Design and Measurement of StrongARM Comparators

Nathan Robert Whitehead
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/8715

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Design and Measurement of StrongARM Comparators

Nathan Robert Whitehead

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Shiuh-hua Wood Chiang, Chair
Stephen Schultz
Michael Rice

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2019 Nathan Robert Whitehead
All Rights Reserved
ABSTRACT

Design and Measurement of StrongARM Comparators

Nathan Robert Whitehead
Department of Electrical and Computer Engineering, BYU
Master of Science

The StrongARM comparator is utilized in many analog-to-digital converters (ADCs) because of its high power efficiency and rail-to-rail outputs. The performance of the comparator directly affects the speed, power, and accuracy of an ADC. However, the StrongARM comparator performance parameters such as delay, noise, and offset measured directly from silicon prototypes are rare in literature and often consist of small sample sets. In addition, existing techniques to measure the comparator require large chip areas, making it impractical to characterize a large number of comparators to obtain stochastic parameters such as offset and noise. This work presents novel circuit techniques to measure a large number of comparators (4,000) in a compact chip area to directly obtain silicon data including delay, noise, offset, and power. The proposed techniques also relax the requirement on the test instruments to measure the small time values. Four comparators with different transistor size ratios have been designed and measured to study the performance tradeoffs. In addition, this work presents a method utilizing supercomputing resources to simulate the large design space of the StrongARM comparator to observe the performance trends. Measurements are compared to simulations showing their accuracy and, for the first time, detailed study on the performance trends with different transistor size ratios.

Keywords: StrongARM, comparator, latch, input-referred noise, delay, energy, supercomputer cadence simulations
ACKNOWLEDGMENTS

Conducting the research for this thesis required the effort and support of those around me. I would like to thank my committee chair, Prof. Shiuh-hua Wood Chiang for his great insight and ideas that made this possible. I couldn’t have completed everything necessary for my master’s degree and this thesis without his help. I value the knowledge I have gained from working with him and other students in the lab. I would like to thank Yixin Song who has always been there to brainstorm ideas. I would also like to thank Matt Padilla, Alexander Petrie, Eric Swindlehurst, Khalil Gammoh, and Andres Galan. Without their help I wouldn’t be where I am today and I will always value the interactions I have had with them at BYU. I would also like to acknowledge and express my gratitude for the help of my other committee members, Prof. Michael Rice and Prof. Stephen Schultz. They have been instrumental in guiding me through my research and have helped me be successful in this endeavor.

Most of all, I would like to thank my wife Dawna, who has tirelessly supported me in every needful way while I worked towards completing my master’s degree. She was always there for the late nights and all-nighters to help and encourage me.
# TABLE OF CONTENTS

Table of Contents ....................................................... iv

List of Tables ........................................................... vi

List of Figures ........................................................... vii

Chapter 1  Introduction ................................................. 1
  1.1 Background ....................................................... 2
  1.2 Objective ........................................................ 4

Chapter 2  Theory and Analysis ....................................... 5
  2.1 Delay ............................................................. 7
  2.2 Noise ............................................................ 8
  2.3 Energy ........................................................... 11

Chapter 3  Design and Layout ......................................... 13
  3.1 Schematic Design ............................................... 13
    3.1.1 Delay ....................................................... 16
    3.1.2 Noise and Offset ......................................... 24
    3.1.3 Power ....................................................... 26
  3.2 Layout .......................................................... 26

Chapter 4  Experimental and Simulation Results ..................... 31
  4.1 Circuit Simulation on a Supercomputer ......................... 31
  4.2 Test Bench Setup ............................................... 33
    4.2.1 PC .......................................................... 33
    4.2.2 FPGA ....................................................... 36
    4.2.3 PCB ......................................................... 36
  4.3 Experimental Results .......................................... 39
    4.3.1 Delay ....................................................... 39
    4.3.2 Energy ...................................................... 43
    4.3.3 Offset ...................................................... 44
    4.3.4 Noise ....................................................... 46
    4.3.5 Energy-Noise and Energy-Noise-Delay Products .......... 51

Chapter 5  Conclusion .................................................. 56
  5.1 Future Research ................................................ 57

References ............................................................... 58

Appendix A  Test Bench Code .......................................... 60
  A.1 FPGA Code for Measuring Noise and Delay (Method 1) ....... 60
A.2 FPGA Code for Measuring Delay using PFD and DFF Outputs (Method 2) . . . . . . 80
A.3 Super Computer Scripts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
A.3.1 C SHell script . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
A.3.2 OCEAN script . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
# LIST OF TABLES

3.1 StrongARM latch transistor sizes ........................................... 13
3.2 NAND3 transistor sizes for each design .................................. 16

4.1 Delay values for the four designs in Fig. 4.7 .............................. 41
4.2 Energy values of the four designs in Fig. 4.9 ............................. 44
4.3 Standard deviation and mean of the offset. Mean of measured offset is larger than simulated mean due to additional mismatches in the DACs and nearby parasitics in the chip. ................................................................. 45
4.4 Mean-squared noise values of the four designs in Fig. 4.14 .......... 51
4.5 Energy-Noise Product (ENP in Joules-Volt$^2$) values of the four designs in Fig. 4.15 .... 52
4.6 Energy-Noise-delay Product (ENDP in Joules-Volt$^2$-seconds) of the four designs in Fig. 4.17 ......................................................... 54
# LIST OF FIGURES

1.1 Flash ADC ......................................................... 1

2.1 StrongARM Comparator Schematic .................................... 6
2.2 Transient Signals of StrongARM Comparator ........................ 6
2.3 Amplification Phase Equivalent Circuit ............................. 7
2.4 Example of cumulative distribution function (CDF) for StrongARM comparator noise .......... 9
2.5 Equivalent half-circuit for noise analysis in Phase 1 ................ 10

3.1 Schematic of comaprator chain ...................................... 14
3.2 Schematic of symmetrical 3-input NAND ........................... 15
3.3 Delay Cell ..................................................... 17
3.4 Schematic Programmed for Delay .................................. 18
3.5 Chain Mux .................................................... 19
3.6 Delay Waveform ............................................... 20
3.7 Loop Schematic ............................................... 21
3.8 PFD and DFF schematic ......................................... 22
3.9 $V_{PFD}$ and $V_{DFF}$ outputs ................................... 23
3.10 Offset Schematic ............................................... 25
3.11 Layout of StrongARM comparator prototypes ..................... 27
3.12 Layout of delay cells .......................................... 28
3.13 Electron micrograph of the integrated circuit chip ............... 29

4.1 Supercomputer simulation work flow ................................. 32
4.2 Block diagram of test bench setup ................................ 34
4.3 Picture of test bench setup ...................................... 35
4.4 PCB top layer ................................................ 37
4.5 PCB bottom layer ............................................. 38
4.6 Simulated and measured delay of the four StrongARM comparator designs ................. 40
4.7 Delay vs. Transistor Size ..................................... 41
4.8 Transient waveform of DFF and PFD signals ....................... 42
4.9 Energy vs. Transistor Size ..................................... 43
4.10 Offset Histograms ............................................. 45
4.11 Transient Waveform of Offset Mode ................................ 47
4.12 CDF noise plots .............................................. 48
4.13 Noise Histograms .............................................. 49
4.14 CDF noise plots .............................................. 50
4.15 Energy-Noise Product vs. $M_{1,2}$ and $M_{3,4}$ ..................... 52
4.16 Energy-Noise Product vs. $M_{3,4}$ ................................ 53
4.17 Energy-Noise-Delay Product VS. $M_{1,2}$ and $M_{3,4}$ ............. 54
4.18 Energy-Noise-Delay Product vs. $M_{3,4}$ ........................ 55
CHAPTER 1. INTRODUCTION

A comparator is a critical building block in analog-to-digital converters (ADCs) with a purpose of helping to convert an analog input voltage, $V_{in}$, into a digital output. The StrongARM comparator is a widely used latched comparator today. It was first introduced by Kobayashi et al in 1992 [1]. Originally developed to consume less power without compromising speed as a sense amplifier for SRAMs, the comparator is now used in many ADCs. The reason for its wide-spread adoption is because of its zero static power, rail-to-rail output, and its small input-referred offset [2] [3]. Fig. 1.1 is an example of the StrongARM comparator being used in a flash ADC, outputting a digital ’1’ or ’0’ based upon $V_{IN}$ and $V_1-V_3$. In this parallel architecture, poor performance, such as a high susceptibility to noise, directly affects the output of the ADC. Successive approximation register (SAR) ADCs utilize one comparator to create digital outputs and the se-
rial operation involves multiple clock cycles per sample. However, in both a flash ADC and SAR ADC, the speed of the comparator directly affects the ADC’s sampling frequency.

The SAR ADC in [4] uses a StrongARM comparator that consumes 21% of the ADC’s power. In applications with strict power constraints, the comparator’s power consumption must be considered. At the same time, the comparator needs to help maintain the ADC’s speed and resolution. Although the StrongARM comparator experiences the same trade-offs as most circuits, it will continue to be used in many ADCs and circuits alike owing to the reasons mentioned above. As CMOS circuits become more compact and power-efficient, either due to smaller feature sizes in newer processes or better designs, the StrongARM comparator will continue to be a major player in CMOS circuits, particularly in ADCs [5].

1.1 Background

The StrongARM comparator is a regenerative circuit. Like a flip-flop, the output will settle to two possible stable states [6]. This is due to a voltage offset that is generated and eventually with time, reaches a stable state. Ideally, the StrongARM comparator is a symmetric circuit and produces a symmetric binary output. As noted in [6], variations or transistor mismatches cause uncertainty of the final state. A level of certainty, or probability, of the outcome of the StrongARM comparator is due to circuit imbalances, or offsets. This is known as a systematic offset. A random offset is instead caused by electronic noise in the circuit. Although this probability can be found from empirical measurements, the circuit dynamics are complex to understand and are visualized in [6].

Analysis of the StrongARM comparator has focused on the different phases of operation [2] [6] [7]. As a dynamic circuit, its operation is often difficult to understand. Due to its complexity the design process heavily involves simulation to verify its performance [5]. It is therefore important to have both analytical and empirical results to make the design process more efficient.

A comparator’s performance is judged by accuracy, speed, and power. Accuracy is the comparator’s ability to decide the output correctly based on the input and is affected by offset and noise. In an ADC, the comparator’s accuracy due to noise contributes to the effective number of bits (ENOB) [8]. In almost every situation, a comparator is designed to be symmetric. Symmetry is important since the StrongARM comparator is a differential circuit and mismatch of
transistor sizes can cause an offset. These mismatches are due to fabrication imperfections. In addition, metal routings and other nearby circuitry may cause differences in parasitic capacitances and thereby affect the comparator’s symmetry. No matter the source of the offset, these imbalances lead to uncertainty in the comparator’s decision. In a paper by Abidi et al., these unbalances are explained [6].

Although a stable equilibrium is eventually reached, noise can lead to the same uncertainty in the comparator’s decision. Noise can be considered a temporary offset, whereas a permanent offset is caused by manufacturing variations and mismatches. All transistors experience noise that can be modeled as random fluctuations of drain current. The power spectral density of the drain current thermal noise is given by $I^2_n = 4kT\gamma g_m$ [9], where $k$ is the Boltzmann constant, $T$ is temperature in Kelvin, $\gamma$ is the thermal noise coefficient that is process dependent, and $g_m$ is the transconductance of the transistor. The flicker noise of the transistor is $I^2_n = \left(\frac{K_f}{f}\right)(1/C_{ox}WL)$ where $W$ and $L$ are the width and length and $C_{ox}$ is the oxide capacitance of the transistor. $K_f$ is a constant and is dependent on the process. It is because of these noise sources that the transistor can momentarily generate a spike or drop in voltage at one of its terminals and this causes it to behave differently from the ideal case. The StrongARM comparator’s susceptibility to these temporary changes constitutes its noise behavior. Noise is an important characteristic to understand in the StrongARM comparator because it affects the accuracy and is often underestimated or ignored.

The StrongARM comparator achieves good power efficiency since it draws no static power after the comparison is complete. Power is derived from transistor sizes and the clock rate at which it is operating. Noise is related to power through $g_m$ as given in the above equations. Also, by changing transistor sizes, noise and offset are affected as well. Since these parameters are dependent on each other, it is often difficult to design for a combined optimum of noise, speed, and power of the StrongARM comparator.

The StrongARM is fundamentally difficult to measure due to its dynamic, high-speed characteristic, unlike the traditional op amp which is static. Therefore, reported StrongARM’s performance in silicon has typically been inferred from that of the overall system (i.e. an ADC). But at the system level, it is difficult to isolate the contributions of the errors of the comparator from the rest of the system. Noise contributions, for example, are difficult to isolate. Furthermore, stochastic parameters such as offset must be measured from a large number of comparators. But this is
impractical in terms of area and cost when the measurement of a single comparator needs to be accompanied by a whole system such as an ADC. Due to the difficulties associated with measuring comparators, there is very little silicon data in the literature on comparator performance from direct measurement. In [10], only ten dynamic comparators are measured to find offset. Or, in [11] and [12] the offset is only simulated but not measured in silicon. Some design improvements on the StrongARM comparator have been done but lack measurements from fabricated comparators [13]. Moreover, there has been no work that systematically characterizes the comparator performance such as speed, noise, and power as a function of the transistor sizing in either silicon or simulations. Yongheng et al. presents a comparator design that is simulated for offset and delay, but does not explore how these parameters change for different transistor sizes [14]. While the StrongARM comparator is universally used in today’s data converters, the lack of silicon data and practical measurement techniques have limited our understanding of and ability to characterize this critical building block.

1.2 Objective

This research seeks to address the problem of StrongARM comparator measurements by demonstrating a compact silicon chip that can directly measure 4,000 comparators of different designs using novel, compact auxiliary circuits. The large-scale measurement provides critical silicon data on the comparator performance previously impractical to obtain. Furthermore, this work simulates the comparator performance across different transistor sizing with the aid of supercomputing to observe the trade-offs, and maps the silicon results to the simulations to validate the accuracy of the new measurement techniques.
CHAPTER 2. THEORY AND ANALYSIS

The StrongARM comparator (Fig. 2.1) is a dynamic circuit with four phases of operation: amplification, propagation, regeneration, and reset [6] [2]. These four phases make up one cycle, or sample, of the comparator. Amplification is initiated by a clock, $CK$ as shown in Fig. 2.2, and the propagation and regeneration phases follow automatically. During the amplification phase, the differential signal $V_{IN}$, where $V_{IN} = V_{IN}^+ - V_{IN}^-$ from Fig. 2.1, is amplified by $M_1$ and $M_2$. The amplification is a result of the transconductance, or $g_m$, of the NMOS transistors. Nodes $X$ and $Y$ are being discharged at a rate dependent on the parasitic capacitances and the gate voltage of $M_1$ and $M_2$. It is assumed that the tail transistor $M_7$ is an ideal switch and at time $t = 0$, $M_{1,2}$ both turn on in saturation. The amplification phase is ended when $V_{X,Y} = V_{DD} - V_{TH,N}$, where $V_{TH,N}$ is the threshold voltage for the NMOS transistors $M_{3,4}$. As a result, $M_{3,4}$ will start conducting current, and the propagation phase starts (see Fig. 2.2).

The propagation phase will propagate $V_{X,Y}$ to $V_{P,Q}$ as the parasitic capacitors on nodes $P$ and $Q$ discharge. When the voltages on both $P$ and $Q$ fall to $V_{DD} - V_{TH,P}$, $V_{TH,P}$ being the threshold voltage of a PMOS, the next phase starts as $M_{5,6}$ enter saturation. The regeneration phase is completed when the cross-coupled inverters resolve to a steady state. The comparator holds the outputs, $P$ and $Q$, and sits in a static state consuming no power. At this point, the comparator’s outputs are valid, and in most applications, are sampled by other circuitry. Before a new cycle can begin, the comparator must be fully reset by the falling edge of $CK$. The reset phased is finished when $P$, $Q$, and the internal nodes $X$ and $Y$ reach the supply voltage. Any time one of these nodes doesn’t fully reach the supply at the end of reset, there will be an undesired offset introduced in the succeeding sample.
Figure 2.1: StrongARM comparator schematic. $V_{IN}^+$ and $V_{IN}^-$ are the two analog voltage inputs and $P$ and $Q$ are the two outputs. $S_{1-4}$ are reset transistors that pull internal nodes back to $V_{DD}$ during the reset phase.

Figure 2.2: Transient waveform for StrongARM comparator. Each cycle consists of four phases: amplification, propagation, regeneration, and reset. Output is valid at the end of the regeneration phase.
2.1 Delay

The total delay, or time for all 4 phases to complete, is measured from the time \( CK \) crosses \( V_{DD}/2 \) to the time the differential output \( |V_P - V_Q| \) crosses \( V_{DD}/2 \). However, only the delay for phase 1 is derived, since this is the delay needed to calculate the noise equation derived in section 2.2. The equivalent circuit of the StrongARM comparator in the amplification phase is shown in Fig. 2.3. All other transistors are off and \( M_7 \) is assumed to be an ideal switch, so only \( M_1 \) and \( M_2 \) are considered. The parasitic capacitance \( C_X, Y \) is the source of the bias current flowing through \( M_{1,2}, I_1 \). Half circuit analysis [Fig. 2.3] is used to find the delay of the amplification phase, or the time when \( V_x = V_{DD} - V_{TH,N} \). The equation for \( V_x \) as a function of time is

\[
V_X(t) = V_{DD} - \frac{I_1 t}{C_X}.
\]  

(2.1)

The initial condition is when \( t = 0 \) and \( V_X = V_{DD} \), so the change in voltage, \( \Delta V_X \), is represented by \( \frac{I_1 t}{C_X} \) in Eq. 2.1. This results in

\[
\Delta V_X = \frac{I_1 I_{D1}}{C_X} = V_{TH,N}
\]  

(2.2)
From eq. 2.3, the delay of the first phase is inversely proportionally to \( I_1 \) which depends on the voltage at the gate of the transistor. When \( t = t_D \) the current \( I_1 \) flowing through \( M_1 \) has discharged the capacitor \( V_{DD} - \Delta V_X \). The

In phase 2, propagation, the delay as defined in [6] is

\[
t_{D2} = \frac{(C_{PQ} + C_{XY})|V_{TH,P}|}{I_1}.
\]

\[(2.4)\]

### 2.2 Noise

Clocked comparators have been previously analyzed with stochastic differential equations [5], linear-time-varying models [7], and simplified noise circuits to find noise [2]. Normally, to find noise of an amplifier, the output noise is divided by the gain to find the input-referred noise. Input-referred noise is used because it is a fairer comparison between different circuits. Although the comparator output signals (\( P \) and \( Q \)) are analog voltages, they turn into a digital signal once they have settled. So a method different than measuring noise of a conventional amplifier is used for measuring the StrongARM comparator and has been used previously [2].

A simple way to understand and measure the noise of the StrongARM comparator is to observe the probability that it outputs a ‘1’ (where \( Q = V_{DD} \) and \( P = V_{SS} \)) for a range of differential input voltages (i.e. \(-V_{DD}/2 \leq V_{IN} \leq V_{DD}/2\)). Because the probability is compared (or referred) to a range of input voltages, it is input-referred. Assume the StrongARM comparator, with no offset, is applied with a 0V differential voltage. The output could result in either a ‘1’ or a ‘0’ (where \( Q = V_{SS} \) and \( P = V_{DD} \)) with equal probability, \( P(1 \mid V_{IN} = 0V) = .5 \), due to circuit noise. However, for a non-zero, positive differential voltage, a ‘1’ is more likely to be produced, \( P(1 \mid V_{IN} = 100 \mu V) = .60 \). A ‘1’ won’t occur 100% of the time for small differential input voltages because the noise from the transistors may generate an internal voltage greater than the differential input voltage thus causing an incorrect output. For large differential input voltages, both positive and negative, it may not be possible for the noise to affect the decision. This is when a ‘0’ or a ‘1’ is produced nearly 100% of the time. The noise of a comparator is found by fitting a cumulative distribution function to the
sampled output points. Fig. 2.4 is a plot showing the probability of the output of a ’1’ as a function of the differential output voltage.

![Example CDF of StrongARM latch](image)

Figure 2.4: The cumulative distribution function (CDF) for the probability that a ’1’ is produced. Each simulated point represents the percentages of 1’s for a given differential input voltage of 2000 samples.

Although the measurement method above is good for simulation or IC chip testing, an equation for noise is found using small-signal analysis. During phase 1, $M_{1,2}$ exhibits thermal, or white noise, $i_n$, which is represented as an ideal current source. Using the equivalent circuit in Fig. 2.5 the output-referred noise at node $X$ can be calculated. The noise voltage at node $X$, $V_{nx}$, is a function of time and is given by

$$V_{nx}(t) = \frac{1}{C_X} \int_0^t i_n(\tau)d\tau,$$

(2.5)

where $C_X$ is the total parasitic capacitance at node $X$, $t$ is time, and $i_n$ is the noise current. Modeling $i_n$ as a white noise process, $V_{nx}$ is a Wiener process that represents the random walk of the voltage $V_{nx}$. This equation shows that $V_{nx}$ is a function of $C_X$ and $t$. Changing the parasitic capacitance by
changing the transistor size affects the output-referred noise and consequently the input-referred noise. By making the width of $M_{1,2}$ larger, the Wiener process $V_{nx}$ will have a smaller variance, $\sigma^2$. Additionally, a smaller variance can be achieved by a smaller $t$. In the amplification phase $t = t_{D1}$.

To find the output-referred noise at node $X$, a few definitions are applied. First, $\bar{V}_{nx}^2(t_{D1})$ is really just the variance so $\bar{V}_{nx}^2(t_{D1}) = E\{V_{nx}^2(t_{D1})\}$. Expanding this equation, we have

$$E\{V_{nx}^2(t_{D1})\} = \frac{1}{C_X^2} \int_0^{t_{D1}} \int_0^{t_{D1}} E\{i_n(\tau_1)i_n(\tau_2)\} d\tau_1 d\tau_2. \quad (2.6)$$

Since $i_n$ is white noise, $E\{i_n(\tau_1)i_n(\tau_2)\}$ is the autocorrelation function which is a delta function by definition. The autocorrelation is then

$$R(\tau_1, \tau_2) = E\{i_n(\tau_1)i_n(\tau_2)\} = q(\tau_1) \delta(\tau_1 - \tau_2), \quad (2.7)$$

where $q(\tau_1)$ is the power spectral density. The results of substituting 2.7 into 2.6 is Eq. 2.8.

$$E\{V_{nx}^2(t_{D1})\} = \frac{1}{C_X^2} \int_0^{t_{D1}} \int_0^{t_{D1}} q(\tau_1) \delta(\tau_1 - \tau_2) d\tau_1 d\tau_2 \quad (2.8)$$

The sifting, or sampling property states that $\int \delta(x - c)f(x)dx = f(c)$. Using the sifting property on Eq. 2.7 results in Eq. 2.9.

$$E\{V_{nx}^2(t_{D1})\} = \frac{1}{C_X^2} \int_0^{t_{D1}} q(\tau) d\tau, \quad (2.9)$$
where $q(\tau)$ is the power spectral density (PSD) of white noise and is a constant so Eq. \ref{eq:2.9} simply becomes

$$E\{V_{mx}^2(t_{D1})\} = \frac{S_{IN}}{C_x^2} t_{D1}. \quad (2.10)$$

By increasing transistor sizes (i.e. increasing $C_X$) the noise decreases. For NMOS transistors $S_{IN} = 4kT\gamma g_{m1.2}$ and since Eq. \ref{eq:2.10} represents the noise of the half circuit, the mean-squared voltage of the differential circuit of Fig. 2.3 from phase 1 is

$$\overline{V_{mx}^2(t_{D1})} = \frac{8kT\gamma g_{m1.2}}{C_{X,Y}^2} t_{D1}. \quad (2.11)$$

Although noise found using Eq. \ref{eq:2.11} is output-referred at nodes $X$ and $Y$ only, this equation shows how the transistor sizes for $M_1 - M_4$ affect the noise in phase 1. Phases 2 and 3 also add noise but Phase 1 contributes the most amount of noise. \ref{eq:2.11} provides sufficient insight of the overall noise of the comparator. The flicker noise is ignored since it only contributes low-frequency noise and white noise dominates in the total noise power.

### 2.3 Energy

Energy is another important characteristic of a circuit and is used to compare different designs of StrongARM comparators for power efficiency. The dynamic energy consumed during one comparator cycle is related to the amount of energy stored in the parasitic capacitors $C_X$, $C_Y$, $C_P$, and $C_Q$. The energy of a capacitor is expressed as

$$E_{cap} = \frac{1}{2}CV_{DD}^2. \quad (2.12)$$

However, half of the energy is lost to heat when a capacitor is charged, so the energy consumed is actually $CV_{DD}^2$. Since the internal nodes are charged back to $V_{DD}$ during the reset phase, we can understand the StrongARM comparator’s energy consumption by the following:

$$E = (C_X + C_Y + C_P)V_{DD}^2. \quad (2.13)$$
Written as average power

\[ P_{\text{avg}} = f_{\text{CK}} (C_X + C_Y + C_Q) V_{DD}^2, \]  

(2.14)

where \( f_{\text{CK}} \) is the frequency of the comparator clock.

Eq. 2.13 omits the parasitic capacitance of node \( Q \) because it is guaranteed that only one of the outputs will be discharged down to \( V_{SS} \). Although there is static current draw from \( V_{DD} \) during the propagation and regeneration phases at node \( Q \), it is minimal for most inputs. The static current draw becomes more substantial when \( V_{IN} \) is small. For an example of this, see Fig. 2.2 where \( Q \) draws current as it charges back up to \( V_{DD} \) in the regeneration phase. Otherwise the results from Eq. 2.13 has an error less than or equal to 3.8\% when \( 50mV \leq V_{IN} \leq 970mV \). If \( V_{IN} = 1.8V \), then either \( C_X \) or \( C_Y \) won’t discharge to \( V_{SS} \) because one of the input transistors (\( M_1 \) and \( M_2 \)) will never turn on. In this case Eq. 2.13 becomes \( E = (C_X + C_P) V_{DD}^2 \).
CHAPTER 3. DESIGN AND LAYOUT

3.1 Schematic Design

Chip fabrication is valuable to understand and validate the behavior and design trade-offs of the StrongARM comparator. The design of the proposed chip is focused on flexibility and is capable of measuring the delay, offset, and noise from a large number of StrongARM comparators. Instead of building test structures that can only measure one type of performance parameter from the comparators, a universal test architecture is proposed to measure delay, offset, and noise. The schematic of the proposed circuit is shown in Fig. 3.1. Four copies of this circuit, each implementing a slightly different comparator design, are fabricated on the same chip to study performance tradeoffs. The only components that change in Fig. 3.1 between different design copies are the StrongARM comparators. By re-using the same comparator for different measurements, better area efficiency in layout allows for more StrongARM comparators to be used. Each design resembles the circuit of Fig. 2.1 but each design copy has different transistor sizes for $M_{1-4}$ and are referred to as Designs 1-4. Table 3.1 shows the transistor sizing for each design of the StrongARM comparator.

<table>
<thead>
<tr>
<th></th>
<th>Design 1</th>
<th>Design 2</th>
<th>Design 3</th>
<th>Design 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{1,2} , (\mu m)$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>$M_{3,4} , (\mu m)$</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>$M_{5,6} , (\mu m)$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$M_{7} , (\mu m)$</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>$S_{1-4} , (\mu m)$</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
</tr>
</tbody>
</table>
Figure 3.1: Schematic of proposed comparator characterization chip to measure delay, offset, and noise of StrongARM comparators.
The switches in Fig. 3.1 are controlled by a shift register and the switch’s label is the mode it is associated with. The modes are chain, loop, and offset; each is responsible for measuring a particular type of comparator performance. The reset switch is on only in offset mode and isn’t a mode by itself because its purpose is only to prevent accidental oscillation of $V_{1...N}$. Chain mode is for delay measurements using a reference clock ($CK_{PASS}$), loop mode is for delay measurements using a phase-frequency detector (PFD) and D-flip-flop (DFF), and offset mode is to measure both offset and noise. All modes will be discussed in the following sections with schematics highlighting how the circuit in Fig. 3.1 is utilized.

The 3-input NAND gate, NAND3, connected to the two comparator outputs in Fig. 3.1 is designed to have equal loading. This prevents unwanted offsets from loading mismatch. $P$ and $Q$ of the comparator connect to $A$ and $B$ of the NAND3 (Fig. 3.2). Commonly, a NAND3 gate consists of three series NMOS transistors between the output, $Y$, and $V_{SS}$ [15]. However, the signal controlling the top NMOS, the NMOS nearest the output, will have different loading affects than the other signals. To have equal loading for signals $A$ and $B$ (signal $C$ does not need to be symmetrical to $A$ and $B$ in this circuit) two parallel branches of two NMOS transistors in series are between $Y$ and $V_{SS}$ where signals $A$ and $B$ both control a top and a bottom NMOS transistor. Designs 1-3 have the same NAND3, but since design 4 has smaller transistors for driving the

![Figure 3.2: Schematic of the 3-input NAND from Fig. 3.1 designed to provide symmetrical gate capacitance on $A$ and $B$ which connect to the output of the StrongARM comparator.](image-url)
Table 3.2: NAND3 transistor sizes for each design

<table>
<thead>
<tr>
<th></th>
<th>Design 1</th>
<th>Design 2</th>
<th>Design 3</th>
<th>Design 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{1-4}$ ($\mu$m)</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td>$M_{5,6}$ ($\mu$m)</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>0.8</td>
</tr>
<tr>
<td>$M_7$ ($\mu$m)</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td>$M_8$ ($\mu$m)</td>
<td>2.18</td>
<td>2.18</td>
<td>2.18</td>
<td>2.18</td>
</tr>
</tbody>
</table>

outputs $P$ and $Q$, the NAND3 is slightly different to maintain the same rise and fall time of $Y$ (table 3.2).

### 3.1.1 Delay

The architecture in Fig. 3.4 shows the circuit in (Fig. 3.1) configured for chain mode. Chain mode turns on the two $CHAIN$ switches and arranges the comparators to have a start node ($V_1$) and an end node ($V_{OUT}$). Everything that isn’t being used in chain mode is grayed out in Fig. 3.4. A clock tree distributes on off-chip clock, $CK_{IN}$, evenly across the chain of comparators, but only the first comparator sees this clock since all 2-input NAND gates output a ‘1’ due to the select signals $S_1...N$. This mode is similar a chain of inverters with one state triggering the next. A stage consisting of a comparator and combinational logic is shown in Fig. 3.3. The signals $V_1...N$ behave similarly to a propagating clock signal in a chain of inverters.

The delay of the entire chain of $N$ delay cells is observed by comparing the time between edges of $V_{OUT}$ and $CK_{PASS}$. The delay of only one comparator isn’t measurable since measuring a hundred picoseconds off-chip is impractical. But by having 1000 comparators arranged in a chain, the delay is amplified and therefore the total delay is a few hundred nanoseconds instead of picoseconds. This is measurable off-chip, and the instruments can give an accurate delay reading of the total delay. Note that this time difference includes the delay of the 3-input NAND but not the delay of both the inverter right before $V_{LAST}$ and the switch after $V_{LAST}$. The inverter and switch is common in the path of $CK_{PASS}$ and $V_{OUT}$ (see Fig. 3.4).
Figure 3.3: Delay cell.
Figure 3.4: The entire schematic when configured in chain mode to measure the delay.
As previously mentioned, delay between $V_{OUT}$ and $CK_{PASS}$ is the absolute delay of the design. By comparing $V_{OUT}$ of other designs, however, the relative delay can be measured. Fig. 3.5 shows how $V_{OUT}$ from any of the four designs and $CK_{PASS}$ can be routed to $OUT_0$ and $OUT_1$. The configurations where $OUT_0$ and $OUT_1$ are sourced from the same signal are meaningless when it comes to delay. The muxes are implemented using transmission gates that are controlled by signals in a shift register. Fig. 3.6 illustrates how the reset and delay times are measured. For example, if $V_{OUT,1}$ was routed to $OUT_0$ and $V_{OUT,3}$ was routed to $OUT_0$, the delay between the two falling edges, as shown in Fig. 3.6 would be the relative delay between design 1 and design 3. Relative delay is mainly used to double check the trend of the absolute delay that was measured.

![Diagram](image)

Figure 3.5: Muxes to route $CK_{PASS}$ and $C_{OUT,1-4}$ to output pads $OUT_{0,1}$.

The delay is not a constant but changes as a function of the input voltage, $V_{IN}$. The smaller the differential voltage applied across the two inputs, the longer the delay [6]. The number of delay cells needed in a chain are based on the lowest delay because of a larger $V_{IN}$. A length of 1000 delay cells, or $N = 1000$, was chosen to strike a balance between chip area and requirements on the measurement instruments. An oscilloscope capable of detecting a timing difference of 1 ns is sufficient for this test chip. If $\Delta t_{total} = 1\, ns$, then the smallest detectable change in one comparator
in the chain is \( \Delta t_{\text{comparator}} = 1 \text{ps} \). Essentially, \( N \) is chosen as a way to amplify the delay of a comparator design. For example if \( N = 1 \) then \( t_d \) is on the order of picoseconds, far too small to capture off-chip with an oscilloscope.

An alternative method for measuring delay is proposed. This method’s plausibility is shown using the same 1000-delay cell chains but would be more useful over the previous method for smaller chains with faster oscillation frequencies where \( t_d \) and \( t_r \) are too small to measure off-chip. For this second method of delay measurement the original circuit (Fig. 3.1) is configured in loop mode (Fig. 3.7). The first comparator, or first stage, is no longer clocked by \( CK_{IN} \) but instead by the output of the last stage, or \( V_{\text{LAST}} \). The comparators basically behave as a 1000-stage ring oscillator.

Due to electronic noise, the comparator loop will begin to oscillate without any external stimulus. An inquiry emerges of how many different signal propagations, or how many rising and falling edges, exist in the loop. Simply answered, it doesn’t matter how many edges exists on the signals \( V_{1...N} \) initially. After some time, all independently propagating signals will converge to one rising edge. The comparators will hold their outputs causing the 3-input NAND to not produce another rising edge until the comparator has been reset again. This means all but one rising edge will naturally die off in the loop. The rising edge will overtake any falling edges in the loop due to the slower propagation rate of the negative edge. The negative edge, or reset edge, propagates more slowly because the comparator is slower at resetting than being clocked. Steady state of the loop is reached where there only exists one rising edge or one falling edge in the comparator loop at a time. Simulations show that multiple signals converge after a short period of ”chasing” each other.
Figure 3.7: Schematic when programmed in loop mode to measure delay.
Consider the case when the rising or falling edges are near one another and an oscilloscope, with limited time resolution, provides an unreliable time measurement between the two edges. Instead of capturing signals $A$ and $B$ on an oscilloscope and observing the delay that way, the PFD and DFF circuits in Fig. 3.8 allow the delay to be correctly measured. The $V_{PFD}$ waveform has

time-varying pulse widths, but each subsequent pulse width is different by $\Delta p$, and is the time by which signal B’s period is different by signal A’s period. A positive $\Delta p$ indicates signal A has the smaller period and a negative $\Delta p$, the opposite. The circuit operation is simple, $A$ sets $V_{PFD}$ high and $B$ resets $V_{PFD}$ low. If signals $A$ and $B$ are close in frequency then $\Delta p$ will be small and may need to be calculated using pulses further away from each other, $\Delta p_{far} = p_4 - p_1$ so $\Delta p = \Delta p_{far}/3$.

The repeating pattern of $V_{PFD}$ is captured by $V_{DFF}$ which is the beat frequency. Since $DFF$ is the beat frequency it indirectly finds the difference in delay of two chains. Note that $V_{DFF}$ doesn’t determine whether $f_A$ or $f_B$ is faster but $V_{PFD}$ does. If the $V_{PFD}$ pulse width, $\Delta p$, is increasing then $f_A > f_B$ (as shown in Fig. 3.9), or if it is decreasing then $f_A < f_B$. Combining information from both $V_{PFD}$ and $V_{DFF}$, both the relative speed and absolute value of the oscillation frequencies of the two chains are found.

---

Figure 3.8: Schematic of the 4-to-1 MUXs and PFD. This allows for any combination of signals $V_{LAST,1−4}$ to be selected for the PFD and DFF circuits.
Starting with the following equation

\[ T_{BEAT} = \frac{T_A T_B}{T_B - T_A}, \]  

(3.1)

we know \( T_{BEAT} \) from \( V_{DFF} \) but this leaves two unknown variables. Using \( V_{PFD} \) we can find \( T_B \) using the following equation

\[ T_B = \Delta p + T_A. \]  

(3.2)

Then substitute 3.2 into 3.1 to get

\[ T_{BEAT} = \frac{T_A (T_A + \Delta p)}{\Delta p}. \]  

(3.3)

From the quadratic formula

\[ T_A = \frac{-\Delta p + \sqrt{(\Delta p)^2 + 4 \times \Delta p \times T_{BEAT}}}{2}, \Delta p \geq 0. \]  

(3.4)

The other solution of the quadratic formula is not used since it gives a negative period for \( T_A \) which is physically not possible. Once \( T_A \) has been solved for using Eq. 3.4 then \( T_B \) can be found using 3.2. Eq. 3.1 also proves that this method is superior when the delay \( t_r \) or \( t_d \) is too small to measure.

Figure 3.9: \( V_{PFD} \) and \( V_{DFF} \) waveforms. Inputs A and B are the oscillation signals from the chain of comparators.
off-chip. The closer the two signals are in frequency, i.e. when $T_B - T_A$ is smaller, $T_{BEAT}$ is larger and thus making $V_{DFF}$ and $V_{PFD}$ easier to observe.

### 3.1.2 Noise and Offset

If the circuit is programmed into the mode shown in Fig. 3.10 then comparators can be measured for noise and offset. Since it requires too many wires to measure the output ($P$ and $Q$) of all 1000 comparators simultaneously only one comparator is selected and measured at a time using one wire route named $Q_{OUT}$. This gives two advantages: one, this makes the layout simpler, and two, this reduces the pin count on the package for the comparator to one since it is impractical to have 1000 pins for 1000 comparators.

The trade-off is that the chip must then be programmed every time a new comparator is selected. Each comparator is selected using the $S_{2...N}$ signals which gate $CK$ with a 2-input NAND gate. If $S_3 = '0'$ for example, then $V_3$ is held at ‘0’ causing the corresponding comparator to not be clocked. However, if $S_2 = '1'$ as shown in Fig. 3.10, then $V_2 = CK$ but delayed slightly due to propagation delay through both NAND gates. $S_2$ is also routed to a switch that allows $Q$, one of the outputs of the comparator to be eventually connected to $V_{OUT}$.

The select signals provide flexibility and ensures that only one comparator pulls $Q_{OUT}$ low at any one time. This allows the least amount of loading on the outputs of the comparator. If instead $Q$ was driving a logic gate, an inverter for example, which in turn was driving a long metal route, then the comparator would be driving more parasitic capacitance. The $Q$ switches are implemented with a single NMOS transistor to reduce loading. If $Q = 1$ then $Q_{OUT}$ is driven low because the two NMOS connect it to ground and over powers the PMOS bleed transistor. If $Q = 0$, then the opposite happens and the PMOS bleed transistor continues to pull $Q_{OUT}$ high. $P$ drives a dummy switch to maintain a symmetric loading. $P$ is not routed to the output since its value is always opposite of $Q$ after the comparator completes the comparison.
Figure 3.10: Schematic when programmed in offset mode, or to measure offset and noise of all the StrongARM comparators.
3.1.3 Power

Power of the StrongARM comparator is measured using the average current and voltage of $V_{DD,1}$ which is the power supply for the comparators and shift register. There is no need to separate out the power consumed by the shift register since it is static and will draws no dynamic power during normal operation and its static power is negligible compared to the comparator power. It is only during the programming operation that dynamic power is consumed in the shift registers. The circuit is in a modified loop mode (see Fig. 3.10). Instead of allowing all designs to be oscillating, only one design is programmed into loop mode where the others are in reset. This ensures that the current measurement is only from one design of comparator and not all four.

The average current needs to be adjusted for the current draw of the NAND3 gate. An extracted simulation, a circuit simulation that captures parasitic capacitances and resistances from layout, provides a good approximation of the current used by the NAND3. It is then subtracted from the measured current to find the real power consumed by the comparators.

3.2 Layout

Layout contributes parasitics such as capacitance and resistance. Comprised of different layers and shapes, the silicon based layout determines the end behavior of the circuit. It is a representation of the schematic as a planar, geometric design, and if it is poorly done, the circuit will not behave as intended. Even if a schematic is symmetric, the layout isn’t guaranteed to have the ideal symmetry of the circuit due to manufacturing defects and variations. These variations present themselves as offset in the StrongARM comparator. The use of dummy transistors and layout guidelines help mitigate mismatches in layout and can help preserve symmetry and transistor sizing.

To avoid ion channeling during implantations, the wafer is set at a slight angle [9]. However, the gate polysilicon causes a shadow during ion implantation due to this angle. This is known as ”gate shadowing” and needs to be considered to achieve symmetric layout [9]. If the polysilicon is perpendicular to the line of symmetry of the comparator, then the effects of gate shadowing are common on both sides of the comparator and the fabricated circuit will have smaller offset (Fig. 3.11).
Figure 3.11: Layout of the four StrongARM comparator designs. Gate polysilicon is oriented perpendicular to line of symmetry to avoid asymmetrical effects from gate shadowing [9]. Each comparator design has same dimensions to make layout less manual.

If the polysilicon is parallel to the line of symmetry then although the layout is drawn symmetrically, the transistors will have more offset from gate shadowing. Correct perpendicular configuration is annotated on the layout image of the four different comparator designs in Fig. 3.11.

The layout of each comparator, although containing different transistors sizes (see Table 3.1), occupies the same overall area. This allows for an easy "plug-and-play" of the comparator designs into the same delay cell circuitry. Fig. 3.11 shows how each comparator design has the same vertical and horizontal dimensions. Layout for only one comparator cell is needed when it can be arrayed together. A snippet of this array is shown in Fig. 3.12, where three copies of the same comparator cell are abutted. Instead of 1000 comparator cells laid out linearly, the comparators are laid out in a snake-like fashion to make the dimensions of the chip more practical. The snake-like structure is shown in Fig. 3.11 which is an electron micrograph of the integrated circuit chip. Highlighted areas, 1, 2, 3, and 4 show areas of different StrongARM comparator designs.

The $CK_{IN}$ clock tree is embedded in the two dimensional chain but isn’t part of the delay cell since a clock buffer isn’t needed for every comparator. An additional clock tree exists for
the shift register and is implemented the same as the $CK_{IN}$ tree; however, it contains connections between some leaf nodes where flip-flops are spatially separated because the flip-flops are a part of different design chains.

There are multiple $V_{SS}$ pads to ensure the return path from the three different power supplies, $V_{DD1,2,3}$, aren’t current limited. $V_{DD1}$ is a 1.8 V supply and is the core voltage that supplies power to the comparators and the shift register. This allows power of only the comparators to be measured because once the chip has been programmed the shift register does not consume dynamic
Figure 3.13: Electron micrograph of the integrated circuit chip. The chip includes all 4000 StrongARM comparators (1000 comparators per design). The snake-like layout of each chain of 1000 comparators makes the chip dimensions practical. It also makes the shift register and sequential clocking (as shown in Fig. 3.1) easy to implement with minimal metal routing between each stage. PFD and DFF circuits are labeled and are located near the output buffers.
power. \(VDD_2\) supplies 1.8 V to the clock tree for \(CK_{IN}\). The third voltage supply, \(VDD_3\), is the 1.8 V power supply of the drivers for all the output signals: \(DFF\), \(PFD\), \(OUT_0\), and \(OUT_1\) and also for the input buffers on \(D\) and \(CKS\) for the shift register.
CHAPTER 4. EXPERIMENTAL AND SIMULATION RESULTS

4.1 Circuit Simulation on a Supercomputer

To adequately measure the input-referred noise of the StrongARM comparator, 2,000 samples are needed per input voltage. A small sample set doesn’t produce consistent results, but 2000 samples are sufficient if the other simulation parameters are set properly. In a transient noise simulation, noise sources cause the comparator to make decisions according to the stochastic properties of the circuit. After the comparator is clocked, a sampling function samples the output of the comparator to determine the percentage of 1’s.

The $F_{\text{max}}$ parameter is the circuit simulator parameter that defines the maximum noise frequency simulated. For a circuit like the StrongARM comparator, transient changes occur in fractions of a nanosecond and some signals may not experience any noise in such a small time window for a low noise frequency. To find an appropriate $F_{\text{max}}$, results (percentages of 1’s for a given input voltage) are compared as $F_{\text{max}}$ is increased. Once the results do not change noticeably when $F_{\text{max}}$ is increased, the appropriate $F_{\text{max}}$ has been found. Using a $F_{\text{max}}$ higher than necessary only increases simulation time and has no significant effect on the results. An $F_{\text{max}}$ of 250 GHz was used in the simulations described next. The second noise parameter is $F_{\text{min}}$ and it defines the lower noise frequency boundary. When set at a very low value, 1 kHz for example, it is automatically changed based upon the transient simulation time. An $F_{\text{min}}$ of 50 kHz was automatically selected based upon the 20 $\mu$s simulation time. If one 20 $\mu$s transient simulation consisting of 2,000 comparator samples requires over an hour of wall time on one CPU, the compute time becomes prohibitively large when iterating over multiple input voltage values and transistor sizes.

For 20 different input-voltage steps (2,000 samplers per step) and 100 different transistor sizes, $4e6$ samples need to be simulated. To reasonably obtain this data, simulations are computed on a supercomputer. Spectre is the simulator tool being used inside of Cadence Virtuoso, a circuit design suite. However, Cadence does not natively support the type of job manager on the super-
computer. Instead, Cadence uses a scripting language, Open Command Environment for Analysis (OCEAN) to extend the design environment. The OCEAN script is a set of Cadence commands and can be run from the command line. However, environmental variables and library directories need to be set up before running the OCEAN script. In Fig. 4.1, there is a C SHell wrapper that does this and then calls the OCEAN script (see A.3 for both C SHell and OCEAN scripts). This is necessary when running the circuit simulator on the supercomputer since user interaction via a GUI is not possible. The simulation job needs to be submitted to a scheduler, which is SLURM in this case. Once a job has been submitted, the scheduler allocates resources as they become available. These resources include: memory, number of CPUs, number of nodes, and wall-time. This work flow is illustrated in Fig. 4.1. It is important that the resources are chosen appropriately for the demands of the job to avoid bottlenecks such as memory or low number of CPUs. Requesting a lot of resources may take longer for the job to start because the scheduler has to wait until the specified resources become available. This is inefficient when many resources are requested but not fully utilized in the job.

Figure 4.1: Using this work flow, the supercomputer is used to run a simulation in 24 parallel jobs greatly reducing simulation time.

Fig. 4.1 shows the job being distributed by the SLURM workload manager to multiple nodes, each containing 24 CPUs. The job is manually broken into 5 smaller jobs, each being
submitted to the scheduler (5 jobs of 20 different transistor sizes). Each of the smaller jobs take 5 hours to complete using the following resources: memory = 32 GB, nodes = 1, numCPU = 24. The 5 output files, or PTSFD files (file extension for Cadence simulation files) are processed by the OCEAN scripts where a sampling function (see line 131 in OCEAN script in section A.3.2) outputs a comma-separated values file (.csv). These csv files are read by MATLAB for curve-fitting the CDF to create a 3D surface plot (see Fig. 4.14). What would otherwise require 2,000 hours of wall time to simulate a total of 40 ms of circuit operations on one CPU can be done in 5 hours on a supercomputer.

4.2 Test Bench Setup

The diagram and picture for the test bench setup are shown in Fig. 4.2 and Fig. 4.3, respectively. The integrated circuit chip that contains the StrongARM comparator prototypes is placed on a PCB for testing. The PCB is the interface for the IC chip to the rest of the test bench which includes a field-programmable gate array (FPGA), oscilloscope, RF generator, and power supply units. The RF generator provides either a 2 MHz or 300 kHz clock for offset or delay mode, respectively. This clock is connected to the $CK_{IN}$ pad which gets routed to the $CK$ signal in the delay cell in Fig 3.3. Power supply units are used to power the FPGA evaluation board at 5V, and the three 1.8V power supplies to the chip $VDD_1$, $VDD_2$, $VDD_3$. The main components of the test bench are described in the sections below.

4.2.1 PC

The PC is used as the central control for all the testing. It initiates tests by sending a command code over the RS-232 UART to the FPGA and analyzes data gathered from both the FPGA and oscilloscope. By making the PC the central controller of the test bench, many of the different tests are automated. A Matlab script controls the RS-232 8-bit UART interface and receives noise and offset data from the FPGA. The same script analyzes the data. For example, noise and offset are calculated using a least-square curve fitting for the error function to generate the CDF. Mean and variance are then used to find offset and noise, respectively.
Figure 4.2: Test bench setup has a Spartan-3E FPGA, two 16-bit digital-to-analog converters (AD5541A), RF signal generator, power supplies, oscilloscope, and a PC.
Figure 4.3: Picture of Spartan-3E FPGA with two 16-bit digital-to-analog converters (AD5541A) connected via header pins on the evaluation board. In addition, a level shifter peripheral PCB that is connected to the FPGA shifts the chip voltage, 1.8 V, to the 3.3 V FPGA voltage domain. SubMiniature version A (SMA) connectors $O_{UT_0}$ and $O_{UT_1}$ are for the oscilloscope, but wires soldered to the backside of the SMA connectors go to the FPGA. The SMAs are for connecting to the oscilloscope.
4.2.2 FPGA

The field programmable gate array (FPGA) is a Spartan-3E on a Digilent Nexys2 evaluation board and executes commands coming from the PC. Each command contains the type of mode (delay, loop, and offset) along with the information for connecting \( OUT_{0,1}, V_{PFD}, \) and \( V_{DFF} \) through the MUXES in Figs. 3.5 and 3.8. Based upon this command the FPGA will program the IC chip through a serial interface, \( CKS \) and \( D \). Once the shift register has been programmed, when the 4023 bits have been loaded using a clock and data serial interface, the FPGA goes into standby until more commands are received. For offset mode, the PC sends additional commands that set the 16-bit DACs and the FPGA returns the number of 1’s sampled on \( Q \). See section A.1 and A.2 in the appendix for VHDL code that implements the testing automation of the chip as well as the command and status handling from and to the PC.

4.2.3 PCB

The custom PCB has a voltage regulator (component U2 in Fig. 4.4) for the core voltage and decoupling capacitors to minimize noise of the power supply signals. The clock and digital signals are shielded from the power supply and analog input voltages. \( V_{IN}^+ \) and \( V_{IN}^- \) in Fig. 4.2 are DC voltages controlled by a DAC, but small perturbations affect the outcome of the comparators in the IC chip. So, it is critical to have stable input voltages by using de-coupling capacitors. Decoupling capacitors are used on both the top and bottom layer of the PCB, Fig. 4.4 and 4.5. Capacitors components are prefaced with a ’C’ and are placed near the pins of the chip to reduce resistance between the capacitors and the pins.

The ground plane, or VSS plane, on the PCB is connected to ground of the power supplies and the ground of the FPGA’s PCB. The ground of the power supply is connected to the earth ground. The RF generator and oscilloscope have floating grounds (grounds that do not connect to the earth ground). This prevents ground loops and helps minimize noise in the test bench setup.
Figure 4.4: The top layer of the PCB where metal is shown in red. The polygon pour is connected to VSS and cut further away from digital outputs and CK input. It also has relief connections for both through-hole and surface-mount components for easier soldering.
Figure 4.5: The bottom layer of the PCB where metal is shown in blue. The polygon pour is connected to VSS with vias that connect it to the top layer polygon pour.
4.3 Experimental Results

This section contains measurements taken using the test bench setup in Fig. 4.2. The measurement results are compared to the simulation results. The results from delay measurement methods 1 and 2, energy, and input-referred noise are given. In addition, the energy-noise product (ENP) and the energy-noise-delay product (ENDP) are plotted to help visualize optimal transistor sizing for the StrongARM comparator.

4.3.1 Delay

Delay Using Method 1

Using the delay measurement method 1 in section 3.1.1, different delays are measured as a function of the differential input voltage $V_{IN}$. Since the value of interest is the delay of just the comparator alone, the delay of the NAND3 is subtracted out from the measurement data, see Fig 4.6. The delay of the NAND3 is determined from layout-extracted simulations.

The differential input voltage $V_{IN}$ and the delay are inversely proportional. A larger $V_{IN}$ and a smaller $V_{IN}$ cause a smaller and larger delay, respectively. Schematic simulation with no offset means there is no inherent bias on the outcome. Therefore, when $V_{IN} = 0V$, the comparator experiences a metastable-like state except for the numerical noise acting in the circuit. This causes a longer comparison time and is seen in Fig. 4.6 as a large delay for $V_{IN} = 0V$. In measurement, the delay at $V_{IN} = 0V$ is not as large because in the chain of 1000 comparators (see section 3.1.1 and Fig. 3.4) only a small number of comparators experience $V_{IN}$ approaching 0 because of the intrinsic offset in the comparators. Thus, many comparators in the chain will not behave as having a zero differential input and will have a smaller delay making the measured delay smaller than that from simulations. It is seen later in Fig. 4.10 that the mean offset is not 0 V, so the delay curves in Fig. 4.6 have been shifted in the plot so that the peak delay is at $V_{IN} = 0$.

As $V_{IN}$ increases in absolute value, the comparator has less delay. The delay of one comparator is found by measuring the delay of 1000 comparators and dividing the total delay by 1000. Schematic simulations and chip measurements agree closely where the maximum error between the two is 2.2% for design 1, 2, and 4; 4% for design 3 for a small input voltage ($V_{IN} = 5mV$).
Figure 4.6: Delay plot of four StrongARM comparator designs.

Transistor sizing is another characteristic which contributes to the delay. Instead of sweeping $V_{IN}$, it is fixed at 5 mV and the widths of transistors $M_{1,2}$ and $M_{3,4}$ are swept from 1 to 10 microns in 1-micron increments and the widths of $M_{5,6}$ are fixed at 2 $\mu$m. The delay as a function of these transistor sizes are plotted in Fig. 4.7. The surface curve is the simulated delay and the red points are Designs 1-4 measured from the chip. A larger $M_{1,2}$ creates a bigger common-mode current, $I_1$ in Eq. 2.3, causing the amplification phase to be shorter. The same common-mode current in Eq. 2.4 affects the delay of the propagation phase as well. Since the delay of these two phases are influenced by the transistor size of $M_{1,2}$ the total delay always increases as $M_{1,2}$ decreases for a given $M_{34}$ (Fig. 4.7) [6].
Figure 4.7: The surface curve is the simulated delay where the widths of $M_{5,6} = 2 \mu m$. The annotated red points are measurements from the chip where $D1 = 183.4$ ps, $D2 = 187.9$ ps, $D3 = 202$ ps, $D4 = 198.7$ ps. X and Y axes are transistor widths in units of microns ($\mu m$).

Table 4.1: Delay values for the four designs in Fig. 4.7

<table>
<thead>
<tr>
<th></th>
<th>Design 1</th>
<th>Design 2</th>
<th>Design 3</th>
<th>Design 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulated Delay (ps)</td>
<td>180.4</td>
<td>168.9</td>
<td>220.4</td>
<td>167.2</td>
</tr>
<tr>
<td>Measured Delay (ps)</td>
<td>183.4</td>
<td>187.9</td>
<td>202</td>
<td>198.7</td>
</tr>
</tbody>
</table>
The amplification and propagation phase delays are determined by the common-mode current (Eq. 2.3 and 2.4). However, the regeneration phase delay is based upon the differential voltage at the output, \( V_{P,Q} \). In [6] \( V_{P,Q} \) at the start of regeneration is proportional to \( V_{IN} \), the differential input voltage. A larger \( V_{IN} \) creates a larger \( V_{P,Q} \) therefore shortening the regeneration delay (see Fig. 4.6).

**Delay Using Method 2**

Instead of measuring delay between falling edges of the feedthrough clock \( CK_{PASS} \) and \( V_{OUT} \), the frequency of \( V_{DFF} \) and the change in pulse width of \( V_{PFD} \) are used to calculate the delay. Delay from method 1 and 2 should match. Using the real waveforms in Fig. 4.8, \( T_{DFF} = T_{BEAT} = 12.9 \mu s \), and \( \Delta p = 100\, ns \). \( T_A = 1.0868\, \mu s \) and \( T_B = 1.1868\, \mu s \) from equations 3.4 and 3.2. In this case signal \( A = V_{OUT,1} \) and \( B = V_{OUT,3} \). The delay from method 1 is \( T_A = 1.0821\, \mu s \) and \( T_B = 1.1823\, \mu s \) which means the error between method 1 and 2 for signal \( A \) and \( B \) is 4.3% and 3.8%, respectively.

![Figure 4.8: Measured DFF and PFD transient waveforms for calculating delay.](image-url)
4.3.2 Energy

Energy per sample is the average power multiplied by the sample time, or the integral of \( P(t) \) from \( t = 0 \) to \( t = t_{CK} \), where \( P(t) \) is the instantaneous power at time \( t \) and \( t_{CK} \) is the clock period. As the comparator is clocked at a faster rate, the power increases, and the energy per sample remains constant. Thus, energy/sample is used to evaluate the comparator since it accounts for the cycle rate of the comparator. In simulations, energy per sample is calculated by multiplying the average power over 200 clock cycles by the clock period. This is done for many different designs, where \( V_{IN} = 5 \, mV \), to create the 3-dimensional surface in Fig. 4.9. The simulated and measured energy per cycle does not include clocking power or the power of the NAND3.

\[ E = \frac{1}{t_{CK}} \int_{0}^{t_{CK}} P(t) \, dt \]

Figure 4.9: Simulated energy with chip measurements annotated as the red points for each of the designs. Transistor widths are in microns (\( \mu m \)) for the X and Y axes. \( V_{IN} = 5 \, mV \).
Table 4.2: Energy values of the four designs in Fig. 4.9

<table>
<thead>
<tr>
<th></th>
<th>Design 1</th>
<th>Design 2</th>
<th>Design 3</th>
<th>Design 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulated Energy</td>
<td>333.2</td>
<td>296.8</td>
<td>406</td>
<td>212.3</td>
</tr>
<tr>
<td>per sample (fJ)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measured Energy</td>
<td>331.6</td>
<td>308</td>
<td>395</td>
<td>240.7</td>
</tr>
<tr>
<td>per sample (fJ)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Energy is linear with transistor size until the transistor width is less than or equal to 1 µm. This is consistent with Eq. 2.13 where the energy increases linearly with parasitic capacitance, or transistor size. Design 3 consumes the most energy per cycle because it has the largest transistors (see Table 3.1). In other words, when $C_{X,Y,P,Q}$ are bigger, they take more energy to recharge in the reset phase. Design 4 has the lowest energy consumption and has the smallest transistors. Energy and noise trade off with each other. Therefore, a complete and fair comparison of a comparator’s energy efficiency must be made in the context of noise. This will be presented shortly.

4.3.3 Offset

Offset is a permanent bias that exists in the comparator due to manufacturing variations. Since offset in a comparator affects the accuracy of an ADC, it is important that a design is chosen that has more resistance to offset. Although layout methods can improve offsets, like dummy transistors and symmetry, the transistor sizes also contribute to offset susceptibility. Take for example the layout shown for Design 3 and 4 in Fig. 3.11. Both designs have dummy transistors and are built with symmetry, but in Fig. 4.10 the standard deviation of Design 4 is 8.5770e-3 and Design 3 is 5.4224e-3 for offset. Designs with smaller transistors have more mismatch [16]. Table 4.3 shows the standard deviation and mean for the offset of all 4 prototype designs. Design 4 does have a higher standard deviation because of its smaller transistors.

Ideally, the simulated mean should be zero; however, due to the number of Monte Carlo samples there is some error. More Monte Carlo iterations would show the mean converging to 0V. The measured offset is further from zero than simulations. In addition to mismatches from fabrication, there are other factors causing more mismatch. This can include mismatch from the DACs.
(Fig. 4.2) and asymmetric parasitics from nearby metal routes. However, the DACs’ mismatch would only cause a shift in mean for the measured results (Table 4.3) and shifts all designs by the same amount. Due to these added sources of mismatch, the Monte Carlo simulations that only account for mismatch due to fabrication are lower and more optimistic.

![Figures 4.10](image)

**Figure 4.10:** Histograms for measured offset of designs 1-4. Standard deviations from right to left, top to bottom: $\sigma = 5.544\, mV$, $\sigma = 5.874\, mV$, $\sigma = 5.422\, mV$, $\sigma = 8.577\, mV$.

<table>
<thead>
<tr>
<th></th>
<th><strong>Design 1</strong></th>
<th><strong>Design 2</strong></th>
<th><strong>Design 3</strong></th>
<th><strong>Design 4</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Simulated</strong></td>
<td>$\sigma = 4.67, mV$</td>
<td>$\sigma = 4.64, mV$</td>
<td>$\sigma = 4.54, mV$</td>
<td>$\sigma = 5.79, mV$</td>
</tr>
<tr>
<td></td>
<td>$\mu = 746, \mu V$</td>
<td>$\mu = 736, \mu V$</td>
<td>$\mu = 639, \mu V$</td>
<td>$\mu = 307, \mu V$</td>
</tr>
<tr>
<td><strong>Measured</strong></td>
<td>$\sigma = 5.544, mV$</td>
<td>$\sigma = 5.874, mV$</td>
<td>$\sigma = 5.422, mV$</td>
<td>$\sigma = 8.577, mV$</td>
</tr>
<tr>
<td></td>
<td>$\mu = -3.88, mV$</td>
<td>$\mu = -3.37, mV$</td>
<td>$\mu = -2.90, mV$</td>
<td>$\mu = -3.47, mV$</td>
</tr>
</tbody>
</table>

Table 4.3: Standard deviation and mean of the offset. Mean of measured offset is larger than simulated mean due to additional mismatches in the DACs and nearby parasitics in the chip.
The transistors in the StrongARM comparator don’t all have the same offset contribution. $M_3$-$M_6$ are initially off, so their offset contribution is smaller. Mismatch between $M_3$ and $M_4$ will only affect the outcome minimally because the input has already been amplified and $V_{X-Y}$ is more likely to be greater than the offset voltage created by mismatch. $M_1$ and $M_2$ contribute the most amount of input referred offset because mismatch between these two transistors affect how the input voltage is being amplified initially [2].

Fig. 4.10 was produced by sweeping the voltage input using the 16-bit DACs (from Fig. 4.2) with a step size of 76.3 $\mu$V and a common-mode voltage of $\frac{V_{DD}}{2}$. Starting from a negative $V_{IN}$ and stepping to a positive $V_{IN}$, a subplot like that in Fig. 4.12 is produced. The offset can be visually inspected from these plots when the error function crosses 50%, or it can be solved for by using the error function in the expression $\frac{1}{2}[1 + erf(\frac{x-\mu}{\sqrt{2}\sigma})]$ where $\mu$ is the mean, or offset. For each of the 300 different input voltages (about a 40 mV sweep range from -20 mV to +20 mV), 5000 samples were taken to find the average percentage of 1’s to reject random noise. To create one of the subplots in Fig. 4.12, 3e6 samples were measured. However, the x-axis is zoomed in to only show -1.3mV to +1.3mV in Fig. 4.12. To measure 999 different comparators per design, data for 999 subplots (subplots like that in Fig. 4.12) was acquired for a total of 11.988e9 comparator samples. All of these samples were used to produce the offset histograms in Fig. 4.10 and also the noise histograms in Fig. 4.13.

### 4.3.4 Noise

Systematic offsets in a comparator can be digitally calibrated out, but issues caused by noise cannot. Therefore, noise in comparators can often be the bottleneck for high-performance ADCs. Systematic offsets in a comparator can be digitally calibrated out. For noise measurements the circuit is configured as shown in Fig. 3.10. Fig. 4.11 is the transient waveform captured on the oscilloscope where $CK_{IN}$ is connected to $OUT_0$ and $Q_{OUT}$ is connected to $OUT_1$ using the MUXs in Fig. 3.5 and appropriate switches in Fig. 3.10. The dashed red lines indicate where the output has settled and is sampled as either a ‘0’ or a ‘1’. $CK_{IN}$ is the sampling clock and $Q_{OUT}$ is the output of the comparator, Fig. 3.10. $Q_{OUT}$ is a return-to-one signal because of the reset phase. Although the FPGA does all the sampling, this transient waveform captured by the oscilloscope
is to illustrate how it is sampled. Note that noise causes the output to be a ’1’ even though $V_{IN}$ is negative.

![Figure 4.11: The transient waveform where $OUT_0 = \overline{CK}_{IN}$ and $OUT_1 = Q_{OUT}$ when the chip is configured like Fig. 3.10. $Q_{OUT}$ is the output of the comparator and it is a logic 1 when comparator senses a positive input and is a logic 0 otherwise. $V_{IN} = -0.5mV$ produces this transient waveform where logic 1’s still occur due to noise.](image)

The percentage of 1’s versus voltage input amplitude forms an error function because the noise is Gaussian. This is described in more detail in [2]. Below is a set of CDF curves for the four designs (Fig. 4.12). Simulated data in Fig. 4.12 consists of 2000 samples/point in 100 $\mu V$ increments from $-1.5 mV$ to $1.5 mV$ and the measured consists of 1500 samples/point in 76.3 $\mu V$ increments for the same range of $V_{IN}$, both at a common-mode of $V_{DD}/2$. However, the CDF for the measured data has some offset and is shifted in post processing to zero out the offset. These initial noise results show that the measured noise is similar to the simulated noise. Design 3 deviates the most from the simulated data which is most likely due to transient fluctuations in the power supply, $V_{DD}$, since it uses more power than the other designs. It also may be true that electromagnetic noise is increasing the noise measurements of design 3 by a higher percentage than the other designs because it’s starting point of intrinsic noise is lower.
Figure 4.12: Cumulative distribution function (CDF) plots of the four different prototype designs. The CDF is curve-fitted to the simulation data.

Below is the histogram for input-referred noise measured from each of the four designs (Fig. 4.13). The noise values were taken from the chip using offset mode and 999 comparators were measured. Although there are 1000 comparators of each design that were fabricated, the first comparator cannot be measured for noise and offset since it does not have a select signal allowing $Q$ to be routed to $Q_{OUT}$ (see Fig. 3.10). The input voltage step size and common mode is the same for measuring offset, 76.3 $\mu$V and 900 $mV$, respectively. Variations in $g_m$ and $\gamma$ will cause the same comparator design to have a spread of input-referred noise values. This is because of the power spectral density of a MOSFET modeled as a white noise current source that is equal to $4kT\gamma g_m$ which shows up in Eq. 2.11. Nearby traces of metal may also cause undesired differences.
in parasitic capacitance between different comparators. The bin width for each of the histograms is $6.67 \text{nV}^2$.

![Histograms of input-referred noise for designs 1-4.](image)

**Figure 4.13**: Histograms of input-referred noise for designs 1-4.

The surface plot in Fig. 4.14 is the input-referred noise of transient simulations on a supercomputer. In order to produce all 100 noise points, 5 jobs of 20 points are simulated in parallel, each point containing 2000 samples. Fig. 4.14 helps the designer understand the design space of a StrongARM comparator and its noise behavior. It would otherwise require prohibitively large and time-consuming simulations. The primary shape of the 3D surface plot is due to the size of $M_{1,2}$ which is consistent with [2] and agrees with the analysis in Eq. 2.10. To a first order, $M_{1,2}$ defines the noise of the comparator with second order effects from $M_{3,4}$. This is because $M_{1,2}$ contributes...
the most noise, as noted earlier, but noise contributed by $M_{3,4}$ is enough to make design 3 better than design 1 or 2 for noise performance.

Figure 4.14: Input-referred noise sweep of $M_{1,2}$ and $M_{3,4}$; $M_{5,6} = 2\mu m$. The red points are the measured input-referred noise of the four designs. The measurements follow the same trend as the surface plot. Input-referred noise values of the four designs in order are: $100\, nV^2$, $117\, nV^2$, $95.7\, nV^2$, $174\, nV^2$.

From Fig. 4.14, a larger $M_{1,2}$ reduces noise more than a larger $M_{3,4}$. Minimum noise performance within the swept transistor sizing range is achieved when both $M_{1,2}$ and $M_{3,4}$ are large. A major reason not to make $M_{3,4}$ larger is due to an increasing cost of power. Essentially, the most cost effective noise reduction comes from enlarging the width of $M_{1,2}$.
4.3.5 Energy-Noise and Energy-Noise-Delay Products

As explained previously, energy and noise trade off — the more energy a comparator consumes the less noise it has. Therefore, we use the energy-noise product (ENP) as a figure-of-merit (in Joules-Volt$^2$) to evaluate the energy efficiency of a comparator. As will be seen next, certain comparator designs are more “optimal” in the ENP sense. That is, for the same amount of energy that a comparator consumes, certain designs achieve lower noise than others. This is because the noise contributions of the transistors are different but all transistors roughly contribute equally to the energy consumption. To visualize where the optimum design region is located, we plot the ENP in Fig. 4.15 using data from Figs. 4.9 and 4.14.

Within the design space studied, the lowest ENP is achieved for $M_{1,2} = 10 \mu m$, $M_{3,4}$ between 2 and 4 $\mu m$, and $M_{5,6} = 2 \mu m$. This is expected because $M_{1,2}$ contributes noise more than the other transistors. Therefore, the strategy to optimize the energy efficiency (minimize ENP) is to use the largest transistors for $M_{1,2}$ followed by $M_{3,4}$, then $M_{5,6}$. Fig. 4.16 provides an alternative visualization of the ENP by plotting the ENP vs $M_{3,4}$ sizes for different $M_{1,2}$ sizes. It can be seen that the optimum ENP region changes slightly for different $M_{1,2}$ sizes. For the smallest $M_{1,2}$ (1 $\mu m$), the optimum $M_{3,4}$ is 2 $\mu m$. For the largest $M_{1,2}$ (10 $\mu m$), the optimum region is more flat and ranges between 2 to 4 $\mu m$. The ENP is smaller for larger $M_{1,2}$ because the load (NAND3) is kept constant. Therefore, when $M_{1,2}$ increases in size, the capacitive load is proportionally smaller. The energy consumed in the capacitive load also gets smaller. Since noise is dominated by $M_{1,2}$, when the energy consumed by the load is proportionally smaller, the ENP improves.
Figure 4.15: Transistor sizes ($M_{1,2}$ and $M_{3,4}$) versus energy-noise product (ENP in Joules-Volt$^2$) vs. $M_{1,2}$ and $M_{3,4}$; $M_{5,6} = 2\mu m$.

Table 4.5: Energy-Noise Product (ENP in Joules-Volt$^2$) values of the four designs in Fig. 4.15

<table>
<thead>
<tr>
<th></th>
<th>Design 1</th>
<th>Design 2</th>
<th>Design 3</th>
<th>Design 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulated ENP</td>
<td>30.1e-21</td>
<td>30.11e-21</td>
<td>33.07e-21</td>
<td>40.57e-21</td>
</tr>
<tr>
<td>Measured ENP</td>
<td>32.9e-21</td>
<td>36.09e-21</td>
<td>37.55e-21</td>
<td>47.8e-21</td>
</tr>
</tbody>
</table>
Finally, the energy-noise-delay product (ENDP) combines energy, noise, and delay as another figure-of-merit for the comparator [17]. The ENDP is shown in Fig. 4.17 and 4.18. For finding an optimal design in the ENDP sense, the trend is like the ENP where increasing $M_{1,2}$, to a certain extent creates a more efficient comparator. The ENDP decreases marginally when $M_{1,2} > 5 \mu m$ given that $M_{3,4} = 2 \mu m$. For $M_{1,2} > 5 \mu m$, the comparator consumes more energy for a marginally superior design in the ENDP sense. However, $M_{3,4}$ affects the ENDP more than the ENP for smaller $M_{1,2}$. Take the curve $M_{1,2} = 1 \mu m$ in Fig. 4.18, for example; $M_{3,4}$ drastically changes the ENDP (compare this to the same curve in Fig. 4.16).
Figure 4.17: Plot of energy-noise-delay product (ENDP in Joules-Volt^2-seconds) versus $M_{1,2}$ and $M_{3,4}$; $M_{5,6} = 2 \mu m$

Table 4.6: Energy-Noise-delay Product (ENDP in Joules-Volt^2-seconds) of the four designs in Fig. 4.17

<table>
<thead>
<tr>
<th>Design</th>
<th>Design 2</th>
<th>Design 3</th>
<th>Design 4</th>
</tr>
</thead>
</table>
Figure 4.18: Plot of energy-noise-delay product (ENDP in Joules-Volt$^2$-seconds) versus $M_{3,4}$; $M_{5,6} = 2 \mu m$
CHAPTER 5. CONCLUSION

The StrongARM comparator is used in many mixed-signal circuits including ADCs. Their performance has a large impact on the performance of the system. To avoid performance bottlenecks, the comparator must have the right transistor sizing. This thesis presents new techniques to measure the comparator performance with a large number of comparators with little area overhead. Simulations that leverage supercomputers explore a large design space of different transistor sizes. The results from the simulations and measurements are in good agreement and they show optimum design points in the ENP and ENDP sense. The energy-noise product shows that a larger $M_{1,2}$, a smaller $M_{3,4}$, and the smallest $M_{5,6}$ have the best energy-noise trade-off. The exact ratios between the transistors have been found in simulations and measurement.

A circuit architecture is used to allow all comparators to be used for each type of measurement (delay, power, offset, and noise) instead of comparators built exclusively for one type only. By implementing the comparators along with test measurement circuitry in an area-efficient manner, a total of 4000 comparators fit on one test chip. A new method for measuring the delay of a StrongARM comparator is used. By chaining together many comparators, the delay is time amplified to make delay measurements off-chip possible. In addition, a phase frequency detection circuit along with a D-Flip-Flop is proved as a second method for measuring delay. This second method allows for even the smallest differences in oscillation frequencies to be measured and used to calculate the delay of a circuit. The results show accurate delay measurements using both methods.

The analysis and simulations of energy, delay, and noise all reasonably agree with chip measurements where the same trend is observed when changing transistor widths. The ENP and ENDP provide the designer key insights into the design space of a comparator in terms of transistor sizing ratios.
5.1 Future Research

Further research could be done to measure the delay and noise at different common-mode input voltages to see how much they change. Analysis for other phases in the comparator cycle using a method similar to the method for finding noise in phase 1 would help simplify the design process. This would require finding the total integrated noise voltage on $C_{X,Y}$ and $C_{P,Q}$ where $g_{m1,2}$ is time-varying and no longer constant. The analysis can then be compared to simulations and measurement for validation. A unified analysis accounting for energy, noise, and delay can rigorously prescribe the optimum transistor sizing in the ENP and ENDP sense to guide the designer.
REFERENCES


APPENDIX A. TEST BENCH CODE

A.1 FPGA Code for Measuring Noise and Delay (Method 1)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity daq_ctrl is
  generic(
    CLK_RATE : natural := 50e6;
    -- BAUD_RATE: natural := 19200;
    BAUD_RATE : natural := 115200;
    SR_SIZE : natural := 1000;  -- number of bits in shift register
    CKS_RATE : natural := 50e3;
    SAMP_CNT_MAX : natural := 1e3
  );
  port(
    clk : in std_logic;
    rst : in std_logic;
    rx_in : in std_logic;
    --opcode : in STD_LOGIC_VECTOR (47 downto 0);
    sr_d : out std_logic;
    cks : out std_logic;
    tx_out : out std_logic;
    led : out std_logic_vector (7 downto 0);
    seg : out std_logic_vector (6 downto 0);
    dp : out std_logic;
    an : out std_logic_vector (3 downto 0);
    sclk_1 : out std_logic;  -- for 16-bit dac output
    din_1 : out std_logic;  -- for 16-bit dac output
    cs_1 : out std_logic;  -- for 16-bit dac output
    ldac_1 : out std_logic;  -- for 16-bit dac output
    sclk_2 : out std_logic;  -- for 16-bit dac output
    din_2 : out std_logic;  -- for 16-bit dac output
    cs_2 : out std_logic;  -- for 16-bit dac output
  );
end entity daq_ctrl;
```
dac_2  : out std_logic;    -- for 16-bit dac output
sampl_clk : in std_logic;
out0      : in std_logic;
btn_3     : in std_logic;

-- SRAM ports --
MemAdr : out std_logic_vector (22 downto 0);
MemOE  : out std_logic;    -- OE#
MemWR  : out std_logic;    -- WE#
RamCS  : out std_logic;    -- CE#
RamLB  : out std_logic;    -- LB
RamUB  : out std_logic;    -- UB
RamCLK : out std_logic;
RamADV : out std_logic;    -- ADV
RamCRE : out std_logic;
MemDB  : inout std_logic_vector (15 downto 0);

-- audio DAC ports --
sdata  : out std_logic;
mclk   : out std_logic;
lrclock: out std_logic;
sclock : out std_logic
end daq_ctrl;

architecture Behavioral of daq_ctrl is

component rx
generic(
  CLK_RATE : natural := 50e6;
  BAUD_RATE : natural := 19200);
port ( clk      : in std_logic;
       rst      : in std_logic;
       rx_in    : in std_logic;
       data_strobe : out std_logic;
       rx_busy   : out std_logic;
       data_out  : out std_logic_vector (7 downto 0)
     );
end component;

component tx
generic(
  CLK_RATE : natural := 50e6;
  BAUD_RATE : natural := 19200);
port ( clk      : in std_logic;
       rst      : in std_logic;
       data_in  : in std_logic_vector (7 downto 0);
       send_character : in std_logic;
       tx_out    : out std_logic;
       tx_busy   : out std_logic
     );
end component;

component seven_segment_control
port (clk : in std_logic;
  data_in : in std_logic_vector (15 downto 0);
  dp_in : in std_logic_vector (3 downto 0);
  blank : in std_logic_vector (3 downto 0);
  seg : out std_logic_vector (6 downto 0);
  dp : out std_logic;
  an : out std_logic_vector (3 downto 0)
); end component;

component srgen
  generic (CLK_RATE : natural := 50e6;
           CKS_RATE : natural := 5e6;
           -- lowest possible is 1 kHz
           SR_SIZE : natural := 1000
           -- number of bits in shift register
          );
  port (clk, rst, start : in std_logic;
        mode_in : in std_logic_vector (3 downto 0);
        ctrl_in : in std_logic_vector (17 downto 0);
        comp_sel : in std_logic_vector (9 downto 0);
        cks_out, d, busy : out std_logic);
end component;

component pmodDA3
  generic (SCLK_MAX : natural := 5
           -- 10 MHz
          );
  port (clk : in std_logic;
        rst : in std_logic;
        data : in std_logic_vector (15 downto 0);
        start : in std_logic;
        -- load din to this block and output new analog voltage
        busy : out std_logic;
        clk_out : out std_logic;
        din : out std_logic;
        -- serial input data tx to DAC
        cs : out std_logic;
        ldac : out std_logic
      ); end component;

type mc_state_type is
  (IDLE, FTCH_WAIT, FTCH, DEC, OPCODE_ERROR, TXMSG, TXMSG_WAIT, TEST0,
   SRTST, SR_START, SR_WAIT, LP000, LP001, LP002, LP003, LP004, CHMODE,
   CH_SET_VIN, WAIT_STB, OF_MODE, SET_VIN, DAC_SET, DAC_SETTLE,
   OFFSET_SAMP, OF_MODE_WAIT_USR, OF_MODE_WAIT_USR_TWO,
   WAIT_BETWEEN_SEL_CMDS, WAIT_BETWEEN_SEL_CMDS_2, SRAM_SAVE, SRAM_SAVE_WAIT,
   OF_SEND_RESULTS,
   SRAM_READ_SETUP, SRAM_READ, SRAM_READ_WAIT, DAC_VOLT_SET_MSB,
   DAC_VOLT_SET_LSB, WAIT_BETWEEN_DAC_VOLT, WAIT_BETWEEN_DAC_VOLT_2);
  signal state_reg, state_next : mc_state_type;
type acq_state_type is (SAMP_EVENT_WAIT, WAIT_TO_ACQ, ACQ_VALUE, WAIT_UNTIL_EVENT);
signal acq_state_reg, acq_state_next : acq_state_type;

-- rx signals
signal data_strobe : std_logic;  -- output
signal rx_busy : std_logic;  -- output
signal rx_data_out : std_logic_vector (7 downto 0);  -- output
signal rx_d : std_logic;
signal rx_d_next : std_logic;
signal rx_dd : std_logic;
signal rx_dd_next : std_logic;

-- tx signals
signal tx_busy : std_logic;
signal tx_data_in, tx_data_in_next : std_logic_vector (7 downto 0);
signal tx_send_character, tx_send_character_next : std_logic;

-- daq_ctrl signals
constant OPCODE_COUNTER_MAX : natural := 6;
constant CMD_PRNT_DLY : natural := 2603;
constant TX_DLY : natural := 4500;  -- 2603*50;  -- 2603 clk cycles per bit in a 8-byte symbol
signal tx_cnt, tx_cnt_next : unsigned (15 downto 0);  -- (32 downto 0)
signal opc_cnt, opc_cnt_next : unsigned (2 downto 0);
signal ftch_cnt, ftch_cnt_next : unsigned (2 downto 0);

signal ftch_cnt_en, clr_ftch_cnt : std_logic;
signal tx_cnt_en, clr_tx_cnt, opc_cnt_en, clr_opc_cnt : std_logic;

signal opcode, opcode_next : std_logic_vector (47 downto 0) := (others => '0');  -- 5 char, 2 4-bit each
signal tx_message, tx_message_next : std_logic_vector (47 downto 0) := (others => '0');

-- seven segment signals
signal disp_data : std_logic_vector (15 downto 0);

-- sr signals
signal cks_t, sr_d_t : std_logic := '0';
signal sel, sel_next : std_logic_vector (9 downto 0) := (others => '0');
signal mode, mode_next : std_logic_vector (3 downto 0);
signal ctrl, ctrl_next : std_logic_vector (17 downto 0);
signal sr_begin, sr_busy : std_logic := '0';

-- DAC signals
signal dac_data_1 : unsigned (15 downto 0) := (others => '0');
signal dac_data_1_next : unsigned (15 downto 0) := (others => '0');
--- OFFSET Signals

signal samp_cnt, samp_cnt_next : unsigned (24 downto 0) := (others => '0');
signal clr_samp_cnt : std_logic;
signal samp_cnt_en : std_logic;
signal vin_cnt_en : std_logic;
signal dac_cnt_en : std_logic;
signal clr_dac_cnt : std_logic;
signal clr_vin_cnt : std_logic;
constant VIN_CNT_MAX : integer := 1500;
— normally 400 with 23017=VIP and 23417=VIN
constant DAC_SETTLE_MAX : integer := 10000;
— dac has 1 us settling time
signal dac_cnt, dac_cnt_next : unsigned (16 downto 0) := (others => '0');
signal vin_cnt, vin_cnt_next : unsigned (12 downto 0) := (others => '0');
signal sel_data_rx : std_logic_vector (15 downto 0);
signal sel_data_rx_next : std_logic_vector (15 downto 0);
signal dac_temp : std_logic_vector (15 downto 0);
signal des_num : std_logic_vector (3 downto 0) := (others => '0');
signal des_num_next : std_logic_vector (3 downto 0) := (others => '0');

--- CHAIN Signals

constant VIN_CNT_MAX_CH_SWP : integer := 4718;
signal rx_en : std_logic := '1'; — allows the opcode register to shift

--- ACQuisition Signals
signal per_cnt_next : unsigned (47 downto 0) := (others=>'0');
signal per_cnt_next : unsigned (15 downto 0) := (others=>'0');
signal per_cnt_next : unsignd (47 downto 0) := (others=>0');
signal per_cnt : unsigned (15 downto 0) := (others=>'0');
signal per_cnt : std_logic;
signal clr_per_cnt : std_logic;
signal samp_stb : std_logic;
signal clr_acq_cnt : std_logic;
signal acq_cnt_en : std_logic;
signal acq_cnt, acq_cnt_next : unsigned (5 downto 0);
constant ACQ_CNT_MAX : integer := 5;
signal nosig_cnt : unsigned (15 downto 0) := (others=>'0');
signal nosig_cnt_next : unsigned (15 downto 0) := (others=>'0');
signal nosig_cnt : std_logic;
signal clr_nosig_cnt : std_logic;
constant NOSIG_MAX_VALUE : integer := 10000;

— SRAM memory Signals

signal sram_start : std_logic := '1'; — LOW assert
signal sram_rw, sram_rw_next : std_logic := '1'; — 1 for read 0
for write
signal sram_addr_next, sram_addr : unsigned (22 downto 0);
signal write_data, write_data_next : std_logic_vector (15 downto 0);
signal read_data : std_logic_vector (15 downto 0);
signal sram_data_valid : std_logic;
signal sram_ready : std_logic;

rx_u1 : rx
    generic map (BAUD_RATE => BAUD_RATE)
    port map (clk => clk, rst => rst, rx_in => rx_dd,
              data_strobe => data_strobe, rx_busy => rx_busy,
              data_out => rx_data_out);

tax_u2 : tx
    generic map (BAUD_RATE => BAUD_RATE)
    port map(clk => clk, rst => rst, data_in => tx_data_in,
             send_character => tx_send_character,
             tx_out => tx_out, tx_busy => tx_busy);

seven_seg : seven_segment_control
    port map(clk => clk, blank => "0000", data_in => disp_data,
             dp_in => "0000", seg => seg, dp => dp, an => an);

sr_gen_u1 : sr_gen
    generic map(
                  CLK_RATE => CLK_RATE,
                  CKS_RATE => CKS_RATE, — lowest possible is 1 kHz
                  SR_SIZE => SR_SIZE — number of bits in shift register
                )
    port map(clk => clk, rst => rst, start => sr_begin,
mode_in => mode,
ctrl_in => ctrl,
comp_sel => sel,
cks_out => cks_t, d => sr_d_t, busy => sr_busy);

pmodDA3_u1 : pmodDA3
generic map(
  SCLK_MAX => 5                           -- sclk freq=10 MHz
)
port map(clk => clk,
  rst => rst,
  data => std_logic_vector(dac_data_1),
  start => dac_start_1, -- load data to this block and output
  new analog voltage
  busy => dac_busy_1,
  sclk_out => sclk_1,
  din => din_1, -- serial input data tx to DAC
  cs => cs_1, -- active low signal
  ldac => ldac_1
);

pmodDA3_u2 : pmodDA3
generic map(
  SCLK_MAX => 5                           -- sclk freq=10 MHz
)
port map(clk => clk,
  rst => rst,
  data => std_logic_vector(dac_data_2),
  start => dac_start_2, -- load data to this block and output
  new analog voltage
  busy => dac_busy_2,
  sclk_out => sclk_2,
  din => din_2, -- serial input data tx to DAC
  cs => cs_2, -- active low signal
  ldac => ldac_2
);

sram_ctl_u1 : entity work.sramController
generic map(CLK_RATE => CLK_RATE)
port map(clk => clk,
  rst => rst,
  mem => sram_start,
  rw => sram_rw, -- 1 for read 0 for write
  addr => std_logic_vector(sram_addr),
  data_m2s => write_data,
  data_s2m => read_data,
  data_valid => sram_data_valid,
  ready => sram_ready,
  MemAdr => MemAdr,
  MemOE => MemOE,
  MemWR => MemWR,
  RamCS => RamCS,
  RamLB => RamLB,
  RamUB => RamUB,
RamCLK => RamCLK,
RamADV => RamADV,
RamCRE => RamCRE,
MemDB => MemDB);

cks <= cks_t;
sr_d <= sr_d_t;

-- state register
process(clk, rst)
begin
  if (rst = '1') then
    state_reg <= IDLE;
    acq_state_reg <= SAMP_EVENT_WAIT;
  elsif (clk 'event and clk = '1') then
    state_reg <= state_next;
    acq_state_reg <= acq_state_next;
  end if;
end process;

-- registers
process(clk, rst)
begin
  if (rst = '1') then
    tx_message <= (others => '0');
    rx_d <= '1';
    rx_dd <= '1';
    opcode <= (others => '0');
    mode <= (others => '0');
    ctrl <= (others => '0');
    sel <= (others => '0');
    dac_data_1 <= (others => '0');
    dac_data_2 <= (others => '0');
    sel_data_rx <= (others => '0');
    des_num <= (others => '0');
  elsif (clk 'event and clk = '1') then
    des_num <= des_num_next;
    tx_message <= tx_message_next;
    rx_d <= rx_d_next;
    rx_dd <= rx_dd_next;
    tx_cnt <= tx_cnt_next;
  end if;
end process;

-- count registers
  tx_cnt <= tx_cnt_next;
  opc_cnt <= opc_cnt_next;

-- tx registers
  tx_send_character <= tx_send_character_next;
  tx_data_in <= tx_data_in_next;

-- rx registers
  ftch_cnt <= ftch_cnt_next;
  opcode <= opcode_next;
  sel_data_rx <= sel_data_rx_next;
--- sr gen registers
ctrl <= ctrl_next;
mode <= mode_next;
seg <= sel_next;

--- dac registers
dac_data_1 <= dac_data_1_next;
dac_data_2 <= dac_data_2_next;
dacData1_cnt <= dacData1_cnt_next;
dacData2_cnt <= dacData2_cnt_next;
inptoggle <= input_toggle_next;
dac_cnt <= dac_cnt_next;
samp_cnt <= samp_cnt_next;
vinstagram <= vin_cnt_next;

--- acquisition registers
acq_cnt <= acq_cnt_next;
per_cnt <= per_cnt_next;
nosig_cnt <= nosig_cnt_next;

--- sram registers
sram_rw <= sram_rw_next;
sram_addr <= sram_addr_next;
write_data <= write_data_next;

--- audio dac registers
data_vip <= data_vip_next;
data_vin <= data_vin_next;

end if;
end process;
rx_d_next <= rx_in;
rxd_next <= rx_d;

--- COUNTERS

tx_cnt_next <= (others => '0') when (tx_cnt = TX_DLY+1 or clr_tx_cnt = '1') else
  tx_cnt + 1 when tx_cnt_en = '1' else
  tx_cnt;

opc_cnt_next <= (others => '0') when (opc_cnt = OPCODE_COUNTER_MAX+1 or
  clr_opc_cnt = '1') else
  opc_cnt + 1 when opc_cnt_en = '1' else
  opc_cnt;

ftch_cnt_next <= (others => '0') when (ftch_cnt = OPCODE_COUNTER_MAX or
  clr_ftch_cnt = '1') else
  ftch_cnt + 1 when ftch_cnt_en = '1' else
  ftch_cnt;
opcode_next <= opcode (39 downto 0) & rx_data_out when (data_strobe = '1' and rx_en = '1') else
          opcode;

vin_cnt_next <= (others => '0') when (clr_vin_cnt = '1') else
          vin_cnt + 1 when vin_cnt_en = '1' else
          vin_cnt;

samp_cnt_next <= (others => '0') when (samp_cnt = SAMP_CNT_MAX or clr_samp_cnt = '1') else
          samp_cnt + 1 when samp_cnt_en = '1' else
          samp_cnt;

dac_cnt_next <= (others => '0') when (clr_dac_cnt = '1') else
          dac_cnt + 1 when dac_cnt_en = '1' else
          dac_cnt;

acq_cnt_next <= (others => '0') when (acq_cnt = ACQ_CNT_MAX or clr_acq_cnt = '1') else
          acq_cnt + 1 when acq_cnt_en = '1' else
          acq_cnt;

per_cnt_next <= (others => '0') when (clr_per_cnt = '1') else
          per_cnt + 1 when per_cnt_en = '1' else
          per_cnt;

nosig_cnt_next <= (others => '0') when (clr_nosig_cnt = '1') else
          nosig_cnt + 1 when nosig_cnt_en = '1' else
          nosig_cnt;

dacData1_cnt_next <= (others => '0') when (clr_dacData1_cnt = '1') else
          dacData1_cnt + 256 when dacData1_cnt_en = '1' else
          dacData1_cnt;

dacData2_cnt_next <= (others => '0') when (clr_dacData2_cnt = '1') else
          dacData2_cnt - 256 when dacData2_cnt_en = '1' else
          dacData2_cnt;

-- next-state logic for acquisition (sampling during offset mode)
process (acq_state_reg, samp_clk, acq_cnt, out0, state_reg)
begin
    state_next <= acq_state_reg;
    clr_acq_cnt <= '0';
    acq_cnt_en <= '0';
    samp_stb <= '0';
    per_cnt_en <= '0';

    case acq_state_reg is
    when SAMP_EVENT_WAIT =>
        clr_acq_cnt <= '1';
        if samp_clk = '0' then
            acq_state_next <= WAIT_TO_ACQ;
        else
            acq_state_next <= acq_state_reg;
    end case;
end process;
end if;

when WAIT_TO_ACQ =>
    acq_cnt_en <= '1';
    if (acq_cnt < ACQ_CNT_MAX) then
        acq_state_next <= WAIT_TO_ACQ;
    else
        acq_state_next <= ACQ_VALUE;
    end if;

when ACQ_VALUE =>
    samp_stb <= '1';
    if out0 = '1' and state_reg = OFFSET_SAMP then
        per_cnt_en <= '1';
    end if;
    acq_state_next <= WAIT_UNTIL_EVENT;

when WAIT_UNTIL_EVENT => -- don't let it sample two times in the same valley of samp_clk
    if samp_clk = '1' then
        acq_state_next <= SAMP_EVENT_WAIT;
    else
        acq_state_next <= WAIT_UNTIL_EVENT;
    end if;
end case;
end process;

-- next-state logic
process(state_reg, tx_cnt, opc_cnt, ftch_cnt, vin_cnt, samp_cnt, dac_cnt, samp_stb, 
    sr_busy, tx_message, data_strobe, rx_data_out, rx_busy, 
    sel_data_rx, opcode, sel, mode, ctrl, input_toggle, dac_busy_1, 
    dac_busy_2, dac_data_1, dac_data_2, per_cnt, write_data, sram_rw, 
    sram_addr, btn_3, 
    sram_ready, sram_data_valid, read_data, data_vio, data_vip)
begin
    state_next <= state_reg;
    sel_data_rx_next <= sel_data_rx;
    clr_tx_cnt <= '0';
    tx_cnt_en <= '0';

    clr_opc_cnt <= '0';
    opc_cnt_en <= '0';

    clr_ftch_cnt <= '0';
    ftch_cnt_en <= '0';

    clr_vin_cnt <= '0';
    vin_cnt_en <= '0';

    clr_dac_cnt <= '0';
    dac_cnt_en <= '0';
clr_samp_cnt <= '0';
samp_cnt_en <= '0';

dac_start_1 <= '0';
dac_start_2 <= '0';

clr_per_cnt <= '0';

clr_nosig_cnt <= '0';
nosig_cnt_en <= '0';

dacDatal_cnt_en <= '0';
clr_dacDatal_cnt <= '0';

dacData2_cnt_en <= '0';
clr_dacData2_cnt <= '0';

dac_data_1_next <= dac_data_1;
dac_data_2_next <= dac_data_2;

input_toggle_next <= input_toggle;

tx_send_character_next <= '0';
tx_message_next <= tx_message;

rx_en <= '1';

sr_begin <= '0';

sram_start <= '1';
write_data_next <= write_data;
sram_rw_next <= sram_rw;
sram_addr_next <= sram_addr;

sel_next <= sel;
mode_next <= mode;
ctrl_next <= ctrl;

data_vip_next <= data_vip;
data_vin_next <= data_vin;

des_num_next <= "0000";

case state_reg is
  when IDLE =>
    sram_addr_next <= (others => '0');
    input_toggle_next <= '0';
    clr_tx_cnt <= '1';
    clr_opc_cnt <= '1';
    clr_ftch_cnt <= '1';
    clr_vin_cnt <= '1';
    clr_nosig_cnt <= '1';
    mode_next <= "0000";
    ctrl_next <= (others => '0');
state_next <= FTCH_WAIT;  -- SRTST; this is for shift register testing
data_vip_next <= "100111000011000111100000";  -- to_signed (-6540832,24);
data_vin_next <= "100111001100111000100000";  -- to_signed (-6500832,24);

when FTCH_WAIT =>
  ctrl_next <= (others => '1');  -- this is here for no reason except to avoid signals being optimized away.
  if (data_strobe = '1') then
    ftch_cnt_en <= '1';
    state_next <= FTCH;
  end if;

when FTCH =>
  if (ftch_cnt < OPCODE_COUNTER_MAX) then
    state_next <= FTCH_WAIT;
  else
    state_next <= DEC;
    clr_ftch_cnt <= '1';
  end if;

when DEC =>
  if (opcode = x"54455354300A") then
    state_next <= TEST0;
  elsif (opcode = x"53525453540A") then
    state_next <= SRTST;
  elsif (opcode = x"4C503030300A") then
    state_next <= LP000;  -- des 1 and des 2
  elsif (opcode = x"4C503030310A") then
    state_next <= LP001;  -- des 2 and des 1
  elsif (opcode = x"4C503030320A") then
    state_next <= LP002;  -- des 3 and des 4
  elsif (opcode = x"4C503030330A") then
    state_next <= LP003;  -- des 4 and des 4
  elsif (opcode = x"4C503030340A") then
    state_next <= LP004;  -- des 1 and des 4
  elsif (opcode(47 downto 32) = x"4348") then
    state_next <= CH_MODE;
  elsif (opcode(47 downto 32) = x"4F46") then
    state_next <= WAIT_BETWEEN_SEL_CMDS;  -- OF_MODE;  --
  elsif (opcode(47 downto 32) = x"5245") then
    state_next <= OF_SEND_RESULTS;  -- "READM" read results from SRAM
  elsif (opcode(47 downto 32) = x"4441") then  -- DAC voltage set
    state_next <= WAIT_BETWEEN_DAC_VOLT;
  else
    state_next <= OPCODE_ERROR;
  end if;

when OPCODE_ERROR =>
  state_next <= TXMSG_WAIT;
tx_message_next <= x"4F5045520A"; — "OP ER" : send to PC opcode error message

when TEST0 => — verify that PC to FPGA link is working
  state_next <= TXMSG_WAIT;
  tx_message_next <= x"434F4D4F0A"; — "COMOK" : send to PC that message was received

— insert more opcode states below

when SRTST =>
  mode_next <= "1111";
  ctrl_next <= "0000010111111110"; — this pattern works on chip
  — ctrl_next <= "000001001101100110"; — this pattern works on chip
  — ctrl_next <= "110001001101100110"; — this pattern works on chip
  sel_next <= "0000010000";
  state_next <= SR_START;

when LP000 => — des 1 and des 2
  mode_next <= "0001";
  ctrl_next <= (others => '0');
  ctrl_next(10) <= '1'; — sets des1 CK_LAST to X1
  ctrl_next(13) <= '1'; — sets des2 CK_LAST to X2
  sel_next <= "0000000000";
  state_next <= SR_START;

when LP001 => — des 2 and des 1
  mode_next <= "0001";
  ctrl_next <= (others => '0');
  ctrl_next(11) <= '1'; — sets des1 CK_LAST to X2
  ctrl_next(12) <= '1'; — sets des2 CK_LAST to X1
  sel_next <= "0000000000";
  state_next <= SR_START;

when LP002 => — des 3 and des 4
  mode_next <= "0001";
  ctrl_next <= (others => '0');
  ctrl_next(14) <= '1'; — sets des3 CK_LAST to X2
  ctrl_next(17) <= '1'; — sets des4 CK_LAST to X1
  sel_next <= "0000000000";
  state_next <= SR_START;

when LP003 => — des 4 and des 3
  mode_next <= "0001";
  ctrl_next <= (others => '0');
  ctrl_next(15) <= '1'; — sets des3 CK_LAST to X2
  ctrl_next(16) <= '1'; — sets des4 CK_LAST to X1
  sel_next <= "0000000000";
  state_next <= SR_START;

when LP004 => — des 1 and des 4
  mode_next <= "0001";
ctrl_next <= (others => '0');
ctrl_next(11) <= '1'; -- sets des1 CK_LAST to X2
ctrl_next(16) <= '1'; -- sets des4 CK_LAST to X1
sel_next <= "0000000000";
state_next <= SR_START;

--- CHAIN MODE

when CH_MODE =>
  mode_next <= "0100";
  --mode_next <= "0100";
  state_next <= SR_START;
  sel_next <= "0000000000";
  ctrl_next <= (others => '0');
dac_data_1_next <= to_unsigned(0, 16); --VIP_initial
dac_data_2_next <= to_unsigned(47180, 16); --VIN_initial = 1.8 V

--- Hook up OUT1 and OUT0 according to opcode instruction
--- OUT0
case opcode (23 downto 16) is
  when x"31" => -- des1
    ctrl_next(0) <= '1';
  when x"32" => -- des2
    ctrl_next(2) <= '1';
  when x"33" => -- des3
    ctrl_next(4) <= '1';
  when x"34" => -- des4
    ctrl_next(6) <= '1';
  when x"43" => --clock
    ctrl_next(8) <= '1';
  when others =>
    state_next <= TXMSG_WAIT;
    tx_message_next <= x"4241445F300A";
    ctrl_next <= (others => '0');
end case;

--- OUT1
case opcode (15 downto 8 ) is
  when x"31" => -- des1
    ctrl_next(1) <= '1';
  when x"32" => -- des2
    ctrl_next(3) <= '1';
  when x"33" => -- des3
    ctrl_next(5) <= '1';
  when x"34" => -- des4
    ctrl_next(7) <= '1';
  when x"43" => --clock
    ctrl_next(9) <= '1';
  when others =>
    state_next <= TXMSG_WAIT;
    tx_message_next <= x"4241445F310A";
    ctrl_next <= (others => '0');
end case;
when CH_SET_VIN =>
  vin_cnt_en <= '1';
  if (vin_cnt < VIN_CNT_MAX_CH_SWP - 1) then
    state_next <= DAC_SET;
    dac_data_1_next <= dac_data_1 + 10; -- increase VIP voltage
    dac_data_2_next <= dac_data_2 - 10; -- decrease VIN voltage
  else
    state_next <= IDLE;
    clr_vin_cnt <= '1';
  end if;

when WAIT_STB =>
  rx_en <= '0';
  if (data_strobe = '1') then
    state_next <= CH_SET_VIN;
  end if;

— END OF CHAIN MODE

— OFFSET MODE

when WAIT_BETWEEN_SEL_CMDS =>
  if (rx_busy = '0') then
    state_next <= OF_MODE_WAIT_USR;
  end if;

when OF_MODE_WAIT_USR =>
  rx_en <= '0';
  if (data_strobe = '1') then
    sel_data_rx_next <= rx_data_out & sel_data_rx (7 downto 0);
    state_next <= WAIT_BETWEEN_SEL_CMDS_2;
  end if;

when WAIT_BETWEEN_SEL_CMDS_2 =>
  if (rx_busy = '0') then
    state_next <= OF_MODE_WAIT_USR_TWO;
  end if;

when OF_MODE_WAIT_USR_TWO =>
  rx_en <= '0';
  if (data_strobe = '1') then
    sel_data_rx_next <= sel_data_rx (15 downto 8) & rx_data_out;
    state_next <= OF_MODE;
  end if;

when OF_MODE =>
  rx_en <= '0';
  sram_addr_next <= (others => '0');
  mode_next <= "1010";
  sel_next <= sel_data_rx (9 downto 0); --"00" & rx_data_out;--
  std_logic_vector(to_unsigned(500,10));
dac.data_1.next <= to_unsigned(22934, 16);  -- VIP_initial
dac.data_2.next <= to_unsigned(24434, 16);  -- VIN_initial

input_toggle_next <= '0';
state_next <= SR_START;
if (opcode (31 downto 24) = x"45") then  -- FPGA uses external clock for sampling
    for sampling
        if (opcode (23 downto 8) = x"3143") then  -- des1 "1C"
            ctrl.next <= (0 => '1', 9 => '1', others => '0');
            des_num.next <= "0011";
        elsif (opcode (23 downto 8) = x"3243") then  -- des2
            ctrl.next <= (2 => '1', 9 => '1', others => '0');
            des_num.next <= "0010";
        elsif (opcode (23 downto 8) = x"3343") then  -- des3
            ctrl.next <= (4 => '1', 9 => '1', others => '0');
            des_num.next <= "0001";
        elsif (opcode (23 downto 8) = x"3443") then  -- des4
            ctrl.next <= (6 => '1', 9 => '1', others => '0');
            des_num.next <= "0000";
        else
            state.next <= OPCODE_ERROR;
        end if;
    end for sampling;
end if;
when SET.VIN =>
    clr_dac_cnt <= '1';
    vin_cnt_en <= '1';
    if (vin_cnt < VIN_CNT_MAX - 1) then
        state.next <= DAC_SET;
    end if;

capture on oscpoe
    dac.data_1.next <= dac.data_1 + 1;  -- increase VIP voltage by one LSB
    dac.data_2.next <= dac.data_2 - 1;  -- decrease VIN voltage by on LSB

    data_vip.next <= data_vip + 100;
    data_vin.next <= data_vin - 100;
else
    state.next <= IDLE;
    clr_vin_cnt <= '1';
end if;
when DAC_SET =>
    clr_dac_cnt <= '1';
    clr_samp_cnt <= '1';
    clr_per_cnt <= '1';
    dac_start_1 <= '1';
when DAC_SETTLE =>  
  if dac_busy_1 = '0' then
    dac_cnt_en <= '1';
    if dac_cnt < DAC_SETTLE_MAX then
      state_next <= DAC_SETTLE;
    else
      case opcode (47 downto 32) is
        when x"4F46" =>  -- "OF" offset mode
          state_next <= OFFSET_SAMP;
        when x"4348" =>  -- "CH" chain mode
          tx_message_next <= x"4348444E450A";  -- "CHDNE"
          state_next <= TXMSG_WAIT;
        when others =>
          state_next <= IDLE;
      end case;
    end if;
  else
    state_next <= DAC_SETTLE;
  end if;

when OFFSET_SAMP =>
  if samp_stb = '1' then
    samp_cnt_en <= '1';
  end if;
  if samp_cnt < SAMP_CNT_MAX then
    state_next <= OFFSET_SAMP;
  else
    state_next <= SRAM_SAVE;
    write_data_next <= std_logic_vector(per_cnt (15 downto 0));
    sram_addr_next <= sram_addr + 1;
    sram_rw_next <= '0';
  end if;

when SRAM_SAVE =>
  if (sram_ready = '1') then
    -- low asserted
    sram_start <= '0';
    state_next <= SRAM_SAVE_WAIT;
  else
    state_next <= SRAM_SAVE;
  end if;

when SRAM_SAVE_WAIT =>
  if (sram_ready = '1') then
    state_next <= SET_VIN;
  end if;
when OF_SEND_RESULTS => — read data from memory, send to PC over UART
  vin_cnt_en <= '1';
  if (vin_cnt < VIN_CNT_MAX - 1) then
    state_next <= SRAM_READ_SETUP;
  else
    state_next <= IDLE;
    clr_vin_cnt <= '1';
  end if;

when SRAM_READ_SETUP =>
  sram_addr_next <= sram_addr + 1;
  sram_rw_next <= '1';
  state_next <= SRAM_READ;

when SRAM_READ =>
  if (sram_ready = '1') then — low asserted
    sram_start <= '0';
    state_next <= SRAM_READ_WAIT;
  else
    state_next <= SRAM_READ;
  end if;

when SRAM_READ_WAIT =>
  if (sram_data_valid = '1') then
    tx_message_next <= x"00000000" & read_data;
    state_next <= TXMSG_WAIT;
  end if;

— END OF OFFSET MODE

— SHIFT REGISTER

when SR_START =>
  sr_begin <= '1';
  if (sr_busy = '1') then — wait for sr_tx.vhd to receive start
    state_next <= SR_WAIT;
  end if;

when SR_WAIT =>
  if (sr_busy = '0') then
    tx_message_next <= x"5352444E450A"; — "SRDNE"
    if (opcode (47 downto 32) = x"4F46") then — "OF" offset mode
      state_next <= SET_VIN;
    else
      state_next <= TXMSG_WAIT;
    end if;
  end if;
end if;
when WAIT_BETWEEN_DAC_VOLT =>
  rx_en <= '0';
  if (rx_busy = '0') then
    state_next <= DAC_VOLT_SET_MSB;
  end if;

when DAC_VOLT_SET_MSB =>
  rx_en <= '0';
  if (data_strobe = '1') then
    dac_data_1_next <= unsigned(rx_data_out & std_logic_vector(
      dac_data_1(7 downto 0)));
    dac_data_2_next <= unsigned(rx_data_out & std_logic_vector(
      dac_data_2(7 downto 0)));
    state_next <= WAIT_BETWEEN_DAC_VOLT_2;
  end if;

when WAIT_BETWEEN_DAC_VOLT_2 =>
  rx_en <= '0';
  if (rx_busy = '0') then
    state_next <= DAC_VOLT_SET_LSB;
  end if;

when DAC_VOLT_SET_LSB =>
  rx_en <= '0';
  if (data_strobe = '1') then
    dac_data_1_next <= unsigned(std_logic_vector(dac_data_1(15 downto 8)) & rx_data_out);
    dac_data_2_next <= unsigned(std_logic_vector(dac_data_2(15 downto 8)) & rx_data_out);
    state_next <= DAC_SET;
  end if;

—— insert more opcode states above

—— TRANSMIT DATA OVER UART TO MATLAB

when TXMSG_WAIT =>
  tx_cnt_en <= '1';
  if (tx_cnt < TX_DLY+1) then
    state_next <= TXMSG_WAIT;
  else
    state_next <= TXMSG;
    opc_cnt_en <= '1';
    clr_tx_cnt <= '1';
  end if;

when TXMSG =>
  if (opc_cnt < OPCODE_COUNTER_MAX+1) then
    tx_message_next <= tx_message (39 downto 0) & x"2A";
    tx_send_character_next <= '1';
A.2 FPGA Code for Measuring Delay using PFD and DFF Outputs (Method 2)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

library UNISIM;
use UNISIM.VComponents.all;

entity daq_ctrl is
  generic(
    CLK_RATE : natural := 50e6;
    -- BAUD_RATE : natural := 19200;
    BAUD_RATE : natural := 115200;
  )
  port (pclk : in STD_LOGIC;
        rx : in STD_LOGIC;
        ccontrol : in STD_LOGIC;
        cdac : in STD_LOGIC;
        creset : in STD_LOGIC;
        cstart : in STD_LOGIC;
        cstop : in STD_LOGIC;
        clk_stat : in STD_LOGIC;
        done : out STD_LOGIC;
        err : out STD_LOGIC;
        sr : out STD_LOGIC;
        dac_val : out integer;
        config : out STD_LOGIC;
        delay : out integer;
        dac_reg : out integer);
end entity daq_ctrl;
```
SR_SIZE : natural := 1000;  -- number of bits in shift register
CKS_RATE : natural := 50e3
);
port(
  clk : in std_logic;
  rst : in std_logic;
  rx_in : in std_logic;
  --opcode : in STD_LOGIC_VECTOR (47 downto 0);
  sr_d : out std_logic;
  cks : out std_logic;
  tx_out : out std_logic;
  led : out std_logic_vector (7 downto 0);
  seg : out std_logic_vector (6 downto 0);
  dp : out std_logic;
  an : out std_logic_vector (3 downto 0);
  sclk_1 : out std_logic;  -- for 16-bit dac output
  din_1 : out std_logic;  -- for 16-bit dac output
  cs_1 : out std_logic;  -- for 16-bit dac output
  ldac_1 : out std_logic;  -- for 16-bit dac output
  sclk_2 : out std_logic;  -- for 16-bit dac output
  din_2 : out std_logic;  -- for 16-bit dac output
  cs_2 : out std_logic;  -- for 16-bit dac output
  ldac_2 : out std_logic;  -- for 16-bit dac output
  samp_clk : in std_logic;
  out0 : in std_logic;
  btn_3 : in std_logic
);
end daq_ctrl;

architecture Behavioral of daq_ctrl is

component rx
generic(
  CLK_RATE : natural := 50e6;
  BAUD_RATE : natural := 19200);
port (clk : in std_logic;
  rst : in std_logic;
  rx_in : in std_logic;
  data_strobe : out std_logic;
  rx_busy : out std_logic;
  data_out : out std_logic_vector (7 downto 0)
);
end component;

component tx
generic(
  CLK_RATE : natural := 50e6;
  BAUD_RATE : natural := 19200);
port (clk : in std_logic;
  rst : in std_logic;
  data_in : in std_logic_vector (7 downto 0);
  send_character : in std_logic;
  tx_out : out std_logic;

81
component seven_segment_control
  port (clk : in std_logic;
        data_in : in std_logic_vector (15 downto 0);
        dp_in : in std_logic_vector (3 downto 0);
        blank : in std_logic_vector (3 downto 0);
        seg : out std_logic_vector (6 downto 0);
        dp : out std_logic;
        an : out std_logic_vector (3 downto 0)) ;
end component;

component pmodDA3
generic (
  SCLK_MAX : natural := 5    — 10 MHz
);
  port (clk : in std_logic;
        rst : in std_logic;
        data : in std_logic_vector (15 downto 0);
        start : in std_logic; — load din to this block and output
new analog voltage
        busy : out std_logic;
        sclk_out : out std_logic;
        din : out std_logic; — serial input data tx to DAC
        cs : out std_logic; — active low signal
        ldac : out std_logic
);
end component;

type mc_state_type is
  (IDLE, FTCH_WAIT, FTCH, DEC, TXMSG, TXMSG_WAIT, SR_START, SR_WAIT,
   OF_MODE, DAC_SET, DAC_SETTLE, OF_MODE_WAIT_USR, OF_MODE_WAIT_USR_TWO,
   WAIT_BETWEEN_SEL_CMDS, WAIT_BETWEEN_SEL_CMDS_2, V_INPUT_STATE,
   V_INPUT_WAIT);
signal state_reg, state_next : mc_state_type;

— rx signals
signal data_strobe : std_logic; — output
signal rx_busy : std_logic; — output
signal rx_data_out : std_logic_vector (7 downto 0); — output
signal rx_d : std_logic;
signal rx_d_next : std_logic;
signal rx_dd : std_logic;
signal rx_dd_next : std_logic;

— tx signals
signal tx_busy : std_logic;
signal tx_data_in, tx_data_in_next : std_logic_vector (7 downto 0);
signal tx_send_character, tx_send_character_next : std_logic;

-- daq_ctrl_signals
constant OPCODE_COUNTER_MAX : natural := 6;
constant TX_DLY : natural := 4500; -- 2603*50; -- 2603 clk cycles per bit in a 8-byte symbol

signal tx_cnt, tx_cnt_next : unsigned (15 downto 0); -- (32 downto 0)
signal opc_cnt, opc_cnt_next : unsigned (2 downto 0);
signal ftch_cnt, ftch_cnt_next : unsigned (2 downto 0);
signal v_input_cnt, v_input_cnt_next : unsigned (2 downto 0);

signal ftch_cnt_en, clr_ftch_cnt : std_logic;
signal tx_cnt_en, clr_tx_cnt, opc_cnt_en, clr_opc_cnt : std_logic;

signal opcode, opcode_next : std_logic_vector (47 downto 0) := (others => '0'); -- 5 char, 2 4-bit each
signal tx_message, tx_message_next : std_logic_vector (47 downto 0) := (others => '0');

-- seven segment signals
signal disp_data : std_logic_vector (15 downto 0);

-- sr signals
signal cks_t, sr_d_t : std_logic := '0';
signal sel, sel_next : std_logic_vector (9 downto 0) := (others => '0');

signal mode, mode_next : std_logic_vector (3 downto 0);
signal ctrl, ctrl_next : std_logic_vector (17 downto 0);
signal sr_begin, sr_busy : std_logic := '0';

-- DAC signals
signal dac_data_1 : unsigned (15 downto 0) := (others => '0');
signal dac_data_1_next : unsigned (15 downto 0) := (others => '0');
signal dac_data_2 : unsigned (15 downto 0) := (others => '0');
signal dac_data_2_next : unsigned (15 downto 0) := (others => '0');

signal dac_start_1, dac_start_2 : std_logic := '0';
signal dac_busy_1, dac_busy_2 : std_logic;

-- OFFSET Signals
signal dac_cnt_en : std_logic;
signal clr_dac_cnt : std_logic;
constant DAC_SETTLE_MAX : integer := 10000;  -- dac has 1 us settling time

signal dac_cnt, dac_cnt_next : unsigned (16 downto 0) := (others => '0');
signal sel_data_rx : std_logic_vector (15 downto 0);
signal sel_data_rx_next : std_logic_vector (15 downto 0);
signal dac_temp : std_logic_vector (15 downto 0);

signal rx_en : std_logic := '1';  -- allows the opcode register to shift
constant VINPUT_COUNTER_MAX : natural := 4;
signal vinput, vinput_next : std_logic_vector (31 downto 0) := (others => '0');
signal clr_vinput_cnt : std_logic := '0';
signal vinput_cnt_en : std_logic := '0';
signal vinput_en : std_logic := '0';

begin

rx_u1 : rx
  generic map (BAUD_RATE => BAUD_RATE)
  port map (clk => clk, rst => rst, rx_in => rx_dd,
            data_strobe => data_strobe, rx_busy => rx_busy,
            data_out => rx_data_out);

tx_u2 : tx
  generic map (BAUD_RATE => BAUD_RATE)
  port map (clk => clk, rst => rst, data_in => tx_data_in,
            send_character => tx_send_character,
            tx_out => tx_out, tx_busy => tx_busy);

seven_seg : seven_segment_control
  port map (clk => clk, blank => "0000", data_in => disp_data,
            dp_in => "0000", seg => seg, dp => dp, an => an);

sr_gen_u1 : entity work.sr_gen
  generic map (CLK_RATE => CLK_RATE,
               CKS_RATE => CKS_RATE,       -- lowest possible is 1 kHz
               SR_SIZE => SR_SIZE          -- number of bits in shift register
               )
  port map (clk => clk, rst => rst, start => sr_begin,
            mode_in => mode,
            ctrl_in => ctrl,
            comp_sel => sel,
            cks_out => cks_t, d => sr_d_t, busy => sr_busy);

pmodDA3_u1 : pmodDA3
  generic map (SCLK_MAX => 5)  -- sclk freq=10 MHz
  port map (clk => clk,
```vhdl
--
-- new analog voltage
-- busy => dac_busy_1,
sclk_out => sclk_1,
din => din_1,  -- serial input data tx to DAC
cs => cs_1,   -- active low signal
ldac => ldac_1
);

pmodDA3_u2 : pmodDA3
  generic map(
    SCLK_MAX => 5  -- sclk freq=10 MHz
  )
  port map(clk => clk,
           rst => rst,
           data => std_logic_vector(dac_data1_cnt),
           start => dac_start_1,  -- load data to this block and output
           new analog voltage
           busy => dac_busy_1,
sclk_out => sclk_1,
din => din_1,  -- serial input data tx to DAC
cs => cs_1,   -- active low signal
ldac => ldac_1
);

cks <= cks_t;
sr_d <= sr_d_t;
-- state register
process(clk, rst)
begin
  if(rst = '1') then
    state_reg <= IDLE;
    elsif(clk'event and clk = '1') then
      state_reg <= state_next;
  end if;
end process;

-- registers
process(clk, rst)
begin
  if(rst = '1') then
    tx_message <= (others => '0');
    rx_d <= '1';
    rx_dd <= '1';
    opcode <= (others => '0');
    mode <= (others => '0');
    ctrl <= (others => '0');
    sel <= (others => '0');
    dac_data_1 <= (others => '0');
  end if;
end process;
```
dac_data_2 <= (others => '0');
sel_data_rx <= (others => '0');
elsif (clk'event and clk = '1') then
  tx_message <= tx_message_next;
  rx_d <= rx_d_next;
  rx_dd <= rx_dd_next;
  tx_cnt <= tx_cnt_next;

  -- count registers
  tx_cnt <= tx_cnt_next;
  opc_cnt <= opc_cnt_next;
  v_input_cnt <= v_input_cnt_next;

  -- tx registers
  tx_send_character <= tx_send_character_next;
  tx_data_in <= tx_data_in_next;

  -- rx registers
  ftch_cnt <= ftch_cnt_next;
  opcode <= opcode_next;
  sel_data_rx <= sel_data_rx_next;
  v_input <= v_input_next;

  -- sr gen registers
  ctrl <= ctrl_next;
  mode <= mode_next;
  sel <= sel_next;

  -- dac registers
  dac_data_1 <= dac_data_1_next;
  dac_data_2 <= dac_data_2_next;
  dac_cnt <= dac_cnt_next;
end if;
end process;

rx_d_next <= rx_in;
rx_dd_next <= rx_d;

-- COUNTERS

tx_cnt_next <= (others => '0') when (tx_cnt = TX_DLY+1 or clr_tx_cnt = '1') else
  tx_cnt + 1 when tx_cnt_en = '1' else
  tx_cnt;

opc_cnt_next <= (others => '0') when (opc_cnt = OPCODE COUNTER_MAX+1 or
clr_opc_cnt = '1') else
  opc_cnt + 1 when opc_cnt_en = '1' else
  opc_cnt;
ftch_cnt_next <= (others => '0') when (ftch_cnt = OPCODE_COUNTER_MAX or clr_ftch_cnt = '1') else
    ftch_cnt + 1 when ftch_cnt_en = '1' else
    ftch_cnt;

opcode_next <= opcode (39 downto 0) & rx_data_out when (data_strobe = '1' and rx_en = '1') else
    opcode;

v_input_next <= v_input (23 downto 0) & rx_data_out when (data_strobe = '1' and v_input_en = '1') else
    v_input;

dac_cnt_next <= (others => '0') when (clr_dac_cnt = '1') else
    dac_cnt + 1 when dac_cnt_en = '1' else
    dac_cnt;

v_input_cnt_next <= (others => '0') when (clr_v_input_cnt = '1') else
    v_input_cnt + 1 when v_input_cnt_en = '1' else
    v_input_cnt;

--- next-state logic
process(state_reg, tx_cnt, opc_cnt, ftch_cnt, dac_cnt,
        sr_busy, tx_message, data_strobe, rx_data_out, rx_busy,
        sel_data_rx, opcode, sel, mode, ctrl, dac_busy_1,
        dac_busy_2, dac_data_1, dac_data_2, btn_3)
begin
    state_next <= state_reg;
    sel_data_rx_next <= sel_data_rx;
    clr_tx_cnt <= '0';
    tx_cnt_en <= '0';

    clr_opc_cnt <= '0';
    opc_cnt_en <= '0';

    clr_ftch_cnt <= '0';
    ftch_cnt_en <= '0';

    clr_dac_cnt <= '0';
    dac_cnt_en <= '0';

    dac_start_1 <= '0';
    dac_start_2 <= '0';

    clr_v_input_cnt <= '0';
    v_input_cnt_en <= '0';
dac_data_1_next <= dac_data_1;
dac_data_2_next <= dac_data_2;

tx_send_character_next <= '0';
rx_en <= '1';
v_input_en <= '0';

sr_begin <= '0';

sel_next <= sel;
mode_next <= mode;
ctrl_next <= ctrl;

case state_reg is
when IDLE =>
  clr_dac_cnt <= '1';
  clr_tx_cnt <= '1';
  clr_opc_cnt <= '1';
  clr_ftch_cnt <= '1';
  clr_v_input_cnt <= '1';
  mode_next <= "0000";
  ctrl_next <= (others => '0');
  state_next <= FTCH_WAIT; ——SRTST; this is for shift register testing
when FTCH_WAIT =>
  ctrl_next <= (others => '1'); ——this is here for no reason except to avoid signals being optimized away.
  if (data_strobe = '1') then
    ftch_cnt_en <= '1';
    state_next <= FTCH;
  end if;
when FTCH =>
  if (ftch_cnt < OPCODE_COUNTER_MAX) then
    state_next <= FTCH_WAIT;
  else
    state_next <= DEC;
    clr_ftch_cnt <= '1';
  end if;
when DEC =>
  if (opcode(47 downto 24) = "4F4645") then
    state_next <= WAIT_BETWEEN_SEL_CMDS; —— program chip only
  elsif (opcode (47 downto 24) = "444143") then
    state_next <= V_INPUT_WAIT; —— program DACs only
  else
state_next <= IDLE;
end if;

---------- OFFSET MODE
----------

when WAIT_BETWEEN_SEL_CMDS =>
  if (rx_busy = '0') then
    state_next <= OF_MODE_WAIT_USR;
  end if;

when OF_MODE_WAIT_USR =>
  rx_en <= '0';
  if (data_strobe = '1') then
    sel_data_rx_next <= rx_data_out & sel_data_rx (7 downto 0);
    state_next <= WAIT_BETWEEN_SEL_CMDS_2;
  end if;

when WAIT_BETWEEN_SEL_CMDS_2 =>
  if (rx_busy = '0') then
    state_next <= OF_MODE_WAIT_USR_TWO;
  end if;

when OF_MODE_WAIT_USR_TWO =>
  rx_en <= '0';
  ctrl_next <= (others => '0');
  if (data_strobe = '1') then
    sel_data_rx_next <= sel_data_rx (15 downto 8) & rx_data_out;
    state_next <= OF_MODE;
  end if;

when OF_MODE =>
  rx_en <= '0';
  mode_next <= "0001"; -- combined loop–chain mode
  mode_next <= "0101"; -- combined loop–chain mode
  mode_next <= "1001"; -- combined loop–qout mode
  mode_next <= "1010"; -- combined qout–reset mode
  sel_next <= sel_data_rx (9 downto 0);

---------- OUT0 or XI

case opcode (23 downto 16) is
  when x"31" =>
    ctrl_next(0) <= '1';
    ctrl_next(10) <= '1';
  when x"32" =>
    ctrl_next(2) <= '1';
    ctrl_next(12) <= '1';
  when x"33" =>
    ctrl_next(4) <= '1';
    ctrl_next(14) <= '1';
  when x"34" =>
    -- des4
\begin{verbatim}
ctrl_next(6) <= '1';
ctrl_next(16) <= '1';
when x"43" =>
  ctrl_next(8) <= '1';
when others =>
  ctrl_next(0) <= '1';
end case;

-- OUT1 or X2
case opcode (15 downto 8) is
when x"31" =>
  ctrl_next(1) <= '1';
ctrl_next(11) <= '1';
when x"32" =>
  ctrl_next(3) <= '1';
ctrl_next(13) <= '1';
when x"33" =>
  ctrl_next(5) <= '1';
ctrl_next(15) <= '1';
when x"34" =>
  ctrl_next(7) <= '1';
ctrl_next(17) <= '1';
when x"43" =>
  --clock
ctrl_next(9) <= '1';
when others =>
  ctrl_next(1) <= '1';
end case;

state_next <= SR_START;

when V_INPUT_WAIT =>
  rx_en    <= '0';
  v_input_en <= '1';
  if (data_strobe = '1') then
    v_input_cnt_en <= '1';
    state_next    <= V_INPUT_STATE;
  end if;

when V_INPUT_STATE =>
  rx_en    <= '0';
  v_input_en <= '1';
  if (v_input_cnt < V_INPUT_COUNTER_MAX) then
    state_next    <= V_INPUT_WAIT;
  else
    state_next    <= DAC_SET;
    clr_v_input_cnt <= '1';
    --dac_data_1_next <= to_unsigned(22934,16);-- VIP
    --dac_data_2_next <= to_unsigned(24434,16);--VIN
    dac_data_1_next <= unsigned(v_input (15 downto 0)); -- VIP
    dac_data_2_next <= unsigned(v_input (31 downto 16)); -- VIN
  end if;

when DAC_SET =>
\end{verbatim}
when DAC_SETTLE =>
if dac_busy_1 = '0' then
dac_cnt_en <= '1';
if dac_cnt < DAC_SETTLE_MAX then
  state_next <= DAC_SETTLE;
else
  state_next <= IDLE;
end if;
else
  state_next <= DAC_SETTLE;
end if;

--- SHIFT REGISTER

when SR_START =>
sr_begin <= '1';
if (sr_busy = '1') then -- wait for sr_tx.vhd to recieve start
  state_next <= SR_WAIT;
end if;

when SR_WAIT =>
tx_message_next <= x"5352444E450A"; -- "SRDNE"
if (sr_busy = '0') then
  state_next <= TXMSG_WAIT;
end if;

--- TRANSMIT DATA OVER UART TO MATLAB

when TXMSG_WAIT =>
tx_cnt_en <= '1';
if (tx_cnt < TX_DLY+1) then
  state_next <= TXMSG_WAIT;
else
  state_next <= TXMSG;
  opc_cnt_en <= '1';
  clr_tx_cnt <= '1';
end if;

when TXMSG =>
if (opc_cnt < OPCODE_COUNTER_MAX+1) then
  tx_message_next <= tx_message (39 downto 0) & x"2A";
\begin{verbatim}
tx_send_character_next <= '1';
state.next <= TXMSG_WAIT;
else
  clr_opc_cnt <= '1';
  state.next <= IDLE;
end if;

end case;
end process;

tx_data_in.next <= tx_message (47 downto 40);

led (7 downto 0) <= "10010000" when state_reg = FTCH_WAIT else
  "00001111" when state_reg = TXMSG_WAIT else
  "00000010" when state_reg = DAC_SET else
  "00000100" when state_reg = DAC_SETTLE else
  "11111111";

disp_data <= sel_data_rx;  --opcode (31 downto 16);

end Behavioral;
\end{verbatim}

A.3 Super Computer Scripts

A.3.1 C SHell script

```bash
#!/bin/csh

setenv LM_LICENSE_FILE 5280@ece-cadence.byu.edu
setenv CDS_Netlisting_Mode Analog
setenv CDS_LOG_PATH .
setenv CDS_LOAD_ENV addCWD
setenv CLS_CDSD_COMPATIBILITY_LOCKING NO
setenv DD_DONT_DO_OS_LOCKS set
setenv CDS_AUTO_64BIT_ALL

# All the documentation we care about
setenv CDSDOC_PROJECT /ee2/Cadence/cdssetup
# for the stuff we always want to run
setenv CDS_SITE /ee2/Cadence/cdssetup

set BASE_CADENCE="/cadence/installs
# IC tools
set IC=$BASE_CADENCE/IC617/tools

set MMSIM=$BASE_CADENCE/MMSIM151/tools.lnx86

# Cadence path
```
set CPATH=$IC/bin
set CPATH=${CPATH}:${IC}/dfl/bin:${IC}/plot/bin:${IC}/iccraft/bin
set CPATH=${CPATH}:${MMSIM}/bin

# c18 variables
setenv ONC18 ~/c18
setenv ONC18_PDK_PATH $ONC18/onc18_1_15p2ext2
setenv ON_DK_CDS yes
setenv onc18VerilogAIncludePath $ONC18_PDK_PATH/models/verilogA
setenv onc18_calibre_path $ONC18_PDK_PATH/verification/calibre

# Calibre setup
setenv CALIBRE_HOME $BASE/CADENCE/aoi_cal_2013.4.15.12
setenv MGLS_LICENSE_FILE 1717@eemodsimlic1.fsl.byu.edu
unsetenv XMODIFIERS # to allow keyboard input in PEX Interactive

setenv OA_UNSUPPORTED_PLAT linux_rhel50_gcc44x

# Set CDS_HOME
setenv CDS_HOME "~/cadence/install/IC617"

# Call ocean command to source OCEAN script to run simulation
ocean -nograph -restore sweep/no_noise_load/by_parts/m34_1_to_2.ocn

A.3.2 OCEAN script

:============================ Set to XL mode
ocnSetXLMode()
ocnxlProjectDir("~/simulation")
ocnxlTargetCellView("comparator_testing" "chain1_sweep" "m34_1_to_2")
ocnxlResultsLocation("")
ocnxlSimResultsLocation("/panfs/pan.fsl.byu.edu/scr/usr/48/nrw01")

:============================ Tests setup

:-------- Test "comparator_testing:chain1_sweep:1"
ocnxBeginTest("comparator_testing:chain1_sweep:1")
simulatord("spectre")
designd("comparator_testing" "chain1_sweep" "schematic")
modelFile(
  "("SONC18_PDK_PATH/models/spectre/base.scs" "")
  "("SONC18_PDK_PATH/models/spectre/mos_and_para1.scs" "typ")
  "("SONC18_PDK_PATH/models/spectre/mos_and_para2.scs" "typ")
  "("SONC18_PDK_PATH/models/spectre/res.scs" "typ")
  "("SONC18_PDK_PATH/models/spectre/cap.scs" "typ")
  "("SONC18_PDK_PATH/models/spectre/vertical.scs" "typ")
  "("SONC18_PDK_PATH/models/spectre/zeno.scs" "typ")
  "("SCDS_WORKAREA/userSetup$USER/onc18/device_models.scs" "advanced_SOA")
)
'*CD$WORKAREA/userSetup$USER/onc18/custom_model_libraries.spectre*"
")

)``

desVar( "m34" 4 )
desVar( "m12" 10 )
desVar( "m7" 2 )
desVar( "VID" −200u )
desVar( "TR" 200p )

evoption(
  'analysisOrderList("tran" "pz" "dcmatch" "stb" "envlp" "ac" "de" "lf" 
  "noise" "xf" 
  "sp" "pss" "pac" "psth" "pnoise" "pxf" "psp" "qps" "qpac" 
  "qpnoise" "qpxf" "qpss"
  "hb" "hbac" "hbstb" "hbnoise" "hbx" "sens"
  "acmatch")
)

option( 'categ 'turboOpts 
  'apsplus t 
  'uniMode "APS"
)

option( 'reltol "1e-3" )
saveOption( 'save "selected"
.save( 'v "/Q"
.temp( 27 )
ocnx1OutputSignal( "/Q" ?save t )
ocnx1EndTest() ; "comparator_testing: chain1_sweep:1"

:====================== Sweeps setup

ocnx1SweepVar("VID" 
  "\{Exclusion List\}−200u\{Exclusion List\}\{From/To\}Linear 
  :−1.3m:100u:1.3m\{From/To\}
)
ocnx1SweepVar("m34" 
  "\{Exclusion List\}4\{Exclusion List\}\{From/To\}Linear:1:1:2\{ 
  From/To\}
)
ocnx1SweepVar("m12" 
  "\{Exclusion List\}10\{Exclusion List\}\{From/To\}Linear 
  :1:1:10\{From/To\}
)

:====================== Model Group setup

cnx1ModelGroup( "slow_process"
  '(' 
    "SONC18_PDK_PATH/models/spectre/base.scs" "section "
    "SONC18_PDK_PATH/models/spectre/vertical.scs" "section \"\"wcl\"\"
    "SONC18_PDK_PATH/models/spectre/cap.scs" "section \"\"hi\"
    "SONC18_PDK_PATH/models/spectre/mos_and_par2.scs" "section \"\"slow 
    \"\"
    "SONC18_PDK_PATH/models/spectre/res.scs" "section \"\"hi\"
    "SONC18_PDK_PATH/models/spectre/mos_and_par1.scs" "section \"\"slow 
    \"\"
    "SONC18_PDK_PATH/models/spectre/zener.scs" "section \"\"lbv\"
    )
)
ocnxlModelGroup( "typical_process"
    ( "ONC18_PDK_PATH/models/spectre/base.scs" ?section ""
    ( "ONC18_PDK_PATH/models/spectre/vertical.scs" ?section ""typ\"
    ( "ONC18_PDK_PATH/models/spectre/cap.scs" ?section ""typ\"
    ( "ONC18_PDK_PATH/models/spectre/mos_and_para2.scs" ?section ""typ\"
    ( "ONC18_PDK_PATH/models/spectre/res.scs" ?section ""typ\"
    ( "ONC18_PDK_PATH/models/spectre/mos_and_par1.scs" ?section ""typ\"
    ( "ONC18_PDK_PATH/models/spectre/zener.scs" ?section ""typ\"
    ( "SCDS_WORKAREA/userSetup$USER/onc18/device_models.scs" ?section ""
advanced_SOA"
    ( "SCDS_WORKAREA/userSetup$USER/onc18/custom_model_libraries.spectre"
?section ""))
)
)
)
ocnxlModelGroup( "fast_process"
    ( "ONC18_PDK_PATH/models/spectre/base.scs" ?section ""
    ( "ONC18_PDK_PATH/models/spectre/vertical.scs" ?section ""wch\"
    ( "ONC18_PDK_PATH/models/spectre/cap.scs" ?section ""lo\"
    ( "ONC18_PDK_PATH/models/spectre/mos_and_para2.scs" ?section ""fast\"
    ( "ONC18_PDK_PATH/models/spectre/mos_and_par1.scs" ?section ""fast\"
    ( "ONC18_PDK_PATH/models/spectre/zener.scs" ?section ""hbv\"
    ( "SCDS_WORKAREA/userSetup$USER/onc18/device_models.scs" ?section ""
advanced_SOA"
    ( "SCDS_WORKAREA/userSetup$USER/onc18/custom_model_libraries.spectre"
?section ""))
)
)
)

; ====================== Corners setup
; ====================== Checks and Asserts setup
ocnxlPutChecksAsserts(?netlist nil)

; ====================== Job setup
ocnxlJobSetup( '(
    "blockemail" "1"
    "configuretimeout" "300"
    "distributionmethod" "Local"
    "lingertimeout" "300"
    "maxjobs" "24"
    "name" "ADE XL Default"
)
"preemptivestart" "1"
"reconfigureimmediately" "1"
"runtimeout" "-1"
"showerrorwhenretrying" "1"
"showoutputlogerror" "0"
"startmaxjobsimmed" "1"
"starttimeout" "300"
"usesameprocess" "1"
)

:==========================  Disabled items
:==========================  Run Mode Options
:==========================  Starting Point Info
:==========================  Run command
ocnxlRun( ?mode 'sweepsAndCorners ?nominalCornerEnabled t ?allCornersEnabled t ?allSweepsEnabled t )
ocnxlOutputSummary(?exprSummary t ?specSummary t ?detailed t ?wave t )

ocnxlOpenResults(?testName "comparator_testing:chain1_sweep:1")
selectResults('tran)
ocPrint( ?output "./sweep/no_noise_load/results/m34_1_to_2_m7_2.csv" ?width 10 ?numSpaces 1 ?numberNotation 'scientific sample(VT("/Q") 1.21e−08 2.5 e−06 "linear" 10e−09))

:==========================  End XL Mode command
ocnxlEndXLMode()
exit()