#10 $finish;

end
endmodule // main

B.2 Closed loop test bench

`timescale 10ps/1ps
module main;

parameter NumBits = 16;
// delay of the DAC+Comparator
parameter DACdelay = 0;
real Vin;
reg Reset;
reg LSBP, LSBN;
wire [NumBits-1:0] Digital;
wire DelayOut, DelayIn, delay1;
digital_block #(NumBits) Async_ADC(Digital,LSBP,LSBN,Reset,DelayOut,DelayIn);

// these two gates are just representative of the DAC+conversion delay, this will be implemented with an RC
not not3(delay1,DelayOut);
not #(DACdelay) not4(DelayIn,delay1);
always @(Reset,Digital)
$DAC_Comp(LSBP,LSBN,Digital,Vin);
initial begin
$display("Starting simulation\n");
$dumpfile("test.vcd");
$dumpvars(0,Vin,Async_ADC);
Reset = 1;
#10
Reset = 0;
#500
Reset = 1;
#10000
$display("Finished simulation\n");
$finish;
end
endmodule // main
C CLOSED LOOP SIMULATION USING VERILOG PLI

C.1 PLI functions to interact with the verilog simulation

// C.CLOSED LOOP SIMULATION USING VERILOG PLI
//
// Copyright 2011 IST/INESC-ID. All rights reserved.
//
// Define DAC_COMP_ARGS 4
#define DAC_COMP_ARGS 4

#include <vpi_user.h>
#include <stdlib.h>
#include "dac_function.h"

typedef struct User_Data {
    long time_init; // initial time in ps
    vpiHandle obj1_h; // LSBP
    vpiHandle obj2_h; // LSBN
    vpiHandle obj3_h; // Digital
    vpiHandle obj4_h; // vin
    FILE *output;
} User_Data_s, *User_Data_p;

int getbit(PLI_INT32 word, int bit_num)
{
    int mask;
    mask = 0x00000001 << bit_num;
    return((word & mask)? 1:0); /* 1 == TRUE, 0 == FALSE */
}

char get_4state_val(int aval, int bval)
{
    if (!bval && !aval) return('0');
    else if (!bval && aval) return('1');
    else if ( bval && !aval) return('z');
    else return('x');
}

PLI_INT32 DAC_EndOfSim(p_cb_data cb_data)
{
    vpiHandle systf_handle;
    User_Data_p vector_data; // pointer to a User_Data_s structure
    systf_handle = (vpiHandle)cb_data->user_data;
    vector_data = (User_Data_p)vpi_get_userdata(systf_handle);
    fclose(vector_data->output);
    vpi_printf("Closed output file\n");
    return(0);
}

PLI_INT32 DAC_StartOfSim(p_cb_data cb_data)
{
    vpiHandle systf_handle, arg_itr, arg1_h, arg2_h, arg3_h, arg4_h;
    s_vpi_time time_current;
    long time;
    User_Data_p vector_data; // pointer to a User_Data_s structure
    vector_data = (User_Data_p)malloc(sizeof(User_Data_s));
    // retrieve system task handle from user data
    systf_handle = (vpiHandle)cb_data->user_data;
    // get argument handles (compiletf already verified only 2 args)
    arg_itr = vpi_iterate(vpiArgument, systf_handle);
    arg1_h = vpi_scan(arg_itr); // LSBP
    arg2_h = vpi_scan(arg_itr); // LSBN
    arg3_h = vpi_scan(arg_itr); // Digital
    arg4_h = vpi_scan(arg_itr); // vin
vpi_free_object(arg_itr); // free iterator -- did not scan to null

// get initial time in ps
time_current.type = vpiSimTime;
vpi_get_time(NULL, &time_current);
time = time_current.high;
time = (time<<32)+time_current.low;
vpi_printf("Initial time: %ld\n", time);

// store file pointer and tfarg2_h in work area for this instance
vector_data->time_init = time;
vector_data->obj1_h = arg1_h;
vector_data->obj2_h = arg2_h;
vector_data->obj3_h = arg3_h;
vector_data->obj4_h = arg4_h;
vector_data->output = fopen("./results.txt","w");
vpi_put_userdata(systf_handle, (void *)vector_data);
return(0);

PLI_INT32 DAC_Comp_compiletf(PLI_BYTE8*user_data)
{
    s_cb_data cb_data_s;
    vpiHandle systf_handle, arg_iterator, arg_handle, cb_handle;
    PLI_INT32 arg_type;

    // obtain a handle to the system task instance
    systf_handle = vpi_handle(vpiSysTfCall, NULL);
    if (systf_handle == NULL) {
        vpi_printf("ERROR: $DAC_Comp failed to obtain systf handle\n");
        vpi_control(vpiFinish,0); // abort simulation
        return(0);
    }

    // obtain handles to system task arguments
    arg_iterator = vpi_iterate(vpiArgument, systf_handle);
    if (arg_iterator == NULL) {
        vpi_printf("ERROR: $DAC_Comp requires %d arguments; has none!\n", DAC_COMP_ARGS);
        vpi_control(vpiFinish,0); // abort simulation
        return(0);
    }

    // get handle for first argument
    arg_handle = vpi_scan(arg_iterator);
    if (arg_handle == NULL) {
        vpi_printf("ERROR: $DAC_Comp requires %d argument; has none!\n", DAC_COMP_ARGS);
        vpi_free_object(arg_iterator); // free iterator memory
        vpi_control(vpiFinish,0); // abort simulation
        return(0);
    }

    // get type for first argument
    arg_type = vpi_get(vpiType, arg_handle);
    if ((arg_type != vpiReg) && (tfarg_type != vpiIntegerVar) && (tfarg_type != vpiConstant)) {
        vpi_printf("ERROR: $DAC_Comp arg %d must be a reg\n", 1);
        vpi_free_object(arg_iterator); // free iterator memory
        vpi_control(vpiFinish,0); // abort simulation
        return(0);
    }

    // get handle for second argument
    arg_handle = vpi_scan(arg_iterator);
    if (arg_handle == NULL) {
        vpi_printf("ERROR: $DAC_Comp requires %d argument; has none!\n", DAC_COMP_ARGS);
        vpi_free_object(arg_iterator); // free iterator memory
        vpi_control(vpiFinish,0); // abort simulation
        return(0);
    }

    // get type for second argument
    arg_type = vpi_get(vpiType, arg_handle);
    if ((arg_type != vpiReg) && (tfarg_type != vpiIntegerVar) && (tfarg_type != vpiConstant)) {
        vpi_printf("ERROR: $DAC_Comp arg %d must be a reg\n", 2);
        vpi_free_object(arg_iterator); // free iterator memory
        vpi_control(vpiFinish,0); // abort simulation
        return(0);
    }

    // get handle for third argument
    arg_handle = vpi_scan(arg_iterator);
    if (arg_handle == NULL) {
        vpi_printf("ERROR: $DAC_Comp requires %d argument; has none!\n", DAC_COMP_ARGS);
        vpi_free_object(arg_iterator); // free iterator memory
    }
vpi_control(vpiFinish,0); // abort simulation
return(0);
}

// get type for third argument
arg_type = vpi_get(vpiType, arg_handle);
if (!vpiNet) {
    vpi_printf("ERRORB: $DAC_Comp arg %d must be a vector\n",3);
    vpi_free_object(arg_iterator); // free iterator memory
    vpi_control(vpiFinish,0); // abort simulation
    return(0);
}

// get handle for fourth argument
arg_handle = vpi_scan(arg_iterator);
if (arg_handle == NULL) {
    vpi_printf("ERROR: $DAC_Comp requires %d argument; has none!\n",DAC_COMP_ARGS);
    vpi_free_object(arg_iterator); // free iterator memory
    vpi_control(vpiFinish,0); // abort simulation
    return(0);
}

// get type for fourth argument
arg_type = vpi_get(vpiType, arg_handle);
if (!vpiRealVar) {
    vpi_printf("ERRORB: $DAC_Comp arg %d must be a real\n",4);
    vpi_free_object(arg_iterator); // free iterator memory
    vpi_control(vpiFinish,0); // abort simulation
    return(0);
}

// check that there are no more system task arguments
arg_handle = vpi_scan(arg_iterator);
if (arg_handle != NULL) {
    vpi_printf("ERROR: $DAC_Comp requires %d arguments; has too many\n",DAC_COMP_ARGS);
    vpi_free_object(arg_iterator); // free iterator memory
    vpi_control(vpiFinish,0); // abort simulation
    return(0);
}

// No syntax errors, setup a callback for start of simulation
cb_data.s.reason = cbStartOfSimulation;
cb_data.s.cb_rtn = DAC_StartOfSim;
cb_data.s.obj = NULL;
cb_data.s.time = NULL;
cb_data.s.value = NULL;
// use user_data to pass systf_h to simulation callback so that the callback can access the system task arguments
    cb_data.s.user_data = (PLI_BYTE8 *)systf_handle;
    cb_handle = vpi_register_cb(&cb_data.s);

// Setup a callback for end of simulation
    cb_data.s.reason = cbEndOfSimulation;
    cb_data.s.cb_rtn = DAC_EndOfSim;
    cb_data.s.obj = NULL;
    cb_data.s.time = NULL;
    cb_data.s.value = NULL;
// use user_data to pass systf_h to simulation callback so that the callback can access the system task arguments
    cb_data.s.user_data = (PLI_BYTE8 *)systf_handle;
    cb_handle = vpi_register_cb(&cb_data.s);

vpi_free_object(cb_handle); // free callback handle -- don't need it
return(0);
}

PLI_INT32 DAC_Comp_event(p_cb_data cb_data) {
    vpiHandle systf_handle, cb_handle;
    s_vpi_time time_current, time_s;
    s_vpi_value value[DAC_COMP_ARGS];
    User_Data_p vector_data;
    s_cb_data data_s;
    int bvalbit, avalbit, vector_size;
    int j, num_elements, bit_num;
    double vin;
    long int current, time_init, time_event;
    int LSBP, LSBN;
    char *binarray;

    // retrieve system task handle from user_data
    systf_handle = (vpiHandle)cb_data->user_data;

    // get ReadVecData pointer from work area for this task instance
// the data in the work area was loaded at the start of simulation
vector_data = (User_Data_p)vpi_get_userdata(systf_handle);
time_init = vector_data->time_init;

// get current time in ps
time_current.type = vpiSimTime;
vpi_get_time(NULL,&time_current);
current = time_current.high;
current = (current<<32)+time_current.low;
vpi_printf("Current time: %ld\n",current);

// third element
value[2].format = vpiVectorVal; // read value as a string
vpi_printf("Signal %s ", vpi_get_str(vpiFullName, vector_data->obj3_h));
vector_size = vpi_get(vpiSize, vector_data->obj3_h);
binarray = (char*)malloc(sizeof(char)*vector_size);
num_elements = ((vector_size - 1) / 32 + 1);
vpi_printf("has the value:\n");
// number of 32bit arrays
for (j=0; j<num_elements; j++) {
    // number of elements in the array
    for (bit_num=0; bit_num<=31; bit_num++) {
        avalbit=getbit(value[2].value.vector[j].aval, bit_num);
bvalbit=getbit(value[2].value.vector[j].bval, bit_num);
        binarray[j*32+bit_num]=get_4state_val(avalbit, bvalbit);
vpi_printf(" bit[\%2d] aval/bval = %d/%d 4-state value = %c\n", (j*32+bit_num), avalbit, bvalbit, binarray[j*32+bit_num]);
        // quit when reach last bit of Verilog vector
        if ((j*32+bit_num) == vector_size-1) break;
    }
}
// calculate vin
vin = vin_function(current-time_init);
// calculate LSBN, LSBP, and vin
time_event = Analog_function(vector_size,binarray,&LSBP,&LSBN,vin,time_init,current,vector_data->output);
if(time_event != 0) {
    // convert to relative delay
time_event = time_event - current;
time_s.type = vpiSimTime;
time_s.high = (PLI_INT32)(time_event>>32);
time_s.low = (PLI_INT32)time_event;
    // schedule callback to this routine when time for next event
data_s.reason= cbAfterDelay;
data_s.cb_rtn= DAC_Comp_event;
data_s.obj= systf_handle; // object required for scaled delays
data_s.time= &time_s;
data_s.user_data= (PLI_BYTE8 *)systf_handle;
    cb_handle = vpi_register_cb(&data_s);
    if (vpi_chk_error(NULL))
        vpi_printf("An error occurred registering DAC_Comp_event callback\n");
    else
        vpi_free_object(cb_handle); // dont need callback handle
}

// write results
// write first element LSBP
value[0].format = vpiIntVal; // read value as a string
value[0].value.integer = LSBP;
vpi_put_value(vector_data->obj1_h, &value[0], NULL, vpiNoDelay);

// write second element LSBN
value[1].format = vpiIntVal; // read value as a string
value[1].value.integer = LSBN;
vpi_put_value(vector_data->obj2_h, &value[1], NULL, vpiNoDelay);

// write result to the fifth system task argument
value[3].format = vpiRealVal;
value[3].value.real = vin;
vpi_put_value(vector_data->obj4_h, &value[3], NULL, vpiNoDelay);

free(binarray);
return(0);

PLI_INT32 DAC_Comp_calltf(PLI_BYTE8*user_data) {

vpiHandle systf_handle, cb_handle;
svpi_time time_current;
s_cb_data data_s;

// obtain a handle to the system task instance
systf_handle = vpi_handle(vpiSysTfCall, NULL);

// setup immediate callback to event routine
// get current time in ps
time_current.type = vpiSimTime;
vpi_get_time(NULL,&time_current);
data_s.reason = cbReadWriteSynch;
data_s.cb_rtn = DAC_Comp_event;
data_s.obj = NULL;
data_s.time = &time_current;
data_s.value = NULL;
data_s.user_data = (PLI_BYTE8 *)systf_handle;
cb_handle = vpi_register_cb(&data_s);
vpi_free_object(cb_handle); // dont need callback handle
return(0);
}

void DAC_Comp_register()
{
    s_vpi_systf_data tf_data;
tf_data.type = vpiSysTask;
tf_data.sysfunctype = 0;
tf_data.tfname = "DAC_Comp";
tf_data.calltf = DAC_Comp_calltf;
tf_data.compiletf = DAC_Comp_compiletf;
tf_data.sizef = NULL;
tf_data.user_data = NULL;
vpi_register_systf(&tf_data);
return;
}

void (*vlog_startup_routines[])() = {
    DAC_Comp_register,
    0
};

C.2 Functions used to emulate the DAC and the Comparator

// Include file dac_function.c
// // Created by Frederico Pratas on 17/Jan/11.
// // Copyright 2011 IST/INESC-ID. All rights reserved.
// #include <stdio.h>
#include <math.h>
#include "dac_function.h"
#define MIN(A,B) ((A)<(B) ? (A) : (B))

// DAC
void dac_calculate(int size, char *bit_array, double *vref_LSBP, double *vref_LSBN){
    int i;
    *vref_LSBP = 0;
    *vref_LSBN = 0;
    for(i=size-1;i>=0;i--){
        if(bit_array[i]=='1'{
            *vref_LSBP = ((double)(VREF)/(double)size)*(i+1);
            *vref_LSBN = ((double)(VREF)/(double)size)*i;
            break;
        }
    }
}

// general DAC+Comparator function (simulates the analog circuitry)
long int Analog_function(int size, char *bit_array, int *LSBP, int *LSBN, double vin, \
    long time_init, long current, FILE *output){
    double vref_LSBP, vref_LSBN;
    int i;
double time_next0, time_next1, time_next2, time_next3;
    long time_next;
// DAC - calculate vref
dac_calculate(size, bit_array,&vref_LSBP,&vref_LSBN);

// comparator
if(vref_LSBP < vin) *LSBP=1;
else *LSBP=0;
if(vref_LSBN > vin) *LSBN=1;
else *LSBN=0;
printf("Vin %f, VrefP: %f, VrefN: %f\n",vin,vref_LSBP,vref_LSBN);
printf("LSBN=%d, LSBP=%d\n",*LSBN,*LSBP);
printf("current=%f\n",(double)current);
fprintf(output,%"f %f %f\n",(double)current,vin,vref_LSBN);

// if the conversion is stable then predict and schedule next event
if(*LSBN==0) && (*LSBP==0){
  if(vin==vref_LSBN || vin==vref_LSBP){
    time_next = current + 1;
  }else{
    
    //LSBP
    for(i=0;i++;)
      time_next0=|(asin((vref_LSBP-(double)OFFSET)/(double)AMPLITUDE)+2*i*3.1415926535897932384626433832795)\n        +1000000)/(double)FREQUENCY*2*3.1415926535897932384626433832795)+time_init;
    if(time_next0>(double)current)break;
  }
  for(i=0;i++;)
    time_next1=(3.1415926535897932384626433832795-asin((vref_LSBP-(double)OFFSET)/(double)AMPLITUDE)\n      +2*i*3.1415926535897932384626433832795)*1000000)/(double)FREQUENCY*2*3.1415926535897932384626433832795)\n      +time_init;
    if(time_next1>(double)current)break;
}

//LSBN
for(i=0;i++;)
  for(i=0;i++;)
    time_next2=|(asin((vref_LSBN-(double)OFFSET)/(double)AMPLITUDE)+2*i*3.1415926535897932384626433832795)\n      +1000000)/(double)FREQUENCY*2*3.1415926535897932384626433832795)+time_init;
  if(time_next2>(double)current)break;
}
for(i=0;i++;)
  for(i=0;i++;)
    time_next3=(3.1415926535897932384626433832795-asin((vref_LSBN-(double)OFFSET)/(double)AMPLITUDE)\n      +2*i*3.1415926535897932384626433832795)*1000000)/(double)FREQUENCY*2*3.1415926535897932384626433832795)\n      +time_init;
    if(time_next3>(double)current)break;
}

  time_next0 = MIN(time_next0,time_next1);
  time_next1 = MIN(time_next2,time_next3);
  printf("%f %f %f %f\n",time_next0,time_next1,time_next2,time_next3);
  time_next0 = MIN(time_next0,time_next1);
  time_next = (long int)time_next0;
  time_next = time_next + (((time_next0-time_next)>0) ? 1 : 0);
}
printf("Next: %ld\n",time_next);
return time_next;
}
return 0;

// simulates the input signal
double vin_function(long time_instant){
  float sine;
  printf("Frequency: %f MHz\n",(double)FREQUENCY);
  printf("Amplitude: %f mv\n",(double)AMPLITUDE);
  printf("OFFSET: %f mv\n",(double)OFFSET);
  sine = OFFSET+AMPLITUDE*sin(2*3.1415926535897932384626433832795+FREQUENCY*((double)(time_instant))/1000000));
  return sine;
}

C.3 Include file with wave characteristics

//
// Include file dac_function.h
/characteristic of the DAC
#define VREF 20 //mv
// characteristics of the sine input signal
#define FREQUENCY 15 //MHz
#define AMPLITUDE 10 //mv
#define OFFSET 10 //mv

long int Analog_function(int size, char *bit_array, int *LSBP, int *LSBN, double vin,
                           long time_init, long current, FILE *output);
double vin_function(long time_instant);
Fig. 21. Full System Architecture
Fig. 22. Conversion time detail