Digital Implementation of a Mismatch-Shaping Successive-Approximation ADC

by

Matthew T. Coe

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented October 15, 2001
Commencement June 2002

ACKNOWLEDGMENT

This thesis acknowledges the following people for their support and assistance:

Dr. Un-Ku Moon

Dr. John T. Stonick

Mr. José Silva

Mr. Roger Traylor

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

In loving memory of my Grandmother, Maxine Gaines

DIGITAL IMPLEMENTATION OF A MISMATCH-SHAPING SUCCESSIVE-APPROXIMATION ADC

1. INTRODUCTION

The performance of analog integrated circuits is generally limited to the matching accuracy of their components. Component matching accuracy can be improved by several methods including careful layout practices using unit-sized elements and common centroid layout, using ratio-independent circuits, laser trimming of components, and various calibration schemes. A recent approach to systematically solving this problem is to use digital logic to perform a shuffling of nominally matched components so that the noise introduced by component mismatch can be controlled. The mismatch noise is filtered to reduce noise energy within the signal band and move it to higher frequencies where it can be removed. This mismatch-shaping approach has been used successfully for the internal digital-to-analog converters (DAC) of delta-sigma analog-to-digital converters (ADC) and DACs, as well as for pipelined and cyclic ADCs. In this thesis we will extend the mismatch-shaping technique to a simple successive-approximation ADC.

1.1 Problem Definition

The performance of data converters is often limited to the matching accuracy of their components. Improving the matching accuracy, while possible, is expensive or requires extra chip area. We would like a circuit that is less sensitive to component matching accuracy.

1.2 Statement of Purpose

The purpose of this research is to improve the performance of a simple data converter by applying mismatch-shaping techniques. A design for a simple mismatch-shaped successive-approximation ADC for audio applications will be presented.

2. REVIEW OF RELATED LITERATURE

This chapter will present some of the previously published solutions that deal with component mismatch in data converters. The ideas put forth in these papers mostly deal with improving linearity in DACs, but these DACs are used internally in many different data converter designs.

2.1 Dynamic Element Matching

A recent approach to improving data converter performance is to perform a digitally controlled shuffling of nominally matched components. The goal of this operation is to filter the noise introduced at the output in such a way that the energy in the signal band is reduced. These designs make use of oversampling to move mismatch noise out of the signal band so that it can be removed by filtering. The drawback of many of these solutions is that they require a large number of unit elements to be used in the shuffling process. This results in increased chip areas and hence fabrication costs. Some of the dynamic element matching schemes include data weighted averaging [1], individual level averaging [2], data-directed scrambling [3], segmented data-directed scrambling [4], vector selection [5], mismatch shaping pipeline elements [6], and mismatch shaping switching [7][8].

A similar approach is used in [9]. It this case, a string of current-steering devices is cascaded to construct a binary weighted array. The outputs of each current-steering device can be swapped to implement noise shaping.

2.2 Compensative Switching

The goal of compensative switching as proposed in [10],[11], is to perform a digitally controlled shuffling of two nominally matched capacitors such that the mismatch error is minimized or cancelled. These methods require complex logic as well as an accurate analog addition, which is extremely difficult to implement.

2.3 Ratio-Independent Methods

A technique for analog to digital conversion that does not rely on component matching is presented in [12]. A ratio-independent multiplication scheme is presented that should allow algorithmic A/D operation that is not effected by element mismatch. This solution requires many clock cycles to perform each multiplication and suffers in the presence of capacitor non-linearity.

3. MISMATCH-SHAPING SUCCESSIVE-APPROXIMATION ADC

3.1 Introduction

The concept of compensative switching to perform mismatch shaping has been successfully applied to a variety of DACs, including those internal to delta-sigma ADCs, as well as to pipeline ADCs. By reversing the switching sequence selection algorithm, this concept may be applied to successive-approximation ADCs. The operation of a successive-approximation ADC requires that the MSB (most significant bit) be determined first and the LSB (least significant bit) last. Since the following bits are undetermined during each switching decision, the resulting error cannot be well controlled. This is an inherent disadvantage for ADCs when compared to any DAC which has the entire data word at hand for each conversion period. In the case of successive-approximation ADCs, the total error made is bound to be more irregular (resulting in nonlinear quantization) and larger (resulting in a higher rms noise).

3.2 Successive-Approximation ADC Basics

An ADC allows us to represent an analog input signal (typically voltage or current) as a digital word. This provides a means for digital logic to be used to process observed analog phenomena. One particular area of interest is the digital encoding of audio signals. With the wide spread use of consumer digital audio products comes a need for high-quality yet inexpensive data converters. A wide variety of ADC architectures are regularly employed in various applications.

Four of the many figures of merit used to evaluate ADC implementations are resolution, linearity, bandwidth, and conversion time. Resolution refers to the total number of distinct states that a converter can represent. This metric is expressed in bits. An N-bit ADC output word corresponds to 2^N digital levels. In consumer audio applications, resolutions of 16 to 24 bits are common. Linearity is the measure of how accurately the actual output of a data converter matches the theoretical output. Linearity is usually specified as an effective number of bits (ENOB), percentage of full-scale value, or some fraction of a least significant bit [13]. Bandwidth and conversion time, in reference to data converters, refers to how quickly a given design can carry out a conversion. Faster conversions mean wider bandwidth signals can be converted. The relationship between conversion time and bandwidth is given by the Nyquist criterion, which states that a band-limited signal must be sampled more than twice per period to avoid distortion due to aliasing. Therefore, for a Nyquist rate converter, the bandwidth is half of the sample rate. To be useful for consumer digital audio applications, a data converter must be able to represent signals ranging from 0 to 20 kHz. The table below groups some common ADC architectures into groups based on typical operating bandwidths. Audio signals fall into the medium bandwidth category.

Narrow Bandwidth	Medium Bandwidth	Wide Bandwidth
Dual-slope	Successive-approximation Algorithmic (Cyclic) Oversampling	Flash Pipeline Interpolating Folding Subranging

Bandwidth Comparison of Several ADC Architectures

A signal is said to be oversampled if a band-limited signal is converted using a sample rate higher than the Nyquist rate. Oversampling ratio (OSR) is defined as

$$OSR = \frac{f_s}{2f_0} \tag{3.1}$$

where f_0 is the band limit of the signal of interest and f_s is the sampling frequency. The advantage of oversampling is that quantization and mismatch noise are distributed through a wider frequency range than they would be in a Nyquist rate converter. After conversion, the extra bandwidth can be removed by filtering and decimation which eliminates the quantization and mismatch noise which is above f_0. Oversampling allows us to take advantage of mismatch-shaping. Oversampling also saves die area by reducing kT/C noise on sampling and integrating capacitors, which allows the use of smaller capacitors.

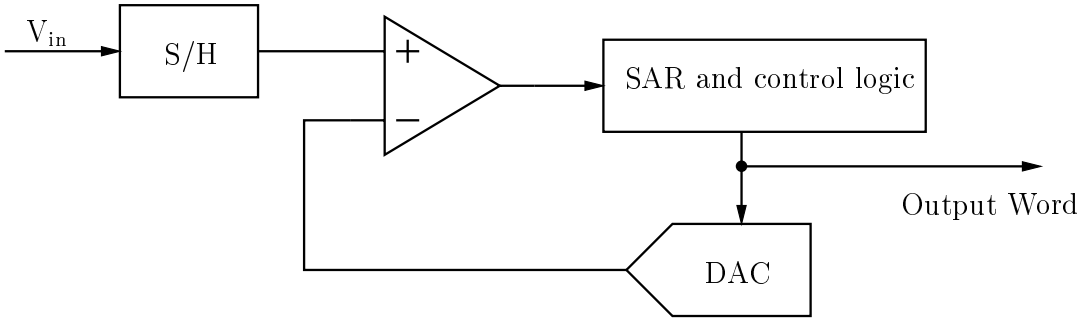

FIGURE 3.1: Basic successive-approximation ADC.

Successive-approximation ADCs employ the binary search algorithm in order to find the digital code that most closely matches the analog input voltage. To obtain an N bit digital word, the binary search algorithm must be carried out N times, requiring N clock cycles. Notice that the input must also be sampled and held, so a complete conversion will require N+1 clock cycles. Successive-approximation data converters are

attractive because they provide medium speed and accuracy without complex circuitry. A block diagram of a DAC-based successive-approximation ADC is shown in Figure 3.1. The successive-approximation register (SAR) holds the results of the N comparisons. The DAC provides an analog estimation of the value stored in the SAR to be used by the comparator in the next clock cycle. The accuracy of the internal DAC is typically the limiting factor for the accuracy of the overall converter.

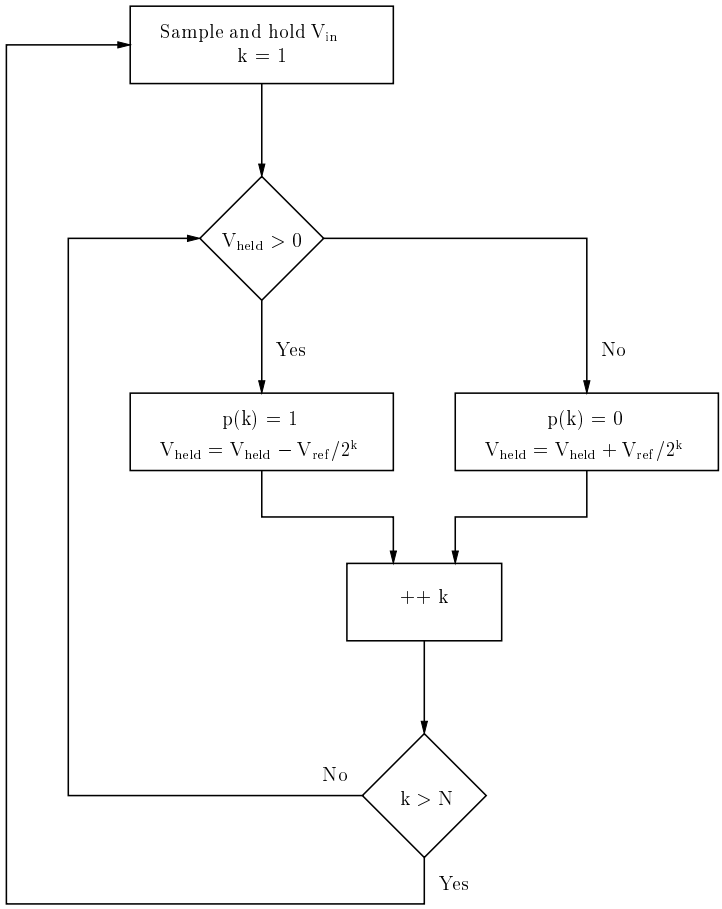

FIGURE 3.2: Flow graph for an N-bit charge-redistribution successive-approximation ADC.

The particular type of successive-approximation converter of most interest here is a charge-redistribution ADC. Instead of comparing the input voltage to a reference voltage,

this architecture operates by comparing the difference of the input value and the DAC output value to ground. Figure 3.2 shows a flow chart describing the conversion algorithm as it would be performed by single-ended circuitry.

In order to build a highly accurate successive-approximation ADC, we require an equally accurate DAC. Ideally, we would also like this DAC to occupy a minimal chip area and consume little power. The simple, pseudo-passive DAC shown in Figure 3.3 was described by Suarez et al. in [14]. The DAC is composed of only two capacitors, a voltage reference, and a few switches. This DAC functions by charging C_1 to V_{ref} or 0 depending on the incoming bits $x(n,k)$ and sharing the charge between C_1 and C_2. N clock cycles are required to convert an N-bit word $x(n)$ into an analog voltage $y(n)$. The linearity of this circuit is primarily limited by the matching accuracy of the capacitors. Methods for reducing or eliminating this nonlinearity have been described in many recent papers [7] [8] [11] [10].

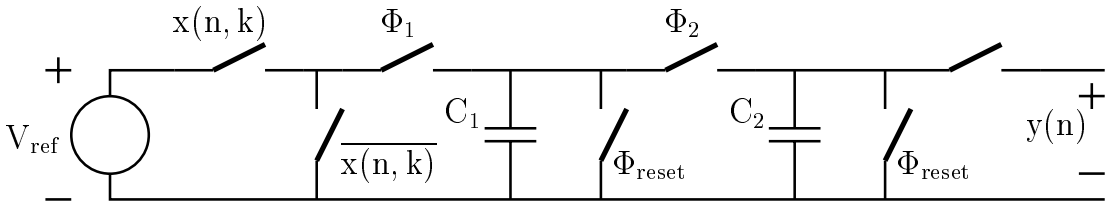

FIGURE 3.3: Basic two-capacitor charge-redistribution DAC topology.

Unfortunately, this pseudo-passive DAC determines the LSB first and the MSB last. We will use a similar topology that is somewhat less efficient in terms of area and power dissipation.

In both of the following examples the circuits shown are simplified by omitting the necessary opamp offset and gain compensation circuits.

The first example of an MSB-first DAC, which we will refer to as DAC-1, is shown in Figure 3.4. DAC-1 operates by initially precharging C_1 to V_{ref}, and discharging C_2 and C_{int} during a precharge cycle (Φ_0). In the first

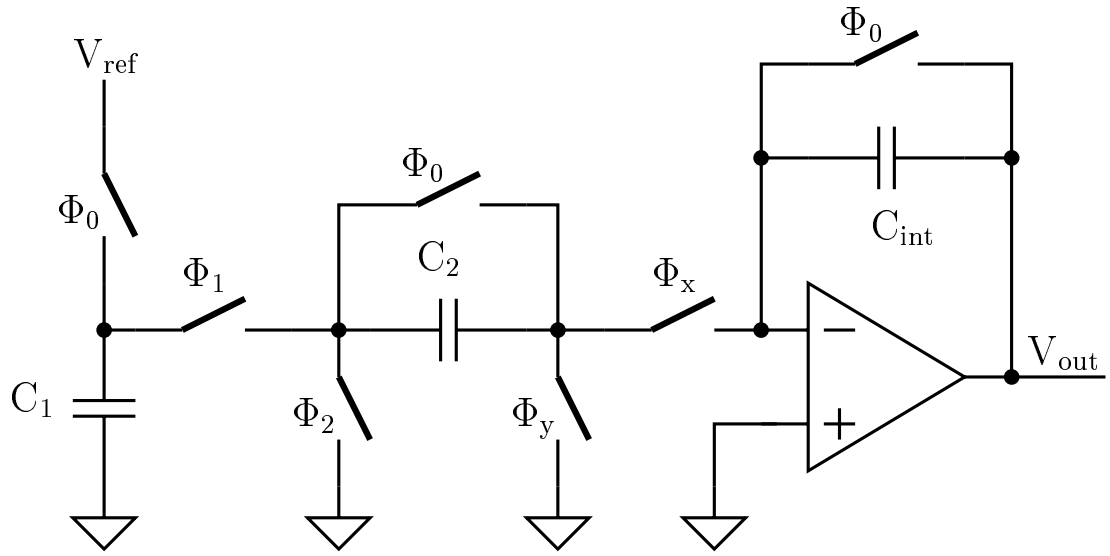

FIGURE 3.4: MSB-first DAC (DAC-1).

phase (Φ_1) of the first clock cycle, capacitors C_1 and C_2 share the charge that was stored on C_1 during the precharge cycle. At the same time (Φ_1), a charge equal and opposite to that on C_2 is dumped either onto the integrating capacitor, C_{int}, or to ground, depending on the incoming data. When the data bit is a 1, Φ_x is closed and the charge is dumped to the integrator. When the data bit is a 0, Φ_y is closed and the charge is dumped to ground. During the second phase, (Φ_2), capacitor C_2 is discharged, while the charge on capacitor C_1, which is now half of what it had been previously, is preserved. For an N-bit converter, the same set of operations is carried out N times requiring N clock cycles (2N phases).

For this specific example, the output range is 0 to $-V_{ref}$.

The second example of an MSB-first DAC, which will be referred to as DAC-2, is shown in Figure 3.5.

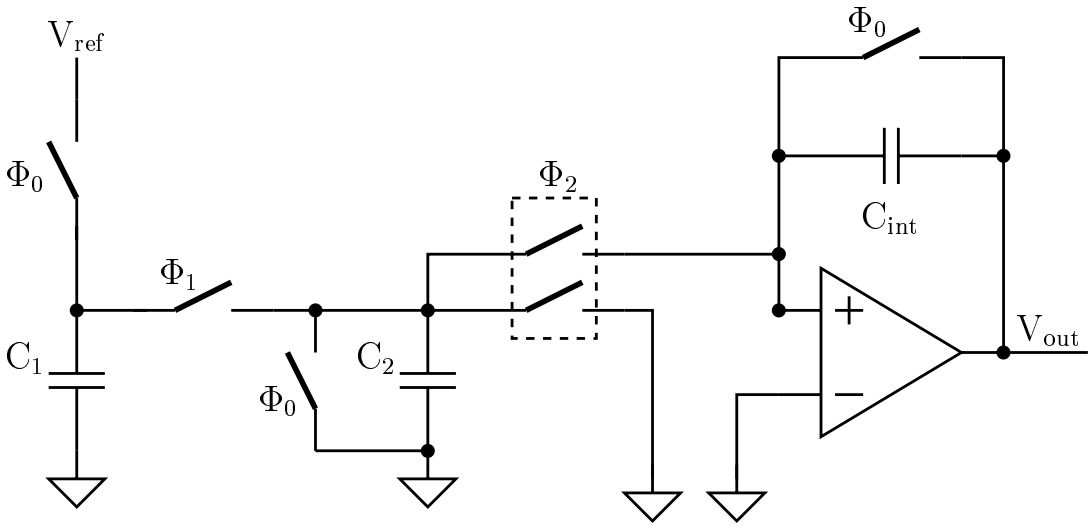

FIGURE 3.5: MSB-first DAC (DAC-2).

The precharge cycle (Φ_0) is identical to that in DAC-1. In the first phase (Φ_1) of the first clock cycle, capacitors C_1 and C_2 share the charge that was stored on C_1 during the precharge cycle. During the second phase (Φ_2) of the first clock cycle, the charge on C_2 is dumped either onto the integrating capacitor, C_{int}, or to ground, depending on the incoming data. When the data bit is a 1, the charge is dumped to the integrator. When the data bit is a 0, the charge is dumped to ground. During the second phase, (Φ_2), capacitor C_2 is discharged, while the charge on capacitor C_1, which is now half of what it had been previously, is preserved. Again, an N-bit conversion requires N clock cycles (2N phases). For this example, the output range is 0 to $+V_{ref}$.

3.3 Mismatch Shaping

In order to shape (i.e. high-pass filter) the error resulting from the capacitor mismatch in the context of a successive-approximation ADC, we need to be able to add or extract error charge from the integration capacitor. Either of the two MSB-first DAC elements described in the preceding section can be adapted to this purpose by adding a few switches. The conceptual schematic of an ADC based on the DAC-2 architecture which allows compensative switching is shown in Figure 3.6. After the first clock cycle, the negative terminal of the comparator will be at the potential $V_{ref}/2$, and depending on the sampled input voltage, the comparator decision will control whether to add or subtract charge from the integrating capacitor in the next clock cycle. This process is repeated for the N cycles during each conversion period. The capacitor mismatch error, which directly results in the inaccurate amount of charge that is either added or subtracted, is controlled by the digital logic block that interchanges the roles of capacitors C_1 and C_2.

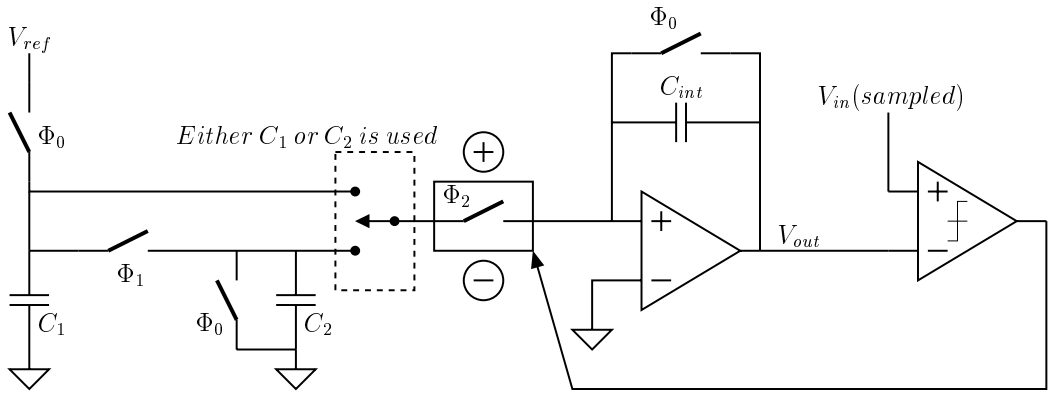

FIGURE 3.6: Conceptual implementation of DAC-2 with C_1 or C_2 and add-or-subtract operations.

We must be able to describe the capacitor mismatch error to successfully control it.

Assuming that C_1 is slightly larger than C_2 we can normalize the capacitance values to $1+\alpha$ and $1-\alpha$, respectively. We can now determine the amount of error charge that is injected into the integrating capacitor in terms of α. The error map shown in Figure 3.7 describes the mismatch errors resulting in each clock cycle in terms of alpha. The error quantity is approximated to the first order (α^2, α^3, ... terms are ignored).

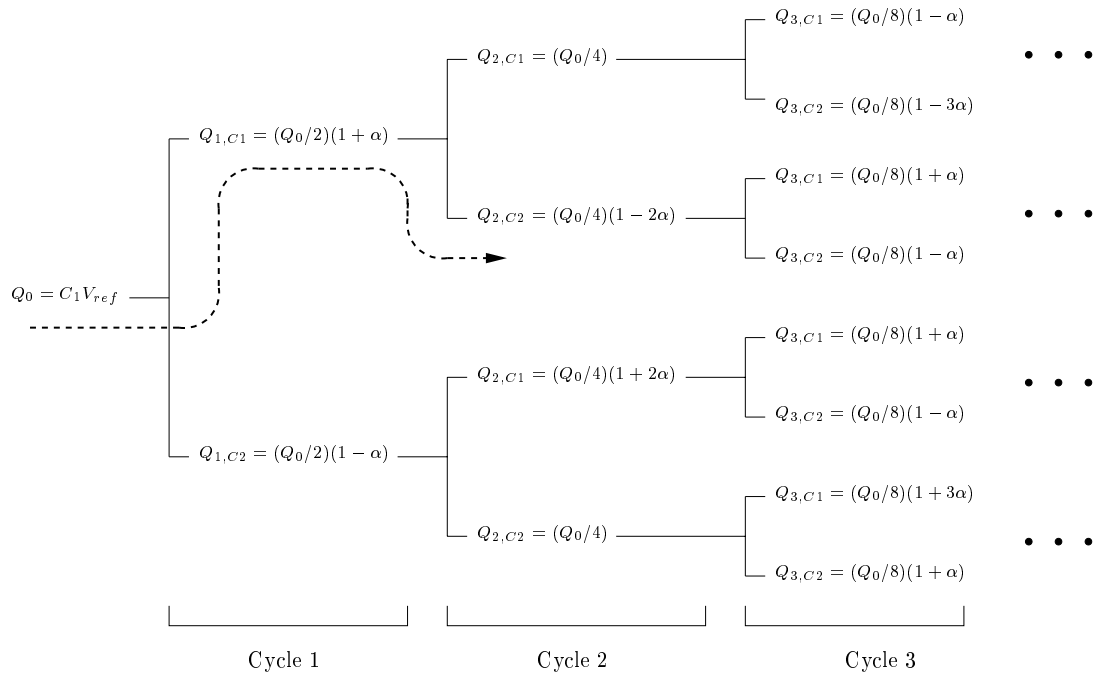

FIGURE 3.7: Error map resulting from DAC-2 architecture.

The example path denoted by the dotted arrow represents the switching sequence in the first two clock cycles. In this case C_1 was used to dump charge to the integrating capacitor in the first cycle and C_2 in the second. Note that the error charge accumulated so far in the example is $(Q_0\alpha)/2 + (-Q_0 2\alpha)/4 = 0$. The error charge on the integration capacitor up to this point has been canceled to a first order approximation. The error accumulation is dependent upon the digital bits that are fed back by the comparator and

upon the choice of either C_1 or C_2 used to transfer charge to the integration capacitor. For each data conversion, it can be shown that the net error accumulated up to any location in the error map is fully described (to a first-order approximation) by the equation

$$Error \simeq \alpha \sum_{k=1}^{N} p(k)2^{-k}[t(k) - \sum_{j=1}^{k-1} t(j)] \tag{3.2}$$

where $t(k) = \pm 1$ denotes the choice of C_1 or C_2, and $p(k) = \pm 1$ represents the comparator output at each cycle. Equation 3.2 is similar to that found in [10].

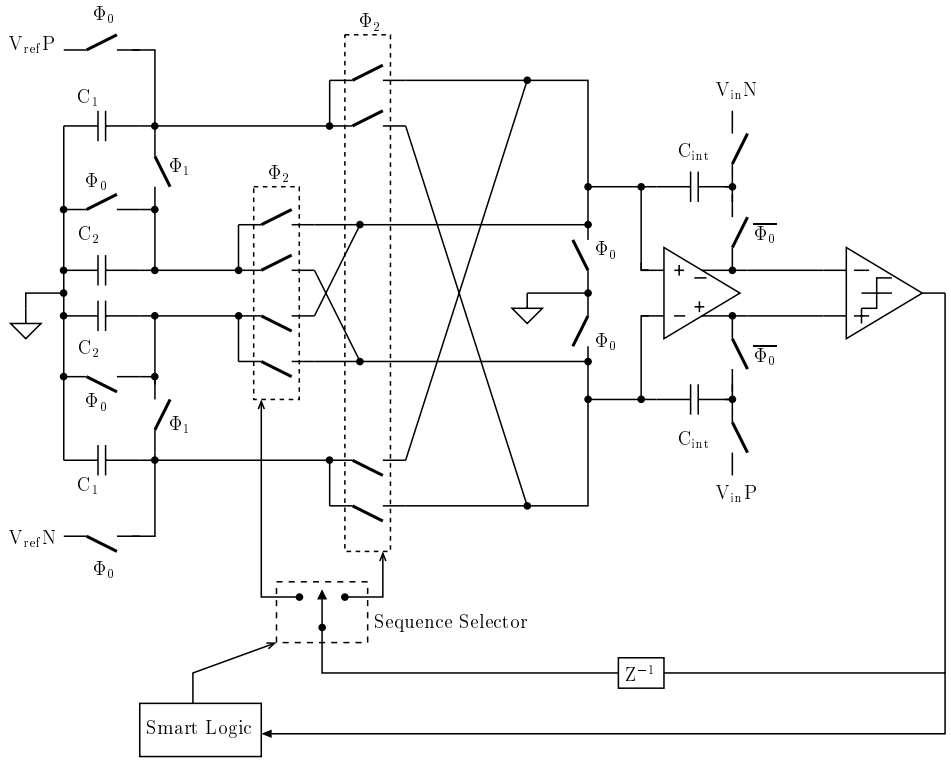

FIGURE 3.8: Differential implementation of mismatch-shaping ADC.

A differential circuit representation for the conceptual ADC of Figure 3.6 is shown in Figure 3.8. Each box controlled by the sequence selector contains a set of switches that can either add or subtract charge from the integrating capacitors. The smart logic

keeps track of the accumulated error over all previous conversion cycles and uses this information to control the switching sequence and shape the mismatch error.

3.4 Digital Implementation

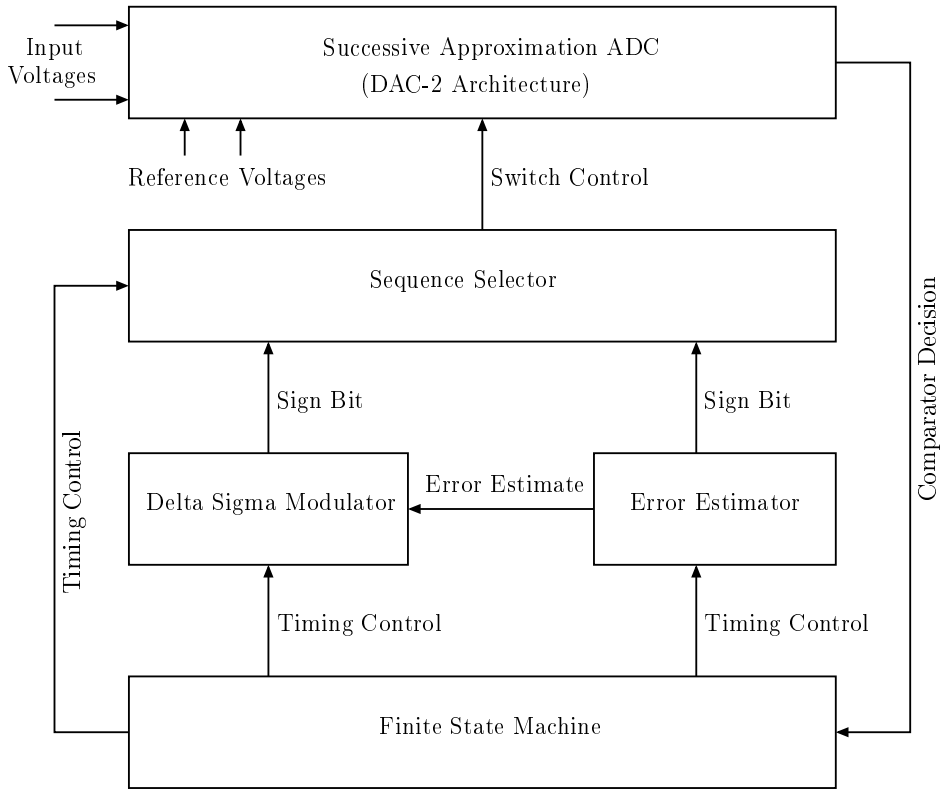

FIGURE 3.9: Block diagram of mismatch-shaping system.

The block diagram of the mismatch-shaping ADC, including some details of the smart logic and sequence selector, is shown in Figure 3.9. Two's complement number representation is preferred and used whenever possible since it facilitates fast arithmetic for signed numbers.

The Error Estimator block implements Equation 3.2. Notice that the magnitude of

the incremental error in each bit-cycle is monotonically decreasing, so the polarity of error made during the first bit-cycle will be the same as the polarity of the accumulated error at the end of the conversion cycle. During each bit-cycle of a conversion period, the Error Estimator keeps track of the accumulated error. At the end of each conversion period the final accumulated error value is passed to the delta-sigma modulator. The Error Estimator circuitry is divided into three stages as shown in Figure 3.10. In the first stage, the past values of $t(k)$ are accumulated and the difference of the current value of $t(k)$ and the sum of the past values is computed. All values in this stage are represented as 5-bit two's complement numbers. The second stage realizes the 2^{-k} operation as well as the possible sign change from the multiplication with $p(k)$. At this point the 5-bit two's complement number from the first stage is converted into a sign-magnitude format and padded to 16-bits. This format conversion is made to facilitate the change in representation from a 16-bit number with a value greater than one to a 16-bit value that is a fraction of one. The multiplication by 2^{-k} is realized by a programmable shifter which is controlled by the finite state machine. At this point the sign bit can simply be inverted, if required, due to the multiplication with $p(k)$ (recall that $p(k)$ only takes on the values of ± 1). The recombination of the 15-bit unsigned magnitude with the computed sign bit yielding a 16-bit two's complement value completes this stage. This number represents a signed value with a magnitude less than one. The final stage accumulates the incremental error values to produce an accumulated error for the conversion cycle.

The Delta Sigma Modulator (DSM) block is an all digital second-order delta-sigma modulator. A second order design was chosen since analytical methods exist to ensure stability of this architecture [15]. Better performance could be achieved by using a higher order modulator with a more aggressive loop-filter transfer function. The DSM attempts to keep the average value of the estimated error $\hat{e}$ equal to a desired value e_d, which is zero in this case. The sampled-data equivalent model of a 2^{nd} order delta-sigma modulator is shown in Figure 3.11. The mismatch noise is high-pass filtered by the DSM which

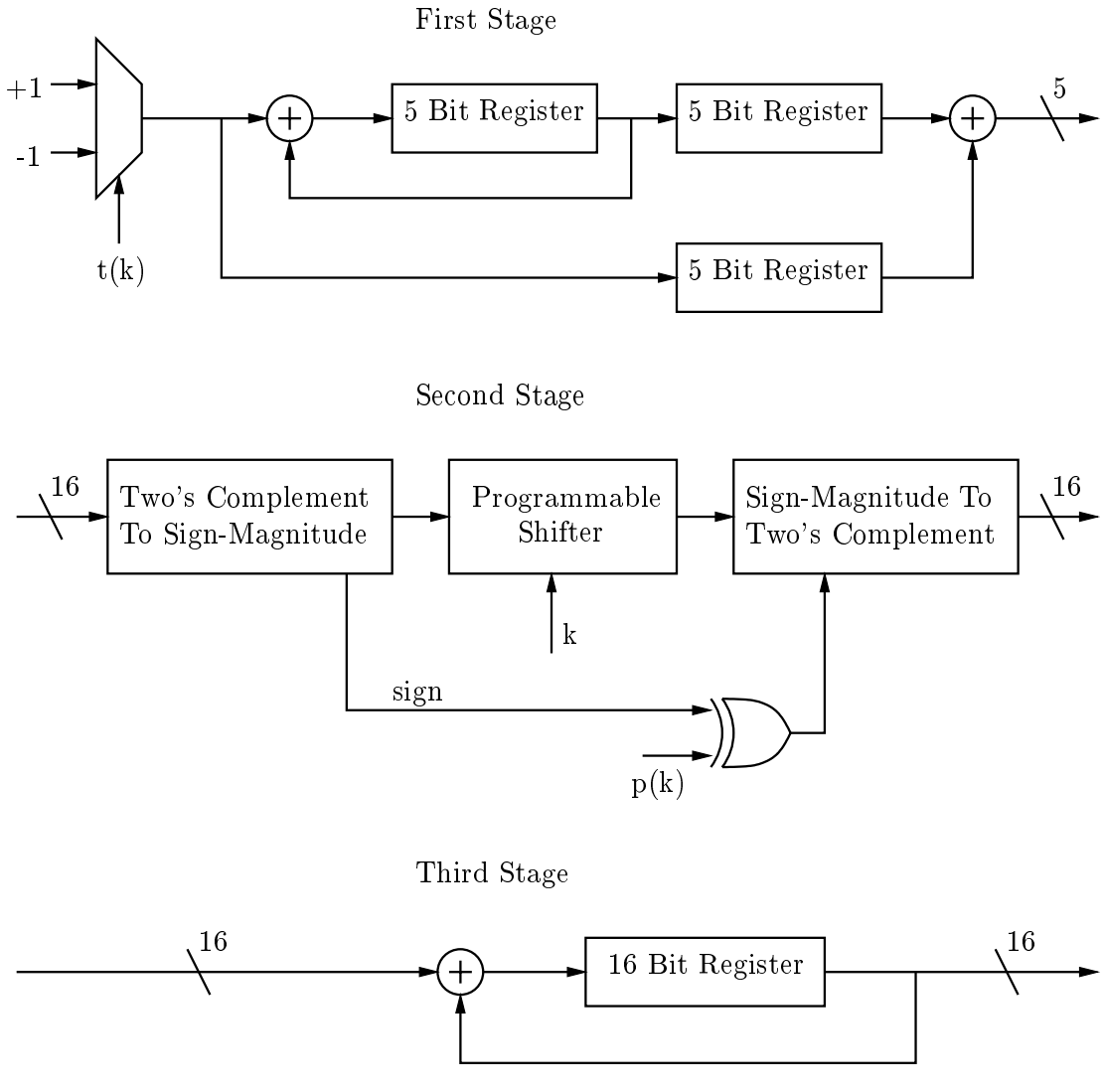

FIGURE 3.10: Block diagram of the error estimator block.

results in decreased energy at low frequencies. Mismatch noise can be greatly suppressed by using a high order and a high gain loop filter. The input to the DSM from the Error Estimator is zero padded to 20 bits to avoid round-off errors. All arithmetic in the DSM is conducted with 20-bit two's complement numbers. The DSM block has the most stringent speed requirements for arithmetic. In order for the DSM output data to be valid for use

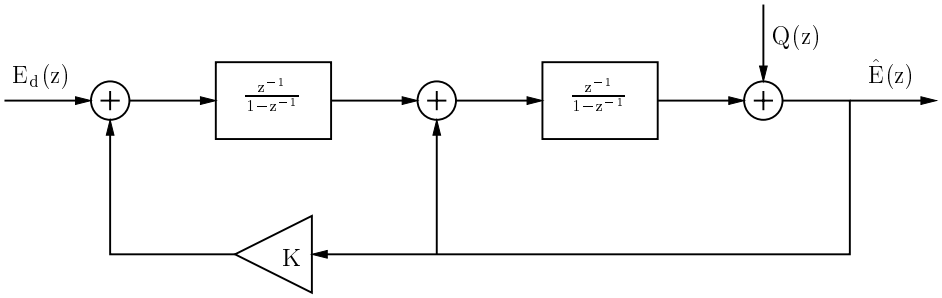

FIGURE 3.11: Sampled-data equivalent model of 2^{nd} order $\Delta\Sigma$ modulator.

in the beginning of one conversion cycle, three 20-bit addition operations and one 20-bit division operation must be performed in one bit-cycle. The 20-bit adders in this block are realized as group ripple adders. The group ripple adders are a cascade of four 5-bit carry look-ahead adders [16]. The divisor was selected to be a power of two so that an arithmetic shifter could be used to perform the division (shown in Figure 3.11 as a gain element of value K).

The Sequence Selector contains the two-phase non-overlapping clock generation circuitry and decoding logic that routes the clock signals to the correct set of switches in each cycle. The sequence selector uses signals from the finite state machine, the sign bit of the DSM output, and the sign bit of the Error Estimator output to select the correct switches. The block diagram of the Sequence Selector is shown in Figure 3.12.

The Finite State Machine is realized as a 5 state "almost" one-hot controller. The initial state is no-hot which allows the 5 states to be realized with only 4 state variables. Having the initial state coded as '0000' also simplifies reset and next-state logic. The state diagram, which shows the state encoding, is shown in Figure 3.13. In the Sample state, the input voltage is sampled and held, the Error Estimator is reset, and the DSM is updated. Also recall from the description of the mismatch-shaping DAC, that the reference voltage is sampled by C_1. No charge sharing occurs in this state. Charge sharing occurs in all

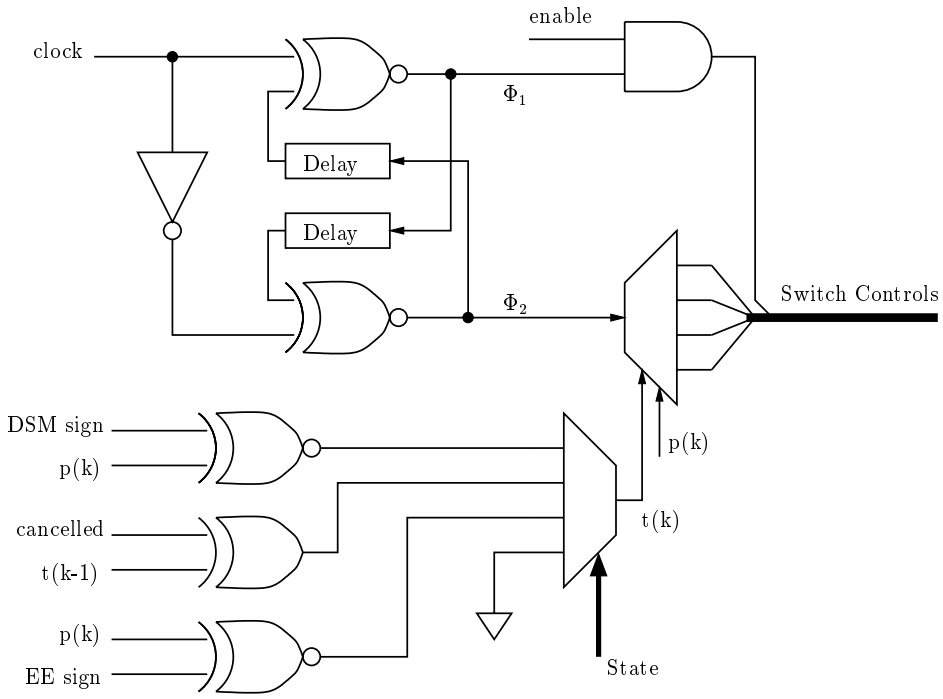

FIGURE 3.12: Sequence selector block diagram.

states except Sample. During state Cancel_1, the output of the DSM is used to make a capacitor choice. Remember that the polarity of the final error in a conversion cycle will be dictated by this first capacitor choice. States Cancel_1 and Cancel_2 always occur as a pair. When two consecutive bits are equal, by using opposite capacitors in consecutive bit-cycles, the mismatch error is cancelled to a first-order approximation (refer to the error map in Figure 3.7). When possible, errors are cancelled. If the data bits obtained during states Cancel_1 and Cancel_2 are not equal, state Minimize is entered. While in the Minimize state, feedback from the Error Estimator is used to ensure that the smallest possible magnitude of error is made during a conversion cycle. The aptly named state Last_Bit is entered during the last bit-cycle of a conversion cycle. During the Last_Bit state, the circuit is prepared for the next sample and hold operation, and the output word

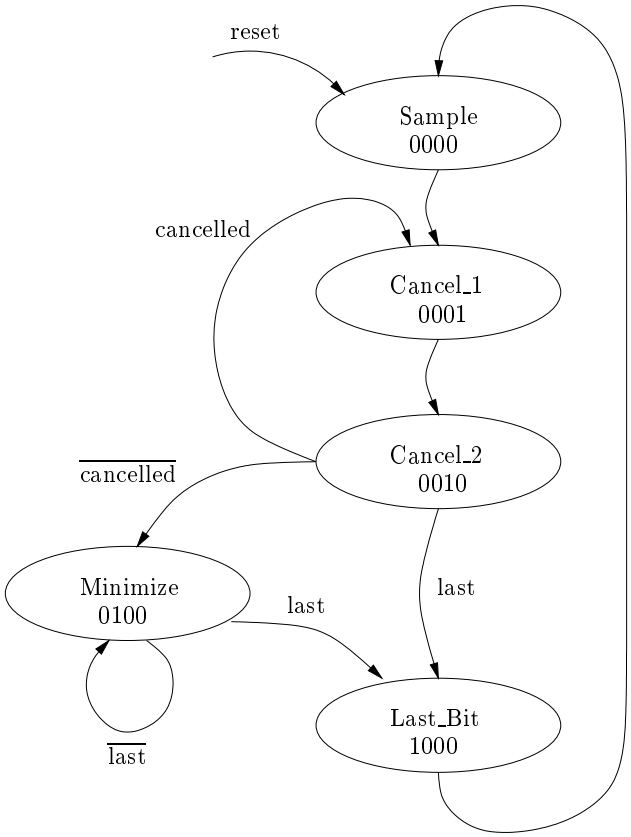

FIGURE 3.13: State diagram for finite state machine with state encodings.

stored in the SAR is buffered into an output register.

The physical layout of the circuit is shown in Figure 3.14.

3.5 Digital Design Flow

The design flow for the implementation of the digital portion of this project, including a list of tools, is shown in Figure 3.15.

At the beginning of the project, MATLAB [17] was used to conduct high level

FIGURE 3.14: Digital portion layout (TSMC 0.35μm process).

system simulations of the proposed solution. MATLAB is an attractive first step in the process since simulations are easy to create, easy to revise, and data processing and visualization is greatly facilitated. One of the major drawbacks to these simulations is that it is difficult to simulate some important characteristics (finite precision calculations, some analog circuit non-idealities) without over-complicating the simulation. The goal is not to write a custom circuit simulation environment. These simulations verify the viability of the approach as well as allow experimentation with several system parameters.

The information and insight gained from the initial system level simulations was used to make better decisions regarding the organization and selection of the hardware structures. Writing a system description in behavioral Verilog requires that you already have a good idea of how each functional block of your system will be implemented in hardware. Behavioral Verilog descriptions are not always synthesizable into physical components. The advantage of using the behavioral features of Verilog is that you can

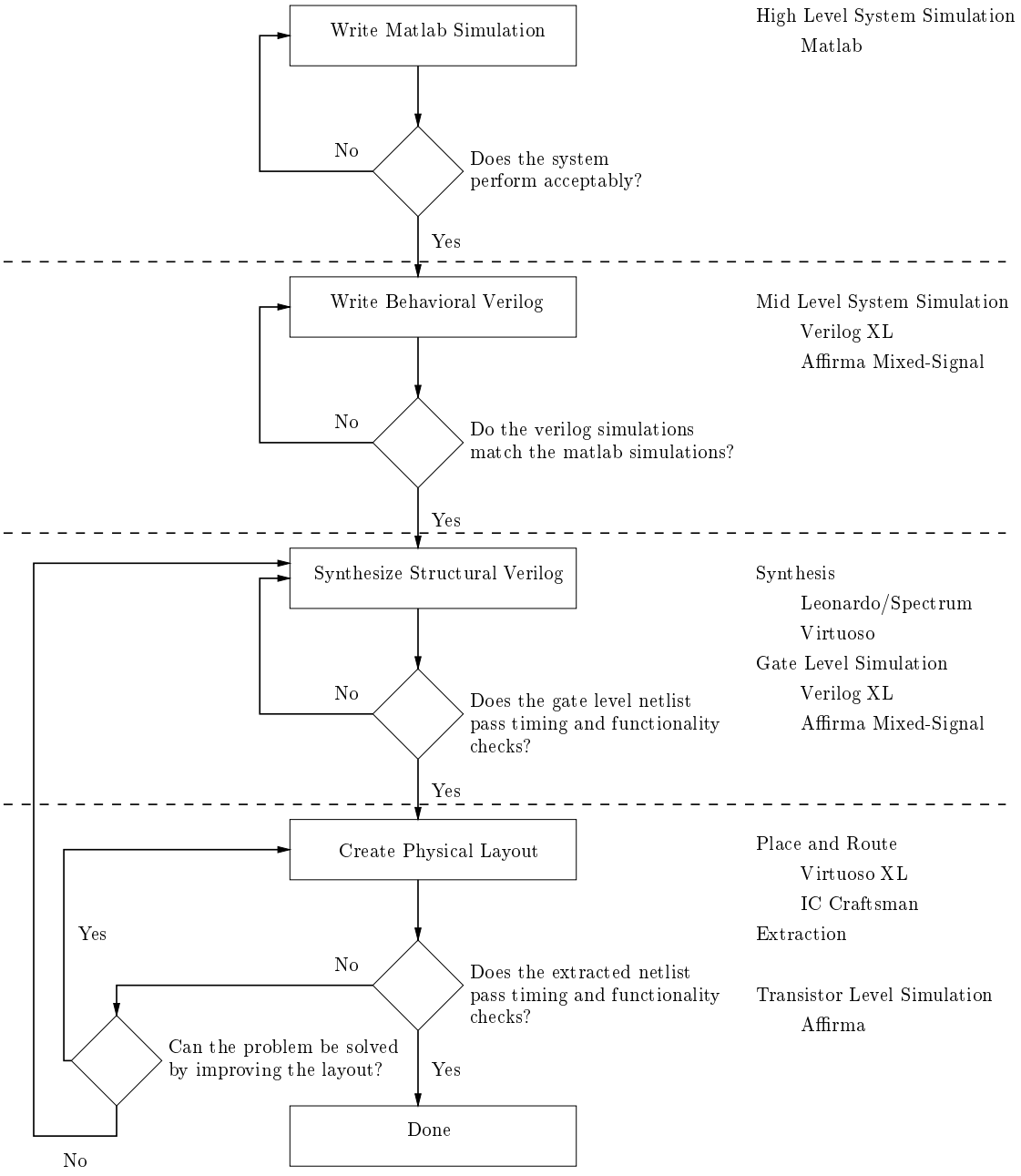

FIGURE 3.15: Design flow diagram.

start with a system description very similar to the MATLAB simulation, but you can begin to integrate some more realistic non-idealities such as signal assignment delays and finite precision calculations.

Functionally verified behavioral Verilog descriptions must be further refined to reflect the actual physical structure that the design will take. As this circuit is not very complex, most of the circuit synthesis was conducted in an ad-hoc fashion. Some components, namely the finite state machine and the programmable shifter, were synthesized with the help of Leonardo/Spectrum from Mentor Graphics [18]. For most of the design, structural Verilog descriptions were written manually at the gate level. The simulations at this level are fairly accurate. While wire delays are still provided by statistical models (wire load tables), the intrinsic delays and capacitances are well defined. At this point, most of the circuit parameters are known and can easily be incorporated into simulations. Fortunately, the level of abstraction does not yet include transistor models, so simulations execute in very little time. The Cadence [19] design entry (Virtuoso) and digital simulation (Verilog XL) tools provide a very powerful and easy to use environment for design verification at this level. Text descriptions of system blocks (Verilog, VHDL) can be combined with digital and analog schematics (Virtuoso, AHDL) to perform fast and accurate mixed-signal simulations. Most of the timing issues are addressed at this stage.

When mixed-signal simulations indicate that the digital and analog parts of the design are correctly interfaced, we must progress to the physical description of the circuit. For the digital blocks, automatic place and route can be performed using Virtuoso XL and IC Craftsman from Cadence. The Virtuoso XL program creates a floor plan of the chip and places gates in a logical order from a routing and delay standpoint. After the gates are arranged inside the chip, the routing lines may be added. This process can also be automated (IC Craftsman). The finished product is a physical layout using the appropriate layers for the targeted process. As a final simulation check before fabrication, a netlist based on extracted values from the physical layout should be generated. Unfortunately,

simulations at this level take an extremely long time to complete. As a reference point, a simulation of the system at the structural Verilog level takes less than a second to complete while the same simulation at the transistor level takes over 28 hours to complete. For fast or complex designs a large percentage of signal delays occur in the routing networks, so these simulations should not be skipped.

3.6 Analog Requirements

While the implementation of the analog portion of the circuit was not included in this project, it was necessary to model the analog portion for testing purposes. The following is a brief description of the models used. Capacitor sizes were selected first. Capacitors are noiseless elements, but they do accumulate noise from other sources. In our case, the resistive channel of the MOSFET switches causes the capacitors to accumulate an rms noise voltage of [13].

$$\mathrm{V_{rms}} = \sqrt{\frac{\mathrm{kT}}{\mathrm{C}}} \tag{3.3}$$

To ensure that this noise does not degrade the performance of the system, the rms noise voltage should be less than one bit level. For a 15-bit converter, the noise voltage should be less than $(2^{15}-1)*V_{ref}$ = 30.5μV. This indicates that the minimum allowable capacitor size is 5 pF. Since we also have an OSR of 10, the minimum allowable capacitor size becomes C/OSR = 0.5 pF. We will allow 12 time constants for the RC network to charge the capacitor to the desired settling accuracy. This makes the RC time constant of the sampling switch and the capacitor 6.6 nsec. We solve for a maximum switch on-resistance of 13 kΩ.

The opamp model used for simulations had a gain of 120 dB and a unity-gain bandwidth of 32 MHz. The opamp and comparator models had no input offset. Response times of integrated comparators of the positive feedback variety are fast enough that it is

unimportant to model it in this design.

4. EXPERIMENTAL RESULTS

In all of the following simulations the capacitor mismatch was 0.1% (α = .001), and the input was a 0.707V_{ref} peak-to-peak sine wave with a frequency 32 times lower than the Nyquist rate.

4.1 Behavioral Simulation Results

As a point of reference, the simulation output from an uncompensated 15-bit successive-approximation ADC is shown in Figure 4.1. The simulated SNDR (signal to noise and distortion ratio) for this circuit is 63 dB. Also notice that the response is very tonal, with the total harmonic distortion at -70 dB.

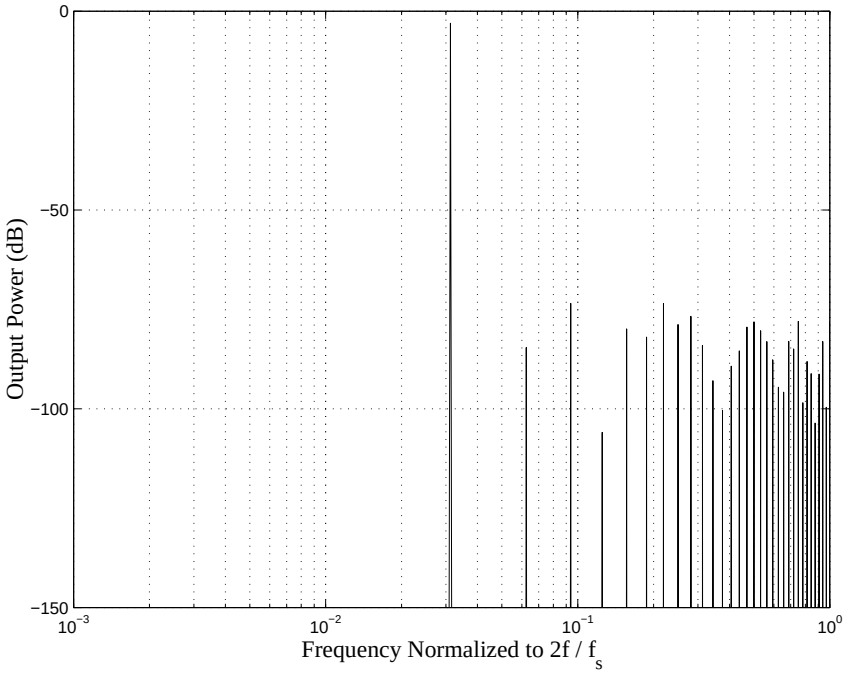

FIGURE 4.1: Output spectrum of uncompensated converter (MATLAB).

Compared to this response, the simulation output from the compensated 15-bit converter, which is shown in Figure 4.2, is much better.

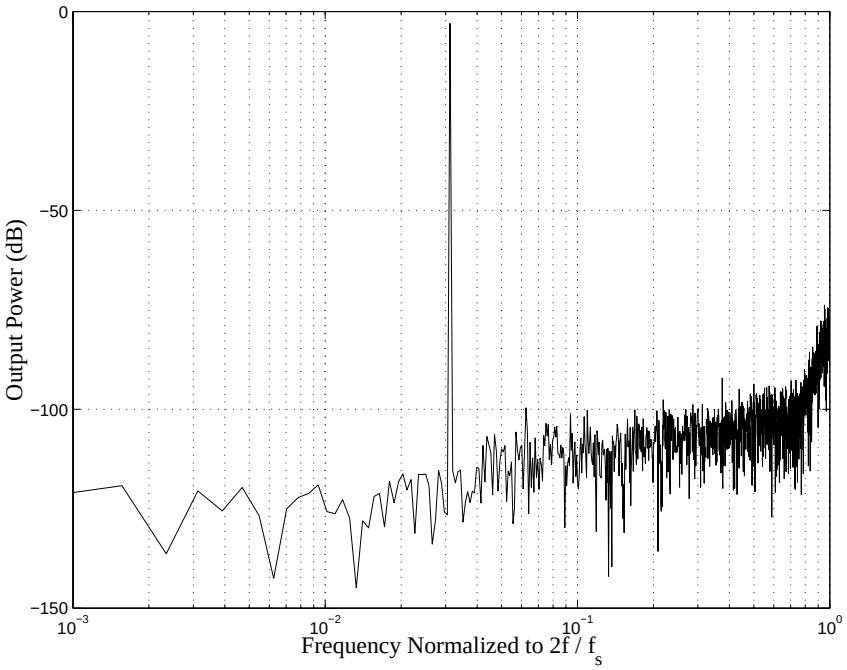

FIGURE 4.2: Output spectrum of mismatch-shaped converter (MATLAB).

The SNDR for this circuit was simulated at 87 dB for an OSR of 10. The response shows no observable harmonic spurs and some noise-shaping is evident. As expected, the noise-shaping is not as well behaved as for similarly compensated DACs but clearly shows a significant improvement.

4.2 Gate Level Simulation Results

A simulation of the gate level abstraction yielded results identical to those of the MATLAB simulation for the digital portion of the circuit. The simulation was conducted with gate level Verilog models which include intrinsic and extrinsic delay estimates for

each gate as well as statistical estimates of routing delays. All degradation of the circuit performance at this level is attributed to the non-idealities of the analog circuit, which are not modeled in the MATLAB simulations. The mixed-mode simulation was found to be flawed. Using ideal components and input waveforms identical to those used in the MATLAB simulations, the the mixed-mode simulation produces very poor results. Since the digital circuit has been verified in separate simulation, and the analog circuit was modeled using ideal components, there is obviously a problem with the mixed-mode simulation or simulator. The output spectrum resulting from this simulation is shown in Figure 4.3. The response shows tones that are likely numerical artifacts produced by the simulator. The SNDR is dramatically decreased to 72 dB. The bandwidth of the converter is 0 to 20 kHz.

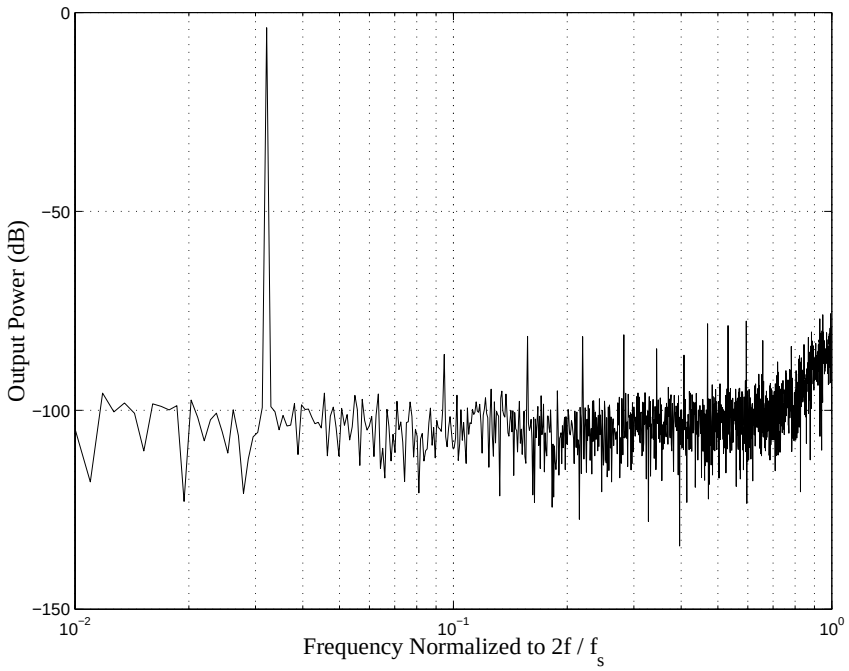

FIGURE 4.3: Output spectrum of mismatch-shaped converter (Verilog).

4.3 Transistor Level Simulation Results

A transistor level simulation long enough to allow performance measurements would require far too much time. A rough estimate shows that to validate the previous simulation at the transistor level would take one and three-quarters years of uninterrupted processing. To lend more credibility to the Verilog mixed-signal simulation, a comparison of the switch control outputs of the transistor level simulation to the switch control outputs of the Verilog simulation is shown in Figure 4.4. Figure 4.4 shows the output of the transistor level simulation for four conversion cycles. All switch transitions are identical to those produced by the Verilog simulations. This confirms the validity of the layout and increases the confidence in the Verilog simulations.

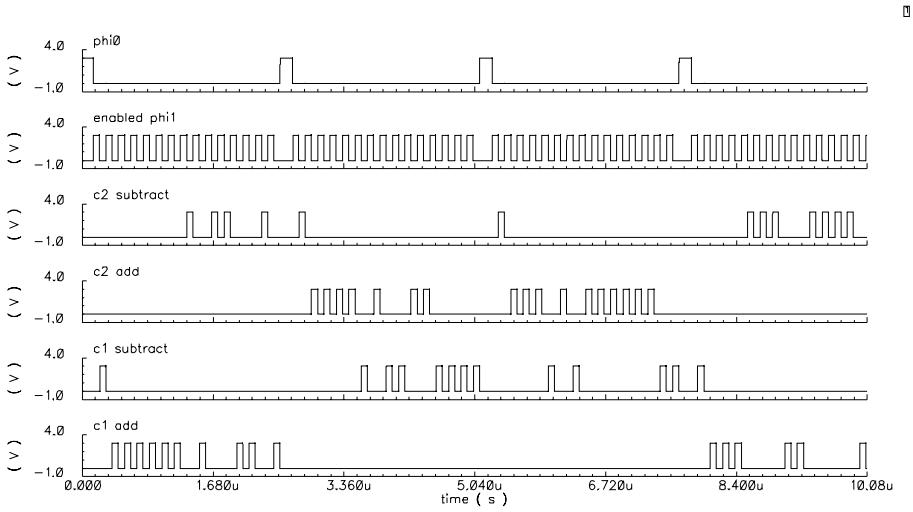

FIGURE 4.4: Switch control outputs from transistor level simulation.

5. CONCLUSIONS

The digital design of a mismatch-shaping successive-approximation ADC has been described. Simulations indicate that for a modest oversampling ratio, mismatch shaping is successful. A digital circuit layout is presented for the TSMC 0.35μm process. The all-digital $\Delta\Sigma$ loop used operates with the calculated mathematical estimates of the capacitor mismatch error that are accurate to a first-order approximation. Higher order errors that are not accounted for in the error estimation algorithm are essentially negligible.

Possible future research could include the design and layout of the analog portion of the circuit, the design and layout of the decimation filter, and the fabrication of the completed system. In addition, better noise shaping could be investigated using a higher order loop filter in the $\Delta\Sigma$ modulator.

REFERENCES

1. R. T. Baird and T. S. Fiez, "Improved Delta-Sigma DAC linearity using data weighted averaging," *Proceedings of the 1995 IEEE Symposium on Circuits and Systems*, pp. 13–16, 1995.

2. C. K. Thanh, S. H. Lewis, and P. J. Hurst, "A second-order double-sampled delta-sigma modulator using individual-level averaging," *IEEE Journal of Solid-State Circuits*, vol. 32, pp. 1269–1273, August 1997.

3. T. Kwan and R. Adams, "Data-directed scrambler for multi-bit noise-shaping d/a converters." U.S. Patent Number 5,404,142.

4. R. Adams, K. Nguyen, and K. Sweetland, "A 113dB SNR oversampling DAC with segmented noise-shaped scrambling," *Digest of Technical Papers for the 1998 International Solid-State Circuits Conference*, vol. 41, pp. 62–63, February 1998.

5. R. Schreier and B. Zhang, "Noise-shaped multibit D/A convertor employing unit elements," *Electronics Letters*, vol. 31, pp. 1712–1713, September 1995.

6. L. Hernandez, "Digital implementation of mismatch shaping in oversampled pipeline A/D converters," *Electronics Letters*, vol. 34, pp. 616–617, April 1998.

7. J. Steensgaard, U. Moon, and G. C. Temes, "Mismatch shaping switching for two-capacitor DAC," *Electronics Letters*, vol. 34, pp. 1633–1634, August 1998.

8. J. Steensgaard, U. Moon, and G. C. Temes, "Mismatch-shaping serial digital-to-analog converter," *Proceedings of the 1999 IEEE Symposium on Circuits and Systems*, vol. 2, pp. 5–8, May 1999.

9. L. Hernandez, "Binary weighted D/A converters with mismatch-shaping," *Electronics Letters*, vol. 33, pp. 2006–2007, November 1997.

10. P. Rombouts, L. Weyten, J. Raman, and S. Audenaert, "Capacitor mismatch compensation for quasi-passive switched-capacitor DAC," *IEEE Transactions on Circuits and Systems-I: Fundamental Theory and Applications*, vol. 45, pp. 68–71, January 1998.

11. L. Weyten and S. Audenaert, "Two-capacitor DAC with compensative switching," *Electronics Letters*, vol. 31, pp. 1435–1436, August 1995.

12. P. W. Li, M. J. Chin, P. R. Gray, and R. Castello, "A ratio-independent algorithmic analog-to-digital conversion technique," *IEEE Journal of Solid-State Circuits*, vol. SC-19, pp. 828–836, December 1984.

13. D. A. Johns and K. Martin, *Analog Integrated Circuit Design*. New York, NY: John Wiley & Sons, Inc., 1997.

14. R. Suarez, P. Gray, and D. Hodges, “All MOSFET charge-redistribution analog-to-digital conversion techniques – Part II,” *IEEE Journal of Solid-State Circuits*, vol. SC-10, pp. 379–385, December 1975.

15. S. R. Norsworthy, R. Schreier, and G. C. Temes, eds., *Delta-Sigma Data Converters: Theory, Design, and Simulation*. Piscataway, NJ: IEEE Press, 1996.

16. J. H. Herzog, *Design and Organization of Computing Structures*. Wilsonville, OR: Franklin, Beedle & Associates, 1996.

17. The MathWorks, Inc., Natick, MA, *MATLAB*.

18. Mentor Graphics Corporation, Wilsonville, OR, *LeonardoSpectrum*.

19. Cadence Design Systems, Inc., San Jose, CA, *Design Framework II*.

APPENDIX

MATLAB simulation code

```
%System Level simulation for differential mismatch shaping
clear all
close all

%debug = 1 for single-bit diagnostics, debug = 0 for SNDR calculations
debug = 0;
testbit = 1:4;
graphics = 1;
epsout = 0;
verilog_outfile = 0; %outfile = 1 to generate verilog stimulus and feedback
spectre_outfile = 0; %outfile = 1 to generate VinN and VinP for Spectre

%Design Variables
N=15;
a=.001;
Vrefp=1;
Vrefn=-1;
Csh=0.5e-12;
Cint=0.5e-12;

%Calculated Quantities
Cshp=(1+a)*Csh;
Cshn=(1-a)*Csh;
gnda=(Vrefp+Vrefn)/2;

window=40*(2*pi);
inc=2*pi/64;
Vinp=0.5*sin(0:inc:window-inc)/sqrt(2)/0.999+0.01;
Vinn=-0.5*sin(0:inc:window-inc)/sqrt(2)/0.999-0.01;

time=([0:inc:window-inc]);
freq=time/max(time+inc);

E=0; E1=0; E2=0; E3=0;
z=1;

if verilog_outfile == 1
   fid = fopen('matout2', 'w');
end

if spectre_outfile == 1
   fid2 = fopen('specout', 'w');
end

if debug == 1
   y=testbit;
else
   y=1:length(Vinp);
end

for j=y
   t(j,:)=zeros(1,N); Ex=zeros(1,N);
   Exz(j,:)=zeros(1,N); ExINCz(j,:)=zeros(1,N);
   k=1; z=1;

   %Precharge Cycle
   Q1a=Vrefp*Cshp;
   Q2a=gnda*Cshn;
   Q1b=Vrefn*Cshp;
   Q2b=gnda*Cshn;
   Qinta=Vinn(j)*Cint;
   Qintb=Vinp(j)*Cint;

   %First Bit-Cycle
```

```
Qat=Q1a+Q2a;
Q1a=Qat*(Cshp/(Cshp+Cshn));
Q2a=Qat*(Cshn/(Cshp+Cshn));
Qbt=Q1b+Q2b;
Q1b=Qbt*(Cshp/(Cshp+Cshn));
Q2b=Qbt*(Cshn/(Cshp+Cshn));

comp(j,k)=sign(sign(Qintb-Qinta)+0.5);
loop(j,k)=1;

if sign(E) >= 0
   if comp(j,k) == 1
      Qdumpa=Q2a;
      Q2a=gnda*Cshn;
      Qdumpb=Q2b;
      Q2b=gnda*Cshn;
      t(j,k)=-1;
   else
      Qdumpa=Q1a;
      Q1a=gnda*Cshp;
      Qdumpb=Q1b;
      Q1b=gnda*Cshp;
      t(j,k)=1;
   end
else
   if comp(j,k) == 1
      Qdumpa=Q1a;
      Q1a=gnda*Cshp;
      Qdumpb=Q1b;
      Q1b=gnda*Cshp;
      t(j,k)=1;
   else
      Qdumpa=Q2a;
      Q2a=gnda*Cshn;
      Qdumpb=Q2b;
      Q2b=gnda*Cshn;
      t(j,k)=-1;
   end
end

%Integrate
if comp(j,k) == 1
   Qinta=Qinta+Qdumpa;
   Qintb=Qintb+Qdumpb;
else
   Qinta=Qinta+Qdumpb;
   Qintb=Qintb+Qdumpa;
end

%Calculate Error this bit cycle
Ex(k) = comp(j,k)*(t(j,k)-sum(t(j,1:k-1)))*(2^-k);
Exz(j,z)=sum(Ex); ExINCz(j,z)=Ex(k); z=z+1;

k=2; F=0;
while F == 0
   Qat=Q1a+Q2a;
   Q1a=Qat*(Cshp/(Cshp+Cshn));
   Q2a=Qat*(Cshn/(Cshp+Cshn));
   Qbt=Q1b+Q2b;
   Q1b=Qbt*(Cshp/(Cshp+Cshn));
   Q2b=Qbt*(Cshn/(Cshp+Cshn));

   comp(j,k)=sign(sign(Qintb-Qinta)+0.5);
   loop(j,k)=2;

   if comp(j,k-1) == comp(j,k)
      if t(j,k-1) == -1
```

```
            Qdumpa=Q1a;
            Q1a=gnda*Cshp;
            Qdumpb=Q1b;
            Q1b=gnda*Cshp;
            t(j,k)=1;
         else
            Qdumpa=Q2a;
            Q2a=gnda*Cshn;
            Qdumpb=Q2b;
            Q2b=gnda*Cshn;
            t(j,k)=-1;
         end
      else
if t(j,k-1) == -1
            Qdumpa=Q2a;
            Q2a=gnda*Cshn;
            Qdumpb=Q2b;
            Q2b=gnda*Cshn;
            t(j,k)=-1;
else
            Qdumpa=Q1a;
            Q1a=gnda*Cshp;
            Qdumpb=Q1b;
            Q1b=gnda*Cshp;
            t(j,k)=1;
        end
      end

      %Integrate
      if comp(j,k) == 1
         Qinta=Qinta+Qdumpa;
         Qintb=Qintb+Qdumpb;
      else
         Qinta=Qinta+Qdumpb;
         Qintb=Qintb+Qdumpa;
      end

      %Calculate Error this bit cycle
      Ex(k) = comp(j,k)*(t(j,k)-sum(t(j,1:k-1)))*(2^-k);
      Exz(j,z)=sum(Ex);ExINCz(j,z)=Ex(k);z=z+1;

      F = (1-(comp(j,k)*comp(j,k-1)));

      if k == N
         F = 3;
      end

      if F == 0
         k = k+1;

         Qat=Q1a+Q2a;
         Q1a=Qat*(Cshp/(Cshp+Cshn));
         Q2a=Qat*(Cshn/(Cshp+Cshn));
         Qbt=Q1b+Q2b;
         Q1b=Qbt*(Cshp/(Cshp+Cshn));
         Q2b=Qbt*(Cshn/(Cshp+Cshn));

         comp(j,k)=sign(sign(Qintb-Qinta)+0.5);
         loop(j,k)=3;

         if sign(E) >= 0
            if comp(j,k) == 1
               Qdumpa=Q2a;
               Q2a=gnda*Cshn;
               Qdumpb=Q2b;
               Q2b=gnda*Cshn;
               t(j,k)=-1;
```

```
         else
            Qdumpa=Q1a;
            Q1a=gnda*Cshp;
            Qdumpb=Q1b;
            Q1b=gnda*Cshp;
            t(j,k)=1;
         end
      else
         if comp(j,k) == 1
            Qdumpa=Q1a;
            Q1a=gnda*Cshp;
            Qdumpb=Q1b;
            Q1b=gnda*Cshp;
            t(j,k)=1;
         else
            Qdumpa=Q2a;
            Q2a=gnda*Cshn;
            Qdumpb=Q2b;
            Q2b=gnda*Cshn;
            t(j,k)=-1;
         end
      end

      %Integrate
      if comp(j,k) == 1
         Qinta=Qinta+Qdumpa;
         Qintb=Qintb+Qdumpb;
      else
         Qinta=Qinta+Qdumpb;
         Qintb=Qintb+Qdumpa;
      end

      %Calculate Error this bit cycle
      Ex(k) = comp(j,k)*(t(j,k)-sum(t(j,1:k-1)))*(2^-k);
      Exz(j,z)=sum(Ex); ExINCz(j,z)=Ex(k); z=z+1;
   end

   k = k+1;
   if k >= N+1
      F = 3;
   end

end

%Magnitude Minimization
for i=k:N
   Qat=Q1a+Q2a;
   Q1a=Qat*(Cshp/(Cshp+Cshn));
   Q2a=Qat*(Cshn/(Cshp+Cshn));
   Qbt=Q1b+Q2b;
   Q1b=Qbt*(Cshp/(Cshp+Cshn));
   Q2b=Qbt*(Cshn/(Cshp+Cshn));

   comp(j,i)=sign(sign(Qintb-Qinta)+0.5);
   loop(j,i)=4;

   %Figure out which cap to dump and where to dump it
   if sum(Ex) >= 0
      if comp(j,i) == 1
         Qdumpa=Q2a;
         Q2a=gnda*Cshn;
         Qdumpb=Q2b;
         Q2b=gnda*Cshn;
         t(j,i)=-1;
      else
         Qdumpa=Q1a;
         Q1a=gnda*Cshp;
```

```
            Qdumpb=Q1b;
            Q1b=gnda*Cshp;
            t(j,i)=1;
         end
      else
         if comp(j,i) == 1
            Qdumpa=Q1a;
            Q1a=gnda*Cshp;
            Qdumpb=Q1b;
            Q1b=gnda*Cshp;
            t(j,i)=1;
         else
            Qdumpa=Q2a;
            Q2a=gnda*Cshn;
            Qdumpb=Q2b;
            Q2b=gnda*Cshn;
            t(j,i)=-1;
         end
      end

      %Integrate
      if comp(j,i) == 1
         Qinta=Qinta+Qdumpa;
         Qintb=Qintb+Qdumpb;
      else
         Qinta=Qinta+Qdumpb;
         Qintb=Qintb+Qdumpa;
      end

      %Calculate Error this bit cycle
      Ex(i) = comp(j,i)*(t(j,i)-sum(t(j,1:i-1)))*(2^-i);
      Exz(j,z)=sum(Ex); ExINCz(j,z)=Ex(i); z=z+1;
   end

   bits(j,:) = (comp(j,:)+1)/2;
   ct(j,:) = (t(j,:)+1)/2;

   if verilog_outfile == 1
     out = [[ct(j,:) 5]; [bits(j,:) 0]];
     fprintf(fid, '%1.0f\t%1.0f\n', out);
   end

   if spectre_outfile == 1
     out2 = [Vinn; Vinp];
     fprintf(fid2, '%f\t%f\n', out2);
   end

   vout(j)=0;
   for i=1:N
      vout(j)=vout(j)+(Vrefp-Vrefn)*bits(j,i)/(2^i);
   end
   %vout(j)=vout(j)-((Vrefp-Vrefn)/2);

   E1=E1+0.0625*sum(Ex); E=E+E1+sum(Ex);
   %E1=E1+0.5*sum(Ex); E2=E2+0.5*(E1+sum(Ex)); E=E+E2+sum(Ex); %third-order
   %E1=E1+sum(Ex); E2=E2+E1; E3=E3+E2; E=E1+0.5*E2+0.25*E3; %third-order
   %E1=E1+.00125*sum(Ex); E2=E2+E1+.0125*sum(Ex); E3=E3+E2+sum(Ex); E=E3; %third-order

   Error(j) = E;
end

if verilog_outfile == 1
   fclose(fid);
end

if spectre_outfile == 1
   fclose(fid2);
```

```
end

if debug == 1
   for i=testbit(1):testbit(end)
      [[bits(i,:)]',[Error(i) zeros(1,N-1)]',Exz(i,:)',ExINCz(i,:)',t(i,:)',loop(i,:)']
   end
else
   %Calculate figure of merit
   norm=length(Vinp)/(2*sqrt(2)^2);

   fin=abs(fft(Vinp-(sum(Vinp)/length(Vinp))));
   fout=abs(fft(vout-(sum(vout)/length(Vinp))));
   fdbin=20*log10(fin/norm);
   fdbout=20*log10(fout/norm);

   SNDR_1=10*log10((sum(fout(1:1281).^2)-fout(41)^2)/(fout(41)^2));
   SNDR_5=10*log10((sum(fout(1:257).^2)-fout(41)^2)/(fout(41)^2));
   SNDR_10=10*log10((sum(fout(1:129).^2)-fout(41)^2)/(fout(41)^2));
   SNDR_20=10*log10((sum(fout(1:65).^2)-fout(41)^2)/(fout(41)^2));
   SNDR_1_5_10_20=[SNDR_1,SNDR_5,SNDR_10,SNDR_20,N]
   if graphics == 1
      figure(1)
      subplot(211)
      semilogx(freq*2,fdbout)
      grid
      axis([0.001,1,-150,0])

      subplot(212)
      plot(Vinp)
      hold on
      plot(Vinn, 'r')
      plot(vout, 'g--')
      axis([0 200 -2 2])
   end
   if epsout == 1
      semilogx(freq*2,fdbout)
      grid
      axis([0.001,1,-150,0])
      ylabel('Output Power (dB)', 'FontSize', 14, 'FontName', 'Times')
      xlabel('Frequency Normalized to 2f / f_s', 'FontSize', 14, 'FontName', 'Times')
      print -deps2 matcomp.eps
   end
end
```

Verilog Structural Descriptions

```
‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

module ADC(word, phi0, phi0bar, enphi1, c1add, c1sub, c2add, c2sub, comp, nibble_clock, reset);

output [14:0] word;
output phi0, phi0bar, enphi1, c1add, c1sub, c2add, c2sub;
input nibble_clock, comp, reset;

wire [3:0] bit;
wire [1:0] select;
wire [4:0] tdiff;
wire [14:0] outword;
wire [15:0] inc, EE;
wire bit_clock, EEsign, p2phi0, p2, p4phi0, p4, t, p, DSMsign, algorithm, readDSM,
```

```
     cancel, outstrobe, clearEE;

CG cg1(enphi1, c1add, c1sub, c2add, c2sub, bit_clock, phi0bar, select);
DSM dsm2(DSMsign, EE, phi0, p2phi0, p4phi0, reset);
EEin in1(tdiff, t, p2, phi0bar);
EEss ss1(inc, tdiff, bit, p);
EEout out1(EEsign, EE, inc, p4, clearEE);
FSM7 fsm1(phi0, phi0bar, p2phi0, p2, p4phi0, p4, outstrobe, bit, readDSM, algorithm, clearSAR,
  clearEE, bit_clock, nibble_clock, cancel, reset);
PHI2 phi21(select, t, p, EEsign, DSMsign, algorithm, readDSM, cancel, bit_clock, p2, reset);
SAR sar1(cancel, outword, p, comp, bit_clock, clearSAR);

reg15 outreg(word, , outword, outstrobe, reset);
endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

//   Two Phase Non-overlapping Clock Generator
module CG(enphi1, c1add, c1sub, c2add, c2sub, bit_clock, enable, select);

//   I/O declarations
output enphi1, c1add, c1sub, c2add, c2sub;
input enable, bit_clock;
input [1:0] select;

//   Internal wires, storage nodes, and constants
wire clk, nclk, phi1, phi1d, phi2, phi2d, c1add, c1sub, c2add, c2sub, enphi1, enphi2;
wire q, qb;

wire wd1, wd2, wd3, wd4, wd5, wd6, wd7, wd8, wd9, wd10, wd11, wd12, wd13, wd14, w1, w2, w3, w4;

//   Functional specification
//   Delays
wand2_1 invd1(VDD, VSS, wd1, phi1, phi1);
wand2_1 invd2(VDD, VSS, wd2, wd1, wd1);
wand2_1 invd3(VDD, VSS, wd3, wd2, wd2);
wand2_1 invd4(VDD, VSS, wd4, wd3, wd3);
wand2_1 invd5(VDD, VSS, wd5, wd4, wd4);
wand2_1 invd6(VDD, VSS, wd6, wd5, wd5);
wand2_1 invd7(VDD, VSS, wd7, wd6, wd6);
wand2_1 invd8(VDD, VSS, phi1d, wd7, wd7);

wand2_1 invd9(VDD, VSS, wd8, phi2, phi2);
wand2_1 invd10(VDD, VSS, wd9, wd8, wd8);
wand2_1 invd11(VDD, VSS, wd10, wd9, wd9);
wand2_1 invd12(VDD, VSS, wd11, wd10, wd10);
wand2_1 invd13(VDD, VSS, wd12, wd11, wd11);
wand2_1 invd14(VDD, VSS, wd13, wd12, wd12);
wand2_1 invd15(VDD, VSS, wd14, wd13, wd13);
wand2_1 invd16(VDD, VSS, phi2d, wd14, wd14);

//    Cross Coupled Nors for non-overlapping clock generation
wnor2_2 nor1(VDD, VSS, phi2, bit_clock, phi1d);
wnor2_2 nor2(VDD, VSS, phi1, nclk, phi2d);
winv_2  not1(VDD, VSS, nclk, bit_clock);

//  1-4 Demux with enable
winv_2 inv1(VDD, VSS, w1, select[1]);
winv_2 inv2(VDD, VSS, w3, w1);
winv_2 inv3(VDD, VSS, w2, select[0]);
winv_2 inv4(VDD, VSS, w4, w2);
wand3_2 andd0(VDD, VSS, c1add, enphi2, w2, w1);
wand3_2 andd1(VDD, VSS, c1sub, enphi2, w4, w1);
```

```
wand3_2 andd2(VDD, VSS, c2add, enphi2, w3, w2);
wand3_2 andd3(VDD, VSS, c2sub, enphi2, w3, w4);

wand2_2 and1(VDD, VSS, enphi2, phi2, enable);
wand2_2 and2(VDD, VSS, enphi1, phi1, enable);

endmodule

//   Mild second order Delta Sigma Modulator
‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

module DSM(DSMsign, EE, ck1, ck2, ck4, rb);

//   I/O declarations

output DSMsign;
input rb, ck1, ck2, ck4;
input [19:0] EE;

//   Internal wires and storage nodes
wire [19:0] reg1in, reg1out, reg2in, reg2out, reg3out, div, inter;

assign DSMsign = reg2out[19];

//   Functional specification
gra20 add2(reg1in, div, reg1out, VSS);
gra20 add3(inter, reg1out, reg3out, VSS);
gra20 add4(reg2in, reg2out, inter, VSS);

//sra20_4 sra1(div, reg3out);

wbuf_2 buf1(VDD, VSS, rbb, rb);

reg20 reg1(reg1out, , reg1in, ck2, rbb);
reg20 reg2(reg2out, , reg2in, ck4, rbb);
reg20 reg3(reg3out, , EE, ck1, rbb);

endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

module EEin(tdiff, tk, ck, rb);

output [4:0] tdiff;
input [4:0] tk;
input ck, rb;

wire [4:0] reg1out, reg1in, theld, tsum;

reg5 reg1(reg1out, , reg1in, ck, rb);
reg5 buf1(tsum, , reg1out, ck, rb);
reg5 buf2(theld, , tk, ck, rb);
cla5 add1(reg1in, , tk, reg1out, 1’b0);
cls5 sub1(tdiff, , theld, tsum);

endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
```

```
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

module EEss(inc, tdiff, bit, one, p);

output [15:0] inc;
input [4:0] tdiff, one;
input [3:0] bit;
input p;

wire [14:0] shft;
wire flip, sign, nsign, xnor1;
wire [4:0] inverted, negated;
wire [3:0] mag;

//  2s Complement to Sign Magnitude Conversion
winv_1 inv1(VDD, VSS, inverted[0], tdiff[0]);
winv_1 inv2(VDD, VSS, inverted[1], tdiff[1]);
winv_1 inv3(VDD, VSS, inverted[2], tdiff[2]);
winv_1 inv4(VDD, VSS, inverted[3], tdiff[3]);
winv_1 inv5(VDD, VSS, inverted[4], tdiff[4]);
wbuf_2 buf1(VDD, VSS, sign, tdiff[4]);

cla5 add1(negated, , inverted, one, VSS);

wmux2_2 mux1(VDD, VSS, mag[3], tdiff[3], negated[3], sign);
wmux2_2 mux2(VDD, VSS, mag[2], tdiff[2], negated[2], sign);
wmux2_2 mux3(VDD, VSS, mag[1], tdiff[1], negated[1], sign);
wmux2_2 mux4(VDD, VSS, mag[0], tdiff[0], negated[0], sign);

//   Sign Logic
wxor2_2 xor1(VDD, VSS, flip, p, sign);
invmux ivm1(inc, shft, flip);

//   2^-k Logic
sr15 sr1(shft, mag, bit);

endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

module EEout(EEsign, EE, inc, ck, rb);

output [15:0] EE;
output EEsign;
input [15:0] inc;
input ck, rb;

wire [15:0] reg1in;

assign EEsign = EE[15];

gra16 add1(reg1in, inc, EE);
reg16 reg1(EE, , reg1in, ck, rb);

endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB
```

```
‘timescale 1ns/10ps

//   Finite State Machine Block

module FSM7(phi0, phi0bar, p2phi0, p2phi0b, p4phi0, p4phi0b, outstrobe, bit, readDSM,
    algorithm, clearSAR, clearEE, bit_clock, nibble_clock, cancel, reset);

//   I/O declarations

output [3:0] bit;
output phi0, phi0bar, p2phi0, p2phi0b, p4phi0, p4phi0b, outstrobe;
output bit_clock, readDSM, algorithm, clearSAR, clearEE;
input  nibble_clock, cancel, reset;

//   Internal wires

wire d0, d1, d2, d3, q0, q1, q2, q3, xq0, xq1, xq2, xq3, n01, n02, n03, n04, n05, n06, n07,
     n11, n12, n13, n14, n21, n22, n23, n24, n25, n26, n31, n32, n33, n34, n35, n36, ncancel,
     nlast, last, toggle0, toggle1, toggle2, toggle3, sample, sample1, pq0, pq1, npq0, npq1,
     pulse2, pulse4, nbit0, bc1, bc2, bc3, bc4, p2n1, p2n2, p4n1, p4n2, lastbit, readDSM,
     algorithm;

wire [3:0] current_state, next_state;

assign current_state = {q3, q2, q1, q0};
assign next_state = {d3, d2, d1, d0};

//   Functional Description
//   State register and next state logic
wdrp_4  state0(VDD, VSS, q0, xq0, d0, bit_clock, reset);
wdrp_4  state1(VDD, VSS, q1, xq1, d1, bit_clock, reset);
wdrp_4  state2(VDD, VSS, q2, xq2, d2, bit_clock, reset);
wdrp_4  state3(VDD, VSS, q3, xq3, d3, bit_clock, reset);

wand4_1 and1(VDD, VSS, n14, xq3, xq2, xq1, q0);
wand4_1 and2(VDD, VSS, n05, xq3, xq2, xq1, xq0);
wand3_1 and3(VDD, VSS, n21, xq3, xq2, q1);
wand3_1 and4(VDD, VSS, n23, xq3, q2, xq1);
wand3_1 and5(VDD, VSS, n11, xq3, xq2, xq1);
wand3_1 and6(VDD, VSS, n01, xq3, xq2, xq1);
wand3_1 and7(VDD, VSS, n03, xq3, xq2, xq0);
wand3_1 and8(VDD, VSS, n22, xq0, ncancel, nlast);
wand3_1 and9(VDD, VSS, n31, xq3, xq2, q1);
wand3_1 and10(VDD, VSS, n33, xq3, q2, xq1);
wand2_1 and11(VDD, VSS, n24, xq0, nlast);
wand2_1 and12(VDD, VSS, n25, n21, n22);
wand2_1 and13(VDD, VSS, n26, n23, n24);
wand2_1 and14(VDD, VSS, n12, q0, ncancel);
wand2_1 and15(VDD, VSS, n13, n11, n12);
wand2_1 and16(VDD, VSS, n02, xq0, ncancel);
wand2_1 and17(VDD, VSS, n06, n01, n02);
wand2_1 and18(VDD, VSS, n04, cancel, nlast);
wand2_1 and19(VDD, VSS, n07, n03, n04);
wand2_1 and20(VDD, VSS, n32, xq0, last);
wand2_1 and21(VDD, VSS, n34, xq0, last);
wand2_1 and22(VDD, VSS, n35, n31, n32);
wand2_1 and23(VDD, VSS, n36, n33, n34);

wor3_1 or1(VDD, VSS, d0, n06, n07, n05);
wor2_1 or2(VDD, VSS, d1, n13, n14);
wor2_1 or3(VDD, VSS, d2, n25, n26);
wor2_1 or4(VDD, VSS, d3, n35, n36);

winv_4 inv1(VDD, VSS, ncancel, cancel);
winv_4 inv2(VDD, VSS, nlast, last);

//  Generate last bit signal
```

```
winv_1 inv5(VDD, VSS, nbit0, bit[0]);
wand4_4 and24(VDD, VSS, last, bit[1], nbit0, bit[2], bit[3]);
wand4_1 and31(VDD, VSS, lastbit, bit[0], bit[1], bit[2], bit[3]);

//   Bit Counter
wbuf_1  del1(VDD, VSS, bckb1, bit_clock);
wbuf_4  del2(VDD, VSS, bckb2, bckb1);
wbuf_4  del3(VDD, VSS, bckb, bckb2);
wdrp_2  count0(VDD, VSS, bit[0], toggle0, toggle0, bckb, reset);
wdrp_2  count1(VDD, VSS, bit[1], toggle1, toggle1, toggle0, reset);
wdrp_2  count2(VDD, VSS, bit[2], toggle2, toggle2, toggle1, reset);
wdrp_2  count3(VDD, VSS, bit[3], toggle3, toggle3, toggle2, reset);

//   Output logic
wnor4_4 nor1(VDD, VSS, phi0, q0, q1, q2, q3);
winv_4 inv3(VDD, VSS, phi0bar, phi0);
wor2_4 or5(VDD, VSS, algorithm, q2, q3);
wnor3_4 nor2(VDD, VSS, readDSM, q1, q2, q3);

//   Bit Clock and Fast Pulse Generators
wbuf_1  del4(VDD, VSS, delp1, npq0);
wbuf_4  del5(VDD, VSS, delp2, delp1);
wbuf_4  del6(VDD, VSS, delp3, delp2);
wdrp_4  pulse1(VDD, VSS, pq0, npq0, delp3, nibble_clock, reset);
wdrp_4  pulse3(VDD, VSS, pq1, npq1, npq1, npq0, reset);

wand2_2 and25(VDD, VSS, pulse2, pq0, npq1);
wand2_2 and26(VDD, VSS, pulse4, pq0, pq1);
winv_4 inv4(VDD, VSS, bit_clock, pq1);

//   2-1 Decoders to Distribute Fast Pulses
wand2_4 and27(VDD, VSS, p2phi0, p2n1, pulse2);
wand2_4 and28(VDD, VSS, p2phi0b, p2n2, pulse2);
winv_2 inv9(VDD, VSS, p2n2, phi0);
winv_1 inv6(VDD, VSS, p2n1, p2n2);
winv_1 inv10(VDD, VSS, clearSAR, p2phi0);

wand2_4 and29(VDD, VSS, p4phi0, p4n1, pulse4);
wand2_4 and30(VDD, VSS, p4phi0b, p4n2, pulse4);
winv_2 inv7(VDD, VSS, p4n2, phi0);
winv_1 inv8(VDD, VSS, p4n1, p4n2);
winv_1 inv15(VDD, VSS, clearEE, p4phi0);

wand3_1 and32(VDD, VSS, outstrobe, lastbit, p4n2, pulse4);
endmodule


‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

//    Phi2 Selection Logic
module PHI2(t, t_bar, p, EEsign, DSMsign, algorithm, readDSM, cancel, ck, p2,rb);

//   I/O declarations

output t, t_bar;
input p, EEsign, DSMsign, algorithm, cancel, ck, p2, rb, readDSM;

//   Internal wires and storage nodes

wire out1, out2, cancel1, cancel2, min, last_t, current_t;
wire m1out, m2out, en1, en2, n1, n2, sel, dummy;

//   Functional specification
```

```
//   4-1 Mux for current_t selection
winv_1 inv1(VDD, VSS, en2, en1);
winv_2 inv2(VDD, VSS, en1, readDSM);
wbuf_2 buf1(VDD, VSS, sel, algorithm);
wmux2_2 mux1(VDD, VSS, m1out, cancel2, min, sel);
wmux2_2 mux2(VDD, VSS, m2out, cancel1, VSS, sel);
wand2_1 and1(VDD, VSS, n1, en2, m2out);
wand2_1 and2(VDD, VSS, n2, en1, m1out);
wor2_2 out(VDD, VSS, current_t, n1, n2);

//   current_t calculation logic
wxor2_2  xor1(VDD, VSS, out1, DSMsign, p);
winv_1 not1(VDD, VSS, cancel1, out1);

wxor2_2  xor2(VDD, VSS, out2, EEsign, p);
winv_1 not2(VDD, VSS, min, out2);

wxor2_2  xor3(VDD, VSS, cancel2, last_t, cancel);

//   last_t, current_t register
wdrp_2 dff1(VDD, VSS, last_t, , current_t, ck, rb);
wdrp_2 dff2(VDD, VSS, t, t_bar, current_t, p2, rb);

endmodule

//   Successive Approximation Register
‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

module SAR(cancel, outword, p, comp, ck, rb);

//   I/O declarations

output [14:0] outword;
output cancel, p;
input rb, ck, comp;

//   Internal wires and storage nodes
wire xorout, ckb, rbb;
assign p = outword[0];

//   Functional specification
wxor2_2 xor1(VDD, VSS, xorout, outword[1], outword[0]);
winv_2 not1(VDD, VSS, cancel, xorout);

wbuf_4 buf1(VDD, VSS, ckb, ck);
wbuf_4 buf2(VDD, VSS, rbb, rb);

wdrp_2 sar1(VDD, VSS, outword[0], , comp, ckb, rbb);
wdrp_2 sar2(VDD, VSS, outword[1], , outword[0], ckb, rbb);
wdrp_2 sar3(VDD, VSS, outword[2], , outword[1], ckb, rbb);
wdrp_2 sar4(VDD, VSS, outword[3], , outword[2], ckb, rbb);
wdrp_2 sar5(VDD, VSS, outword[4], , outword[3], ckb, rbb);
wdrp_2 sar6(VDD, VSS, outword[5], , outword[4], ckb, rbb);
wdrp_2 sar7(VDD, VSS, outword[6], , outword[5], ckb, rbb);
wdrp_2 sar8(VDD, VSS, outword[7], , outword[6], ckb, rbb);
wdrp_2 sar9(VDD, VSS, outword[8], , outword[7], ckb, rbb);
wdrp_2 sar10(VDD, VSS, outword[9], , outword[8], ckb, rbb);
wdrp_2 sar11(VDD, VSS, outword[10], , outword[9], ckb, rbb);
wdrp_2 sar12(VDD, VSS, outword[11], , outword[10], ckb, rbb);
wdrp_2 sar13(VDD, VSS, outword[12], , outword[11], ckb, rbb);
wdrp_2 sar14(VDD, VSS, outword[13], , outword[12], ckb, rbb);
wdrp_2 sar15(VDD, VSS, outword[14], , outword[13], ckb, rbb);
```

```
endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

//16-bit sign inversion logic
module invmux(outvec, invec14, one, flip);

output [15:0] outvec;
input [15:0] one;
input [14:0] invec14;
input flip;

wire [15:0] invec, inverted, negated;
wire pb;

// Zero pad invec
wbuf_1 buf2(VDD, VSS, invec[15], VSS);
assign invec[14:0] = invec14[14:0];

// Invert all bits and add 1
winv_1 inv15(VDD, VSS, inverted[15], invec[15]);
winv_1 inv14(VDD, VSS, inverted[14], invec[14]);
winv_1 inv13(VDD, VSS, inverted[13], invec[13]);
winv_1 inv12(VDD, VSS, inverted[12], invec[12]);
winv_1 inv11(VDD, VSS, inverted[11], invec[11]);
winv_1 inv10(VDD, VSS, inverted[10], invec[10]);
winv_1 inv9(VDD, VSS, inverted[9], invec[9]);
winv_1 inv8(VDD, VSS, inverted[8], invec[8]);
winv_1 inv7(VDD, VSS, inverted[7], invec[7]);
winv_1 inv6(VDD, VSS, inverted[6], invec[6]);
winv_1 inv5(VDD, VSS, inverted[5], invec[5]);
winv_1 inv4(VDD, VSS, inverted[4], invec[4]);
winv_1 inv3(VDD, VSS, inverted[3], invec[3]);
winv_1 inv2(VDD, VSS, inverted[2], invec[2]);
winv_1 inv1(VDD, VSS, inverted[1], invec[1]);
winv_1 inv0(VDD, VSS, inverted[0], invec[0]);

gra16 add1(negated, inverted, one);

wbuf_4 buf1(VDD, VSS, pb, flip);
wmux2_2 mux15(VDD, VSS, outvec[15], negated[15], invec[15], pb);
wmux2_2 mux14(VDD, VSS, outvec[14], negated[14], invec[14], pb);
wmux2_2 mux13(VDD, VSS, outvec[13], negated[13], invec[13], pb);
wmux2_2 mux12(VDD, VSS, outvec[12], negated[12], invec[12], pb);
wmux2_2 mux11(VDD, VSS, outvec[11], negated[11], invec[11], pb);
wmux2_2 mux10(VDD, VSS, outvec[10], negated[10], invec[10], pb);
wmux2_2 mux9(VDD, VSS, outvec[9], negated[9], invec[9], pb);
wmux2_2 mux8(VDD, VSS, outvec[8], negated[8], invec[8], pb);
wmux2_2 mux7(VDD, VSS, outvec[7], negated[7], invec[7], pb);
wmux2_2 mux6(VDD, VSS, outvec[6], negated[6], invec[6], pb);
wmux2_2 mux5(VDD, VSS, outvec[5], negated[5], invec[5], pb);
wmux2_2 mux4(VDD, VSS, outvec[4], negated[4], invec[4], pb);
wmux2_2 mux3(VDD, VSS, outvec[3], negated[3], invec[3], pb);
wmux2_2 mux2(VDD, VSS, outvec[2], negated[2], invec[2], pb);
wmux2_2 mux1(VDD, VSS, outvec[1], negated[1], invec[1], pb);
wmux2_2 mux0(VDD, VSS, outvec[0], negated[0], invec[0], pb);

endmodule

//
// Verilog description for cell sr15,
// Mon 17 Sep 2001 09:34:21 PM PDT
//
```

```
// LeonardoSpectrum Level 3, v20001a2.72
//


module sr15 ( outvec, invec, bit ) ;

    output [14:0]outvec ;
    input [3:0]invec ;
    input [3:0]bit ;

    wire nx16, nx36, nx42, nx50, nx60, nx72, nx76, nx84, nx90, nx98, nx104,
         nx197, nx199, nx201, nx204, nx206, nx208, nx211, nx214, nx221, nx229,
         nx231, nx238, nx243, nx248, nx252, nx254, nx267, nx269, nx271, nx273,
         nx275, nx277, nx283, nx285;



    nor03 ix7 (.Y (outvec[0]), .A0 (nx267), .A1 (nx199), .A2 (nx201)) ;
    nand03 ix198 (.Y (nx197), .A0 (bit[0]), .A1 (bit[1]), .A2 (invec[0])) ;
    inv04 ix200 (.Y (nx199), .A (bit[3])) ;
    inv08 ix202 (.Y (nx201), .A (bit[2])) ;
    inv02 ix207 (.Y (nx206), .A (bit[1])) ;
    mux21 ix209 (.Y (nx208), .A0 (invec[0]), .A1 (invec[1]), .S0 (bit[0])) ;
    inv02 ix215 (.Y (nx214), .A (bit[0])) ;
    nor03 ix57 (.Y (outvec[3]), .A0 (nx269), .A1 (nx199), .A2 (nx201)) ;
    aoi21 ix222 (.Y (nx221), .A0 (bit[1]), .A1 (nx50), .B0 (nx42)) ;
    nor02 ix43 (.Y (nx42), .A0 (bit[1]), .A1 (nx208)) ;
    and02 ix79 (.Y (outvec[4]), .A0 (bit[3]), .A1 (nx76)) ;
    ao21 ix77 (.Y (nx76), .A0 (bit[2]), .A1 (nx72), .B0 (nx60)) ;
    mux21 ix73 (.Y (nx72), .A0 (nx229), .A1 (nx271), .S0 (bit[1])) ;
    mux21 ix230 (.Y (nx229), .A0 (invec[1]), .A1 (invec[2]), .S0 (bit[0])) ;
    nand02 ix232 (.Y (nx231), .A0 (nx214), .A1 (invec[3])) ;
    nor02 ix61 (.Y (nx60), .A0 (bit[2]), .A1 (nx267)) ;
    and02 ix93 (.Y (outvec[5]), .A0 (bit[3]), .A1 (nx90)) ;
    ao21 ix91 (.Y (nx90), .A0 (nx201), .A1 (nx283), .B0 (nx84)) ;
    nor02 ix17 (.Y (nx16), .A0 (nx206), .A1 (nx208)) ;
    nor03 ix85 (.Y (nx84), .A0 (nx273), .A1 (nx201), .A2 (bit[1])) ;
    mux21 ix239 (.Y (nx238), .A0 (invec[2]), .A1 (invec[3]), .S0 (bit[0])) ;
    and02 ix107 (.Y (outvec[6]), .A0 (bit[3]), .A1 (nx104)) ;
    ao21 ix105 (.Y (nx104), .A0 (nx201), .A1 (nx285), .B0 (nx98)) ;
    oai21 ix37 (.Y (nx36), .A0 (nx206), .A1 (nx229), .B0 (nx243)) ;
    nand03 ix244 (.Y (nx243), .A0 (bit[0]), .A1 (nx206), .A2 (invec[0])) ;
    nor03 ix99 (.Y (nx98), .A0 (nx271), .A1 (nx201), .A2 (bit[1])) ;
    nor03 ix113 (.Y (outvec[7]), .A0 (nx269), .A1 (nx199), .A2 (bit[2])) ;
    oai22 ix137 (.Y (outvec[8]), .A0 (nx248), .A1 (nx275), .B0 (nx267), .B1 (
          nx277)) ;
    nand02 ix253 (.Y (nx252), .A0 (bit[3]), .A1 (nx201)) ;
    nand02 ix255 (.Y (nx254), .A0 (nx199), .A1 (bit[2])) ;
    oai32 ix147 (.Y (outvec[9]), .A0 (nx275), .A1 (bit[1]), .A2 (nx273), .B0 (
          nx204), .B1 (nx277)) ;
    oai32 ix157 (.Y (outvec[10]), .A0 (nx275), .A1 (bit[1]), .A2 (nx271), .B0 (
          nx211), .B1 (nx277)) ;
    nor03 ix119 (.Y (outvec[11]), .A0 (nx269), .A1 (bit[3]), .A2 (nx201)) ;
    and02 ix123 (.Y (outvec[12]), .A0 (nx199), .A1 (nx76)) ;
    and02 ix127 (.Y (outvec[13]), .A0 (nx199), .A1 (nx90)) ;
    and02 ix131 (.Y (outvec[14]), .A0 (nx199), .A1 (nx104)) ;
    inv02 ix249 (.Y (nx248), .A (nx72)) ;
    inv02 ix51 (.Y (nx50), .A (nx238)) ;
    inv02 ix212 (.Y (nx211), .A (nx36)) ;
    inv02 ix205 (.Y (nx204), .A (nx16)) ;
    buf02 ix266 (.Y (nx267), .A (nx197)) ;
    buf02 ix268 (.Y (nx269), .A (nx221)) ;
    buf02 ix270 (.Y (nx271), .A (nx231)) ;
    inv02 ix272 (.Y (nx273), .A (nx50)) ;
    buf02 ix274 (.Y (nx275), .A (nx252)) ;
    buf02 ix276 (.Y (nx277), .A (nx254)) ;
    and03 ix19 (.Y (outvec[1]), .A0 (nx283), .A1 (bit[3]), .A2 (bit[2])) ;
```

```
    and03 ix39 (.Y (outvec[2]), .A0 (nx285), .A1 (bit[3]), .A2 (bit[2])) ;
    inv02 ix282 (.Y (nx283), .A (nx204)) ;
    inv02 ix284 (.Y (nx285), .A (nx211)) ;
endmodule


module and03 ( Y, A0, A1, A2 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input A2 ;



    wand3_1 and1(VDD, VSS, Y, A0, A1, A2) ;
endmodule


module buf02 ( Y, A ) ;

    output Y ;
    input A ;



    wbuf_2 buf1(VDD, VSS, Y, A);
endmodule


module oai32 ( Y, A0, A1, A2, B0, B1 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input A2 ;
    input B0 ;
    input B1 ;

    wire n1, n2, n3, n4, n5, n6, n7, n8;

    winv_1 inv1(VDD, VSS, n1, B0);
    winv_1 inv2(VDD, VSS, n2, B1);
    winv_1 inv3(VDD, VSS, n3, A0);
    winv_1 inv4(VDD, VSS, n4, A1);
    winv_1 inv5(VDD, VSS, n6, A2);

    wand2_1 and1(VDD, VSS, n7, n1, n2);
    wand2_1 and2(VDD, VSS, n5, n3, n4);
    wand2_1 and3(VDD, VSS, n8, n5, n6);

    wor2_1 or1(VDD, VSS, Y, n7, n8);
endmodule


module oai22 ( Y, A0, A1, B0, B1 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input B0 ;
    input B1 ;

    wire NOT_B0, NOT_B1, nx4, NOT_A0, NOT_A1, nx10;
```

```
    winv_1 inv1(VDD, VSS, NOT_B0, B0);
    winv_1 inv2(VDD, VSS, NOT_B1, B1);
    wand2_1 and1(VDD, VSS, nx4, NOT_B0, NOT_B1) ;
    winv_1 inv3(VDD, VSS, NOT_A0, A0);
    winv_1 inv4(VDD, VSS, NOT_A1, A1);
    wand2_1 and2(VDD, VSS, nx10, NOT_A0, NOT_A1) ;
    wor2_1 or1(VDD, VSS, Y, nx4, nx10) ;
endmodule


module oai21 ( Y, A0, A1, B0 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input B0 ;

    wire NOT_A0, NOT_A1, nx4, NOT_B0;



    winv_1 inv1(VDD, VSS, NOT_A0, A0);
    winv_1 inv2(VDD, VSS, NOT_A1, A1);
    wand2_1 and1(VDD, VSS, nx4, NOT_A0, NOT_A1) ;
    winv_1 inv3(VDD, VSS, NOT_B0, B0);
    wor2_1 or1(VDD, VSS, Y, nx4, NOT_B0) ;
endmodule


module nand02 ( Y, A0, A1 ) ;

    output Y ;
    input A0 ;
    input A1 ;


    wnand2_1 nand1(VDD, VSS, Y, A0, A1);
endmodule


module ao21 ( Y, A0, A1, B0 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input B0 ;

    wire nx0;



    wand2_1 and1(VDD, VSS, nx0, A0, A1) ;
    wor2_1 or1(VDD, VSS, Y, nx0, B0) ;
endmodule


module and02 ( Y, A0, A1 ) ;

    output Y ;
    input A0 ;
    input A1 ;



    wand2_1 and1(VDD, VSS, Y, A0, A1) ;
```

```
endmodule


module nor02 ( Y, A0, A1 ) ;

    output Y ;
    input A0 ;
    input A1 ;



    wnor2_1 nor1(VDD, VSS, Y, A0, A1);
endmodule


module aoi21 ( Y, A0, A1, B0 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input B0 ;

    wire NOT_A1, NOT_B0, nx4, NOT_A0, nx8;



    winv_1 inv1(VDD, VSS, NOT_A1, A1);
    winv_1 inv2(VDD, VSS, NOT_B0, B0);
    wand2_1 and1(VDD, VSS, nx4, NOT_A1, NOT_B0) ;
    winv_1 inv3(VDD, VSS, NOT_A0, A0);
    wand2_1 and2(VDD, VSS, nx8, NOT_A0, NOT_B0) ;
    wor2_1 or1(VDD, VSS, Y, nx4, nx8) ;
endmodule


module mux21 ( Y, A0, A1, S0 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input S0 ;

    wire muxout;


    wmux2_2 mux1(VDD, VSS, muxout, A0, A1, S0);
    winv_2 inv1(VDD, VSS, Y, muxout);
endmodule


module inv02 ( Y, A ) ;

    output Y ;
    input A ;



    winv_1 inv1(VDD, VSS, Y, A);
endmodule


module inv08 ( Y, A ) ;

    output Y ;
    input A ;
```

```
    winv_4 inv1(VDD, VSS, Y, A);
endmodule


module inv04 ( Y, A ) ;

    output Y ;
    input A ;



    winv_2 inv1(VDD, VSS, Y, A);
endmodule


module nand03 ( Y, A0, A1, A2 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input A2 ;



    wnand3_1 nand1(VDD, VSS, Y, A0, A1, A2);
endmodule


module nor03 ( Y, A0, A1, A2 ) ;

    output Y ;
    input A0 ;
    input A1 ;
    input A2 ;



    wnor3_1 nor1(VDD, VSS, Y, A0, A1, A2);
endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

//  20-bit Group Ripple Adder based on 5-bit Carry Look Ahead Adders
module gra20(sum, a, b);

output [19:0] sum;
input [19:0] a, b;

wire c0, c1, c2, c3;


cla5  add1(sum[4:0], c1, a[4:0], b[4:0], VSS);
cla5  add2(sum[9:5], c2, a[9:5], b[9:5], c1);
cla5  add3(sum[14:10], c3, a[14:10], b[14:10], c2);
cla5  add4(sum[19:15], , a[19:15], b[19:15], c3);

endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB
```

```
‘timescale 1ns/10ps

//  5-bit Carry Look Ahead Adder
module cla5(sum, cout, a, b, c0);

output [4:0] sum;
output cout;
input [4:0] a, b;
input c0; // c0 = cin

wire c0, p0, g0, p1, g1, p2, g2, p3, g3, p4, g4, hs0, hs1, hs2, hs3, hs4;
wire c51, c52, c53, c54, c55, c56, c57, c58, c59, c510, c41, c42, c43,
     c44, c45, c46, c47, c31, c32, c33, c21, c22, c11;

//  c5 = cout
wand4_1 and51(VDD, VSS, c57, p4, p3, p2, g1);
wand3_1 and52(VDD, VSS, c51, p4, p3, p2);
wand3_1 and53(VDD, VSS, c52, p1, p0, c0);
wand3_1 and54(VDD, VSS, c53, p4, p3, p2);
wand3_1 and55(VDD, VSS, c58, p4, p3, g2);
wand2_1 and56(VDD, VSS, c54, p1, g0);
wand2_1 and57(VDD, VSS, c55, c51, c52);
wand2_1 and58(VDD, VSS, c56, c53, c54);
wand2_1 and59(VDD, VSS, c59, p4, g3);
wor4_1 or51(VDD, VSS, c510, c57, c58, c59, g4);
wor3_1 or52(VDD, VSS, cout, c55, c56, c510);

//  c4
wand4_1 and41(VDD, VSS, c43, p3, p2, p1, g0);
wand3_1 and42(VDD, VSS, c41, p3, p2, p1);
wand3_1 and43(VDD, VSS, c44, p3, p2, g1);
wand2_1 and44(VDD, VSS, c42, p0, c0);
wand2_1 and45(VDD, VSS, c46, c41, c42);
wand2_1 and46(VDD, VSS, c45, p3, g2);
wor4_1 or41(VDD, VSS, c47, c43, c44, c45, g3);
wor2_1 or42(VDD, VSS, c4, c47, c46);

//  c3
wand4_1 and31(VDD, VSS, c31, p2, p1, p0, c0);
wand3_1 and32(VDD, VSS, c32, p2, p1, g0);
wand2_1 and33(VDD, VSS, c33, p2, g1);
wor4_1 or31(VDD, VSS, c3, c31, c32, c33, g2);

//  c2
wand3_1 and21(VDD, VSS, c21, p1, p0, c0);
wand2_1 and22(VDD, VSS, c22, p1, g0);
wor3_1 or21(VDD, VSS, c2, c21, c22, g1);

//  c1
wand2_1 and11(VDD, VSS, c11, p0, c0);
wor2_1 or11(VDD, VSS, c1, c11, g0);

//  Propagate
wor2_4 orp4(VDD, VSS, p4, a[4], b[4]);
wor2_4 orp3(VDD, VSS, p3, a[3], b[3]);
wor2_4 orp2(VDD, VSS, p2, a[2], b[2]);
wor2_4 orp1(VDD, VSS, p1, a[1], b[1]);
wor2_4 orp0(VDD, VSS, p0, a[0], b[0]);

//  Generate
wand2_4 andg4(VDD, VSS, g4, a[4], b[4]);
wand2_4 andg3(VDD, VSS, g3, a[3], b[3]);
wand2_4 andg2(VDD, VSS, g2, a[2], b[2]);
wand2_4 andg1(VDD, VSS, g1, a[1], b[1]);
wand2_4 andg0(VDD, VSS, g0, a[0], b[0]);

//  Half-Adders
```

```
wxor2_2 ha41(VDD, VSS, hs4, a[4], b[4]);
wxor2_2 ha42(VDD, VSS, sum[4], c4, hs4);
wxor2_2 ha31(VDD, VSS, hs3, a[3], b[3]);
wxor2_2 ha32(VDD, VSS, sum[3], c3, hs3);
wxor2_2 ha21(VDD, VSS, hs2, a[2], b[2]);
wxor2_2 ha22(VDD, VSS, sum[2], c2, hs2);
wxor2_2 ha11(VDD, VSS, hs1, a[1], b[1]);
wxor2_2 ha12(VDD, VSS, sum[1], c1, hs1);
wxor2_2 ha01(VDD, VSS, hs0, a[0], b[0]);
wxor2_2 ha02(VDD, VSS, sum[0], c0, hs0);

endmodule

‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

//  5-bit Carry Look Ahead Subtractor (a-b)
//  This subtractor is NOT cascadable

module cls5(diff, bout, a, b);

output [4:0] diff;
output bout;
input [4:0] a, b;

wire b0;
wire [4:0] nb;

winv_1 inv1(VDD, VSS, nb[0], b[0]);
winv_1 inv2(VDD, VSS, nb[1], b[1]);
winv_1 inv3(VDD, VSS, nb[2], b[2]);
winv_1 inv4(VDD, VSS, nb[3], b[3]);
winv_1 inv5(VDD, VSS, nb[4], b[4]);

cla5  add1(diff[4:0], bout, a[4:0], nb[4:0], 1’b1);

endmodule

//   20 bit register with asyncronous reset
‘define LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/nwb libext=.v
‘define UDP_LIB dir=/nfs/guille/analog/c/cdsmgr/lib97A/cmosp35.3.0/models/verilog/udp libext=.v
‘uselib ‘LIB ‘UDP_LIB

‘timescale 1ns/10ps

module reg20(outvec, outvecb, invec, ck, rb);

//   I/O declarations
output [19:0] outvec, outvecb;
input [19:0] invec;
input ck, rb;

//   Internal wires, storage nodes, and constants
wire ckb, rbb;

//   Functional specification
wbuf_4 buf1(VDD, VSS, ckb, ck);
wbuf_4 buf2(VDD, VSS, rbb, rb);

wdrp_2 dff1(VDD, VSS, outvec[0], outvecb[0], invec[0], ckb, rbb);
wdrp_2 dff2(VDD, VSS, outvec[1], outvecb[1], invec[1], ckb, rbb);
wdrp_2 dff3(VDD, VSS, outvec[2], outvecb[2], invec[2], ckb, rbb);
wdrp_2 dff4(VDD, VSS, outvec[3], outvecb[3], invec[3], ckb, rbb);
wdrp_2 dff5(VDD, VSS, outvec[4], outvecb[4], invec[4], ckb, rbb);
```

```
wdrp_2 dff6(VDD, VSS, outvec[5], outvecb[5], invec[5], ckb, rbb);
wdrp_2 dff7(VDD, VSS, outvec[6], outvecb[6], invec[6], ckb, rbb);
wdrp_2 dff8(VDD, VSS, outvec[7], outvecb[7], invec[7], ckb, rbb);
wdrp_2 dff9(VDD, VSS, outvec[8], outvecb[8], invec[8], ckb, rbb);
wdrp_2 dff10(VDD, VSS, outvec[9], outvecb[9], invec[9], ckb, rbb);
wdrp_2 dff11(VDD, VSS, outvec[10], outvecb[10], invec[10], ckb, rbb);
wdrp_2 dff12(VDD, VSS, outvec[11], outvecb[11], invec[11], ckb, rbb);
wdrp_2 dff13(VDD, VSS, outvec[12], outvecb[12], invec[12], ckb, rbb);
wdrp_2 dff14(VDD, VSS, outvec[13], outvecb[13], invec[13], ckb, rbb);
wdrp_2 dff15(VDD, VSS, outvec[14], outvecb[14], invec[14], ckb, rbb);
wdrp_2 dff16(VDD, VSS, outvec[15], outvecb[15], invec[15], ckb, rbb);
wdrp_2 dff17(VDD, VSS, outvec[16], outvecb[16], invec[16], ckb, rbb);
wdrp_2 dff18(VDD, VSS, outvec[17], outvecb[17], invec[17], ckb, rbb);
wdrp_2 dff19(VDD, VSS, outvec[18], outvecb[18], invec[18], ckb, rbb);
wdrp_2 dff20(VDD, VSS, outvec[19], outvecb[19], invec[19], ckb, rbb);

endmodule
```