Design of the Digital Control Logic for a 12-Bit Two-Step Flash ADC

by Naga Chaitanya Yelchuri, Bachelor of Science

A Thesis Submitted in Partial
Fulfillment of the Requirements
for the Master of Science Degree

Department of Electrical and Computer Engineering
in the Graduate School
Southern Illinois University Edwardsville
Edwardsville, Illinois

December 2009

ABSTRACT

DESIGN OF THE DIGITAL CONTROL LOGIC FOR A 12-BIT TWO-STEP FLASH ADC

by

Naga Chaitanya Yelchuri

Advisor: Dr. George L Engel

This thesis presents the design of the digital control logic for a 12-bit, 2 MSample/sec two-step flash Analog-to-Digital Converter (ADC). A standard cell library compatible with the AMI (American MicroSystems Incorporated) design kit was created from an existing OSU (Oklahoma State University) 0.5 μm AMI standard cell library using a script written in Perl. The thesis describes the design flow for generating automatic layout compatible with the AMI design kit beginning with the creation of behavioral Verilog code to describe the control logic. The control logic described herein is for an ADC that is intended for use in a family of integrated circuits (ICs) used in the detection of ionizing radiation. The ICs are being developed by the IC Design Laboratory at Southern Illinois University Edwardsville (SIUE).

The converter described in this thesis employs a two-step flash technique employing a resistive DAC and is configured as a fully differential circuit. It performs 7-bit coarse flash conversion followed by 6-bit fine flash conversion. The results from the two steps are combined using a digital error correction algorithm to produce the 12-bit output. The completed automated layout of the digital control logic was performed, and the area of the resulting circuit was found to be 1.55mm^2 (2035μm x 760μm). Electrical simulations performed on the ADC produced the desired output. The effective number of bits (ENOB) for the converter in the

absence of typical offsets and mismatch errors was found to be 11.9 bits. This work was initiated by the heavy-ion nuclear chemistry and physics group at Washington University in Saint Louis and was funded by NSF Grant #06118996 and a grant from Los Alamos National Laboratory (LANL).

ACKNOWLEDGEMENTS

I wish to thank my advisor, Dr. George Engel, whose encouragement, guidance and support from the initial to the final level enabled me to complete this project. I would also like to thank my fellow researchers: Dinesh Kumar Dasari, Taraka Neelakanteswara Rao Yerra, Hara Valluru, Satya Mohan Raju Gudidevuni and Nam Nguyen for their constant support.

A thank you goes out to Dr. Scott Smith, Dr. Oktay Alkin and other ECE faculty members for their support in completing my studies at SIUE. Last, but not least, I would like to thank Arokia Nirmal for his help in the preparation of the drawings included in this thesis. I am grateful to my mother, my brother Pavan Kumar Yelchuri, cousins and friends for their constant guidance and support in completing my Masters.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Background

The IC Design Research Laboratory at Southern Illinois University Edwardsville (SIUE) is part of an interuniversity collaboration which has as its long-term goal the development of a family of multi-channel custom integrated circuits (ICs) suitable for use in a wide variety of low-energy and intermediate-energy nuclear physics experiments where the detection of ionizing radiation is needed [Spi:05]. The greater collaboration includes researchers at Washington University in Saint Louis (WUSTL), Michigan State University (MSU), Western Michigan University (WMU), and Indiana University (IU).

The reasons why the collaboration became interested in developing a family of custom chips for use in a wide assortment of nuclear physics experiments include: (1) the desire for high density signal processing in the low-energy and intermediate-energy nuclear physics community is widespread, (2) no commercial chips were available that were capable of doing precisely what the researchers wanted, and (3) the scientists deemed it necessary for the "experimenter" to be in the "designer's seat" with the latter the most important of the three reasons listed.

Preliminary work on the ICs began in the year 2000 [Gan:00, Mal:01]. Initially, the chip development was funded through support from the nuclear reactions group located in the Department of Chemistry at WUSTL. Because of the early success of the project; however, the work was later funded by the National Science Foundation (NSF Grant #06118996). Currently, the work is sponsored by a group of researchers at Los Alamos National Laboratories (LANL) under the direction of Dr. Mark Wallace. It is the generous support of this LANL group along

with support from NSF that has made much of the work described in this thesis possible.

To date, the IC design research group at SIUE has developed two integrated circuits. The first IC which the group designed and fabricated was a sixteen-channel shaped and peak-sensing analog chip referred to as HINP16C [Sad02, Eng:07a]. The chip is used in applications were excellent energy resolution is required, but the researcher is not interested in particle identification. The second IC, known as PSD8C [Hal:07, Pro:07, Eng:09], is an eight-channel analog IC which performs pulse-shape discrimination (PSD) and thus is capable of particle identification if the time dependence of the light output of the scintillator depends on particle type. The two chips, HINP16C and PSD8C, logically complement one another.

Both of these ICs produce sparsified analog pulse train outputs along with synchronized addresses (chip and channel) for off-chip digitization with a pipelined ADC. While both chips perform well, low-level analog output signals are sensitive to picking up environmental noise. Moreover, currently the analog signals from the various channels associated with dozens of chips in a typical system are digitized sequentially. This can lead to relatively long acquisition times if the analog signals from many channels must be digitized by a single ADC.

This thesis looks at incorporating the ADC, along with a RAM buffer to hold the results and an I^2C-like serial interface to transmit the digital data to a host, onto the custom ICs themselves. This thesis represents a continuation of the work begun by earlier graduate students Dasari [Das:08], Valluru [Val:08], and Nguyen [Ngu:08]. While the work presented here could be used with either the HINP16C or PSD8C chips, for sake of brevity we will just describe the most recent IC, PSD8C, and how incorporating the ADC and its companion circuits onto the chip can

improve overall system performance. While there are no immediate plans to integrate the ADC onto the PSD8C chip it is the more likely candidate than is HINP16C for this endeavor at some point in the future. It is for this reason why we will briefly describe [Eng:09] the PSD8C micro-chip and the current PSD8C system.

PSD8C Integrated Circuit

The PSD8C IC was fabricated in the AMI 0.5 μm n-well process (C5N) available through MOSIS (MOS Implementation Services). The PSD8C design makes use of this CMOS technology to: a) provide integrations of several regions of the analog pulse generated by the detector, b) provide time-to-voltage conversion, and c) prepare each of the above as analog data streams for a pipelined ADC. A typical implementation of a system using the PSD8C chip is illustrated in Figure 1.1.

As shown in Figure 1.1, each detector output must be split to provide signals for both the “logic” and “linear” branches. The timing signals are generated by off-chip discriminators. The linear signals are delayed, either by cable (when retention of the signal shape is essential) or by LC delay chips.

The individual timing signals and delayed linear signals are sent to the PSD8C chip. The individual discriminator signals, logically ANDed with a global enable signal, provide individual channel enables. For each linear signal, three different integrations (named A, B and C) can be performed with start times referenced to the individual discriminator firings. In addition, an amplitude, T, is produced which is proportional to the difference in time between the individual discriminator and an external common stop reference. The T amplitude eliminates the need for conventional TDCs (time-to-digital converters).

This amplitude and the three integrals are sequenced to a *single* pipelined ADC. As stated earlier, if PSD8C possessed an on-chip ADC, then every chip in the system would digitize the data concurrently. This would greatly improve the acquisition time whenever a large number of discriminators fire!

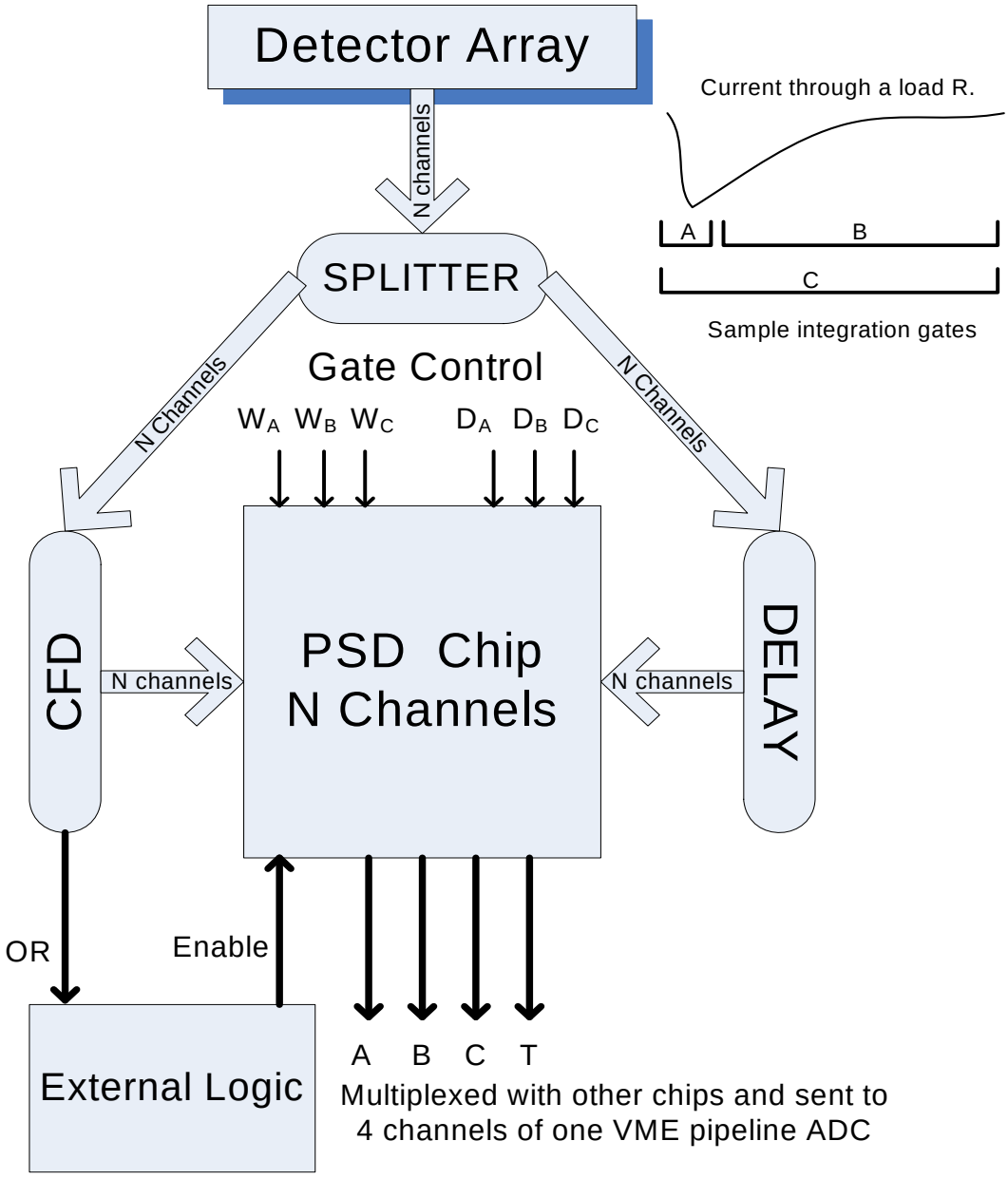

Figure 1.1: Illustration of typical system employing PSD IC (N = 8)

The delays in the integrators' starting times (D_A, D_B, D_C) and the widths (W_A, W_B, W_C) of the integration windows are controlled by the user on a chip-by-chip basis. In Figure 1.1, both the delays (voltage-controlled) and the widths (voltage-controlled) are converted to times on chip. When a channel's associated discriminator fires, it starts a time-to-voltage converter (TVC). The TVC circuit has two selectable full-scale measurement ranges: 500 ns and 2 μs. The charging concludes with a common stop signal applied to all channels. The TVC circuit and the integrators are automatically reset after a user-controlled variable delay time, referenced to when the discriminator fires.

In order to acquire the analog information, the user must supply a pulse to veto the reset. This veto thus selects an event for readout and digitization, and it must be delivered to the chip before the aforementioned user-controlled delay times out. The fast logical 'OR' signal and an analog output which is proportional to the number of channels that were hit, 'MULT', are available for off-chip high-level logical decisions and to decide, for example, if the veto of the reset is to be sent. The logical 'OR' and 'MULT' are also automatically reset unless vetoed by the user.

A central *common* channel provides biasing for the eight processing channels, contains the readout electronics, and also contains the logic used to configure the chip. A 48-bit configuration register allows the user to select: processing for either positive or negative input pulses, bias-mode ("high" or "low"), TVC measurement range, charging rates, delays, integration widths, and allows the user to selectively disable channels.

A representative channel is illustrated in Figure 1.2. Each channel in PSD8C is composed of three sub-channels, a time-to-voltage converter (TVC), and read-out related electronics. The three sub-channels are identical in topology. The sub-channels produce the three different integrations (referred to as A, B and C).

The TVC produces the amplitude, T, which is proportional to the difference in time between the channel's discriminator firing and an external common-stop reference.

Each sub-channel consists of a gated integrator and a corresponding gate generator. The gate generator is formed by a pair of externally programmable Voltage-to-Time Converters (VTC) that define the start and width of the integration period. A representative sub-channel is depicted in Figure 1.3. The *Event* signal in Figure 1.3 is either the CFD_i (i^{th} channel discriminator signal) or the *GlobalEn* signal (depending on the mode as described above). The gate generator (described below) produces a *Startint*, a *StopInt*, and a *DUMP* signal.

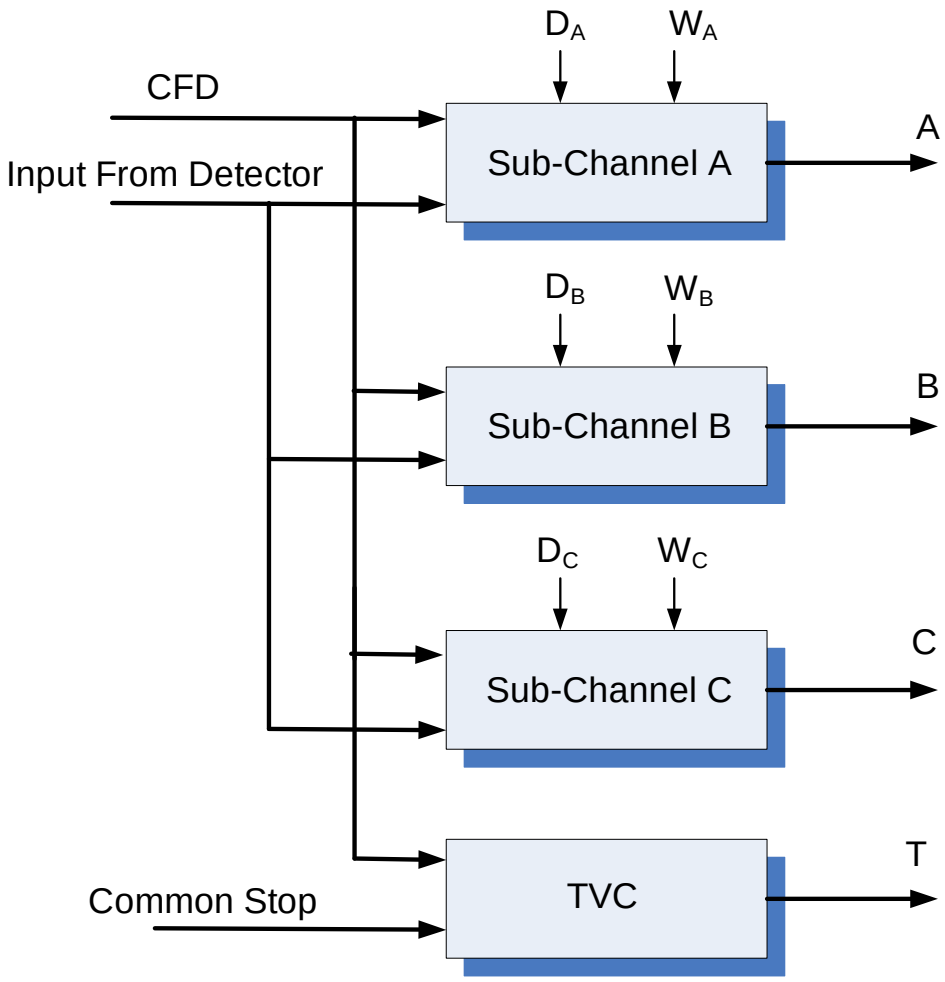

Figure 1.2: Structure of a representative PSD8C channel

The *Startint* signal starts the integrator; the *Stopint* signal stops the integrator; and the *DUMP* signal when asserted resets the integrator. The *Startint* and *StopInt* signals are reset and the DUMP signal is preset when either the user issues a *ForceRst* (pin on chip which affects all of the channels on the chip) or when an auto reset occurs as described above.

As shown Figure 1.3, one VTC (left) determines the start of the integration period relative to the channel's discriminator signal. The second VTC (right) determines the duration of the integration. The start of the integration period for the A sub-channel is set by an externally supplied voltage. Similarly, the period of integration for all A sub-channels on the chip, is set by a second externally supplied voltage. The same can be said for the B and C sub-channels.

The layout of the PSD8C chip (2748 μm x 5659 μm) is presented in Figure 1.4. The biasing and circuits used for configuring the IC, as well as for readout, are located in the center ("common" channel) of the chip.

The PSD8C chips have been successfully integrated into a prototype system. Our present implementation components are shown in Figure 1.5. The detector outputs are initially routed to the ASD (Amplifier-Splitter-Delay - ASD) board. The ASD amplifies and splits the detector signals and delays one branch. The amplified but "non-delayed" branch drives a discriminator while the amplified and delayed branch provides the analog inputs to PSD8C.

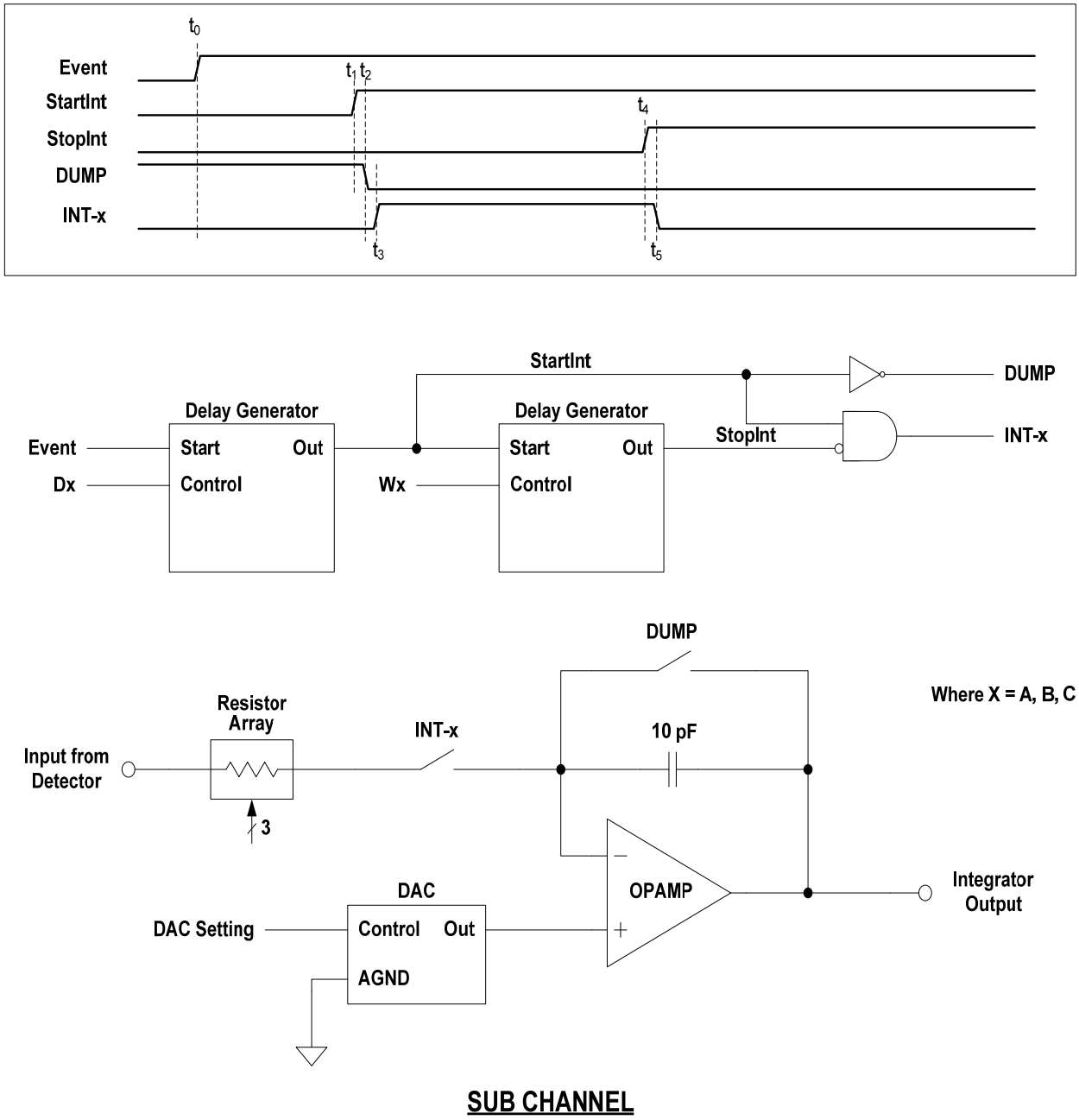

Figure 1.3: Structure of a sub-channel

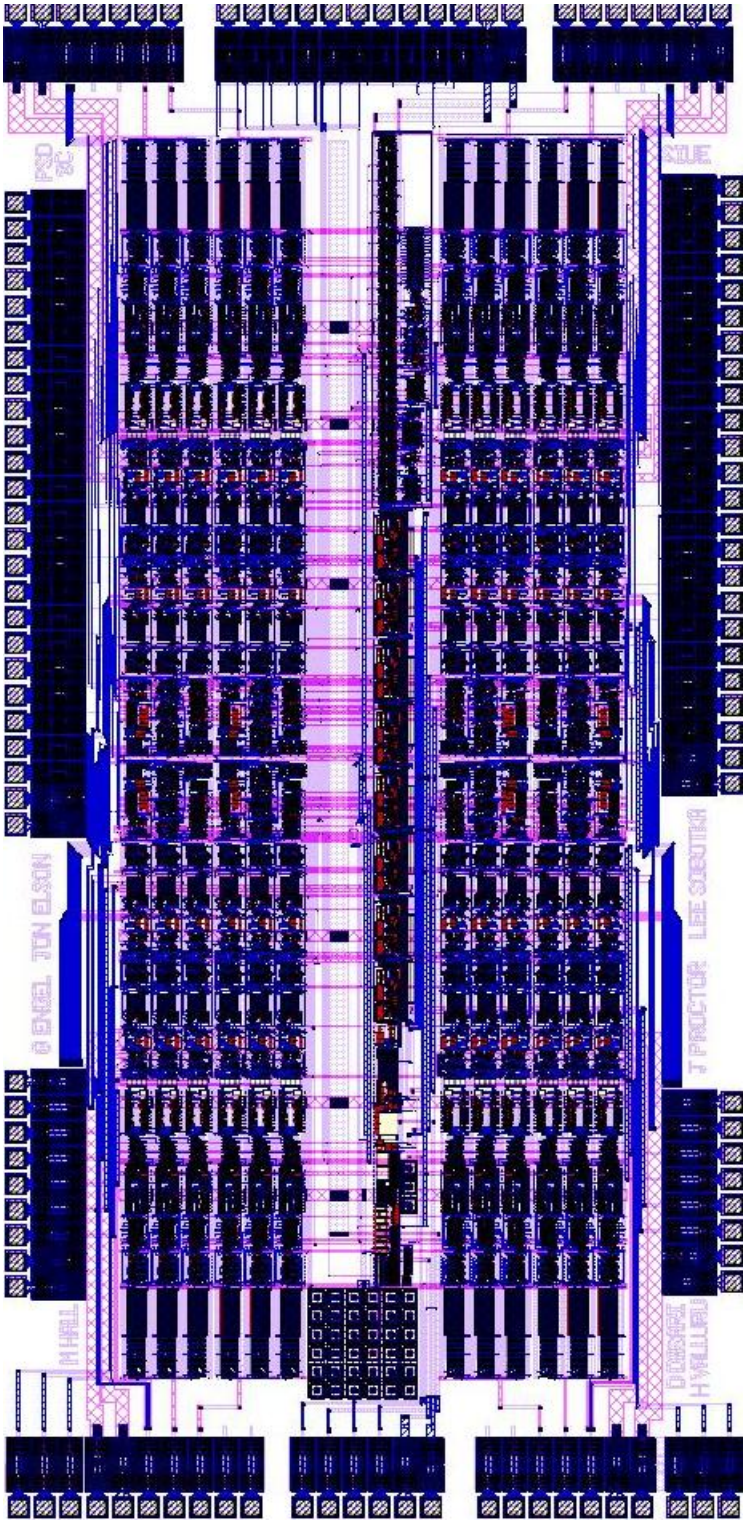

Figure 1.4: Layout of PSD8C

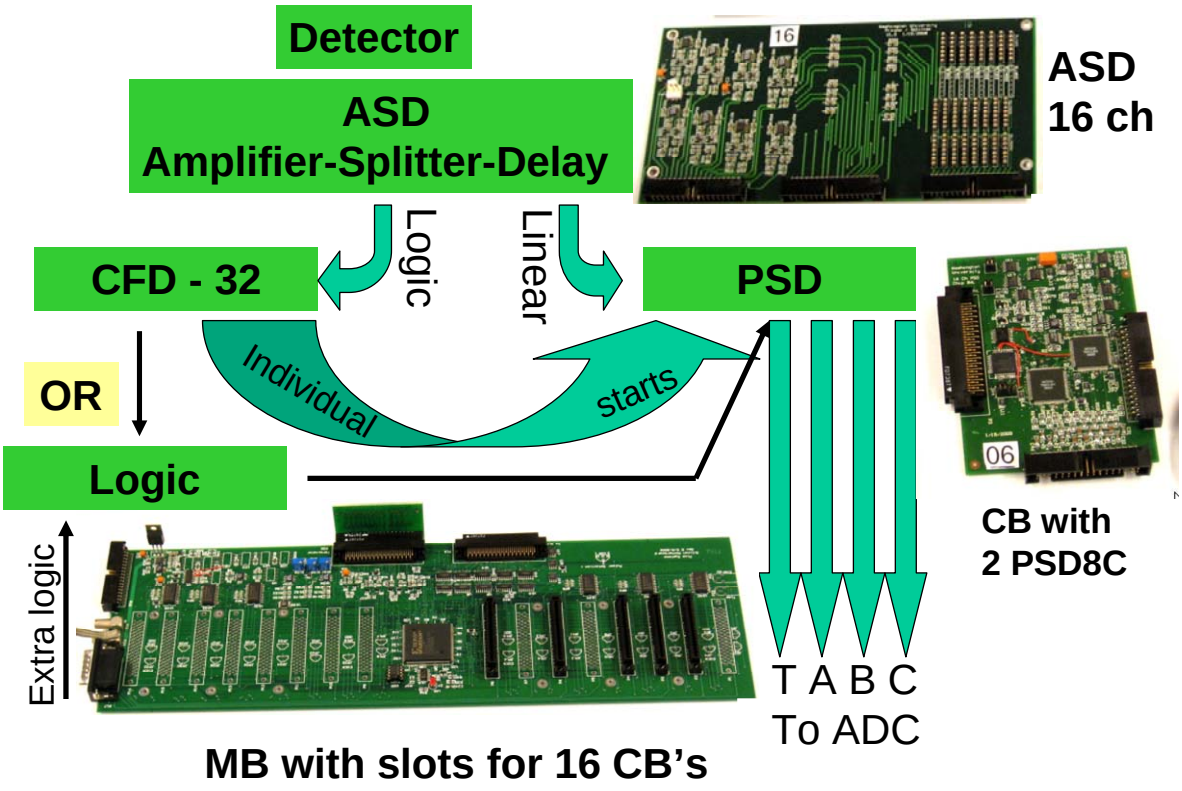

Figure 1.5: Present implementation strategy and components

A Particle-IDentification (PID) map for a BC501A liquid-scintillator detector used for γ-n discrimination is shown in Figure 1.6. The bottom locus corresponds to gamma rays while the top to neutrons. This figure demonstrates PSD8C's ability to perform particle identification [Eng:09]. PSD8C is also useful in non-PID applications where only modest energy resolution is needed.

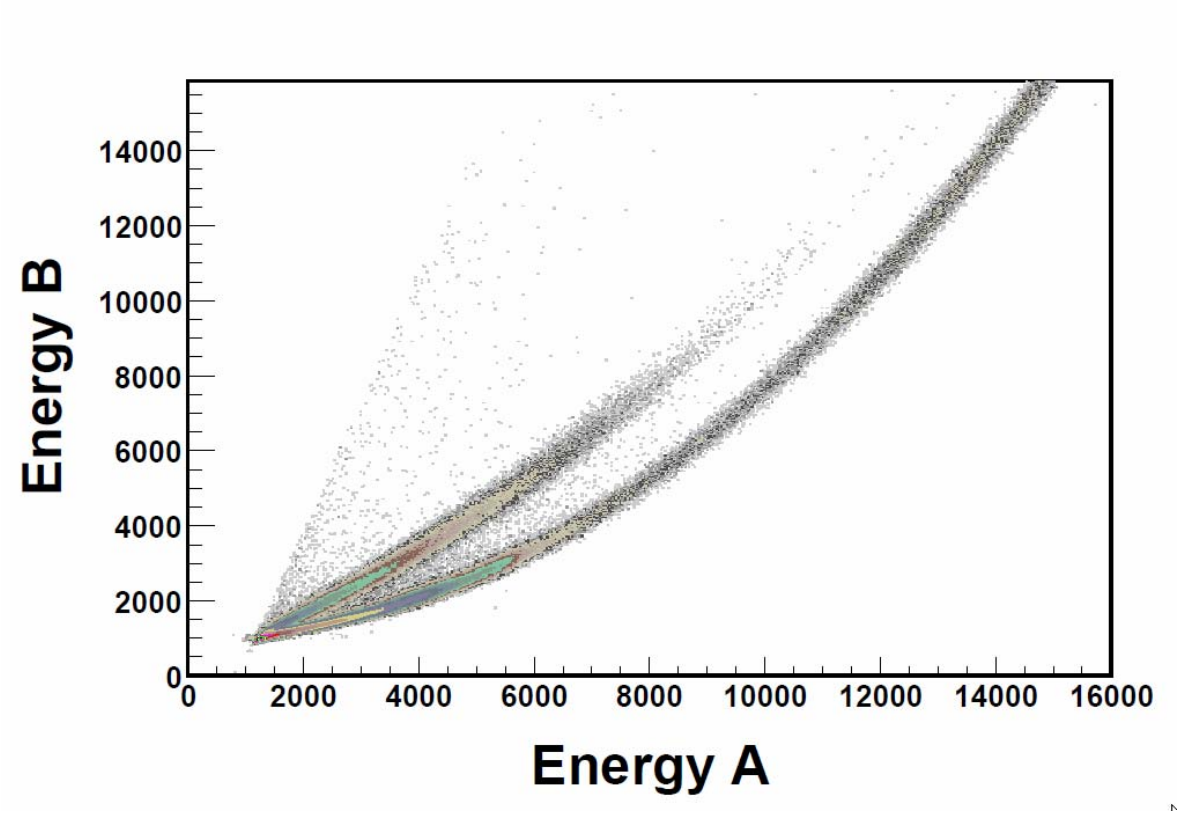

Figure 1.6: PID map taken using PSD8C chip

Need for On-Chip ADC

Prominent in the previous section is the fact that PSD8C outputs (A, B, C, T) come off chip in an analog format. The transmission of sensitive high-speed analog pulse trains to an off-chip VME ADC is a very difficult task [Das:08]. Low-level analog signals are easily corrupted. Moreover, even if the system is correctly designed, it is the nuclear physicists who set up and conduct the experiments. It is easy to degrade the performance of an analog system through improper grounding or shielding. The transmission of high-speed digital signals even over long distances is, on the other hand, relatively easy. Digital signals have high noise immunity.

Furthermore, VME ADCs are relatively expensive, large, and power-hungry. As described in an earlier section, it is important that systems proposed for use in nuclear threat detection systems to be used by the Department of Homeland Security (DHS) be compact and low-power. These attributes can best be achieved if an on-chip ADC is employed. Since the IC described in the previous section reasonably large (15 mm^2), the proposed ADC [Das:08] and companion RAM [Val:08], used to store the digitized results, will increase the overall chip area (and chip costs) by an estimated 10 – 30%.

PSD8C possesses eight channels. Each channel consists of 3 sub-channels and a time-to-voltage converter (TVC). The digitization when the proposed onboard ADC is used can be completed in a maximum of 16 μs. *This is true regardless of the number of chips in the system since all of the chips would digitize in parallel.* Clearly, incorporating the ADC on-chip makes the most sense if the PSD8C chip is to be used in systems with a very large number of detectors (several thousand) the Only a small 32 location memory is needed per chip to store the digitized values. Finally, it is possible to transmit the digital data out of the PSD8C chip back to a host at a very high rate (in excess of 20 Mbits/sec). Therefore, transmission of even large amounts of digital data will take a relatively short period of time.

To minimize the system-level interconnect, we propose to transmit the contents of the RAM storing the ADC results back to a host computer via a serial "I2C-like" interface [Ngu:08]. This greatly simplifies the system level design and in particular the design of mother board and associated chip boards.

Object and Scope of Thesis

The object of this thesis is to describe the design of the digital control logic for an on-chip Analog-to-Digital Converter (ADC) suitable for integration with the PSD8C integrated circuit described in an earlier section of this chapter. The central features of the converter design include 12-bit resolution with a sampling rate of approximately 2 MSamples/sec.

There are five chapters in this thesis. A brief description of the PSD8C IC was presented in this chapter along with a rationale for integrating the ADC onto the PSD8C chip in the future. In Chapter 2 we will describe the design of the 12-bit two-step flash ADC designed by Dinesh Dasari [Das:08, Raz:92] (based on a design by B. Razavi) and subsequent changes made in order to improve performance. Chapter 3 discusses the design of the digital control logic for the ADC. We describe in detail the standard cell design flow that was used to produce layout for the control logic. In Chapter 4 we present simulation results. The ADC was simulated at the behavioral level using MathCAD® and also at the electrical level using Cadence's Spectre® program. The MathCAD simulations were used to compare the results from the electrical simulations as part of the debugging process. Chapter 5 will give conclusions and discuss the future direction of this research work.

CHAPTER 2

ADC ARCHITECTURE

Two-Step Flash Converter

Two-step (sometime referred to as "two-stage" or "half-flash") flash architectures are an effective means of realizing high-speed, high-resolution analog-to-digital converters because they can be implemented without the need for operational amplifiers having either high gain, large gain-bandwidth product, or a large output swing. Moreover, with conversion rates approaching half those achievable by fully parallel designs, half-flash architectures provide both a relatively small input capacitance and lower power dissipation than their full-flash relatives [Raz:92].

As discussed in Chapter 1, an ADC suitable for integration on the various ICs under development should be capable of 12-bit resolution and a sampling rate of a few MSamples/sec. The design described in this thesis is the improved version of two-step flash ADC design by Dinesh Dasari [Das:08] which was based on an earlier design by Wooley and Razavi [Raz:92]. The Switched Capacitor (SC) Digital-to-analog converter (DAC) of the ADC is replaced with a resistive DAC to improve the ADC's performance.

The two-step flash ADC has a 7-bit first stage that produces a "coarse" estimate of the input analog voltage. The 6-bit second stage then uses this "coarse" estimate to produce a "fine" estimate of the input analog voltage. Digital correction logic is then used to combine the two estimates in order to produce the desired 12-bit result. One bit of redundancy or overlap is used in the proposed architecture to enable the second stage to correct for out-of-range errors in the first-stage

(resulting from mismatch errors), thereby relaxing the precision required of the first-stage comparators [Das:08].

The ADC is designed with an input range of -1.2 Volts to 1.2 Volts relative to the analog signal ground, which we call AGND. The AGND voltage in our design is 2.4 Volts. Therefore, the effective input range of the ADC is from 1.2 Volts to 3.6 Volts. This range is exactly what is needed for both the amplitude and time pulse trains currently produced by the ICs discussed in Chapter 1 of this thesis. The ADC is designed such that it produces a 12-bit zero output for an effective input analog voltage of 1.2 Volts (approximate bandgap voltage) and a 12-bit all ones output for an effective input analog voltage of 3.6 Volts (3 times the bandgap voltage). The digital output word could easily be converted to a 2's-complement representation if it proves more convenient.

Two-Step Algorithm

The two-step algorithm that is used in the ADC architecture is as shown in Figure 2.1 [Das:08]. We will describe the algorithm [Raz:92] in detail in this section of the thesis. As illustrated in Figure 2.1, the input analog voltage, V_{in}, is applied to the first stage of the ADC. The first stage produces a "coarse" digital output for the applied input analog voltage. The 7-bit coarse digital output is then passed to a 7-bit resistive DAC which produces an analog estimate of the "coarse" value.

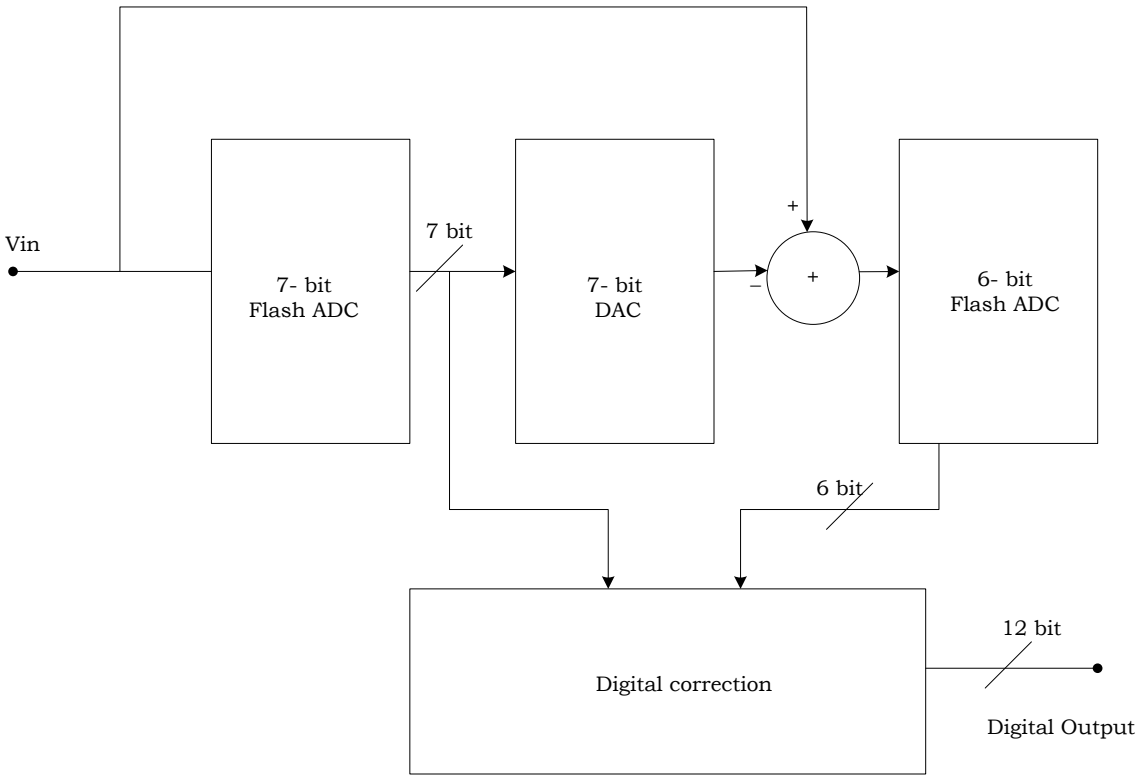

Figure 2.1: ADC converter architecture

This analog estimate of the "coarse" code is then subtracted from the input analog voltage by the subtractor (which we refer to as the "residue generator" in the remainder of this thesis) in order to produce the "residue". The "residue" voltage is then passed, as an analog input, to the second stage of the converter. The second stage is a 6-bit flash ADC that produces a "fine" digital output for the applied "residue" voltage.

The 7-bit coarse digital output is passed through a delay register (not shown in the figure) and then to the digital correction logic circuit where it is combined with the 6-bit fine digital output of the second stage to produce the 12-bit final digital output. Details on how the two digital values are combined to form a single 12-bit digital word will be presented in a later section of this thesis.

First Stage

A block diagram of the modified first-stage electronics is shown in Figure 2.2. The first stage of the ADC is composed of a resistive ladder, an array of 129 comparators, a resistive DAC and its controls, a residue generator and a thermometer- to-binary converter. The main difference between the design proposed in this thesis and the design described in [Das:08] is that here we have 129 comparators rather than 128 comparators. Also, and more importantly, a resistive voltage scaling DAC is used in place of linear SC DAC. A circuit for generating control signals for the resistive DAC controls is also added.

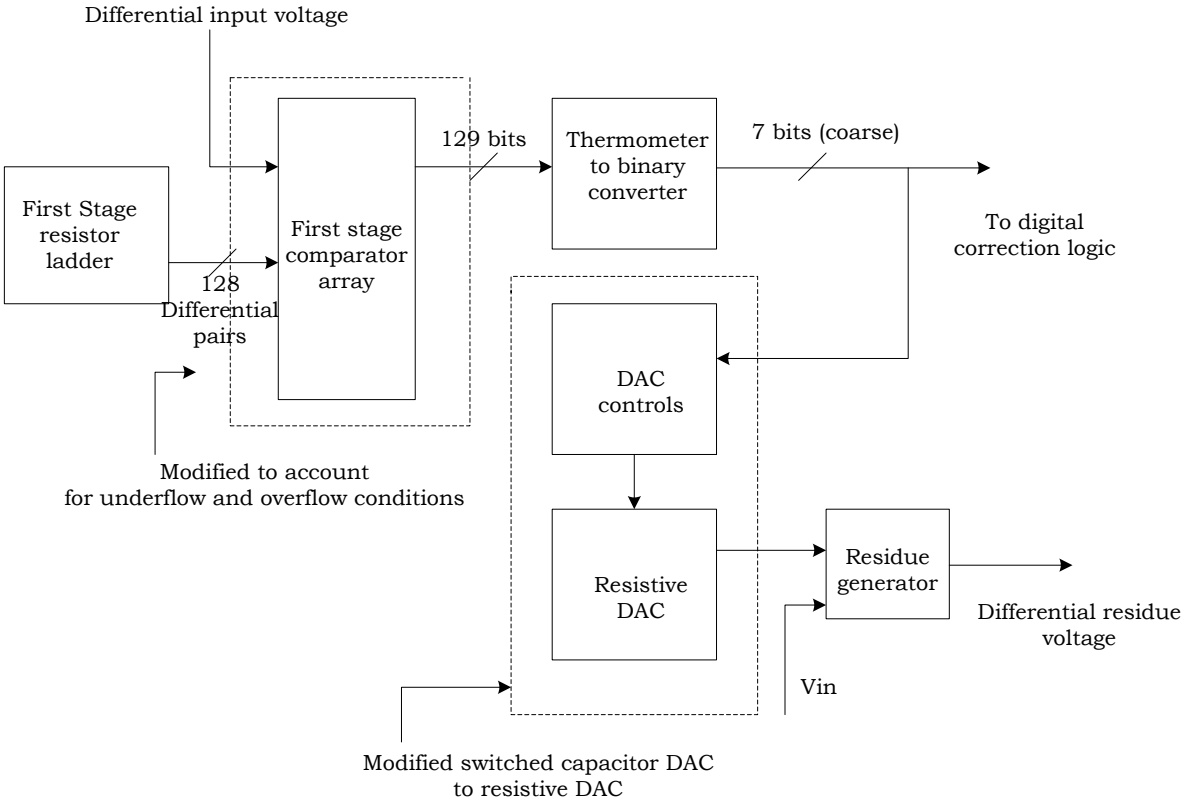

Figure 2.2: Modified block diagram of first-stage electronics

The reason for having 129 comparators is that out of 129 comparators, 127 are used for digitizing the analog input and remaining two comparators are used for checking underflow and overflow conditions. A first-stage comparator produces a 'one' if the applied tap voltage is greater than the analog input voltage $V_{in,}$ else it produces a 'zero'. Thus the array of 127 first-stage comparators produces a 127-bit "thermometer" code. The conversion of thermometer code to binary code is discussed later in the thesis.

Comparison of Capacitive and Resistive DACs

Both resistive and capacitive DACs can be constructed using an unit cell. This greatly simplifies the layout of the converter. The advantage of using a resistive DAC is that it does not suffer from the charge injection problem as capacitive DACs do. Charge injection is the name given to the effect that when a MOSFET switch turns OFF, a packet of charge is injected at both the source and drain terminals of the FET [All:03]. When a FET is turned ON, an inversion charge layer is present in the channel. When the device is switched OFF, the charge is injected from the drain and source terminals. If this charge is injected onto a capacitor, then the voltage across the capacitor is altered.

Because of the charge injection problem, the capacitive DAC introduces a systematic error which is not good. A resistive DAC has very small systematic error associated with it compared to a capacitive DAC. Simulation results showing the improvement in performance when a resistive DAC is used is presented in Chapter 4 of this thesis. The other added advantage of using a resistive DAC is that monotonicity is assured. Moreover, the resistive DAC uses less area than the capacitive DAC but, unfortunately, significantly more power.

Resistive DAC

The design of 7-bit differential resistive DAC is as shown in Figure 2.3. It consists of resistive ladder circuit, a DAC_plus switch array, a DAC_minus switch array (only a single switch array is illustrated in Figure 2.3,) and a DAC control circuit. The tap outputs from the resistive ladder feed switch arrays where the output of the DAC is obtained. The control signals for the switches are obtained from the DAC Control circuit. The output of this circuit is based on the number of comparators which fired in the first stage. The description of the DAC Controls circuit is discussed in detail in the next chapter.

For an N-bit resistive DAC, there are 2^N resistors connected between the positive and negative reference voltages. Thus, a total of 128 resistors are required to implement a 7-bit DAC. To account for offsets of the first stage comparators, tap voltages which are reduced by half of the quantization step are routed for the DAC switch array as shown in Figure 2.3. The analog approximation produced by the DAC is subtracted from the analog input voltage, V_{in} to obtain the “residue” using the residue generator.

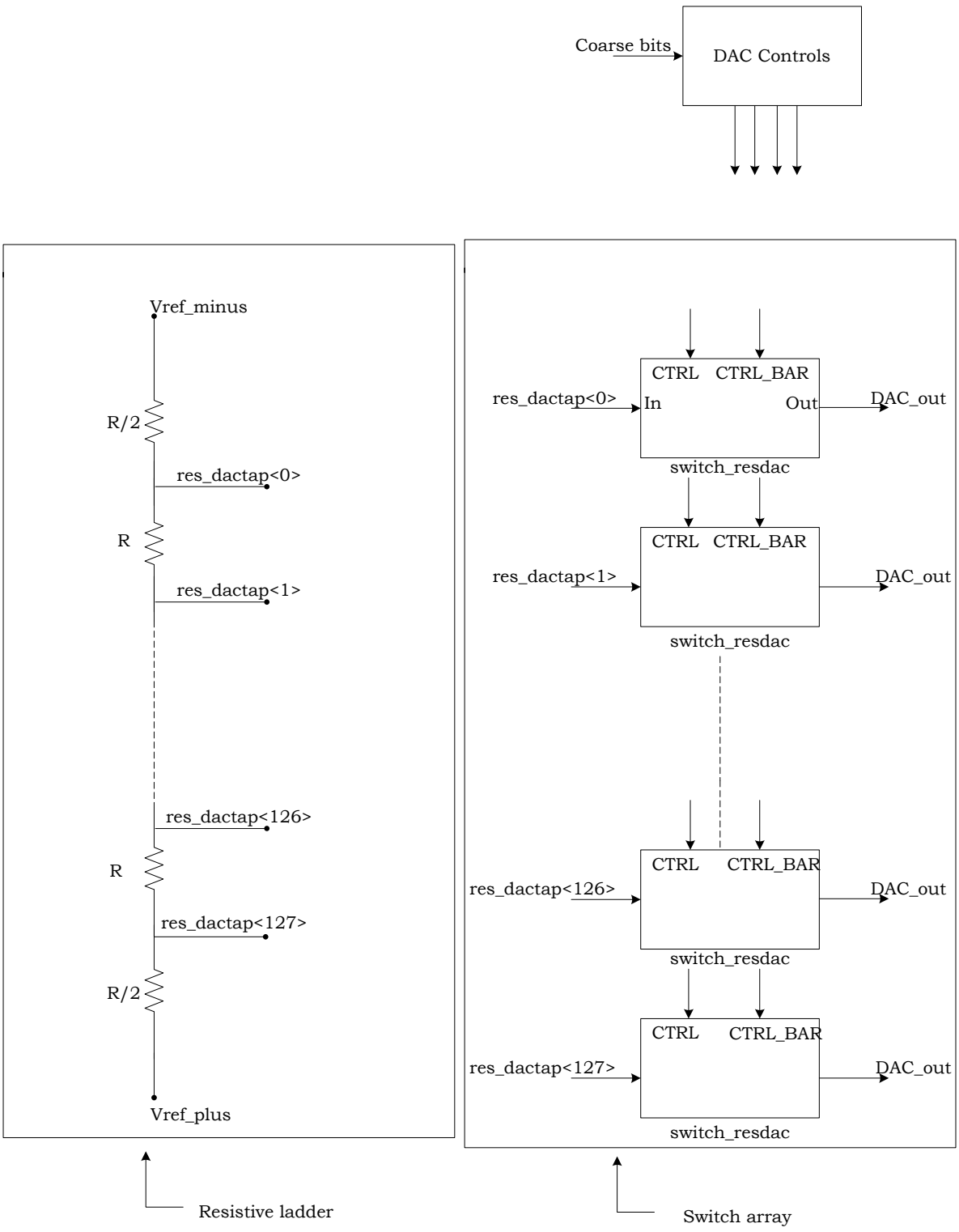

Figure 2.3: Resistive DAC architecture

Second Stage

A block diagram illustrating the second-stage electronics is shown in Figure 2.4 [Das:08]. The second (or "fine") stage digitizes the differential output of the residue generator to encode the six least significant bits (fine) of the ADC. The second-stage consists of a resistive ladder with a ladder loading correction circuit, an array of 64 comparators, and a thermometer-to-binary converter.

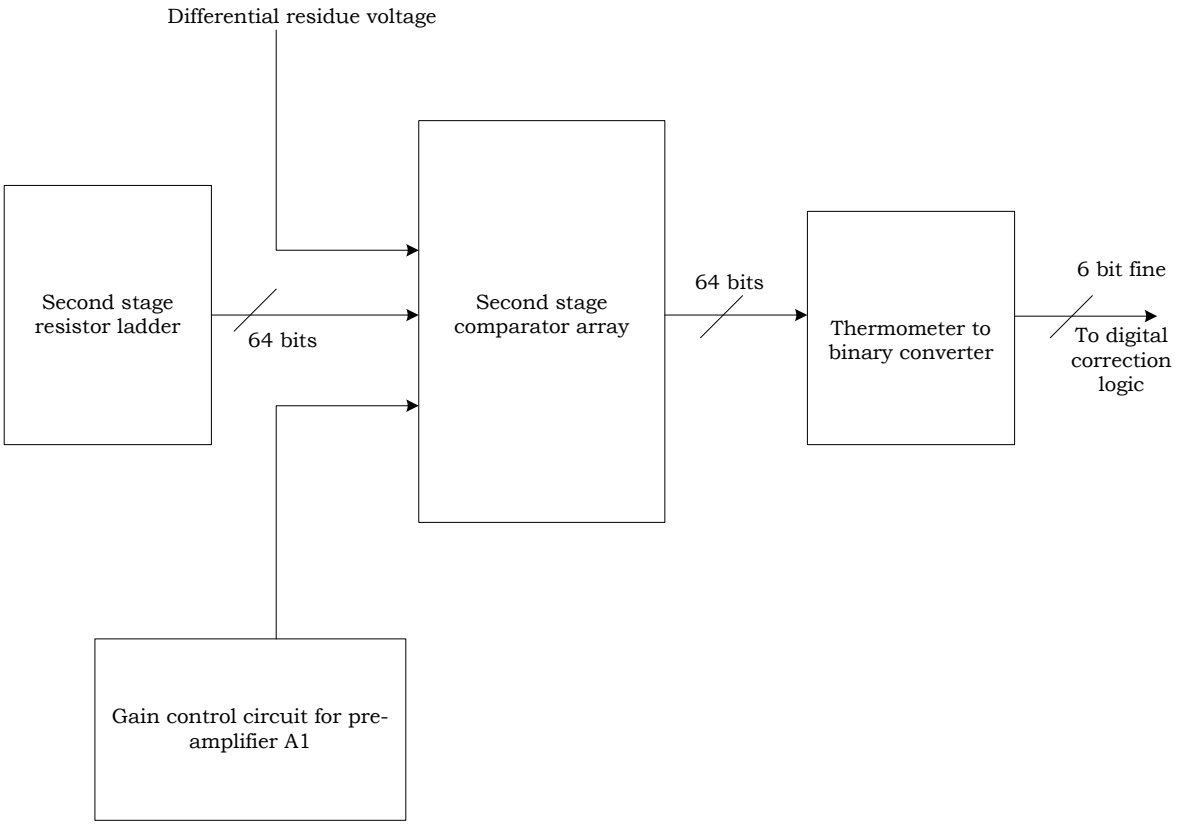

Figure 2.4: Block diagram of second-stage electronics

The second-stage comparators produces a 'one' if the applied tap voltage is greater than the residue voltage else it produces a 'zero'. Thus, the array of 64 second-stage comparators produces a 64-bit thermometer code. This code is then applied to a thermometer-to-binary converter where it is encoded into the 6-bit

(fine) value which is then passed on to the digital correction logic. No changes have been made to the second stage [Das:08].

Digital Encoding

Digital encoding performs the function of converting the thermometer code outputs of the comparators into a binary code. The design of the digital encoding and correction logic (at the circuit-level) is explained later in this thesis. The digital encoding is used to obtain a 7-bit "coarse" output from the first stage and a 6-bit "fine" output from the second stage. The digital encoding is modified to correct out of order 'ones' and 'zeros' caused by large offsets in the comparators by sensing three adjacent levels in the thermometer code [Raz:92]. The encoding logic used in this ADC consists of a series of AND gates followed by an encoder. These binary outputs are then corrected digitally to obtain the 12-bit ADC output.

Redundancy and Digital Correction Algorithm

The first-stage comparators are designed for high-speed and modest resolution, while potential moderate errors are accommodated by using one bit of overlap between the two stages [Raz:92]. The overlap makes sure that the residue (difference between V_{in} and the DAC output voltage) of the first stage falls within the input range of the second stage. Also the one bit overlap between two stages and digital correction logic relaxes the precision required of the first stage comparators and hence allows high speed in the coarse conversion.

The digital outputs of the two stages of the ADC are added using the digital correction logic to obtain the final 12-bit output of the ADC. The digital correction algorithm is depicted in Figure 2.5. The algorithm first checks the 7-bit digital output from the first stage. If it is zero, the entire output of the first stage is

shifted to the left by five places, and the 6-bit second stage output of the ADC is added to the first-stage output to get the 12-bit final output [Das:08].

If the output of the first stage is not zero, we shift it to the left by five places and shift it down by 16 LSBs. This is done to emulate shifting the DAC output down by half of the quantization step i.e., by 16 LSBs. The 6-bit second stage output of the ADC is then added to the first stage output to get the 12-bit final output. The only way it differs from the logic described in [Das:08] is that underflow and overflow conditions are included in the digital correction logic to account for out of range errors. The circuit level realization of digital encoding and digital correction is explained later in this thesis.

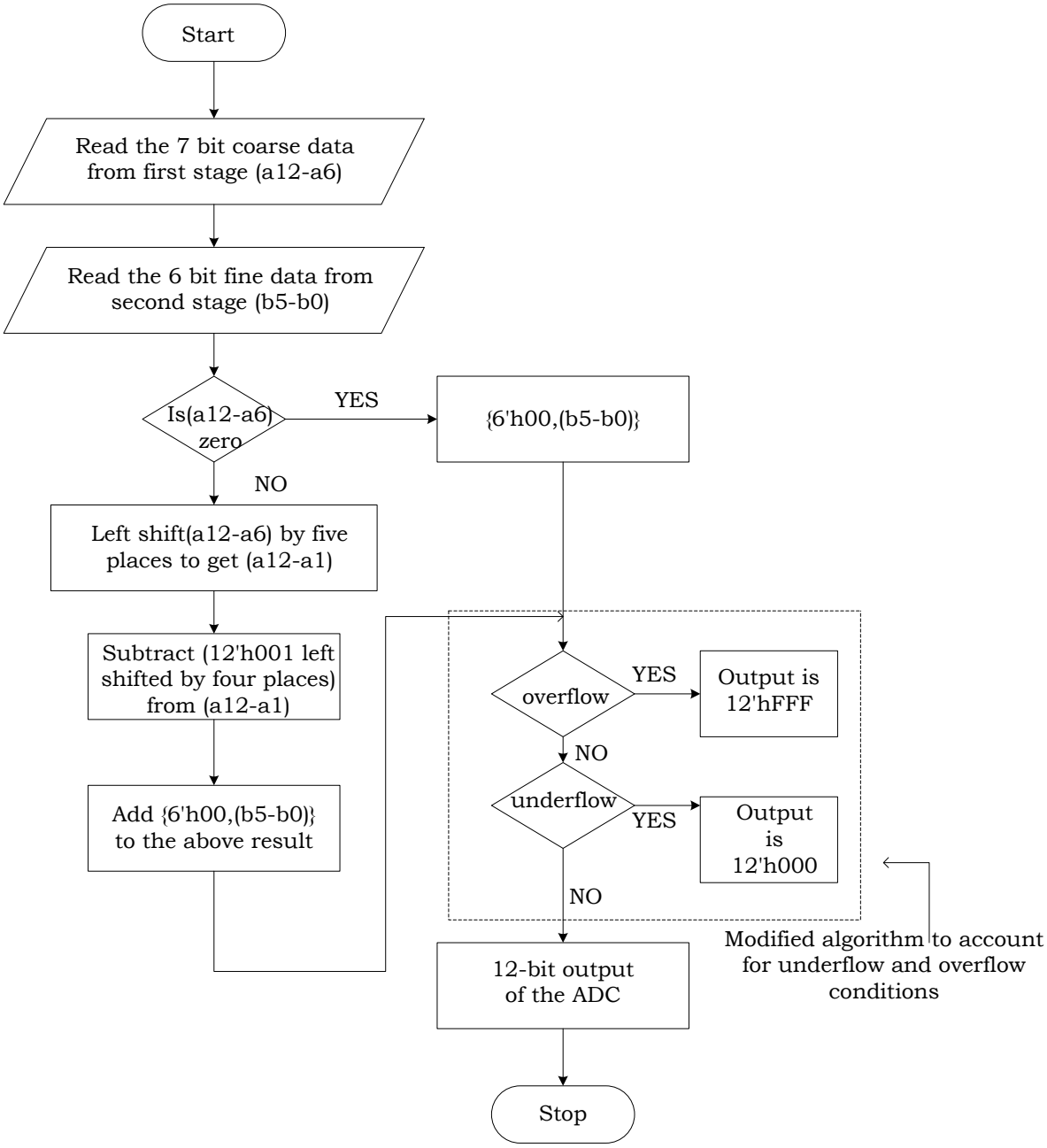

Figure 2.5: Digital correction algorithm flowchart

System Timing

The system timing of the ADC is as shown in Figure 2.6.

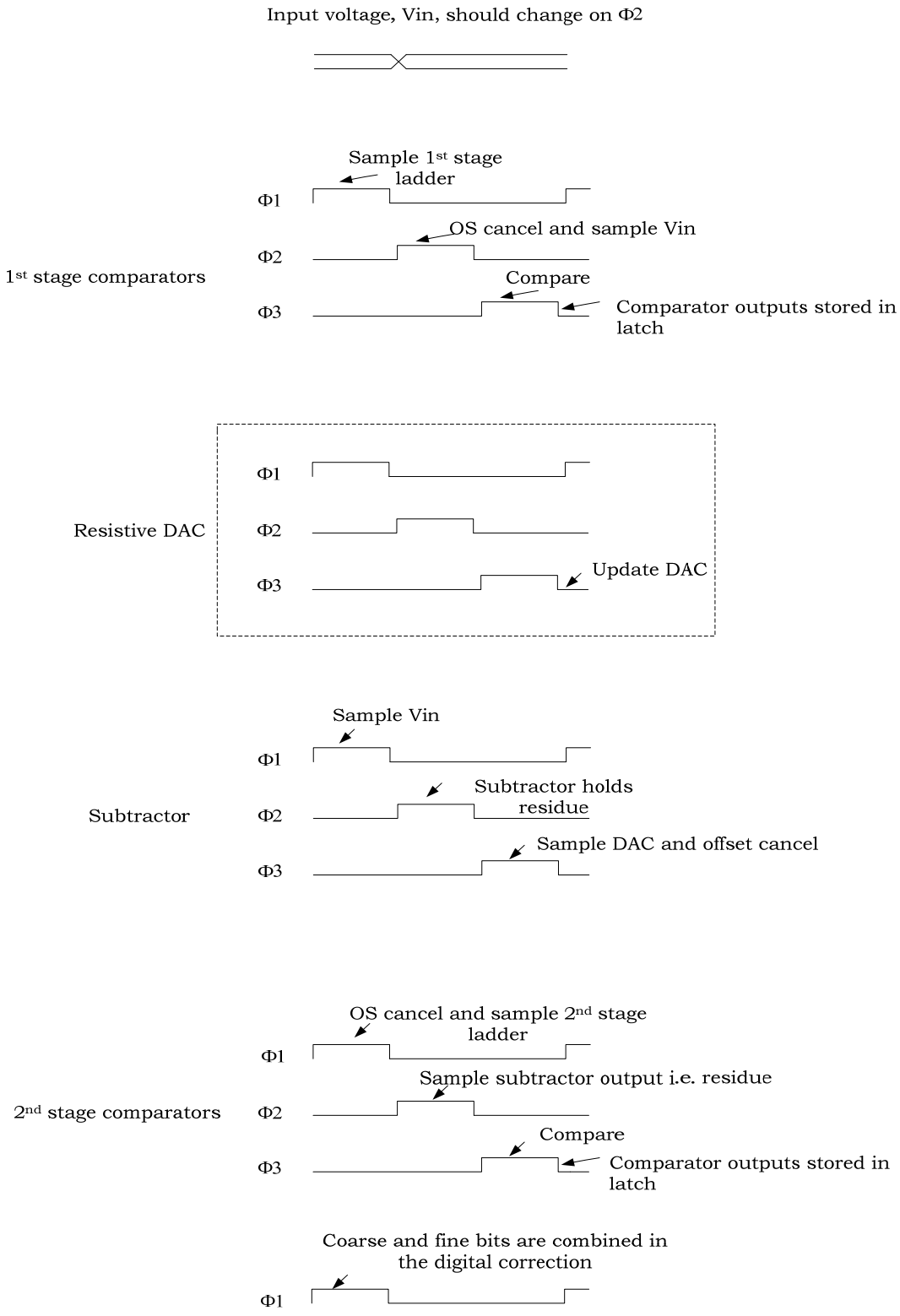

Figure 2.6: System timing diagram

The operation of the individual components in the ADC is controlled by three non-overlapping mutually exclusive clocks $Ø_1$, $Ø_2$, $Ø_3$. The entire operation of the ADC is pipelined and output of ADC is obtained on every raising edge of $Ø_1$.

The operation of first stage and second stage components is totally same as described in [Das:08] except for the resistive DAC which is updated on the falling edge of $Ø_3$. There is no resetting of the DAC, since it is resistive. The register logic block used in digital encoding and digital correction is clocked on $Ø_1$. A VerilogA model of the three phase clock generator presented in Appendix C was used in all electrical simulations of the ADC presented in Chapter 4, and its design is not discussed in this thesis.

CHAPTER 3

DESIGN OF DIGITAL CONTROL LOGIC

The design of the digital control logic for the proposed two-step flash ADC consists of creating hardware descriptions for the encoder logic, register logic, digital correction logic, and DAC control logic modules. A block diagram for the design is shown in Figure 3.1. Designing digital blocks at the circuit level involves writing Verilog code for the block and testing it, synthesizing the code using Cadence's RTL Compiler® and generating the layout for the module using Cadence's Silicon Encounter® software. In this chapter we will guide the reader through the entire process of designing the digital control logic.

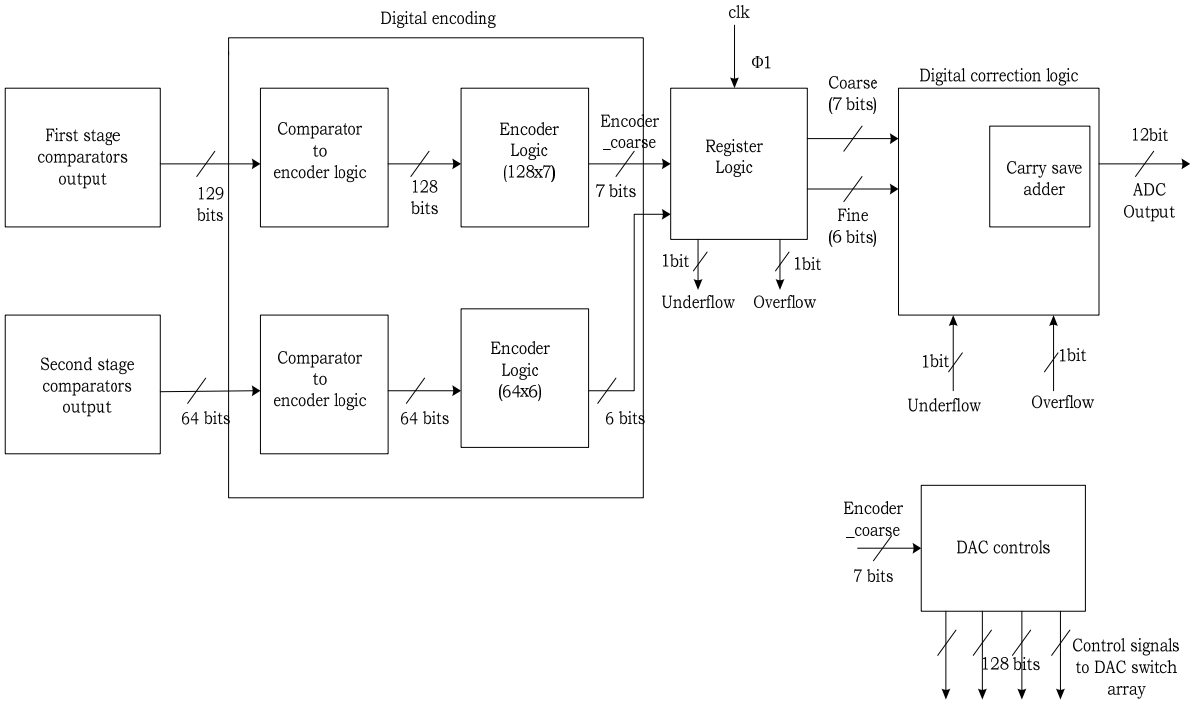

Figure 3.1: Block diagram of digital control logic

Logic to Correct Out-of-Order Ones and Zeros

The digital encoding block performs the function of converting the thermometer code output of the comparators into a binary code in two steps. It consists of logic to correct out of order 'ones' and 'zeros' followed by an encoder block. This section is devoted to describing the logic used to correct out of order ones and zeros. The out of order ones and zeros can be caused by offsets larger than expected in the first-stage comparators. This problem can be corrected by sensing three adjacent levels in the thermometer code emitted by the first-stage comparator array.

In order to correct out of order ones and zeros the outputs of the 129 first-stage comparators (0^{th} comparator output to 128^{th} comparator output) are used. The 0^{th} comparator output is used for checking for underflow condition and the 128^{th} comparator output is used for checking for overflow. An underflow occurs if the applied (single-ended) ADC input voltage is less than 1.2 V while an overflow occurs when the applied voltage exceeds 3.6 V. In the future if we refer to comparator outputs we will be referring to the outputs from comparators 1-127. since the first and last comparators are only used for out-of-range detections.

Whenever data is available for this block, all the outputs of this block are cleared at very beginning. Then the block will check for number of comparators fired. If no comparator is fired, then the 0^{th} bit of the output is set to '1'. If at least one of the comparators has fired, the output of the comparators are loaded in to temporary buffer with '0' appended at the bottom and '1' appended at the top. Then the buffer is fed to series of 127 'and' gates, where the third input to every 'and' gate is complemented as shown in Figure 3.2

The logic used for the “fine” step (first-stage comparators) is similar to the block defined for the “coarse” step (second-stage comparators) except that it does not have underflow and overflow bits.

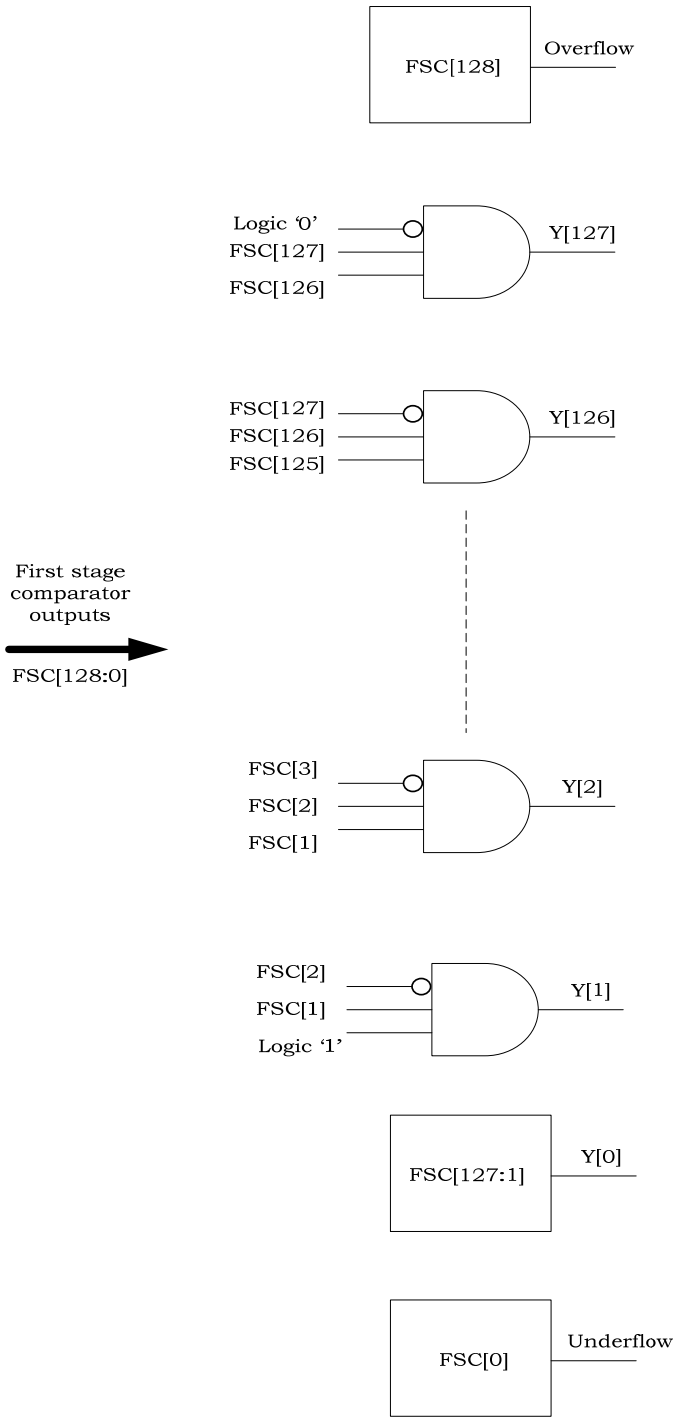

Figure 3.2: Block diagram of logic for correcting out-of-order ones and zeros

Encoder Logic

The block diagram for the encoder is presented in Figure 3.3.

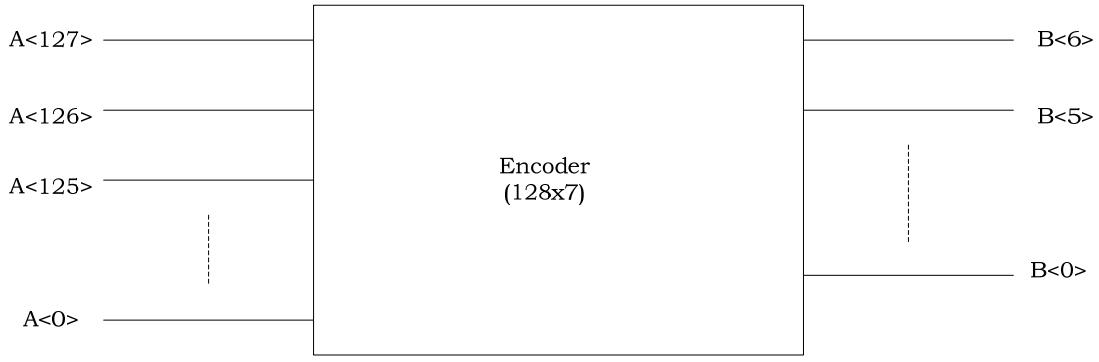

Figure 3.3: Block diagram of encoder

The logic encodes the output block used to correct for out-of-order ones and zeros into an equivalent binary code. It performs the function of priority encoding of the given input vector. For a given input vector, if the n^{th} bit is high then the equivalent binary output value is n. If more than one input bit is high then the output value corresponds to highest input bit value because of the priority encoding scheme which was used.

Register Logic

The register logic block is shown in Figure 3.4. This block is used synchronize the "coarse" value, the "fine" value, the underflow bit, and the overflow bits to the clock. It consists of four n-bit registers. Among them one is delayed register block because the digital correction logic requires the "coarse" data to be delayed by one clock cycle due to the two steps required for a conversion.

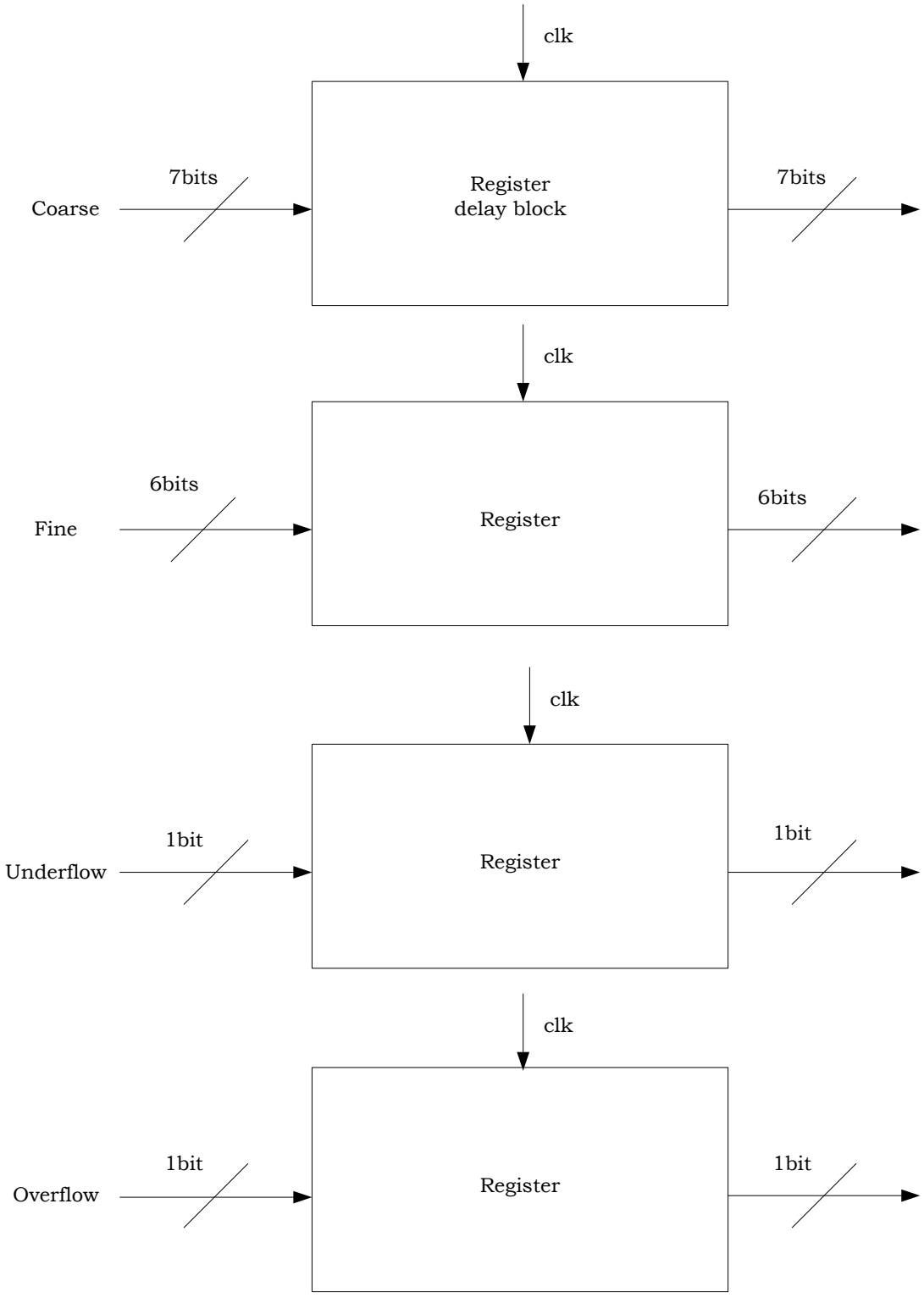

Figure 3.4: Block diagram of register logic

All of the registers in Figure 3.4 are clocked on the rising edge of the Φ1 clock. We wrote a single Verilog® description but with the width of the register as a parameter that could be set upon instantiation.

Carry Save Adder Block

A Carry Save Adder (CSA) is a digital adder that adds three or more n-bit binary numbers. The main reason for using a CSA is that we refrain from directly passing carry information until the very last step unlike what happens with other digital adders. The difference between a full adder and CSA is shown in Figure 3.5 [Loh:05]. It differs from other digital adders in that it outputs two vectors of the same dimensions, one which is a sequence of partial sum bits and other which is a sequence of carry bits. These two vectors must then be added using a convention adder.

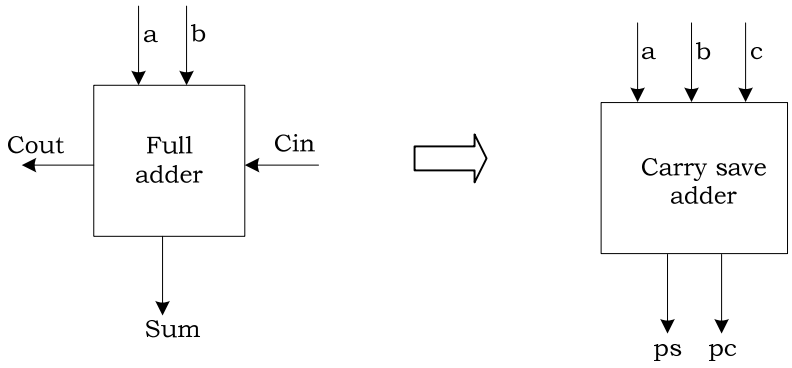

Figure 3.5: Comparison between full adder and carry save adder

A CSA is a simple full adder with some of the inputs and outputs renamed. From the Figure 3.5, we can see that c_{in} is renamed to c and c_{out} is renamed to pc. The carry-save unit consists of n independent full adders, each of which computes a single sum and carry bit based solely on the corresponding bits of the three input

numbers. Given the three n-bit numbers a, b and c, it produces a partial sum ps and a shift-carry sc.

$$ps_i = a_i \oplus b_i \oplus c_i$$

$$sc_i = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)$$

The entire sum can be calculated by

- Shifting the carry sequence left by one place.
- Appending a '0' before the Most Significant Bit (MSB) of the partial sum.
- Using a ripple carry adder to add the above sequences producing an n+1 bit value [Wik].

Digital Correction Logic Block

The digital correction logic block contains the necessary logic for combining coarse and fine information [Das:08]. It is necessary because offset voltages in the first stage comparators can cause the residue to be shifted upward or downward. By allowing an additional bit of resolution in the fine stage, we can compensate for the first stage offset error. This compensation can be done by the following equation:

$$Y = A*(2^{N-1}) - 2^{N-2} + B \qquad \text{Eqn. 3-1}$$

where A = $\{a_{M-1}, a_{M-2}.,. ., a_N, a_{N-1}\}$ = Result from "coarse" stage

B = $\{b_{N-1}, b_{N-2}, \ldots, b_1, b_0\}$ = Result from "fine" stage

Y = Output value of ADC

In order to efficiently implement the above logic we require a carry save adder and a ripple carry adder to add the partial sum and carry sequence generated by the CSA block. So the digital correction logic block consists of a CSA

block defined earlier and ripple carry adder logic to implement the equation defined above. The block diagram of digital correction implementation for the 4-bit case is shown in Figure 3.6 [Das:07].

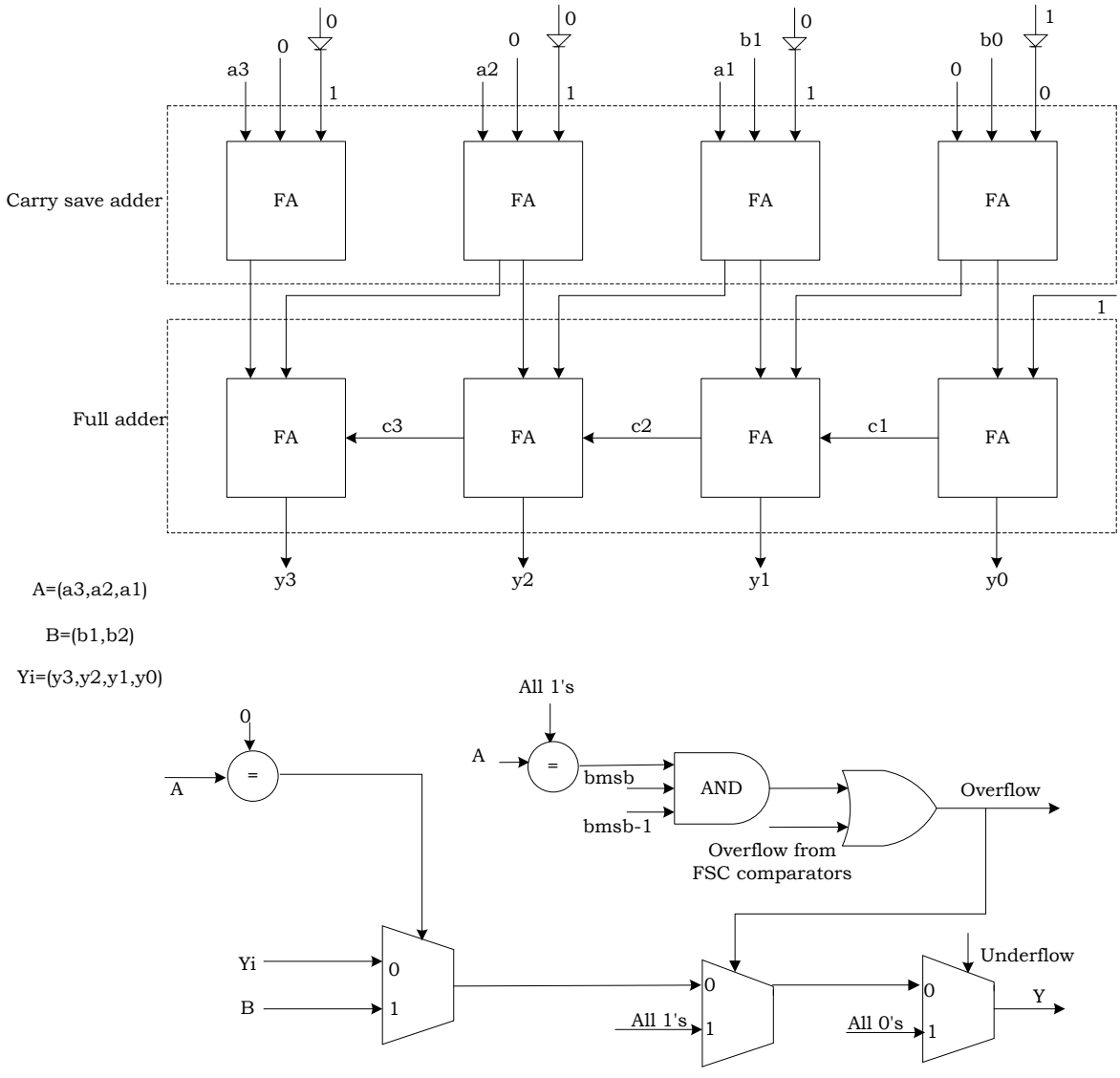

Figure 3.6: Digital correction logic for 4-bit sample design

The digital correction block also requires the underflow and overflow bits as inputs. Depending on the status of these bits, the output of the ADC is determined. If the under flow bit is high, the output of 12-bit ADC is set to '0'. If the over flow

bit is high, the output of ADC is set to all 1's (i.e., decimal equivalent of 2^{12}-1 = 4095).

Controls for DAC

The necessary control signals for the resistive DAC are generated using this block. Depending on the number of first-stage comparator outputs which are high, the "coarse" value is generated. By giving the coarse value as input to this block, the control signals for switches in resistive DAC are generated. Since the resistive DAC contains a DAC plus switch array and a DAC minus switch array, four control signals are generated.

Consider the case where ten comparators have fired, then the coarse value is '10'. Then corresponding to this coarse value, the control signal for the 9th switch in the switch array should be high for the DAC plus block and for DAC minus block (129-coarse = 119) the control signal for the 118th switch should be high. The complemented signals for these controls are generated by negating them. Complementary signals are needed because the switches were implemented using transmission gates (parallel NFET and PFET driven by signals which are logical complements of one another).

The complete Verilog® description of all the blocks defined above is provided for the interested reader in Appendix A of this thesis.

Layout Generation of Digital Blocks

So far we have discussed the design of various digital blocks *i.e.*, the logical descriptions. To complete the design of the digital control logic for the two-step ADC, we need to test our Verilog code, synthesize it and generate the layout from with the help of a standard cell library. The design flow for testing the Verilog descriptions to layout generation is illustrated in Figure 3.7. The Cadence tool used to accomplish the task identified inside of the block is written above the corresponding block. The specific Cadence tool suite which was used in this design is known as SoC (System on Chip) Encounter®.

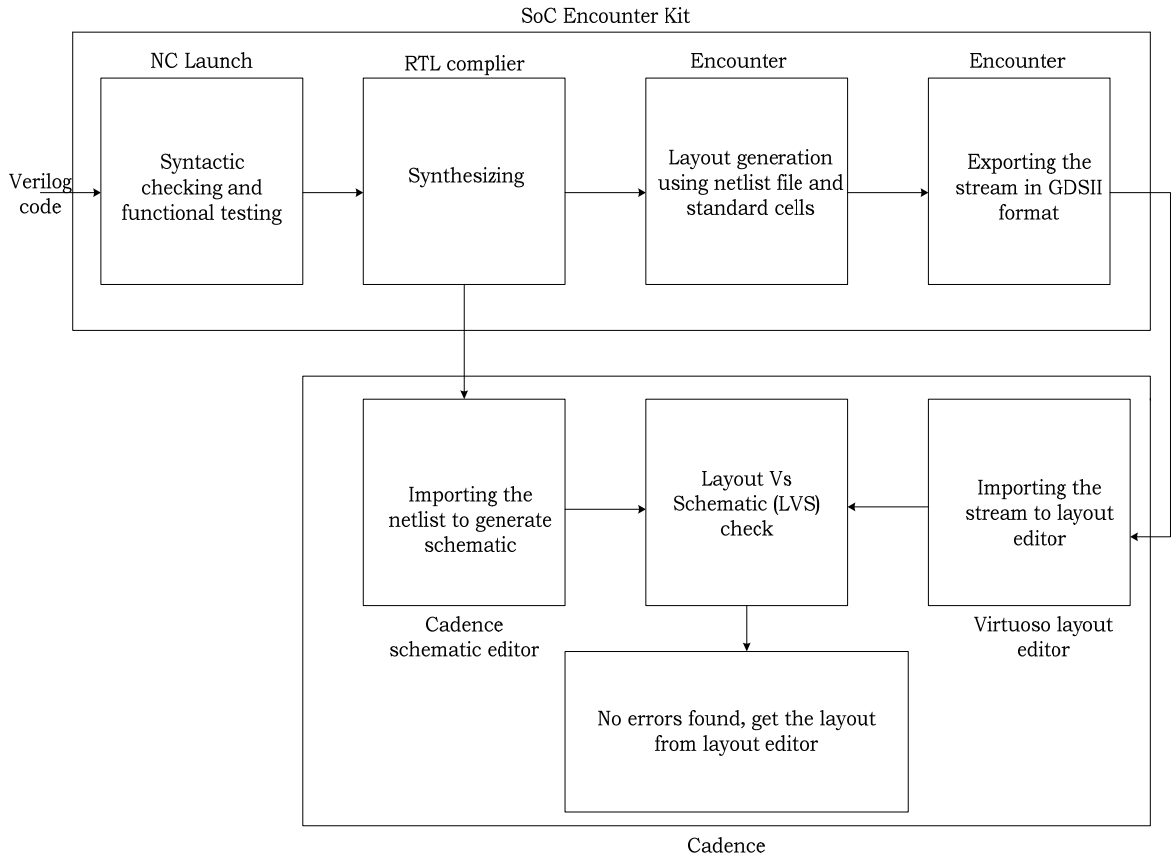

Figure 3.7: Design flow from Verilog code testing to layout generation

Testing the Verilog Code

NC Launch® is the graphical interface used to simulate and debug our Verilog code (it also supports VHDL and mixed-language designs). There are three steps involved in the functional verification of the Verilog design using NC Launch. They are (1) Syntactic Checking, (2) Elaboration and (3) Simulation.

Syntactic Checking: This step is done using NC-Verilog® which performs syntactic checking on the Verilog design HDL (Hardware Description Language) units and if no errors are found it produces an internal representation for each HDL design unit in the source files.

Elaboration: The design is elaborated using NC-Elaborator® which constructs a design hierarchy based on the instantiation and configuration information in the design. It also establishes signal connectivity and computes initial values for all objects in the design.

Simulation: This step is performed using Incisive Unified Simulator® which runs our design with a set of test inputs (known as a testbench) and helps us determine whether the design behaves as we intended or not [Ncl:08]. The complete test bench is given in Appendix B. The simulated waveforms can be viewed using a tool called Simvision®.

In addition to above steps, we can annotate the timing check and delay information in SDF (Standard Delay Format) file to Verilog using the SDF Compiler®. SDF annotation is performed during elaboration. The simulator needs only SDF compiled files. The elaborator looks for a corresponding compiled file (sdf_filename.X) [Ncv:08].

Synthesis

Once the Verilog code is checked for syntax and tested for its functionality, we can go ahead with the next step which is synthesis. The compiled Verilog code is given as input to RTL Compiler® for synthesis. Interaction with RTL Compiler® occurs within RTL Compiler® shell.

RTL Compiler® uses the standard scripting language known as Tool Control Language (Tcl) to efficiently automate the tasks. For performing the synthesis with this compiler we write a Tcl script. The complete Tcl script is shown in Appendix E. This script automates the process of loading the standard cell technology library, loading the Verilog design, elaborating it, setting the constraints, synthesizing it, mapping the design and finally writes out the netlist file and Standard Delay Format (SDF) file [Rtl:08]. The netlist file contains the standard cells description and their interconnect based on the Verilog design. The netlist file generated needs to be fed to Encounter® to generate the layout.

Standard Cell Library

Standard cell library contains a collection of components that are standardized at the logic or functional level. It consists of cells or macro cells based on the unique layout that designers use to implement the function of their ASIC. The economic and efficient accomplishment of an ASIC design depends heavily up on the choice of library [Asr].

A standard cell library based on MOSIS (MOS Implementation Services) submicron scaleable CMOS (SCMOS_SUBM) rules has been developed by Johannes Grad and James Stine for student projects. The library is targeted for the AMI 0.5µm process [Gra:03]. This standard cell library was developed at Okhalama State University (OSU) jointly with North Carolina State University (NCSU). The

standard cell library makes heavy use of the NCSU Design Kit. From now we will refer to this standard cell library as the OSU standard cell library.

Unfortunately, the SIUE IC Design Research Laboratory does not use the NCSU Design Kit. Rather the Lab uses a design kit obtained directly from the AMI foundry. So in its original form, we are unable to use the OSU standard cell library. Before it could be used, it was necessary to transform the OSU AMI standard cell library into one which is compatible with the AMI design kit rather than with the NCSU design kit. In the remainder of this thesis, we will refer to this transformed library as the "AMI cell library". A Perl script (see Appendix D) was written to help in the transformation.

The basic difference between the OSU cell library and the AMI cell library is that OSU cell library is λ based where λ is 0.3, whereas AMI cell library is micron based. This is a result of the fact that OSU cell library is based on NCSU design kit while the AMI cell library must be based on the native AMI design kit. Also, the contact size in the OSU cell library is 0.6μm x 0.6μm, but for AMI cell library it is 0.5μm x 0.5μm. Finally, the layer names for metals and vias are different in the two libraries.

For the above reasons, instead of designing a totally new standard cell library compatible with the AMI design kit, we decided to make modifications to each cell in OSU library to make it compatible with the AMI design kit. Each cell in OSU library was exported to Caltech Intermediate Form (CIF) which is a text file. The modifications for the CIF file (in text format) such as contact size reduction and layer name change is done easily using a script written in Perl (Appendix D) which is explained later in this thesis.

The script was executed and all of the cells in the OSU library where transformed and the AMI library was created. The AMI standard cell library is used

in Cadence Virtuoso to get the layout of the digital block based on the AMI design kit.

The standard cell library used by the RTL Compiler® and Encounter® is the OSU cell library. To generate the layout (i.e., interconnections based on the netlist file) in Encounter®, the OSU cell library is used. For the final layout generation in Cadence, the AMI cell library is used.

Layout Generation using Netlist File in Encounter®

To generate the automatic layout (i.e., interconnections based on the netlist file) from the synthesized Verilog code, the Encounter® tool is used. For layout generation we require a netlist file from RTL Compiler® and the OSU standard cell library and its technology files. The layout generation in Encounter involves design importing, floor planning, power planning, placing the design, routing, verifying the geometry and exporting the GDSII stream. We will explain each of these steps in additional detail.

Design Import: For design import, the synthesized netlist file from RTL Compiler® is given as input. The global power and ground rails are specified. The design rules for placement and routing, interconnect resistance and capacitance data for generating RC values and wire load models is provided in a technology file known as a Library Exchange Format (LEF) file.

The LEF file also contains information for the metal interconnect layers, including metal thickness, metal resistance, and line to line capacitance values of metal layers, for determining coupling capacitance. The timing information (in ASCII format) of all standard cells, blocks and I/O pad cells is given in timing library format (tlf) file [Enc:08].

Floor Planning: Floor planning the design is an important task of physical design in which the location, size and shape of the modules are decided. Depending on the design style or purpose, it also can include row creation, I/O pad or pin placement, bump assignment (flip chip), bus planning and power planning.

For the current design, the design area is core area which is surrounded by specified core margins. The height/width ratio and row spacing of the design is specified while floor planning.

Power Planning: For creating connection between preliminary power structures in the local cells to the global power nets, power planning is required. By creating core ring and stripes to the core area, the encounter can create power connections between pins of specified nets on the blocks and pads to nearby ring or stripes.

Once the floor planning is completed, for making final connections Special Route (SRoute) ® needs to be used. SRoute® creates pad rings, routes power and ground nets to block pins, standard cell pins and unconnected stripes. Also if we need metal layer change or allow jogging, it needs to be specified in SRoute®. After power planning, the design is ready for combined power and rail analysis to determine whether power structures and connections provide sufficient power to the design [Enc:08].

Placing the Design: After floor planning, we need to place the design (placing standard cells based on a Verilog netlist). Placement considers the modules that were placed during floor planning and takes in to account the hierarchy and connectivity of the design. It takes floor planning constraints in to account while placing.

Routing: Routing consists of detailed and global routing of the design. Global routing breaks the design into global routing cells (gcells) and finds the shortest path through the gcells without making actual connections. The router also generates a map of the gcells, called a congestion map, to examine the approximate number of nets assigned to the gcells.

The detailed router uses the global routing plan and lays down actual wires that connect the pins to their corresponding nets. It also creates shorts or spacing violations rather than leaving unconnected nets. Then it runs search-and-repair routing to locate shorts and spacing violations. After that, the design is rerouted on affected areas to eliminate as many shorts as possible without leaving shorts or spacing violations [Enc:08].

Verifying the Geometry: To check the physical layout of the design, including width, length, spacing, area, overlap, enclosure, wire extension, and via stacking violations, verify geometry is used. At the end it gives a report stating number of violations and warnings.

Exporting the GDSII Stream: GDSII stream format is the binary file format representing planar geometric shapes, text labels and other information about the layout in hierarchical form. It can be used to reconstruct all or part of the art work to be used in sharing layouts, transferring artwork between different tools.

Once the auto layout has been completed based on the netlist file and OSU standard cell library, we need to export the layout in GDSII stream format so that it can be imported to Cadence Virtuoso for getting the layout based on AMI standard cell library.

Layout Generation in Cadence based on AMI Cell Library

To get the layout for the digital logic based on AMI design kit, the GDSII stream from the Encounter needs to be imported. A schematic need to be generated based on the synthesized netlist file from the RTL Compiler.

Importing the GDSII Stream: The GDSII stream from Encounter needs to be imported as stream into Cadence virtuoso layout editor. A layer map file based on AMI design kit is specified for importing the stream. Also specified is the location of the AMI standard cell library (obtained from the OSU cell library after running the Perl script). Then a layout is obtained corresponding to the stream given compatible with the AMI design kit.

Generating the Schematic based on Netlist File: A schematic is generated in Cadence schematic editor based on the netlist file from RTL Compiler using import netlist file option. The AMI standard cell library location is specified for importing. The schematic generated consists of AMI standard cells and interconnections based on the netlist file.

Layout Vs Schematic (LVS):

To check the validity of the obtained layout, the schematic based on the netlist file and layout is checked for LVS errors. If no errors are obtained, it completes the total flow of obtaining the complete design of digital logic. In case errors are found, we need to go back, check and repeat each step to obtain zero errors.

Perl Script to Convert OSU Standard Cells to AMI Standard Cells

Each cell in OSU cell library is exported to CIF format which needs to be fixed to convert it in to AMI standard cell. A script is written in Perl to perform following functions on a CIF file (ASCII text file format). The script does the following:

- It creates a writable output file with same file name as input file but with the added extension of "_AMI".
- The input file data is read into an array and a subroutine is called for editing the data in the array.
- The subroutine changes the contact size from (0.6µm x 0.6µm) to (0.5µm x 0.5µm).
- The subroutine gets the "Pselect" layer specifications (specs) into an array, removes the "Nselect" layer specs and pastes the "Pselect" layer specs.
- The subroutine gets the "Nfield" layer specs in to an array, creates a "TUB" layer and pastes the "Nfield" specs.
- The edited data from the subroutine is written to the output file.

CHAPTER 4

SIMULATED PERFORMANCE OF ADC

Chapter 4 presents the results of simulations performed on the ADC. The comparison of differential resistive and capacitive DAC errors is presented. The ADC has been verified with behavioral-level and electrical-level simulations. The random offsets and mismatches that would be present in the ADC if it were fabricated are not accounted for in the electrical simulations performed using Cadence's Spectre® program. However, analyses were performed in MathCAD® to estimate the effects of random offsets and element mismatches. The results of these analyses are presented [Das:08]. The complete layout of digital control logic obtained from the flow described in earlier sections is also presented.

DAC Error Simulation

The performance of DAC greatly impacts the overall performance of the ADC. The 7-bit differential DAC has been tested with 128 digital input combinations and the outputs of the DAC obtained are compared with the ideal values to obtain the error. The error plots of resistive and capacitive DAC for 128 digital input combinations are shown in Figure 4.1.

From the Figure 4.1, we can observe that differential DAC error for ADC with capacitive DAC varies linearly with a maximum variation of +/- 1mV due to non-ideal effects such as charge injection. This corresponds to 2 LSBs of the 12-bit ADC (where 1LSB is $2.4/2^{12}$ = 0.5mV) which is equivalent to one bit quality of the ADC. Because of this systematic offset, the effective number of bits (ENOB) of the ADC with capacitive DAC is only 11 bits as described in [Das:08].

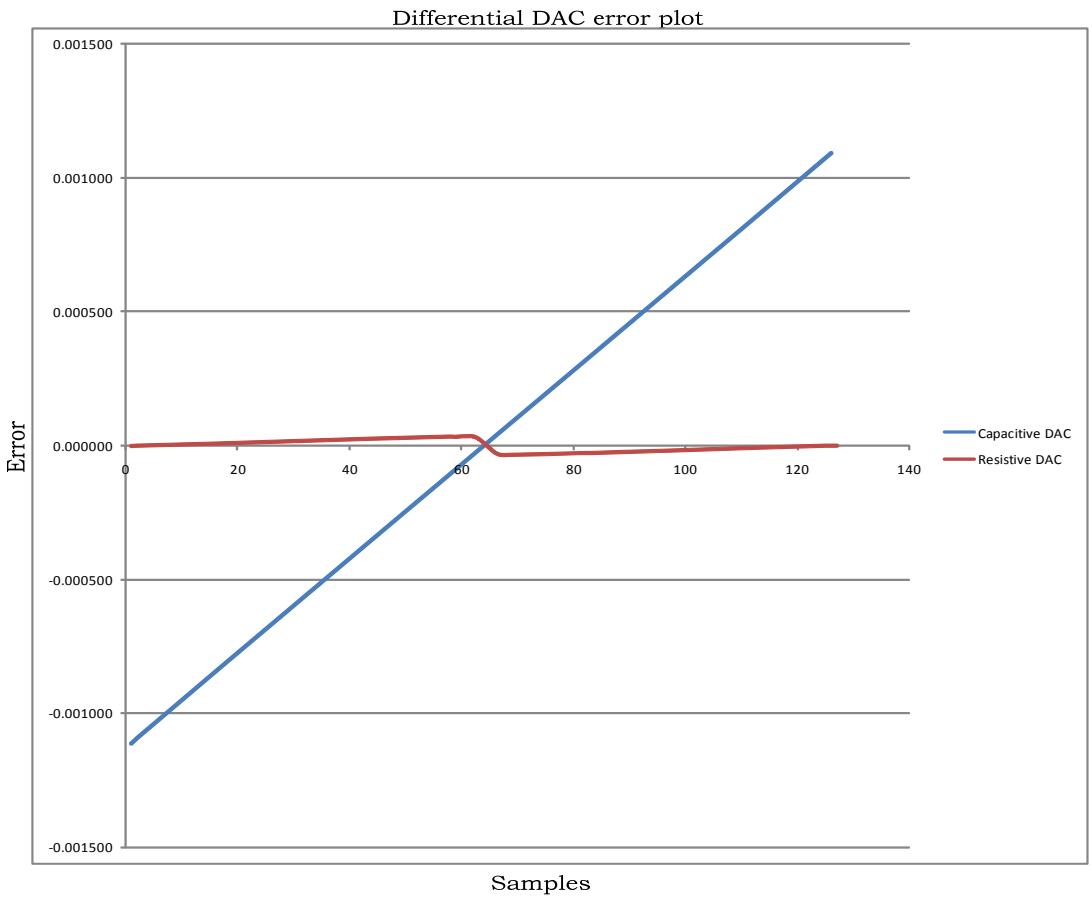

Figure 4.1: Differential capacitive and resistive DAC error plot

But for the ADC with resistive DAC (see Figure 4.2), the error is nearly constant with a maximum variation of +/- 35μV which is equivalent to 7% of 1 LSB. This error has negligible impact on the performance of the ADC. Even the parasitic capacitance associated with the resistor is modeled while obtaining the error of the DAC. The other added advantage of the resistive DAC is that there is no charge injection problem associated with it. The overall improved performance of the ADC is shown in next section.

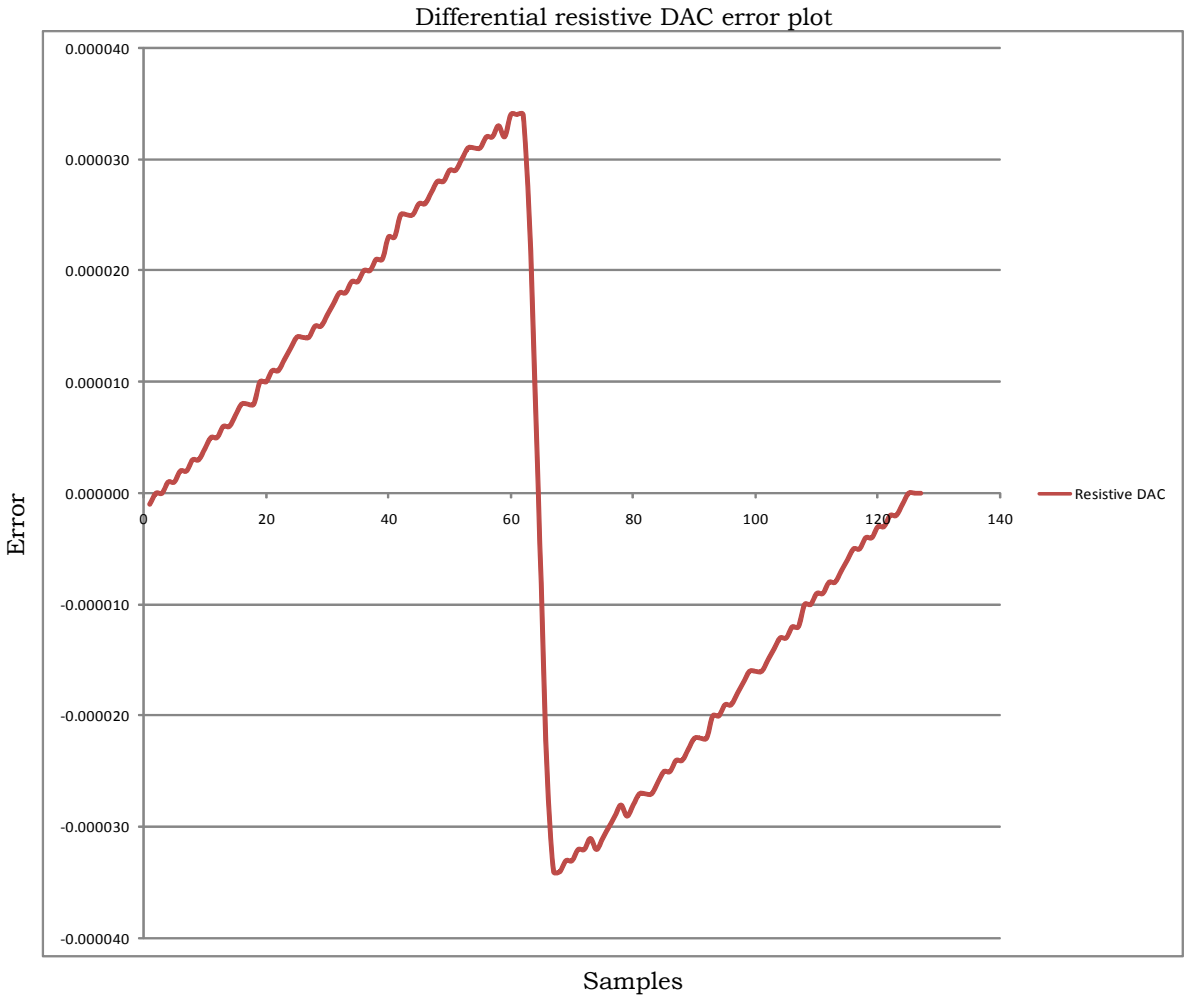

Figure 4.2: Differential resistive DAC error plot

Verification of Two-Step Algorithm

Electrical simulations are performed on the ADC with resistive DAC using Cadence's Spectre®. For performing these simulations, FET level schematics are used for all components except for clock generator and reference voltage generator circuits. The ADC is supplied with ramp input and uniformly distributed analog voltages ranging from -1.1 Volts to +1.1 Volts (referenced to AGND) and simulated to get the digital outputs for typical corner. The digital outputs obtained are supplied to MATHCAD® which is converted to an analog output vector using the

ideal DAC. The analog input voltage vector to the ADC was then subtracted from the analog output voltage vector to form the error vector. The mean and standard deviation of the error vector were computed.

The standard deviation of the error associated with a linear quantizer is δ/sqrt (12) where δ is the effective step size. In other words, δ = sqrt (12) * standard deviation. The ENOB can then be computed using the equation ENOB = log (FS/δ) / log (2) where FS is the full-scale range of the converter (in this design 2.4Volts) [All:03].

The ENOB for ramp input to the ADC was found to be 12 bits. For uniformly distributed 100 inputs case it took 10 days to complete the simulation and the ENOB was found to be '11.91' bits, which is a very good improvement in the performance of the ADC when compared with capacitive DAC case (whose ENOB was only 11 bits). The ADC needs to be simulated with 1000 input uniformly distributed case to get more accurate ENOB which takes lot more time to complete.

In addition to the FET level schematic of digital control logic, the extracted layout of the digital control logic is also used and electrical simulations are performed on the ADC which gave an ENOB of 11.9 bits.

The verification of the two-step algorithm is also done using a comprehensive simulator developed using MathCAD®. Every effort was made for the MathCAD code to emulate operation of the ADC at the *circuit level*. The mismatches in the resistor ladders, DAC capacitor mismatches, first-stage comparator offsets, residue generator gain error, and the second-stage comparator offsets have also been modeled in the MATHCAD® simulator [Das:08]. The complete description of MATHCAD® simulator is shown in Appendix F.

A vector of 100,000 analog inputs was generated which were uniformly distributed between -1.1 Volts and +1.1 Volts (referenced to AGND). The analog

inputs were digitized using the two-step flash algorithm (emulating circuit-level operation) coded in MathCAD. The resulting digital output vector was then converted back to vector of analog output voltages using an ideal DAC and the ENOB of the ADC was found to be 12 bits.

The analog input vector was sorted and then the quantization error was plotted as a function of input level. The error plot is presented in Figure 4.3. Note the error is not zero mean. This implies that the ADC had a DC gain error of about 10 mV which is not important in our application.

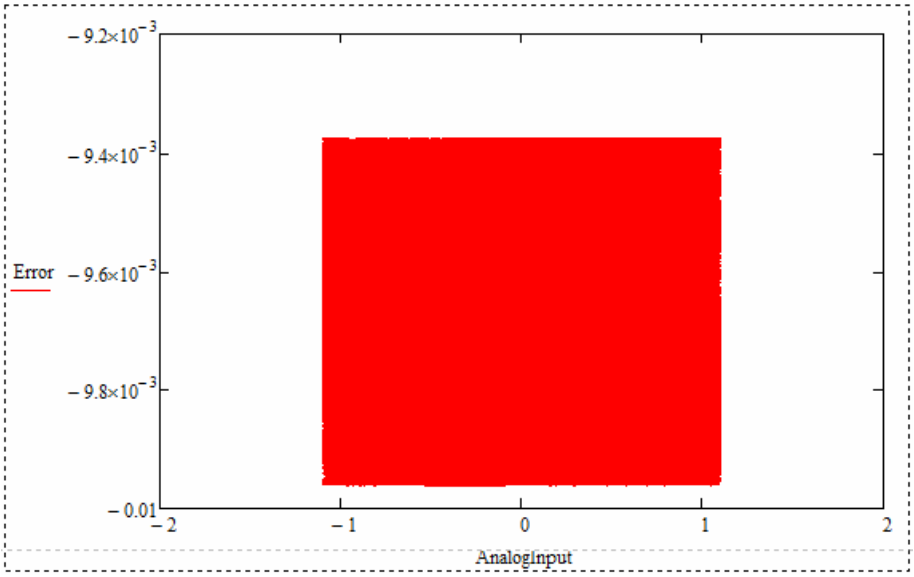

Figure 4.3: Error plot of ideal 12-bit ADC

The complete layout of the digital control logic obtained from the behavioral code which was made to pass through the flow explained earlier is shown in Figure 4.4. The layout area is found to be 1.55mm^2(2035µm x 760µm).

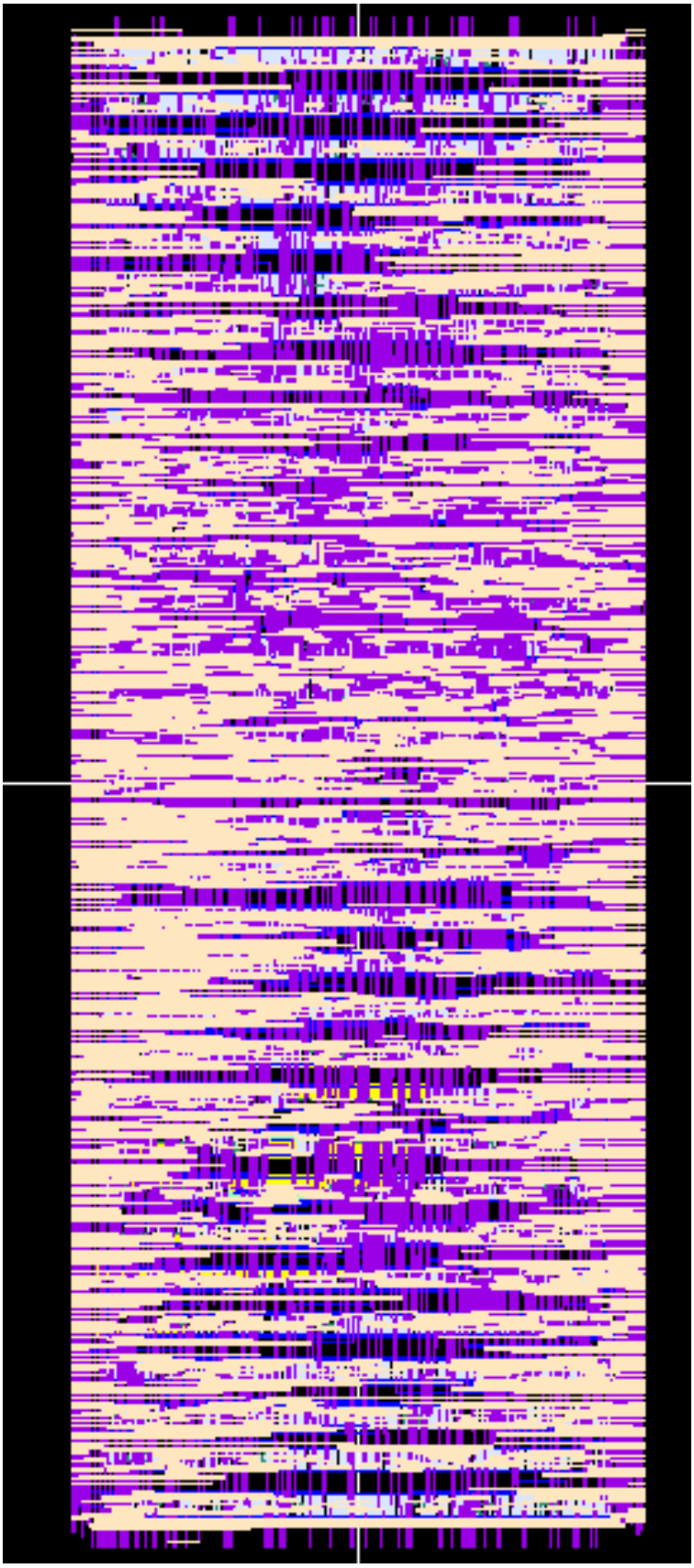

Figure 4.4: Layout of digital control logic

CHAPTER 5

SUMMARY/FUTURE WORK

Summary

This thesis presented the design of the digital control logic for a 12-bit, 1.7 Mega Samples/sec two-stage flash Analog-to-Digital Converter (ADC) intended for use in a family of integrated circuits (ICs) used in the detection of ionizing radiation. A standard cell library compatible with the AMI design kit is developed based on the OSU standard cell library using a script written in Perl. We also explained the complete flow for generating automatic layout compatible with the AMI design kit starting with behavioral level Verilog code.

The ADC may be implemented in the future in a 5-Volt AMIS 0.5 μm, double-poly, tri-metal CMOS process (C5N). The converter described in this thesis employs a two-step flash technique with resistive DAC and is configured as a fully differential circuit. It performs 7-bit coarse flash conversion followed by 6-bit fine flash conversion combined through a digital error correction to produce the 12-bit output. The complete automatic layout of the digital control logic is done and the area is found to be 1.55mm^2 (2035μm x 760μm).

An electrical simulation of the ADC (utilizing a resistive rather than a capacitive DAC) was performed. A FET-level schematic, produced by the synthesis flow, was used to simulate the digital control logic. In the simulation, the ADC is supplied with input analog voltages which are uniformly distributed between -1.1 Volts and +1.1 Volts (referenced to AGND). The digital output of the ADC is captured and then analyzed using a tool developed in MathCAD®. The systematic error of the DAC was found to be very small. The effective number of bits (ENOB)

for the converter in the absence of typical offsets and mismatch errors was found to be 11.9 bits.

Future Work

The clock generator circuit needs to be designed which has only VerilogA model in this thesis. The reference voltage generator and the input voltage driver circuit needs to be designed. The operational amplifier used in the single-ended-to-differential converter and reference generator circuits need to be modified so that they settle more quickly. The modified operational amplifier must be integrated with the other components to check for any loading effects.

The complete layout of the ADC needs to be performed. After complete layout of the ADC is done, the ADC has to be tested across worst case speed and worst case power corners since it is already tested across typical corner. The ADC also needs to be tested at clock frequency of 5 Msamples/sec. Finally, the ADC has to be combined with the on-chip RAM and I2C interface which needs to be integrated onto the PSD-8C chip.

REFERENCES

[All:03] Phillip E. Allen and Douglas R. Holberg, *CMOS Analog Circuit Design*, Second Edition, Oxford University Press, (2003).

[Asr] Asral bin Bahari Jambek, Ahmad Raif bin Mohd Noor Beg and Mohd Rais Ahmad, "Standard Cell Library Development", Library Development Group, Malaysia.

[Das:07] Dinesh Dasari and Michael Hall, "Modeling and simulation of a Two Stage Flash ADC" SIUE EE ECE 585.REPORT [2007].

[Das:08] Dinesh Dasari, "Design of On-chip ADC for custom ASICS Used in the Detection of Ionizing Radiation" SIUE EE M.S. Thesis, (2008).

[Enc:08] Cadence Encounter User Guide, Product Version 8.1, November 2008.

[Eng:07a] G. Engel, M. Sadasivam, M. Nethi, J. M. Elson, L. G. Sobotka and R. J. Charity, "Multi-channel integrated circuit for use in low and intermediate energy nuclear physics - HINP16C" in Nucl. Instru. Meth. A 573, 418-426, (2007).

[Eng:07b] G. Engel, NSF-MRI proposal. This can be found at the PI's WEB site.

[Eng:09] G. Engel, M. Hall, J. Proctor, J. Elson, L. Sobotka, R. Shane, R. Charity, "Design and performance of a multi-channel, multi-sampling, PSD-enabling integrated circuit" in Nucl. Instru. Meth. A 612, 161-170, (2009).

[Gan:00] M. Ganesan, "CMOS low-noise IC for capacitive detector," SIUE EE M.S. Thesis, (2000).

[Gra:03] Johannes Grad and James E. Stine, "A Standard Cell Library for Student Projects", Department of Electrical Engineering, IIT Chicago.

[Hal:07] M. Hall, "Design Considerations in Systems Employing Multiple Charge Integration for the Detection of Ionizing Radiation" SIUE EE M.S. Thesis, (2007).

[Loh:05] Prof. Loh, "Carry Save Addition CS3320-Processor Design Spring 2005",February 2005.

[Mal:01] M. Malikansari, "A CMOS low noise IC for use in colliding particle experiments," SIUE EE M.S. Thesis, (2001).

[Ncl:08] Cadence NCLaunch User Guide, Product Version 8.1, May 2008.

[Ncv:08] Cadence NC-Verilog Simulator help, Product Version 8.1, May 2008.

[Ngu:08] N. Nguyen, "Design of I^2C Interface for Custom ASICs Used in the Detection of Ionizing Radiation," SIUE EE M.S. Thesis, (2008).

[Pro:07] J. Proctor, "Design of a Multi-Channel Integrated Circuit for Use in Nuclear Physics Experiments Where Particle Identification is Required", SIUE EE M.S. Thesis, (2007)

[Raz:92] B. Razavi, B.A. Wooley, "A 12 b 5Msample/s Two-Step CMOS A/D Converter", JSSC, Vol. 27, No. 12, 1667-1678 (1992).

[Raz:01] B. Razavi, *Design of Analog CMOS Integrated Circuits*, McGraw Hill, (2001).

[Rtl:08] Using Cadence Encounter RTL Compiler, Product Version 8.1.2, December 2008.

[Sad:02] M. Sadasivam, "A Multi-channel Integrated Circuit For Use With Silicon Strip Detectors In Experiments In Low and Intermediate Physics," SIUE EE M.S. Thesis, (2002).

[Spi:05] Helmut Spieler, *Semiconductor Detector Systems*, Oxford University Press, (2005).

[Wik] Wikipedia Carry Save Adder reference http://en.wikipedia.org/wiki/Carry_save_adder

APPENDIX A
Verilog Code

```
//Verilog HDL for "nagaLib", "ADC_digital_logic" "behavioral"

//
// Verilog behavioral model of digital control logic
//
// We model the following:
//
// Logic to Correct Out-of-Order Ones and Zeros
// Encoder Logic
// Register Logic
// Carry Save Adder Block
// Digital Correction Logic Block
// Controls for DAC
//

module ADC_digital_logic(Y,Over_flow,Under_flow,DAC_plus_ctrl,DAC_minus_ctrl,
                         DAC_plus_ctrl_bar, DAC_minus_ctrl_bar,Data_coarse,Data_fine,
                         clear,clk,my_vdd,my_gnd);

    parameter M = 7; // Declaration of bits for the first stage of the two-step flash ADC
    parameter N = 6; // Declaration of bits for the first stage of the two-step flash ADC
    parameter J = 128;
    parameter K = 64;

    input  [J:0]   Data_coarse; //FSC comparators output array as input
    input  [K-2:0] Data_fine; //SSC comparators output array as input
    input        clear;
    input          clk;//Clock signal for digital control logic
    input         my_vdd,my_gnd ; //Dummy supply pins for layout purpose

    output [M+N-2:0] Y; //Output of the ADC
    output Over_flow;
    output Under_flow;
    output [J-1:0] DAC_plus_ctrl,DAC_minus_ctrl; // DAC controls
    output [J-1:0] DAC_plus_ctrl_bar,DAC_minus_ctrl_bar; // DAC controls

    wire [J:0]   FSC ;
    wire [K-2:0] SSC ;
    wire [M-1:0] Coarse ;
    wire [N-1:0] Fine ;
    wire [M-1:0] Coarse_out ;
    wire [N-1:0] Fine_out ;
    wire         u_flow,o_flow ;
    wire         Under_flow,Over_flow,Over_flow1;
    wire [M+N-2:0] Y,Y1;

    wire [J-1:0] DAC_plus_ctrl,DAC_minus_ctrl;
    wire [J-1:0] DAC_plus_ctrl_bar,DAC_minus_ctrl_bar;

    assign FSC = Data_coarse;
    assign SSC = Data_fine;
```

```
//
//Instantiation of logic to correct out of order ones and zeros and to convert the
//thermometer code output of the first stage comparators into binary code module
//
    test_encoder       #(J,M)    t_Coarse(Coarse,u_flow,o_flow,FSC,my_vdd,my_gnd);
//
//Instantiation of logic to correct out of order ones and zeros and to convert the
//thermometer code output of the second stage comparators into binary code module
//
    test_encoder_fine  #(K,N)    t_Fine(Fine,SSC,my_vdd,my_gnd);
//
//Instantiation of logic to store the digitized outputs of first and
//second stage in register logic module
//
    registers_logic    #(M,N)  r3(Coarse_out,Fine_out,Under_flow,Over_flow,Coarse,Fine,
                                      u_flow,o_flow,1'b1,clear,clk,my_vdd,my_gnd);
//
//Instantiation of digital correction logic module
//
    digital_correction #(M,N)    d1(Y1,Over_flow1,Coarse_out,Fine_out,Under_flow,Over_flow
                                       clk,my_vdd,my_gnd);
//
//Instantiation of DAC controls logic module
//
    DAC_controls     #(M,J)  d_controls(DAC_plus_ctrl,DAC_minus_ctrl,DAC_plus_ctrl_bar,
                                            DAC_minus_ctrl_bar,Coarse,my_vdd,my_gnd);
//
//Instantiation of regsiter module to store the 12-bit output of the ADC
//
    register         #(M+N-1) r4(Y,Y1,1'b1,clear,clk,my_vdd,my_gnd);

endmodule

//
//Logic to correct out of order ones and zeros and convert the thermometer code outputs of
//first stage comparators in to binary code
//

module test_encoder(Y,under_flow,over_flow,Data,my_vdd,my_gnd);
 parameter  N=8;
 parameter  M=3;
 input      [N:0] Data;
 input          my_vdd,my_gnd ;

 output     [M-1:0] Y;
 output     under_flow,over_flow;

 wire  [N-1:0] Y1;

comparator_to_encoder #(N)  c1(Y1,under_flow,over_flow,Data,my_vdd,my_gnd);
encoder #(N,M)   e1(Y,Y1,my_vdd,my_gnd);
endmodule
```

```
//
//Logic to correct out of order ones and zeros and convert the thermometer code outputs of
//second stage comparators in to binary code
//

module test_encoder_fine(Y,Data,my_vdd,my_gnd);
 parameter  N=8;
 parameter  M=3;
 input      [N-2:0] Data;
 input          my_vdd,my_gnd ;

 output     [M-1:0] Y;

 wire  [N-1:0] Y1;

comparator_to_encoder_fine #(N)  c1(Y1,Data,my_vdd,my_gnd);
encoder #(N,M)   e1(Y,Y1,my_vdd,my_gnd);
endmodule

//
//Logic to correct out of order ones and zeros for the first stage
//

module comparator_to_encoder(Y,under_flow,over_flow,Data,my_vdd,my_gnd);
 parameter  N = 8;
 input      [N:0] Data;
 input          my_vdd,my_gnd ;

 output     [N-1:0] Y;
 output     under_flow,over_flow;
 reg  [N:0] w1;
 reg  [N-1:0] Y;
 reg  [N-1:0] Y1;
 reg        under_flow,over_flow;
 integer     k;


 always @ (Data)
  begin
 Y[N-1:0] = 0;
 under_flow = 0;
 over_flow  = 0;
 if (Data[N-1:1] == 0)
  Y[0] = 1'b1;
 else
   begin
  w1[N:0]  = {1'b0,Data[N-1:1],1'b1};
  for(k=1;k<N;k=k+1)
       begin
         Y[k] = ~w1[k+1]*w1[k]*w1[k-1];
        end
    end
  if(Data[0] == 0)
  under_flow = 1;
  if(Data[N] == 1)
  over_flow  = 1;
```

```
 end

endmodule
//
//Logic to correct out of order ones and zeros for the second stage
//

module comparator_to_encoder_fine(Y,Data,my_vdd,my_gnd);
 parameter  N = 8;
 input      [N-2:0] Data;
 input          my_vdd,my_gnd ;

 output     [N-1:0] Y;

 reg  [N:0] w1;
 reg  [N-1:0] Y;
 reg  [N-1:0] Y1;
 integer     k;


 always @ (Data)
  begin
 Y[N-1:0] = 0;
 if (Data == 0)
  Y[0] = 1'b1;
 else
   begin
  w1[N:0]  = {1'b0,Data,1'b1};
  for(k=1;k<N;k=k+1)
       begin
         Y[k] = ~w1[k+1]*w1[k]*w1[k-1];
        end
    end
  end

endmodule
//
//Logic to encode the input vector into an equivalent binary code
//

module encoder(Y,Data,my_vdd,my_gnd);
 parameter  N=8;
 parameter  M=3;
 input      [N-1:0] Data;
 input          my_vdd,my_gnd ;

 output     [M-1:0] Y;

 reg [M-1:0] Y;
 integer     i;

 always @(Data)
  begin

   for (i=0;i<N;i=i+1)
     if (Data[i])    Y <= i;
```

```
 end
endmodule

//
//Logic to store the digitized outputs of first and
//second stage in register logic module
//

module
registers_logic(out1,out2,out3,out4,inp1,inp2,inp3,inp4,load,clear,clk,my_vdd,my_gnd);
       parameter M          = 2;
       parameter N          = 2;
       parameter L          = 1;
          input  [M-1:0] inp1;
       input  [N-1:0] inp2;
       input  [L-1:0] inp3;
       input  [L-1:0] inp4;
       input  load,clear,clk;
       input          my_vdd,my_gnd ;

          output [M-1:0] out1;
       output [N-1:0] out2;
       output [L-1:0] out3;
       output [L-1:0] out4;

register_delay #(M)    r1(out1,inp1,load,clear,clk,my_vdd,my_gnd);
register       #(N)    r2(out2,inp2,load,clear,clk,my_vdd,my_gnd);
register       #(L)    r3(out3,inp3,load,clear,clk,my_vdd,my_gnd);
register       #(L)    r4(out4,inp4,load,clear,clk,my_vdd,my_gnd);

endmodule

//
//Logic for delaying the first stage comparator outputs by one clock cycle
//

module register_delay(out,inp,load,clear,clk,my_vdd,my_gnd);
       parameter N =  4;
       input  [N-1:0] inp;
       input  load,clear,clk;
       input          my_vdd,my_gnd ;

       output [N-1:0] out;
       reg    [N-1:0] outp;
       reg    [N-1:0] out;

 always@(posedge clk)
 begin
 if(clear)
 outp <= 0;
 else if(load)
 outp <= inp;
 else
 outp <= outp;
 end
```

```
 always@(posedge clk)
 begin
 out <= outp;
 end

endmodule

//
//Register logic
//

module register(outp,inp,load,clear,clk,my_vdd,my_gnd);
       parameter N          = 4;
          input  [N-1:0] inp;
       input  load,clear,clk;
       input          my_vdd,my_gnd ;

          output [N-1:0] outp;
       reg    [N-1:0] outp;

 always@(posedge clk)
 begin
   if(clear)
   outp <= 0;
   else if(load)
   outp <= inp;
   else
   outp <= outp;
 end

endmodule

//
//Digital correction logic
//

module digital_correction (Y,Overflow,A,B,u_flow,o_flow,clk,my_vdd,my_gnd);
    parameter A_size = 3;
    parameter B_size = 2;

    parameter total_size = A_size + B_size - 1;

    output [total_size-1:0] Y ;
    output Overflow;

    input [A_size-1:0] A;
    input [B_size-1:0] B;
    input            clk;
    input            u_flow,o_flow;
    input          my_vdd,my_gnd ;


    wire  [total_size-1:0] Op1 ;
    wire  [total_size-1:0] Op2 ;
    wire  [total_size-1:0] Op3 ;
```

```
    wire  [total_size-1:0] Cout1 ;
    wire  [total_size-1:0] Sum1 ;
    wire  [total_size-1:0] Sum2 ;
    wire  Cout2,Cout2b ;
    wire  [A_size-1:0] nA;
    wire  [total_size-1:0] Yi,Y1,Y ;

    assign nA = ~A;

    assign Op1 = A << (B_size-1);
    assign Op2 = B;
    assign Op3 = ~(1 << (B_size-2));

//
// Instantion of carry save adder block
//
    CSA_block #(total_size) CSA1(.Cout(Cout1), .Sum(Sum1), .A(Op1), .B(Op2),
                                 .C(Op3),.my_vdd(my_vdd),.my_gnd(my_gnd));

    assign {Cout2,Sum2} = Sum1 + {Cout1,1'b1} ;
    assign Yi = (A==0) ? B : Sum2;
    assign Overflow = ((nA == 0) && B[B_size-1] && B[B_size-2])|o_flow ;
    assign Y1 = (Overflow ? ~0 : Yi);
    assign Y  = ( u_flow  ?  0 : Y1);

 endmodule

//
// Carry save adder block
//

module CSA_block(Cout,Sum,A,B,C,my_vdd,my_gnd);
 parameter size = 3;

 output [size-1:0] Cout;
 output [size-1:0] Sum;

 input [size-1:0] A;
 input [size-1:0] B;
 input [size-1:0] C;
 input         my_vdd,my_gnd ;

 assign Sum = A ^ B ^ C;
 assign Cout = A & B | A & C | B & C ;

endmodule
```

```
//
// DAC controls logic
//

module DAC_controls(DAC_plus_ctrl,DAC_minus_ctrl,DAC_plus_ctrl_bar,
                        DAC_minus_ctrl_bar,Coarse,my_vdd,my_gnd);
    parameter M = 3;
    parameter J = 8;

    input  [M-1:0] Coarse ;
    input          my_vdd,my_gnd ;

    output [J-1:0] DAC_plus_ctrl,DAC_minus_ctrl;
    output [J-1:0] DAC_plus_ctrl_bar,DAC_minus_ctrl_bar;

    reg    [J-1:0] DAC_plus_ctrl,DAC_minus_ctrl;
    wire   [J-1:0] DAC_plus_ctrl_bar,DAC_minus_ctrl_bar;

    integer i;

    always@(Coarse)
      begin
        for(i=0;i<J;i=i+1)
          begin
          DAC_plus_ctrl[i] = 0;
          DAC_minus_ctrl[i] = 0;
          end
        for(i=1;i<=J;i=i+1)
          begin
          case(Coarse)
                        i:        DAC_plus_ctrl[i-1] = 1;
          endcase
          case((J+1)-Coarse)
                        i:        DAC_minus_ctrl[i-1] = 1;
          endcase
          end
      end
   assign DAC_plus_ctrl_bar  =  ~DAC_plus_ctrl;
   assign DAC_minus_ctrl_bar =  ~DAC_minus_ctrl;

endmodule
```

APPENDIX B
Testbench Code

```
module ADC_digital_logic_tb;
    parameter M = 7;
    parameter N = 6;
    parameter J = 128;
    parameter K = 64;
    parameter real T = 625;
    parameter real phi1 = T/3;
    parameter real d = 2*phi1;
    parameter real finish_time = J*K*T;

    reg  [J:0]   Data_coarse;
    reg  [K-2:0] Data_fine;
    reg         clear;
    reg           clk;

    wire  [M+N-2:0] Y;
    wire     Overflow;

    integer       j,k;
    integer         fid;    /* file handle for output file */


            /* invoke an instance of a test_ADC_logic within the test bench */
      ADC_digital_logic #(M,N,J,K)
ADC_digital_logic1(Y,Over_flow,Under_flow,DAC_plus_ctrl,DAC_minus_ctrl,
                                    DAC_plus_ctrl_bar,
DAC_minus_ctrl_bar,Data_coarse,Data_fine,clear,clk);

            /* first we need to setup the clock signal */
            initial begin
                clk = 1'b0;
                forever begin #d    clk <= ~clk;
                            #phi1    clk <= ~clk;
                $display("At Time: %d  Y = %d",$time,Y);
                      end
                end

            initial
               begin
                  /* Setup the loging facilities */

                  $shm_open("shm.db",1); // Opens a waveform Database
                     $shm_probe("AS");    // Saves all signals to Database

                  /* open a file handle for the output file */
                  fid = $fopen("./ADC_logic.out");

                  $fmonitor(fid, " %d ",Y) ;

                  /* Start stimulating the module */

                  clear <= 1'b1;
                #(1.5*T)  clear <= 1'b0;
```

```
                Data_coarse[J:0] <= 0;

              for(j=1;j<J;j=j+1)
              Data_coarse[j] <= 0;
            #(T)  Data_fine[K-2:0] <= Data_coarse[K-1:1];
                 Data_coarse[0]   <= 1 ;

            for(j=1;j<J;j=j+1)
              begin
              Data_coarse[j] <= 1;

                 for(k=0;k<K-1;k=k+1)
            #T  Data_fine[k] <= 1;
               Data_coarse[0]   <= Data_fine[0] ;
            #T  Data_fine[K-2:0] <= #(K)'h00 ;
               Data_coarse[0]   <= Data_fine[0] ;
              end

            #T      Data_coarse[J:0] <= 0 ;
            #T      Data_coarse[J]   <= 1 ;
            #T      Data_coarse[J:0] <= 0 ;
            #T      Data_coarse[0]   <= 0 ;

              #(T+T)      $finish;
              #(T+20) $shm_close();   // Closes the waveform Database
             end
endmodule
```

APPENDIX C
VerilogA Code

Three phase Clock Generator

```
// VerilogA for ADCdkdLib, FSC_three_phase_clock, veriloga

`include "constants.vams"
`include "disciplines.vams"

module FSC_three_phase_clock(Phi1, Phi1bar, Phi2, Phi2bar, Phi3, Phi3bar);
output Phi1;
electrical Phi1;
output Phi1bar;
electrical Phi1bar;
output Phi2;
electrical Phi2;
output Phi2bar;
electrical Phi2bar;
output Phi3;
electrical Phi3;
output Phi3bar;
electrical Phi3bar;


parameter real Fs = 1.6M from (0:inf);              // Clock frequency
parameter real tr = 100p from (0:inf);              // Rise time of clock
parameter real tf = 100p from (0:inf);          // Fall time of clock

parameter real VDD = 5;
parameter real VSS = 0;

parameter real Ts = 1/Fs;
parameter real td12 = 10e-9;

integer Phi1_val;
integer Phi2_val;
integer Phi3_val;

analog begin

@(initial_step) begin
                Phi1_val = 1;
                Phi2_val = 0;
                Phi3_val = 0;

                end

                // Create phi 1 & 2 logic signals using timers
                @(timer(0,Ts))
                        Phi1_val = 1;
                @(timer((Ts/3)-0.1*td12,Ts))
                        Phi1_val = 0;

                @(timer(0,Ts))
                        Phi2_val = 0;
                @(timer((Ts/3),Ts))
```

```
                Phi2_val = 1;
            @(timer((2*Ts/3)-0.1*(td12),Ts))
                Phi2_val = 0;
            @(timer(0,Ts))
                Phi3_val=0;
            @(timer(2*Ts/3,Ts))
                Phi3_val=1;
            @(timer((Ts)-(0.1*td12),Ts))
                Phi3_val=0;

            // Convert those logic signals to voltages and apply a smooth transition.
            V(Phi1) <+ transition((Phi1_val ? VDD : VSS), 0, tr, tf);
            V(Phi2) <+ transition((Phi2_val ? VDD : VSS), 0, tr, tf);
            V(Phi1bar) <+ transition ((Phi1_val ? VSS : VDD), 0, tr, tf);
            V(Phi2bar) <+ transition ((Phi2_val ? VSS : VDD), 0, tr, tf);
            V(Phi3) <+ transition ((Phi3_val ? VDD : VSS), 0, tr, tf);
            V(Phi3bar) <+ transition ((Phi3_val ? VSS : VDD), 0, tr, tf);
        end
endmodule
```

APPENDIX D
Perl Script

```
#! /usr/bin/perl
#
# Reading the input file name as a argument
#
chomp($Inp_File = @ARGV[0]);
if(length($Inp_File) == 0)
{
print "You should provide a valid file name \n ";
}

if($Inp_File =~ /(\w+)/)
{
$Inp_File = $1 ;
}
#
#Creating a output file name same as input,but with addition of "_AMI" at the end
#
$Out_File = $Inp_File."_AMI";
#
#Opening the input file in readable mode
#
open $fid, "<$Inp_File.cif";
#
#Creating the output file in writable mode
#
open $fid1,">$Out_File.cif";
#
#Reading the input file data in to an array
#
@Inp_data = <$fid>;
#
#Calling a subroutine for editing the input data
#
@Edited_data   = &Editing_cif(@Inp_data);
#
#Calling a subroutine for fixing the Contact size,Pselect Nselect,Nfield and Tub
#
@Fixed_contact = &Fixing_contact(@Edited_data);
#
#Printing the fixed data in the output file
#
print $fid1 @Fixed_contact ;

close($fid) ;
close($fid1) ;
#
#Subroutine for editing the input data
#
sub Editing_cif
{
my(@Inp_data) = @_ ;
my @Edited_data = ();
my @A1;
```

```
foreach(0..$#Inp_data)
        {
                $Inp_data[$_] =~ s/\;(\w)/\;\n$1/g ;
                @A1 = split (/(?<=\n)/, $Inp_data[$_]);
                push(@Edited_data,@A1);
      }
return @Edited_data ;
}
#
#Subroutine for fixing the Contact size,Pselect Nselect,Nfield and Tub
#
sub Fixing_contact
{
my(@Data) = @_ ;
my $count;
my @PPLS = () ;
my @NFIELD  = () ;

foreach(0..$#Data)
  {
#
#If the layer name is contact,fix the contact size from 0.6umx0.6um to 0.5umx0.5um
#
if($Data[$_] =~ /^L\s+CCC/)
        {

        for($count=1;$count>=1;$count++)
          {

                if($Data[$_+$count] =~ /^[B]/)
        {
                $Data[$_+$count] =~ s/^[B]\s+60\s+60/B 50 50/ ;
                }
                elsif($Data[$_+$count] =~ /^[^B]/)
        {
                last ;
                }
       }
        }
#
#Getting the Pselect layer specs in to an array
#
if($Data[$_] =~ /^L\s+CSP/)
        {
        for($count=1;$count>=1;$count++)
          {
                if($Data[$_+$count] =~ /^[^B]/)
        {
                last ;
                }
        push(@PPLS,$Data[$_+$count]) ;

          }
        }
#
```

```
#Removing the Nselect layer specs and pasting the Pselect specs
#
if($Data[$_] =~ /^L\s+CSN/)
        {
        for(;1;)
          {
                if($Data[$_+1] =~ /^[^B]/ )
        {
                last ;
                }
        splice(@Data,$_+1,1);
          }
        splice(@Data,$_+1,0,@PPLS);
        }
#
#Getting the Nfield specs in to an array and creating TUB layer with the Nfield specs
#
if($Data[$_] =~ /^L\s+CWN/)
        {
        for($count=1;$count>=1;$count++)
          {
                if($Data[$_+$count] =~ /^[^B]/)
        {
                last ;
                }
        push(@NFIELD,$Data[$_+$count]) ;
          }
        splice(@Data,$_+$count,0,"L CNS \n");
        splice(@Data,$_+$count+1,0,@NFIELD);
        }
  }
#
#Returning the fixed data
#
return @Data ;
}
```

APPENDIX E
Tcl Script

```
##rtl.tcl file adapted from http://ece.colorado.edu/~ecen5007/cadence/
##this tells the compiler where to look for the libraries

set_attribute lib_search_path
/home/cdsadmin/vendors/OSUstdcell/cadence/lib/ami05/signalstorm

## This defines the libraries to use

set_attribute library {osu05_stdcells.lib}

##This must point to your VHDL/verilog file
##it is recommended that you put your VHDL/verilog in a folder called HDL in
##the directory that you are running RC out of

## CHANGE THIS LINE to your VHDL/verilog file name, if you follow the tutorial
## you do not need to change anything

##read_hdl ../HDL/accu.v
read_hdl -v2001 ../verilog.src/ADC_logic/ADC_logic_12bit.v

## This buils the general block
elaborate

##this allows you to define a clock and the maximum allowable delays
## READ MORE ABOUT THIS SO THAT YOU CAN PROPERLY CREATE A TIMING FILE
#set clock [define_clock -period 300 -name clk]
#external delay -input 300 -edge rise clk
#external delay -output 2000 -edge rise p1

##This synthesizes your code
synthesize -to_mapped

## Instructing the RTL compiler to remove assign statements in the design if any.
remove_assigns  /designs/ADC_logic_12bit/

## This writes all your files
## change the tst to the name of your top level verilog
## CHANGE THIS LINE: CHANGE THE "accu" PART REMEMBER THIS
## FILENAME YOU WILL NEED IT WHEN SETTING UP THE PLACE & ROUTE
write -mapped > ../netlists/ADC_logic_12bit_synth.v

## THESE FILES ARE NOT REQUIRED, THE SDC FILE IS A TIMING FILE
write_script > ADC_logic_12bit_script

##write_sdc > ../sdc/ADC_logic_12bit.sdc

## SDF FILE IS THE STANDARD DELAY FORMAT FILE
write_sdf > ../sdf/ADC_logic_12bit.sdf
```

APPENDIX F
MATHCAD® Simulator

ADC simulator model parameters

Number of bits will be N.

$N := 12$

Reference voltage

$V_R := 1.2V$

Analog signal ground.

$V_{AGND} := 2 \cdot V_R = 2.4\,V$

Standard deviation of first stage offsets.

$FSosVal := 0.0001mV$

$R_{nom} := 40$

$\varepsilon_{res} := 0.000000025$

Predict capacitor matching characteristics.

$k_c := 1.28\mu m$

$C_{pp} := 0.0009 \frac{pF}{\mu m^2}$

$n := 1$

$\varepsilon_C(C) := \frac{4k_c \cdot C_{pp}^{\frac{1}{2}} \cdot n^{\frac{1}{6}}}{\sqrt{C}}$

Relative error in unit DAC capacitor.

$C_u := 1000$

$\varepsilon_{cap} := 0.01\,\varepsilon_C(C_u \cdot 1pF) = 4.857 \times 10^{-5}$

Standard deviation of second stage offsets.

$A2 := 10000$

$SSosVal := \frac{FSosVal}{A2} = 1 \times 10^{-11}\,V$

Gain of subtractor.

$SubtractorGain := 1$

Generate NUM voltage samples uniformly distributed

$NUM := 99$

$j := 0, 1 .. NUM - 1$

$Last := NUM - 1$

$AnalogInput := runif\left(NUM, \frac{-V_R + 100mV}{1V}, \frac{V_R - 100mV}{1V}\right) \cdot 1V$

The vector analog inputs are then quantized.

$DigitalEquivalent_j :=$ DigitalValue ← TwoStepADC(AnalogInput$_j$)

return DigitalValue

Put the results into a table and then write the table out to an Excel file.

$$\text{OutputTable} := \text{augment}\left(\frac{\text{AnalogInput}}{1\text{V}}, \text{DigitalEquivalent}\right)$$

OutputTable

Read the excel file into an input table

InputTable :=

AnalogInput := 1V submatrix(InputTable, 0, Last, 0, 0)

DigitalEquivalent := submatrix(InputTable, 0, Last, 1, 1)

Use an ideal DAC to convert the digital values back to an analog voltage.

$$\text{AnalogOutput}_j := \left| \begin{array}{l} \text{AnalogValue} \leftarrow \text{DAC}\left(\text{DigitalEquivalent}_j\right) \\ \text{return AnalogValue} \end{array} \right.$$

Compute the error voltage.

$$\text{Error}_j := \left| \begin{array}{l} \varepsilon \leftarrow \text{AnalogOutput}_j - \text{AnalogInput}_j \\ \text{return } \varepsilon \end{array} \right.$$

$$\text{AverageError} := \text{mean}(\text{Error}) = -7.545 \times 10^{-5}\,\text{V} \qquad \text{StandardDeviation} := \text{Stdev}(\text{Error}) = 1.797 \times 10^{-4}\,\text{V}$$

$$\text{Expected} := \frac{\Delta}{\sqrt{12}} = 1.691 \times 10^{-4}\,\text{V}$$

Compute the effective number of bits. $\delta := \sqrt{12} \cdot \text{StandardDeviation} = 6.226 \times 10^{-4}\,\text{V}$

$$\text{ENOB} := \frac{\log\left(\frac{2 \cdot V_R}{\delta}\right)}{\log(2)} = 11.912$$