Backtracking Algorithm-Aided Design of a 10-Bit SAR ADC

by

David Berton, **B.Eng.**

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Electronics
Carleton University
Ottawa, Ontario
August, 2018

Abstract

A 10-bit successive approximation register (SAR) analog to digital converter (ADC) was designed in a 130 nm CMOS process. The reference voltage generator in the ADC is made with an array of 25 unit capacitors, in comparison to binary-weighted SAR ADCs, which would use 1024 unit capacitors. The capacitor array is controlled using a state machine whose logic was determined and automatically generated using a recursive backtracking algorithm.

Schematic-level simulation results for the ADC predict an effective number of bits (ENOB) of 9.39, worst case integral non-linearity (INL) and differential non-linearity (DNL) of 1.15 and 1 least significant bit (LSB), respectively, and average power consumption of 1.65 µW at 10 kSps. The Walden figure of merit is calculated to be 246 fJ/conv-step. The core layout area is 0.458 mm^2.

Acknowledgments

I would like to thank Professor MacEachern for his support throughout this project. He helped to keep me calm when I panicked about insignificant problems and always had helpful suggestions for significant ones. Thanks also to Nagui Mikhail for always being around to help with frantic pleas for hardware and software.

I'd also like to acknowledge Colin Fernandes and Jacob Pike, who at different points were always around to offer advice and encouragement, or distractions when all else failed.

Finally, thank you to my parents. There is no way I would be where I am without their seemingly infinite patience and generosity.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADC Analog to Digital Converter

CMOS Complementary Metal Oxide Semiconductor

DC Direct Current (0 Hz)

DNL Differential Nonlinearity

DRC Design Rule Check

ECG Electrocardiography

EEG Electroencephalography

EMG Electromyography

ENOB Effective Number Of Bits

FFT Fast Fourier Transform

INL Integral Nonlinearity

LSB Least Significant Bit

MSB Most Significant Bit

NFET N-Channel Field Effect Transistor

NMOS N-Channel Metal Oxide Semiconductor

P&R Place and Route

PCB Printed Circuit Board

PFET P-Channel Field Effect Transistor

PMOS P-Channel Metal Oxide Semiconductor

S/H Sample and Hold

SAR Successive Approximation Register

SNDR Signal to Noise and Distortion Ratio

SPDT Single-Pole Double-Throw

Chapter 1

Introduction

1.1 Motivation

Biomedical signals such as ECG, EEG, EMG, and others, operate in the frequency range from DC to a few kilohertz, and sensors commonly convert them to digital signals to take advantage of digital signal processing techniques [1, pp 10-11]. In order to increase the portability and reduce the cost of these sensors, the layout area and power consumption of the Analog to Digital Converters (ADCs) in them must be kept low [2].

Successive Approximation Register (SAR) ADCs are well-suited to low-frequency, low-power, and medium resolution applications [3]. SAR ADCs use an internal voltage generator to generate reference voltages which are compared to the sampled input signal. Typically a binary-weighted capacitor array and simple logic are used to generate this voltage (e.g. [4, 5, 6, 7, 8, 9]), but [3] proposed a charge-sharing switching scheme which allowed the number of unit capacitors to be reduced. Having fewer unit capacitors eases the burden on analog designers and layout engineers. In [10],

an 8-bit SAR ADC was designed which improved on the charge-sharing concept and further reduced the number of unit capacitors. In that work, the number of capacitors required and the design of the state machine to control them were determined with a manual search using spreadsheets.

Automation techniques are commonly used to make complex designs easily transferable and repeatable, and to reduce the work required by analog designers and layout engineers. These techniques typically focus on the sizing and layout of analog components or placement and routing of digital circuit blocks in order to meet design specifications (e.g. [11, 12, 13, 14, 15]).

Additionally, advanced process nodes are designed to improve the performance of digital circuits. Therefore, there is a trend in the industry toward digital correction and control of analog circuits in order to take advantage of these improvements.

In this thesis, the number of required unit capacitors for the SAR ADC proposed in [3] and [10] is further reduced. The improvement is made possible by using a recursive backtracking algorithm implemented in Matlab that determines the precise switching scheme required for a given number of bits and capacitors. The output of the algorithm is then fed to a script which automatically generates the Verilog code to implement the switching scheme determined. A 10-bit prototype using the improved switching scheme was designed in Cadence Virtuoso and fabricated in a 130 nm CMOS process.

1.2 Contributions

The major contribution of this work is the algorithm designed to automate the design of the logic for the SAR ADCs proposed in [3] and [10]. By automating the logical design and transferring much of the complexity to digital circuitry, the burden

on analog designers can be reduced and the advantages of advanced process nodes optimized for digital circuitry can be leveraged. For a 10-bit ADC, the algorithm developed in this work requires 60% fewer unit capacitors than [3] and 97.5% fewer unit capacitors than a traditional binary weighted SAR ADC, so the complexity of the analog layout is greatly reduced. In addition, the number of switches required in the capacitor array for the ADC is reduced by two-thirds compared to [3] and [10]. The fabricated prototype designed in this thesis was non-functional due to a design error unrelated to the core principles developed in the research, but simulations indicate the potential for performance comparable with other recently published SAR ADCs with similar sample rate and resolution.

1.3 Overview

This thesis is organized as follows. Chapter 2 covers some background information about SAR ADCs and backtracking algorithms. Chapter 3 goes into the details of the algorithm used to generate the state machine logic and discusses the structure of the logic itself. Chapter 3 also includes a detailed example conversion. Chapter 4 covers the circuit level design and layout of each component of the ADC. Chapter 5 discusses the verification the circuit-level blocks, and the system-level simulation results. Chapter 6 discusses the testing of the fabricated chip and why it was non-functional, and verifies this with simulations. Chapter 7 concludes the thesis, compares what was achieved to recent publications, and suggests possible future work on this topic.

Chapter 2

Background

This chapter covers the basic operation of a SAR ADC. It then discusses the original design of the "charge-redistribution" converter, and briefly covers some recent designs that are variants of this principle. Finally, backtracking algorithms will be covered as in introduction to the algorithm that will be discussed in detail in Chapter 3.

2.1 SAR ADCs

SAR ADCs typically rely much more heavily on digital circuitry than other ADC methods. Among their benefits are the ability to operate with nominally zero static power dissipation, their relatively small area, and their inherent improvement in performance with scaling process nodes [16][17, pp. 45-47].

A typical SAR ADC block diagram is shown in Figure 2.1. Its basic structure is a negative feedback loop. Unlike many other ADC architectures, SAR ADCs convert the sampled analog input voltage to a digital output one bit at a time, and each bit is fed back to inform the conversion of the next bit. The analog input V_{in} is

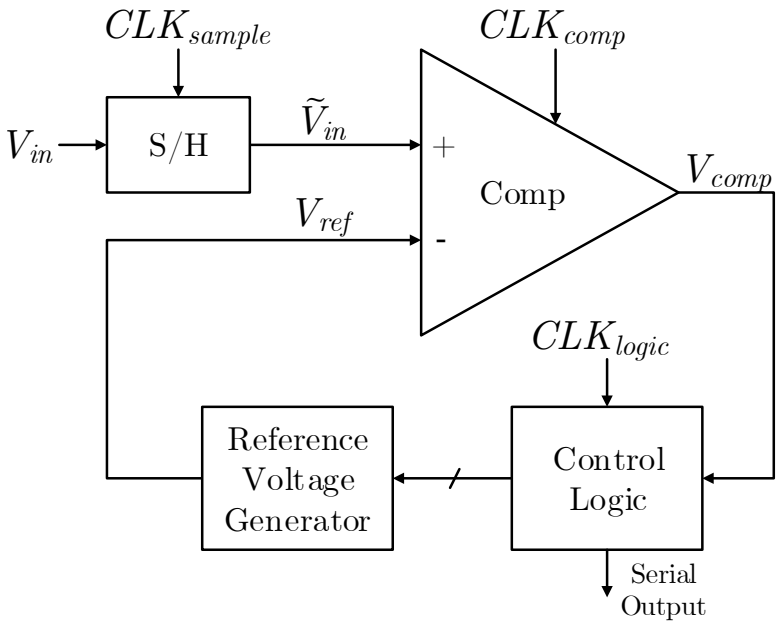


Figure 2.1: Basic SAR ADC block diagram. Note that the design here uses a serial output, but many SAR ADCs have parallel outputs (i.e. the eponymous register).

fed to a sample and hold (S/H) circuit, which holds its output $\tilde{V}_{in}$ constant for the duration of the conversion. Then for each successive output bit, $\tilde{V}_{in}$ is compared to the reference voltage, V_{ref}. On the first comparison, the reference is set to the center of the input range of the ADC (e.g. 0.5 V for a 1 V range), and then for each successive comparison, V_{ref} follows a binary search (see [18, p. 165] for a brief discussion of binary search algorithms and [16] for how they apply to ADCs). If $\tilde{V}_{in}$ is higher than V_{ref}, then the comparator's output will be a logic high and the control logic will send a signal to the reference voltage generator to increase its output for the next step of the conversion; if $\tilde{V}_{in}$ is lower than V_{ref}, then the comparator's output will be a logic low and the control logic will send a signal to the the reference voltage generator to decrease its output, again for the next step of the conversion. The output of the comparator is both a control signal and the binary value for each successive bit. In this way, the circuit generates one bit of the output at a time, from the Most

Significant Bit (MSB) to the Least Significant Bit (LSB).

For example, a 5-bit ADC would be able to generate all the reference voltages shown in the binary tree in Figure 2.2. Figure 2.3 shows how the reference voltage approaches $V_{in} = 0.33$ V over the binary search.

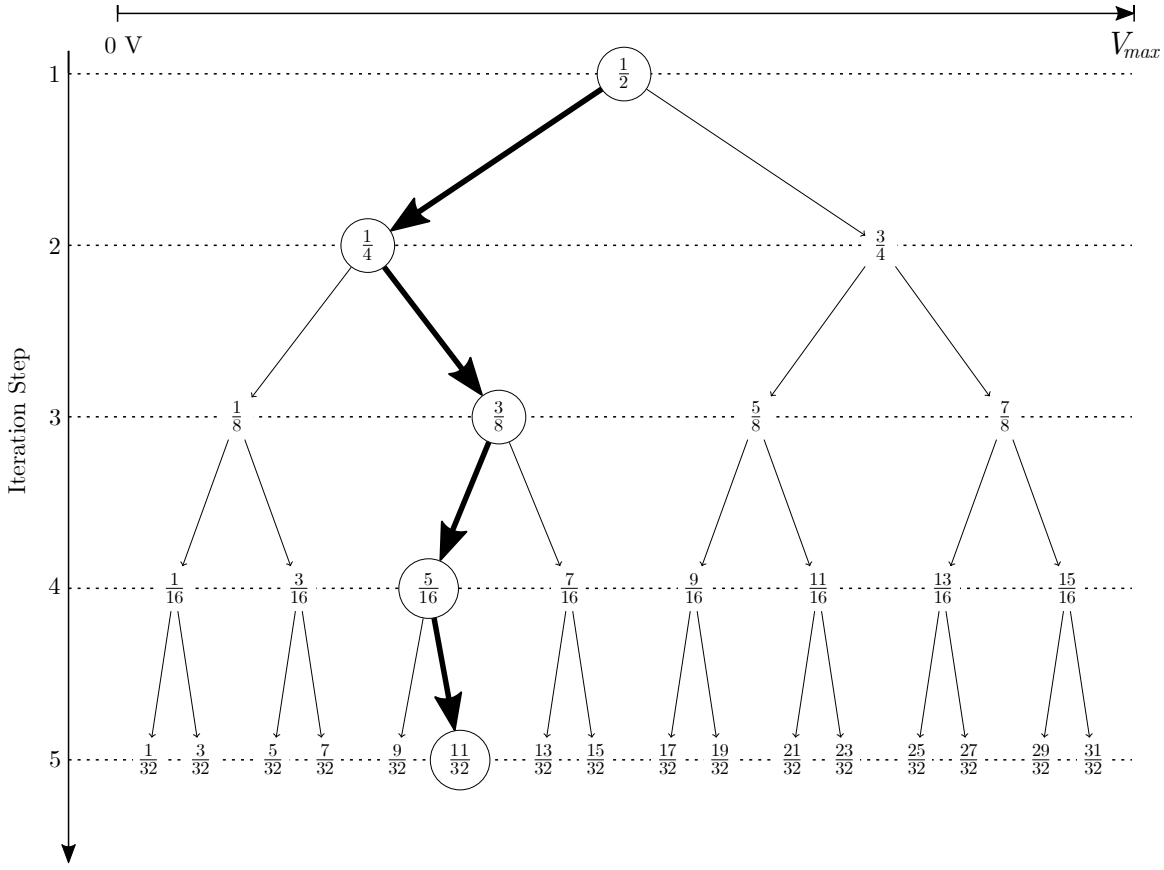


Figure 2.2: Voltages generated by a 5-bit SAR Reference Voltage Generator during a binary search. The "children" (i.e. the two values its arrows point at) of each node v_n, where v_n is in row $n = 1 \rightarrow N$, where N is the number of bits, are calculated as $v_n \pm \frac{1}{2^{n+1}}$. The values generated in Figure 2.3 are circled. All voltages are normalized to V_{max}.

Note that the following terminology will be used in this thesis with regard to trees. Each "node" in the tree is a set of data with connections to other nodes. The first node in the tree is the "root" (e.g. the node containing $\frac{1}{2}$ in Figure 2.2). Nodes

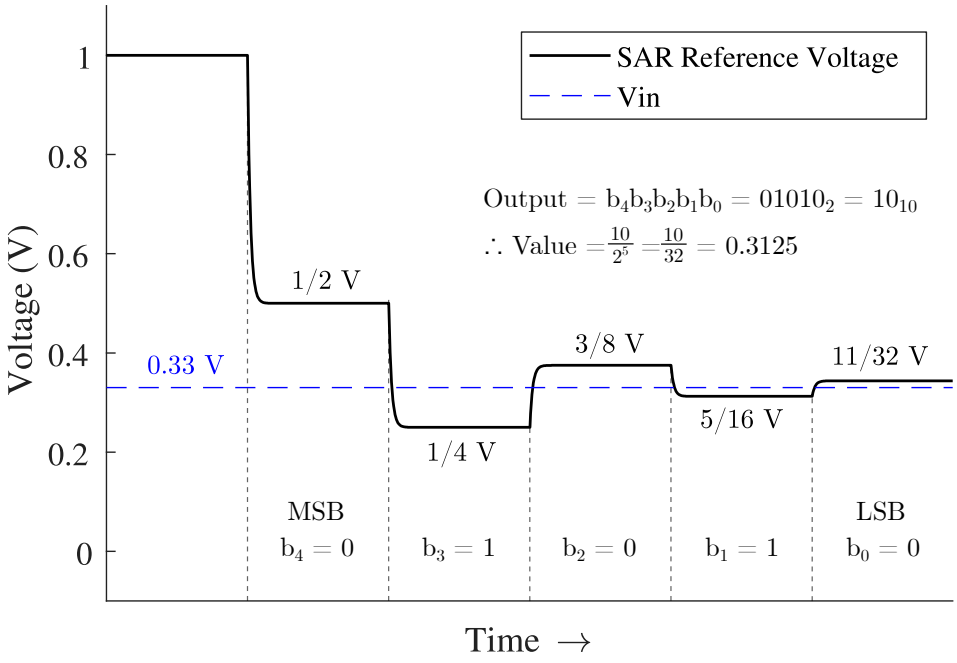


Figure 2.3: The SAR reference voltage approaches the input voltage by performing a binary search.

that follow and precede other nodes are called "parents" and "children", respectively. In Figure 2.2, the root node's children are the two nodes at iteration step 2 (which contain $\frac{1}{4}$ and $\frac{3}{4}$), and conversely the parent of of the nodes at iteration step 2 is the root node. Since every node with children in a binary tree has two children, they will be referred to as the "left" child and the "right" child. Nodes with no children are called "leaves", so every node at iteration step 5 in Figure 2.2 is a leaf.

2.1.1 Binary-Weighted SAR ADC

The binary-weighted charge redistribution SAR ADC on a single chip was first proposed in [5]. The method uses charge on a binary-weighted capacitor array as the conversion medium in the reference voltage generator. The capacitors in the array have values C, $C/2$, ... $C/2^{N-1}$, where N is the resolution (i.e. the number of bits)

of the converter. There are two copies of the smallest capacitor in the array, $C/2^{N-1}$, so that the sum of all the capacitors in parallel is $2C$.

The conversion is performed in three major steps: the "sample mode", the "hold mode", and the "redistribution mode"[1] [19]. Figure 2.4 shows the schematic of the capacitor array during sample mode and the hold mode, as well as their equivalent circuits. In sample mode, S_{A} connects the top plates of the capacitors (which is connected to the negative input of the comparator, and we will call node x) to ground and S_{B} connects their bottom plates to V_{in}. This causes charge $Q_x = -2C \times V_{in}$ to accumulate on node x. Thus in hold mode, when S_{A} opens and $S_{4...0}$ (and S'_0) connect the bottom plates of the capacitors to ground, the voltage on node x is held at $-\tilde{V}_{in}$ (where $\tilde{V}_{in} = V_{in}$ at the instant the switches switch from sample mode to hold mode). The hold mode also connects S_{B} to V_{max} in preparation for the redistribution mode.

Figure 2.5 shows the capacitor array during the conversion of the first two bits (b_4 and b_3) in the redistribution mode. To decide b_4, S_4 connects the bottom plate of the largest capacitor to V_{max}. This causes the voltage at node x to step to $V_x = -\tilde{V}_{in} + V_{max}/2$. If $V_{in} > V_{max}/2$, then $V_x > 0$ and the comparator will output a binary 1, indicating $b_4 = 1$. If $V_{in} < V_{max}/2$, then $V_x < 0$ and the comparator will output a binary 0, indicating $b_4 = 0$.

The value of b_4 is then fed to the logic circuitry to inform the conversion of the next bit. S_3 will connect the bottom plate of the second largest capacitor to V_{max} regardless of the value of b_4. If $b_4 = 1$, then S_4 will connect the bottom plate of largest capacitor back to ground, and the voltage at node x will be $V_x = -\tilde{V}_{in} + 3V_{max}/4$. If $b_4 = 0$, then S_4 will keep the bottom plate of largest capacitor connected to V_{max}, and the voltage at node x will be $V_x = -\tilde{V}_{in} + V_{max}/4$. Then, V_x is compared to 0

[1] Note that one benefit of this method is that the sampling function is accomplished by the array itself, eliminating the need for additional S/H circuitry.

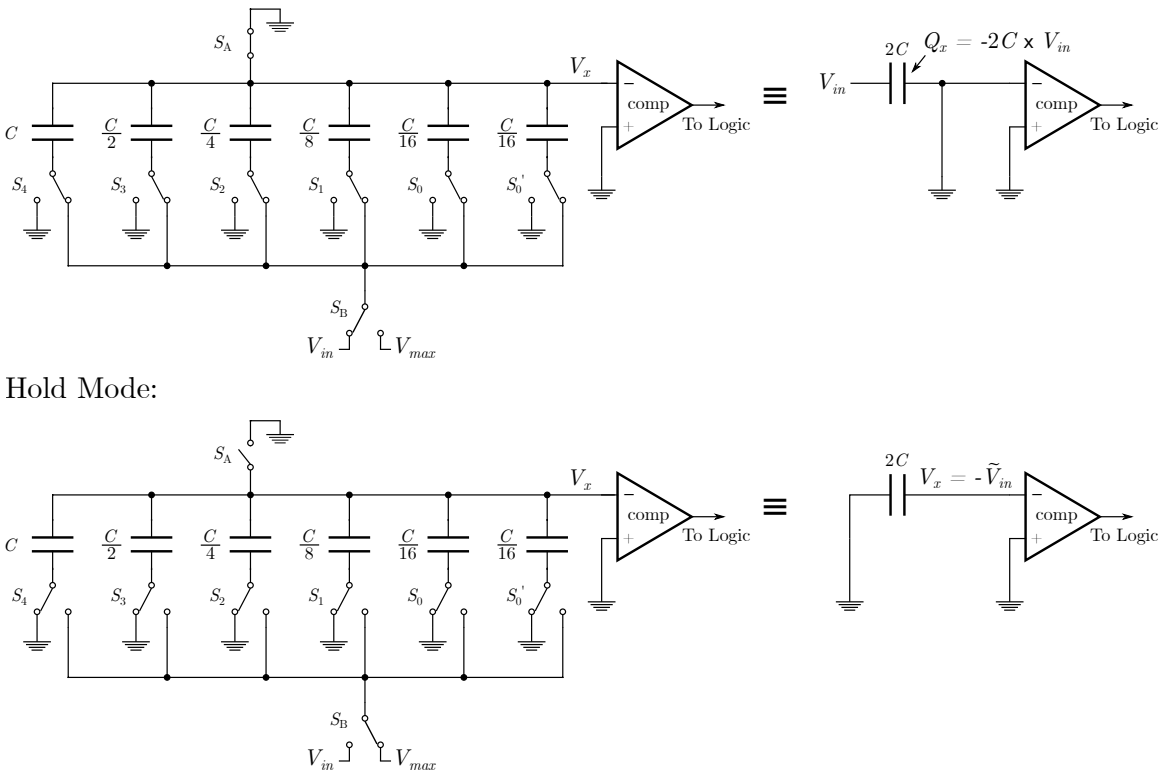


Figure 2.4: Sample mode and Hold mode for a binary-weighted charge redistribution SAR ADC, reproduced from [19]. The equivalent circuit for each mode is shown on the right. In "sample mode" the input voltage is sampled on the bottom plates of the capacitors in the array, with the top plates connected to ground. In "hold mode" the top plates are disconnected from ground, and the bottom plates are connected to ground, so the top plates end up charged to $-V_{in}$.

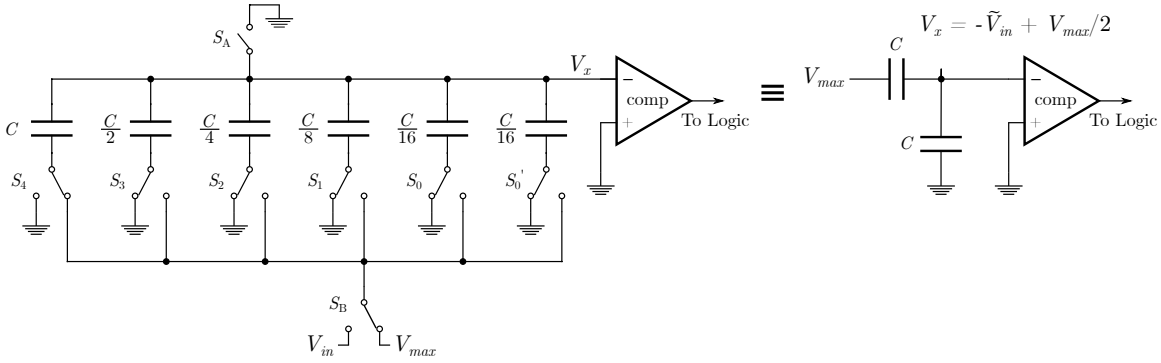
V again, and the same logic as above is applied to determine the value of b_3. This conversion continues this way for each bit, where the result of each comparison is the value of the bit being determined, and also tells the logic whether to keep previous switch connected to ground or to remain connected to V_{max}.

Note that this method is slightly different from that described above and shown in Figure 2.1, as the voltage is being subtracted from the sampled input and compared to ground, rather than the sampled input being directly compared to the generated voltage. However, the principle of generating voltages which perform a binary search applies to both instances and specific architectures vary slightly.

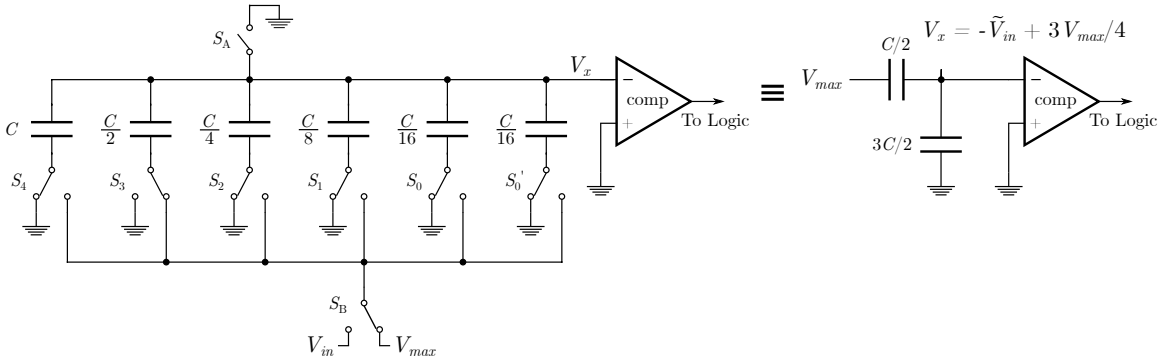
In order to minimize mismatch in the capacitors, a unit capacitor, C_u, is usually chosen as the capacitor for the least significant bit (LSB) - i.e. the smallest capacitor(s) in the array. The remaining larger capacitors are then built up of many of these unit capacitors in parallel, such that the largest capacitor, representing the most significant bit (MSB), $C_{MSB} = 2^{N-1} \times C_u$ and the capacitors representing each following bit is half that. This means that the total number of unit capacitors is 2^N, which means the number of capacitors doubles for each bit added. Assuming the capacitors stay the same size, this also corresponds to the area and the amount of energy required to initially charge the capacitors for each conversion doubling. Splitting the capacitors into arrays of unit capacitors allows them to be laid out in a common-centroid pattern, and also maintains consistent parasitics caused by edge effects [23, p. 300].

The method of using a binary-weighted charge redistribution capacitor array is still widely used today. Recent publications using the idea or variations of it include [7], [8], [9], and [20]. [7] and [9] use a binary-weighted array for the LSBs, but use a thermometer-coded array for the MSBs in order to avoid problems associated

Conversion Step for bit 4 (MSB):



If bit 4 = 1, V_{in} is compared to $3\,V_{max}/4$:



If bit 4 = 0, V_{in} is compared to $V_{max}/4$:

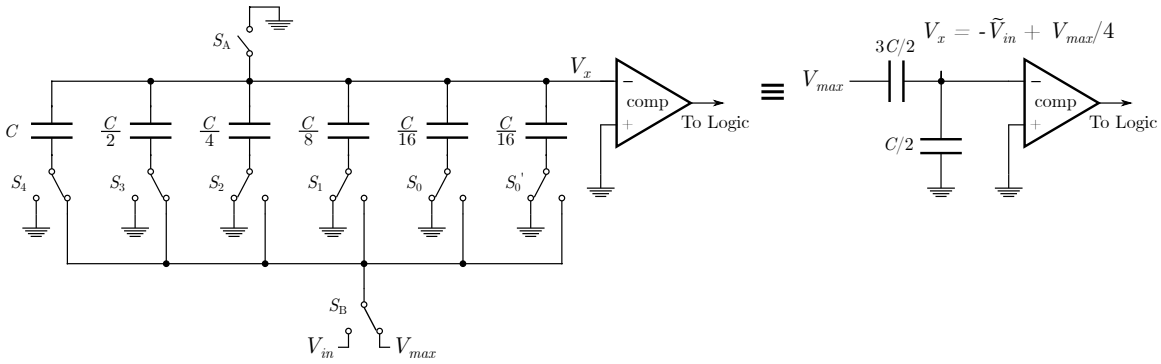


Figure 2.5: The first two conversion steps for a binary-weighted SAR ADC, reproduced from [19]. The equivalent circuit for each step is shown on the right.

with matching the largest capacitors in the binary-weighted array. [8] uses a binary-weighted array, and focuses on low-power by reducing leakage currents and using a dual-supply to operate the digital circuitry at 0.4 V and the analog circuitry at 1 V. [20] uses a variation of a "split" or "bridge" capacitor array, which is a technique used to reduce the number of unit capacitors required in the array [16].

2.1.2 Unit Capacitor Array Charge Sharing

In [3], a variation on the charge-sharing SAR ADC was proposed which does not use a binary weighted capacitor array.

When two equally sized capacitors, C_1 and C_2, with charge on them are connected together, the charge is distributed evenly between them. Since they are being connected in parallel, the capacitance adds and the voltage is

$$V = \frac{Q_1 + Q_2}{C_1 + C_2}. \tag{2.1}$$

Since all the capacitors in the proposed array are the same value, C_u, (2.1) can be simplified as follows:

$$V = \frac{Q_1 + Q_2}{2C_u} = \frac{C_u V_1 + C_u V_2}{2C_u} = \frac{V_1 + V_2}{2}, \tag{2.2}$$

where V_1 and V_2 are the voltages on the two capacitors before connecting them together, and V is the steady state voltage on them after. In other words, when two capacitors of equal value are connected together, the voltage on them will be the average of the voltages on each before they were connected together. The above logic

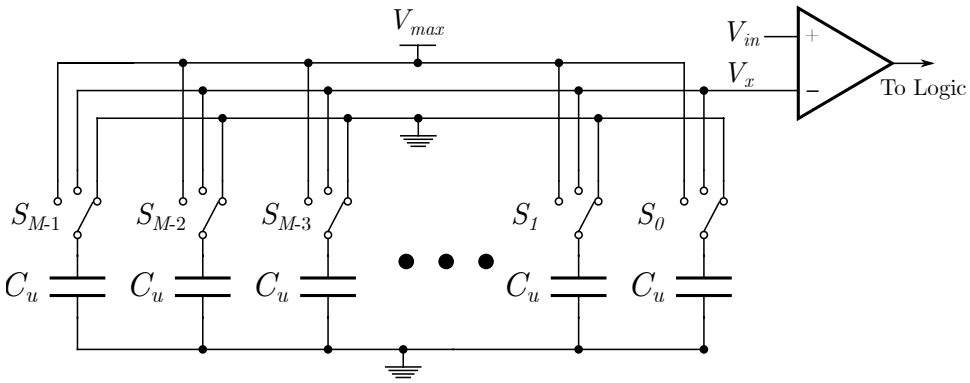


Figure 2.6: The capacitor and switch array proposed in [3] uses an array of M unit capacitors to generate the required voltages.

can be extended to N equal-sized capacitors being connected together in parallel:

$$V = \frac{\sum_{n=1}^{N} V_n}{N}. \tag{2.3}$$

In [3] an array of unit capacitors is proposed instead of a binary-weighted array. Each capacitor in the array has its bottom plate connected to ground, and its top plate can be switched between V_{max}, ground, and the positive input of the comparator, as shown in Figure 2.6. The observation is made that there is overhead in the typical binary weighted capacitor array, since at any step it is capable of generating any of the required output voltages at a given step by connecting the appropriate switches, but there are only two voltages that it actually needs to be able to generate at that step. Further, it is observed that at each step the voltage that needs to be generated is always the average of the voltage currently present on the array and a voltage that has previously been generated. Combining this observation with (2.3), the proposed circuit stores the required voltages on its unit capacitors and uses these to generate the next voltage on V_x.

Figure 2.7 illustrates the idea using the same example as the binary tree in Figure

2.2. First, half the capacitors are charge to V_{max} and half to ground. Then a subset of them are connected to V_x, where half of those connected were charged to V_{max} and half to ground, so by (2.3) they share their charge and all end up charged to $\frac{1}{2}V_{max}$. Since $V_{in} < V_x$, the comparator will output a 0, telling the logic to reduce V_x. It will connect the switches such that half those connected are storing $\frac{1}{2}V_{max}$ and half are storing 0 V, giving $\frac{1}{4}V_{max}$ by (2.3). Next, it generates $\frac{3}{8}V_{max}$ by connecting capacitors charged to $\frac{3}{8}V_{max}$ and $\frac{1}{2}V_{max}$. If instead it needed to generate $\frac{1}{8}V_{max}$ at that point, it would have connected the switches such that half the capacitors were at $\frac{1}{4}V_{max}$ and half were at 0 V. In this design, there need to be sufficient capacitors in the array that the voltages required to generate the next voltages in the binary tree can always be stored. Consequently, for the unit capacitor array charge sharing technique introduced in [3], the 8-bit ADC designed required 28 unit capacitors and a 10-bit ADC would require 64 unit capacitors.

The logic used to control the switches is not discussed in [3], but a reasonably simple switching scheme appears to have been used. In [10], the concept proposed in [3] was used to design a similar circuit that uses a state machine to control the switches. In addition, the number of required unit capacitors was reduced. For the 8-bit ADC designed in [10] the number was reduced from 28 to 15. This was made possible by observing that the generation of voltages in [3] was always performed by averaging two previously generated voltages together. By allowing the possibility of using more than two voltages that had previously been generated, the number of capacitors required to store those voltages decreases.

For example, there are many ways of generating 0.375 V. We could connect together two capacitors, one charged to 0.25 V and one charged to 0.5 V. Or four capacitors, two charged to 0.25 V and two charged to 0.5 V. Or two charged to 0

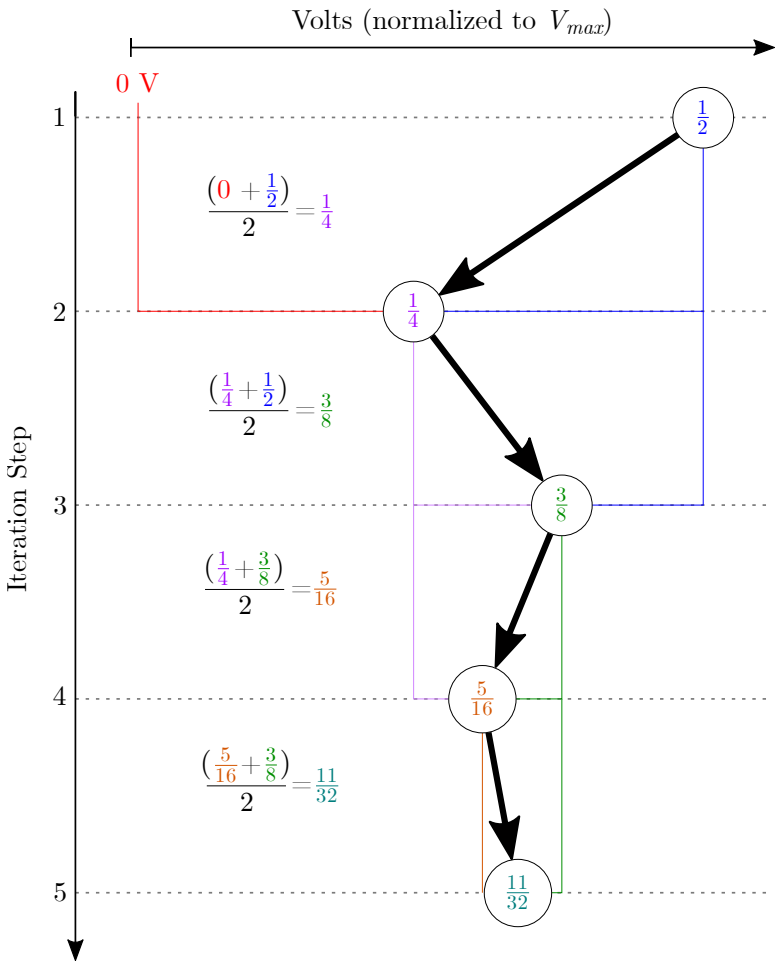


Figure 2.7: The way the charge-sharing unit capacitor array SAR ADC approaches the input voltage of 0.33 V by storing and averaging two voltages that have previously been generated on the array.

V, one charged to 0.5 V and one charged to 1 V, and so on. Deciding which combination to use will depend on which combination was used to generate the voltages before that (i.e. 0.25 V, and 0.5 V in the case of 0.375 V), and which voltages will be required to generate the ones after it (i.e. 0.3125 V and 0.4375 V). The reduction in the number of unit capacitors afforded by this added flexibility comes at the cost of increased digital complexity.

In [10], the determination of the switching scheme was in part computer-aided, but much of it was done manually, making it difficult to apply the concept to designs with different resolutions. In this work, a 10-bit ADC is designed. For 10 bits, there are 1023 voltages that need to be generated, or 512 unique paths through the binary decision tree (since there are 512 nodes in the final row of the tree and there is a unique path to each). This makes designing the tree manually infeasible. Therefore, a "backtracking" algorithm is presented that fully automates the determination of the switching scheme for this type of SAR ADC.

2.2 Backtracking Algorithms

Backtracking algorithms use recursion to search for valid solutions to complex problems. Backtracking is a useful technique when a solution can be built incrementally and at any given step it is relatively easy to check whether an extension to a partial solution can never lead to a full solution. At that point the search can "backtrack" by abandoning the extension and trying a different one [18, p.201].

The pseudo-code for a basic backtracking function is as follows:

```
// Explore extensions to a partial solution
Boolean: testSolution(partial_solution)

   // Check if we can immediately discard this partial solution
   If <partial_solution cannot lead to full solution>
```

```
      then return false

   // Check if this partial solution constitutes a full solution
   If <partial_solution is a full solution>
      then return true

   // Try different ways of extending the partial solution
   Loop <over all possible extensions to partial_solution>

      <Extend partial_solution>

      // Recursive call to testSolution
      If testSolution(partial_solution)
         then return true

      <Undo the extension to partial_solution>

   end Loop

// If the function gets here, it has looped through every possible
// extension, and none of them lead to full solutions
return false

end function
```

The `testSolution` function takes whatever data it needs as a parameter to keep track of the partial solution, and returns a Boolean that indicates whether that partial solution can be extended to lead to a full solution. The function first checks whether it can immediately determine that the partial solution either cannot lead to a full solution, or already is itself a full solution, in which cases it returns `false` or `true`, respectively. If neither of those is the case, it will loop through possible ways of extending that partial solution, and recursively call the `testSolution` function, passing in the extended partial solution. If the extension can in fact lead to a full solution, it immediately returns `true`. Otherwise, it reverts the extension and tries a different way of extending the partial solution. If it reaches the end of the loop and has not found an extension that leads to a full solution, it returns `false`.

A simple example of a problem that might be solved by a backtracking function is a maze, as shown in Figure 2.8. Here, each step (i.e. to the next grid space) in the

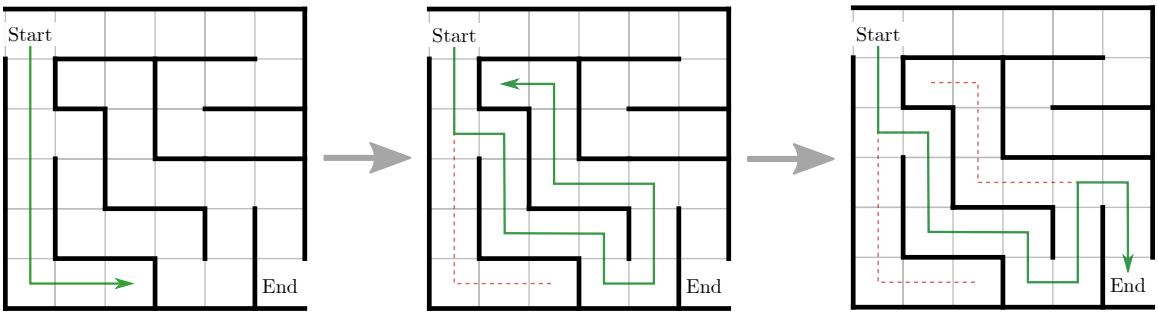


Figure 2.8: Solving a maze is a simple example of a problem that can be solved with backtracking. When the algorithm reaches a fork it tries one of the options. If it reaches a dead end it backs up and tries the next option. This repeats until it reaches the end or has tried every possible solution.

maze is a recursive call to the `testSolution` function. The extensions to the partial solution represent moving to the next space - if there are multiple options they will be looped through as necessary. The function can easily check if the current space is a dead end if all three directions (left, right, forward) are walled off. When it reaches a dead end it backs up until it reaches a space where it has another direction to try. This continues until it reaches the end.

2.3 Summary

In this chapter, SAR ADCs were introduced. The binary-weighted charge redistribution method is a common method for generating the reference voltage in SAR ADCs. A variant of this method, the unit capacitor charge-sharing method, was proposed in [3] and expanded on in [10]. This work expands further on these methods by using a backtracking algorithm to automate the generation of a state machine that controls the switches in the array. In the next chapter, the implementation of this algorithm and the state machine it generates will be discussed.

Chapter 3

Improved Charge Sharing SAR ADC Algorithm

In this chapter, the backtracking algorithm developed in Matlab that generates the SAR state machine logic is covered. Then, the state machine logic itself is discussed. Finally, we go through an example of how the logic converts the analog input to a digital output.

As discussed in Chapter 2, a unit capacitor array will be used to generate each required reference voltage in the binary tree[1]. A simplified schematic of this array is shown in Figure 3.1. Note that in contrast to the capacitor array in [3] (Figure 2.6), each capacitor has only a single switch connecting it to a common node, V_x. Then, two additional switches connect V_x to ground and V_{dd}. Arranging the array in this way allows us to reduce the number of outputs that the logic will require.

The SAR logic is implemented as a finite state machine. As mentioned in Section

[1]The reference voltages required for N bits are $n/2^N$, where n is every integer from 1 to $2^N - 1$, and the order in which they are generated follows a binary search, as explained in Figure 2.2.

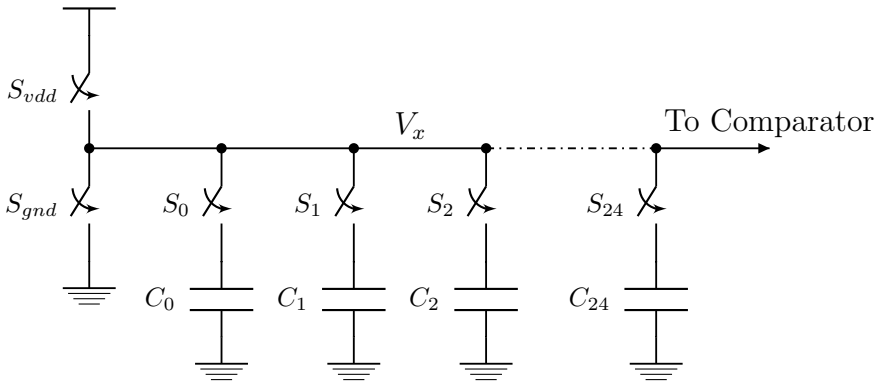


Figure 3.1: A simplified schematic of the capacitor array.

2.1, for a 10-bit ADC there are $2^{10} - 1 = 1023$ voltages that the capacitor array needs to be able to generate, so each one will require its own state. Because the final row in the binary tree has $2^9 = 512$ nodes (i.e. voltages to generate), there are 512 unique paths through the tree. Rather than trying to manually design the logic to generate the voltages in the tree, the design of the logic is automated by using a backtracking algorithm (Section 2.2).

It is possible that an analytic approach could be used to design the switching scheme, similar to [3]. However, by using more complex switching schemes that do not follow a regular pattern that could be captured by simple equations, we allow for the possibility of reducing complexity in other areas, such as the number of capacitors required. In this work, 25 capacitors are found to be required for 10 bits, versus 64 required by the analytical design in [3]. Another potential advantage is that more sophistication could be added to the algorithm in the future to account for expected performance in light of non-idealities in circuitry. Optimization techniques could then be applied to improve the performance of the ADC along different metrics. One could envision how such improvements and trade-offs could be incorporated into to the algorithm incrementally in a way that simple equations might have more trouble capturing.

The algorithm is given a number of unit capacitors that we wish to use in the array, and it tries to generate every voltage required in the binary tree by averaging together some of the voltages stored on those capacitors, essentially simulating the operation of the array. It then reports whether or not it was successful in generating every voltage, and if so it returns several arrays that tell us which switches to open and close at each step. These arrays are then passed to another function that uses them to generate most of the Verilog code that implements the state machine. The Matlab scripts that implement the backtracking algorithm and Verilog generation are given in Appendix A and Appendix B, respectively.

3.1 Implementation of the Backtracking Algorithm to Determine the State Machine Logic

In this implementation of the backtracking algorithm, the partial solution that is being tracked represents the voltage on each capacitor in the array at that step. The backtracking function will determine whether the partial solution can lead to a full solution by checking if any subset of the voltages present can be averaged together to generate the voltage required at that node in the binary tree.

3.1.1 Table of Ways to Average Previously Generated Voltages

It is therefore convenient to pre-compute a table that lists the voltages that can be averaged together to generate each voltage required on the tree. To avoid unnecessary computation, the table should only include voltages that could already be present on

a capacitor at each node - that is, the voltages that were generated by the parent of the node, and that node's parent, and so on. For example, when we are trying to generate 0.375, the only voltages that could already be present on the array are 0, 1, 0.5, and 0.25, so considering any other values would be a waste of time and memory. An example of a subset of the output array is shown in Table 3.1.

The function that generates this table takes in two values, N and M . N is the number of bits of the converter we are designing, so that the most precise numbers being generated will be integer multiples of $1/2^N$. M is the highest number of capacitors we will allow the logic to connect together at any given step. In theory, M should be as large as the total number of capacitors in the array. In this design, it was decided to use M = 6. For 10 bits it was found that increasing M from 6 to 8 offered no reduction in the number of capacitors, and the algorithm takes ∼55 times longer to run. It is possible that the number of capacitors could be further reduced if M were increased, but for the purposes of this design it was decided to stop here.

Although 6 is the *most* capacitors we will connect together at once, we still need to allow for connecting fewer together as well, so the table generated for M = 6 also includes rows for connecting 4 and 2 capacitors together at a time. The number of voltages to average together will always be an even number, since an odd number of negative powers of 2 averaged together will not generate a negative power of 2.

The function works by stepping through each row in the binary tree as discussed in Section 2.1. Then for each voltage in that row it generates a list of voltages along the "path" down the binary tree that it followed (plus 1 and 0) - i.e. the list of voltages that might already be present on the array. It then makes a list of all the combinations of those voltages of length M (the number of capacitors allowed to be connected). These combinations can have repeated values (e.g. a valid combination if the current

Table 3.1: Example of part of the table showing the values that might already be present on the array that can be averaged together to generate the required voltage for a given node.

Value to be Generated	**Values to Average Together**					
0.375	0.25	0	0	0	1	1
0.375	0.25	0.25	0.25	0.25	0.25	1
0.375	0.5	0.25	0.25	0.25	0	1
0.375	0.5	0.5	0.25	0	0	1
0.375	0.5	0.5	0.5	0.25	0.25	0.25
0.375	0.5	0.5	0.5	0.5	0.25	0
0.125	0.25	0.25	0.25	0	0	0
0.125	0.5	0.25	0	0	0	0
0.75	0.5	0	1	1	1	1
0.75	0.5	0.5	0.5	1	1	1
0.25	0.5	0	0	0	0	1
0.25	0.5	0.5	0.5	0	0	0
0.5	0	0	0	1	1	1

voltage is 0.25 and M is 4 could be [0.5, 0.5, 0.5, 0]), but no two combinations can be rearranged to be identical (e.g. [0.5, 0.5, 0.5, 0] and [0, 0.5, 0.5, 0.5] are not both generated). The generation of the list of combinations with repeated values was performed using an open-source script from Matlab Central called VChooseKR [21], which provides a Matlab wrapper around compiled C code for improved efficiency. Finally, the list of combinations is looped through and the rows that average to the current voltage are saved and passed to the output. The Matlab script is provided in Appendix A.

The rows in the generated table are grouped together based on the number of values to be averaged together, so the example shown in Table 3.1 shows all the possible combinations of the voltages on 6 capacitors that could be combined to generate 0.375, 0.125, 0.75, 0.25, and 0.5. Note that the order of the voltages to average together is irrelevant because of the way the backtracking algorithm uses these values (it goes through each voltage required one at a time and searches for it on the capacitor array). The full array for N = 10 and M = 6 has 19,489 rows.

3.1.2 The Backtracking Function

The recursive backtracking function takes in a number representing the current node, the number of bits of the converter we want to design, the desired number of capacitors on the array, the arrays that represent the decision trees to be filled in, the voltages currently on the capacitors, and the table that lists the voltages that can be averaged together to generate the voltage required on each node (as described above). The routine follows a modified version of the backtracking algorithm described in Section 2.2. The pseudo-code describing this function is as follows (the Matlab code is provided in Appendix A):

```
Boolean: fillCapacitorDecisionTree(current node, number of bits,
number of caps, tree arrays, table of possible connections)

   // Try different ways of extending the partial solution
   Loop 1: <Iterate through each possible connection decision>

      <Get the appropriate row from the table of possible connections>

      Loop 2: <Iterate through each capacitor we might want to charge or discharge>

         <Get the current voltages on the capacitors (passed into the function)>

         // Check if this partial solution can lead to a full solution
         Loop 3: <Iterate through each required voltage in the current row of
         possible connections>

            <Check if the required voltage is available on the capacitors>

            If <the current required voltage is available on the capacitors>
               <Save the index of the capacitor the voltage was found on>
               <Update that capacitor to the voltage we're trying to generate on
                  this node of the tree>

            Else If <the current required voltage is a 1 or a 0 and the current
            capacitor we might want to charge or discharge has not already
            been used>
               <Save the index of that capacitor>
               <Update that capacitor to the voltage we're trying to generate
                  on this node of the tree>

            Else <break out of Loop 3 and go to the next value in Loop 2>

         end Loop 3

         If <every voltage needed was found>

            // Check if this partial solution constitutes a "full" solution
            If <this node is a leaf>
               <Update the trees with the appropriate values found in LOOP 3>
               Return true

            Else
               // Recursive call to left child
               <Call fillCapacitorDecisionTree() left child of this node>

               If <the call to fillCapacitorDecisionTree() left child returned true>

                  // Recursive call to right child
                  <Call fillCapacitorDecisionTree() on the right child of this node>

                  If <the call to fillCapacitorDecisionTree() on the next node
```

```
                down on the right returned true>

                   <Update the state trees with the values returned to this
                   node from the recursive calls>
                   <Update the state trees with the values determined the last
                   time Loop 3 succeeded in finding all the required voltages>
                   // The values to update are: which capacitor to pre-charge or
                   // discharge at each node (if any), which capacitors to
                   // connect together at each node, and what the voltage on
                   // each capacitor is at each node

                   Return true, and the updated trees

      end Loop 2

   end Loop 1

end function
```

The structure of this function is modified slightly from that presented in Section 2.2, but each general operation is still performed. Loop 1 and Loop 2 iterate through two different ways of updating the partial solution. Loop 3 checks whether the updated partial solution is valid and can lead to a full solution. If the updated partial solution is valid, the function then checks if it can return `true` immediately. If it cannot return `true` immediately, it makes recursive calls to the same function. If the recursive calls return `true` then it can also return `true`. Otherwise, it continues through Loop 1 and Loop 2 and tries updating the partial solution differently.

Because the function here is trying to build a tree, there are two significant differences from the basic backtracking algorithm discussed in Section 2.2. First, the function makes two recursive calls instead of one - first to check if the voltages now present on the capacitors can generate the voltages required by the left child (and its children), and then to check if they can generate the voltages required by the right child (and its children). If either of those returns `false`, that partial solution is abandoned and the loops continue. The other difference is that the function cannot on its own determine whether a full solution has been reached. Instead, it only returns

`true` without making more recursive calls if the voltage on the current node has been generated and the node is a leaf. Once *all* the leaves return `true`, all their parents and will return `true` up the tree until the root node, at which point it returns `true`, indicating that a full solution has been determined.

To try different ways of updating the partial solution, Loop 1 iterates through every row in the table discussed above that would generate the voltage required on the current node. Loop 3 can then check if the current row in the table can lead to a full solution by searching the capacitors for each voltage required. When a required voltage is found, that capacitor's index is saved and its voltage updated to the voltage being generated. If all the required voltages are found, the function can pass the updated voltages on the capacitors to the recursive function calls.

In Loop 3, when a required voltage is not found on the capacitors, Loop 2 provides a second way to update the partial solution. As shown in Figure 3.1, the capacitor array is arranged such that while no comparisons are happening or voltages being averaged together, any of the capacitors can be connected to ground or V_{dd}. This means that if having 1 V or 0 V on the array is required to generate the new voltage but is not already available on the array, we can charge or discharge one or more capacitors in advance to make it work. Loop 2 iterates through a set of capacitors we might want to charge or discharge. This way, if inside Loop 3 one of the voltages required to use in the average is 1 V or 0 V, but it is not found on the array, the capacitor designated by Loop 2 can be used. For each capacitor we would want to allow to be charged or discharged in this way, another loop would need to be added. Therefore it was decided to only allow a single capacitor to be charge or discharged at a given time. In addition only a subset of the available capacitors are looped through as options for charging or discharging in the implementation used here (see Section 3.1.3).

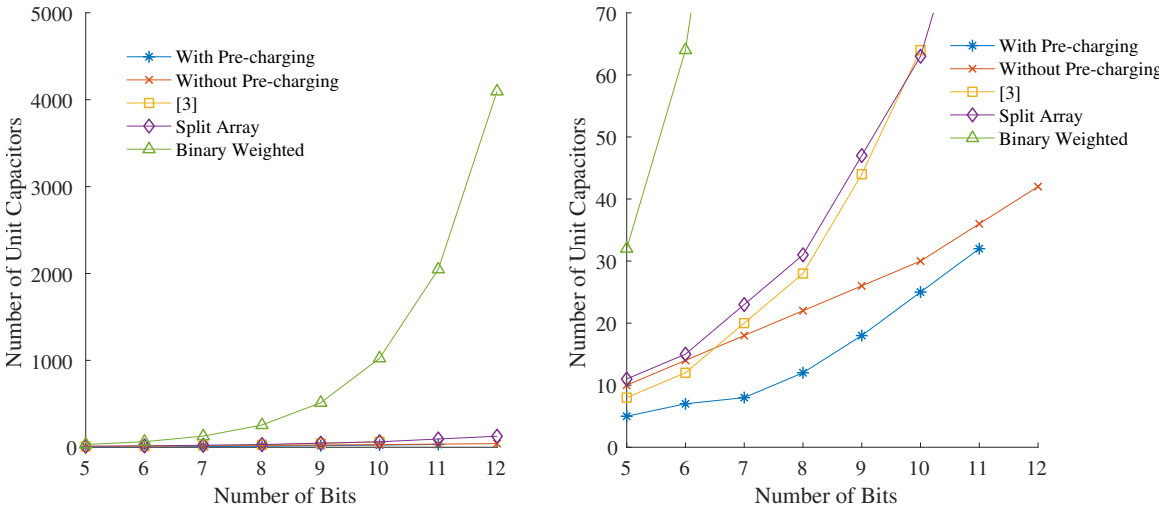


Figure 3.2: The number of unit capacitors required for a SAR ADC determined from the algorithm above with and without pre-charging/discharging compared with the number required in [3], and a split-array SAR [4], and a typical binary weighted SAR. The plot on the right shows a zoomed version of the one on the left.

Without Loop 2, the minimum number of capacitors required found by the function was 30. With Loop 2, the minimum number found was 25. It is possible that the number of required unit capacitors could be reduced further if more capacitors were allowed to charge or discharge by Loop 2.

Table 3.2 lists the minimum number of capacitors required for a SAR ADC at various resolutions determined the above algorithm with and without allowing for capacitors to be charged or discharged in between comparisons. These values are then compared to the numbers determined in [3], the number required for a typical binary weighted SAR ADC, and a split-array SAR, which is a variation on the binary weighted architecture used to reduce the number of capacitors required [4, p. 980]. These are also plotted in Figure 3.2. We can see that the algorithm here requires the fewest capacitors compared to the other approaches. It also has the gentlest slope, requiring the fewest capacitors added for each extra bit.

Table 3.2: The number of unit capacitors required for a SAR ADC determined from the proposed algorithm with and without pre-charging/discharging, compared with the number required in [3], a split-array SAR [4], and a typical binary weighted SAR ADC.

Number of Bits	**Number of Unit Capacitors Required**				
	This Work		Other Techniques		
	With Pre-Charging/ Discharging	Without Pre-Charging/ Discharging	[3]	Split-Array SAR	Binary Weighted SAR
5	5	10	8	11	32
6	7	14	12	15	64
7	8	18	20	23	128
8	12	22	28	31	256
9	18	26	44	47	512
10	25	30	64	63	1024
11	32	36	-	95	2048
12		42	-	127	4096

Note that here the number of unit capacitors is being compared, but not necessarily the layout area. The capacitor size required to offset parasitics varies between different architectures and a detailed analysis to determine this minimum size has not yet been performed for the proposed architecture. A more detailed analysis of mismatch and parasitics inherent in this architecture will be required.

In addition, because the number of states required in the state machine is $2^N - 1$, it will double with each added bit. The layout area of the logic will then correspondingly increase.

However, requiring fewer unit capacitors does have the advantage of reduced layout complexity. For analog designers and layout engineers, there is a clear benefit to having to layout and minimize mismatch for 25 capacitors vs 1024 capacitors. Additionally, much of the layout complexity in this design is offset to digital circuitry whose layout can be generated and optimized using automatic tools. In addition, the size and power consumption of digital circuits can scale much better than analog components with decreasing technology nodes [22], so a promising trend is shown here.

3.1.3 Computation Time

Table 3.3 lists the time and number of recursive calls for the algorithm to successfully build the solution trees with the minimum number of capacitors for which it succeeds for 8-13 bits. The algorithm used for the results in the table does not allow for capacitors to charge or discharge between comparisons (i.e. Loop 2 is omitted - in the code in Appendix A, the lines to remove are noted). These results were measured with Matlab R2017b on a computer with an Intel Xeon CPU E5-2697 v3 Processor @ 2.60 GHz and 512 GB of RAM @ 2155 MHz. The code was interpreted rather than compiled (except for the VChooseKR function, as discussed in Section 3.1.1).

Compiling the code is one way the speed might be improved.

Table 3.3: Computation time and number of recursive calls to compute the solution tree for 8-13 bits with the minimum number of capacitors for which the algorithm succeeds. The version of the backtracking algorithm that does not allow pre-charging or discharging capacitors is used.

Number of Bits	Number of capacitors allowed to be connected at once	Number of Unit Capacitors	Rows in the Table of Voltages that can be Averaged Together	Time to Compute (s)	Number of Recursive Calls
8	6	22	4329	0.76	1441
9	6	26	9283	1.99	10554
10	6	30	19489	8.65	62725
11	6	36	40301	43.9	410867
12	6	42	82445	1680	2509553
13	6	48	167399	28276	21457534

These results are plotted against the number of bits in Figure 3.3. Since these are plotted on semi-log axes, it is clear that the number of recursive calls and the number of rows in the possible connections table are each approximately exponential. The number of calls to the recursive function increases by a factor of approximately 6 to 9 for each added bit, and the number of rows in the table of possible connections approximately doubles with each added bit. The pattern for the time to run the algorithm is harder to discern. We would expect it to be approximately the number of calls to the recursive function times the amount of time spent in the function on each call, so it is possible the time in the function varies greatly for different numbers of bits. In addition, the run time may be inconsistent because the state of the computer it is run on could be different for each call (i.e. we don't know what other processes are running). In any case, we can approximate that the run-time

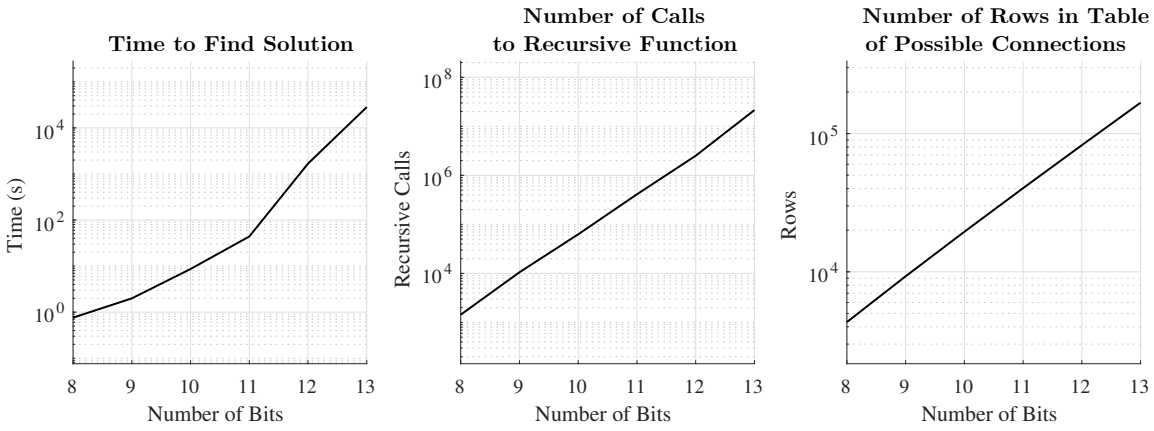


Figure 3.3: Computation time, number of recursive calls, and number of rows in table of possible connections vs number of bits, for the version of the backtracking algorithm that does not allow pre-charging or discharging of capacitors between comparisons.

increases by roughly an order of magnitude for each bit added. So while 13 bits took approximately 8 hours to compute, we could expect 14 bits to take over 3 days, and 15 bits to take about a month.

This is of course only the time for the algorithm to complete when it succeeds - if it is looped to reduce the number of capacitors required this increases the time required. Alternatively, multiple instances of the algorithm with different input parameters could be run in parallel. Backtracking itself does not lend itself easily to parallel computation because every recursive call is dependent on the current state of the partial solution. However it may be possible to parallelize certain parts of the recursive function in order to improve its performance. This will be left as future work on the topic.

With the algorithm modified to allow for charging or discharging a capacitor in between comparisons, an extra loop is added to the function which greatly increases the run time. Due to a minor error when originally writing the code, the number of capacitors available for charging or discharging was set to equal the number of

elements in the current row of the possible connections (i.e. 2, 4, or 6 depending on the row) instead of a fixed value. The solution found with this result is the one used in the fabricated chip, and was still a valid solution despite the minor error. With this setting, the algorithm took 79.9 hours to find a solution, and made about 990 million calls to the recursive function. With the number of capacitors allowed to charge or discharge fixed at 2, the algorithm did not find a solution, and completed after 39.3 hours and 750 million recursive calls. Increasing the number of capacitors allowed to charge or discharge to 3, the algorithm found a solution but it took 214 hours and 3.24 billion recursive calls. In this case, increasing the number of capacitors allowed to charge or discharge from 2 to 3 resulted in an increase in processing time by a factor of 5.4.

3.2 SAR State Machine

The SAR state machine logic was written in Verilog HDL, and its structure was adapted from that designed in [10]. Most of the Verilog code was automatically generated using a Matlab script that takes the outputs of the algorithm discussed in Section 3.1 and uses them to determine which switches to open and close in each state. The script that generates the Verilog code is provided in Appendix A.

For each conversion, the state machine goes through 21 states - two for each of the 10 bits plus an initialization state. The states change on the rising edges of the clock. The comparator in the ADC resets when the clock is high and performs its comparisons on the falling edges of the clock. The logic and the comparator operate on opposite clock edges so that the comparator's output is available to be sampled

when the states change[2].

Figure 3.4 shows a simplified functional flowchart for the state machine logic. The conversion for each bit is split into two states. Each `b` state checks whether the comparator's output was high or low, and checks a register that keeps track of what the last voltage generated was. Together, these two values tell us what node in the tree we are on now, and therefore what voltage to generate next. The `b` state then sets the capacitor switches required to generate that voltage (as determined by the algorithm from Section 3.1). The `a` states check whether a capacitor needs to be charged to V_{dd} or discharged to ground in order to generate the values in the corresponding `b` state that follows, and set the switches accordingly (again, as determined by the algorithm). If no capacitors need to be charged to V_{dd} or discharged to ground, the `a` state does nothing. After each `b` state, the comparator compares the sampled analog input to the generated reference voltage and outputs a 1 or a 0 depending on which one was higher. This result is stored so that the `a` and `b` states can use it to inform what voltage to generate next. After bit 0 (the LSB) is determined, the sample clock will go low and the state machine will wait in the `INIT` state for it to go high again, at which point the next conversion will begin again with bit 9 (the MSB).

The structure of the Verilog code that defines the state machine consists mainly of two `always` blocks. One `always` block triggers when the `state` register changes, and describes the combinational logic that determines the next state and current outputs. The other `always` block triggers on the falling edge of the clock (or the falling edge of the reset signal). It sets the state register based on the values set by the combinational logic, and saves the outputs so they can persist for two clock cycles

[2]Note that in addition to operating on opposite edges, the comparator should operate on a clock that is either delayed or half the speed of the logic so that the logic has time to sample the decision before the comparator is reset. This point was missed during the design and caused the fabricated chip to not function properly, as will be discussed further in Chapter 6.

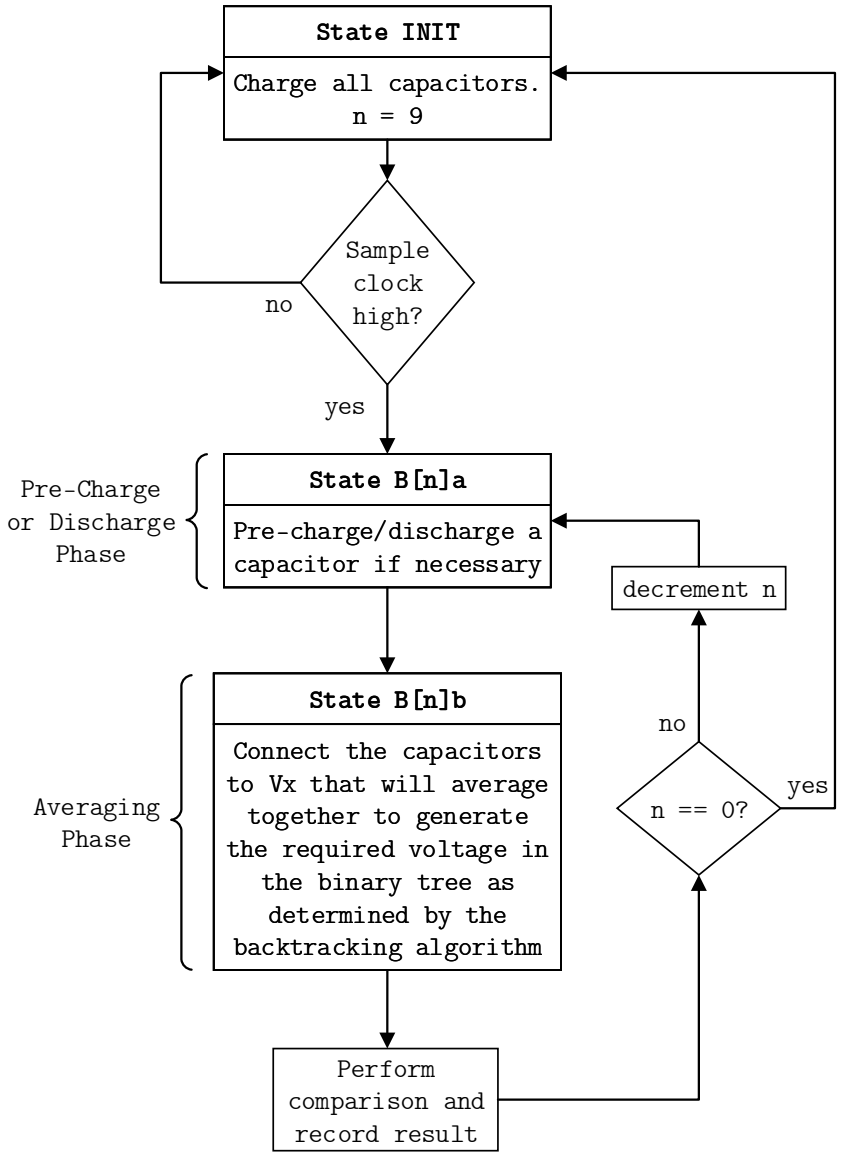


Figure 3.4: Simplified functional flowchart for the SAR logic state machine.

in order for the next state and any external circuitry to have time to read them.

The code for the `RESET` (not shown in Figure 3.4) `INIT` states, and the states for the conversion of bits 9 through 7 (where bit 9 is the MSB) in the combinational logic block is as follows:

```
always @ (state) begin

   // DEFAULT VALUES
   cap_switches  = cap_switches_reg; // Retain the previous value
   array_gnd     = 1'b0; // Active high switch that connects Varray to ground
   array_vdd     = 1'b1; // Active low switch that connects Varray to Vdd
   next_state    = INIT; // The default next state is INIT
   prev_state    = prev_state_reg; // The prev_state register keeps track of
                                   // what the last voltage generated was.
                                   // Here, we hold it so the states can check it
   data_clk      = 1'b0; // Every b state will set this to 1, and every a state
                         // will keep it at 0. On its falling edges, the data
                         // is ready at the output.
   get_comp      = 1'b0; // Set to 1 in states where the comparator output is
                         // ready to be captured as the output bit

   case (state)

    RESET:
    begin
      next_state   = INIT;
      // Connect all the capacitors to Vx
      cap_switches = 25'b1111111111111111111111111;
      // Discharge all the capacitors to ground
      array_gnd    = 1'b1; // Active high
      array_vdd    = 1'b1; // Active low
    end

    INIT:
    begin
      next_state    = B9a;
      // When we initialize, first discharge all the caps
      // that won't be charged in the next state
      cap_switches  = 25'b1111111111111000000000000;
      array_gnd     = 1'b1;
      array_vdd     = 1'b1;
    end

    B9a:
    begin
      next_state    = B9b;
      // Charge half the caps to 1
      cap_switches  = 25'b0000000000000111111111111;
      array_gnd     = 1'b0;
```

```
  array_vdd     = 1'b0;
end

B9b:
begin
  next_state    = B8a;
  data_clk      = 1'b1;
  get_comp      = 1'b1;
  // Must create 1/2*Vdd
  // Connect capacitors to Vx as determined by the algorithm
  cap_switches  = 25'b1110000000000111000000000;
end

B8a:
begin
  next_state = B8b;
end

B8b:
begin
  next_state = B7a;
  data_clk = 1'b1;
  get_comp = 1'b1;

  if (data_out == 1'b0) begin
    // Must create 1/4*Vdd
    cap_switches = 25'b1001111000000000100000000;
    prev_state = B8_1;
  end
  else begin
    // Must create 3/4*Vdd
    cap_switches = 25'b1110000000000000111000000;
    prev_state = B8_3;
  end
end

B7a:
begin
  next_state = B7b;
end

B7b:
begin
  next_state = B6a;
  data_clk = 1'b1;
  get_comp = 1'b1;

  if (data_out == 1'b0 && prev_state_reg == B8_1) begin
    // Must create 1/8*Vdd
    cap_switches = 25'b1001100111000000000000000;
    prev_state = B7_1;
```

```
    end
    else if (data_out == 1'b1 && prev_state_reg == B8_1) begin
      // Must create 3/8*Vdd
      cap_switches = 25'b1000000111000000011000000;
      prev_state = B7_3;
    end
    else if (data_out == 1'b0 && prev_state_reg == B8_3) begin
      // Must create 5/8*Vdd
      cap_switches = 25'b1001100000000000000111000;
      prev_state = B7_5;
    end
    else begin
      // Must create 7/8*Vdd
      cap_switches = 25'b1110000000000000000111000;
      prev_state = B7_7;
    end
  end
```

At the start of this `always` block, default values are set for each register that might be changed by the state logic. If the logic is in a state where one of these values is not explicitly set, its default value will be used.

The `cap_switches` register is a 25-bit output where each bit controls one of $S_{0...24}$ in Figure 3.1 (the active high switches that each connect one of the capacitors in the array to V_x). The `array_vdd` and `array_ground` outputs control the S_{vdd} (active low) and S_{gnd} (active high) switches, respectively. In each state, the values that these are set to in order to generate the required voltages was determined by the outputs of the algorithm discussed in Section 3.1.

There are twenty-two states that are set using the "state" register and chosen using a `case` statement[3]. Each state used in the `case` block is represented in the logic using a 5-bit parameter. The `RESET` state discharges all the capacitors to ground, and is only entered at start-up and when the reset signal is activated.

To help clarify the operation of the state machine, Figure 3.5 shows a flowchart for the code above. First, the `INIT` and `B9a` states discharge thirteen of the capacitors

[3]Not including those used for comparator calibration, which adds 5 extra states that set the calibration signal high when an external signal is triggered. See Section 4.3.2 and Appendix B.2.

to ground and charge the other twelve to V_{dd}, respectively. As the flow chart shows, the state machine then implements the binary tree as discussed in Chapter 2. State `B9b` connects six of the capacitors (as determined by the backtracking algorithm) to V_x to generate $\frac{1}{2}V_{dd}$, and sets `get_comp` to 1 to indicate that the comparator output should be recorded. The comparator output is then the result for bit 9, and is stored in `data_out`. If `data_out` is 0, state `B8b` will generate $\frac{1}{4}V_{dd}$; if it is 1, state `B8b` will generate $\frac{3}{4}V_{dd}$. The `prev_state` register then records which voltage was generated in state `B8b` - where `B8_1` indicates $\frac{1}{4}V_{dd}$ and `B8_3` indicates $\frac{3}{4}V_{dd}$. State `B7b` then checks both `data_out` and `prev_state_reg` (in which the value of `prev_state` is stored on the rising clock edges) to determine which voltage it should generate.

Note that states `B8a` and `B7a` do nothing except set the default values in the registers and the `next_state` to `B8b` and `B7b`, respectively. State `B6a`, on the other hand, sets the outputs under certain conditions as follows:

```
B6a:
begin
  next_state = B6b;
  if (data_out == 1'b1 && prev_state_reg == B7_1) begin
    // Must create 3/16*Vdd
    // Discharging C24
    array_gnd = 1'b1;
    cap_switches = 25'b0000000000000000000000001;
  end
  else if (data_out == 1'b0 && prev_state_reg == B7_7) begin
    // Must create 13/16*Vdd
    // Charging C21
    array_vdd = 1'b0;
    cap_switches = 25'b0000000000000000000001000;
  end
end
```

This is because the algorithm discussed in Section 3.1 determined that in order to generate $\frac{3}{16}V_{dd}$ in state `B6b`, state `B6a` should first discharge C_{24} to ground. Similarly it determined that in order to generate $\frac{13}{16}V_{dd}$ in state `B6b`, state `B6a` should first charge

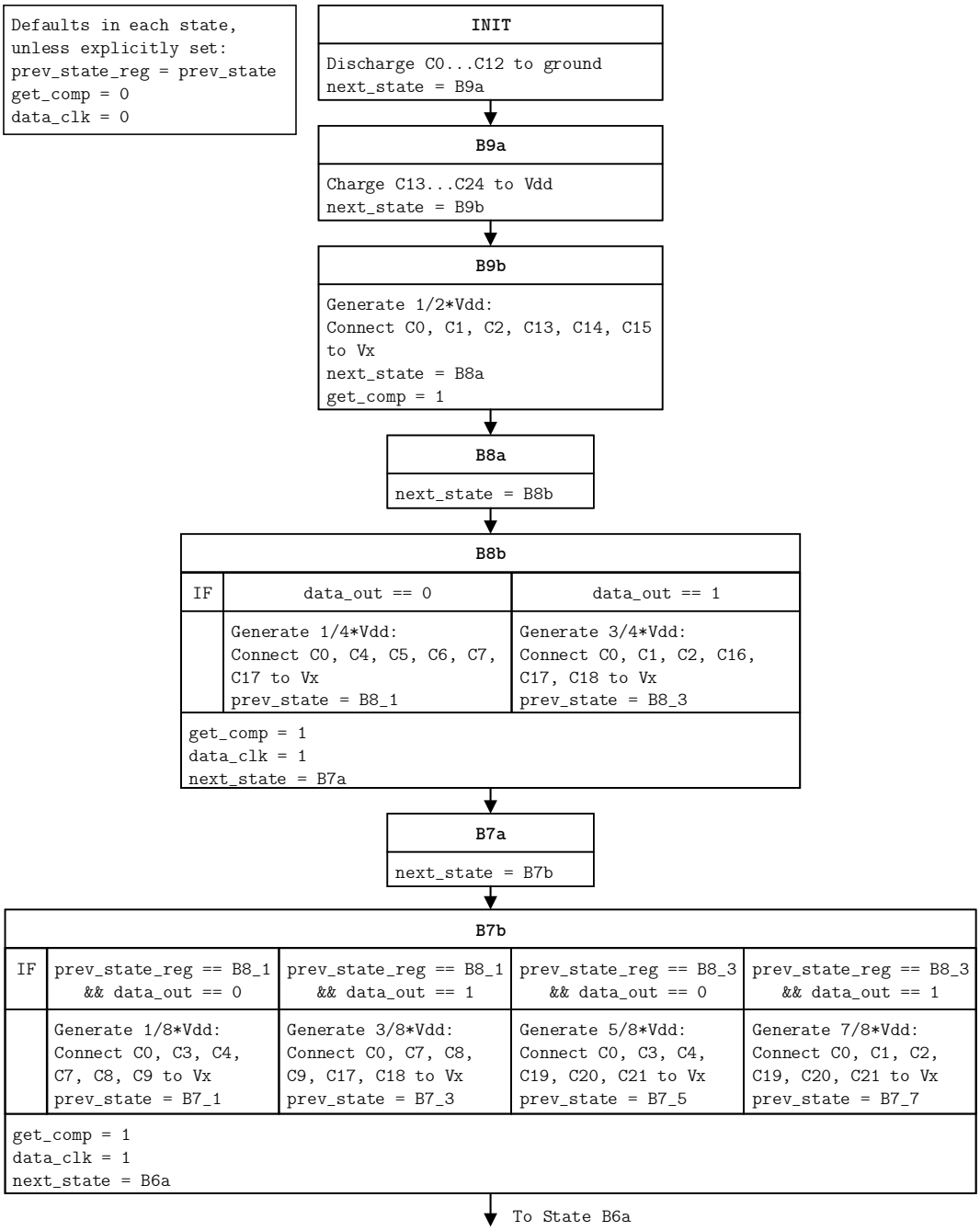


Figure 3.5: Flowchart for the SAR logic state machine for the `RESET` and `INIT` states and states `B9a` through `B7b`.

C_{21} to V_{dd}. States `B5a` through `B0a` also have conditions where they will charge or discharge one of the capacitors so the `b` state can generate its required voltages.

The sequential `always` block updates the state registers and is triggered by the clock and the reset signal, as follows:

```
always @ (negedge clk or negedge rst_n)
begin
  if (!rst_n) begin // Reset triggered the always block
    state           <= RESET;
    prev_state_reg  <= NONE;
    data_out        <= 1'b0;
    data_clk_out    <= 1'b0;
  end
  else begin // Clock triggered the always block

    // Hold these values
    cap_switches_reg   <= cap_switches;
    prev_state_reg     <= prev_state;
    data_clk_out       <= data_clk;

    // Only move to next state if sampling clock is high
    // (or performing comparator calibration)
    if (sh_clk || CAL_ext)
      state           <= next_state;
    else
      state           <= INIT;

    // Capture the comparator output
    if (get_comp)
      data_out        <= comp;
  end
end
```

If the reset signal caused the block to be triggered, then the outputs are set to 0 and the state set to `RESET`. Otherwise, a falling clock edge triggered the block. In this case, the values stored in `cap_switches` and `prev_state` are stored in intermediate registers so they can be retained for two clock cycles. `cap_switches` is retained so that the voltage on V_x is kept constant long enough for the comparator to see it at the next rising clock edge. `prev_state` is retained so that the next state can read it when determining what it should do. `data_clock` is sent to an output register that is

not reset when the state changes. `data_clock` is set to 1 in the `b` states and 0 in the `a` states, and the falling edges of `data_clock_out` indicate that `data_out` is a valid bit. `sh_clock` is an input signal that comes from the sampling clock, which controls the S/H. When it is high it means a conversion is ongoing, so we want the state machine to act normally and therefore the value in `next_state` is passed on to `state`. Otherwise the state is held at `INIT`. The `CAL_ext` signal is an external signal that controls when the comparator is calibrated. The calibration happens when `sh_clk` is low, but uses the state machine, so it must also allow the state to change. Finally, the comparator output is stored when the `get_comp` signal is high. As shown above, this is set during the `b` states.

One problem that arises from the state machine as it was described above is that the `cap_switches`, `array_vdd`, and `array_gnd` control lines do not reset in between states. This means that some switches will be turning on while others are turning off and vice versa, which could cause undesired charging or discharging of the capacitors. One option to prevent this would be to add another clock cycle between the time when the comparison happens and when the next state is determined. However, this is undesirable as "droop" on the capacitors caused by charge leaking through the switches causes errors in the conversions. To avoid this, an extra conditional is added to the always block that determines the next state as follows:

```
always @ (state or comp or compP)
begin

  if (clk && (!comp || !compP) && sh_clk) begin
    cap_switches   = 25'b0000000000000000000000000;
    array_vdd      = 1'b1;
    array_gnd      = 1'b0;
  end
  else begin

  // SET DEFAULTS
  ...
```

```
case(state)
...
```

Since the capacitors' switches only need to be held open until a comparison is made, we can close the switches as soon as one of the comparator's outputs goes to 0[4]. Both of the comparator's outputs are added to the trigger list of the `always` block so that the switches can be set when they change. If either of them is low and both the logic clock and sample clock are high, the comparison has happened and all of the switches can be opened.

3.3 Example Conversion

Here, we will go through what happens on the capacitor array during an ideal conversion in order to generate the required reference voltage. For this example, we will arbitrarily assume an input voltage of 0.45 V.

Figure 3.6 shows how the array voltage (this is the "reference" voltage, which is fed into the negative input of the comparator) iteratively approaches the input voltage by performing a binary search. Figure 3.7 shows this zoomed in on the second half of the conversion. The details of the voltage generation are shown in Tables 3.4 and 3.5 (the table is split into two pages for legibility). Each row in the table labeled Cn represents the voltage on the nth capacitor at each clock step, with time moving left to right. Then the voltage on the array, the input voltage, and the result of the comparison at that clock step (such that the comparison gives a 1 if $V_{in} > V_{array}$, and a 0 otherwise) are shown at the top. The outlined values in each column show which capacitors are connected to the array at that clock step, either to generate the voltage required or to charge or discharge the capacitor.

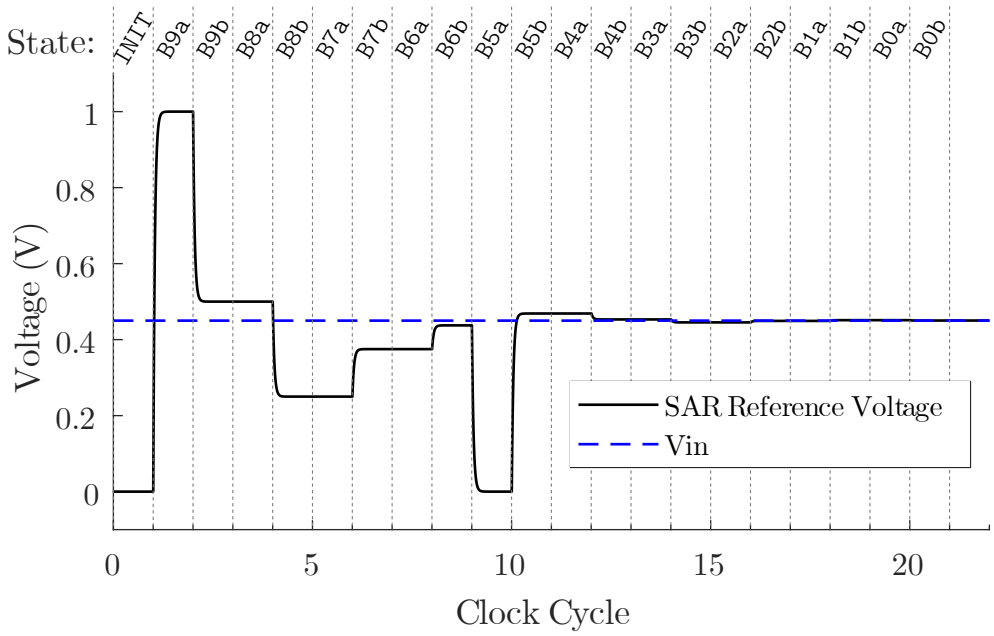


Figure 3.6: The voltage on the output of the capacitor array during an ideal conversion, for an input voltage of 0.45 V.

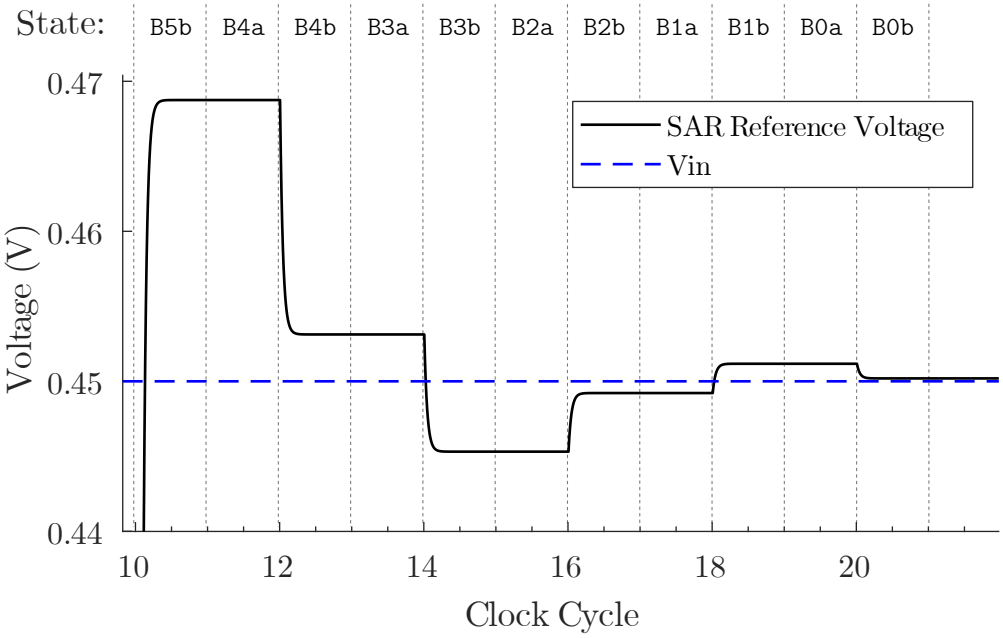


Figure 3.7: The voltage on the output of the capacitor array during an ideal conversion, for an input voltage of 0.45 V, zoomed in on the second half of the conversion.

Table 3.4: The voltage on every capacitor during conversion for an input voltage of 0.45 V (clock cycles 1-13).

Clock Step	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
State	**INIT**	**B9a**	**B9b**	**B8a**	**B8b**	**B7a**	**B7b**	**B6a**	**B6b**	**B5a**	**B5b**	**B4a**	**B4b**
Array Voltage	**0**	**1**	**0.5**	**0.5**	**0.25**	**0.25**	**0.375**	**0.375**	**0.4375**	**0**	**0.46875**	**0.46875**	**0.453125**
Input Voltage	**0.45 V**												
Comparison			0		1		1		1		0		0
C0	**0**	0	**0.5**	0.5	**0.25**	0.25	0.375	0.375	**0.4375**	0.4375	**0.46875**	0.46875	**0.453125**
C1	**0**	0	**0.5**	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C2	**0**	0	**0.5**	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C3	**0**	0	0	0	**0.25**	0.25	0.25	0.25	**0.4375**	0.4375	0.4375	0.4375	**0.453125**
C4	**0**	0	0	0	**0.25**	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
C5	**0**	0	0	0	**0.25**	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
C6	**0**	0	0	0	**0.25**	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
C7	**0**	0	0	0	0	0	**0.375**	0.375	0.375	0.375	**0.46875**	0.46875	**0.453125**
C8	**0**	0	0	0	0	0	**0.375**	0.375	0.375	0.375	0.375	0.375	0.375
C9	**0**	0	0	0	0	0	**0.375**	0.375	0.375	0.375	0.375	0.375	0.375
C10	**0**	0	0	0	0	0	0	0	**0.4375**	0.4375	0.4375	0.4375	**0.453125**
C11	**0**	0	0	0	0	0	0	0	**0.4375**	0.4375	0.4375	0.4375	**0.453125**
C12	**0**	0	0	0	0	0	0	0	0	0	**0.46875**	0.46875	**0.453125**
C13	x	**1**	**0.5**	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C14	x	**1**	**0.5**	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C15	x	**1**	**0.5**	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C16	x	**1**	1	1	**0.25**	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
C17	x	**1**	1	1	1	1	**0.375**	0.375	0.375	0.375	0.375	0.375	0.375
C18	x	**1**	1	1	1	1	**0.375**	0.375	0.375	0.375	0.375	0.375	0.375
C19	x	**1**	1	1	1	1	1	1	**0.4375**	0.4375	0.4375	0.4375	0.4375
C20	x	**1**	1	1	1	1	1	1	**0.4375**	0.4375	0.4375	0.4375	0.4375
C21	x	**1**	1	1	1	1	1	1	1	1	**0.46875**	0.46875	0.46875
C22	x	**1**	1	1	1	1	1	1	1	1	**0.46875**	0.46875	0.46875
C23	x	**1**	1	1	1	1	1	1	1	1	1	1	1
C24	x	**1**	1	1	1	1	1	1	1	**0**	**0.46875**	0.46875	0.46875

The switches connecting the capacitors together are controlled by the state machine described in Section 3.2, whose outputs were determined by the algorithm described in Section 3.1. The states from the state machine are labeled in both the plots and the tables.

The first two steps of the conversion (the `INIT` and `B9a` states) bring the voltages

[4]Note that the comparator output is active low, so the default "off" state for its outputs is 1. When the comparator makes a decision one of the two outputs will go to 0, depending on which input was higher.

Table 3.5: The voltage on every capacitor during conversion for an input voltage of 0.45 V (clock cycles 14-21).

Clock Step	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
State	**B3a**	**B3b**	**B2a**	**B2b**	**B1a**	**B1b**	**B0a**	**B0b**
Array Voltage	**0.453125**	**0.4453125**	**0.4453125**	**0.44921875**	**0.44921875**	**0.451171875**	**0.451171875**	**0.4501953125**
Input Voltage	**0.45 V**							
Comparison		1		1		0		0
C0	0.453125	**0.4453125**	0.4453125	**0.44921875**	0.44921875	**0.451171875**	0.451171875	**0.450195313**
C1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C2	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C3	0.453125	**0.4453125**	0.4453125	**0.44921875**	0.44921875	**0.451171875**	0.451171875	**0.4501953125**
C4	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
C5	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
C6	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
C7	0.453125	**0.4453125**	0.4453125	**0.44921875**	0.44921875	**0.451171875**	0.451171875	**0.4501953125**
C8	0.375	**0.4453125**	0.4453125	0.4453125	0.4453125	**0.451171875**	0.451171875	0.451171875
C9	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375
C10	0.453125	0.453125	0.453125	**0.44921875**	0.44921875	0.44921875	0.44921875	**0.4501953125**
C11	0.453125	0.453125	0.453125	**0.44921875**	0.44921875	0.44921875	0.44921875	**0.4501953125**
C12	0.453125	0.453125	0.453125	**0.44921875**	0.44921875	0.44921875	0.44921875	**0.4501953125**
C13	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C14	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C15	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
C16	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
C17	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375
C18	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375
C19	0.4375	0.4375	0.4375	0.4375	0.4375	0.4375	0.4375	0.4375
C20	0.4375	0.4375	0.4375	0.4375	0.4375	0.4375	0.4375	0.4375
C21	0.46875	**0.4453125**	0.4453125	0.4453125	0.4453125	**0.451171875**	0.451171875	0.451171875
C22	0.46875	**0.4453125**	0.4453125	0.4453125	0.4453125	0.4453125	0.4453125	0.4453125
C23	1	1	1	1	1	1	1	1
C24	0.46875	0.46875	0.46875	0.46875	0.46875	**0.451171875**	0.451171875	0.451171875

on the capacitors to a known state, first connecting C0 - C12 to ground, and then connecting C13 - C24 to V_{dd} (1 V in this case). On clock step 3 (state `B9b`), C0 - C2 and C13 - C15 are connected to the array. As explained in Section 2.1.2, their charge is evenly distributed, so the voltages on them are averaged, resulting in 0.5 V on the array. Since $V_{in} < V_{array}$ the comparator outputs a 0, which tells the state machine to decrease the voltage. Therefore, on clock step 5 (state `B8b`) it connects C0 (0.5 V), C3 - C6 (all at 0 V), and C16 (1 V) to the array in order to generate 0.25 V. Since now $V_{in} > V_{array}$, the comparator outputs a 1, which tells the state machine to

increase the voltage.

There are two "clock steps" for each voltage generation step. These are the `a` and `b` states as discussed in Section 3.2. This is because at any given step, the state machine may require a capacitor to be charged or discharged in order to generate the required voltage. We can see this at clock step 10 (state `B5a`), when only C24 is connected to the array and discharged so that at the next clock step (state `B5b`) it can be connected to the array to generate the required voltage. Since the capacitors can only be charged and discharged through the "array" node, nothing else should be connected to that node while the charging or discharging is happening (e.g. only C24 is connected to the "array" node at clock step 10). An alternative to this would be to give each capacitor its own switches to V_{dd} and ground, as in [3] and [10]. However, this would require two extra switches and two extra state machine outputs for each capacitor, thus some speed is traded off for reduced complexity[5].

This process continues for ten comparisons (since this is a 10-bit conversion). The digital output is then the results of each comparison, concatenated to form a binary number, where the first comparison gives the MSB and the last comparison gives the LSB. The output value here is

$$0111001100_2 \rightarrow 460_{10},$$

$$\frac{460}{2^{10}-1} = 0.4496579,$$

which is what we expect given the input voltage of 0.45 V.

[5] Although [10] still required the extra clock step between each comparison due to the nature of the state machine logic, and it is likely it would have been required here as well.

3.4 Summary

In this chapter, the SAR ADC's state machine machine logic and the backtracking algorithm that automatically generate it were discussed. Finally, a detailed ideal 10 bit conversion was shown. Now that the logical operation of the ADC is understood, the next chapter will discuss the circuit-level design and layout of each major block.

Chapter 4

Circuit Level Design and Layout

This chapter will outline the circuit-level design of each component of the ADC. The layout of the comparator and capacitor array will also be discussed, as well as the placement and routing (P&R) of the SAR logic using Cadence Encounter.

The ADC was implemented in a 130 nm CMOS process owned by Global Foundries (GF). The process uses a p-type substrate and has eight metal layers. All the metal layers are copper with the exception of layers 6 and 8, which are aluminum. The process offers dual MIM capacitors which are made using layer 6 and two thin silicon nitride and aluminum layers to increase capacitance density (see Section 4.4.1).

All circuit-level simulations were performed using Cadence Virtuoso 6.1.6-64b. For detailed circuit-level simulations, the Spectre simulator was used. For simulations involving circuit blocks and functional Verilog blocks, the Analog Mixed-Signal (AMS) simulator was used. For simulations including the circuit-level digital block, the UltraSim simulator was used.

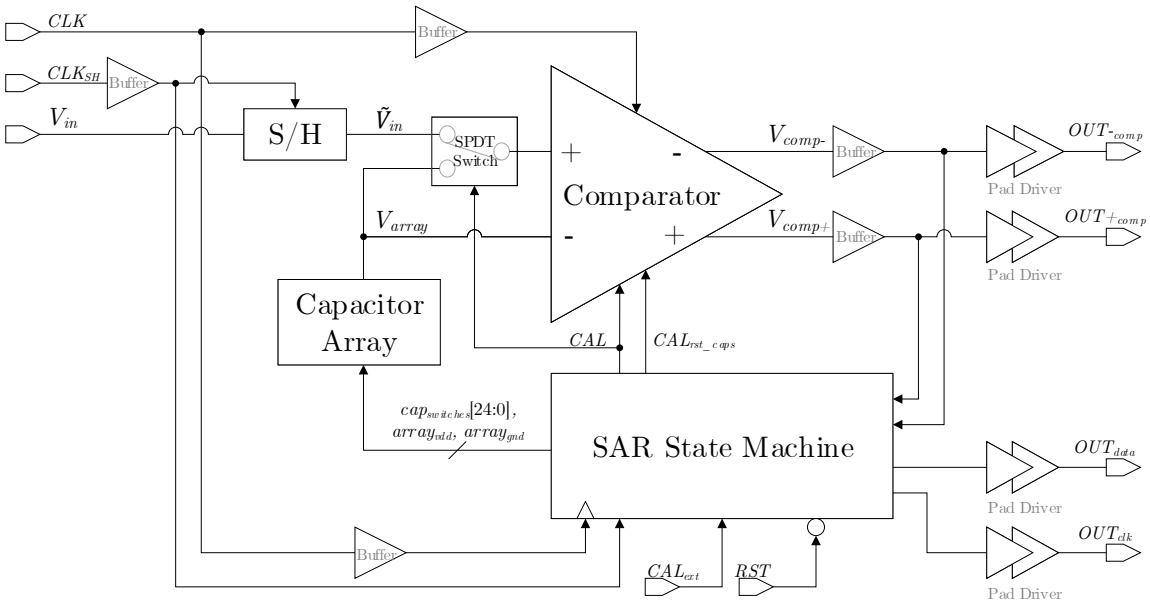


Figure 4.1: Full block diagram of the SAR ADC.

4.1 Overall Design

As described in Chapter 2, the ADC consists of three major components: a clocked comparator, a capacitor and switch array, and a logic block. Other components designed include the S/H, output drivers and logic buffers. Figure 4.1 shows a block diagram of the ADC.

The layout of the whole chip is shown below in Figure 4.2. The combined layout area of the SAR logic, capacitor array, comparator, S/H is $\sim$ 0.458 mm^2. The capacitor array and comparator were placed along the axis of symmetry of the die, which can help reduce mismatch due to thermal and stress gradients [23][24]. A second comparator was also added to the chip with all inputs and outputs directly connected to bond pads. Having this allows us to debug if there are any major performance issues.

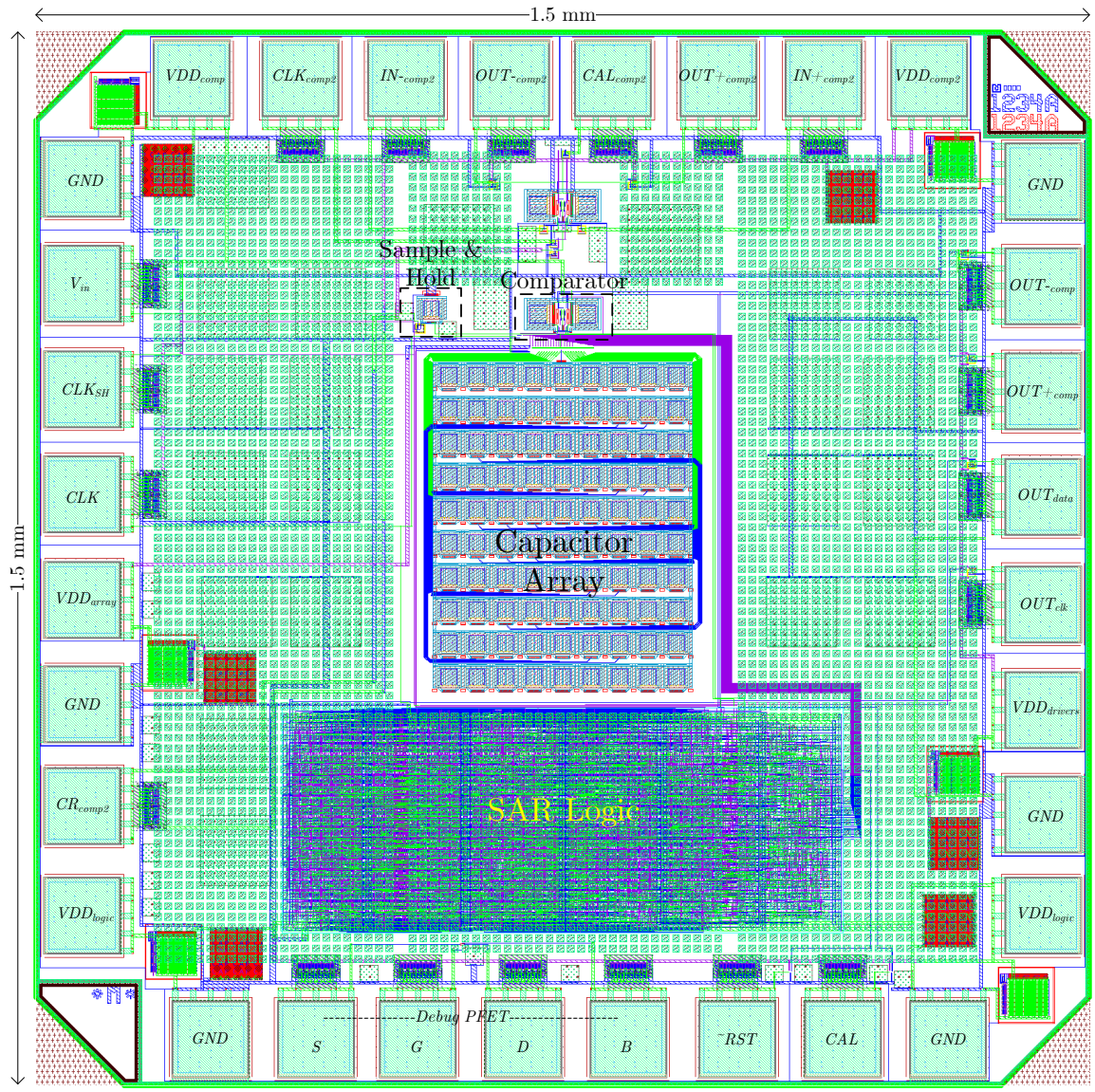


Figure 4.2: Full SAR ADC chip layout. Each block of the ADC (sample and hold, comparator, capacitor array, SAR logic), is labeled, as well as pads used for the main operation of the ADC. A second copy of the comparator is also added near the top for further testing. The chip was fabricated in a 130 nm process owned by Global Foundries. The combined layout area of the S/H, comparator, capacitor array, and SAR logic is ~ 0.458 mm^2.

The frequency of the sampling clock is $f_s = 10$ kHz. As discussed in Chapter 3, for each conversion the state machine requires 21 falling edges of the logic clock while the sampling clock is high (50 µs with a 50% duty cycle). A 410 kHz clock has a period of ∼2.44 µs, so 20.5 clock periods occur during a 50 µs interval, allowing for 21 falling clock edges. Therefore, $f_{clk} = 410$ kHz was chosen as the logic clock frequency.

4.2 Sample and Hold Design

The purpose of the S/H is to sample the analog input to the ADC and hold it constant for the comparator during the conversion. A simple version of the S/H is a switch and a capacitor, as seen in Figure 4.3. During the "track" phase (when CLK_{SH} is high), the switch is closed and the signal simply passes through. During the "hold" phase (when CLK_{SH} is low) the switch is open and the capacitor holds a constant voltage. An example is shown in Figure 4.4. The conversions take place during the "hold" phase. There exist many variations on the S/H that boost its performance, such as clock bootstrapping, adding dummy switches, and feedback amplifiers [10][25]. However, at a sampling frequency of 10 kHz none of these additions were found to be necessary, so the simple S/H in Figure 4.3 was chosen.

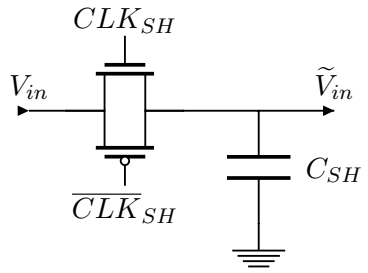


Figure 4.3: Sample and hold schematic.

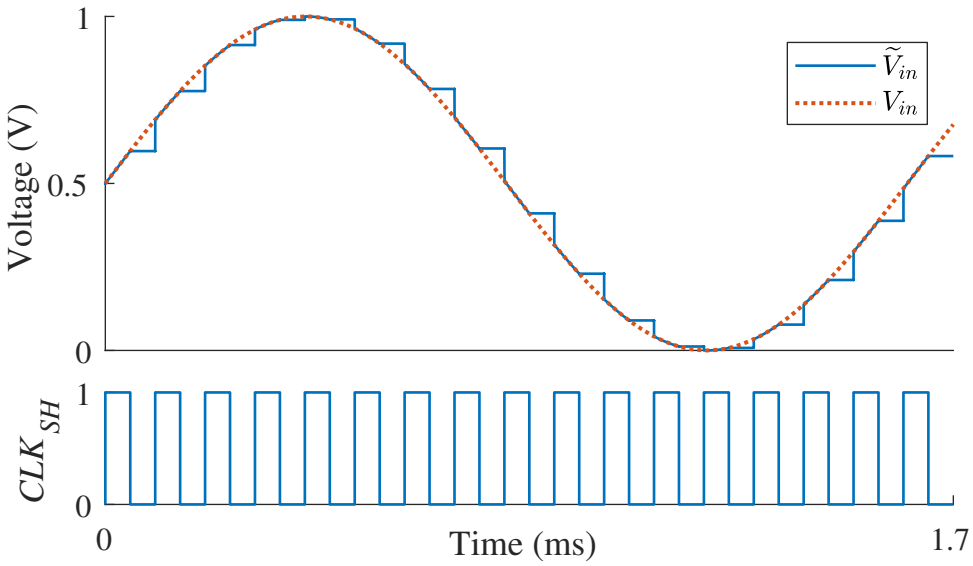


Figure 4.4: Sample and Hold example waveform. During the "track" phase (clock is high), the sampled output follows the input voltage. During the "hold" phase (clock is low), the sampled output is held constant.

The size of the transistors in the transmission gate switch and the capacitor are shown in Tables 4.1 and 4.2, respectively. The capacitor was chosen to be 2.72 pF. These were chosen so that the capacitor would have no trouble tracking the input in the "track" and leakage back through the switch (resulting in "droop") would be kept low in the "hold" phase. For simulation results verifying this, see Section 5.1.

Note that charge injection caused by the switch in the S/H was not considered during the design process. Charge injection is when the charge that creates the channel in a transistor travels out of the drain and source when the transistor is turned off. This can cause unwanted changes in the voltage that is held on the capacitor. Since charge injection is a high frequency event outside the signal band, it could be filtered out. However, it should still be considered in future designs.

Table 4.1: Sample and Hold switch transistor dimensions.

	Width (nm)	Length (nm)
NMOS	280	120
PMOS	280	120

Table 4.2: Sample and hold capacitor dimensions and nominal capacitance.

	Width (µm)	Length (µm)	Nominal Capacitance
C_{SH}	21.55	21.55	2.72 pF

4.3 Comparator Design

The inputs of the comparator are connected to the S/H and the capacitor array output as shown in Figure 4.1. The comparator's input capacitance adds to the total capacitance of the array and therefore will have an effect on the voltages generated. Because of this, one requirement of the comparator is that its input capacitance be kept low. Other requirements include sufficient decision speed, input offset voltage calibration, and low or no static power consumption.

The design of the comparator with bulk tuning was based on the design in [26].

4.3.1 Comparator Without Calibration

The schematic of the clocked comparator is shown in Figure 4.5. The comparator consists of two stages: an input gain stage (left), and a positive-feedback latch stage (right).

The operation of the comparator is as follows. When $\overline{CLK}$ is high, M_5 is off and M_3 and M_4 discharge the output of the first stage, V_{mid+} and V_{mid-}, to ground. This

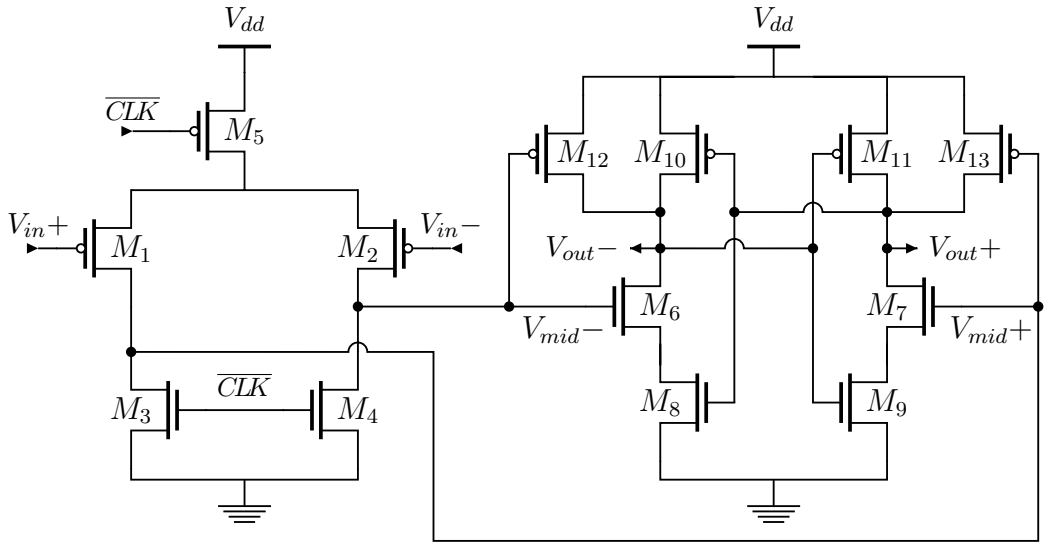


Figure 4.5: The clocked comparator circuit without calibration components added. On the left is the gain stage, and on the right is the positive-feedback latch stage [26].

then turns M_6 and M_7 off, and M_{12} and M_{13} charge V_{out+} and V_{out-} to V_{dd}. Since M_5, M_6, and M_7 are all off, the comparator draws no current when $\overline{CLK}$ is high.

When $\overline{CLK}$ switches from high to low, M_5 turns on and M_3 and M_4 turn off. Current will then flow from M_5 through M_1 and M_2, and V_{mid+} and V_{mid-} will start to increase. V_{mid+} and V_{mid-} increasing will cause M_{12} and M_{13} to turn off and M_6 and M_7 to turn on. Since one of the inputs is higher than the other, one of the cross-coupled inverters formed by M_8, M_{10}, M_9, and M_{11} will turn on faster than the other, causing its output to decrease. Since each inverter's output is tied to the other's input a positive feedback loop is formed, i.e. one of their outputs decreasing will cause the other one's output to increase, reinforcing the decision until one is at V_{dd} and the other is at ground.

As an example, Figure 4.6 shows the voltages in the comparator during a comparison with $V_{in}+ = 600$ mV and $V_{in}- = 400$ mV. Since $V_{in}- < V_{in}+$, M_2 has more drain current than M_1 and therefore $V_{mid}-$ increases faster than $V_{mid}+$. This

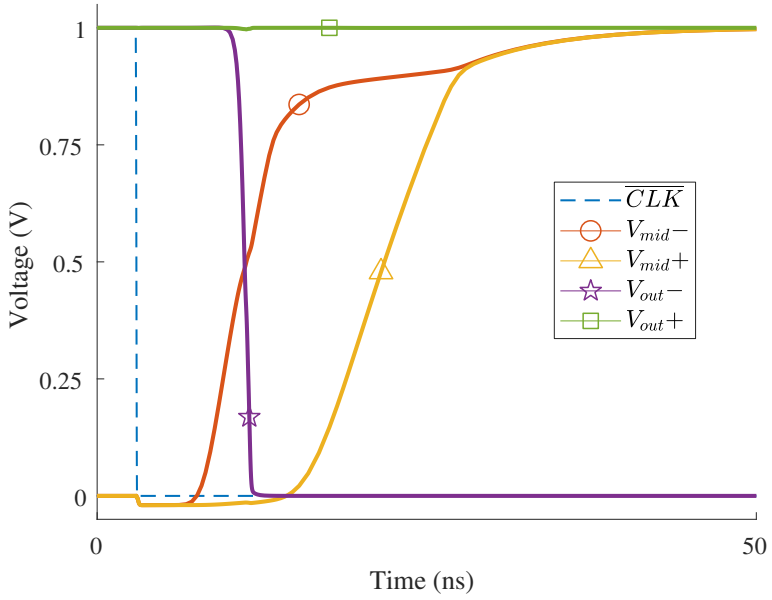


Figure 4.6: Signals within the comparator during a comparison with $V_{in}+ = 600\,\text{mV}$ and $V_{in}- = 400\,\text{mV}$.

in turn causes $V_{out}-$ to decrease much faster than $V_{out}+$. The positive feedback of the cross-coupled inverters quickly reinforces this, forcing $V_{out}-$ to ground and $V_{out}+$ to V_{dd}, which reflects the fact that $V_{in}+$ was higher than $V_{in}-$.

Because the comparator only draws current when making comparisons and has no DC path to from V_{dd} to ground, it consumes no static power (ignoring leakage currents).

4.3.2 Calibration Method and Design

The comparator's calibration scheme is also based on [26]. The method uses a discrete feedback loop to tune the bulk voltages, and therefore the threshold voltages, of the two input transistors, M_1 and M_2, in order to reduce input-referred voltage offset. The calibration is performed over 6 clock cycles during the S/H's "track" phase (i.e. while conversions are not happening), and is refreshed at regular intervals. A block

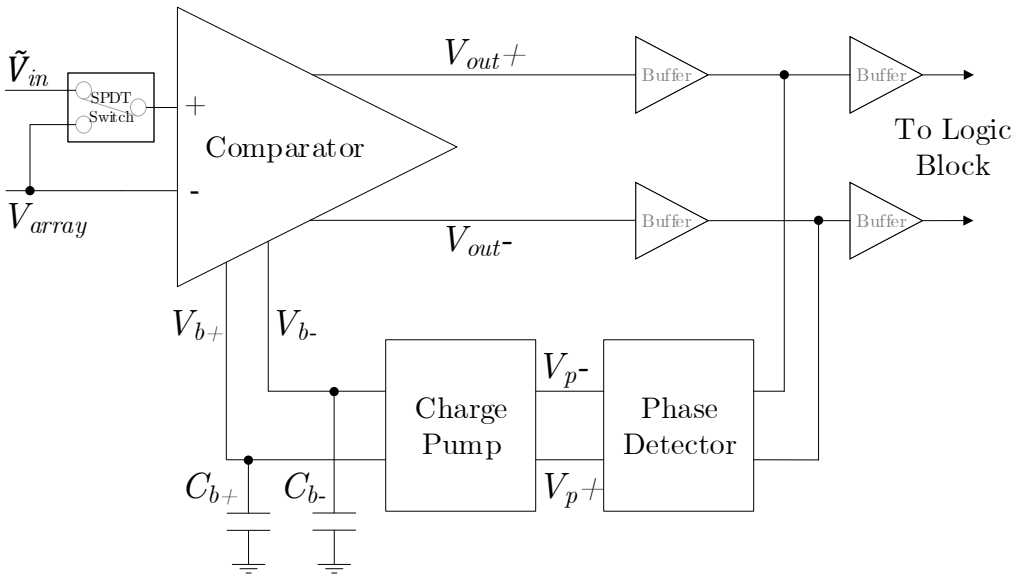


Figure 4.7: Block diagram of the calibration feedback loop [26].

diagram of the calibration feedback loop is shown in Figure 4.7. The components added are the phase detector, charge pump, input single-pole double-throw (SPDT) switch, two capacitors, and several voltage buffers.

To make the comparator compatible with the calibration loop, several modifications are made to its schematic from Figure 4.5, as shown in Figure 4.9. First, inputs are added to the bulks of M_1 and M_2 in order to tune their bulk voltages. M_1 and M_2 are implemented in separate n-wells so their bulks are isolated. Second, four switches ($S1$ - $S4$) are added to de-couple the cross-coupled inverters when the CAL control signal from the SAR logic is high. Note that $S1$ and $S2$ are transmission gates that are open when CAL is high, and $S3$ and $S4$ are PMOS switches that are closed when CAL is high. Finally, an SPDT switch is added which connects the positive and negative inputs to the same voltage (in this case, 0.5 V) when CAL is high. Figure

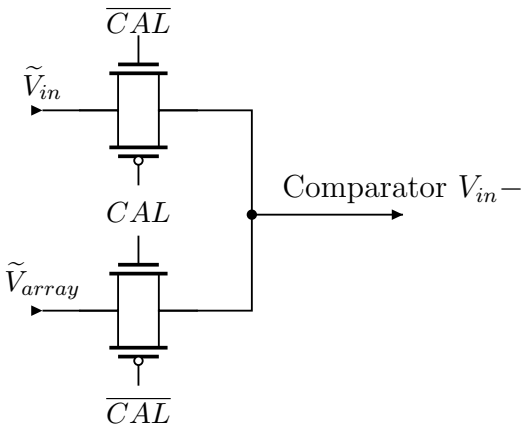


Figure 4.8: Schematic of the SPDT switch.

4.8 shows a schematic of the SPDT switch (the transmission is sized the same as the one in the S/H, as listed in Table 4.1).

The state machine discussed in Section 3.2 is also modified to supply the CAL and capacitor reset signals (which reset V_{b+} and V_{b-} to 1 V at the start of calibration) to the comparator when the external calibration signal is high. Five calibration states are added to the state machine, and state `B9a` is modified. If the calibration signal is high, state `B9a` will set `next_state` to `CAL1`, and send the signal to charge C_{b+} and C_{b-} to V_{dd}. State `B9a` will also charge half the capacitors to V_{dd} as usual; `INIT` will already have discharged the other half to ground. Then the five calibration states `CAL1` to `CAL5` hold 24 of the capacitors' switches closed to generate 0.5 V on the array. See Appendix B.2 for the modifications made to the state machine code.

When CAL is high the inverters are no longer cross-coupled, so both outputs will fall from V_{dd} to ground when $\overline{CLK}$ goes low. The SPDT switch causes both inputs to see the same voltage, so the outputs should ideally fall at the exact same time. However, due to process variations, there will be some mismatch between the two

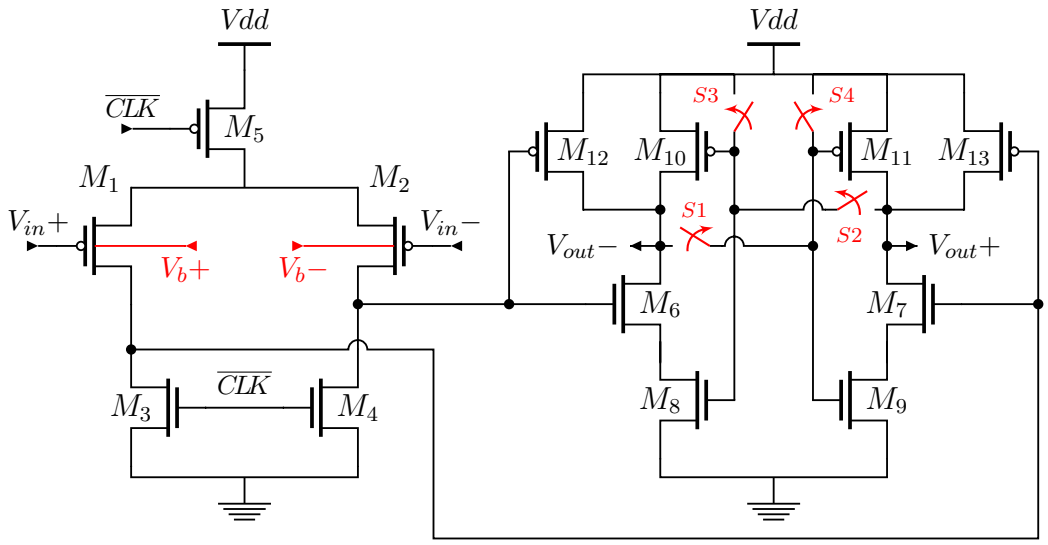


Figure 4.9: The clocked comparator circuit with switches and inputs required for calibration added [26]. The arrow direction on the switches indicates their state when the calibration control signal, CAL, is high.

branches (e.g. threshold voltage, capacitance, etc.) which will cause one output to fall faster than the other. This timing difference represents an offset voltage for the comparator decision during normal operation. Therefore, if the timing difference is reduced, so is the offset voltage.

In order to reduce this timing difference, the outputs are fed through the calibration feedback loop, which tunes the bulk voltages to compensate for changes in threshold voltage. Example signals through the calibration loop are shown in Figure 4.10, taken from a Monte Carlo simulation that adds mismatch to the devices. Figure 4.11 zooms in on the first and last steps around the loop to show how the timing difference is reduced.

The basic operation of the loop is as follows. The output signals from the comparator are fed into a phase detector (after each being passed through a buffer, which helps isolate the calibration loop from the regular operation of the comparator). The

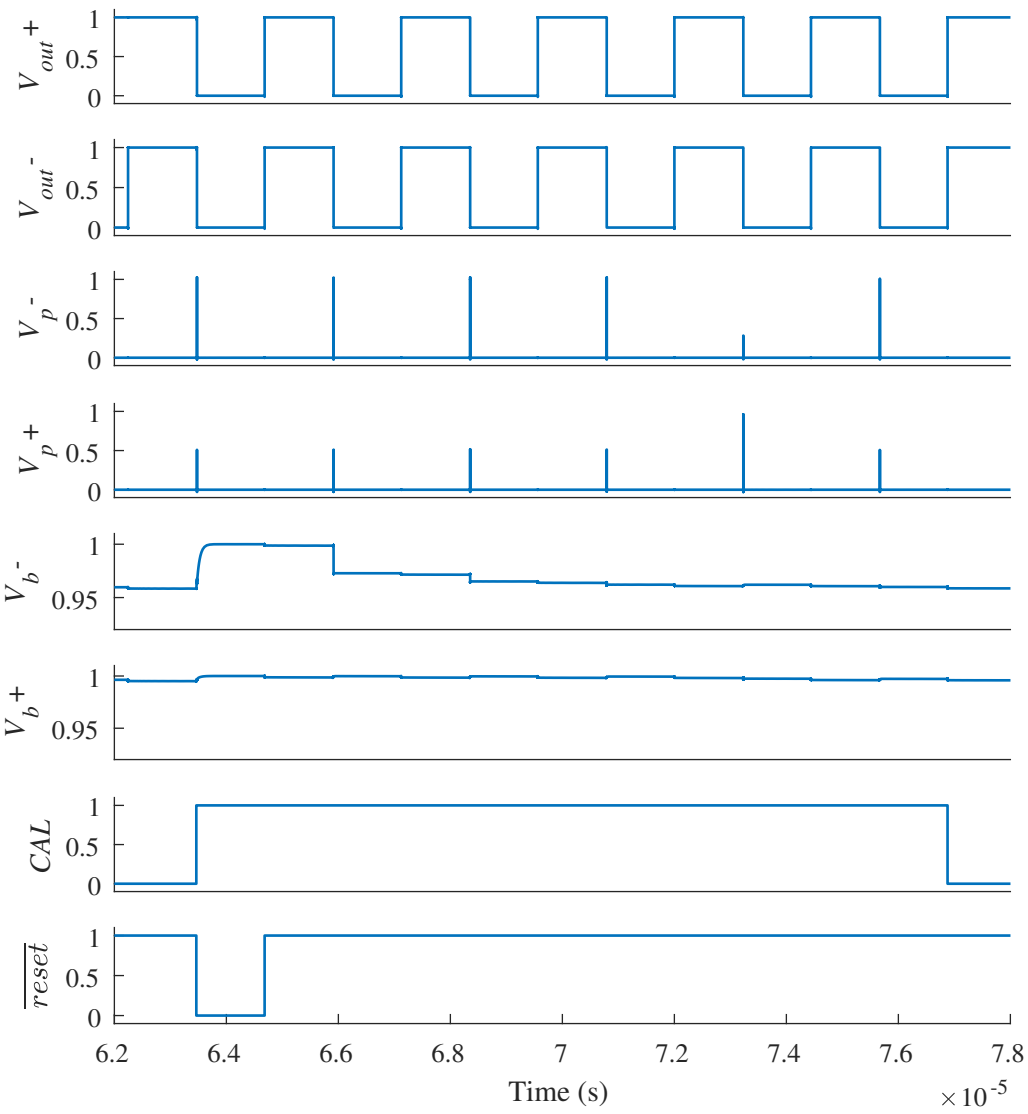


Figure 4.10: Signals within the comparator calibration feedback loop during the calibration cycle of a single point in a Monte Carlo run, for inputs at 0.5 V

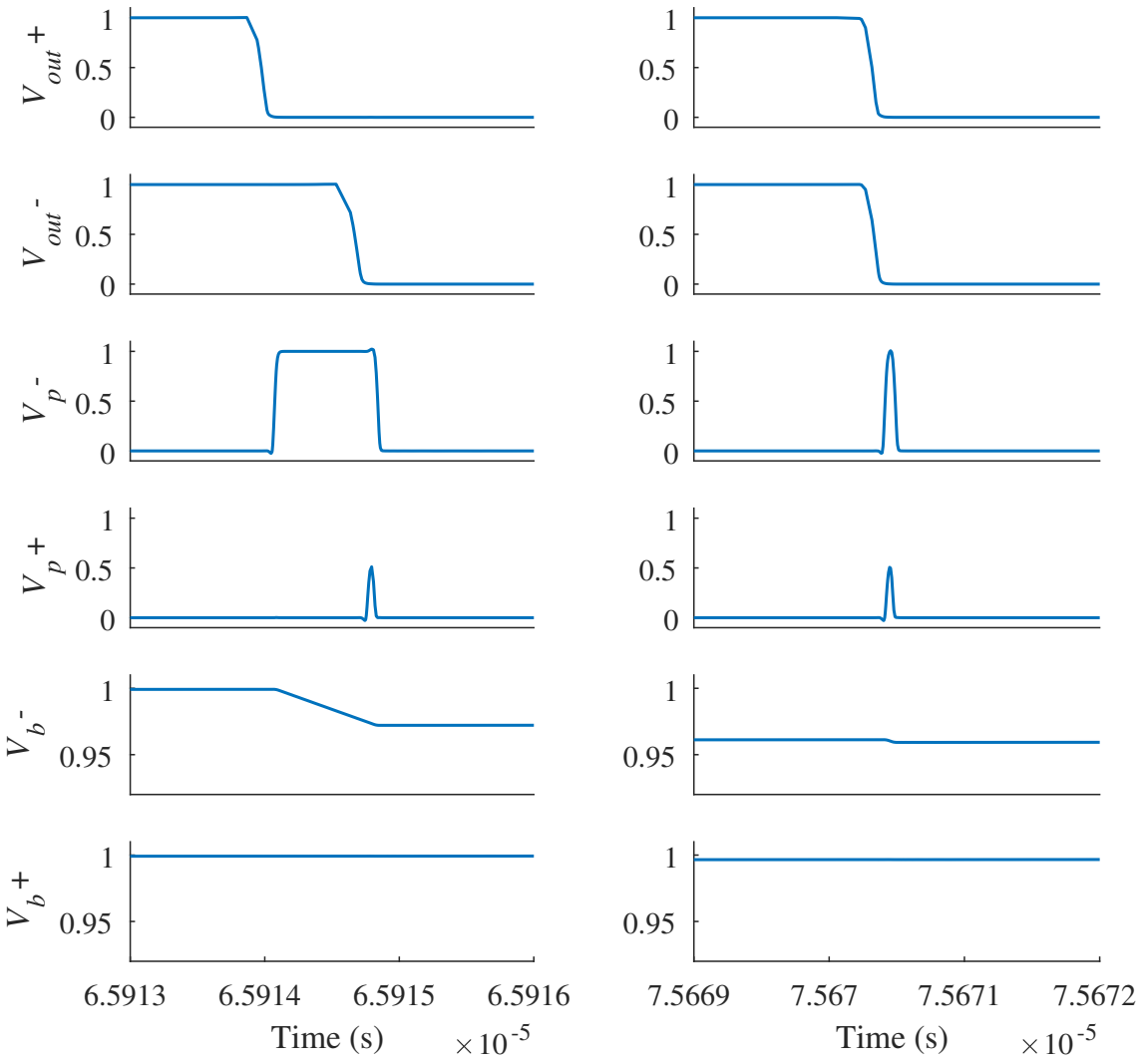


Figure 4.11: Signals within the comparator calibration feedback loop during the calibration cycle of a single point in a Monte Carlo run, for inputs at 0.5 V, zoomed in on the first and last cycles. On the left is the first calibration cycle, where we see a significant delay between the positive and negative outputs. On the right is the fifth calibration cycle, where that delay has been corrected by the bulk voltage. See Section 5.2 for the full results of the Monte Carlo simulation.

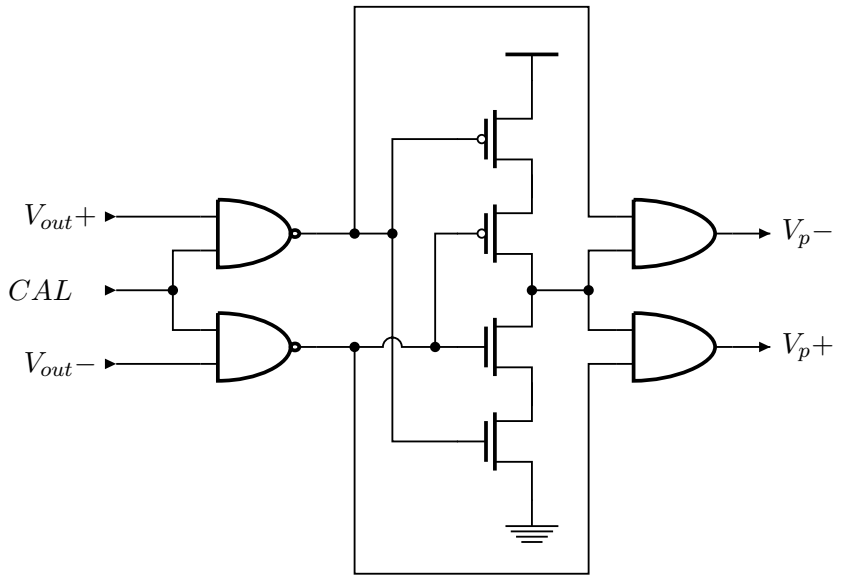


Figure 4.12: Schematic of the phase detector used in the comparator calibration feedback loop [26].

phase detector outputs a pulse on one of its two branches depending on which output fell first, whose width is approximately equal to the timing difference between the comparator outputs. We can see in Figure 4.11 that $V_{out}+$ falls approximately 10 ns before $V_{out}-$, resulting in a pulse on V_p-. A schematic of the phase detector is shown in Figure 4.12.

The phase detector's output pulses are fed into a charge pump (Figure 4.13[1]), which charges and discharges a pair of capacitors, C_b+ and C_b-, which are connected to the bulks of the comparator input transistors, V_b+ and V_b-, as shown in Figures 4.7 and 4.9. At the start of each calibration phase, when $\overline{reset}$ goes low, the charge

[1]Note that the two "dummy" PFETs connected to the inverter in Figure 4.13 were added during the design phase to offset observed "charge injection", which is a phenomenon that causes spikes in current as the charge held by switches is released when they switch on. The "dummy" transistors can be added to absorb some of this charge. However, simulations after the chip was fabricated found that this effect was minimal, having little effect on the performance of the calibration loop, so the "dummy" transistors are likely unnecessary. For more information on charge injection and dummy switches, see [4, p. 832]

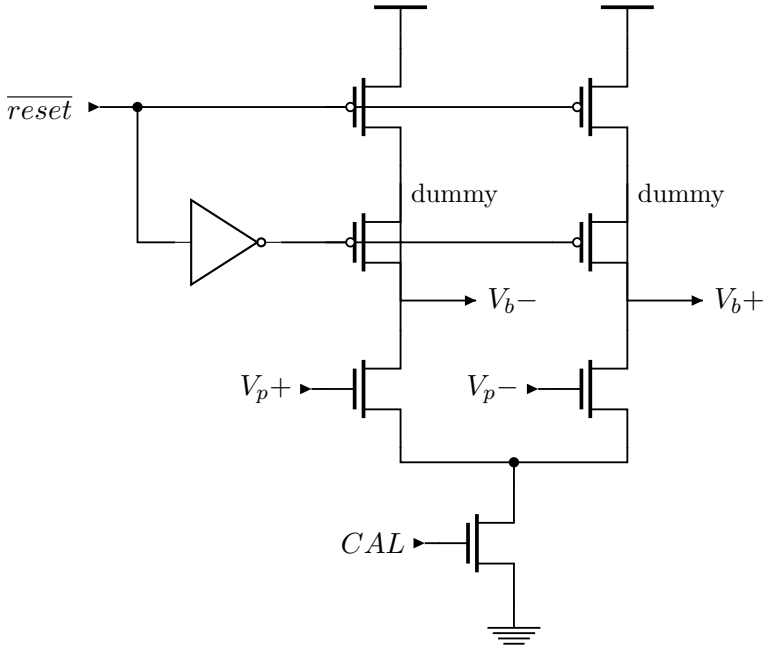


Figure 4.13: Schematic of the charge pump used in the comparator calibration feedback loop [26].

pump charges each capacitor to V_{dd}. Then when a pulse comes in from one of the phase detector's outputs, V_p+ or V_p-, the corresponding capacitor, C_b+ or C_b-, is discharged for the duration of the pulse. The amount of current sunk is set by the tail transistor (i.e. the NFET whose gate is connected to the CAL signal in Figure 4.13). This discharging decreases the voltage on the capacitor, which will modify the threshold voltage of the connected input PFET through the body effect. In Figure 4.11, we see V_b- decreases during the pulse, and by the fifth calibration cycle is nearly 5 mV below V_b+.

The change in the threshold voltage of one of the input transistors will counteract the offset voltage, thus closing the feedback loop. The feedback happens in discrete steps (once per rising clock edge), and can be repeated multiple times in a given calibration cycle in order to reduce the offset. In this design each time the calibration

is reset this cycle is repeated five times.

Note that overshoot, leakage, and glitches (as seen on V_p+ in Figure 4.11) will cause V_b+ to decrease slightly as well. This is expected, and the resulting delta between the bulk voltages (and therefore threshold voltages) of M_1 and M_2 is what reduces the input voltage offset, rather than their absolute values.

The loop gain of the feedback loop is

$$G = G_1 G_2 G_3, \tag{4.1}$$

where G_1 is the change in the time difference of the comparator output falling edges with respect to the input offset voltage in s/V, $G_2 = I_{tail}/C_b$ is the pulse time to output voltage gain of the charge pump in A/F (or V/s), and G_3 is the change in threshold voltage with respect to bulk voltage (in V/V) caused by the body effect. According to [26], convergence for the feedback loop is achieved with $0 < |G| < 2$, and the fastest convergence without overshoot should ideally be achieved with $G = 1$.

From Monte Carlo simulations it was found that with no calibration, the input offset voltage never exceeds about 20 mV (see Section 5.2 for Monte Carlo simulation results). In addition, V_b+ and V_b- do not go below about 0.9 V. The loop was designed with these considerations in mind.

G_3 is fixed by the process, and a sweep of the bulk voltage found it to be approximately 143 mV/V in this range as shown in Figure 4.14 (linearity is assumed here, as we can see the slope only changes very slightly in the range of expected bulk voltages).

G_2 is set by adjusting the size of C_b+ and C_b- and the tail current of the charge pump. In order to keep the droop of V_b+ and V_b- low, C_b+ and C_b- were made reasonably large. A value of 3.49 pF was used (its dimensions are listed in Table

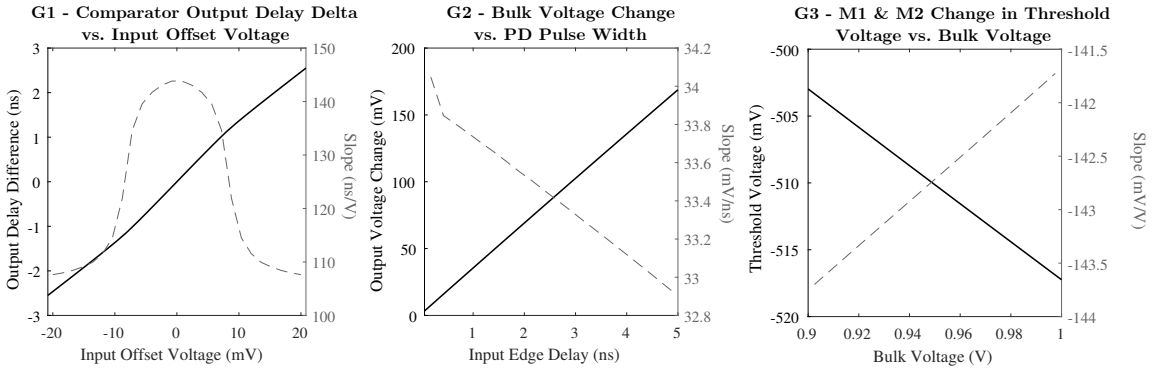


Figure 4.14: Simulation results for G_1, G_2, and G_3, in order to estimate the comparator calibration's loop gain.

Table 4.3: Comparator calibration capacitor dimensions and nominal capacitance

	Width (µm)	Length (µm)	Nominal Capacitance
$C_{b+,-}$	25.0	25.0	3.49 pF

4.3), and the tail transistor's width set to 1 µm (and minimum length, 0.12 µm), providing 118 µA to C_b+ or C_b- when V_p+ or V_p-, respectively, is high. This gives $G_2 = I_{tail}/C_b = 33.8$ µA/pF $= 33.8$ mV/ns, which matches the simulated result in Figure 4.14. The other NMOS in the charge pump were sized the same as the tail so that they could allow the tail current through them. The PMOS were kept at the minimum size. The sizes of the devices in the charge pump are listed in Table 4.13.

G_1 is set primarily by the size of M_1 and M_2 and the current supplied to them by M_5. To avoid adding extra parasitic capacitance to the capacitor array, M_1 and M_2 should be made as small as possible. Therefore, to increase G_1 the tail current was reduced. M_1 and M_2 were each given a width of 5 µm and minimum length of 0.12 µm, and M_5 was given a width of 0.5 µm and a length of 5 µm (the tail current supplied on the clock edge during calibration is about 44 µA). The input

Table 4.4: Charge Pump transistor dimensions (Figure 4.13).

		Width (µm)	Length (µm)
	NMOS	0.5	
	PMOS	0.16	
	Tail	1	0.12
INV	NMOS	0.2	
	PMOS	0.45	

capacitance of the comparator with these sizes is 4.26 fF. The other transistors in the comparator were simply sized to be small, but not minimum size so that they would be less susceptible to mismatch. The sizes of the transistors in the comparator are listed in Table 4.5. As seen in Figure 4.14, G_1 ranges from 144 ns/V for small offset voltages to 108 ns/V for large offsets[2].

The phase detector by design has a gain of approximately 1 (i.e. its output pulses are the same length as the delay between the inputs). The devices in the phase detector were sized to reduce the size of input delays that it can resolve. The NAND gates at the input were sized to be larger than the other devices and are driven by buffers (as shown in Figure 4.7) so that their slew rate is kept low. The remaining devices were chosen to be small but not minimum size. The sizes of the devices in the phase detector are listed in Table 4.6. The minimum input delay difference it can resolve is about 15 ps, which according to G_1 (for small values), corresponds to an input offset voltage of 104 µV.

By (4.1), the overall loop gain ranges from $G = 0.52$ V/V to $G = 0.70$ V/V. Initial designs had the estimated value of G closer to 1 V/V, but better results were obtained

[2]A detailed analysis of the effect of the non-linearity of G_1 is outside the scope of this thesis.

Table 4.5: Sizes for all components in the comparator (Figure 4.9). Note that S1 and S2 are each a single PMOS, and S3 and S4 are transmission gates (the PMOS and NMOS are sized the same in each).

	Width (µm)	Length (µm)
M5	0.5	5
M1, M2	5	0.12
M3, M4	0.5	
M6, M7	1	
M8, M9	1	
M10, M11	1	
M12, M13	0.5	
S1, S2	0.5	
S3, S4	0.5	

Table 4.6: Phase Detector transistor dimensions (Figure 4.12). Note that "AND INV" refers to the inverters inside the AND gates, not explicitly pictured in the schematic.

		Width (µm)	Length (µm)
	NMOS	0.4	0.12
	PMOS	0.9	
NAND	NMOS	1.2	
	PMOS	1.35	
AND	NMOS	0.4	
	PMOS	0.45	
AND INV	NMOS	0.2	
	PMOS	0.45	

in Monte Carlo simulations with the slightly lower simulated value of G found here (see Section 5.2).

4.3.3 Comparator and Calibration Layout

The layout of the comparator is shown in Figure 4.15. We can see that the comparator itself, near the middle, is a very small portion of the layout, with the calibration capacitors taking up more than two thirds the area. However, this method of calibration still takes much less area in total than the common method of connecting switched capacitors to the output nodes of the comparator.

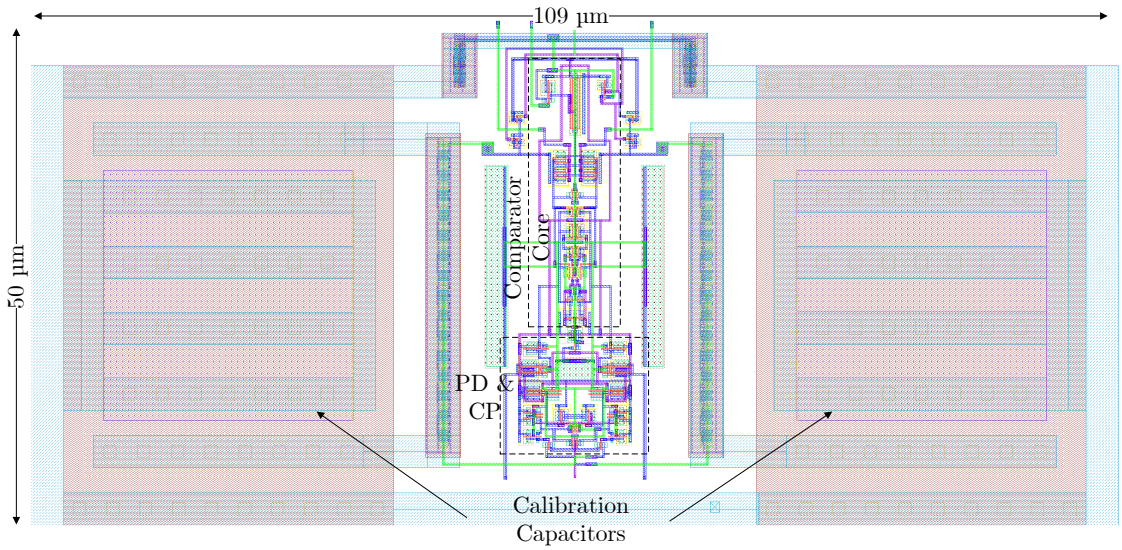


Figure 4.15: Comparator with calibration full layout. Total area is 0.00545 mm^2.

Every effort was made to keep the comparator's layout compact and symmetric about the central axis of the chip in order to minimize mismatch between the positive and negative paths. One drawback of the chosen calibration method is that the input PFETs could not be interdigitated, as their body connections (n-wells) will be set to different voltages. However, they were placed as close together as possible.

4.4 Capacitor Array and Switch Design

The capacitor array is controlled by the digital logic block and generates the voltage that the comparator compares to the sampled input signal.

A simplified schematic of the capacitor array is shown in Figure 4.16. The control signals for the switches $S_{0...24}$, S_{vdd}, and S_{gnd} come from the logic block. S_{vdd} and S_{gnd} connect the array and any capacitor whose switch is closed to ground and V_{dd}, respectively (note that S_{gnd} is an NFET so it is active high, and S_{vdd} is a PFET so it is active low). One major difference between this scheme and that used by [3] and [10] is that the switches for connecting to ground and V_{dd} are common to all the capacitors, rather than giving each one three switches (one to ground, one to V_{dd}, and one to connect to V_x). This helps to prevent the logic from becoming even more complex, and also saves some layout space and complexity by effectively reducing the number of switches by a factor of 3, at the cost of reduced flexibility in the control logic (e.g. one capacitor cannot be connected to V_{dd} while another is connected to ground).

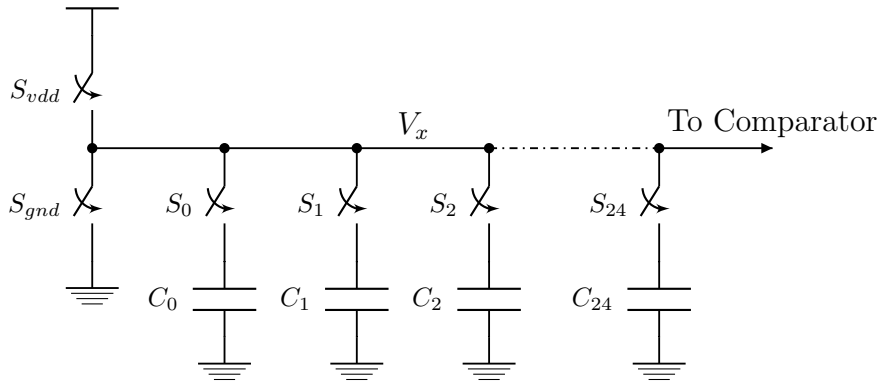


Figure 4.16: Capacitor array with switches schematic.

The size of the capacitors in the array was chosen to be 6 pF, as explained below.

Table 4.7: Array capacitor dimensions and effective capacitance. Each 6 pF capacitor $C_{0...24}$ consists of two 3 pF capacitors in parallel, so these dimensions are for the individual capacitors.

	Width (µm)	Length (µm)	Nominal Capacitance
$C_{0...24_{half}}$	22.86	22.86	3.0 pF

Each capacitor in the array was made using two 3 pF capacitors in parallel in order to accommodate a common-centroid layout (discussed further below). The dimensions of the individual capacitors are shown in Table 4.7.

There are clear tradeoffs for performance, layout area, power consumption, and speed when choosing the size of the capacitors. For example, since 12 of the capacitors are charged to 1 V at the start of each conversion and the charge on them is reused, we can approximate the energy consumption of the capacitor array per conversion (ignoring the cases where an extra capacitor is charged during the conversion, which will slightly increase the average power):

$$E = \frac{1}{2}CV^2 = \frac{1}{2}\left(12 * 6 \text{ pF}\right)1^2 = 36 \text{ pJ/conversion.} \tag{4.2}$$

Then the power consumption at the sampling frequency of 10 kHz is

$$P = \frac{E}{T} = \frac{36 \text{ pJ}}{100 \text{ µs}} = 360 \text{ nW}, \tag{4.3}$$

which scales linearly with the size of the capacitors, so smaller capacitors would reduce power consumption.

Simulations using an ideal logic block and comparator and circuit-level capacitor array were performed to calculate the effective number of bits (ENOB) of the ADC with varying unit capacitor sizes and parasitic capacitance on node V_x, the results

of which are shown in Figure 4.17. The closer the ENOB is to the actual number of bits, the better (it is directly related to the noise and distortion generated by the converter, as will be discussed further in Chapter 5). Each switch in the capacitor array adds about 0.16 fF of parasitic capacitance and the input capacitance of the comparator is 4.26 fF, so the total parasitic capacitance on V_x is about 8.7 fF.

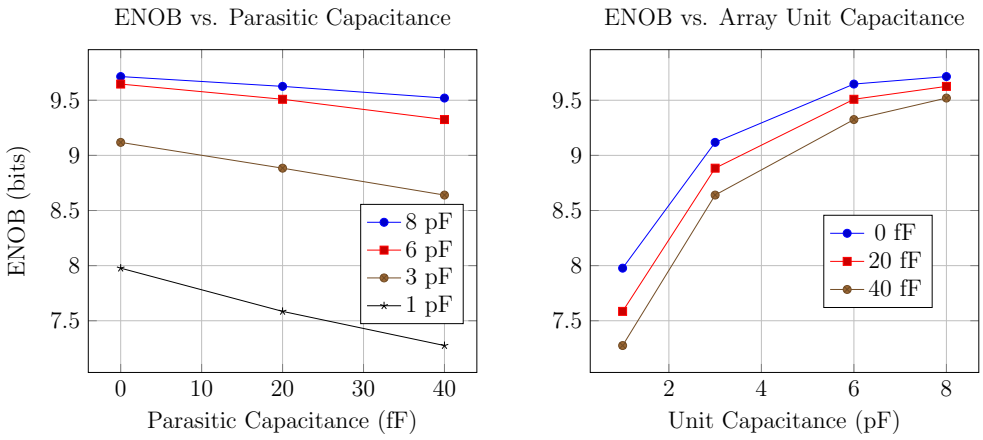


Figure 4.17: Effective number of bits (ENOB) using ideal comparator and logic and schematic level capacitor array versus array parasitic capacitance for multiple values of unit array capacitance (right) and versus unit array capacitance for multiple values of array parasitic capacitance (left).

We can see that increasing the size of the capacitors on the array can significantly improve the ENOB. The bigger capacitors also make the ENOB less susceptible to parasitic capacitance on V_x (e.g. from traces or the input capacitance of the comparator), as we can see from the reduced slope for larger unit capacitors on the ENOB vs. Parasitic Capacitance plot. However, we can see that the improvement in ENOB tapers off as we increase the unit capacitor size. With a 40 fF parasitic capacitor, going from 6 pF to 8 pF unit capacitors offers a 0.195 bit improvement in ENOB at the cost of 33% more power (as per (4.2) and (4.3)) and 33% more layout area. Therefore 6 pF was chosen as the size of the unit capacitors in this design as a reasonable trade-off in performance vs. power and layout area.

Table 4.8: Capacitor array switch dimensions and on and off resistance (determined from DC simulation with $V_{in} = 1$ V).

		Width (µm)	Length (µm)	On Resistance	Off Resistance
$S_{0...24}$	NFET	0.3	0.2	9.8 kΩ	8.9 GΩ
	PFET	0.3	0.2		
S_{gnd}	NFET	1.5	0.2	2.4 kΩ	3.9 GΩ
S_{vdd}	PFET	4.5	0.2	2.6 kΩ	1.2 GΩ

The sizes and on/off resistance of the array's switches are shown in Table 4.8. Note that each capacitor is connected to its switch with interconnect wiring whose average lengths are about 900 µm (see Section 5.3.1 for more details). For the wiring widths used in the layout, the resistance is ~ 0.1 Ω/µm, giving a total series interconnect resistance of about 90 Ω for each capacitor. This is negligible compared to the on and off resistances of the switches and will be ignored in the following calculations.

For the case of charging one capacitor to 1 V, we get a time constant of

$$\tau_{charge} = R_{ON_{charge}} C_{unit} = (2.6\ \text{k}\Omega + 9.8\ \text{k}\Omega)(6\ \text{pF}) = 74.4\ \text{ns}, \tag{4.4}$$

and for discharging to ground we get

$$\tau_{charge} = R_{ON_{discharge}} C_{unit} = (2.4\ \text{k}\Omega + 9.8\ \text{k}\Omega)(6\ \text{pF}) = 73.2\ \text{ns}, \tag{4.5}$$

which are both sufficiently fast. The worst case for speed is in the INIT state of the state machine, when twelve of the capacitors are charged to V_{dd} at once. Since the twelve switches and capacitors being connected in parallel are identical we can say

$$C_{eq} = 12 C_{unit} = 72\ \text{pF} \tag{4.6}$$

and for the array switch

$$R_{eq} = \frac{R_{ON}}{12} = 817\,\Omega. \tag{4.7}$$

So the time constant for the worst case is

$$\tau_{wc} = (R_{ON_{vdd}} + R_{eq})(C_{eq}) = (2.6\text{ k}\Omega + 817\,\Omega)(72\text{ pF}) = 246\text{ ns}. \tag{4.8}$$

These capacitors have one period of the 410 kHz clock to reach 1 V, or 2.44 µs. After this time the voltage should be

$$V_{charge} = 1 - e^{-t/\tau_{wc}} = 1 - e^{-2.44\,\mu\text{s}/246\,\text{ns}} = 1 - (4.92 \times 10^{-5}) \approx 0.99995\,\text{V}, \tag{4.9}$$

which is sufficiently close to 1 V. We can see that there is an inverse relationship between the size of the capacitors and the speed of the circuit. So for a circuit sampling at a faster speed, smaller capacitors could likely be used.

The amount of droop on the capacitors can similarly be estimated. For simplicity the case of 1 V on a single capacitor and 0 V on the array will be considered. This gives a time constant of

$$\tau_{droop} = R_{OFF}C_{unit} = (8.9\,\text{G}\Omega)(6\,\text{pF}) = 53.4\,\text{ms}. \tag{4.10}$$

The amount of voltage droop on a capacitor after one period of the 410 kHz clock is then

$$V_{droop} = 1 - e^{-t/\tau_{droop}} = 1 - e^{-2.44\,\mu\text{s}/53.4\,\text{ms}} = 45.8\,\mu\text{V}. \tag{4.11}$$

In some cases, this charge could be held on a capacitor for 19 clock periods before being connected to the array to be averaged, so the droop could be as high as 870.2 µV, which is close to 1 LSB. However, in practice there are two reasons it is unlikely

to be a large source of error. First, there are few codes for which 1 V (or 0 V) is held on a capacitor for the entirety of the conversion before being used. Second, the voltage on the common node of the array is not held constant for the duration of the conversion, so there would never be 1 V across any of the switches for longer than one period of the clock at a time. Because of this, the amount of droop expected is considered acceptable. This is also partially confirmed by the simulated ENOB as discussed above in Figure 4.17. However, further investigations after the fabrication of the chip found that reducing leakage through the switches would likely result in significant improvement in the ENOB for smaller capacitors. This will be discussed further in Section 7.2.

Note that charge injection from the switches in the capacitor array was not considered during the design and will be left as future work.

4.4.1 Capacitor Array Layout

The capacitors chosen for the array were the dual layer metal-insulator-metal capacitors (dual mimcaps) available in the Global Foundries design kit. A cross-section and functional view of the capacitors are shown in Figure 4.18. They are built by stacking three metal (aluminum) layers and two thin dielectric (silicon nitride) layers, essentially giving two capacitors which can then be connected in parallel, as shown. The per unit area capacitance of the dual mimcaps in this kit is 4.10 $\text{fF}/\mu\text{m}^2$ in the regions where the metal layers overlap.

The layout of the capacitor array is shown in Figure 4.19. There are many things that must be taken into account when laying out a capacitor array when we want all the devices to be matched. A good resource for some rules of thumb is [23, p. 300].

Many of these rules of thumb were applied here. These include:

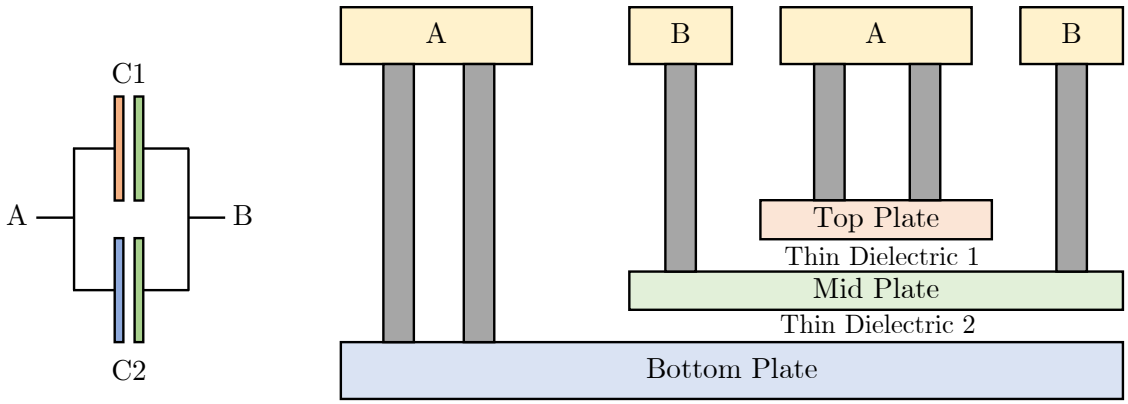


Figure 4.18: Cross-sectional view of the dual mimcaps used in the capacitor array (right) and functional view of the same (left). The capacitors C1 and C2 are formed where the top plate and midplate overlap and where the mid plate and bottom plate overlap, respectively. The approximate capacitance is 4.10 fF/µm^2 in the overlap areas.

- The capacitors are all the same dimensions. This helps to minimize differences in edge parasitics.

- All the capacitors are square. Squares have the lowest perimeter to area ratio of any rectangle, therefore the edge parasitics are minimized relative to capacitance.

- The capacitors are not the minimum size, as smaller devices are more susceptible to variation.

- Capacitors are cross-coupled. Each 6 pF capacitor here was divided into two 3 pF capacitors which were connected in parallel. Then the array was arranged so that each pair shares a common centroid - that is, a straight line drawn between the two capacitors in a pair crosses through the same point (the center of the array) for each pair. This helps to minimize variation due to oxide-, stress-, and thermal-gradients. In Figure 4.19, each pair of capacitors is labeled with

Figure 4.19: Layout of the capacitor and switch array. Each 6 pF capacitor is made of two 3 pF dual mimcaps in parallel. Dashed red lines connect each pair of capacitors, showing the common centroid. In addition, dummy capacitors (labeled "D") are added so that each capacitor that is used is surrounded on all sides by the same geometry. The switches are arranged in a row at the top of the array. Total area is 0.194 mm^2

the same number (e.g. the two capacitors labeled "0" are connected in parallel) and connected with dashed red lines. We can see that all the lines intersect at a point in the middle of the array.

- The array is placed on the axes of symmetry of the die, which helps cross-coupling minimize variation due to stress-related gradients.
- The capacitors are placed as close together as possible.
- "Dummy" capacitors are placed around the edge of the array. These are devices with the same geometry as the capacitors used, but which are not functionally attached to any part of the circuit. These are added to eliminate variation due to etch rate and differences in edge effects between devices. The dummy capacitors have their electrodes connected together to prevent static charge from accumulating. In Figure 4.19 the dummy capacitors are labeled with the letter "D".

Note that the "dummy" devices used here are full size. However, the ones around the edge of the array could be cut in half while maintaining the same dimensions on the edges that are adjacent to active capacitors. This would reduce each dimension of the capacitor array by about 22 µm.

4.5 SAR Logic

One of the major advantages of this work is the automatic generation of the state machine logic. Figure 4.20 shows the design flow. As discussed in Chapter 3 the outputs of the backtracking algorithm were fed to a custom script that automatically generates the Verilog Code for the state machine logic. The Verilog code was then imported into Synopsys Design Vision for synthesis using a custom standard cell

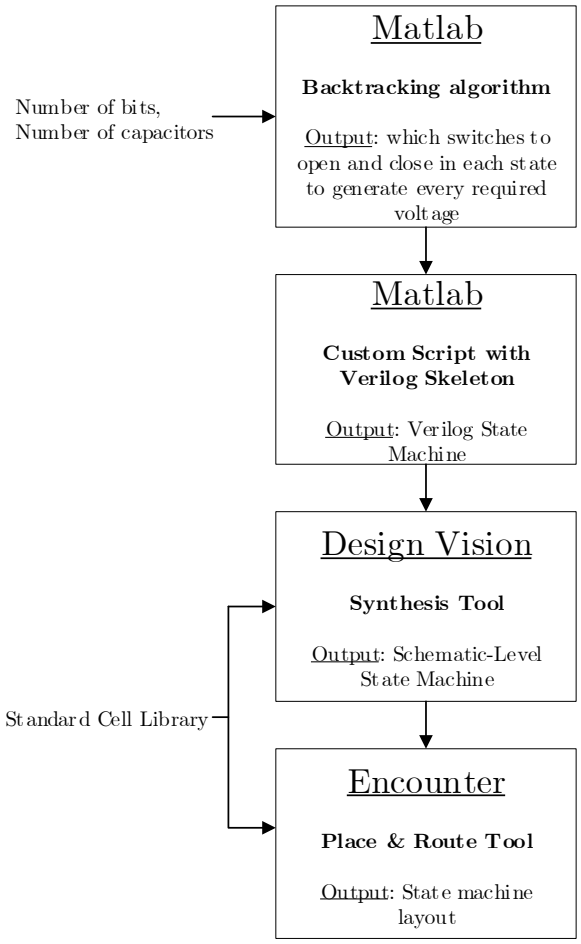


Figure 4.20: Design flow for automatic generation of the state machine logic using the outputs of the backtracking algorithm.

library. The standard cells used were: a flip flop with asynchronous set, a flip flop with asynchronous reset, two input NAND and NOR gates, an inverter, and an SR Latch. Each cell is 12 tracks tall. The tracks are the sum of minimum metal 1 minimum wiring pitch plus the minimum metal 1 spacing, giving 0.32 µm, so the total height for each cell is 3.84 µm. All the standard cells except the Latch were custom made for [10].

After synthesis, the schematic of the logic block consists of 458 inverters, 3166 NAND gates, 3051 NOR gates, 38 flip flops with synchronous reset, 4 flip flops with

asynchronous reset, and 18 latches - a total of 6735 cells. All the NAND and NOR gates used are 2-input, and every standard cell used has a drive strength of 1. A more robust library with higher drive strengths and multiple input gates would likely result in smaller layout area and lower power consumption for the logic block.

After being synthesized, the logic was imported into Cadence Encounter to be placed and routed. The final layout is shown in Figure 4.21, and has a total layout area of 320 µm by 800 µm. To simplify the process and minimize post-route DRC errors, only the lowest three metal layers (M1, M2, and M3) were used for routing.

One potential downside of this ADC design is that the SAR logic is much more complex than that used for a other SAR ADCs. For example, binary-weighted charge redistribution SAR ADCs require a set of flip flops for each bit, and some simple combinational logic to decide when they are clocked, so the logic's complexity increases linearly with the resolution. In comparison, the number of states (and therefore the size and complexity) in this design increases exponentially with the resolution. Although not considered in this design, at higher speeds this may cause issues related to clock skew and could limit the allowable operating frequency. Any future designs at higher speeds will need to consider this.

4.6 Pad Drivers

In order to charge the capacitance of the output pads, large transistors were required. The pad drivers used for the digital outputs, along with their test setup, are shown in Figure 4.22. The drivers consist of a series of four inverters of increasing size. All buffers shown in block diagrams in this chapter (i.e. Figure 4.1 and Figure 4.7) consist of the first two inverters in Figure 4.22. The pad drivers consist of all four

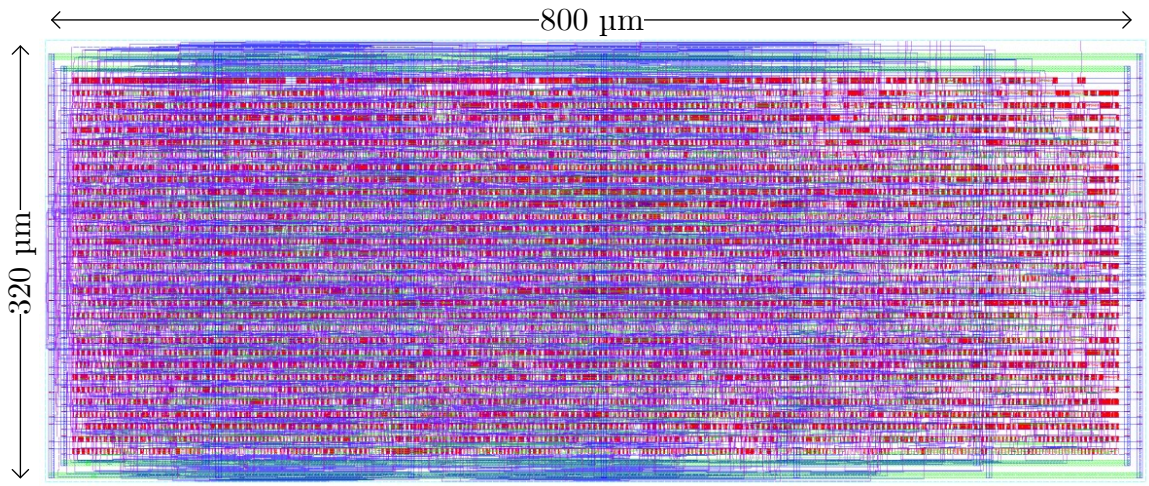


Figure 4.21: Final layout of the SAR Logic block. Total area is 0.256 mm^2.

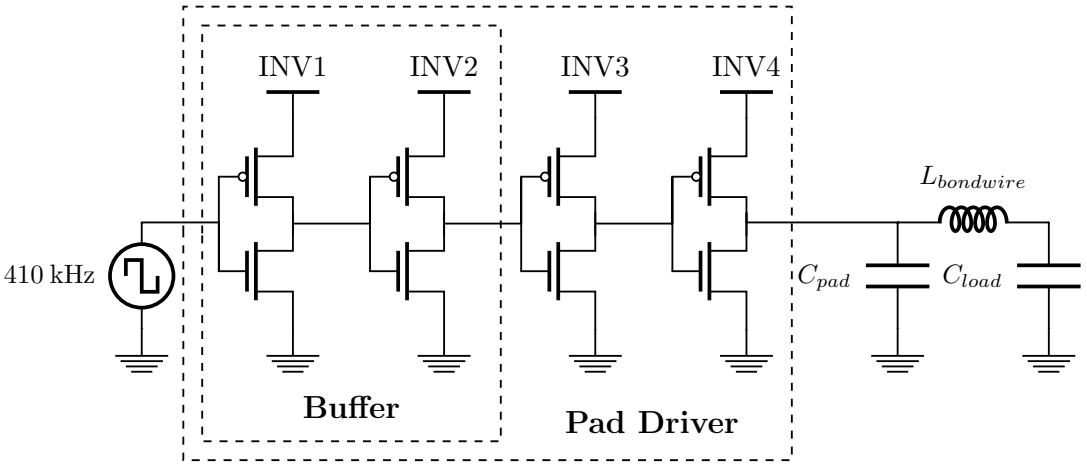


Figure 4.22: Schematic of the output drivers.

Table 4.9: Transistor dimensions for the four inverters used in the output drivers.

		Width (µm)	Length (µm)
INV1	NFET	0.16	0.12
	PFET	0.32	
INV2	NFET	0.5	0.12
	PFET	1	
INV3	NFET	1	0.12
	PFET	4	
INV4	NFET	6	0.12
	PFET	18	

Table 4.10: Component values for output driver verification.

	Value	Description
C_{pad}	$\sim$150 fF	108 µm × 108 µm bondpad cell
C_{load}	50 pF	Estimated worst-case load capacitance
$L_{bondwire}$	2 nH	Estimated bondwire inductance

inverters. Transistor sizes are shown in Table 4.9.

The drivers were tested with a 410 kHz square wave input while loaded with a bondpad model, a 2 nH inductor to model a bondwire, and a 50 pF capacitor to account for any PCB pad capacitance or external circuitry input capacitance[3]. Figure 4.23 shows the simulation results from this test. We can see that the drivers exhibit virtually no delay on this timescale, and their rise and fall times are less than 100 ns.

[3]Note that a very large capacitor was used in these simulations simply because at the time of the design the size of the drivers' external load was unknown.

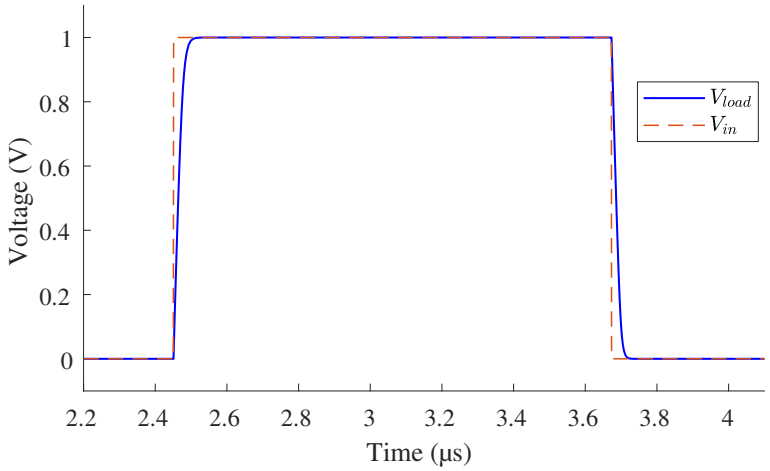


Figure 4.23: Simulated input and output voltages of the output driver during verification.

4.7 Summary

In this chapter, the circuit-level design of each block of the SAR ADC was discussed. The next chapter looks at the overall simulated performance of the S/H and comparator. Then the system is connected together and the overall simulated performance of the ADC is discussed.

Chapter 5

Simulation Results

This chapter will discuss the simulated verification of the various blocks of the SAR ADC. First the S/H and comparator performance will be discussed. Then the system as a whole will be simulated, and several common ADC metrics calculated. Finally, the power consumption of the major blocks will be discussed.

The simulation results in this section demonstrate the the proposed algorithm leads to a functional design despite a design flaw that was found after fabrication that led to the chip not functioning (see Chapter 6). Unfortunately the flaw cannot be fixed or mitigated in the manufactured hardware. However, in future designs the error can be corrected easily, and once it is it is expected that this architecture will work as desired.

5.1 Sample & Hold Performance

The S/H must be verified in order to ensure it is not the limiting factor in performance.

To see that charge time would not cause problems, a simulation was performed with the S/H switch closed and a fullscale (0.5 V amplitude centered on 0.5 V DC)

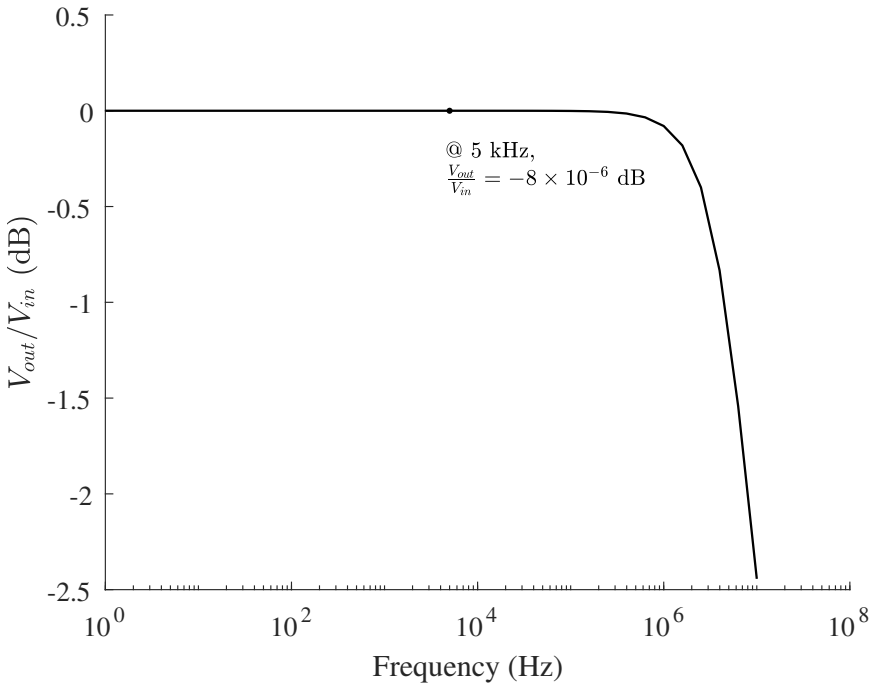


Figure 5.1: Transfer curve of the S/H when the switch is closed.

sine wave swept from 1 Hz to 10 MHz. The resulting transfer curve is shown in Figure 5.1. We can see that with the switch closed the S/H acts like a lowpass filter with a cut-off frequency above 10 MHz, so it should have no trouble charging below the Nyquist frequency (5 kHz).

As mentioned in Section 4.2, because of the low frequency of operation, "droop" of the signal during the "hold" phase may be a concern. A simulation was performed with a 622 Hz, fullscale sinewave. The droop of the S/H was determined by taking the value of the output 1 µs after the start of the hold phase and subtracting the output 1 µs before the end of the hold phase. A visual inspection of several points verified that at 1 µs the signal was sufficiently settled. Figure 5.2 shows the droop calculated this way versus time and versus the initial voltage (i.e. 1 µs after the start of the hold phase). We can see that the droop never approaches 0.5 LSB (488 µV),

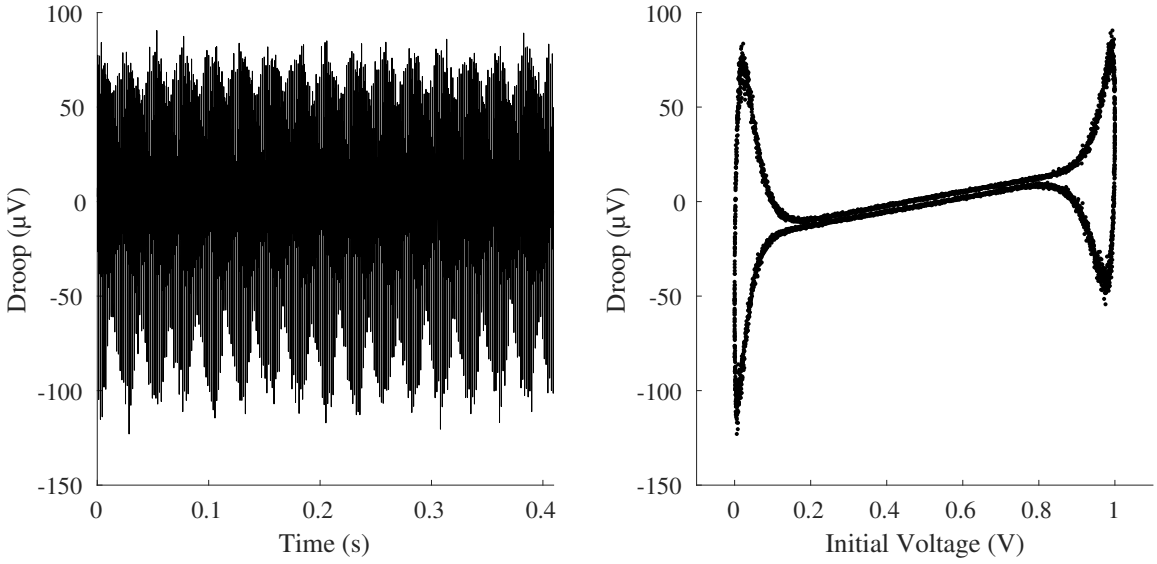


Figure 5.2: Sample and Hold droop versus time (left) and versus initial voltage (right) for a 622 Hz input signal.

with the worst case being -123 µV.

Next, the above simulation was repeated with transient noise enabled. Figure 5.3 shows the results of performing an FFT on the output, taking the points at the start of the hold phase (left) and the end of the hold phase (right). We can see that the Signal to Noise and Distortion Ratio (SNDR) degrades slightly due to droop. However, the Effective Number Of Bits (ENOB) is well above 10 bits, so we should not expect the S/H to be the limiting factor in performance for the ADC. ENOB will be discussed further in Section 5.3.1.

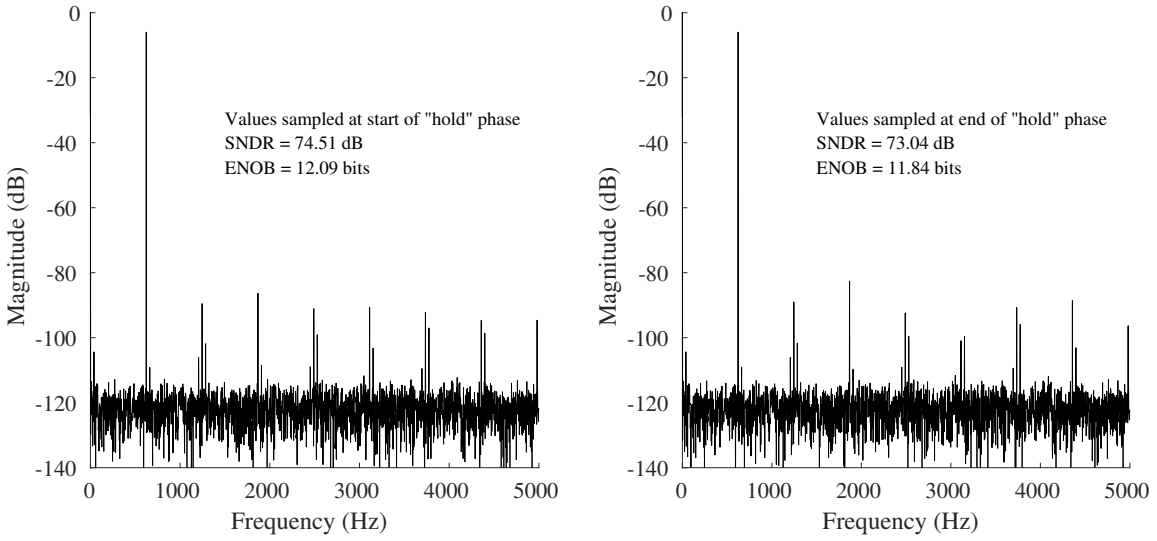


Figure 5.3: Spectra of S/H values for a 622 Hz input sine wave, values taken at the start of the "hold" phase (left) and end of the "hold" phase (right).

5.2 Comparator Performance

5.2.1 Comparator Monte Carlo Simulations

In order to verify the operation of the comparator's calibration, transient Monte Carlo simulations were performed to determine the input offset voltage, V_{os}. The Monte Carlo simulations randomly vary the parameters of the design kit components according to data gathered by the manufacturer. Settings for mismatch-only simulations were used for these simulations in order to see how well the calibration works.

Monte Carlo simulations with four hundred trials were performed on the comparator with and without calibration enabled. For the case without calibration enabled, the capacitors attached to the bulks of the bulks of M_1 and M_2 in the comparator (C_b+ and C_b-) were kept charged to V_{dd}. For the case with calibration enabled, calibration was refreshed every 20 clock cycles. Histograms from each of these are

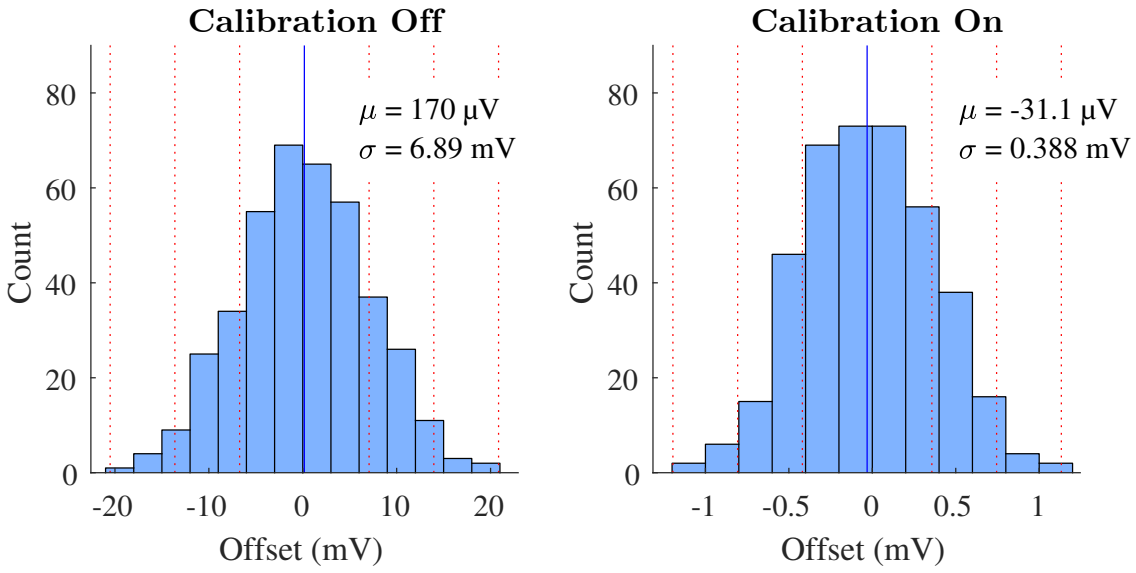


Figure 5.4: Mismatch-only Monte Carlo results of the schematic level comparator without calibration (left), and with calibration (right). Each was performed with 400 trials. The blue solid line represents the mean of each, and the red dotted lines steps of one, two, and three standard deviations from the mean (assuming a Gaussian distribution). The mean (μ) and standard deviation (σ) for each are labeled.

shown in Figure 5.4.

The input offset voltage of the comparator was determined by holding the positive input of the comparator constant at 0.5 V and applying a ramp to the negative input. The clock was simulated at 410 kHz. While $V_{in-} < V_{in+} + V_{os}$, V_{out-} falls on the rising clock edges and V_{out+} remains high. As soon as V_{out+} falls on the clock edge, the simulation stops and we record $V_{os} = V_{in-} - V_{in+}$. Note that because the comparisons are performed at discrete times, the resolution of this simulation is limited by the clock frequency and the slope of the input ramp. For the case without calibration, the resolution was 50 µV. For the case with calibration the resolution was 10 µV.

We can see that the standard deviation, σ, of the offset voltage is 6.89 mV without calibration and 0.388 µV with calibration, a reduction by a factor of 17.8. This is less

than the ∼100x improvement with a 0.5 µm process and the 65x improvement in a 90 nm process reported in [26]. However, the comparator in this work was designed to work in a system and not simply in isolation. We can see that the calibration brings the offset of the comparator to less than 1 LSB and therefore is acceptable for this ADC.

5.2.2 Comparator Speed

Since the intended operating speed of ADC is 10 kHz, the speed of the comparator was not a major concern during the design process. Figure 5.5 shows the results of a simulation with $V_{in+} = 500\,\text{mV}$ and $V_{in-} = 499.5\,\text{mV}$. We can see that V_{out-} falls 12.8 ns after the rising clock edge. This time delay was found to be consistent over most of the comparator's input range, and is significantly less than half a clock period (1.22 µs), so the speed of the comparator is sufficient.

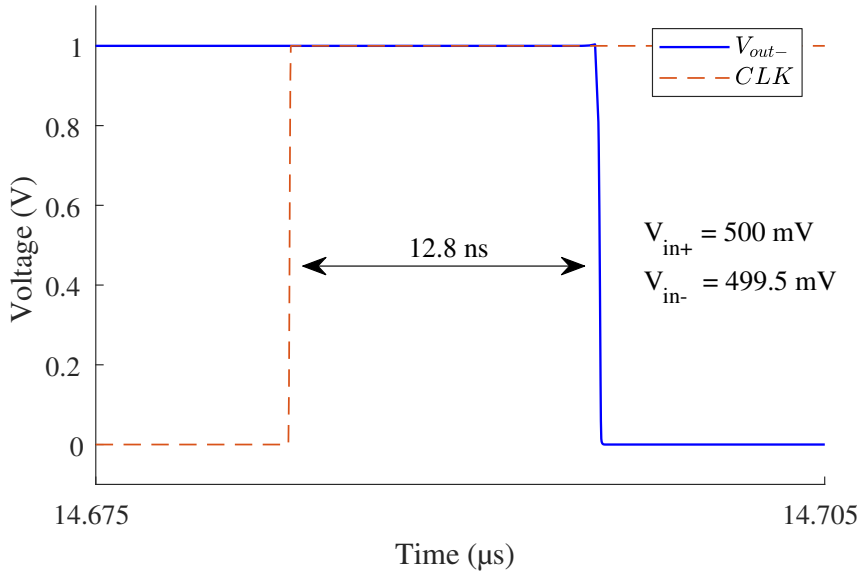


Figure 5.5: The time required for the comparator's V_{out-} to fall with $V_{in+} = 500\,\text{mV}$ and $V_{in-} = 499.5\,\text{mV}$.

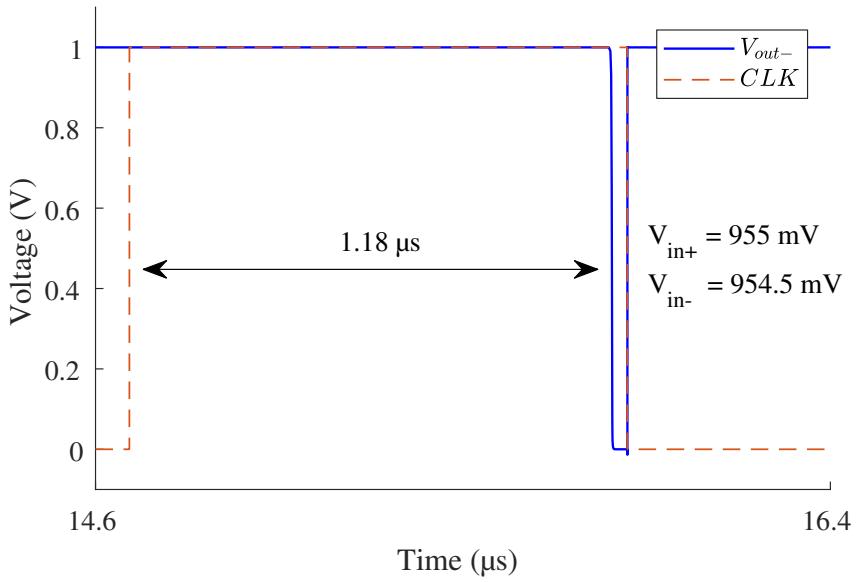


Figure 5.6: The time required for the comparator's V_{out-} to fall with $V_{in+} = 955$ mV and $V_{in-} = 954.5$ mV.

There is, however, a speed problem when the input voltages approach V_{dd}. Figure 5.6 shows a similar simulation to above, but with $V_{in+} = 955$ mV and $V_{in-} = 954.5$ mV. Here the delay from the rising clock edge to V_{in-} falling is 1.18 µs. For a common mode voltage any higher than this, the delay is longer than half a clock period, meaning the comparator never actually switches (since it is reset when the clock rises). Figure 5.7 shows the output delay for a differential input of 0.5 mV while sweeping the common mode voltage from 0 to 0.95 V. We can see that the delay increases rapidly as the voltage increases.

The possible cause of this increased delay is that V_{sg} for the two input transistors is well below their threshold voltage of about 400 mV as the common mode voltage increases. In subthreshold operation the current that a PFET can source decreases exponentially with V_{sg} [4]. Since the current from the first stage charges the input capacitance of the second stage, it follows that it would be proportional to the delay

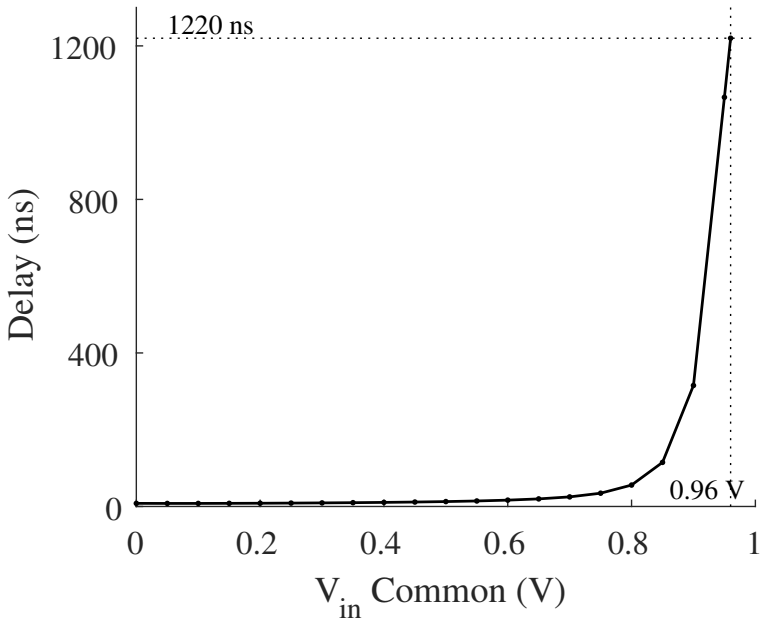


Figure 5.7: Output delay of the comparator with $V_{in+} - V_{in-} = 0.5\,\text{mV}$, with V_{CM} swept from 0 V to 0.95 V. Note that for common mode voltages above ∼0.96 V the delay was greater than half a clock period.

of the outputs. This matches the somewhat exponential shape of Figure 5.7.

Unfortunately, this issue was not discovered until late in the design process. Therefore it was decided that instead of redesigning the comparator, input voltages being fed to the ADC would be kept below the range where the comparator is too slow to operate. In general for system-wide simulations, the input voltage was kept below 0.9 V.

5.3 System Performance

5.3.1 Effective Number of Bits

The effective number of bits (ENOB) of an ADC is directly related to its signal to noise and distortion ratio (SNDR or SINAD) and is commonly calculated as

$$\text{ENOB} = \frac{\text{SNDR} - 1.76}{6.02}, \tag{5.1}$$

which comes from the equation for ideal quantization noise given a number of bits, and solving for the number of bits instead [27][28]. The SNDR is determined using a single tone input at or near the full-scale input of the ADC, taking the FFT of the output and comparing the power of the fundamental to everything else (except DC). When a full-scale input is not used (and assuming the gain of the ADC is 1), the ENOB can be calculated as [28]

$$\text{ENOB} = \log_2(\text{SNDR}) - \frac{1}{2}\log_2(1.5) - \log_2\left(\frac{A}{FSV}\right), \tag{5.2}$$

where SNDR is the ratio of the rms power of the fundamental to the rms power of the noise and distortion, A is the peak to peak amplitude of the input signal, and FSV is the full scale voltage of the ADC. (5.2) tends to give more optimistic results because any non-linearities or noise above the amplitude of the input signal will be ignored [29]. Because of the issues with the comparator discussed above, the ENOB tests here will not use a full scale signal. Both (5.1) and (5.2) will be calculated here. The ENOB calculated with (5.1), without adjusting for amplitude, represents the actual performance of the ADC as it is now and will be refered to as ENOB*. The ENOB calculated with (5.2), with the amplitude adjustment, represents the performance of

the ADC assuming the linearity for input voltages above A would be consistent with those below A if the comparator worked to full scale, and will be refered to as ENOB.

When measuring the SNDR of the ADC it is helpful to use a coherent signal (i.e. the first and last sample should line up in the period of the signal). This prevents spectral leakage and the need for applying a windowing function to the signal. In addition, in order to keep the quantization noise random, the ratio of the input frequency, f_{in} to the sampling frequency, f_s should be

$$\frac{f_{in}}{f_s} = \frac{N_{cycles}}{N_{record}}, \tag{5.3}$$

where N_{cycles} is the number of cycles of the input sinewave recorded and is not a sub-multiple of N_{record}, and N_{record} is the total number of samples taken and is a power of 2. This prevents repeated samples from being taken [27].

Because of the complexity of the SAR's digital logic block, post-layout simulations of the entire chip were infeasible to perform. In fact, circuit-level simulations of the logic block for the length of time required for measuring ENOB, INL, and DNL with any accuracy were infeasible. For this reason, all system-level simulations were performed using a functional block for the SAR logic. Note, however, that short schematic-level simulations of the logic block were performed in order to verify its general functionality.

Several simulations were performed to calculate the achieved ENOB. All of these were done using $N_{cycles} = 255$ and $N_{record} = 4096$, giving $f_{in} \approx 622.6$ Hz. This was then performed using schematic-level comparator and capacitor array blocks with and without transient noise, and then with extracted layouts without transient noise. Note that the simulations with extracted layouts include the layouts of the capacitor array and comparator, but not the interconnects between them. The FFTs of the

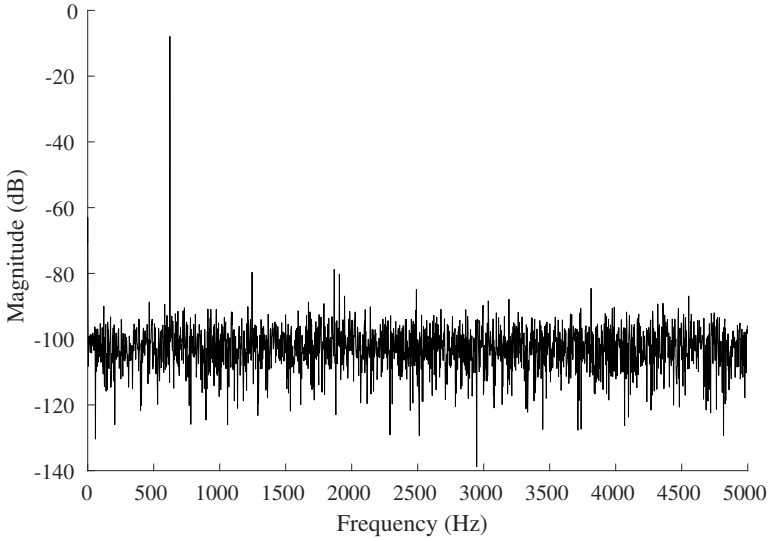


Figure 5.8: Simulated ADC output spectrum using a functional block for the SAR logic and schematic level circuits for the remaining blocks. The input tone frequency was ∼622.6 Hz with an amplitude of 0.4 V centred around 0.5 V DC (the DC component was removed from the signal before the FFT was performed). This simulation does not include transient noise. The SNDR was calculated to be 57.4 dB.

outputs are shown in Figures 5.8 - 5.10. The calculated results are shown in Table 5.1.

We can see that an ENOB of 9.57 bits is achieved on the schematic-level ADC, with only a slight degradation in performance to 9.39 bits due to noise. However, the SNDR is degraded by about 6.5 dB between the schematic and layout simulations without noise, resulting in 8.49 bits, more than 1 effective bit lost. Because this simulation took an exceedingly long time to perform[1], this was not discovered until after the design had been submitted for fabrication and it was not possible to improve the layout.

[1]About 3 weeks for 4096 points.

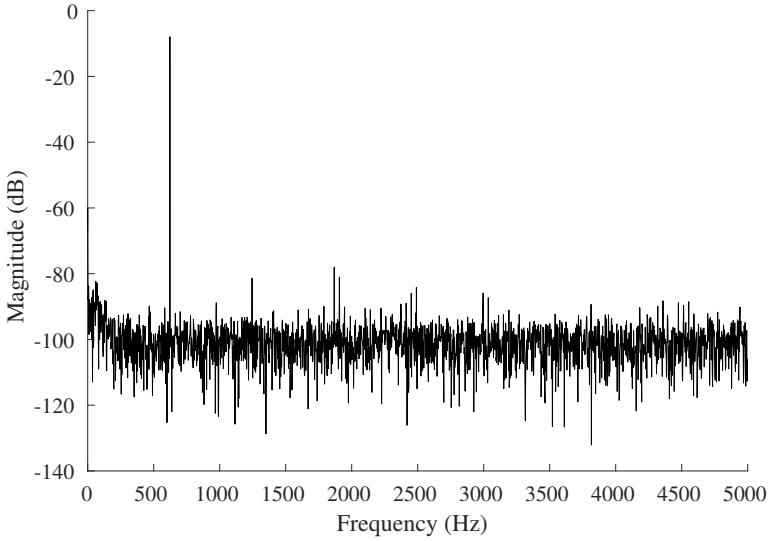


Figure 5.9: Simulated ADC output spectrum using a functional block for the SAR logic and schematic level circuits for the remaining blocks. The input tone frequency was $\sim$622.6 Hz with an amplitude of 0.4 V. This simulation includes transient noise. The SNDR was calculated to be 56.4 dB.

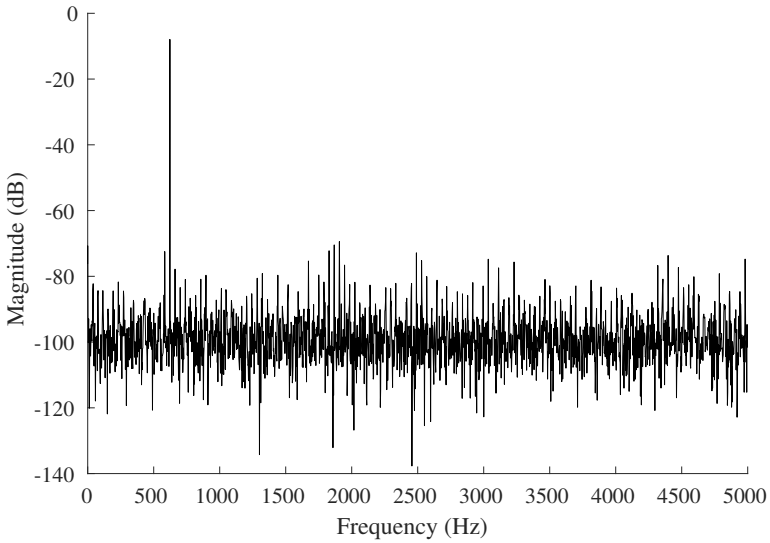


Figure 5.10: Simulated ADC output spectrum using a functional block for the SAR logic, extracted layout views for the comparator and capacitor array, and schematic level circuits for the remaining blocks. The input tone frequency was $\sim$622.6 Hz with an amplitude of 0.4 V. This simulation does not include transient noise. The SNDR was calculated to be 50.9 dB.

Table 5.1: ENOB results from system-level simulations using the FFT test.

FFT Length	Transient Noise	Capacitor Array	Comparator	Logic Block	SNDR	ENOB*	ENOB
4096	No	Schematic	Schematic	Functional	57.4	9.25	9.57
	Yes	Schematic	Schematic	Functional	56.4	9.07	9.39
	No	Layout	Layout	Functional	50.9	8.17	8.49
256	No	Schematic	Schematic	Functional	56.8	9.15	9.47
	No	Layout	Schematic	Functional	49.4	7.91	8.23
	No	Schematic (with parasitic capacitors)	Schematic	Functional	53.3	8.57	8.89

To attempt to explain the degradation in the ENOB, simulations of only 256 points were performed. The results of these are also included in Table 5.1. First we observe that using both the layout and schematic-level comparator and capacitor array, the results agree reasonably well with the 4096 point FFT, so this should be sufficient for determining where performance degrades. It was found that the comparator's layout did not degrade the ENOB, but as shown in the table, the capacitor array's layout causes it to degrade by 1.24 effective bits in the 256 point FFT.

A major cause of this degradation appears to be mismatch in the length of interconnects in the capacitor array's layout, resulting in a commensurate mismatch in the parasitic capacitance that combines with any given capacitor combination. The lengths of the interconnects for each row in the capacitor array were approximated as shown in Table 5.2, with the assumption that the total interconnect length for the capacitors in a given row are roughly the same. Then assuming approximately 0.25 fF/µm on the interconnects, the parasitic capacitance on each set of capacitors was calculated as shown in the table. These values were then added to the schematic of the capacitor array and a 256 point simulation run, where a SNDR of 53.3 dB and ENOB

Table 5.2: Approximate interconnect lengths and parasitic capacitance for the capacitor array.

Capacitors	Approximate Interconnect Length (µm)	Approximate Interconnect Capacitance (fF, assuming $\sim$ 0.25fF/µm)
$C_{0...5}$	973	243
$C_{6...11}$	886	221
$C_{12...18}$	832	208
$C_{19...24}$	899	225

of 8.89 calculated as shown in Table 5.1. This accounts for 0.58 of the 1.24 effective bits lost between the schematic and layout simulations. This is a very rough approximation of the parasitic capacitance, and a more careful treatment should account for more of the degraded ENOB.

The reduction of mismatch in capacitor arrays due to interconnects in an ongoing area of research, and applying techniques such as those proposed in [30] and [31] would help mitigate this issue. As with [3], one advantage of this capacitor array over a typical binary-weighted array is that only the parasitics on the top plates of the capacitors are of concern since there are no switches on the bottom plate. This, combined with the facts that the capacitors are all the same value and the total number of capacitors is much smaller, should help make the reduction of these mismatch parasitics less of a challenge compared to a binary weighted capacitor array.

However, this issue may have been caught earlier by performing simulations on the capacitor array alone after layout. This would have allowed for faster turn-around time during design by avoiding the need to wait for long system-level simulations. In future designs this should be done.

Despite the issue of interconnect mismatch, the results from the schematic-level simulations are promising and indicate that very good performance (i.e. >9 effective bits) is possible using this design. However, it is clear that the ADC is sensitive to mismatch on the capacitor array and any future designs would need extra work to ensure that mismatch in the layout is kept as low as possible. See Chapter 7 for a comparison with published results.

5.3.2 Differential and Integral Nonlinearity

Differential nonlinearity (DNL) and integral nonlinearity (INL) are two useful metrics for measuring the linearity of an ADC [27].

In an ideal ADC, 1 LSB of change in the analog input would correspond to 1 LSB change in the digital output. The DNL measures the deviation of the actual output from this ideal for each output code. A DNL of -1 LSB means there is a "missing code" - i.e. the output of the ADC skips this value entirely. A DNL below -1 means the converter is "nonmonotonic" - meaning the slope of the transfer curve is negative at these points. The INL is the deviation at each point on the transfer curve from an actual straight line. It is a measure of the linearity of the overall transfer curve.

The DNL and INL were simulated here using a histogram test. This is done by applying a ramp voltage to the input of the ADC from slightly below 0 to slightly above full scale. The speed of the ramp is set such that a perfect ADC would measure $N(i)_{theoretical}$ samples for each output code i. The DNL and INL for each point are then calculated as

$$DNL(i) = \frac{N(i)_{actual}}{N(i)_{theoretical}} - 1 \tag{5.4}$$

and

$$INL(i) = \sum_{i=0}^{i} DNL(i), \tag{5.5}$$

where $N(i)_{actual}$ is the number of samples the ADC measures for output code i.

The ramp input for the histogram test was chosen to give $N(i)_{theoretical} = 20$. This should give a DNL resolution of $1/20 = 0.05$ LSB [27]. The ramp time is then:

$$t_{ramp} = \frac{2^{N_{bits}} \times N(i)_{theoretical}}{f_s} = \frac{2^{10} \times 20}{10\,\text{kHz}} = 2.048\ \text{s}. \tag{5.6}$$

Since this is a very long simulation time, it was performed using circuit-level (rather than layout) components, and the functional block of the SAR logic and without transient noise. The results of this test, with calibration every 50 samples, are shown in Figure 5.11. We can see that above output code 985 (corresponding to $V \approx 0.962$), the DNL and INL drop off sharply. This is due to the problem found in the comparator near V_{dd}, as discussed in Section 5.2.2.

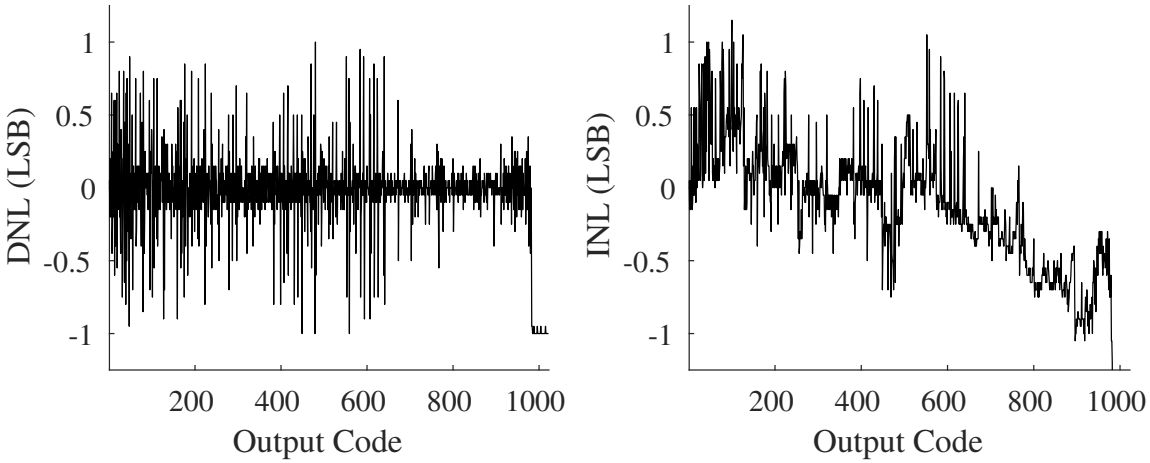


Figure 5.11: Simulated DNL (left) and INL (right) of the ADC.

We can also see that the DNL never goes below -1, so the ADC is monotonic. However, it does reach -1 at 3 output codes, implying that these are missing codes: 449, 479, and 559, corresponding to approximately 0.4385 V, 0.4678 V and 0.5459 V, respectively. The worst case INL is found to be 1.15 LSB.

Note that although the DNL and INL appear visually random, they are in fact deterministic for any given ADC in the absence of noise. When noise is included it will have a small effect on the value for each code.

As with the ENOB, it is likely this relatively poor performance is due to a relative lack of robustness against parasitics on the capacitor array. To check this, the DNL was simulated again using an ideal comparator and a capacitor on its negative input to simulate the parasitic capacitance on the array. The results are shown in Figure 5.12. In order to reduce the time required for these simulations, the resolution was reduced to 0.2 LSB, so the absolute values of DNL will be somewhat inaccurate. However, as it is the trend that interests us here, this should be sufficient.

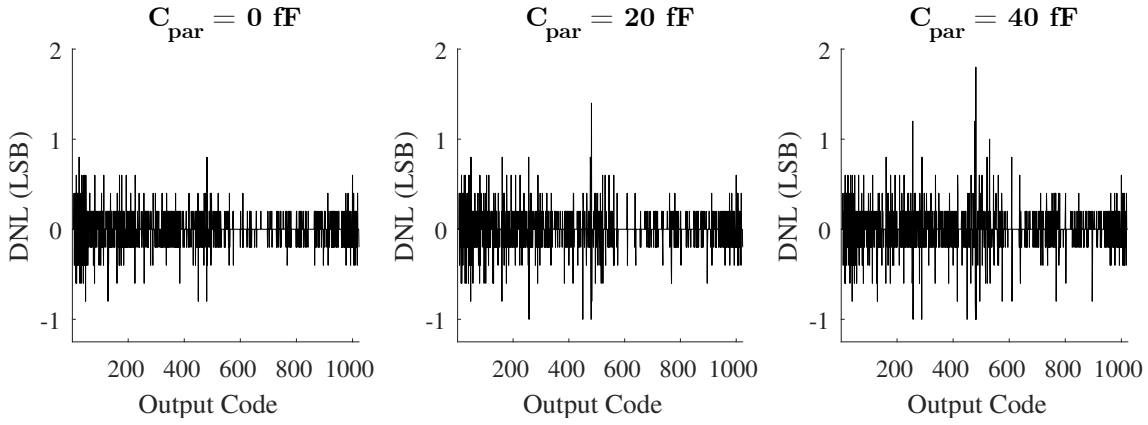


Figure 5.12: DNL results with an ideal comparator for increasing parasitic capacitance. Note that in order to reduce the time required for these simulations, a resolution of 0.2 LSB was used. Maximum DNL from left to right: 0.8, 1.4, 1.8.

We can see that with no parasitic capacitance on the input of the comparator (aside from whatever is built into the array itself), the DNL is reasonably low and the worst case is 0.8 LSB. In general we can see the DNL increases with the parasitic capacitance. The worst case is 1.4 LSB for 20 fF added to the array, and 1.8 LSB for 40 fF added to the array. It will be left to future work to try to improve the

robustness against parasitics.

5.3.3 Power Consumption

The power consumption of the comparator and capacitor array were simulated using their pre-layout schematics and the functional logic block, using the AMS simulator. A linear sweep of a DC input voltage was performed from 0 V to 1 V in 100 steps. For each step, a transient simulation was performed. The mean current for a single conversion in each simulation is plotted in Figure 5.13. Since the supply voltage is 1 V, this corresponds directly to the power.

The power consumption of the logic block was simulated using its pre-layout schematic and the Spectre simulator. The DC input voltage was swept from 0 V to 1 V in only 10 steps due to the complexity of the circuit. Again, for each step, a transient simulation was performed and the mean current calculated for a single conversion. This is also plotted in Figure 5.13.

The mean simulated power consumption for the capacitor array is 372 nW, which is very close to the expected 360 nW as calculated in Section 4.4. The increase we see in the array's power consumption in Figure 5.13 for some inputs is due to the occasional pre-charging of a capacitor in the `a` states of the state machine. For inputs below 0.5 V, capacitors are more likely to need to be discharged to ground which costs no power, which is why the plot is flat for these inputs.

The comparator's mean power consumption is 87.4 nW. If we consider only the input range over which the ENOB simulations were performed (0.1 V to 0.9 V), the comparators mean power consumption is 63.9 nW. We can also see that between 0 V and $\sim$0.7 V, its power consumption is quite flat around 50 nW, then increases as the input approaches 0.95 V, where it drops off again. This corresponds with the behaviour discussed above in Section 5.2.2, where the comparator's delay increases

until it reaches ∼0.95 V, at which point it surpasses half a clock period (which is why we see the power drop off again).

The logic block's mean power is 1.19 µW. The power consumption of the logic block is relatively constant with respect to the input voltage (the maximum deviation from the mean for the simulated measurements is 1.75%). This is expected, as the logic will traverse through the state machine regardless of what the input voltage is, and the only differences will be in the specific combinatorial logic used to determine the next state.

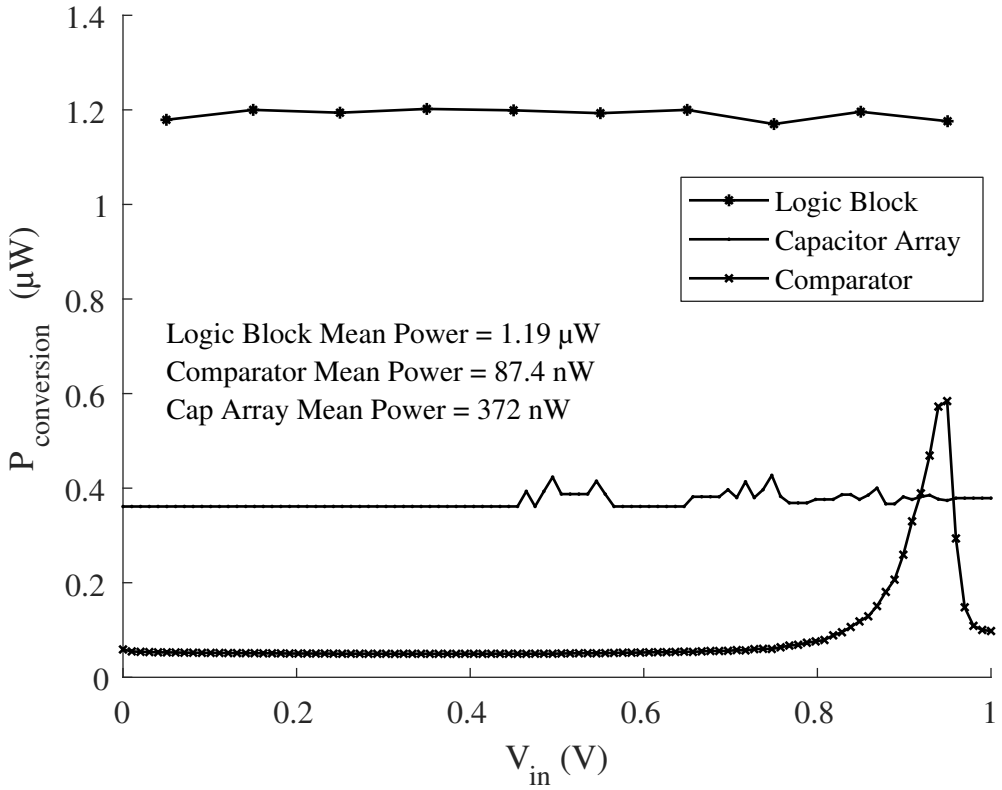


Figure 5.13: Simulated power consumption of the logic block, capacitor array, and comparator vs input voltage. The logic block's power consumption was taken from Spectre simulations, and the capacitor array and comparator's power consumption were taken from AMS simulations with an ideal logic block.

Summing the average power consumption of the logic block, capacitor array, and

comparator, the total power consumption of the ADC comes to ∼1.65 µW.

5.4 Temperature Effects

Note that due to time constraints, all simulations were performed at room temperature. Increasing temperature decreases the threshold voltage and carrier mobility of MOSFETs [4, pp. 293-296]. We would expect most of the chip to be at the same temperature since there are no high power devices on-chip. Since the much of the performance of the ADC depends on the matching of components, it is not likely that changes in temperature would result in significant reductions in performance. However, since ENOB is directly related to the SNDR of the ADC, and thermal noise increases with temperature, we should expect some degradation as the temperature increases.

It is possible that the problem with the speed of the comparator would be exacerbated at high temperatures when devices are slower. In addition, the time to charge the array may increase which could result in errors.

Any future designs should perform simulations at various temperatures before fabrication to ensure the impact on performance is minimal.

5.5 Summary

This chapter went over the simulated performance of the ADC. First, the S/H and comparator were confirmed to have adequate performance for this design, although the comparator does have a problem with delay for input voltages above 950 mV that would need to be addressed in future designs. Then, the ADC was shown to have a simulated ENOB of 9.39 at the schematic level with transient noise included.

Worst case DNL and INL were found to be -1 and +1.15 LSB, respectively. These results confirm that the algorithm presented in Chapter 3 produce a functional ADC, despite the issues found with lack of robustness to parasitic capacitance. Finally, the total simulated power consumption of the ADC was found to be about 1.65 µW on average.

Chapter 6 discusses the testing and debugging of the fabricated circuit, and the flaw found in the design, to which a solution is proposed.

Chapter 6

Testing and Debugging

The ADC was fabricated in a 130 nm CMOS process by Global Foundries. Testing was performed at Carleton University to verify the operation of the chip. Unfortunately, due to a design flaw the device did not perform as expected. This chapter will discuss the measurements taken and the reason for the problem.

Micrographs of the fabricated die are shown in Figures 6.1. The pattern fill is clearly visible around the sides of the chip, which was included due to pattern density requirements on the top three metal layers. In the center of the image we see the capacitor array. The other visible capacitors near the top are the S/H capacitor, and the calibration capacitors for the comparators (both the one in the device and the second one added for testing purposes).

6.1 PCB and Test Setup

The dice were packaged in 44-pin Quad Flat Pack (QFP) packages, and soldered onto a custom PCB as shown in Figure 6.2. The PCB was designed in Altium Designer

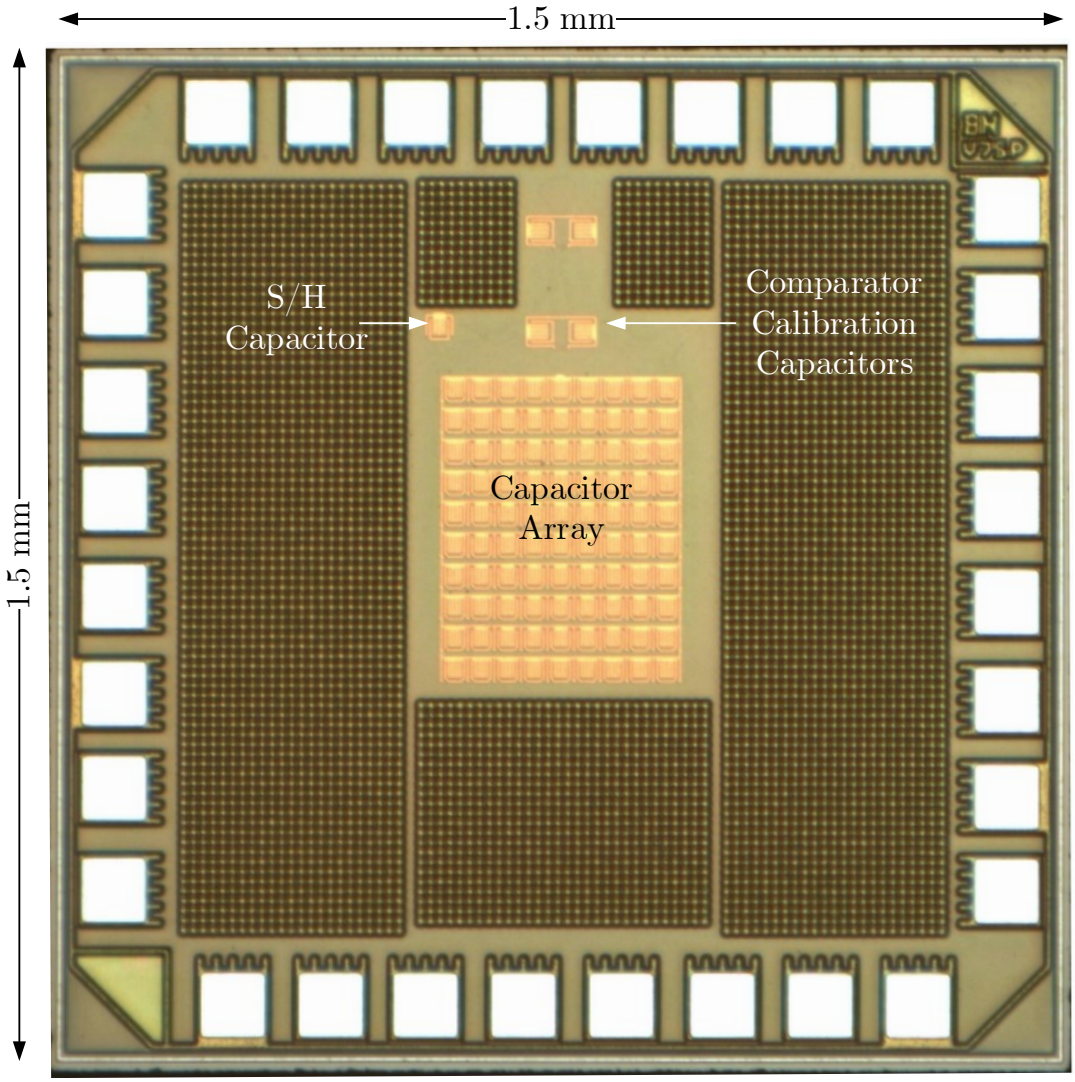


Figure 6.1: Micrograph of the die, fabricated in the 130 nm process by Global Foundries. Note that pattern fill was required on the top three metal layers. Aside from the pattern fill, capacitors are the only distinguishable components.

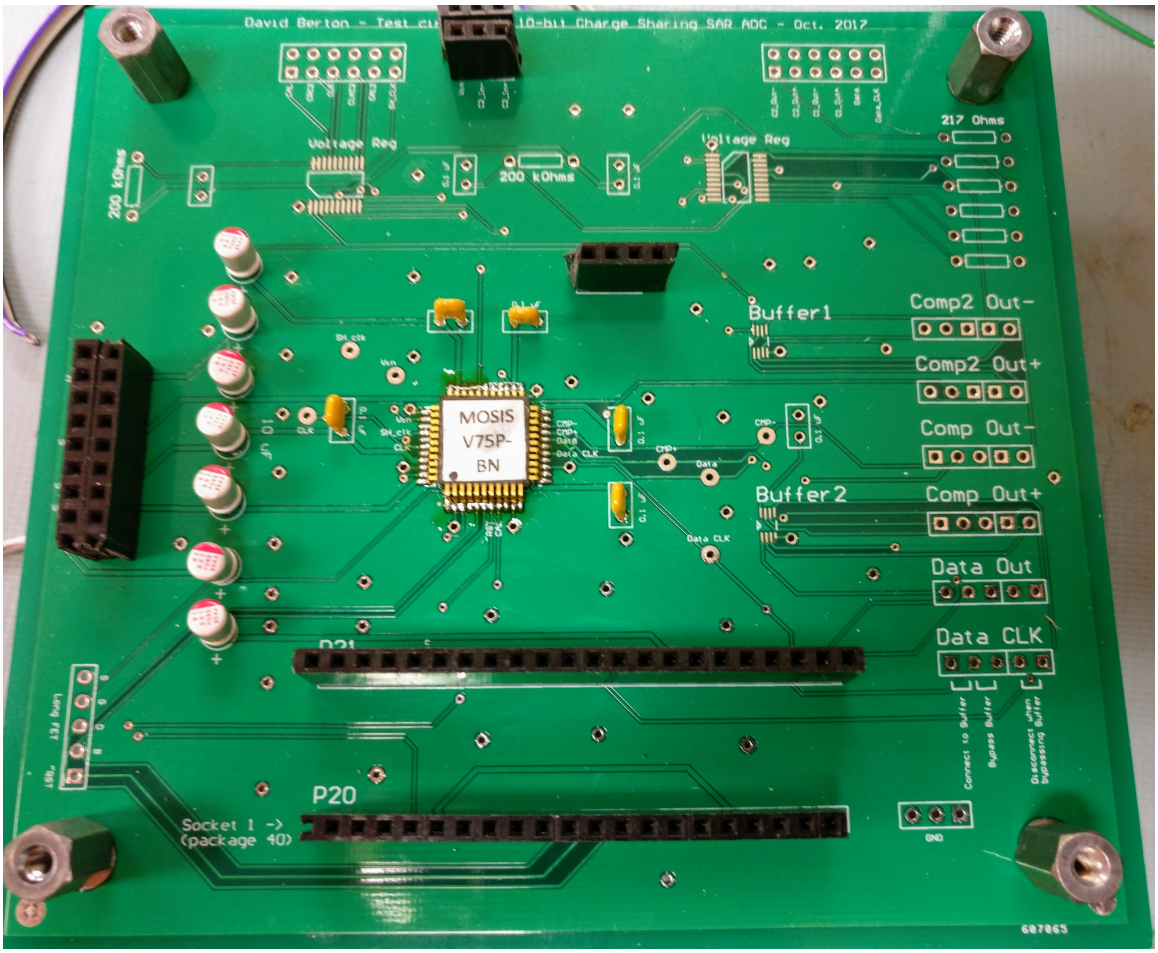


Figure 6.2: The packaged die on a custom PCB. Note that many components were never populated on the PCB since preliminary testing showed the chip was not operating as expected.

and its schematic and layout are included in Appendix C. The decoupling capacitors used on the power pins are 0.1 µF near the chip, and 10 µF near the headers. Note that since preliminary testing showed the device was not working properly, only the decoupling capacitors and some of the headers for external connections were populated on the board.

Figure 6.3 shows a block diagram of the test setup. The device was powered using an Anker PowerCore 20100 rechargeable battery converted from 5 V to 1.2 V using a Texas Instruments LM317 voltage regulator. Note that 1.2 V was used instead of 1

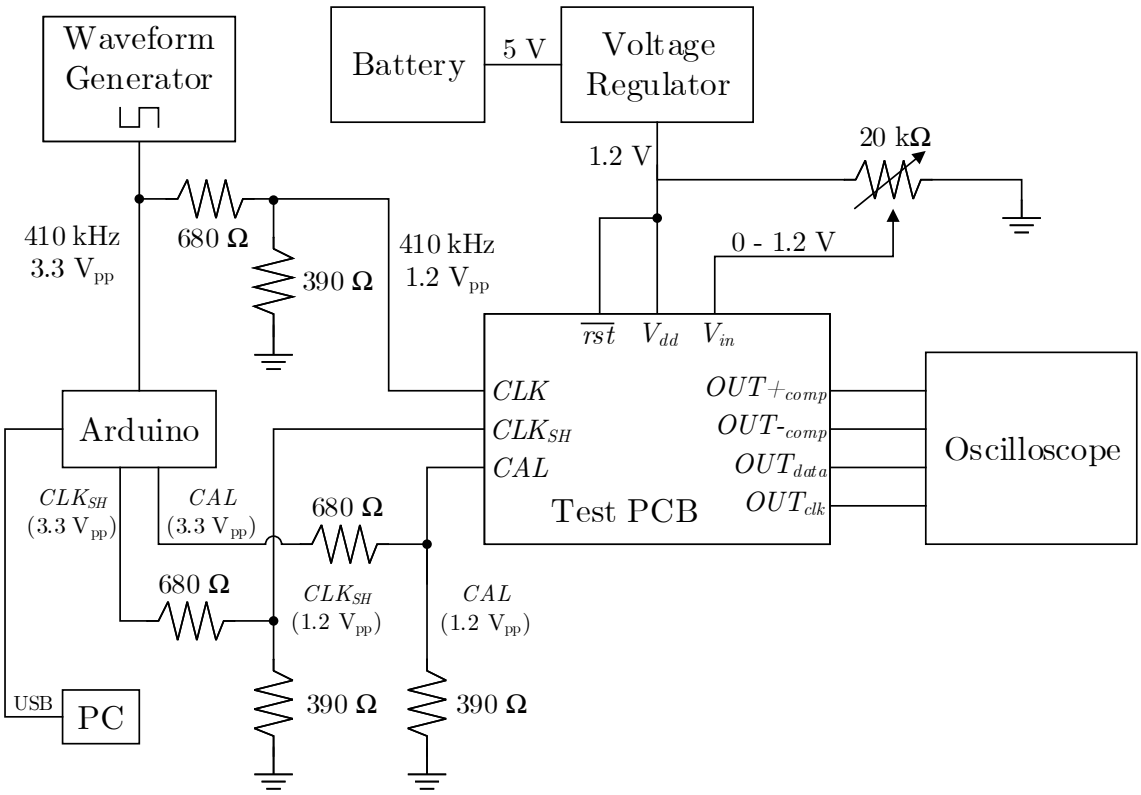


Figure 6.3: Block diagram of the test setup.

V (as per simulations) simply because the LM317 was readily available and its lower limit for regulation is 1.2 V. The transistors used in this design can easily handle a 1.2 V power supply. A battery was used here because initially it was suspected that noise in the external power supply might be an issue.

An Agilent 33250A Arbitrary Waveform Generator was used to generate the 410 kHz clock. This was fed into an Arduino Due to generate the 10 kHz sampling clock and calibration signal. Since the Arduino Due operates at 3.3 V, voltage dividers were used to convert these signals to 1.2 V. The clock signals fed to the board are shown in Figure 6.4(a). An Agilent DSO5054A Oscilloscope was used for all AC measurements and a multimeter for DC measurements.

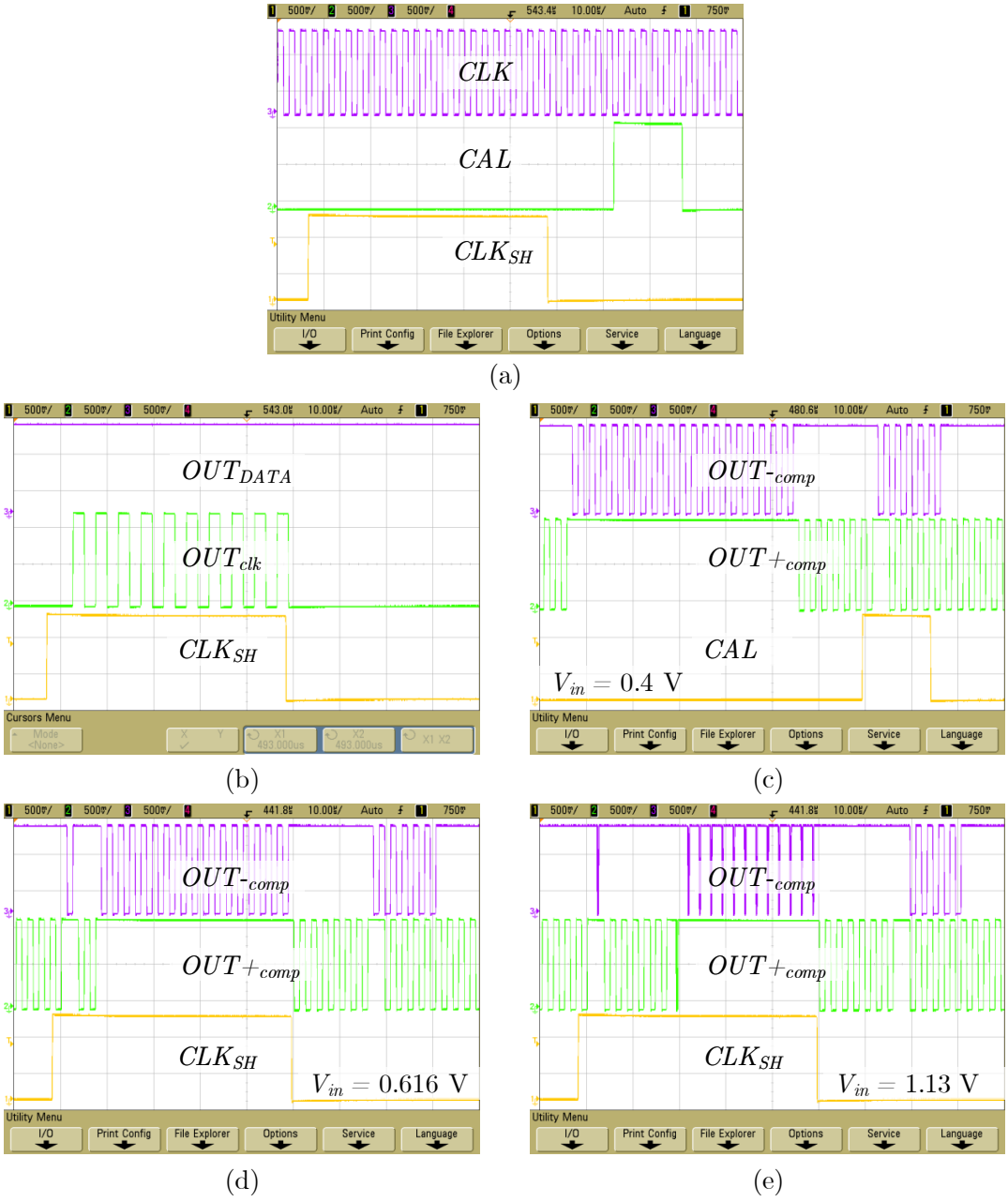


Figure 6.4: Oscilloscope screenshots during testing and debugging of the ADC.

6.2 DC Input Measurements

All the following measurements were taken using a DC input voltage, which was generated using a voltage divider from the power supply and controlled using a 20 kΩ potentiometer.

It was found that independent of the input voltage, the data output stays at the upper rail of 1.2 V while the device is powered. However, the data clock (whose falling edge should indicate valid data) appears to work correctly, ticking ten times during each sample period (once per bit). These are both shown in Figure 6.4(b). Since both of these signals are generated from the digital logic block, we can conclude that this block must be functioning correctly at least in part. In addition, during some states that do not represent normal operation (for example, with power to the comparator disconnected), some switching was observed on the data output. Therefore we can also conclude that the output drivers for this signal are functioning.

The comparator outputs were also measured. First, we observe that during the calibration cycle, the comparator outputs appear to act correctly, as shown in Figure 6.4(c). The external calibration signal lasts six clock periods and is fed to the digital block, which then generates two internal signals which are sent to the comparator. During the first clock period during calibration the array should be charged to V_{dd}, which is why we see $OUT+_{comp}$ stay high and $OUT-_{comp}$ fall. During the next five clock periods, the calibration is taking place as discussed in Section 4.3.2, so both comparator outputs fall when the clock rises. Again, since this behaviour is controlled by the logic block, we can conclude that it is at least partly functional.

For every DC input voltage below $\sim\frac{1}{2}V_{dd} = 0.61$ V[1], $OUT+_{comp}$ falls and

[1] V_{dd} was actually measured to be approximately 1.22 V.

$OUT-_{comp}$ stays high during the sampling period. This can also be seen in Figure 6.4(c), which was taken at an input voltage of 0.4 V.

Figure 6.4(d) shows the comparator outputs during the sampling period with a DC input of 0.616 V. Here we can see that $OUT-_{comp}$ goes low and $OUT+_{comp}$ goes high for two clock periods. Similar behaviour is seen as the input voltage is increased. At $V_{in} = 0.914$ V, $OUT-_{comp}$ falls for four clock periods, at $V_{in} = 1.059$ V it falls for six, and at $V_{in} = 1.132$ V it falls for seven.

In addition, as V_{in} approaches V_{dd}, the comparator slows down significantly as shown in Figure 6.4(e), which shows the comparator outputs with $V_{in} = 1.132$ V. This behaviour is not unexpected, as discussed in Section 5.2.2.

The data output should be a delayed copy of $OUT-_{comp}$ that is held constant for two clock periods. Therefore every other decision on the negative comparator output during the sampling period (starting on the third) represents one bit of the output.

From this we can determine that approximately below $V_{in} = \frac{1}{2}V_{dd}$, the "output" as read from $OUT-_{comp}$ is 0000000000_b. When V_{in} surpasses $\frac{1}{2}V_{dd}$, the output is 1000000000b. When it surpasses $\frac{3}{4}V_{dd}$, the output is 1100000000_b. Above $\frac{7}{8}V_{dd}$ the output is 1110000000b. This trend continues until the comparator becomes too slow to operate. However, when the supply on the capacitor array alone was reduced from 1.2 V to 1 V, it was found this trend was identical, but happened around 0.5 V, 0.75 V, etc. instead of 0.6 V, 0.9 V, etc. In addition, the trend continued until the input reached 1 V and the output read from $OUT-_{comp}$ was 1111111111_b.

From these observations we can conclude that the logic block is always reading a digital 1 from the comparator, even when its decision is a 0. This explains why the data output is always a 1, and why the voltage on the capacitor array appears to always be increasing (as shown by the pattern of changes on the comparator output).

The most likely reason for the logic always reading a 1 from the comparator is

due to a design flaw. The state machine samples the comparator output on the same clock edge on which it is reset. When there is no delay in the state machine logic (i.e. when a functional block is used in simulations) this is not a problem, as the logic will sample the comparator output before it is reset. However, further investigative simulations found that only a small amount of delay was required to cause the same behaviour as was found in testing.

Figure 6.5 shows the result of a simulation with a DC input voltage of 510 mV (and a 1 V supply) with a 215 fF capacitor connected to the clock line fed to the ideal logic block after a buffer. This amounts to a ∼810 pS delay on the clock (Figure 6.6), which is sufficient to cause the comparator output to be sampled after it is reset. This in turn causes the voltage on the capacitor array to always increase regardless of what the comparator's output is. The comparator outputs here are identical to those in Figure 6.4(d). The data output also stays at 1, as seen in Figure 6.4(b)[2].

This error was not caught before fabrication because most of the verification simulations were performed with the ideal logic block with no delay modeled. The few simulations with the schematic-level logic block that were performed to verify operation also did not catch this issue. It is possible that the delay caused by gate capacitance of the schematic-level logic block was not enough to cause it to miss the comparator output. However, interconnect capacitance may add enough to delay to cause the problem.

As a quick check of this theory, a simulation was performed of a single inverter driving 40 inverters in parallel, which is roughly on the order of the number of gates the clock is driving. The delay from the falling edge of the input to the rising edge of the output is about 500 ps, not enough to sample the comparator after it has reset.

[2]Note that the only reason we see it step from 0 to 1 here is that the simulation shows the first conversion and its initial state is 0 - after this it stays at 1 permanently.

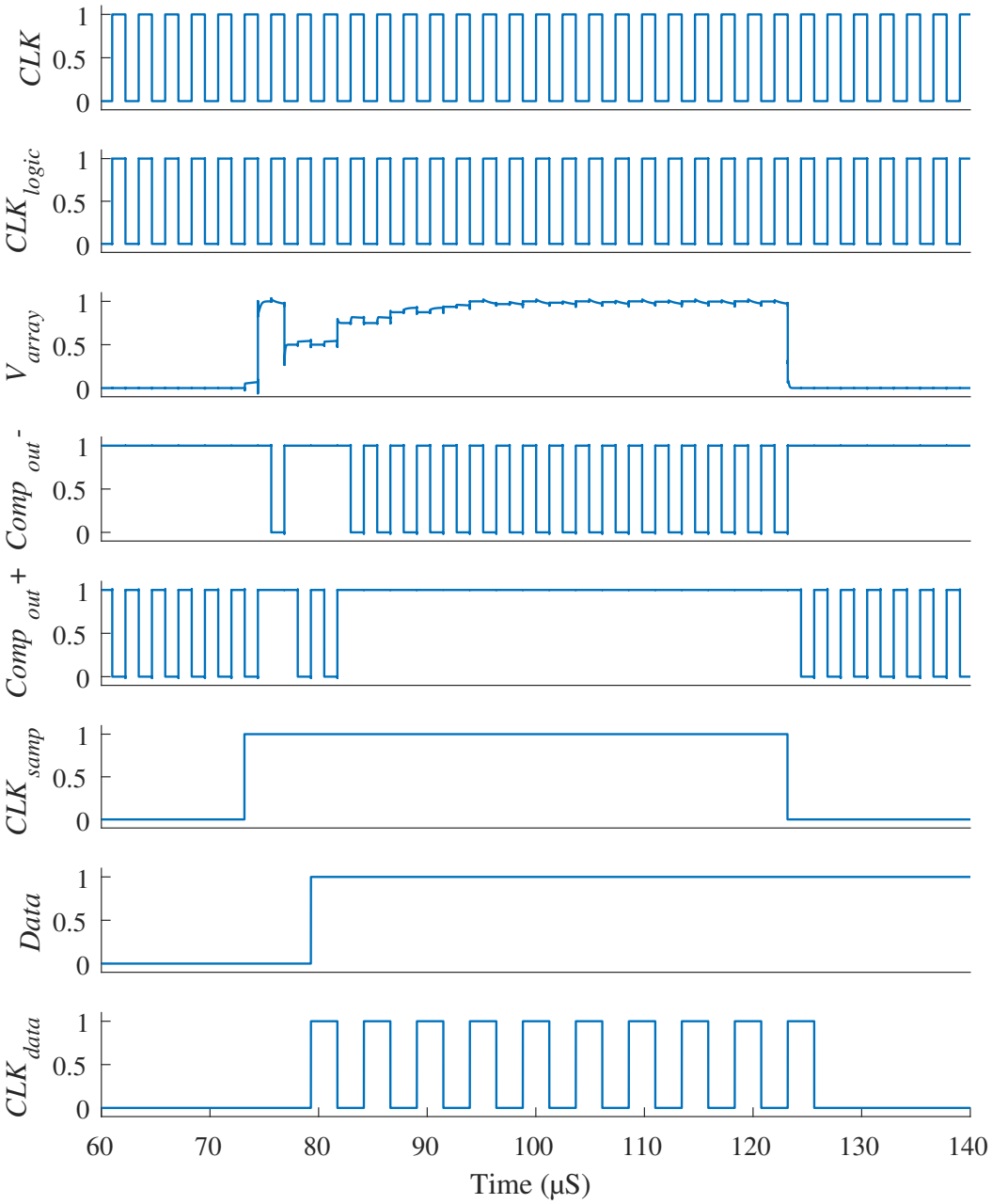


Figure 6.5: Simulation results of ADC operation with 215 fF added to the clock fed to the SAR logic, resulting in 810 pS delay on its clock.

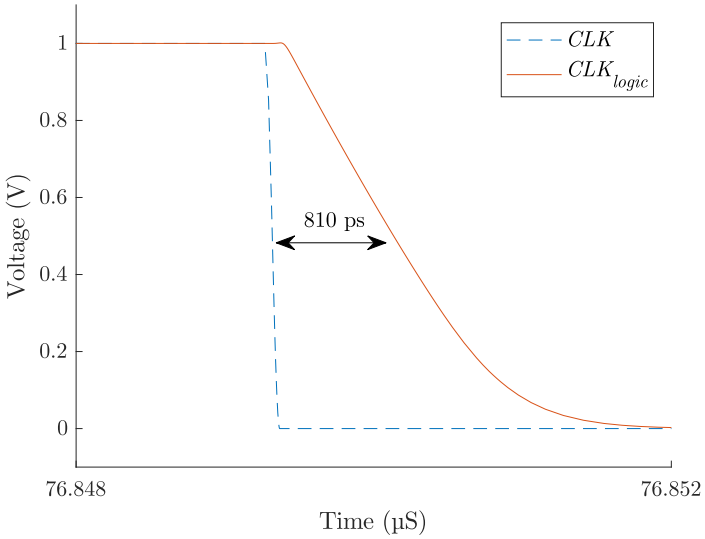


Figure 6.6: The delay on the logic clock with a 215 fF load.

The parasitic capacitance of metal layers M1, M2, and M3 is on the order of 0.25 fF/µm. Even if we estimate only 100 µm of interconnect and therefore ∼25 fF of capacitance (an underestimation given the dimensions of the logic are 800 µm × 320 µm), the delay ends up well over 800 ps, which is more than enough to cause the logic to sample the comparator after it has reset.

Due to the size of the logic block, post-layout simulations of the digital block were not performed. It is possible that had they been, this problem would have been found sooner. More realistic delay being built into the functional logic block also would have helped to find the issue.

Since the comparisons only happen every other clock cycle, the simplest way to solve this problem is to halve the frequency of the clock to the comparator. In order to simulate this, a behavioural block was generated to toggle the comparator's clock on the rising edges of the logic clock, and only when the sample clock is high. In reality this function could be added to the state machine. Figure 6.7 shows the results of a simulation with the logic-controlled comparator clock and input voltage of 510

mV. The 215 fF capacitor is still present on the logic clock here, but we see that the signals now act as expected. V_{array} approaches 510 mV, and the data output gives $1000001000_2 = 520_{10}$ which corresponds to 507.8 mV.

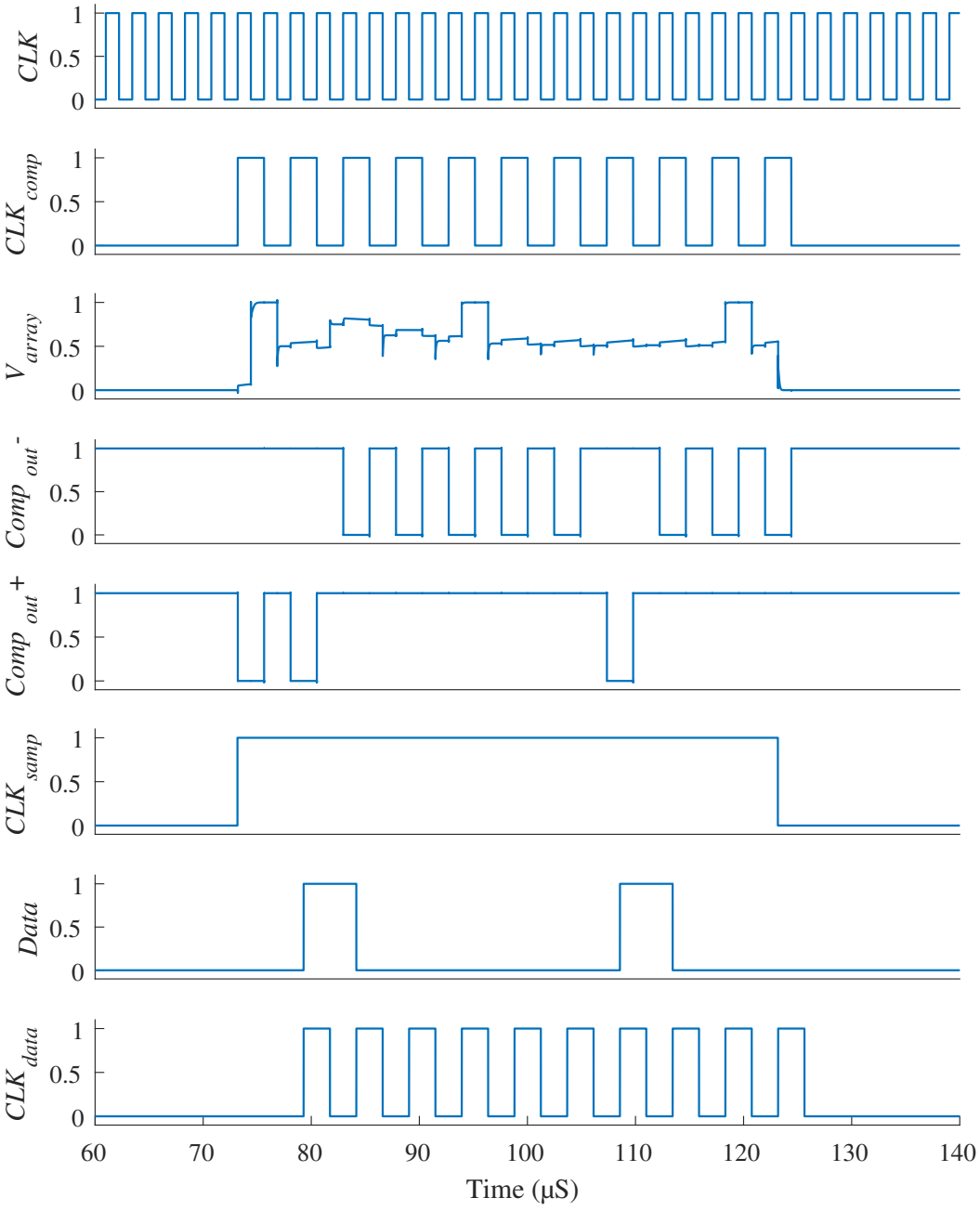


Figure 6.7: The results of a simulation with the comparator clock controlled by the logic.

6.3 Comparator M_5 Verification

The comparator's gain stage tail transistor, M_5, has a length of 5 µm and a width of 0.5 µm. Because this is an unusual aspect ratio, there was an initial concern that its models might be inaccurate and therefore that it might cause the comparator to not function properly. To check that this was not a problem, a copy of this device was placed on the fabricated chip so it could be verified independently.

The measurements were performed with $V_s = 1.2\,\mathrm{V}$ and a 1 MΩ drain resistor as shown in Figure 6.8. 1 MΩ was used here in order to get a voltage variation in the range of 1 V, since the device is only capable of sourcing currents in the range of 1 µA. Then the gate voltage was swept and the drain voltage measured with a multimeter (with 40 MΩ output impedance, so its effect on the measurement is minimal). Note that the same battery described in the general test setup was used here, and the gate voltage generated using a 20 kΩ potentiometer as a voltage divider.

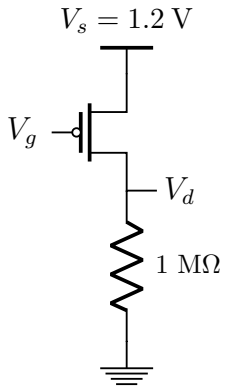


Figure 6.8: Test setup for the test copy of the comparator's tail transistor.

The results of this sweep are shown in Figure 6.9 and compared to simulated results. We can see that the measurements match the simulation reasonably well. Even the difference found here is of little concern because M_5 in the comparator is

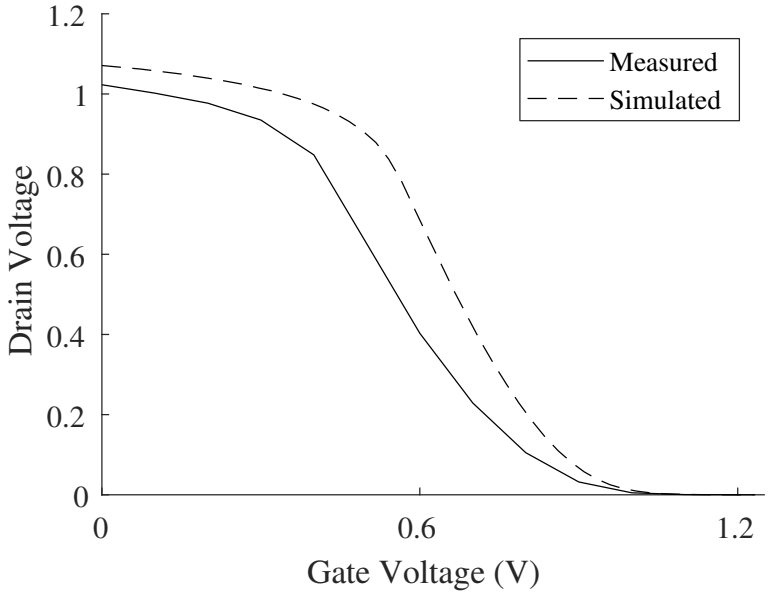


Figure 6.9: Measured vs. simulated comparator tail transistor.

switched by a rail-to-rail gate voltage and it is clear that the test device turns on and off completely. Therefore there is no reason to suspect that this device causes the comparator or the ADC as a whole to not function.

6.4 Summary

This chapter went over the test setup of the fabricated ADC. A design flaw was found in testing that caused the chip to not function properly because the logic always samples a 1 on the comparator outputs. In order to fix this problem, the clock controlling the comparator could be generated by the logic block to operate at half the speed of the logic. The next chapter will conclude this thesis, compare it to other designs in literature, and propose ideas for future work on this topic.

Chapter 7

Conclusion

In this work, a recursive backtracking algorithm was developed to reduce number of unit capacitors a SAR ADC requires for a state machine-based switching scheme. This is the only instance known to the author of such an algorithm being used to design a finite state machine for an ADC. The algorithm was used to design a 10-bit SAR ADC which was fabricated in a 130 nm CMOS process.

The ADC requires 25 unit capacitors compared to the 64 that would be required for the scheme proposed in [3] or the 1024 that would be required in a binary-weighted SAR ADC. The ADC also requires only one switch per capacitor compared to three required in [3] and [10]. The reduction in the number of switches per capacitor reduces the number of control signals coming from the state machine, which prevents the state machine from becoming more complex.

The SAR ADC's schematic-level simulations show promising results, with an ENOB of 9.39 in simulations including noise. The worst case INL and DNL in simulations are 1.15 and 1.0, respectively, and the post-layout simulated ENOB is 8.49. Investigations determined that an issue with this switching scheme is a sensitivity to

Table 7.1: Performance summary and comparison of simulated results to published measurement results. Note that FOM for the post-layout is calculated assuming power consumption is not affected by layout.

	[7]	[8]	[9]	[20]	This Work (Simulation Only)	
					Pre-Layout	Post-Layout
Year	2016	2012	2016	2017	2018	
Resolution (bits)	11	10	12	15	10	
Technology	180 nm	130 nm	65 nm	40 nm	130 nm	
Fs (kS/s)	40	1	40	20	10	
Capacitor Array	Binary/Thermometer-Weighted	Binary-Weighted	Binary/Thermometer-Weighted	Split-Array	Unit Capacitor Array	
Active Area	0.195 mm^2	0.191 mm^2	0.105 mm^2	0.315 mm^2	0.458 mm^2	
ENOB	10.23	9.1	10.37	12.02	9.39	8.49
INL (LSB)	-0.76 / +0.72	-0.46 / +0.45	-0.6 / +0.8	5.2 (max)	-1.05 / +1.15	-
DNL (LSB)	-0.49 / +0.48	-0.61 / +0.45	-0.6 / +0.4	1.9 (max)	-1 / +1	-
Power Consumption	1.8 µW	53 nW	0.375 µW	1.17 µW	1.65 µW	-
FOM (fJ/conv-step)	37	94.5	7.1	14.1	246	459

parasitic capacitance and mismatch on the capacitor array. Improving robustness to parasitics will be left as future work (see Section 7.4 for suggested ways of doing this).

Table 7.1 summarizes these results and compares them to some recent published SAR ADCs of similar speed and resolution. Note that this table compares the simulated results of this work to measured publication results, so the comparison is not perfect; worse results should generally be expected from a fabricated chip than the simulation. The figure of merit (FOM) used here is widely used, and normalizes the power consumption, P, to the conversion rate, f_s and ENOB [32]:

$$FOM = \frac{P}{2^{ENOB} \times f_s}. \tag{7.1}$$

We can see that the performance, chip area, and power consumption of the device are all somewhat comparable, if generally worse than the published results.

Unfortunately a second device could not be fabricated for this thesis due to time constraints. Preparing a design for fabrication can take several months in addition to

approximately six months to fabricate the device. However, the correction of the error in the design that caused the fabricated chip to be nonfunctional and the following improvements to the digital circuitry and capacitor array could be implemented in future designs.

7.1 Possible Improvements in the Digital Circuitry

Since the digital block takes up over 50% of the area, and over 70% of the power consumption, there is significant room for improvement in these two metrics. For example, [8] reduces power consumption by using a 1 V supply for the analog blocks, and a 0.4 V supply for the digital blocks, using level shifters to interface between them. Taking the overhead of the level shifters into account a ∼50% improvement in power consumption was achieved. If a similar strategy were used here, it would require about 34 level shifters. Assuming the same power consumption as [8] and linear scaling with frequency, the level shifters would require 370 nW of power. Assuming the control logic's power consumption would scale with the square of the power supply voltage, it would consume 190 nW with a 0.4 V supply. The total power consumption of the logic would then be 560 nW, which is a 53% reduction.

In addition, a limited standard cell library was used to synthesize the digital block. Use of a more optimized or complex library with higher drive strengths and gates with more than two inputs would likely result in significant improvements in both area and power consumption [33][34]. In addition, the fact that most of the area and power consumption are taken up by digital circuitry suggests that this ADC would scale better with respect to process nodes than more analog-heavy designs.

To verify this, the logic was synthesized in a sample standard cell library in a

generic 90 nm process. Using this, Synopsys Design Vision estimated the area of the block to be 0.0156 mm^2. With the simple standard cell library in the 130 nm process used here, Design Vision estimated the area to be 0.0759 mm^2 (the final area was made much larger to make the place and route process easier). This amounts to an 80% reduction in area if compared directly. If we assume the total area scales with the square of the gate length, this implies we could expect an area reduction of 57% in the 130 nm process with a more robust library, which would mean a digital area of 0.146 mm^2. This area reduction does not account for the extra area savings in the routing of the digital block. The logic synthesized in the custom 130 nm standard cell library has 6659 nets, and when synthesized in the 90 nm library it has 3583 nets. Fewer nets means much less wiring between cells, which further reduces the required size.

Design Vision also estimates a dynamic power consumption of 590 nW in the 90 nm process. Assuming the same current density, we would expect the power consumption to be 852 nW in a 130 nm process, which is a 28% reduction in power consumption. Assuming a lower supply voltage could be used as discussed above, we could expect a total power consumption for the digital block of 506 nW.

7.2 Possible Improvements in the Capacitor Array

Because the switching of the capacitors does not follow a regular pattern, performing a mathematical analysis on the effects of parasitics, leakage, and mismatch in the capacitors is infeasible. Instead, a function was written in Matlab that takes the output of the algorithm from Section 3.1 (i.e. the list of which capacitors to connect together when), and uses this to simulate the operation of the ADC. The function

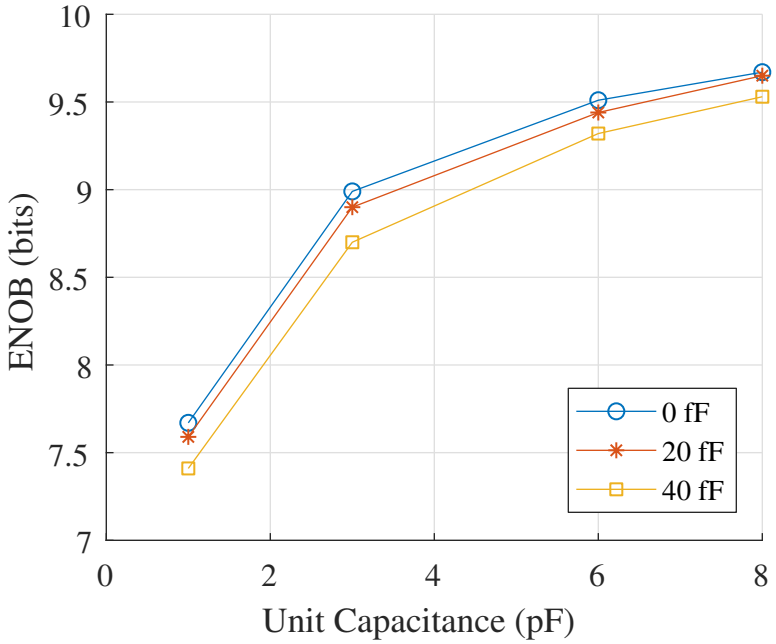


Figure 7.1: Sweep of the capacitor array's unit capacitance for different amounts of parasitic capacitance using the script that simulates the ADC's theoretical behaviour. The ENOB can be compared to Figure 4.17 to confirm a reasonable degree of agreement between the script and Cadence simulations.

follows a loop similar to that shown in Figure 3.4, looping through each bit, averaging the voltages on the capacitors as determined by the backtracking algorithm, performing the comparison to the input voltage, and recording the result. Then, the effects of parasitic capacitance, leakage through the switches, and capacitor mismatch are included. By passing a sine wave or a ramp as the input to this function, the ENOB, DNL, and INL can be calculated. The script is provided in Appendix D.

First, to validate that the script gives reasonably accurate results, the sweep of unit capacitance and parasitic capacitance on the comparator's negative input, as performed in Section 4.4 and plotted in Figure 4.17, was reproduced. The switch off resistance is set to 8.9 GΩ so the leakage can be estimated. The results are shown in Figure 7.1, and we can see that they align closely with the results in Figure 4.17.

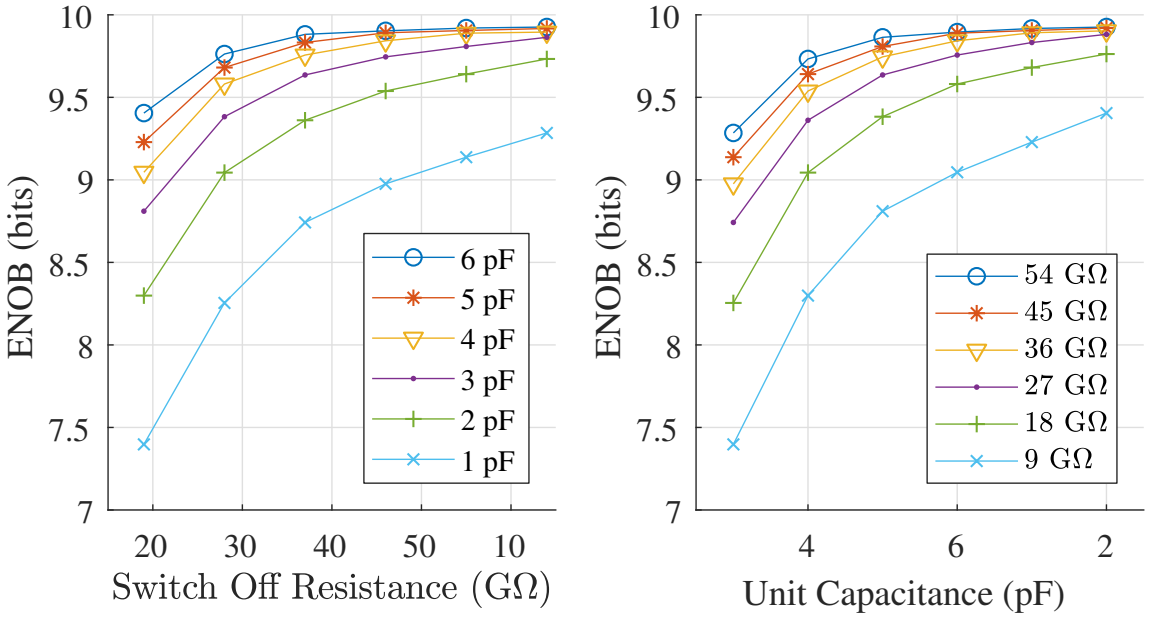


Figure 7.2: Expected effective number of bits for the ADC, sweeping the estimated off resistance of the capacitor array's switches and the size of the unit capacitors, using the Matlab script that simulates the ADC's theoretical behaviour. The array's parasitic capacitance is set to 10 fF for this sweep.

Then, the off resistance of the switches is swept for different values of unit capacitance, as shown in Figure 7.2. The parasitic capacitance on the array was set to 10 fF for this sweep. We can see that, for example, if the switch off resistance were increased to 45 GΩ, the size unit capacitors could be reduced from 6 pF to 2 pF while still expecting an ENOB of 9.6 bits.

The DNL and INL predicted by the script with 2 pF unit capacitors, 45 GΩ switch off resistance, and 10 fF of parasitic capacitance are shown in Figure 7.3. We can see that the DNL and INL are predicted to be within ±1 LSB.

Future work could look into what effect the precise switching scheme has on performance. For example, Figure 7.4 shows the predicted DNL and INL using the same settings as the results in Figure 7.3, but with slightly different outputs from the backtracking algorithm that results in a different solution to the switching scheme. We

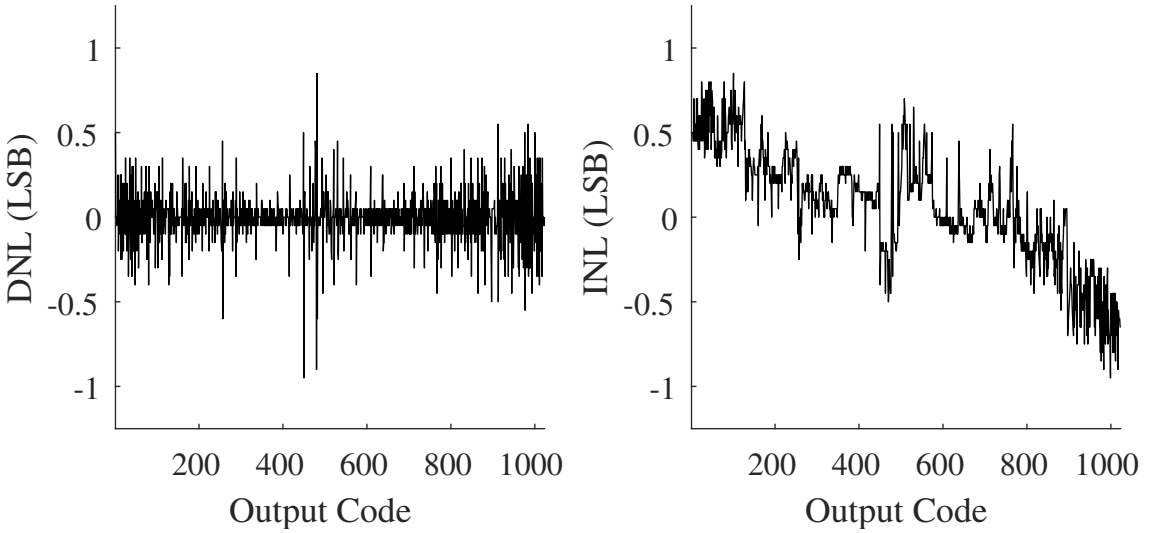


Figure 7.3: DNL (left) and INL (right) predicted by the Matlab script that simulates the ADC's theoretical behaviour. The array's parasitic capacitance is set to 10 fF, the unit capacitance is set to 2 pF, and the off resistance of the switches is set to 45 GΩ.

can see that there is some effect on the expected DNL and INL. Further investigations will be required to determine the solution that gives the best performance while keeping the capacitors and total layout area small.

Combining the possible improvements to the digital circuit and the capacitor array would put the total active area of the ADC at 0.217 mm^2 and the power consumption at 727 nW. This would make the results comparable with [8]. However, in [8], the digital circuitry already occupies a small portion of the total layout area. Even given the reductions discussed here, the digital circuitry would still be 69% of the total layout area and 70% of the power consumption. We therefore expect the total layout area and power consumption of the work presented here to scale well with decreasing process nodes.

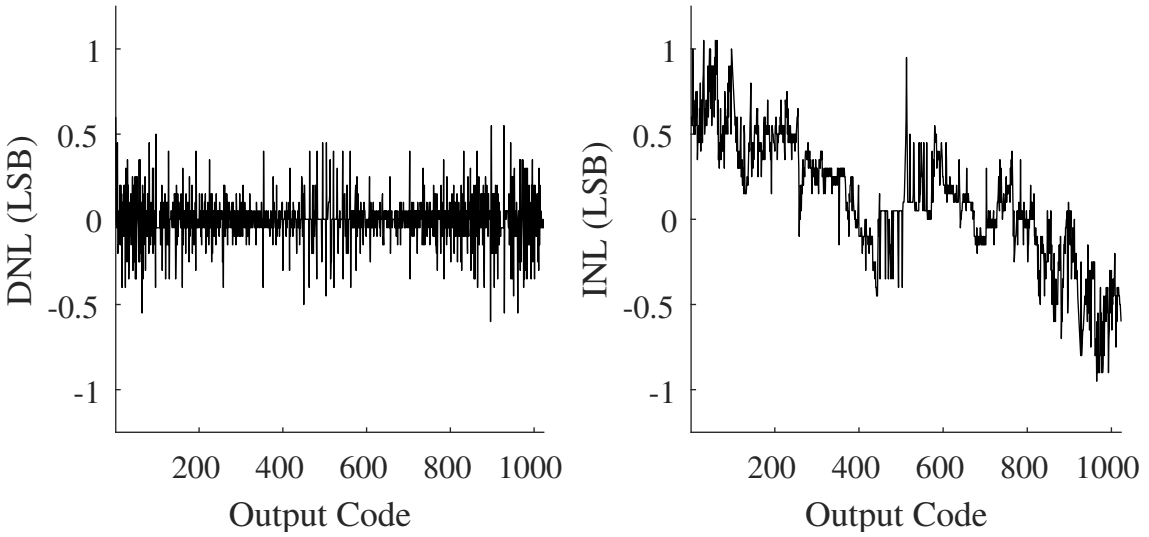


Figure 7.4: DNL (left) and INL (right) predicted by the Matlab script that simulates the ADC's theoretical behaviour, with a modified decision tree. The array's parasitic capacitance is set to 10 fF, the unit capacitance is set to 2 pF, and the off resistance of the switches is set to 45 GΩ.

7.3 Advantages With Process Scaling

In order to make a comparison with other published ADCs as technology scales, several assumptions will be made. All of these assumptions are simplifications, but help give an idea of how well we could expect a circuit to scale. First, we will assume that digital layout area scales with the square of the ratios of the process size (e.g. the scaling factor from 130 nm to 90 nm would be $(90/130)^2 = 0.48$). Second, we will assume that the layout area of the digital circuitry is the only thing that scales with process. We will also assume that the improvements discussed above to bring the layout area of the logic to 0.146 mm^2 and the capacitor array to 0.065 mm^2 in the 130 nm process can be incorporated (the areas of the comparator and S/H are negligible and will be ignored for this discussion). Finally, we will assume that the number of capacitors required by the algorithm presented in this thesis scales linearly after 10 bits. Using these assumptions, we will compare the expected layout area to

[20].

First we will use these assumptions to scale the design presented in this thesis to 15 bits in a 40 nm process. 60 capacitors would be required for 15 bits in this case, giving an area for the capacitor array of $0.065\ \text{mm}^2 \times 60/25 = 0.156\ \text{mm}^2$. The state machine requires twice as many states for each added bit, so its area should scale as well: $0.146\ \text{mm}^2 \times 2^5 = 4.672\ \text{mm}^2$. Then accounting for process scaling the area of the state machine should be $4.672\ \text{mm}^2 \times (40/130)^2 = 0.442\ \text{mm}^2$. This gives a total area of $0.146\ \text{mm}^2 + 0.442\ \text{mm}^2 = 0.588\ \text{mm}^2$, which is still almost twice the area reported in [20].

However, the advantages start to show themselves as we scale further. If we scale the above to 7 nm, we get the following total area: $0.146\ \text{mm}^2 + (7/40)^2 \times 0.588\ \text{mm}^2 = 0.164\ \text{mm}^2$. In [20], the logic appears to occupy about 1/3 of the layout area. So using our assumptions, we would expect that when implemented in a 7 nm process the total layout area would be $(2/3) \times 0.315\ \text{mm}^2 + (7/40)^2 \times (1/3) \times 0.315\ \text{mm}^2 = 0.220\ \text{mm}^2$, which is 34% larger than what we expect using the algorithm designed here. This trend will naturally continue as technology nodes optimized for digital circuits continue to shrink.

7.4 Future Work

In addition to the possible improvements to the area and power consumption of the digital block and the reduction in the size of the capacitor array that would be possible with increased switch resistance as mentioned above, there are several areas in which the algorithm and the design of the ADC could be improved.

7.4.1 Future Work on Circuit Implementation

Most notably, the error that prevented the fabricated device from working could be rectified. As shown in Chapter 6, simple logic could be added that clocks the comparator at half the logic block's clock rate when the sampling clock or calibration signal is high. This works because the comparisons only need to be performed in every other state.

Any future designs should also make sure to design for testability. There are several examples of how this could be done. As a first test prior to fabricating the chip, the logic could be tested using an FPGA and surface mount components on a PCB. In addition, as many internal signals on the chip as possible should be accessible when testing. Having the option to supply the signals controlling the switches in the capacitor array from an external source (and supplying them through a shift register to minimize the number of pins required) would allow that block to be tested independent of whether the logic worked. Similarly, being able to test the logic independently from the other blocks would be useful.

The comparator's speed is another problem with several possible solutions. The current in its tail transistor was kept low in order to improve the calibration scheme, however this is likely the limiting factor for its speed. Therefore, either a different calibration scheme could be used, or an attempt could be made to increase the current and maintain the calibration performance in another way. Alternatively, it might be possible to modify the design of the ADC to subtract the binary-weighted voltages from the input voltage in the same way as [5] while still applying the algorithm designed here. This would remove the need for rail-to-rail inputs on the comparator, as its voltage would always be compared to ground.

Parasitics on the capacitor array were found to significantly degrade the ENOB,

INL, and DNL. An important consideration for future designs is that the capacitor array layout should be generated in such a way as to minimize the mismatch between interconnects as much as possible. In addition, a more detailed analysis of the effects of parasitics could be performed in order to optimize capacitor size in terms of area, power consumption, and performance. The script discussed in Section 7.2 could be used to help with this analysis and the effects of mismatch in the capacitors as well as other non-idealities could be added to it to help with the design process. Finally, the algorithm which determines the switching scheme could be modified to help improve resistance to parasitics, as will be discussed next.

7.4.2 Future Work on the Backtracking Algorithm

As mentioned in [3], the error on the MSB comparisons dominates the overall error of each conversion. In that work, it was found that the unit capacitance should be 10x the parasitic capacitance on the common mode of the capacitor array in order to keep INL below ± 0.5 LSB. Much worse INL was found here, as discussed in Chapter 5. Therefore, a possible improvement to the algorithm would be to allow more capacitors to be connected for the MSBs than the LSBs. Since the MSB comparisons dominate the error, allowing (or requiring) more capacitors to be averaged together on the first few comparisons for each conversion could be a way to improve performance without significantly increasing the required number of capacitors or the time to compute the solution trees.

Another important way to improve the algorithm is to make it faster, thus allowing for high resolutions or other modifications that make it more complex to be possible. As discussed in Chapter 3, the algorithm that does not consider precharging/discharging capacitors in between comparisons starts to approach days of computation time around 13 bits on the hardware used. Allowing extra capacitors to

be charged or discharged between comparisons significantly increases the computation time (e.g. from 8 seconds to 80 hours for 10 bits). One direct way to improve the speed of the algorithm might be to find a way to parallelize some of its operations. Another way could be to rewrite some of the operations as functions and compile them as executables (".mex" files in Matlab), which are known to be much faster than interpreted Matlab code. Alternatively the entire algorithm could be re-written in a lower-level compiled language such as C. If the speed of the algorithm were less of an issue, it is possible that more than one capacitor could be allowed to charge or discharge between comparisons (thus reducing the number of capacitors required), or that allowing more capacitors to be connected at once during conversions to improve resistance to parasitics would be easier.

List of References

[1] J. W. Clark, M. R. Neuman, W. H. Olson, R. A. Peura, F. P. Primiano, M. P. Siedband, J. G. Webster, and L. A. Wheeler, *Medical instrumentation: Application and design*, vol. 5. John Wiley & Sons, Inc., 4 ed., feb 1997.

[2] L. Wong, S. Hossain, A. Ta, J. Edvinsson, D. Rivas, and H. Naas, "A very low-power CMOS mixed-signal IC for implantable pacemaker applications," *IEEE Journal of Solid-State Circuits*, vol. 39, pp. 2446–2456, dec 2004.

[3] F. Chen, A. P. Chandrakasan, and V. Stojanovic, "A low-power area-efficient switching scheme for charge-sharing DACs in SAR ADCs," in *IEEE Custom Integrated Circuits Conference 2010*, pp. 1–4, IEEE, sep 2010.

[4] R. J. Baker, *CMOS: Circuit Design, Layout, and Simulation.* Hoboken, NJ, USA: John Wiley & Sons, Inc., 3 ed., aug 2010.

[5] J. McCreary and P. Gray, "All-MOS charge redistribution analog-to-digital conversion techniques. I," *IEEE Journal of Solid-State Circuits*, vol. 10, pp. 371–379, dec 1975.

[6] Z. Zhu and Y. Liang, "A 0.6-V 38-nW 9.4-ENOB 20-kS/s SAR ADC in 0.18-μm CMOS for Medical Implant Devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 9, pp. 2167–2176, 2015.

[7] B. S. Rikan, S. Y. Kim, and K. Y. Lee, "11 -bit 1.8uW 40KS/s segmented SAR ADC for sensor applications," in *ISOCC 2016 - International SoC Design Conference: Smart SoC for Intelligent Things*, pp. 55–56, IEEE, oct 2016.

[8] D. Zhang, A. Bhide, and A. Alvandpour, "A 53-nW 9.1-ENOB 1-kS/s SAR ADC in 0.13-μm CMOS for medical implant devices," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 7, pp. 1585–1593, 2012.

[9] M. Liu, A. van Roermund, and P. Harpe, "A 7.1fJ/conv.-step 88dB-SFDR 12b SAR ADC with energy-efficient swap-to-reset," in *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, no. 2, pp. 409–412, IEEE, sep 2016.

[10] D. Lanfranconi, *Reduced Capacitor Eight Bit SAR Analog to Digital Converter.* Master's, Carleton University, 2014.

[11] S. J. Patel and R. A. Thakker, "Parasitic Aware Automatic Analog CMOS Circuit Design Environment Using ABC Algorithm," *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, pp. 445–446, 2018.

[12] C. N. J. Liu, Y. L. Chen, T. Y. Liu, and T. C. Chen, "Reliability-aware design automation flow for analog circuits," *ISOCC 2015 - International SoC Design Conference: SoC for Internet of Everything (IoE)*, pp. 1–2, 2016.

[13] I. Canturk and N. Kahraman, "Comparative analog circuit design automation based on multi-objective evolutionary algorithms: An application on CMOS opamp," in *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*, pp. 1–4, IEEE, jul 2015.

[14] N. N. Eleyan, K. Lin, M. Kamal, B. Mohammad, and P. Bassett, "Semi-custom design flow: Leveraging Place and route tools in Custom Circuit design," in *2009 IEEE International Conference on IC Design and Technology*, pp. 143–147, IEEE, may 2009.

[15] D. Sayed and M. Dessouky, "Automatic generation of common-centroid capacitor arrays with arbitrary capacitor ratio," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pp. 576–580, IEEE Comput. Soc, 2002.

[16] B. Razavi, "A Tale of Two ADCs," *IEEE Solid-State Circuits Magazine*, pp. 38–46, 2015.

[17] P. Harpe, A. Baschirotto, and K. A. A. Makinwa, eds., *High-Performance AD and DA Converters, IC Design in Scaled Technologies, and Time-Domain Signal Processing*. Springer, 2014.

[18] R. Stephens, *Essential Algorithms: A Practical Approach to Computer Algorithms*. Indianapolis, IN: John Wiley & Sons, 2013.

[19] T. Kugelstadt, "The Operation of the SAR–ADC Based on Charge Redistribution," *Texas Instruments Analog Applications Journal*, pp. 10-12, Feb. 2000.

[20] M. Shim, S. Jeong, P. D. Myers, S. Bang, J. Shen, C. Kim, D. Sylvester, D. Blaauw, and W. Jung, "Edge-Pursuit Comparator: An Energy-Scalable Oscillator Collapse-Based Comparator With Application in a 74.1 dB SNDR and 20 kS/s 15 b SAR ADC," *IEEE Journal of Solid-State Circuits*, vol. 52, pp. 1077–1090, apr 2017.

[21] J. Simon, "VChooseKR", version 1.0.0.0, 2010, https://www.mathworks.com/matlabcentral/fileexchange/26277-vchoosekr, Date Accessed: 2017-01-14.

[22] B. Murmann, "Digitally assisted data converter design," *European Solid-State Circuits Conference*, pp. 24–31, 2013.

[23] A. Hastings, *The Art of Analog Layout*. New Jersey: Pearson Prentice Hall, 2 ed., 2006.

[24] F. Balasa and K. Lampaert, "Symmetry within the sequence-pair representation in the context of placement for analog design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 721–731, jul 2000.

[25] B. Razavi, "The Bootstrapped Switch [A Circuit for All Seasons]," *IEEE Solid-State Circuits Magazine*, vol. 7, no. 3, pp. 12–15, 2015.

[26] J. Lu and J. Holleman, "A low-power high-precision comparator with time-domain bulk-tuned offset cancellation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 5, pp. 1158–1167, 2013.

[27] W. Kester, ed., *The Data Conversion Handbook*. New York: Elsevier Ltd, 2005.

[28] "IEEE Standard for Terminology and Test Methods for Analog-to-Digital Converters," in IEEE Std 1241-2010 (Revision of IEEE Std 1241-2000) , vol., no., pp.1-139, 14 Jan. 2011.

[29] A. Schaefer, "The Effective Number of Bits (ENOB) of my R&S Digital Oscilloscope," tech. rep., Rohde & Scharz, 2011.

[30] M. P.-H. Lin, V. W.-H. Hsiao, C.-Y. Lin, and N.-C. Chen, "Parasitic-Aware Common-Centroid Binary-Weighted Capacitor Layout Generation Integrating Placement, Routing, and Unit Capacitor Sizing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, pp. 1274–1286, aug 2017.

[31] K.-H. Ho, H.-C. Ou, Y.-W. Chang, and H.-F. Tsao, "Coupling-Aware Length-Ratio-Matching Routing for Capacitor Arrays in Analog Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, pp. 161–172, feb 2015.

[32] B. E. Jonsson, "A survey of A/D-Converter performance evolution," in *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pp. 766–769, IEEE, dec 2010.

[33] C. Fisher, R. Blankenship, J. Jensen, T. Rossman, and K. Svilich, "Optimization of standard cell libraries for low power, high speed, or minimal area designs," in *Proceedings of Custom Integrated Circuits Conference*, pp. 493–496, IEEE, 1996.

[34] Binghong Guan and C. Sechen, "Large standard cell libraries and their impact on layout area and circuit performance," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pp. 378–383, IEEE Comput. Soc. Press, 1996.

Appendix A

Backtracking Function Code

The following is the Matlab implementation of the backtracking function discussed in Section 3.1. The function itself is called `fillCapacitorDecisionTree`, and is called by a script which sets up the variables to be passed into the function, partially using the `buildConnectArray` helper function, also discussed in Section 3.1.

```
%%
% Main script to be run, which calls the buildConnectArray helper
% function and the fillDecisionTree backtracking function.
% The setup shown here initializes the number of bits, max caps per
% connection, and max total caps to 10, 6, and 25, respectively,
% as this is what the final design used. However, to arrive at these
% numbers the script could be modified to loop through them to find
% the minimum number of capacitors the function will find for a
% given resolution for example.

%%
% Number of bits we're trying to achieve
max_num_bits = 10;
N = max_num_bits;

% Max number of capacitors to be connected at any given step (M)
max_caps_per_con = 6;
M1 = max_caps_per_con;

% Total number of capacitors available
max_total_caps = 25;
M2 = max_total_caps;

results_array = [];
connect_array = [];
```

```
% Call the function to build the connect array for each number of caps we
% might connect at once
for M1 = 2:2:max_caps_per_con
    connect_array = [connect_array, nan(length(connect_array), 2); ...
        buildConnectArray(M1, N)];
end

connect_array = flip(connect_array);
connect_labels = connect_array(:,1);
[~, w] = size(connect_array);
connect_values = connect_array(:,2:w);

% The tree_builder function simply preallocated empty arrays for the
% vol_tree, con_tree, and oz_tree, and fills in the val_tree with all
% the voltages we will need to generate in the binary search
[val_tree, vol_tree, con_tree, oz_tree] = tree_builder(N, M2);
connections = cell(2^N-1,1);

for node = 1:2^N-1
    value_index = find(connect_labels == val_tree(node))';
    connections{node} = connect_values(value_index, :);
end

% initial_caps is the initial matrix of available capacitors
% Sets more 0s than 1s when M is odd
initial_caps = [zeros(1, ceil(M2/2)) ones(1, floor(M2/2))];

% Call the backtracking function:
[works, vol_tree, con_tree, oz_tree] = fillCapacitorDecisionTree(1, N, M2, ...
    val_tree, vol_tree, con_tree, oz_tree, connections, initial_caps);

if works
    fprintf('Works!\n');
else
    fprintf('Failed\n');
end


% buildConnectArray - Helper function for the backtracking algorithm.
% Builds the array of voltages that can be averaged together to generate
% each voltage required in the ADC's binary search. The resulting array
% does not contain any repeated sets.

function output = buildConnectArray(num_caps, num_bits)

   % num_caps: Max number of capacitors that can be connected at
   % once (or that we'll allow to be connected at once) (M)

   % num_bits: Total number of bits (N)

   % Arbitrarily pre-allocate array of size 5e6 * num_caps + 1
```

```
% (since we do not know the exact size in advance)
output = zeros(5e6,num_caps+1,'double');
output_index = 1;

for k = 0:num_bits

    %%
    % This first loop generates the array of values (X) between 0 and 1
    % in multiples of 1/(2^(k-1)). So for k = 3 it generates
    % [0 0.25 0.5 0.75 1]

    % Xtotal is all the possible values that could have been generated
    % at the previous level (i.e. the values we will be averaging to
    % generate the values at the current level

    Xlast = zeros(1,0);
    if k > 0
        Xlast = X;
    end

    X = zeros(1, 2^k);
    for n1 = 1:length(X)
        X(n1) = n1/(2^k);
    end
    X = [0 X];

    if k > 0
        % Xdiff is all the values we want to try to generate at the
        % current level using any value we might have generated in any
        % previous level.
        Xdiff = setdiff(X, Xlast);

        %%
        % This set of loops will go through each value in Xdiff and find
        % all combinations of 2 to M values from Xlast (including repeats)
        % that average to that value

        % Doing this part first means running catpad and nchoosek many
        % more times, but with much smaller arrays each time
        for i = 1:length(Xdiff)

            % The current value we're trying to average to
            current_value = Xdiff(i);

            % Get the path of previous voltages to the current value
            % i.e. the values that are actually available to average
            % together to produce the current value. Doing this means we
            % take out a lot of redundancy:

            % All the previous values the DAC will have had to produce
            % to get to this one
```

```
            path = zeros(1, k + 1);

            % The last element is 1, the second last is 0
            path(length(path)) = 1;

            if k > 1

                % The first element for k = 2+ is always 0.5
                path(1) = 0.5;

                for j = 2:k-1

                    if current_value > path(j-1)
                        path(j) = path(j-1) + 1/(2^j);
                    else
                        path(j) = path(j-1) - 1/(2^j);
                    end
                end
            end

            % Info and License for the VChooseKR function used below:

% VChooseKR - Combinations of K elements (repetitions) [MEX]
% VChooseKR(V, K) creates a matrix, which rows are all combinations of
% choosing K elements of the vector V without order and with repetitions.
% https://www.mathworks.com/matlabcentral/fileexchange/26277-vchoosekr

% Copyright (c) 2010, Jan Simon
% All rights reserved.
%
% Redistribution and use in source and binary forms, with or without
% modification, are permitted provided that the following conditions are
% met:
%
%     * Redistributions of source code must retain the above copyright
%       notice, this list of conditions and the following disclaimer.
%     * Redistributions in binary form must reproduce the above copyright
%       notice, this list of conditions and the following disclaimer in
%       the documentation and/or other materials provided with the distribution
%
% THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
% AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
% IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
% ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
% LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
% CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
% SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
% INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
% CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
% ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
% POSSIBILITY OF SUCH DAMAGE.
```

```
                Y = VChooseKR(path, num_caps);

                for n2 = 1:length(Y)
                    x = Y(n2,:);
                    average = sum(x)/length(x);
                    if average == current_value
                        add_row = [current_value, Y(n2,:)];
                        if output_index <= length(output)
                            output(output_index, :) = add_row;
                        else
                            output = [output; add_row];
                        end
                        output_index = output_index + 1;
                    end
                end
            end
        end
    end

    % Truncate the array if the original pre-allocated size was not surpassed
    if length(output) > output_index
        output = output(1:output_index-1,:);
    end

end


% Depth first recursive search to fill in the Trees
function [... RETURN VALUES:
        works,      ... False when the current state of the capacitor
                    ...  array is found to lead to an invalid solution,
                    ...  otherwise, True.
        vol_tree,   ... Array containing the voltage on each capacitor
                    ...  at each node.
        con_tree,   ... Array containing which capacitors' switches to
                    ...  close at each node.
        oz_tree     ... Array containing a 1 or 0 at the same index of
                    ...  the capacitor to be charged or discharged in
                    ...  con_tree.
          ]=        ...
    fillCapacitorDecisionTree(... FUNCTION PARAMETERS:
        node,       ... The current node in the tree.
        N,          ... The number of bits of the desired ADC.
        M2,         ... The total number of capacitors in the array.
        val_tree,   ... Array containing the voltage to be generated at
                    ...  each node.
        vol_tree,   ... Array containing the voltage on each capacitor
                    ...  at each node.
        con_tree,   ... Array containing which capacitors' switches to
                    ...  close at each node.
```

```
    oz_tree,    ... Array containing a 1 or 0 at the same index of
                ...  the capacitor to be charged or discharged in
                ...  con_tree. (Not used without pre-charge/discharge)
    connections,... A cell array containing the rows of the array
                ...  generated by buildConnectArray that average to
                ...  make the desired voltage at each node.
    initial_caps... The voltages present on the capacitors when the
                ...  function is called.
    )

%% SETUP
first_leaf = (2^N)/2;
% The value of the node of the tree (eg. the first one is 0.5)
value = val_tree(node);
connect_values = connections{node};
[W, ~] = size(connect_values);

%% PREALLOCATE ARRAYS
Z = zeros(1, M2);
Na = nan(1, M2);

%%
% Iterate through each possible connection decision (if necessary)
for k1 = 1:W

    % Get the appropriate row from the connection array
    conn = connect_values(k1,:);
    % Remove the NaNs
    conn = conn(~isnan(conn));

    % Flips because connecting more at once is probably
    % better for the circuit (less error-prone)
    k2_flip = flip((M2 - length(conn)):M2);
    for k2 = k2_flip % k2 is where to put a 1 or 0

        % Passed in. The current values on the cap array
        c1 = initial_caps;
        % Goes to connection tree
        % (the indices of the caps that are connected together)
        c2 = Z;
        % Goes to one-zero tree (the indices in the connection
        % tree of the caps that are connected together)
        c3 = Na; % Comment out without pre-charge/discharge

        % Set to true unless proven otherwise
        found = true;

        % This loop checks if each cap voltage
        % required is available in the array
        for k3 = 1:length(conn)
            index = find(conn(k3)==c1, 1);
```

```
            available = ~isempty(index);
            if available
                % if it is replace it with the current value.
                c1(index) = value;
                c2(k3) = index;
            % Comment out both elseif's without pre-charge/discharge
            elseif conn(k3) == 0 && c1(k2) ~= value && c1(k2) ~= 0
                c1(k2) = value;
                c2(k3) = k2;
                c3(k3) = 0;
            elseif conn(k3) == 1 && c1(k2) ~= value && c1(k2) ~= 1
                c1(k2) = value;
                c2(k3) = k2;
                c3(k3) = 1;
            else
                found = false; % If it's not, set found to false
                break; % ---------------------------------->>>
                % Causes the for k2 loop to go on to the next value
            end
        end

        if (found)

            if node >= first_leaf % it was found and we're at a leaf
                vol_tree(node, :) = c1;
                con_tree(node, :) = c2;
                oz_tree(node, :) = c3; % Comment out without pre-charge/discharge
                works = true; % So this node was successful
                return; % ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^/\/\
            else
                % Remove oz_tree from below without pre-charge/discharge
                [works, vol_tree, con_tree, oz_tree] = ...
                    fillCapacitorDecisionTree(2*node, N, M2, val_tree, ...
                    vol_tree, con_tree, oz_tree, connections, c1);
                if works
                    % Call the function on the next node down on the
                    % right
                    [works, vol_tree, con_tree, oz_tree] = ...
                        fillCapacitorDecisionTree(2*node + 1, N, M2, ...
                        val_tree, vol_tree, con_tree, oz_tree, ...
                        connections, c1);
                    if works
                        vol_tree(node, :) = c1;
                        con_tree(node, :) = c2;
                        oz_tree(node, :) = c3;
                        return; % ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^/\/\
                    end
                end
            end
        end
    end
```

```
    end

    % If we get here, then the capacitors passed in won't lead to a valid
    % solution, so return works = false.
    works = false;
    return;
end
```

Appendix B

Verilog Generation Code

B.1 Automated Verilog Generation

The following Matlab script takes the output of the backtracking algorithm discussed in Section 3.1 and provided in Appendix A, and generates most of the Verilog code that implements the state machine discussed in Section 3.2. Note that while most of the Verilog code was generated automatically by this script, some was added manually. These additions are shown in Section B.2.

```
% This script generates much of the SAR state machine Verilog code
% based on the trees generated by the backtracking function. It assumes
% these arrays are already present in the Matlab workspace:
%
% val_tree,  % Array containing the voltage to be generated at
%            % each node.
% con_tree,  % Array containing which capacitors' switches to
%            % close at each node.
% oz_tree,   % Array containing a 1 or 0 at the same index of
%            % the capacitor to be charged or discharged in
%            % con_tree.

fileID = fopen('SAR.v', 'w');

M = 25; % Number of capacitors in the array
cases = string.empty(N,0);
states = [];
con_tree_counter = 1;
oz_tree_counter = 1;
```

```
%% Filler for module and I/O declarations
fprintf(fileID, '\r\n');
fprintf(fileID, ['module SAR(cap_switches, array_gnd, array_vdd, '...
                 'data_out, data_clk_out, comp, compP, clk, rst_n,'...
                 'sh_clk, CAL_ext, CAL_out, CAL_rst_caps);\r\n\r\n']);
fprintf(fileID, '\r\n');
fprintf(fileID, 'output [%i:0] cap_switches;\r\n', M-1);
fprintf(fileID, ['output array_gnd, array_vdd, data_out,'...
                 'data_clk_out, CAL_out, CAL_rst_caps;\r\n']);
fprintf(fileID, 'input  comp, compP, clk, rst_n, sh_clk, CAL_ext;\r\n\r\n');

%% Generate 'case' and 'state' names (one for each bit)
% The 'cases' are technically the states of the state machine,
% but they go in the case statement so we will call them 'cases'
% to distinguish them from the logic conditions in each state
% which go in the 'prev_state' variable and determine the output
for n1 = fliplr(1:2*N)
    cases(n1) = strcat('B', string(floor((n1-1)/2)));
    if mod(n1,2) ~= 0
        suffix = 'b';
    else
        for n2 = 1:(2^(N-n1/2))
            states = [states; strcat(cases(n1),'_',string(2*n2-1))];
        end
        suffix = 'a';
    end
    cases(n1) = strcat(cases(n1), suffix);
end

%% Print the 'case' name parameter declarations
case_length = ceil(log2(2*N+1));
fprintf(fileID, '// \''Cases\''\r\n');
fprintf(fileID, 'parameter [%i:0] INIT   = %i\''b%s,\r\n', ...
    case_length - 1, case_length, (dec2bin(0,case_length)));
for n1 = fliplr(1:length(cases))
    n2 = length(cases) - n1 + 1;
    fprintf(fileID, '%s%s     = %i\''b%s', repmat(' ',1,16), ...
        cases(n1), case_length, dec2bin(n2,case_length));
    fprintf(fileID, ',\r\n');
end

% Calibration state parameters ('cases'):
num_cal_cases = 5;
for n1 = 1:num_cal_cases
    n2 = length(cases) + n1;
    fprintf(fileID, '%s%s%s    = %i\''b%s', repmat(' ',1,16), ...
        'CAL', num2str(n1), case_length, dec2bin(n2,case_length));
    fprintf(fileID, ',\r\n');
end
fprintf(fileID, '%s%s   = %i\''b%s;', repmat(' ',1,16), ...
        'RESET', case_length, ...
```

```
        dec2bin(length(cases) + num_cal_cases + 1, case_length));
fprintf(fileID, '\r\n');

%% Print the 'state' name parameter declarations
fprintf(fileID, '\r\n');
fprintf(fileID, '// \''States\''\r\n');
fprintf(fileID, 'parameter [%i:0] NONE     = %i\''b%s,\r\n', ...
    N-1, N, dec2bin(0,N));
for n1 = 1:length(states)
    fprintf(fileID, '%s%s%s= %i\''b%s', repmat(' ',1,16), states(n1),...
        repmat(' ',1,9-strlength(states(n1))), N, dec2bin(n1,N));
    if n1 == length(states)
        fprintf(fileID, ';\r\n');
    else
        fprintf(fileID, ',\r\n');
    end
end

%% Generate register declarations
fprintf(fileID,'\r\n');
fprintf(fileID,'reg [%i:0]   state;\r\n', case_length - 1);
fprintf(fileID,'reg [%i:0]   next_state;\r\n', case_length - 1);
fprintf(fileID,'reg [%i:0]   prev_state_reg;\r\n', N-1);
fprintf(fileID,'reg [%i:0]   prev_state;\r\n', N-1);
fprintf(fileID,'reg [%i:0]  cap_switches;\r\n', M-1);
fprintf(fileID,'reg [%i:0]  cap_switches_reg;\r\n', M-1);
fprintf(fileID,'reg         array_vdd;\r\n');
fprintf(fileID,'reg         array_gnd;\r\n');
fprintf(fileID,'reg         data_clk_out;\r\n');
fprintf(fileID,'reg         data_clk;\r\n');
fprintf(fileID,'reg         data_out;\r\n');
fprintf(fileID,'reg         get_comp;\r\n');
fprintf(fileID,'reg         CAL_out;\r\n');
fprintf(fileID,'reg         CAL_rst_caps;\r\n');
fprintf(fileID,'\r\n\r\n');

%% Generate the first always block
fprintf(fileID, 'always @ (negedge clk or negedge rst_n) begin\r\n');
fprintf(fileID, '  if (!rst_n)\r\n    begin\r\n');
fprintf(fileID, '      state           <= RESET;\r\n');
fprintf(fileID, '      prev_state_reg  <= NONE;\r\n');
fprintf(fileID, '      data_out        <= 1\''b0;\r\n');
fprintf(fileID, '      data_clk_out    <= 1\''b0;\r\n');
fprintf(fileID, '    end\r\n  else\r\n    begin\r\n');
fprintf(fileID, '      cap_switches_reg  <= cap_switches;\r\n');
fprintf(fileID, '      prev_state_reg    <= prev_state;\r\n');
fprintf(fileID, '      data_clk_out      <= data_clk;\r\n\r\n');
fprintf(fileID, '      if (sh_clk || CAL_ext)\r\n');
fprintf(fileID, '         state          <= next_state;\r\n');
fprintf(fileID, '      else\r\n');
fprintf(fileID, '         state          <= INIT;\r\n\r\n');
```

```
fprintf(fileID, '      if (get_comp)\r\n');
fprintf(fileID, '         data_out       <= comp;\r\n');
fprintf(fileID, '    end\r\nend\r\n\r\n');

%% Generate the beginning of the second always block
fprintf(fileID, 'always @ (state or comp or compP) begin\r\n\r\n');

fprintf(fileID,['  if (clk && (!comp || !compP) && sh_clk) begin\r\n',...
                '    cap_switches = 25\''b0000000000000000000000000;\r\n',...
                '    array_vdd    = 1\''b1;\r\n',...
                '    array_gnd    = 1\''b0;\r\n',...
                '  end else begin\r\n\r\n']);

fprintf(fileID, '  // DEFAULT VALUES\r\n');
fprintf(fileID, ['  cap_switches  = cap_switches_reg;\r\n',...
                 '  array_gnd     = 1\''b0;\r\n',...
                 '  array_vdd     = 1\''b1;\r\n',...
                 '  next_state    = INIT;\r\n',...
                 '  prev_state    = prev_state_reg;\r\n',...
                 '  data_clk      = 1\''b0;\r\n',...
                 '  get_comp      = 1\''b0;\r\n'...
                 '  CAL_out       = 1\''b1;\r\n'...
                 '  CAL_rst_caps  = 1\''b0;\r\n\r\n']);

fprintf(fileID, '  case (state)\r\n');

%% Generate INIT State
fprintf(fileID, '  INIT:\r\n');
fprintf(fileID, '    begin\r\n');
fprintf(fileID, '      next_state = %s;\r\n', cases(end));
fprintf(fileID, '      // When we initialize, first discharge all caps\r\n');
fprintf(fileID, '      cap_switches = %i\''b%s;\r\n', M, dec2bin(2^M-1,M));
fprintf(fileID, '      array_gnd = 1\''b1;\r\n');
fprintf(fileID, '      array_vdd = 1\''b1;\r\n');
fprintf(fileID, '    end\r\n');

%% Generate the case logic - TWO blocks per bit
% with if blocks corresponding to the number of possible
% nodes at that level of the tree
for n1 = fliplr(1:2*N)
    fprintf(fileID, '  %s:\r\n', cases(n1));
    fprintf(fileID, '    begin\r\n');
    if n1 == 1
        next = 'INIT';
    else
        next = cases(n1-1);
    end
    fprintf(fileID, '        next_state = %s;\r\n', next);

    Q = zeros(1,M);
```

```
%% EVEN = a
    if mod(n1,2) == 0

        % interclock stuff

        % n1 = 2N is the first state after INIT (i.e. the charge state) -
        % charge half the caps to vdd (use the initial_caps array)
        if n1 == 2*N
            Q = '';
            for n2 = 1:M
                Q = strcat(Q, int2str(initial_caps(n2)));
            end
            fprintf(fileID, '        // Charge half the caps to 1 \r\n');
            fprintf(fileID, '        cap_switches = %i\''b%s;\r\n', M, Q);
            fprintf(fileID, '        array_gnd = 1\''b0;\r\n');
            fprintf(fileID, '        array_vdd = 1\''b0;\r\n');
            oz_tree_counter = oz_tree_counter + 1;

        else

            OZ_array = zeros(0,1);
            tree_index = zeros(0,1);
            for n2 = oz_tree_counter:(2*oz_tree_counter - 1)
                C0 = oz_tree(n2,:);

                if any(~isnan(C0))
                    OZ_array = [OZ_array; C0];
                    tree_index = [tree_index; n2];
                end
            end

            for n2 = 1:length(OZ_array(:,1))

                C0 = OZ_array(n2,:);
                Cconn = con_tree(tree_index(n2),:);

                    if n2 == 1
                        el = '';
                    else
                        el = 'else ';
                    end
                    prev_state_reg = states(floor(tree_index(n2)/2));

                    comp = mod(tree_index(n2),2);

                    if n1 == 2*(N-1)
                        fprintf(fileID, ['        ',...
                            'if (comp_reg == 1\''b%i)\r\n'], comp);
                    else
                        fprintf(fileID, ['        %sif (comp_reg == 1\''b%i'...
                            '&& prev_state_reg == %s)\r\n'], el, ...
```

```
                        comp, prev_state_reg);
                end

                fprintf(fileID,'            begin\r\n');

                [Num, Den] = rat(val_tree(tree_index(n2)));
                fprintf(fileID, ['                ',...
                    '// Must create %i/%i*Vdd\r\n'], Num, Den);

                index = Cconn(find(~isnan(C0)));
                if C0(~isnan(C0)) == 0
                    fprintf(fileID, ['                ',...
                        '// Discharging C%i\r\n'], index);
                    fprintf(fileID, ['                ',...
                        'array_gnd = 1\''b1;\r\n']);
                else
                    fprintf(fileID, ['                ',...
                        '// Charging C%i\r\n'], index);
                    fprintf(fileID,['                ',...
                        'array_vdd = 1\''b0;\r\n']);
                end

                fprintf(fileID,'                cap_switches = %i\''b', M);
                Q = zeros(1,M);
                Q(index) = 1;
                for n3 = 1:M
                    fprintf(fileID, '%i', Q(n3));
                end
                fprintf(fileID, ';\r\n            end\r\n');
            end
            oz_tree_counter = 2*oz_tree_counter;
        end


%% ODD = b
    else
            fprintf(fileID, '        data_clk = 1\''b1;\r\n');
            fprintf(fileID, '        get_comp = 1\''b1;\r\n');
        if n1 == 2*N - 1
            [Num, Den] = rat(val_tree(con_tree_counter));
            fprintf(fileID, '        // Must create %i/%i*Vdd\r\n', Num, Den);
            caps = con_tree(con_tree_counter,:);
            caps = caps(caps ~= 0);
            con_tree_counter = con_tree_counter + 1;
            Q(caps) = 1;
            fprintf(fileID, '        cap_switches = %i\''b', M);
            for n2 = 1:M
                fprintf(fileID, '%i', Q(n2));
            end
            fprintf(fileID, ';\r\n');
        else
```

```
            for n2 = 1:(2^(N-((n1+1)/2)))
                if n2 == 1
                    el = '';
                else
                    el = 'else ';
                end
                prev_state_reg = strcat('B',string((n1+1)/2),...
                    '_',string(n2-1 + mod(n2,2)));
                prev_state = states(con_tree_counter);

                comp = mod(n2+1,2);

                if n2 == (2^(N-((n1+1)/2)))
                    fprintf(fileID, '        else\r\n');
                elseif n1 == 2*(N-1)-1
                    fprintf(fileID,...
                        '        if (comp_reg == 1\''b%i)\r\n', comp);
                else
                    fprintf(fileID, ['        %sif (comp_reg == 1\''b%i',...
                        ' && prev_state_reg == %s)\r\n'],...
                        el, comp, prev_state_reg);
                end

                fprintf(fileID,'            begin\r\n');
                [Num, Den] = rat(val_tree(con_tree_counter));
                fprintf(fileID, ['                ',...
                    '// Must create %i/%i*Vdd\r\n'], Num, Den);
                fprintf(fileID,'                cap_switches = %i\''b', M);
                Q = zeros(1,M);
                caps = con_tree(con_tree_counter,:);
                caps = caps(caps ~= 0);
                con_tree_counter = con_tree_counter + 1;
                Q(caps) = 1;
                for n3 = 1:M
                    fprintf(fileID, '%i', Q(n3));
                end
                fprintf(fileID,[';\r\n                prev_state = %s;\r\n',...
                    '            end\r\n'], prev_state);
            end
        end

    end

    fprintf(fileID, '    end\r\n');
end

fprintf(fileID, '  endcase\r\n');
fprintf(fileID, 'end\r\n');
fprintf(fileID, 'endmodule');
```

```
fclose(fileID);
```

B.2 Manually Added Verilog Code

The following blocks of code are manually modified or added to the file generated by the script in Section B.1. The `RESET` and `B9a` states are changed as follows:

```
RESET:
begin
  next_state    = INIT;
  cap_switches  = 25'b1111111111111111111111111;
  array_gnd     = 1'b1;
  // array_vdd is off when high
  array_vdd     = 1'b1;
  CAL_out       = 1'b0;
  CAL_rst_caps  = 1'b1;
end

B9a:
begin
  // CAL_ext is set externally and controls when calibration happens
  if (CAL_ext) begin
    next_state    = CAL1;
    CAL_rst_caps  = 1'b0; // Set to 1 by default otherwise
  end
  else begin
    next_state    = B9b;
  end
  // Charge half the caps to 1
  cap_switches  = 25'b0000000000000011111111111;
  array_gnd     = 1'b0;
  array_vdd     = 1'b0;
end
```

Then the the five `CAL` states are added (which control the switches for the comparator's calibration), as follows:

```
// 5 STEP CALIBRATION
CAL1:
begin
  next_state    = CAL2;
  cap_switches  = 25'b0111111111111111111111111;
  CAL_out       = 1'b1;
end
```

```
CAL2:
begin
  next_state    = CAL3;
  cap_switches  = 25'b0111111111111111111111111;
  CAL_out       = 1'b1;
end

CAL3:
begin
  next_state    = CAL4;
  cap_switches  = 25'b0111111111111111111111111;
  CAL_out       = 1'b1;
end

CAL4:
begin
  next_state    = CAL5;
  cap_switches  = 25'b0111111111111111111111111;
  CAL_out       = 1'b1;
end

CAL5:
begin
  next_state    = INIT;
  cap_switches  = 25'b0111111111111111111111111;
  CAL_out       = 1'b1;
end
```

Appendix C

Test PCB Schematic & Layout

The PCB used to test the ADC was designed in Altium Designer. The schematic of the PCB is included in Figure C.1. The layout is included in Figure C.2.

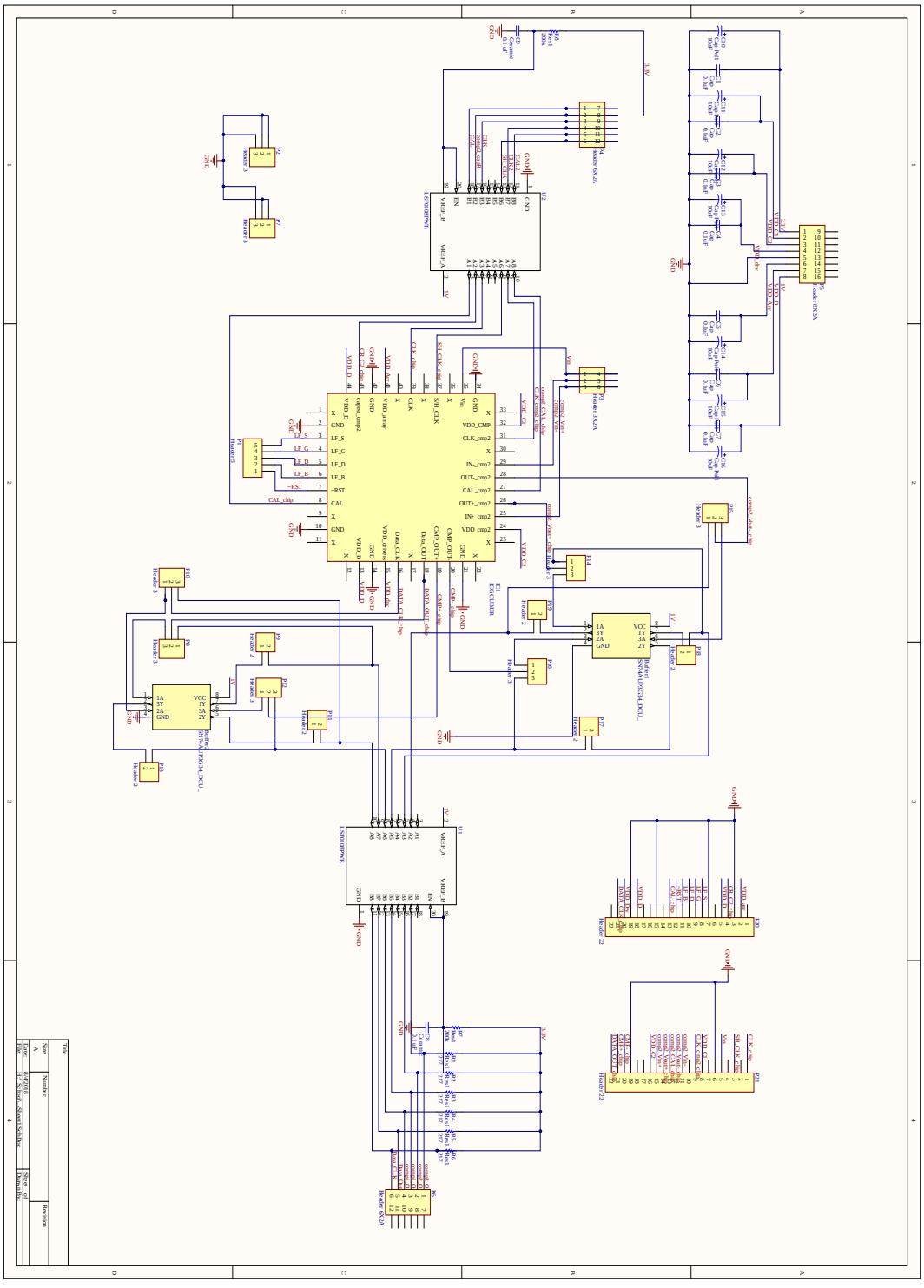


Figure C.1: Schematic of the PCB used to test the ADC, generated in Altium Designer.

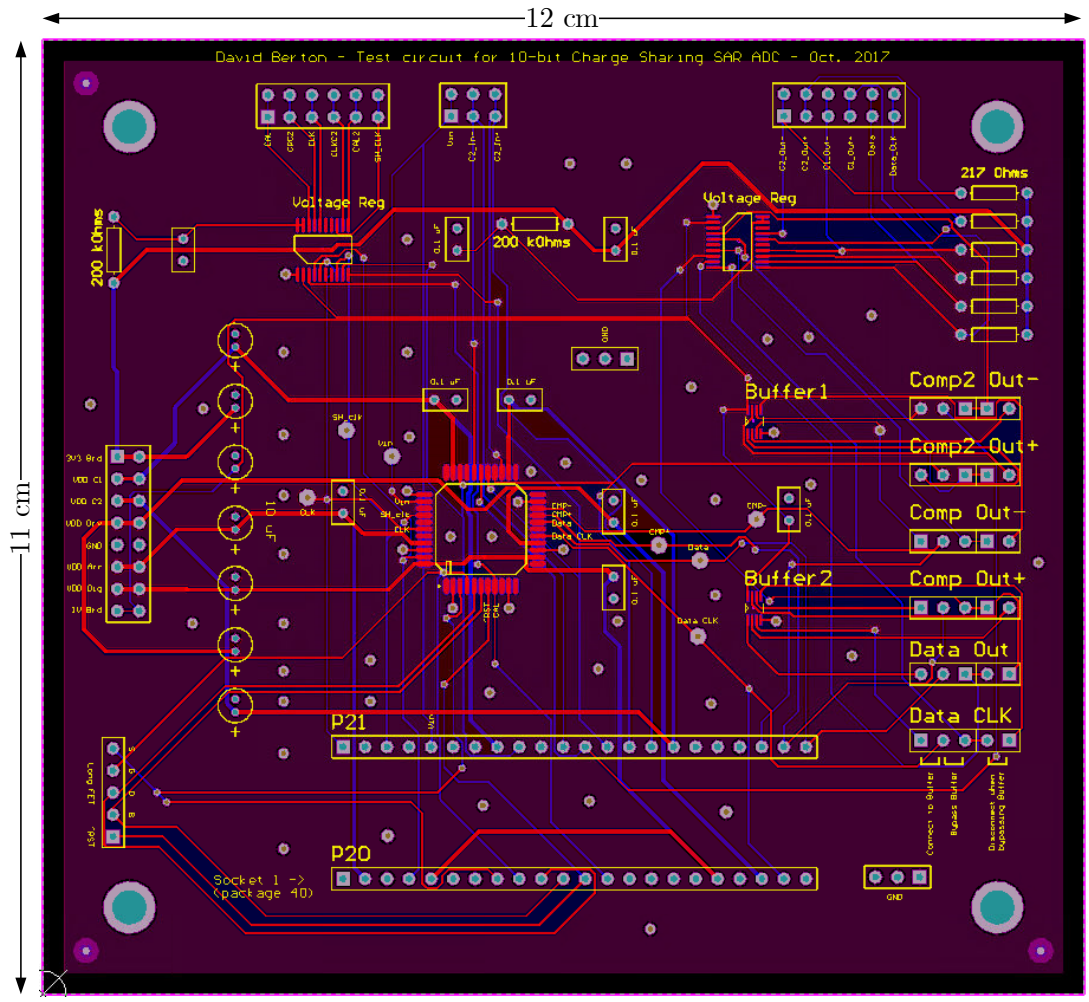


Figure C.2: Layout of the PCB used to test the ADC, generated in Altium Designer. Traces in red are the top layer metal, and traces in blue are the bottom layer metal.

Appendix D

ADC Performance Prediction Script

The following are the scripts discussed in Section 7.2 to help predict performance improvements in the ADC. The function `simulate_adc` is called by the two test bench scripts, which calculate the INL & DNL and ENOB, respectively.

```
%% Script for Determining INL & DNL
clear;
load('backup_for_results_N10_M1_6_M2_25_interclock3_with_bug_flipped_backwards.mat', ...
    'con_tree', 'oz_tree', 'N', 'initial_caps');

N = 10;
Nrecord = 20481;

Fs = 10e3;
fin = Fs*255/4096;
vin = linspace(0, 1, Nrecord);
M = length(initial_caps);

Tdroop = 1/410e3;
Cp_array = 30e-15;
sigma_percent = 0;
distribution = (randn(1,M)*sigma_percent/100) + 1;
Rswitch_off = 36e9;
Cu = 4e-12;

LSB = 1/(2^10);
edges = linspace(0,1 - LSB,1024);

for n1 = 1:length(Cu)
    for n2 = 1:length(Rswitch_off)
        Caps = Cu(n1) * distribution;
```

```
        vout = simulate_adc(vin, initial_caps, N, con_tree, oz_tree, Caps, ...
                Cp_array, Tdroop, Rswitch_off(n2));

        z = zeros(1, length(edges));
        for n = 1:length(edges)
            z(n) = nnz(vout == edges(n));
        end
        dnl = (z/20) - 1;
        inl = cumsum(dnl);

        dnl_max(n1, n2) = max(abs(dnl));
        inl_max(n1, n2) = max(abs(inl));
        dnl_mean_deviation(n1, n2) = mean(abs(dnl));
        inl_mean_deviation(n1, n2) = mean(abs(inl));
    end
end

%% 3D Plots
% [C, R] = meshgrid(Cu*1e12, Rswitch_off/1e9);
% figure();
% surf(C, R, dnl_max);
% xlabel('Unit Capacitance (pF)')
% ylabel('Switch Off Resistance (G\Omega)');
% zlabel('dnl_max');
% figure();
% surf(C, R, inl_max);
% xlabel('Unit Capacitance (pF)')
% ylabel('Switch Off Resistance (G\Omega)');
% zlabel('inl_max');
% figure();
% surf(C, R, dnl_mean_deviation);
% xlabel('Unit Capacitance (pF)')
% ylabel('Switch Off Resistance (G\Omega)');
% zlabel('dnl_mean_deviation');
% figure();
% surf(C, R, inl_mean_deviation);
% xlabel('Unit Capacitance (pF)')
% ylabel('Switch Off Resistance (G\Omega)');
% zlabel('inl_mean_deviation');

%% Calculate/plot DNL & INL
figure()
s(1) = subplot(1,2,1);
plot(dnl(1:end-1), 'Color', 'k');
ylim([-1.25 1.25]);
xlim([1 1023]);
box off
set(gca, 'FontName', 'CMU Serif', ...
    'FontSize', 12, 'xtick', [200 400 600 800 1000])
xlabel('Output Code')
ylabel('DNL (LSB)')
```

```
s(2) = subplot(1,2,2);
plot(inl(1:end-1), 'Color', 'k');
ylim([-1.25 1.25]);
xlim([1 1023]);
box off
xlabel('Output Code')
ylabel('INL (LSB)')



%% Script for Determining ENOB
clear;
load('backup_for_results_N10_M1_6_M2_25_interclock3_with_bug_flipped_backwards.mat', ...
    'con_tree', 'oz_tree', 'N', 'initial_caps');

N = 10;
Nrecord = 4096;

Fs = 10e3;
fin = Fs*255/4096;
t = linspace(0, (Nrecord - 1)/Fs, Nrecord);
vin = 0.5 + 0.5*sin(2*pi*fin*t);
[~, M] = size(con_tree);

Tdroop = 1/410e3;
Cp_array = 30e-15;
sigma_percent = 0;
distribution = (randn(1,M)*sigma_percent/100) + 1;

Rswitch_off = linspace(9e9, 54e9, 6);
Cu = linspace(1e-12, 6e-12, 5);

%% Sweep
for n1 = 1:length(Cu)
    for n2 = 1:length(Rswitch_off)
        Caps = Cu(n1) * distribution;
        vout = simulate_adc(vin, initial_caps, N, con_tree, oz_tree, Caps, ...
                Cp_array, Tdroop, Rswitch_off(n2));
        SINAD = sinad(vout,Fs);
        ENOB(n1, n2) = (SINAD - 1.76)/6.02;
    end
end

%% Plot
[C, R] = meshgrid(Cu*1e12, Rswitch_off/1e9);

figure(1);
surf(C, R, ENOB', 'FaceAlpha', 0.5);
xlabel('Unit Capacitance (pF)')
ylabel('Switch Off Resistance (G\Omega)');
```

```
zlabel('ENOB (bits)');

figure(2);
plot(R, ENOB');

figure(3);
plot(C', ENOB);



%% Function to "Simulate" the ADC
function decimal_out = simulate_adc(vin, ...
                                initial_caps, ...
                                N, ...
                                con_tree, ...
                                oz_tree, ...
                                Caps, ...
                                Cp_array, ...
                                Tdroop, ...
                                Rswitch_off)

    decimal_out = zeros(1,length(vin));

    for k = 1:length(vin)
        Vpar = 1;
        digital_out = zeros(1,N);
        current_caps = initial_caps;
        node = 1;
        for n = 1:N

            con = con_tree(node, :);
            con = con(con ~= 0);
            oz = oz_tree(node, :);

            %% A States
            current_caps(con(oz == 0)) = 0;
            current_caps(con(oz == 1)) = 1;

            if any(oz == 0)
                Vpar = 0;
            elseif any(oz == 1)
                Vpar = 1;
            end

            % Calculate A states droop
            Idroop = (current_caps - Vpar)/Rswitch_off;

            Vdroop_Cu = (Idroop*Tdroop)./Caps;
            current_caps = current_caps - Vdroop_Cu;

            Vdroop_par = (sum(Idroop)*Tdroop)/Cp_array;
```

```
            Vpar = Vpar + Vdroop_par;

            %% B States

            % Calculate new voltages on caps incl. parasitics
            total_charge = (sum(current_caps(con).*Caps(con))) + Cp_array*Vpar;
            total_capacitance = sum(Caps(con)) + Cp_array;
            avg = total_charge/total_capacitance;
            current_caps(con) = avg;

            % Calculate B states droop
            Idroop = (current_caps - avg)/Rswitch_off;

            Vdroop_Cu = (Idroop*Tdroop)./Caps;
            current_caps = current_caps - Vdroop_Cu;

            Vdroop_par = (sum(Idroop)*Tdroop)/(total_capacitance);
            avg = avg + Vdroop_par;

            Vpar = avg;
            current_caps(con) = avg;

            % Comparison
            if vin(k) < avg
                node = 2*node;
                digital_out(n) = 0;
            else
                node = 2*node + 1;
                digital_out(n) = 1;
            end
        end

        decimal_out(k) = bin2dec(num2str(digital_out))/2^N;
    end
end
```