This is to certify that the
thesis entitled

DESIGN AND EVALUATION OF AN AUTOMATED TEST PLATFORM FOR LARGE-SCALE ANALOG FLOATING GATE ARRAY PROGRAMMING

presented by

Paul R. Kucher IV

has been accepted towards fulfillment
of the requirements for the

M.S. degree in Electrical and Computer Engineering

[illegible]
Major Professor's Signature

10/19/2007
Date

DESIGN AND EVALUATION OF AN AUTOMATED TEST PLATFORM FOR LARGE-SCALE ANALOG FLOATING GATE ARRAY PROGRAMMING

By

Paul R. Kucher IV

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Electrical and Computer Engineering

2007

ABSTRACT

DESIGN AND EVALUATION OF AN AUTOMATED TEST PLATFORM FOR LARGE-SCALE ANALOG FLOATING GATE ARRAY PROGRAMMING

By

Paul R. Kucher IV

Due to advances in microfabrication technology, modern digital systems can process large data sets using discrete algorithms with high precision. However, due to the increasing clock frequencies required to operate on data in real-time applications, analog circuit topologies have become attractive for computation. Such computational blocks require an analog data store that can achieve at least eight bits of accuracy for coarse classification. This work creates an automatic means of programming subthreshold floating gate circuits used as analog storage elements. The system consists of a test platform designed with a flexible configuration for both topology and process-neutral large-scale floating gate array programming. A system-on-chip with analog floating gates has been fabricated in a standard 0.5μm CMOS process and is used to validate the performance of the test platform. A novel algorithm for floating gate programming has been developed based on experimental observation and the test unit is capable of programming analog floating gate arrays to within 0.5% accuracy.

ACKNOWLEDGMENTS

I would like to begin by thanking my advisor, Dr. Shantanu Chakrabartty. I feel honored to have worked under his direction for the past two years, and have gained significant insight into designing mixed-signal VLSI systems for machine learning applications. He has provided me with the opportunity to pursue the areas of research I find most intellectually stimulating and has always guided me in accomplishing my goals. I look forward to continued collaboration and wish him the best in his own research endeavors.

I thank my committee members, Dr. Greg Wierzba and Dr. Andrew Mason, for taking time out of their busy schedules to be a part of my project. I also credit Dr. Mason with introducing me to research through working in his lab during my undergraduate career. I thank him for this opportunity and his thoughtful review of my thesis work. I thank Dr. Greg Wierzba for his advice and inspiration. I have enjoyed our many discussions and the advice he has given extending back to my undergraduate years. His courses have taught me many of the skills I needed to complete this work and I thank him for giving me these tools.

I thank my friend, Arthur Matteson, who helped in populating the test platform printed circuit board. I am impressed with his ability to work with fine-pitch, surface mount components and am grateful of being able to learn his technique. I also thank Arthur for his constant input and review of my thesis.

Finally, I thank my parents, Paul and Elaine Kucher, who have done their absolute best in raising and guiding me. They have always supported my interests and have never failed to help in time of need.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Analog VLSI computational methods become an attractive alternative to comparable digital VLSI techniques when systems demand high computational density and do not require high precision [1]. The former is the case in any application whose data depends on multi-dimensional datasets, and generally takes the form of a matrix-vector multiplication (dot product) [2]. These two requirements are the case in many machine learning applications where the output is represented as the confidence of a decision-making algorithm. In the case of digital systems, these algorithms are generally sequential in their execution, and require computationally-intensive operations such as the above mentioned matrix-vector multiplication as well as implementing polynomial functions [3].

Creating ultra-low-power signal processors often requires a substantial portion of the architecture implemented in the analog domain [4]. Additionally, when the input has a resolution of around 10 bits or less, analog computation can have significant advantages when compared to an architecture implemented in the digital domain. For example, there are many application circuits that are both very elegant in their implementation and scalability as well as their power consumption. An example of such a circuit is the winner-take-all or maximum circuit [5]. What appears to be an O(n) problem in a computer science context becomes a parallel operation in analog.

Creating a parallel architecture requires a parallel data store that can allow trained templates to be directly accessed by analog hardware. In the past, such parameters were implemented as external potentiometers that had to be tuned by hand. Furthermore, this method also used a significant number of external I/O channels, increasing package size and reducing the number of available diagnostic pins [6]. One increasingly popular method involves the use floating gates to store bias currents on-chip. These circuits have the advantage of eliminating off-chip, configurable biases such as DACs or potentiometers, and of implementing them directly in the signal path, eliminating the need for transmission gate multiplexers to share these sources. Additionally, because these circuits are simple in their architecture, they may be densely integrated.

1.1 Previous Work

Floating gate devices are traditionally divided into three primary areas: as analog memory elements; as adaptive circuit elements; and as capacitive circuit elements [7]. This section provides an overview of previous work in important selected applications incorporating these three primary uses for floating gate devices. Interestingly, however, many applications fully realize the potential of floating gates. For example, field-programmable analog arrays incorporate both analog memory and trimming elements, as well as capacitive elements. Likewise, SVMs incorporate floating gates as analog memories and as adaptive circuit elements.

1.1.1 Field-Programmable Analog Arrays

Analog integrated circuit (IC) design often requires substantial expertise in the field to design, fabricate, and test a system successfully. Additionally, such work sometimes requires several design iterations, and is therefore expensive and time consuming. The digital design flow includes several tools for the designer, such as hardware description languages, synthesis tools, and field-programmable gate arrays (FPGAs) to test designs before they are synthesized on-chip for fabrication. However, such tools for the analog designer do not exist, making it difficult to study non-ideal effects such as noise and mismatch.

Modeling and simulation provide a first principle approach for sub-system implementation. Yet, testing will ultimately be required, and for some large-scale designs, is the only means of verifying the entire design. Field-programmable analog arrays (FPAA) have been proposed [8, 9, 10] to alleviate some of these design challenges to allow rapid prototyping of analog designs using reconfigurable hardware.

Reconfigurable analog hardware may sound like an attractive alternative to traditional analog VLSI. However, it has the disadvantage of requiring a larger die area to achieve similar functionality, a problem shared by FPGAs. Additionally, FPAAs have additional parasitics due to switch matrices necessary to route signal paths. These shortcomings lead to higher power consumption and reduced bandwidth, making them impractical for some designs.

Floating gates are attractive for FPAAs because they may be used as switches in the configuration network and may be used directly as analog elements [11]. The impedance of a floating gate is inversely proportional to the number of electrons on the floating node. Thus, the quality of the switch is determined by its finite on and off impedances. FPAAs are comprised of computational analog blocks (CAB), which are analogous to slices in FPGA terminology. The use of floating gates as in-circuit elements leads to a dense architecture, reducing the size of these CABs. This is possible by eliminating the need for on-chip resistors that consume significant area. Additionally, without fixed-value resistors, signal routing complexity is further reduced as a floating gate may act as a variable resistor.

Basic analog building-blocks such as current summation and subtraction, integration, differentiation, amplification, and thresholding may be integrated into a single CAB with few transistors. These CABs include floating gates combined with operational transconductance amplifiers to perform these tasks. Furthermore, floating gates have been used directly in operational amplifiers for offset cancellation through on-chip trimming circuits [12, 13, 35].

Matrix vector multipliers have also been included on FPAAs [14]. The charge on the floating gate can act as a weight and its control gate can act as the input. Signed numbers are represented using a differential pair configuration and if cascaded together across multiple CABs, a matrix vector multiplier may be realized.

Since matrix vector multiplication (MVM) provides the foundation for many machine learning-related operations, specialized hardware for this task has been developed to exceed the performance of a general purpose FPAA hardware. One realization of this approach is in the form of a support vector machine.

1.1.2 Support Vector Machine

A support vector machine (SVM) is a type of supervised learning method for creating functions based on a set of labeled training data. These functions could be for classification or generalized regression [15]. For machine learning applications, SVMs are applied to the classification task, and have been used extensively for recognition with image [16, 17], acoustic [18, 19, 20], olfaction [21, 22], temperature [23], acceleration or vibration [24], and even biomedical signature [25] front-ends. However, these applications have largely been confined to software implementations and on digital signal processors, where speed and storage capabilities are not the limiting factor as on a system-on-chip.

SVMs have only recently been mapped onto analog structures [28], which are becoming viable alternatives to digital signal processors as a means of implementing SVMs in hardware. Utilizing parallel architectures, it is possible to compute the decision score as a single operation, based on the DC response of the system. Furthermore, where speed is not critical, it is possible to bias these circuits in the subthreshold region at reduced voltage headroom to dramatically reduce power consumption beyond that of digital implementations.

Floating gates become very important template storage elements for analog SVMs since they may be placed directly in the signal path for continuous time classification. Since SVM parameters are represented as floating-point values, they may be directly mapped onto analog hardware as configurable current sources. Additionally, SVM performance may be controlled directly through the bias of the global control gate voltage, which capacitively couples with the floating node to set bias current.

However, before realizing an SVM using analog structures, it is important to understand the training methods used and how they affect chip architecture and floating gate programming. SVM training involves finding the maximum margin between data classes, which means that the classifier attempts to maximize the distance between data points of different classes along the decision boundary (hyperplane), which in turn minimizes error when classifying incoming data points. This topic, along with soft-margin regularization theory, are important when developing the training algorithm [26].

Furthermore, these topics are critical for the designer as they have implications on any hardware implementations, which have both limited resolution and linearity, as well as device imperfections. For example, noise robustness and temperature sensitivity at the circuit-level can affect the generalized performance of an SVM by corrupting SVM parameters through capacitive coupling or temperature-dependent offsets. This often leads to deterioration of the equal error rate, a common figure of merit for SVM performance.

Analog SVMs may also implement a modified variant of the general SVM formulation. One such implementation involves working in the log domain where floating gate parameters are not stored as linear mappings of their floating point representations [27]. Here, the decision surface becomes warped due to circuit topology, resulting in a need for floating gate support vector compensation. The log domain SVM also has the unique advantage of simplifying the hardware implementation by not having to implement the SVM formulation directly and has no inherent temperature dependency. However, it still relies on floating gates for parameter storage, leading to nonlinear noise and temperature dependency.

Analog SVMs are an ideal application for large-scale floating gate programming due to the high demand for accurate and high-density parameter storage to achieve performance on par with digital SVM implementations. However, the need for a large number of support vectors has lead to increasing requirements on die area. In addition, increasing density can also result in reducing the size of the floating gate capacitors, which has implications on resolution, and is discussed in Section 2.6.

In the design of neuromorphic systems [29], SVMs have broad applications. Consequently, analog SVMs are of direct interest to this work, which will be validated through the testing of an analog SVM's floating gate array. Further details on the test chip are given in Section 5.2.

1.1.3 Imagers and Adaptive Sensors

Floating gates also have applications in the feature front-end space of neuromorphic systems. One such application is in active pixel sensors (APS), where traditional machine vision systems separate the image acquisition and processing modules. Inspired from biology, a vision chip that can integrate adaptive elements directly on the imager has the advantage of higher speed and parallelism, and higher integration.

Bandyopadhyay et al. proposed a CMOS transform imager [30] with on-chip floating gates capable of programmable matrix operations and filtering. Here, the floating gates store arbitrary analog waveforms for image transforms and nulling mismatch during matrix operations. These basis function bias generators are stored in a matrix and are multiplexed to the active row of the imager. I-V converters are used to provide voltage-mode output.

In addition to on-chip filtering, mismatch cancellation is another area of interest in designing APS imagers. Process variation can create undesirable artifacts in the image from a phenomenon called fixed pattern noise (FPN). This leads to random, deterministic spacial noise across the pixel array. FPN has traditionally been eliminated through the use of correlated double sampling, but Wong et al. has proposed a current-mode imager with self-adapting mismatch reduction [31]. Here, floating gates were added at the pixel level. During a calibration step, uniform light was shown on the imager and the pixel output voltages were read. Each pixel was adapted through hot-electron injection to produce a desired constant output voltage under these conditions.

1.2 Motivation: Rapid Configuration of Analog Memories

Due to the proliferation of analog floating gate technology and its applications, it becomes increasingly important to have a generalized framework for precision programming of floating gate arrays. As will be discussed in Chapter 2, both circuit topology and process technology will determine programming methods. Thus, a configurable programming interface is necessary to accommodate different mixed-mode designs.

This work aims to provide a testing platform for mixed-signal systems incorporating analog floating gates. This system will implement a generic interface that is easy to use for rapid testing, and is both modular and customizable in its software and firmware interfaces. Finally, the work will validate the performance of the system on a mixed-mode system-on-chip in the form of an analog support vector machine. This thesis will cover the following topics in detail.

1.3 EEPROMs for Analog Parameter Storage

This chapter covers the background of non-volatile semiconductor memory technology, how it has been used in the past in the digital domain and the evolutionary steps that have led to the use of floating gates as analog memory elements. The theory of these devices are discussed, including the important hot-electron injection and Fowler-Nordheim tunneling programming and global erasure methods, as well as previous work in the area of floating gate characterization and device limitations.

1.4 Floating Gate Test Station Design

This chapter covers the design considerations and implementation of the testing platform required for floating gate programming. Noise and shielding considerations are discussed, as well as the circuits required on the periphery of a mixed-mode design incorporating floating gates. Important topics include the design of a precision current measurement system, a tunneling and injection supply and associated control schemes, a voltage-mode digital-to-analog converter bank for bias and signal generation, and digital interfaces to mixed-signal designs.

1.5 Floating Gate Test Station Control

Chapter 4 covers the design of a hardware-based controller using the Xilinx Spartan-3 XC3S200 FPGA. This controller is responsible for providing all serial interfaces to the sub-modules of the test station, as well as instruction interfacing with a PC-based host. All logic has been written using the VHSIC Hardware Description Language (VHDL) and was optimized to minimize slice utilization.

1.6 Testing and Results

This chapter begins with the validation of the mixed-mode test platform. Such tests are necessary for calibration of the measurement circuits acting on the periphery of the floating gate chip. Next, the test chip is discussed, outlining the programming methods and some architectural considerations in the testing process. Finally, floating gate results are provided, demonstrating functionality of the complete system.

1.7 Conclusions

This thesis concludes with an overview of the work accomplished and some suggestions for future work. Also, further resources are available in the appendices, including layouts from the fabricated test chip, schematics and component lists for the test station, all code used in the hardware controller and PC-based interfaces, as well as testing scripts for the floating gate experiments. A brief tutorial is given on using the software interface.

CHAPTER 2

EEPROMs for Analog Parameter Storage

An Electrically Erasable Programmable Read-Only Memory (EEPROM) is a non-volatile storage medium typically used to store configuration parameters. EEPROMs have been used for many years in digital systems but have only recently made their way into the field of analog computation. Additionally, EEPROMs have the advantage of data retention for prolonged periods of time, typically ten years or more. This chapter covers the history of PROM technologies through the first digital EEPROM devices, as well as the theory of floating gate transistors and their limitations. This basis will then be used to design an interface for a floating gate programming test platform.

2.1 History

2.1.1 PROM

One of the earliest forms of programmable memory is the Programmable Read-Only Memory (PROM), invented in 1956 by Wen Tsing Chow at the American Bosch Arma Corporation for the US Air Force's Atlas ICBM. This memory device utilizes fuses and anti-fuses to either establish an open or short circuit connection, respectively, effectively writing a one or zero permanently to that cell of the device.

For example, if the PROM initially contained all cells programmed to logic zero, burning an anti-fuse would bridge a connection between the output of the given cell and the chip's supply voltage, establishing a digital high or '1' at the output node. A fuse will likewise produce the opposite effect. It is important to remember that the breakthrough of PROM technology meant that configuration data could be stored onto the integrated circuit post-fabrication for the first time. PROM's greatest advantage is therefore its permanent data retention capability. However, what became PROM's greatest advantage is also its greatest limitation, which lead to the development of erasable non-volatile memories.

2.1.2 EPROM

Next came the Erasable Programmable Read-Only Memory (EPROM), invented by Dr. Dov Frohman in 1971. EPROM technology utilizes floating gate transistors, which are described in greater detail in Section 2.2. EPROMs are erased by exposing the die to an ultra-violet light source. This is accomplished by penetrating light with a typical wavelength of 235 nm through a quartz erasing window in the packaging.

Many EPROM chips are mounted inside a plastic rather than ceramic package to reduce costs. These types of EPROM-based circuits are OTP or One-Time-Programmable in that they do not include a quartz window.

2.1.3 EEPROM

EEPROM technology is similar to EPROM technology, but does not require an ultra-violet light source for erasure. The methods required for writing and erasing these types of memory cells include injection and tunneling, which are discussed in greater detail in Sections 2.3 and 2.4.

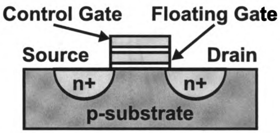


Figure 2.1. Crossection of a Floating Gate nMOS Transistor

2.1.4 Flash Memory

Flash memory has become standard in many consumer electronic devices that require large non-volatile data stores. Flash memory utilizes the same technology of EEPROMs, except they allow erasure of all cells simultaneously. This differs from regular EEPROMs that may program and erase each cell individually. By this definition, analog floating gates with a single erase may be termed analog flash memory, as the tunneling operation is a global function of the programming process.

2.2 Floating Gate Transistors

Figure 2.1 illustrates the crossectional structure of a floating gate transistor. Floating gate transistors are constructed using a traditional MOSFET with an additional gate layer (ELEC in the AMI 0.5 μm fabrication technology). Thus, a floating gate is a polysilicon layer encapsulated by silicon dioxide. Charge may then be stored on the polysilicon gate indefinitely, provided that no charge may leak through the surrounding insulator.

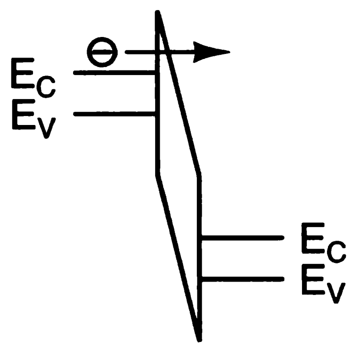


Figure 2.2. Energy band of a structure influenced by Fowler-Nordheim tunneling

The voltage of the floating gate is determined by the capacitively coupled input voltage, called the control gate voltage. The amount of charge on the floating gate determines the potential difference between the floating node and the control gate voltage. By increasing the amount of charge on the floating node, the potential across the POLY1-POLY2 capacitor increases, thus decreasing the voltage of the floating node with respect to ground, and increasing the current in a floating gate pMOS device and decreasing current in a floating gate nMOS device. Increasing the charge of the floating node is achieved through a process called hot-electron injection. Charge is removed through a process called Fowler-Nordheim tunneling.

2.3 Fowler-Nordheim Tunneling

Fowler-Nordheim (FN) tunneling is a field-assisted electron tunneling method used to remove negative charge from the floating node [32]. When a large potential is applied across a polysilicon-silicon dioxide-silicon structure, typically implemented as a MOS-capacitor, its band structure will be modified as shown schematically in Figure 2.2.

In the presence of a high electric field, electrons in the conduction band of the floating gate electrode will see a triangular energy barrier whose width is a function of the applied electric field. Adequately high electric fields will cause the barrier to become small enough for electrons to tunnel through the barrier and into the SiO_2 conduction band. Equation (2.1) gives the Fowler-Nordheim tunneling current density where h is Planck's constant, $\hbar = h/2\pi$, ϕ_b is the energy barrier at the Si-SiO_2 interface (3.2 eV), E_T is the electric field at the tunneling interface, q is the charge of an electron (1.6 x 10^{-19} C), m is the mass of a free electron (9.1 x 10^{-31} kg), and m^* is the effective mass of an electron in the band gap of SiO_2 $(0.42 \cdot m)$.

$$J = \frac{q^3}{8\pi h\phi_b}\frac{m}{m^*}E_T^2 exp\left[\frac{-4\sqrt{2m^*}\frac{\phi_b^{3/2}}{3\hbar q}}{E_T}\right] \tag{2.1}$$

The tunneling mechanism is independent of temperature. However, the number of electrons available for tunneling in the conduction band of the polysilicon gate is dependent on temperature. In addition, the Fowler-Nordheim tunnel current density is exponentially dependent on the applied electric field.

Using Equation (2.1), it is possible to calculate the current density in the AMI C5N process, which has a 13.5 nm gate oxide thickness [33]. Equation (2.2) shows a numerical expression of the tunneling current density with respect to applied tunneling voltage (V_T). For 15 V applied at the tunneling pin (and 0 V at the floating gate), it is expected that the Fowler-Nordheim tunneling current density will be 176.2 A/m^2 or 176.2 pA/μm^2, which translates to 158.6 pA through the 1.5μm/0.6μm MOS capacitor.

$$J[A/m^2] = 1.147 \cdot 10^{-6}\left[\frac{F}{Wb}\right]\left[\frac{V_T}{13.5 \cdot 10^{-9}}\right]^2\left[\frac{V^2}{m^2}\right] exp\left[\frac{-25.34 \cdot 10^{-9}\left[\frac{N}{C}\right]}{\left[\frac{V_T}{13.5 \cdot 10^{-9}}\right]^2 [V^2/m^2]}\right] \tag{2.2}$$

Although the tunneling current may be estimated by multiplying the current density by the area of the MOS capacitor, this assumes that the current density is uniform across the interface. In fabricated devices, however, this is an unlikely scenario due to fringe fields and device mismatch.

2.4 Hot-Electron Injection

Hot-electron injection is a process by which electrons are put onto the floating node by gaining enough energy to surmount the SiO_2 barrier. When the minority carriers that flow through the channel of a MOS device are in the presence of a large source-to-drain bias (for a pMOS transistor), the carriers are heated by this large electric field and their energy distribution is increased. This leads to impact ionization at the drain of the device, generating both majority and minority carriers. The minority carriers are collected at the drain, and can overcome the SiO_2 barrier if they gain sufficient energy. This process moves these carriers from the drain and onto the gate, a process commonly called the hot-electron injection gate current.

It is important to note that with hot-electron injection, it is only feasible to move electrons onto the floating node and they cannot be removed by the same means. A mechanism called hot-hole injection has been demonstrated as a complementary operation to neutralize the negatively charged gate, but is not widely used due to its low hot-hole injection gate current.

There are several models that have been used to characterize the hot-electron injection current such as the lucky, effective electron temperature, and other physical models. However, unlike Fowler-Nordheim tunneling, there is no closed form expression for the gate current and therefore these models are simply quantitative. An empirical model that can be used for programming floating gate arrays was proposed by Bandyopadhyay et al. [34] and is given in Equation (2.3).

$$ln\left(\frac{\Delta I}{I_{S_0}}\right) = K_2\left(V_{DS}\right)\left[ln\left(\frac{I_{initial}}{I_{S_0}}\right)\right]^2 + K_1\left(V_{DS}\right) ln\left(\frac{I_{initial}}{I_{S_0}}\right) + K_0\left(V_{DS}\right) \quad (2.3)$$

Here, K_2, K_1, and K_0 are unitless functions of the source-to-drain voltage and I_{S_0} is a bias current. This expression has been used to accurately model the best source-to-drain voltage at the injection node during programming to minimize the

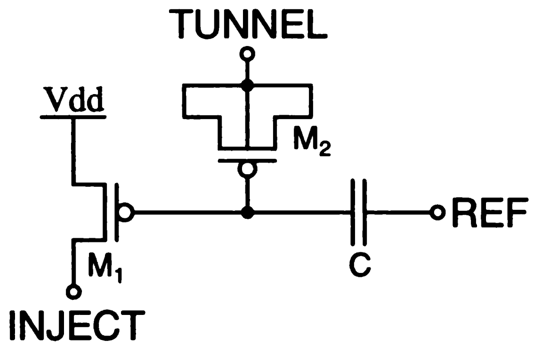


Figure 2.3. Floating Gate Cell Schematic

number of injection pulses.

Figure 2.3 shows the schematic representation of a floating gate cell. Hot-electron injection is achieved by pulsing the drain of transistor M_1. Transistor M_2 is the MOS capacitor discussed in the Fowler-Nordheim tunneling section. The TUNNEL pin is a global erase, meaning it is tied to all floating gates in the array. In this example, the injection pin is also the cell output current. The REF pin is the control gate voltage which capacitively couples with the floating node to set the gate voltage of the device. The capacitance C is a parallel plate capacitor (shown in layout in Section A.1). It is comprised of a parallel plate (POLY1-POLY2) capacitor separated by an SiO_2 dielectric layer.

2.5 Analog Floating-Gate Programming Procedure

An algorithm to initialize floating gate arrays is given in Figure 2.4. First, the array characteristics must be determined to equalize the floating gate cells to a constant current. This is done by measuring the output source current of each floating gate cell. An initial control gate voltage is chosen such that all floating gates are conducting a measurable current. The maximum cell current in the array is then programmed to every floating gate.

Next, the control gate voltage is increased until the floating gate cells output a

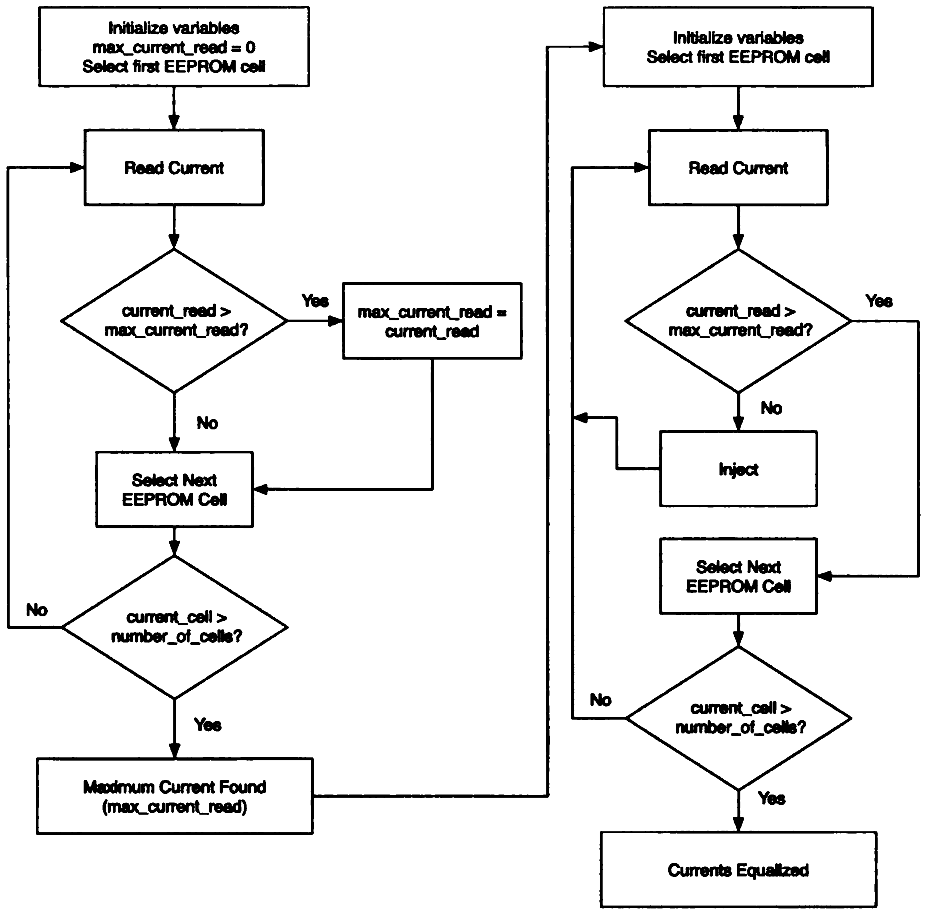


Figure 2.4. Floating Gate Transistor Equalization Procedure

minimally resolvable current. Due to mismatch, not all cells will maintain the initial targeted current during control gate voltage scaling. Thus, the equalization procedure must be repeated. This process continues until the array is equalized at less than 10 nA at an unspecified control gate voltage. Finally, the array is programmed to its targeted current levels specified by the application using the previously discussed hot-electron injection methodology.

2.6 Programming Precision

It has been shown that floating gates can reliably achieve an equivalent accuracy of greater than 13 bits of resolution [6]. The accuracy of floating gate programming is highly dependent on the measurement circuit's ability to resolve drain current changes between injection pulses, as well as the minimum charge transfer possible using hot-electron injection. Srinivasan et al. [35] defined a figure of merit (FOM) given in Equation (2.4) and showed that programming accuracy was constant in the subthreshold region and improves in strong inversion.

$$FOM = -log_2\left(\frac{\Delta I}{I}\right) \tag{2.4}$$

Here, ΔI is the minimum change in drain current possible for a given I, or the existing bias current of the floating gate transistor. Essentially, the FOM shows the resolution of an injection pulse, which for the 0.5 μm process is between 3.2-4.6 bits up to 1 μA of current and increases quadratically beyond this limit. This increase is due to a modified $\Delta I/I$ relationship as the drain current equation for strong inversion contains an overdrive term that depends on the initial floating gate voltage. Thus, to attain high accuracy, higher floating gate currents are required.

The precision is theoretically limited to the ability to inject a single electron onto the floating gate. In addition, resolution is also determined by the floating gate capacitance (C_{FG}). Since $\Delta V_{FG} = \Delta Q/C_{FG}$, the change in charge will produce a larger change in potential across the floating gate if capacitance is decreased. Thus, larger floating gate capacitors will increase resolution.

With a C_{FG} of 1 pF, [35] showed it is theoretically possible to achieve 17.82 bits of accuracy with a charge transfer of one electron in weak inversion and 20.09 bits in strong inversion. However, due to the probabilistic nature of hot-electron injection, it is difficult to achieve transfer of a single electron to the floating gate.

2.7 Charge Retention Characteristics

Due to the high quality SiO_2 insulator surrounding the floating node, floating gates have the ability to store charge for long periods of time. Charge retention is limited only by defect densities, which increase under stress such as a high-temperature bake during PCB population or from a high number of injection/tunneling cycles [32]. Following an initial programming cycle, a slight drift in the floating gate current results from interface trap site settling. This loss of charge from the SiO_2 is when backtunneling to the silicon bands occurs. Interface trap density may be reduced substantially, however, through a hydrogen annealing step during device fabrication. Unfortunately, however, this step is not present in a standard digital CMOS process.

Thermionic emission is responsible for long-term charge loss in floating gate transistors and is a function of temperature and time. This phenomenon results when electrons are emitted over the energy barrier toward the control gate or substrate. Equation (2.5) expresses the fraction of charge lost where $Q(t)$ is the floating gate charge at time t, $Q(0)$ is the initial charge, k is Boltzmann's constant ($1.38 \cdot 10^{-23} J/K$), T is temperature in Kelvin, ν is the relaxation frequency of electrons in polysilicon, and ϕ_B is the Si-SiO_2 barrier potential.

$$\frac{Q(t)}{Q(0)} = exp\left[-t\nu \cdot exp\left(\frac{-\phi_B}{kT}\right)\right] \tag{2.5}$$

It can be seen that thermionic emission increases with temperature. Therefore, retention tests typically use a series of accelerated conditions, such as storing the device at temperatures up to 350°C. From these experiments, the relaxation frequency and Si-SiO_2 barrier potential may be extracted. Srinivasan et al. [35] found a ϕ_B of 0.9 eV and ν of 60 Hz for the 0.5 μm process by plotting $Q(t)/Q(0)$ for temperatures between 250°C and 350°C and applying the data to the above model. Thus, over the course of ten years, a charge loss of $1.14 \cdot 10^{-3}\%$ is expected at room temperature

(25°C). Even in extreme environments where device temperatures can reach 100°C, charge retention is still 98.5% over the same period of time. Consequently, floating gates are very attractive as long-term analog storage elements.

2.8 Temperature Dependency

Floating gates suffer from the same temperature dependency as a standard MOS device. The carrier mobility and threshold voltage are the predominant temperature-dependent parameters. These are evident in the source current equations for a pMOS transistor in both weak (2.6) and strong (2.7) inversion regions [36].

$$I_S \cong \frac{W}{L} I_{D0} exp\left(\frac{V_{DD} - V_{CG} + V_{FG}}{n(kT/q)}\right) \tag{2.6}$$

$$I_S = K'\frac{W}{L}\left[(V_{DD} - V_{CG} + V_{FG} - V_T) - \frac{V_{SD}}{2}\right] V_{SD} \tag{2.7}$$

In subthreshold, I_{D0} is a process-dependent pre-exponential constant dependent on V_T and n is the subthreshold slope factor, which typically $1 < n < 3$. Note that Equation (2.6) does not model the moderate inversion transition region. In strong inversion, K' is a process parameter that is dependent on the mobility and capacitance of the gate oxide layer. It can be seen in these equations that the pMOS transistor source current is directly proportional to temperature.

Various topologies have been used to offset temperature dependency in current references. However, these circuits traditionally involve fixing the operating point through fixed-width transistors. One proposed method [37] uses a floating gate as a trimmable element in a temperature-insensitive current source. However, such references will require current mirrors to bias the control gate voltage, and due to device mismatch, mirrors can suffer from offset errors. Figure 2.5 shows a floating gate-based topology for offset removal and was proposed in [38]. Here, C_1 and C_2 act

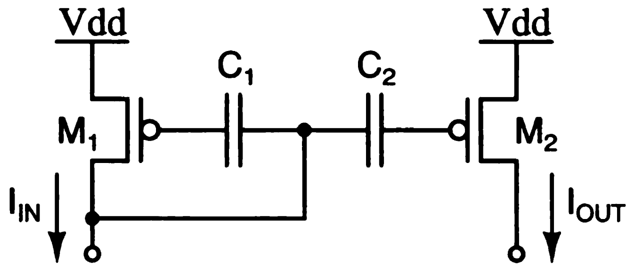


Figure 2.5. Floating Gate Current Mirror

as programmable multiplicative factors of the mirror. An ideal mirror can be realized by calibrating the floating gates such that both devices have the same threshold voltages.

Using a current reference instead of a fixed voltage bias for the control gate has an additional advantage of being supply voltage invariant, and thus increases the supply rejection capability of the floating gate cell. This is on account of the control gate reference V_{CG} tracking the source-to-gate voltage of a diode-connected pMOS transistor. Consequently, a direct relationship (neglecting mismatch) may be derived between the fixed reference current and the floating gate output current, scaled by the floating gate voltage V_{C2} as given in Equation (2.8) below.

$$\frac{I_{OUT}}{I_{IN}} = exp\left(\frac{V_{C2} - V_{C1}}{nkT/q}\right) \tag{2.8}$$

Equation (2.8) gives the output current scaling factor for weak inversion and is based on the difference in charge stored on the floating gate cell and the floating gate reference. Furthermore, it is possible to decrease floating gate currents in relation to the reference current through increasing the charge on C_1, which may be useful if the sink current I_{IN} is significantly large.

This analysis has shown that techniques exist to compensate for device and temperature variations in floating gate elements. Although these topologies solve some operational deficiencies, they create new problems such as increased circuit complexity, which leads to calibration and testing challenges. Also, these circuits offer reduced flexibility as the temperature-independent biases must be designed around a target current or a specific region of operation. Furthermore, such bias circuits add to the overall power budget of the chip and must be designed to meet the targeted specifications.

2.9 Summary

In this chapter, the fundamentals of erasable programmable solid-state memories were described. The history of non-volatile semiconductor memories was discussed, laying the foundation for an analog memory storage element. Floating gate devices have the ability to replace fixed current sources and sinks in analog integrated circuits. Their flexibility in reprogrammability allows them to be used for not only calibration and trimming, but as template storage, opening up the possibility for on-chip learning.

Next, the mixed-signal test station with automatic floating gate cell programming support will be described. It can be seen that mixed-signal systems-on-chip with floating gate cells require extensive peripheral circuitry. This work aims to provide this support in a modular fashion for rapid design evaluation and validation.

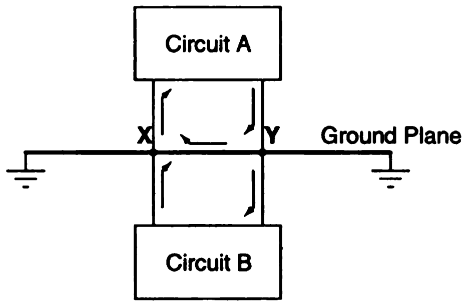
CHAPTER 3

Floating Gate Test Station Design

The need for a test station to automatically calibrate floating gate transistors and the difficulty in setting precision current sources for analog computation was outlined in Chapters 1 and 2. The system required to perform the operations of hot-electron injection and tunneling, as well as performing data acquisition for the forward-feedback process of floating gate programming has been carefully designed to meet the targeted specifications. In addition, this system must also be a fully-functional mixed-signal test station, and be able to set the bias conditions for the device under test and provide digital control interfaces. This chapter discusses the design of each subsystem and its targeted performance.

3.1 Noise and Shielding Considerations

When designing analog and mixed-signal systems, noise becomes an increasingly apparent limitation when trying to resolve signals in the millivolt and sub-millivolt range. Additionally, noise may not only be a random phenomenon, as detector systems can pick up correlated, spurious signals as well.



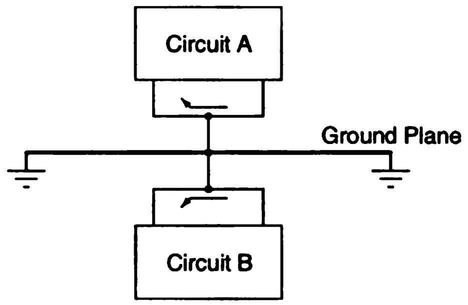


Figure 3.1. Circuits with Ground Loops **Figure 3.2.** Local Return Paths Only

3.1.1 Shared Current Paths

The most common type of unwanted signal transfer is caused by ground loops, also known as shared signal paths [39]. Figure 3.1 illustrates shared current paths for two independent circuits. Circuits A and B both have their own dedicated current return paths. However, because current will target the path of least resistance, it will flow through the ground plane. Large currents between nodes X and Y cause a potential difference across the ground plane, inducing an additional voltage at node X.

Figure 3.2 shows an improved layout of circuits A and B that limits current flow in the ground plane to currents entering or leaving circuits A and B only. Each circuit has its own dedicated current return path, so no shared DC path exists, thus limiting any transient spikes on circuit A to affect circuit B.

However, paths need not exist explicitly for ground loops to form. In time-varying signals, parasitic capacitances may exist between these nodes and the ground plane. Thus, spurious voltages are formed on the critical node as a result of AC coupling and charge injection with the ground plane. Furthermore, induction can cause charge injection onto critical nodes as well if they exist between time-varying signals and the return path. Therefore, it is essential to tightly route the signal and its return path, minimizing induction through field cancellation.

Shared current paths have been minimized in layout by connecting all ground and power pins for each circuit to centralized nets [40]. These nets are connected to the power planes though single or concentrated vias. Furthermore, AC coupling through parasitic capacitance has been reduced by routing time-varying signals away from sensitive DC signals as well as through shielding techniques.

3.1.2 Shielding Techniques

One method to reduce the effects of electromagnetic interference is to use contiguous shielding. A contiguous shield is one that completely surrounds sensitive signals, thus attenuating and reflecting the majority of any incident wave. The fraction of the wave that is reflected is given by Equation (3.1).

$$E_{0,r} = E_0 \left(1 - \frac{Z_{shield}}{Z_0}\right) \tag{3.1}$$

Reflection is high because the impedance of free space (Z_0) is approximately 377 Ω and the impedance of the conductor is much lower, thus $E_{0,r} \approx E_0$. Additionally, the signal is attenuated because any absorbed wave produces a local current whose magnetic field opposes the incident electromagnetic wave. The total current in the shield decreases as the wave penetrates deeper into the material, and the wave penetration depth (δ) may be calculated by the equation given in (3.2) where f is the frequency of the incident electromagnetic wave, μ_r is the permeability and ρ is the resistivity of the conductor.

$$\delta = \frac{1}{2 \cdot 10^{-4}} \left[\sqrt{cm \cdot s^{-1}}\right] \sqrt{\frac{\rho}{\mu_r f}} \tag{3.2}$$

Thus, if a shield is adequately thick, has a high enough conductivity and low permeability, the majority of the incident electromagnetic interference will be isolated from the inside of the enclosure. Therefore, during testing, a contiguous shield will be placed over the test setup to reduce the effects of light and RF pickup.

In addition to contiguous shielding, additional internal shielding methods are employed to prevent capacitive coupling from transferring interference onto sensitive nodes. The technique employed is called field line pinning and works by placing a conductor between the interfering source and the critical node, thus absorbing the field lines and shielding the node.

This is implemented by placing all digital signals on the bottom layer and analog signals on the top layer. Two inner layers act as ground and supply planes, providing the intermediate conductor. The dielectric material between the outer signal layers and inner planes is fiberglass with a dielectric constant of approximately 4–4.9 and a thickness of 12 mils [41]. Furthermore, the inner planes are separated by a 28 mil core of laminate. Thus, a high capacitance exists between the interference node and the intermediate node, allowing the ground plane to absorb the field lines of noisy digital signals.

3.2 System Architecture

Figure 3.3 shows a block diagram of the mixed-signal floating gate test station and its interface with external digital control via a field-programmable gate array. All voltage-mode channels are passed through a 100-pin connector to an adjacent board that houses the device under test. Current-mode circuits interface through BNC connectors that provide contiguous shielding for noise reduction.

The main digital bus runs vertically on the underside of the board and is responsible for controlling all sub-circuits. In addition, 13 channels of digital I/O are reserved for the 100-pin header, and connect to the daughterboard. Each sub-circuit is discussed in the remaining sections. The FPGA-based controller is discussed in detail in Chapter 4.

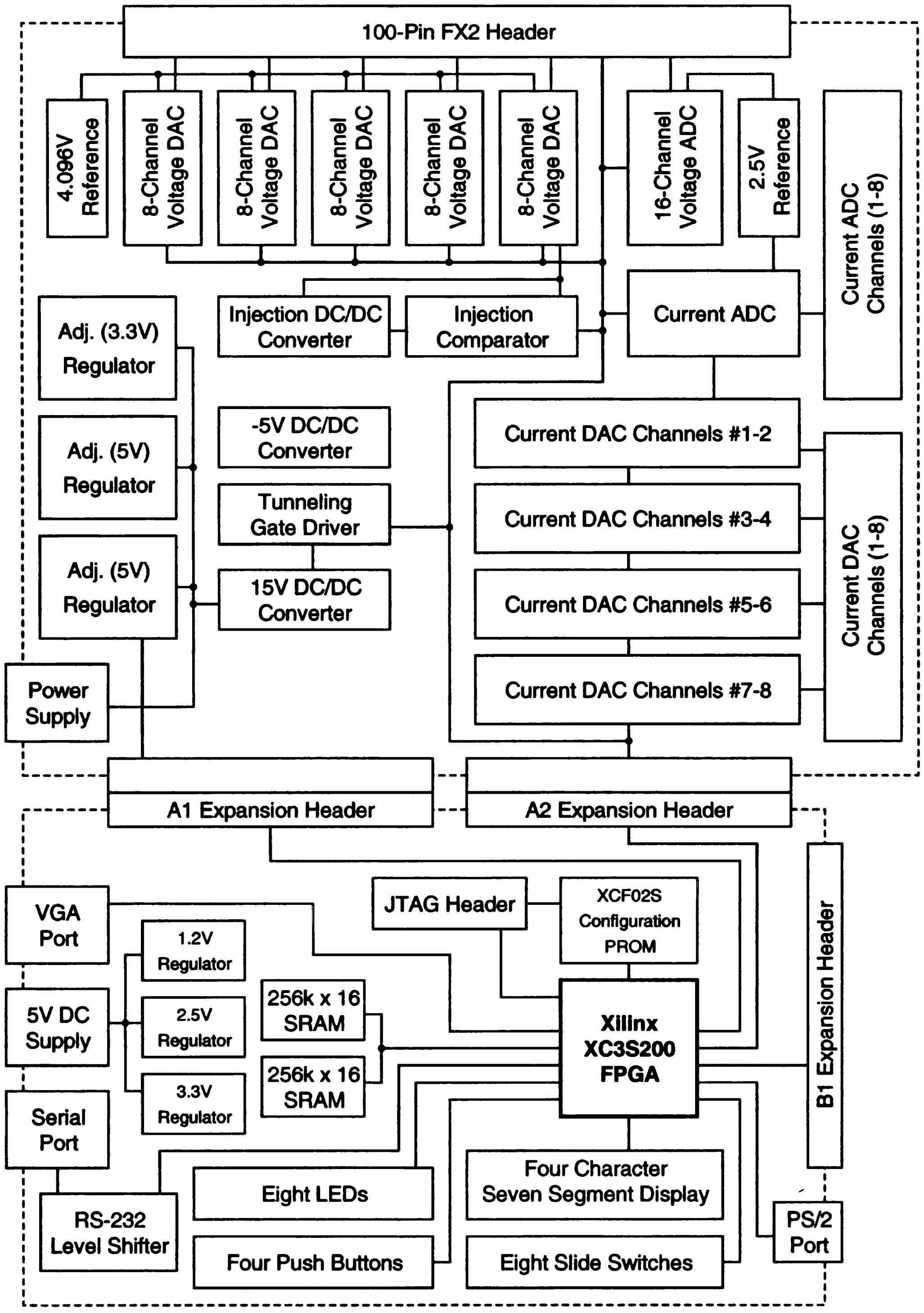


Figure 3.3. Block Diagram of the Mixed-Signal Test Station

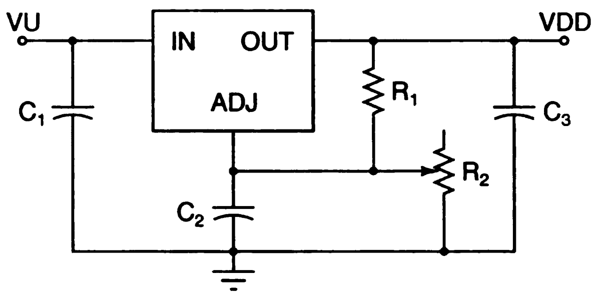


Figure 3.4. Voltage Regulation

3.3 Power Circuits

Proper supply regulation is important because it provides a stable voltage to sensitive analog circuits whose output may become distorted if the power supply rejection ratio is low. Furthermore, it is important to have separate supplies for both analog and digital signals, since Section 3.1 showed that digital switching can cause considerable noise on the supply rail. The board contains three linear voltage regulators and three DC/DC converters. A 15 V boost converter is required for Fowler-Nordheim tunneling in the AMI C5N 0.5 μm CMOS process. Additionally, the digital potentiometers and operational amplifiers in the current ADC and DAC circuits require a -5 V VSS supply rail. The hot-electron injection circuit requires a -2 V VSS supply rail. These are provided by two switched-capacitor voltage converters.

3.3.1 Voltage Regulation

The board is powered by three LM1086 1.5 A low dropout positive voltage regulators [42] using the adjustable topology shown in Figure 3.4. The voltage VU is provided by a 9 V nominal unregulated power adapter with an open circuit voltage measured at 13 V. Capacitors C_1, C_2, and C_3 are 10 μF tantalums rated for 16 V. Additionally, resistors R_1 and R_2 make up a voltage divider that allows adjustment of the regulated voltage.

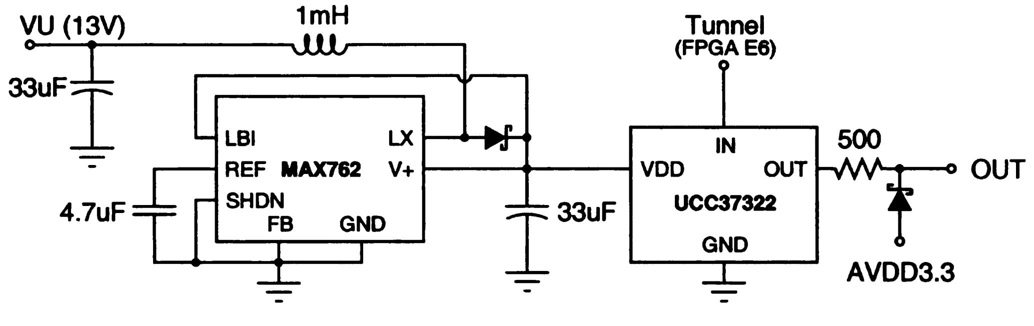


Figure 3.5. Tunneling Control Circuit Schematic

Tuning the regulators becomes important when it comes to precision measurement because it determines the supply rails and reference voltages of the ADCs and DACs. Any small fluctuations will directly affect the scaling of any digital input/output codes.

The three voltage regulation circuits are shown in Appendix B.16. U0 is calibrated to 5 V and can provide power directly to the FPGA development board, eliminating the need for the board's own regulated 5 V supply. U1 also provides 5 V, but is used to supply the test station's own circuits. U2 provides 3.3 V to the test chip, as well as to the Fowler-Nordheim tunneling circuit for the idle voltage.

3.3.2 Fowler-Nordheim Tunneling Supply and Control

The tunneling supply is provided by the Maxim MAX762 15 V step-up switching regulator [43], which is capable of providing 150 mA of output current at an efficiency greater than 80%. Figure 3.5 shows the configuration of the tunneling supply connected to its control logic via the UCC37322 gate driver [44].

The input voltage range for the MAX762 is 2 V to 16.5 V. To reduce the load on the 5 V regulators, the unregulated supply is connected directly to the boost converter, which also increases the efficiency of the converter by reducing the switching frequency and increasing the gate-to-source voltage of the internal MOSFET. The device is

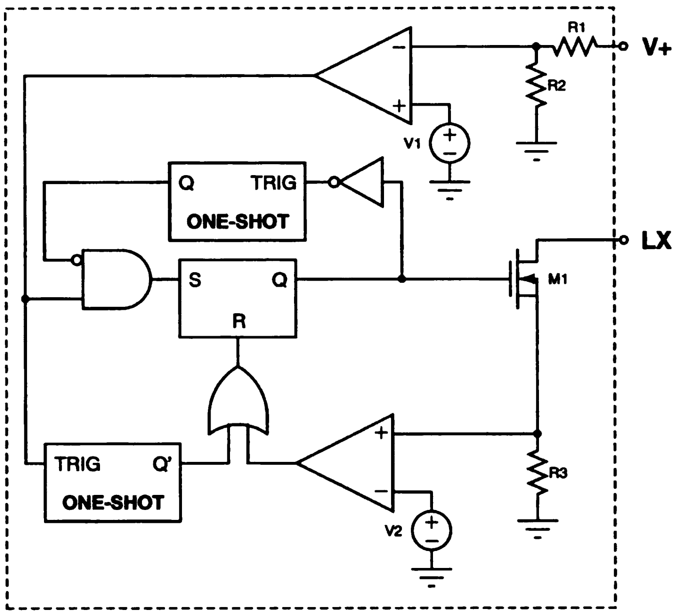


Figure 3.6. MAX762 Simplified Schematic Diagram

operated in the bootstrapped mode, meaning its supply current is drawn through the output node.

Figure 3.6 shows a block diagram of the MAX762's internals, simplified from the datasheet's own diagram to emphasize the features used in the circuit given in Figure 3.5. The basic theory of operation of the device is as follows. The regulated output voltage is set by charging up the 33 μF output filter capacitor. This is accomplished by pulse-frequency modulating the LX line. First, LX is pulled to ground, causing current to ramp up inside the inductor. Next, LX is released and becomes a high-impedance node. Since current cannot change instantaneously inside an inductor, the current is forced through the Schottky barrier diode and charges up the output capacitor.

A closer look at Figure 3.6 reveals that there are two primary functions of the MAX762: to detect undervoltages as well as to limit the current flowing through the inductor. Resistors R_1 and R_2 act as a voltage divider and connect to the undervoltage comparator. This output is fed to an S-R latch as well as a one-shot monostable timer. The purpose of the latch is to determine whether the internal N-channel power MOSFET M_1 should be turned on.

When enabled, the power MOSFET allows the increased current in the inductor to flow through R_3. When the voltage across this resistor exceeds V_2, the peak current comparator output goes high. This causes the S-R latch's reset to go high and sets the output Q to low, turning off the power MOSFET. This prevents the chip from sinking more than its 1 A peak current limit.

In addition to a current limit, the MAX762 also limits the "on" pulse width through the one-shot monostable multivibrator at the output of the undervoltage comparator. When the comparator detects a low voltage at V+, the output of the timer goes low for 8 μs. This sets the maximum time at which the LX pin is enabled through M_1. If the peak current is exceeded, the reset on the S-R latch is still tripped via the current comparator.

A minimum delay between pulses is controlled by the second one-shot monostable connected to the output of the S-R latch through an inverter. This disables the S input on the latch for a duration of 1.3 μs. After this minimum time, M_1 either remains off if the output is in regulation, otherwise the cycle repeats if the output is out of regulation.

The output of the MAX762 acts as the power supply to the Texas Instruments UCC37322 high-speed, low-side MOSFET driver. The UCC37322 provides an output of either 0 V or 15 V based on input logic issued from the FPGA. The output of this MOSFET driver is connected to a resistor and Schottky barrier diode in series. When the output of the driver is 0 V, the diode prevents the 'OUT' node in Figure 3.5 from falling below one diode drop less than the output of the 3.3 V regulator. The resistor provides a low current path back to the grounded output of the UCC37322.

When the output of the MOSFET driver is 15 V, the diode does not conduct

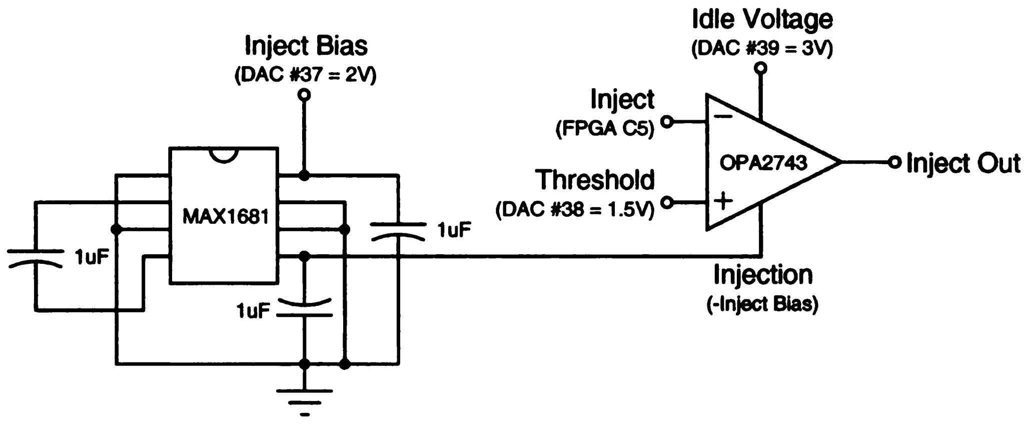


Figure 3.7. Injection Comparator Circuit

and the 'OUT' node is set directly to the output of the driver. Because the global tunneling pin is a high-impedance node and tunneling currents are on the order of picoamperes, the voltage drop across the resistor is primarily due to the reverse bias leakage of the Schottky barrier diode, which can be in excess of 1 mA.

3.3.3 Hot-electron Injection Supply and Control

Hot-electron injection is the method used to add charge to the floating gate. Figure 3.7 shows the schematic of a comparator circuit to switch the global injection pin from a user-set idle voltage and the -2 V injection voltage.

The MAX1681 [45] is a frequency-selectable, switched-capacitor voltage converter that inverts a 2 V input voltage to -2 V for the VSS supply rail on the OPA2743 [46] op-amp comparator. Since the injection current is proportional to the drain-to-source voltage drop across a floating gate transistor, the inverted voltage of the MAX1681 may be adjusted during runtime to modulate the amount of injection current. Thus, both amplitude modulation and pulse-width modulation programming methods are supported. However, the minimum input voltage is 2 V, therefore only larger injection currents are supported.

Since transistors can break down at 8 V in the AMI C5N process, it is not recommended to exceed a 4 V positive voltage input. This is prevented by the 4.096 V precision reference used in the voltage DAC, which is used to set the positive voltage supply of the MAX1681.

The OPA2743 is a high speed, rail-to-rail operational amplifier. The device has a slew rate of 10 V/μs, allowing injection pulse widths as small as 1 μs. The positive supply rail is connected to voltage DAC channel #39, and is initialized to 3.3 V during test station startup. The negative supply rail is connected to the output of the MAX1681 and is controlled by DAC channel #37. The threshold voltage for the comparator is set by DAC channel #38 and is configured at startup to 1.5 V, a midpoint for FPGA logic signals.

The injection pulse is provided by the FPGA and uses negative logic. Therefore, when the negative terminal of the op-amp is set to low, no injection occurs and the 'Inject Out' node shown in Figure 3.7 is set to the positive, idle voltage. Likewise, a logic high on the negative input terminal produces the negative, injection bias as output.

3.4 Voltage Digital-to-Analog Conversion

Five LTC2600 octal 16-bit voltage-output DACs [47] are used to provide 40 channels of voltage digital-to-analog conversion. A block diagram of the LTC2600 is shown in Figure 3.8.

The LTC2600 is programmed in a straightforward manner using a serial peripheral interface that is described in detail in Section 4.9. After receiving the update instruction in the 32-bit shift register, a decoder updates the DAC register associated with the selected channel. The ideal output voltage is given by the following transfer function:

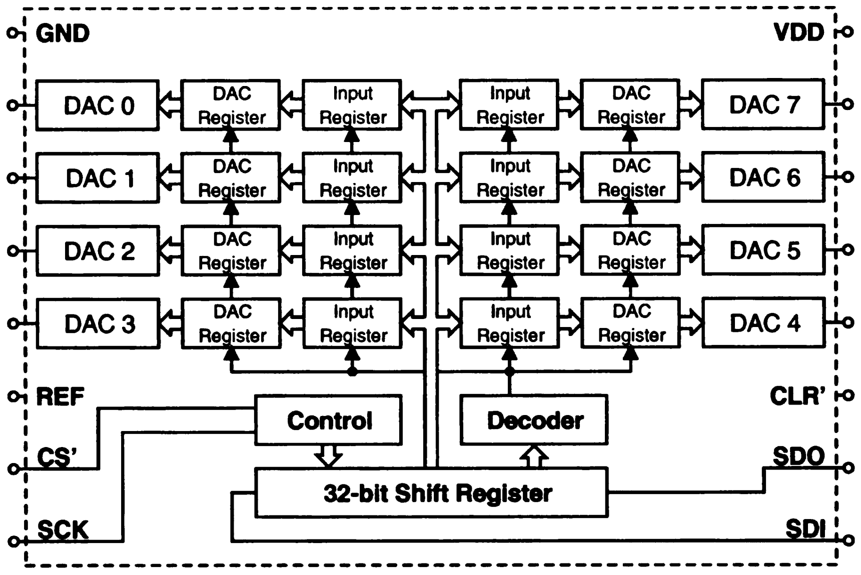


Figure 3.8. LTC2600 Block Diagram

$$V_{OUT} = \left(\frac{x}{2^{16}}\right) \cdot V_{REF} \tag{3.3}$$

where x is the input code given in 24-bit binary and V_{REF} is the reference voltage set via the REF pin.

The REF pin is connected to an LT1461 precision voltage reference [48] set to 4.096 V, which sets the maximum voltage output of each DAC channel. In addition, each channel is capable of sinking or sourcing up to 15 mA, making it an ideal supply for the hot-electron injection circuit.

3.5 Voltage Analog-to-Digital Conversion

Sixteen channels of ADC input are made available through the use of the Linear Technology LTC2418 24-bit delta-sigma converter [49]. Figure 3.9 shows a block diagram of the LTC2418's internals, as well as its configuration on the board.

The LTC2418 is a 3rd order $\Delta\Sigma$ ADC, which is an advantageous ADC architec-

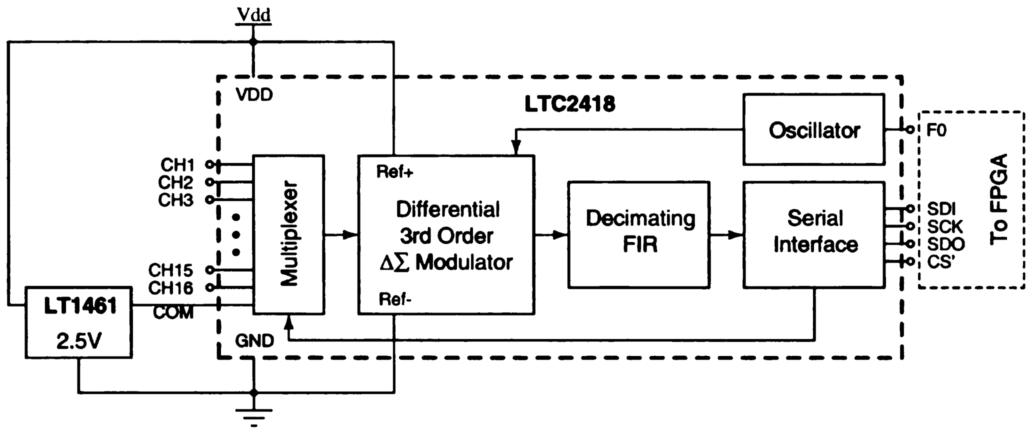


Figure 3.9. 16-Channel Voltage ADC

ture due to its high resolution and linearity, which is critical for precision measurement. Also, unlike traditional ΔΣ ADCs, this part has zero latency, meaning data is available before beginning the next conversion cycle. The ADC is capable of handling differential input, however, an LT1461 precision voltage reference of 2.5 V connects to the common terminal for single-ended input. Therefore, each of the sixteen channels has a rail-to-rail input voltage range as is given by the following constraint equation:

$$-0.5 \cdot V_{REF} \leq V_{IN} \leq 0.5 \cdot V_{REF} \tag{3.4}$$

where $(REF_{+} - REF_{-}) = V_{REF}$ and is equal to 5 V and $V_{IN} = IN_{+} - 2.5$. Additionally, the digital output code monotonically increases with input voltage, meaning the output code is not represented in two's complement. Equation (3.5) translates the received binary code into an equivalent analog voltage between 0 and 5 V.

$$2 \cdot 5 \cdot \frac{x}{2^{24}} - 2.5 \tag{3.5}$$

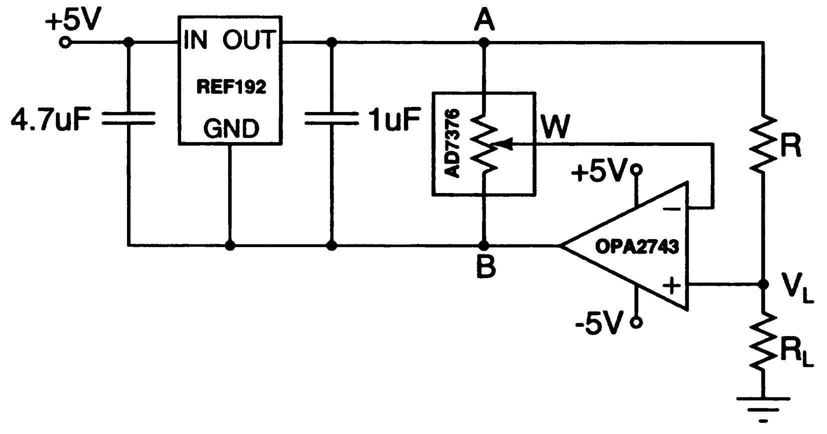


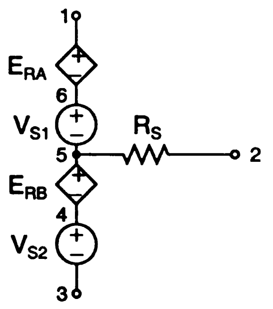
Figure 3.10. Precision Current Source

3.6 Multi-channel Current Digital-to-Analog Conversion

Figure 3.10 shows the schematic for a single channel precision current reference [50]. Eight channels have been included for providing input bias currents for nMOS-based input current sinks.

The basic circuit operation is as follows. The REF192 precision voltage reference [51] maintains a constant 2.5 V between nodes A and B, and is bypassed by a 1 μF tantalum capacitor. An AD7376 digital potentiometer [52] acts as a voltage divider by setting the value of the wiper at node W. The op-amp is in a negative feedback loop, forcing the positive and negative input terminals to a virtual ground. This causes nodes V_L and W to be set to the same voltage if ignoring the offset of the amplifier.

Therefore, by setting the value of the wiper resistance in the digital potentiometer, the voltage across the resistor R is fixed by the voltage divider, and Ohm's Law determines the current flowing through the resistor and into the load. Also, since the current range is determined by the value of the resistor, the value of R has been spaced exponentially across channels to give the highest range.



```
* AD7376 Model
*              A W B
.SUBCKT POT10K 1 2 3 7 8
ERA 1 6 VALUE = {V(7,8)*10K*I(VS1)}
VS1 6 5 0
RS  5 2 1
ERB 5 4 VALUE = {(1-V(7,8))*10K*I(VS2)}
VS2 4 3 0
.ENDS
```

Figure 3.11. AD7376 Digital Potentiometer SPICE Model Schematic and Code

The output voltage range at V_L is from -5 to 2.4 V. The AD7376 digital potentiometer has a VSS of -5 V, making its wiper voltage swing capable of ±5 V. The OPA2743 has a rail-to-rail input voltage swing, allowing V_L to drop to the negative supply. If V_L exceeds 2.4 V, the precision voltage reference will decrease its output voltage if V_{AW} is set to 2.5 V, since the supply range requires a minimum of 2.6 V.

Figure 3.12 shows a SPICE simulation of the current DAC's DC response. The macromodels for the REF192 and OPA2743 were used during simulation. However, the AD7376 was modeled separately as two voltage-controlled voltage sources in series with two independent DC sources set to 0 V. The digital potentiometer is modeled in Figure 3.11. The voltage across the E_{RA} dependent source is proportional to the resistance of the potentiometer multiplied by the current through the V_{S1} independent source and the value of an external source ($V_{7,8}$, which sets the wiper ratio). The other dependent source is set in a similar manner, but its voltage is set according to the remaining fraction of the wiper.

The value of R in Figure 3.10 was set to 2.2 MΩ, which is the value of R9, R11, R13, and R15 on the test station. These correspond to the first four DAC channels. Channels #5 and #6 are set to 100 kΩ and the remaining two channels are set to 10 kΩ, providing a larger source range. It can be seen from simulation that the current DAC is linear across its entire range and is inversely proportional to the code set in the potentiometer's internal register.

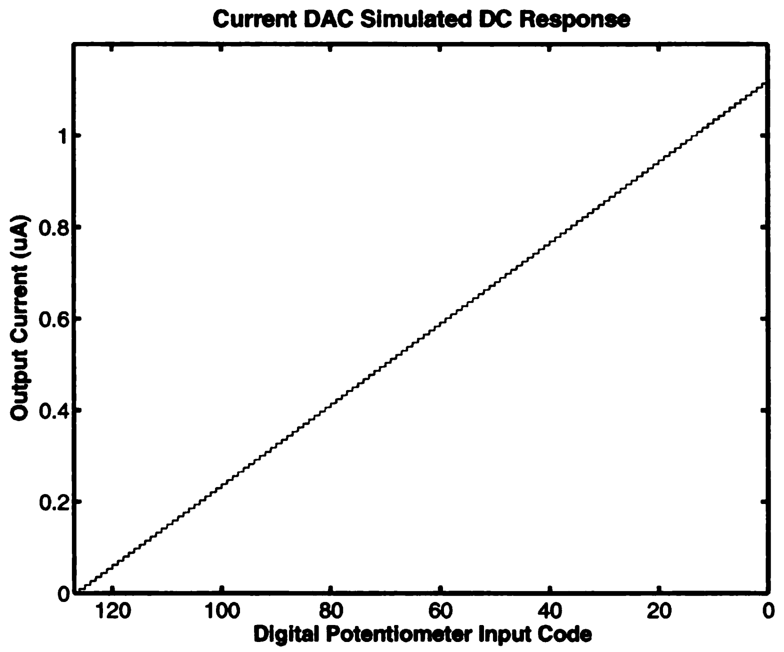


Figure 3.12. Precision Current Source DC Response

Figure 3.13 shows the step response of the current DAC, which tests the stability of the circuit. Although there is a slight overshoot and ringing, the circuit settles in under 3 μs. This ringing may be eliminated by changing to a lower bandwidth op-amp such as the TLC2252 as used in the current ADC; however, this will result in a longer settling time.

3.7 Multi-channel Current A/D Conversion

Precision current measurement is accomplished through the use of a I-V converter as shown in Figure 3.14. The ADG715 [53] contains eight channels of serially-controlled, single pole, single throw switches that are used to multiplex input currents. The TLC2252 [54] that is in a negative feedback configuration allows I-V conversion across the AD7376 digital potentiometer. Additionally, the output voltage of the op-amp is sampled by the 24-bit, LTC2415-1 $\Delta\Sigma$ ADC [55].

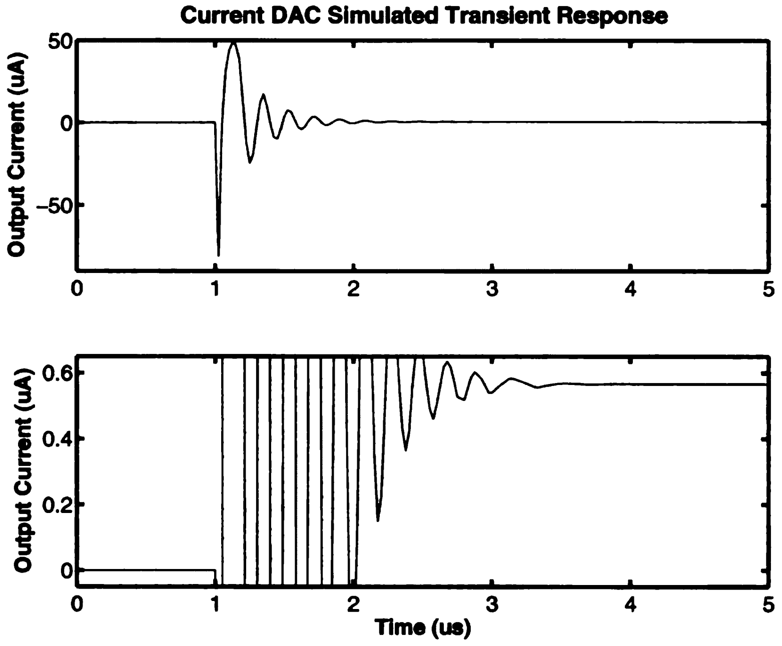


Figure 3.13. Precision Current Source Transient Startup Response

Not shown in the figure for simplicity is a unity gain buffer between the output of the I-V converter and the input of the ADC. This buffer isolates the switched-capacitor front-end of the ADC from the I-V circuit. The LTC2415-1 has a dynamic input current that is set by this switched capacitor network at a frequency of half the conversion clock rate (see Section 4.4) and is dependent on the source impedance and input capacitance of the external circuit.

Analog-to-digital conversion involves turning on a single switch in the ADG715 and calibrating the gain of the I-V by setting the wiper on the digital potentiometer. The I-V output is given by Equation (3.6).

$$I_{IN} = (V_A - V_W)/R_{AW} \tag{3.6}$$

The reference $V_A \approx V_{REF} - V_{AMP\_OFFSET}$ and V_{AMP_OFFSET} is the amplifier input offset, which is sampled during an initial calibration step. V_{REF} is an adjustable offset

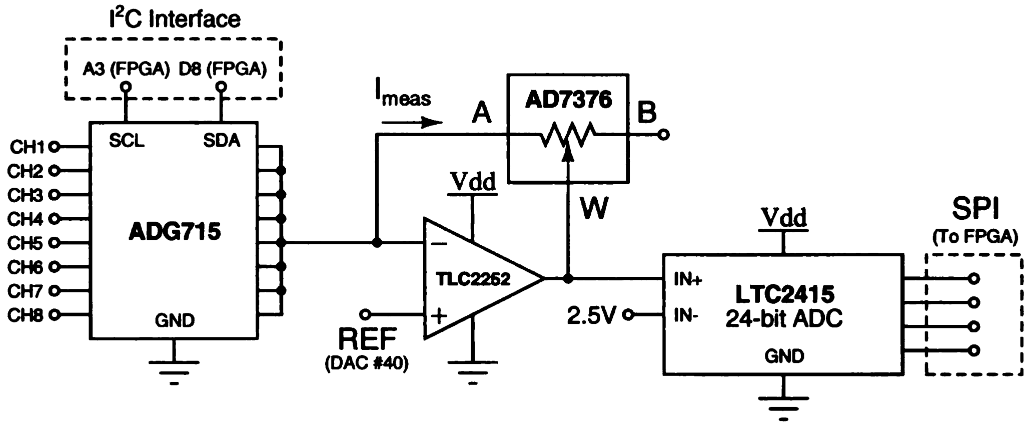


Figure 3.14. Current Measurement Circuit

and is set by the 40th channel of the voltage DACs. Its default value is set to 1 V during system startup. The value of R_{AW} is iteratively chosen to maximize the potential across V_{AW}, which helps to minimize the measurement error caused by the series resistance R_W internal to the potentiometer. The output V_W is measured directly by the ADC and the current computed in Equation (3.6) is handled in software.

Figure 3.15 shows the DC response of the I-V converter. For the simulation, the AD7376 was replaced with a 2.2 MΩ resistor, enabling resolution of sub-nanoampere currents, a modification that has also been made to the board for testing the floating gate transistors.

Figure 3.16 shows the stability of the current ADC's I-V converter by simulating its step response. This I-V circuit was initially designed with the OPA2743 that was used previously in the injection and current DAC circuit, but was shown to cause stability problems both in testing and in simulation. A beta network analysis also reveals the OPA2743 to be marginally stable in this configuration.

Due to small source currents, the input impedance is very large, and β approaches one as given by the following equation where R_1 is the input impedance to the I-V converter and R_2 is the feedback resistance.

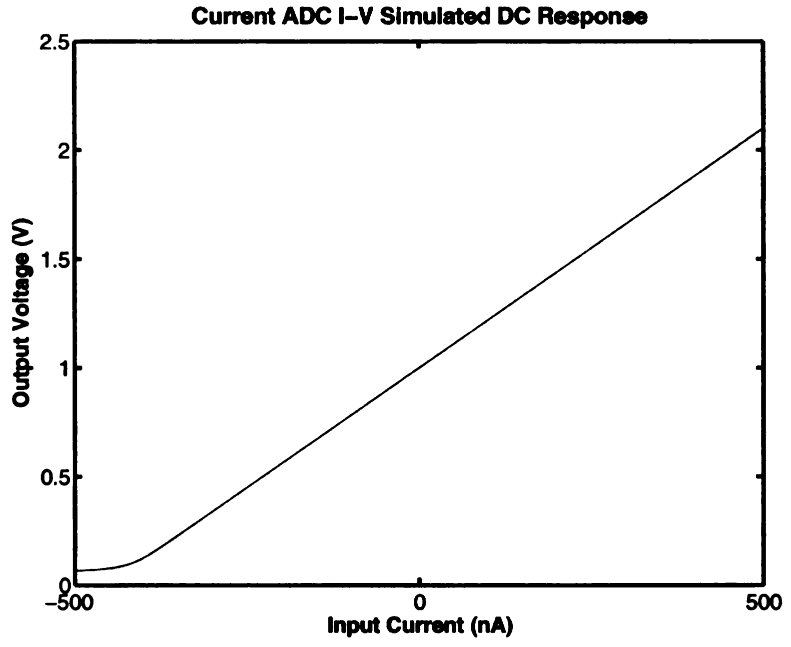


Figure 3.15. Current ADC I-V Converter DC Response

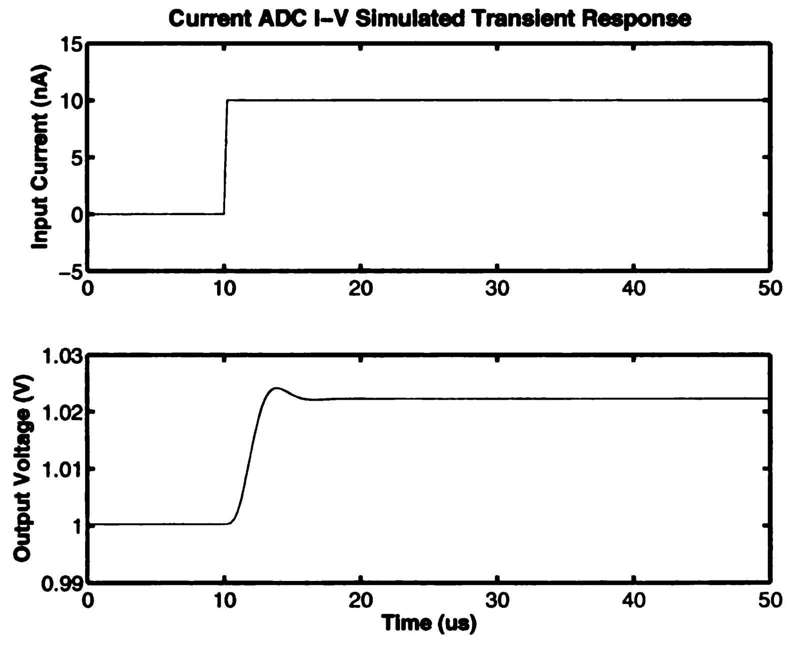


Figure 3.16. Current ADC I-V Converter Transient Response

$$\beta = \frac{R_1}{R_1 + R_2} = \frac{\frac{1}{I_{IN}}}{\frac{1}{I_{IN}} + 2.2M\Omega} \tag{3.7}$$

As I_{IN} is small, $\beta \rightarrow 1$. A plot of the open loop gain and phase versus frequency in the OPA2743 datasheet shows that the phase margin is less than 20° when the open loop gain crosses unity. Although marginally stable, a SPICE simulation of the step response showed oscillations. However, the TLC2252 has approximately a 30° phase margin at the unity gain crossover frequency and is shown to be stable with a settling time of approximately 6 μs.

3.8 Testing Considerations

Each of the previously described circuits are globally enabled through a series of jumpers, which are described in Table 3.1. These jumpers aid in the initial population and testing phase, and act as power-on resets during control logic testing. These jumpers also enable supply current measurement for each sub-circuit.

Designator	Description
JP0	DGND to AGND Inductor Bypass
JP1	Unregulated Supply
JP2	FPGA Supply
JP3	5 V Regulator Output
JP4	3.3 V Regulator Output
JP5	Current ADC Supply
JP6	Current DAC Supply
JP7	-5 V DC/DC Converter Input Supply
JP8	Boost Converter Input Supply

Table 3.1. Jumper Descriptions

3.9 Summary

This chapter covered the design and implementation of the floating gate mixed-signal test station. To obtain high precision, both noise and shielding issues have been addressed through layout and part selection. Additionally, reference calibration through manual potentiometer adjustment and offset sampling techniques have been implemented to improve converter accuracy. These details will become apparent when trying to resolve sub-nanoampere currents during the floating gate programming procedure. Figure 3.17 shows a picture of the completed test station motherboard.

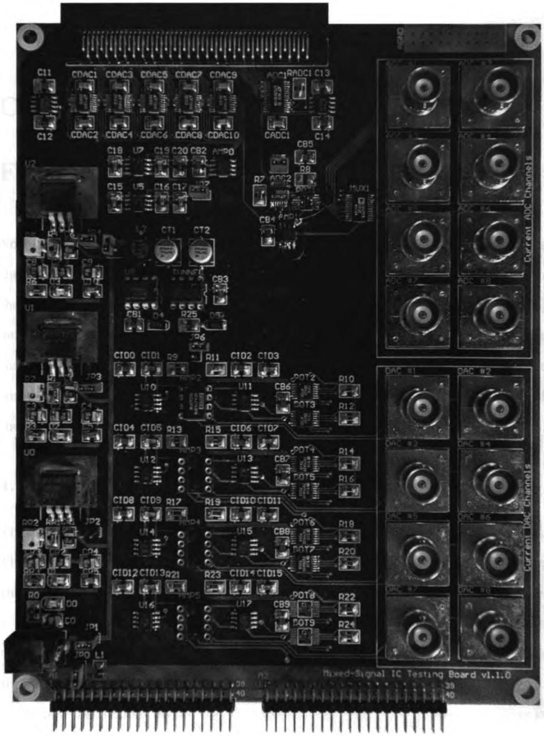


Figure 3.17. Mixed-Signal Test Station Populated Printed Circuit Board

CHAPTER 4

Floating Gate Test Station Control

Now that the system architecture and circuits have been well-defined in the previous chapter, a control logic is necessary to handle high-speed digital communication with the motherboard. The test station is controlled via a Xilinx XC3S200 Spartan-3 [56] field-programmable gate array. A commercially-available development board [57] for this FPGA is directly connected to the floating gate test station motherboard. The control logic has been written in VHDL and synthesized for direct representation of the FPGA's firmware as a bitstream file and is stored on an adjacent EEPROM chip.

4.1 Instruction Decoding and Execution

The test station's main controller is shown in block diagram form in Figure 4.2. This main module is responsible for handling system startup as well as enabling and disabling each sub-module based on the incoming instruction. Each instruction is sent serially over the RS-232 interface as a 16-bit packet and is stored in the *current_instruction* variable. A global instruction pointer is then incremented, which allows the module's state machine to recognize incoming instructions.

As new instructions arrive, they are immediately decoded and executed. There is no FIFO queue for instructions, so the main controller must complete execution of the previous instruction before beginning the next instruction. Although a limitation of

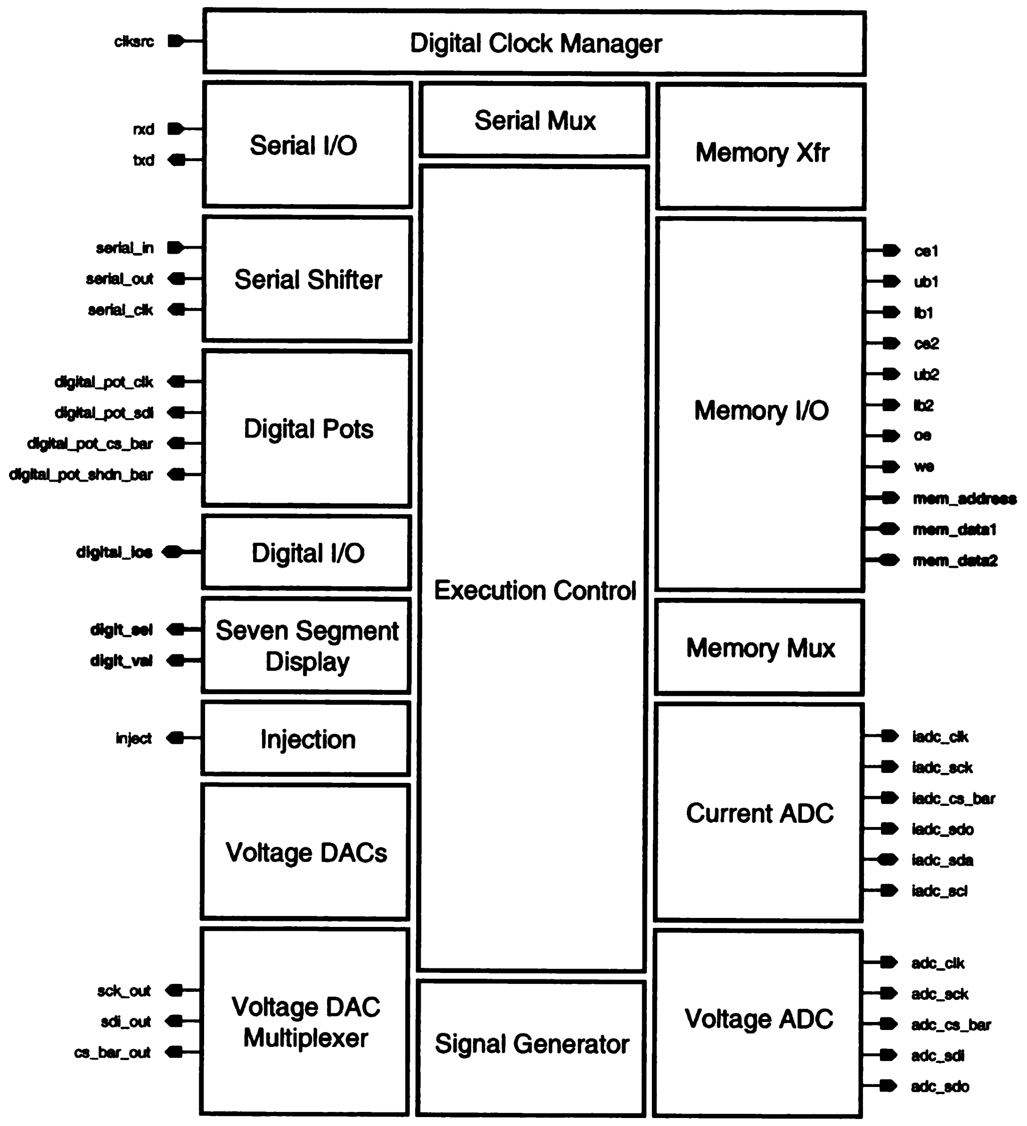


Figure 4.1. System Controller Block Diagram

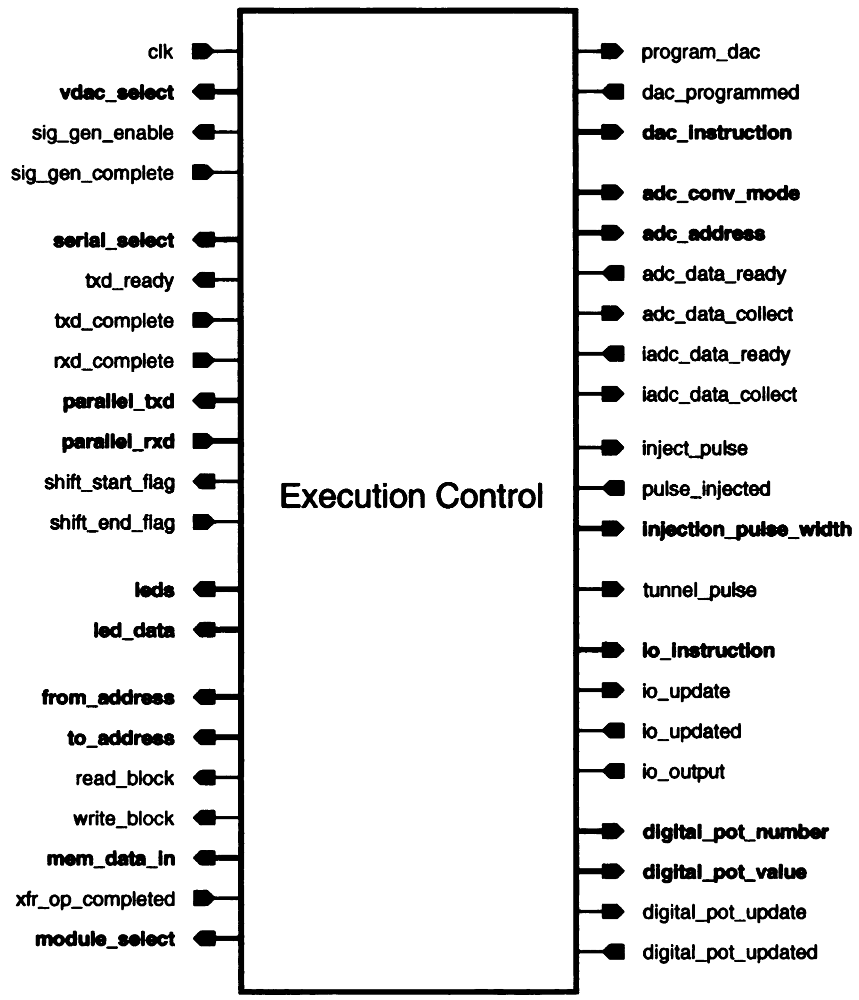


Figure 4.2. The Instruction Decoding and Execution VHDL Module

the instruction decoding architecture, this does not pose a great risk to the operation of the system due to low overhead and latency of instruction execution. Additionally, the PC-based software interface always waits for an acknowledgment packet before sending additional instructions.

During the idle state, each module is disabled and the three system multiplexers are put in their neutral states. Upon receiving a new instruction, the instruction pointer increments, which is identified in the idle state and instruction decoding begins.

4.1.1 Instruction Set

The system uses the four most significant bits of the first packet to decode the instruction. Table 4.1 details the instructions available and their basic function.

Code	Description
0000	Loopback
0001	Memory Transfer
0010	Voltage Digital-to-Analog Conversion
0011	Voltage Analog-to-Digital Conversion
0100	Current Digital-to-Analog Conversion
0101	Current Analog-to-Digital Conversion
0110	Floating-Gate Transistor Injection
0111	Floating-Gate Transistor Tunneling
1000	Digital I/O
1001	Signal Generation
1010	Serial Shifter

Table 4.1. System Instruction Set

The system then jumps to the appropriate state based on the properly decoded instructions. Any illegal instruction is ignored and the controller enters the idle state on the rising edge of the next clock pulse.

Some instructions are longer than one, 16-bit word and require subsequent packets to be sent and decoded. An example of this is any operation that requires memory access. The FPGA development board includes 512 kB of SRAM available through an external chip. These memory addresses are 19 bits wide and block memory transfers require two addresses to be sent: one for the starting address and one for the ending address. Additionally, if data is to be written to the on-board memory, an additional packet is required. This structure is handled automatically by the internal states of the system controller. Since the instruction packet is incremented whenever a new instruction arrives, the system controller expects additional packets whenever a multi-packet instruction is decoded. Additionally, during each of these states, the *inst_pointer_last* variable is updated in order to distinguish changes on the main instruction pointer.

After an instruction is executed, an acknowledgment word is sent back to the PC so it may resume its own script execution. This is handled by sending the controller into the loopback state (state 3). The data stored in the *current_instruction* variable is transferred to the serial I/O module for data transfer. This variable is reset to zero prior to this operation to ensure data integrity.

Next the system controller returns to the idle state and sets the instruction pointer and its comparison register *inst_pointer_last* to zero. This is necessary to prevent the instruction pointer register from rolling over after receiving a large number of instructions. Since the controller can only execute one instruction at a time, the value of the instruction pointer does not have any effect on future instructions.

4.2 Digital Input/Output

The board includes a 13-channel digital I/O interface between the PC and the device under test. Figure 4.3 shows the block diagram of this module.

The signal *digital_ios* is a 13-bit wide input/output bus that connects directly to the FPGA's I/O pins. All other signals are connected to the instruction execution module. The *instruction* signal is a 6-bit wide bus that contains the channel to update

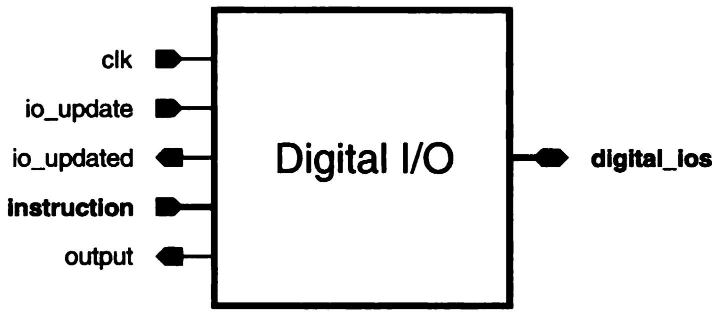


Figure 4.3. Digital I/O VHDL Module

(stored in the four least-significant bits) as well as the direction of the data flow and any data that must be written to the digital line.

Two internal state machines control the current state of the module and the state of the *digital_ios* signal. During system startup, all channels are set to a high impedance state. Next, the module sits in an idle state and waits to receive the update flag when the *io_update* signal is high on the rising edge of the FPGA clock. Then, the channel to update is decoded from the *instruction* signal and it is set to a high impedance state for one clock cycle. If the most-significant bit of the *instruction* signal is high, the channel enters the writing state, otherwise it enters the reading state. If the channel enters the reading state, its data is latched to the *output* signal's register where it remains until it is overwritten during the idle state. Finally, the *io_updated* flag is set to high for a duration of one clock pulse for the system controller to acknowledge that the I/O operation has completed. The module then returns to the idle state and resets the acknowledgment flag.

4.3 Digital Potentiometer Control

The board contains nine digital potentiometers that are daisy-chained together for control via a three-wire serial peripheral interface. Each potentiometer is seven-bits, which implies a 63-bit serial chain to program all potentiometers simultaneously.

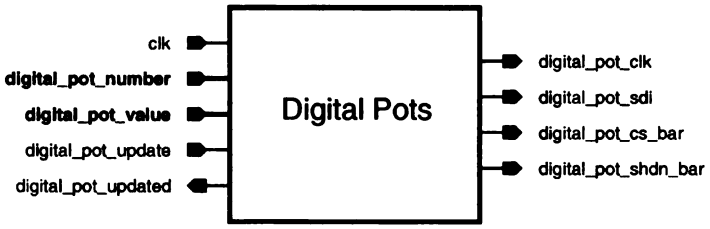


Figure 4.4. Digital Potentiometer VHDL Module

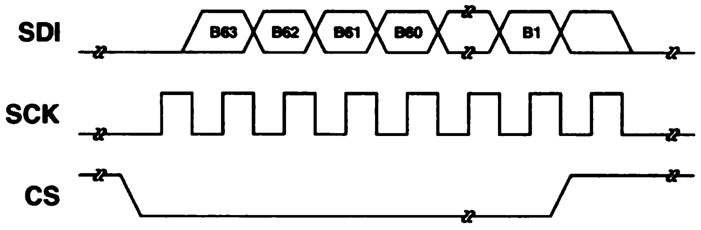


Figure 4.5. Digital Potentiometer Serial Peripheral Interface

Figure 4.4 shows the signal assignments for the digital potentiometer controller. The input signals from the execution controller include the number of the device in the daisy chain, the value to store in the device, and an update flag for when these registers are ready to be sampled. The clock, serial data input, chip select, and shutdown lines connect directly to the digital potentiometer.

Upon receiving the update flag, the module decodes the device number and goes into serial shifting mode. During the shifting operation, the serial chain is rotated in a 63-bit register to retain the values previously updated in the other devices. The shifting operation occurs at 390 kHz, allowing an update frequency of 6.2 kHz. After the shifting operation completes, an acknowledgment flag is set to high for one clock pulse before returning to the idle state.

Since the digital potentiometers are used for both the current analog-to-digital converter as well as the current digital-to-analog converter, the update process is controlled globally by the execution controller. This eliminates the requirement of an

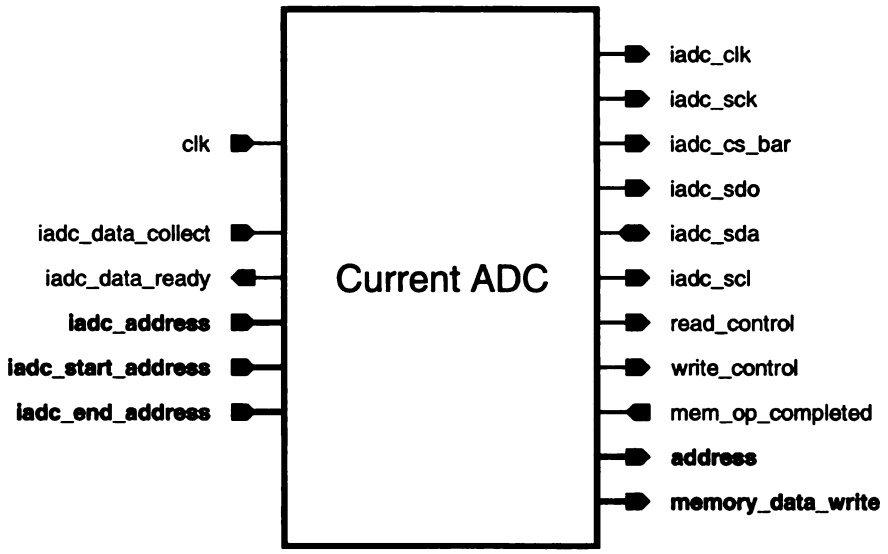


Figure 4.6. Current ADC VHDL Module

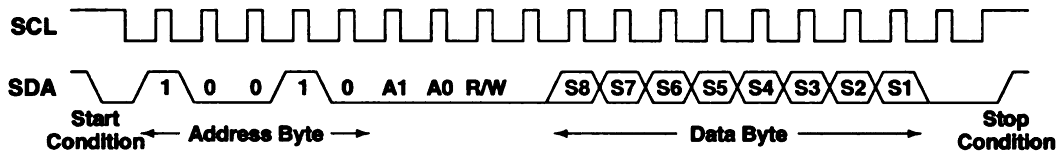


Figure 4.7. Current Channel Multiplexer I^{2}C Interface

additional multiplexer for the potentiometer control signals from both the execution controller and the current ADC.

4.4 Current Analog-to-Digital Conversion

The board incorporates a 24-bit delta-sigma voltage mode analog-to-digital converter in conjunction with a 128-position digital potentiometer, an eight-channel analog multiplexer and operational amplifier to make up the current ADC module. This FPGA module controls the three interfaces between the ADC, potentiometer, and multiplexer to resolve currents in the nanoampere range.

From a data acquisition perspective, the channel number, gain and number of

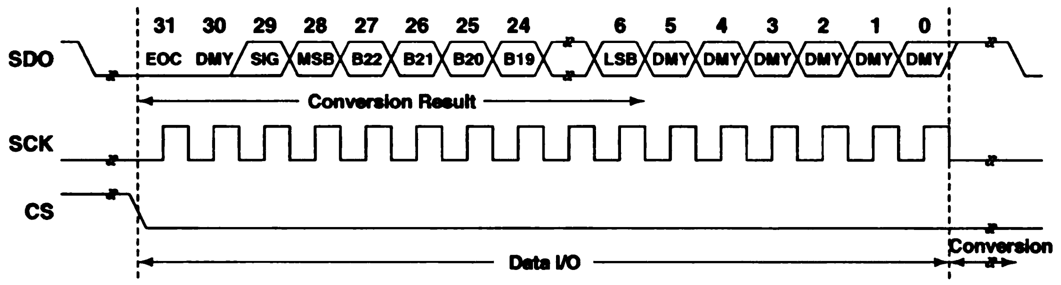


Figure 4.8. Current ADC Serial Peripheral Interface

samples are specified. These are provided by four, 16-bit packets to the execution controller, which then slices them into the starting address, ending address, channel address, and gain. The memory addresses are required because the data is first stored into the FPGA board's RAM modules before being transferred over the communication bus, acting as a buffer to allow real-time data acquisition. Data is ultimately transferred over the serial line via the block memory transfer module. The three most-significant bits of the memory addresses as well as the ADC channel number are obtained from the first instruction packet.

The gain update is provided by the seven least-significant bits of the fourth instruction packet. Gain update only occurs when the most-significant bit of this packet is set to '1', otherwise no gain update occurs. This reduces the overhead and improves the performance of the ADC module.

During startup, the module initializes all of the internal variables and external SPI and I^{2}C signals, and waits in the idle state until the *iadc_data_collect* signal is high. Next, it initializes the multiplexer serial chain and enters shifting mode to update the analog multiplexer. The analog multiplexer uses an I^{2}C interface, however only the writing mode of the protocol has been implemented.

After the multiplexer is updated, the single-channel, 24-bit ADC is re-initialized by shifting out the previous conversion. This is completed since the ADC begins conversion immediately following the last bit during the serial shifting operation, thus the existing data in the ADC is from a previous data collection and should be ignored.

After shifting out the previous conversion, the ADC goes into a waiting state for the serial output line to transition from high to low. This transition signifies data conversion is complete. The data is shifted into a 32-bit register where it is sliced into two 16-bit words for storage into two sequential memory cells in the FPGA development board's RAM. After writing to the memory, the current memory address is incremented and the process repeats for however many samples are specified.

The data may be stored anywhere in memory thanks to the *iadc_start_address* and *iadc_end_address* input signals. This allows the flexibility of filling the entire memory with samples if necessary or allowing space for other modules.

After reaching the ending addresses, the module sets the *iadc_data_ready* flag to high for one clock cycle and then returns to the idle state.

4.5 Injection Control

The floating gate transistors are written to using the hot-electron injection process discussed in Chapter 2. The circuit that controls this process requires one digital output from the FPGA to enable/disable injection. However, the injection control module uses a logarithmic-based injection scheme to achieve very small injection pulses on the order of tens of nanoseconds to approximately ten seconds.

This is achieved by the use two 30-bit registers. First, a comparison register is set to all zeros with its least significant bit set to '1'. The signal *injection_pulse_width* shown in Figure 4.9 is provided by the execution controller and is an 8-bit integer representing the pulse width given by the following equation

$$y = 20 \cdot 10^{-9} \cdot 2^{x} \tag{4.1}$$

where y represents the output pulse in seconds and x represents the value given

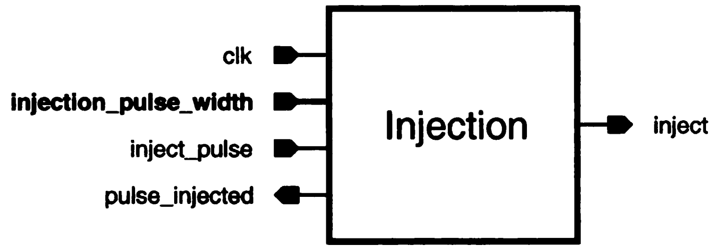


Figure 4.9. Hot-Electron Injection VHDL Module

by *injection_pulse_width*. After initializing the comparison register, it is rotated left x number of times. Although x can take values from 0–255, the number is practically limited to 29, since there are 30 bits in the comparison register. Any values greater than 30 will simply cause the comparison register to make a complete rotation, keeping the injection pulse width at a maximum of 10.73 seconds.

After setup of the comparison register, the injection pulse goes high and the module enters a counting mode where it increments the 30-bit *injection_register* variable once per clock cycle until it exceeds the value of the comparison register. Thus, $20\log(2^{29})$ or approximately 174.5 dB of dynamic range is achieved using this method.

Following the injection pulse, an acknowledgment flag is set high for the duration of one clock cycle to allow the execution controller to resume operation and return to the idle state.

4.6 Memory I/O

The system uses the FPGA development board's SRAM chips to buffer all data acquired from the test platform, as well as configuration data for the serial shifting module and signal generator. It is therefore essential to have a robust interface to the physical memory cells. The Memory I/O controller shown in Figure 4.10 interfaces with the physical control pins on the two RAM modules as well as a 5:1 memory multiplexer to allow multiple modules direct access to the memory without passing

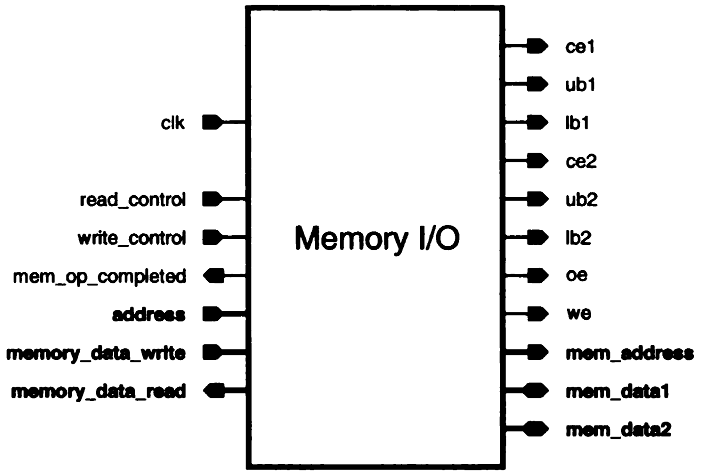


Figure 4.10. Memory I/O VHDL Module

data through the execution controller.

During the idle state, the two input control flags *read_control* and *write_control* determine how the RAM cells are enabled. The read mode always has a higher precedence than the write mode. Thus, if both signals are high, the memory controller will read the data from the current address and latch it to *memory_data_read.* During the read mode, the active low write enable signal is set to high and the output enable signal is low. The dual is true during the write mode. The RAM chips themselves are each 256 kB and thus have an 18-bit addressing scheme. Internally, however, these two chips are addressed in a 19-bit addressing scheme where the 19th bit acts as the upper and lower byte enable as well as the chip enable.

Each RAM chip shares the same address bus, but has separate data buses. The data buses are bidirectional, meaning they must be set to a high impedance state when not in use. When both input control flags are set to logic low, the memory address is set to zero, both output and write control flags are disabled, and the memories are disabled.

After receiving a transition on one of the two input control flags, the internal

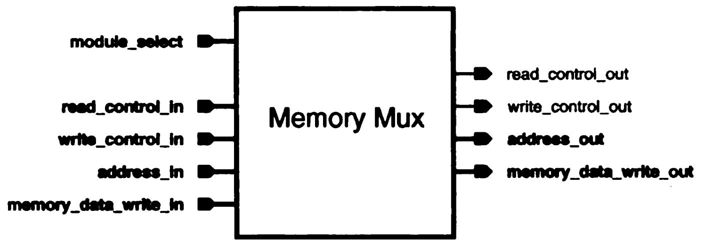


Figure 4.11. Memory Multiplexer VHDL Module

state machine goes into the read or write state. In the read state, the data latched onto the *memory_data_read* signal by the two data buses is determined by the most-significant bit of the memory address. Likewise, during the write state, this bit also determines the data put onto the bus from the *memory_data_write* register. Finally, the *mem_op_completed* acknowledgment flag is enabled to return control to the parent module before returning to the idle state.

4.7 Memory Multiplexer

The Memory I/O controller discussed in Section 4.6 interfaces with a 5:1 multiplexer to allow the memory transfer, current ADC, voltage ADC, signal generator, and serial shifter to gain direct access to memory without passing data indirectly through the execution controller. Figure 4.11 shows the signals passed through the memory multiplexer.

4.8 Memory Transfer Control

Although the Memory I/O controller handles low-level access to the RAM, all data transferred to and from the board is handled by the Memory Transfer Control module. It also interfaces directly with the serial I/O module for direct outbound data transfer. Figure 4.12 shows a block diagram of this module and its associated pins, where all

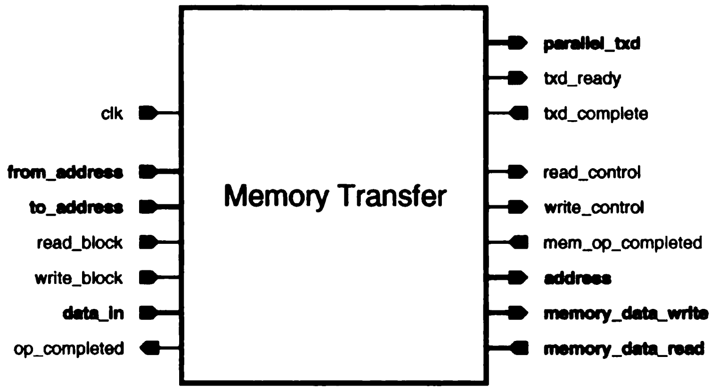


Figure 4.12. Memory Transfer VHDL Module

pins on the left-hand side connect to the execution controller.

This module contains two internal state machines, one for block memory reads and the other for writes. The memory transfer control module is also connected to the memory multiplexer. Therefore, the execution controller must select the second input channel, otherwise the Memory Transfer Controller will never receive an update acknowledgment from the Memory I/O module.

Just as the *read_control* and *write_control* flags initiate the memory I/O controller to complete an operation, the Memory Transfer module uses the *read_block* and *write_block* control flags to begin a block memory transfer. These signals can never be high simultaneously, otherwise both state machines will break out of their idle states and will attempt to access the same memory signals at the same time. This condition is prevented by the execution controller, which ensures both signals are logic low during the idle state and only go high based on the 6th most-significant bit of a memory transfer instruction packet.

On the execution controller side, the first instruction packet contains the read or write operation flag, as well as the three most-significant bits of the address registers. Two additional states collect the starting and ending addresses for reading or writing. If a writing operation is required, a fourth packet is sent. The fourth data packet may be written to one address or a range of addresses. This is used to initialize memory cells, or for verification purposes as seen in the system initialization script.

Initially, there were two instructions to handle memory transfer. However, since a block data transfer mode was required, a more elegant solution was to include one extra instruction packet for every transfer, providing an address range. If the transfer included a single read or write, both addresses would be the same. The expectation, however, is that most data transfers will be block transfers, which is the case for analog-to-digital conversion and serial shift chain verification. This approach reduces the need for additional overhead in the FPGA to handle single address reads and writes, as well as reduces the need for an additional op code in the instruction set.

When either of the two state machines break out of the idle state, they iterate through the range of addresses provided by the *from_address* and *to_address* signals. It uses a temporary *current_address* variable for comparison and increments after each read or write. During a memory read operation, the module may access the serial I/O directly given the execution controller has enabled the serial I/O multiplexer accordingly. Upon completion of a block data transfer, the *op_completed* signal is set to logic high for one clock pulse before returning to the idle state.

4.9 Voltage Digital-to-Analog Conversion

The voltage DACs consist of five, eight-channel daisy-chained devices programmed using a serial peripheral interface at a frequency of 25 MHz. Each device contains a 32-bit register used to store a command op code, DAC channel, and 16-bit value representing the voltage output. Thus, the module requires an internal 160-bit serial chain for device configuration. Figure 4.13 shows a block diagram of the DAC control module.

Upon DAC update, the execution controller receives two packets from the serial communications interface. The first packet contains the upper 12 bits of the

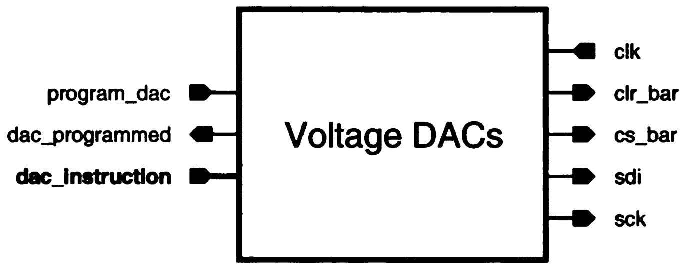


Figure 4.13. Voltage DAC VHDL Module

dac_instruction signal and the second packet contains the remaining 16 bits. The upper four bits of this signal contain the device address, which is parsed after the *program_dac* signal becomes a logic high and initiates a DAC update. The device address determines which 32 bits in the internal 160-bit shift chain register are updated prior to the shifting operation. The remaining 24 bits in the *dac_instruction* register are written directly to the shift chain register based on the device address. Furthermore, the upper eight bits of the instruction register contain four bits representing the DAC command code followed by four bits representing the DAC channel.

The command code has multiple configurations, but only the write and update feature is used. For example, the DAC channels may have their internal registers written to, but not their outputs updated, or may have their outputs disabled entirely. This feature may be useful, for instance, if all channels need to be updated simultaneously. Given that the *dac_instruction* is passed directly from the software interface, such behavior is user configurable.

After updating the serial chain, the module enters the shifting mode. Since this module updates only one device per serial shift cycle, all other devices receive a command code of no operation (0xF). Finally, the *dac_programmed* signal is set to logic high to acknowledge the DAC operation is complete before returning to the idle state.

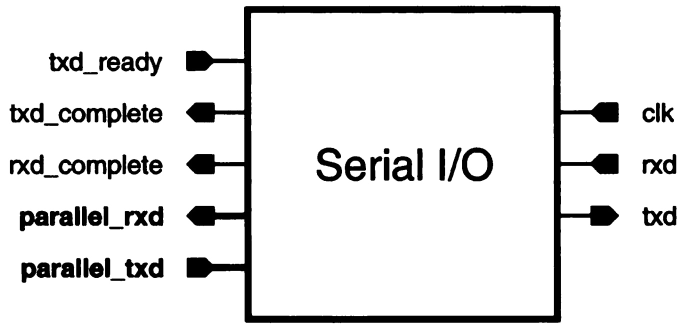


Figure 4.14. Serial I/O VHDL Module

4.10 Serial I/O

The serial communications module allows data transfer between the PC-controlled software interface and the FPGA development board. The protocol used is the standard asynchronous serial communications found in typical RS-232 implementations. The hardware used is the minimal 3-wire RS-232, which contains only transmission, receive, and ground connections. Figure 4.14 shows the Serial I/O module and its associated pins.

Figure 4.15 shows one byte of asynchronous data using RS-232 signal levels. From this figure, it can be seen that the signal levels use a negative logic. There is a voltage translator that resides between the interface cable and the FPGA to convert between these signal levels and 0–3.3 V levels, as well as converting the signals to positive logic.

The line sits idle at the mark voltage level. There is an undefined region between positive and negative 3 V to eliminate invalid start bit detection, particularly when the cable is unplugged. When a transition from high to low occurs (positive logic), the start bit is sent, followed by eight data bits and a specified number of parity and stop bits. The figure also illustrates mark parity and two stop bits, which has been defined for this system. Additional packets may be sent on the next clock cycle

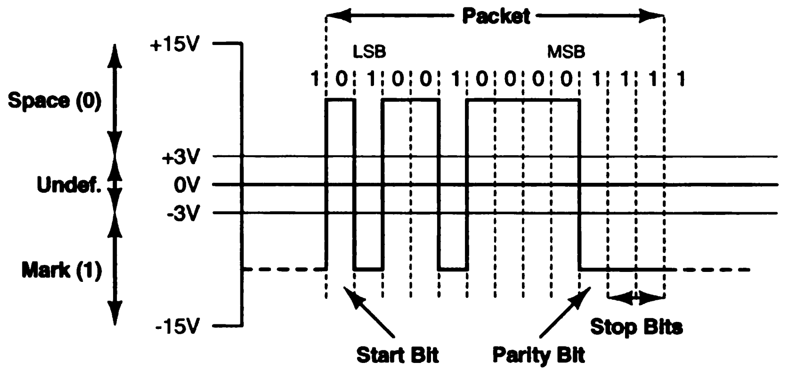


Figure 4.15. RS-232 Signal Timing Diagram

following the last stop bit.

This transceiver contains two registers for storing both received data and data to transmit, *parallel_txd* and *parallel_rxd*. It also has acknowledgment signals for these two modes whenever a serial to parallel or parallel to serial conversion is complete. Finally, in the transmit mode, the transceiver has an additional control flag whenever data stored in the *parallel_txd* register is ready to be sent serially.

The module operates at a baud rate of 115200 bps, which is the maximum sustainable frequency of the data terminal equipment (DTE). To set the baud clock, the FPGA's 50 MHz external clock is divided using a counter. There are two baud rate generators depending on the direction of data being transferred. The transmission baud rate generator may run continuously because it is the responsibility of the DTE to detect the start bit and to synchronize its own baud rate clock. However, during receiving, the baud rate clock is synchronized during start bit detection, and is set to a logic low otherwise.

A standard packet size is typically eight bits (although seven bits is often found in legacy devices) for this type of serial communications. However, due to the 16-bit word widths of the RAM chips and data from the voltage DACs, it became natural to implement a 16-bit packet structure for serial communications. This is accomplished by sending two, 8-bit packets sequentially and then reconstructing the data as a single 16-bit word following transmission on both ends. A state machine handles each bit transmitted as well as the start, stop, and parity bits. During each state of the transmit operation, another state machine handles the shifting of the *parallel_txd* register.

On the receive side, the input signal is buffered to prevent glitches. This is accomplished by incrementing or decrementing a counter based on the current sample of the input signal. The counter is initially set to 50 and the output is set to logic '1', which also corresponds to the idle state of the serial protocol. If the input signal goes low, the counter will begin to decrement until it reaches zero for every '0' it samples per clock cycle or will increment its value by one for every '1' it samples. After at least fifty '0' samples, the output is set to logic '0'. Therefore, by utilizing this method, any spurious logic highs during which time the input should be logic low will be avoided. This specifically prevents false triggering of the start bit.

As in the transmit mode, the receiver has its own state machine to iterate through the received bits during the serial to parallel conversion. This state machine is synchronized on the start bit, which is accomplished by oversampling the filtered serial receiver input. Once a high to low transition is detected, a counter increments until it reaches half of the desired clock cycle length. A flag then instructs the receiver state machine to begin collecting each data bit. The baud rate generator counts 434 clock cycles per bit and sets the *get_next_bit* signal high for one clock cycle before resetting the counter. After receiving the last data bit of the second packet, the receiver state machine returns to the idle state.

Since mark parity and two stop bits are used, the next three bits following the data bits during serial transmission are logic '1', or the same as the RXD line's idle state. This is designed to give the data terminal equipment extra time between packets. However, in the receiver, only one stop bit and no parity bits are recognized.

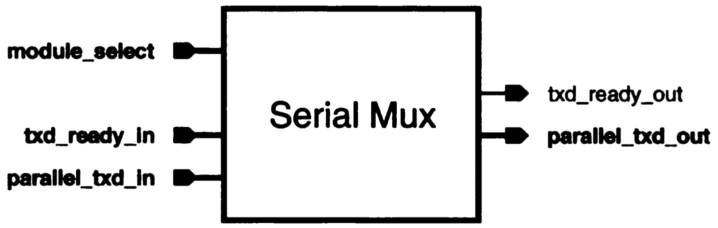


Figure 4.16. Serial Multiplexer VHDL Module

Therefore, higher transmission rates may be achieved by configuring the data terminal equipment accordingly.

4.11 Serial Multiplexer

Two modules require access to the serial I/O module, the execution controller and the memory transfer module. Figure 4.16 shows the serial multiplexer that allows each of these modules to gain control of the *parallel_txd* and *txd_ready* signals. This multiplexer may also directly disable serial I/O transmission by setting the *module_select* signal to zero.

4.12 Serial Shifter

A general-purpose serial shift chain controller has been included for the purpose of testing mixed-signal designs with D flip-flop-based shift registers. Such shift registers are used to enable and disable portions of the chip to aid in the testing process. However, the length of these chains can vary depending on design and application, but they all have the same basic structure and control pins.

Figure 4.17 shows the block diagram for the serial shift chain VHDL module. Three external pins are needed for serial shifting: *serial_clk*, *serial_in*, and *serial_out*. The other signals and buses are required for handshaking and for loading and storing data from the memory.

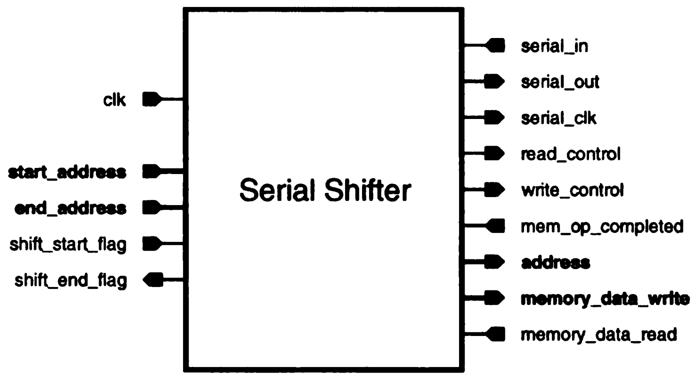


Figure 4.17. Serial Shifter VHDL Module

The controller works by reading the bitstream from sequential memory locations and shifting the data out one bit per serial clock cycle. Additionally, the controller will read the data output from the opposite end of the serial shift chain if available, and will store it in subsequent memory locations.

Setup requires writing '1's and '0's to the least-significant bits of sequential memory cells in any portion of the SRAM. This is accomplished by using the memory transfer module and writing '1's and '0's to each address. The addresses which this data spans is given by the *start_address* and *end_address* registers.

During the idle state, the end shifting flag and serial out pin are cleared. During a shifting operation, the execution controller configures the memory multiplexer to allow the serial shifter module direct access to the SRAM. Three packets are received from the PC-based host containing the instruction op code and the start and end memory locations where the serial chain data is configured.

The current address is set to the start address and the cell count is determined by subtracting the start address from the end address. The host controller ensures that the end address is always greater than the start address, and that the selected memory cells are not less than *cell_count* locations away from the end of memory. Such a condition will cause the data written back as read from *serial_in* to overflow

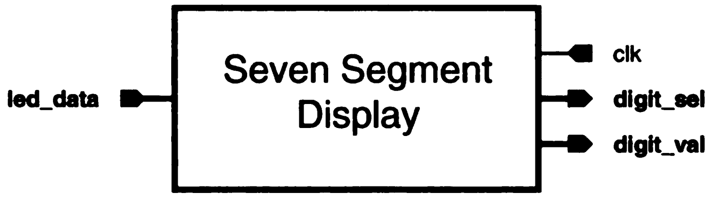


Figure 4.18. Seven Segment Display VHDL Module

the memory space.

The controller then iterates through each memory location until the current address is greater than the end address. Next, the *shift_end_flag* is set for one clock cycle and the controller returns to the idle state.

Data read from the *serial_in* pin will be written to the next memory location following the last cell containing data to add to the serial chain. Thus, if memory cells 1–500 contain data for shifting, the data written back will be contained in memory cells 501–1000.

4.13 Seven Segment Display

A seven segment display decoder is used for debugging purposes to view the internal states of the FPGA. It is set by default to display the value of the *current_instruction* register. Figure 4.18 shows the block diagram for the control module.

The theory for updating the seven segment display is as follows. Each display module has a parallel interface to control which of the seven LEDs are enabled. Additionally, each module has its own enable pin, making a four-bit address bus to control which display module is to be updated. Only one seven-segment module may be enabled at a time. The controller iterates through the four display modules and turns each on individually to set the display segments. Although one digit is on at any given time, the human eye perceives the update of the display as continuous.

This module uses four internal processes. A clock divider is used to reduce the

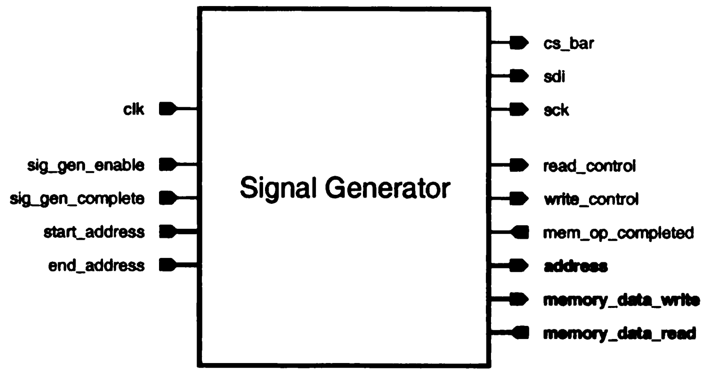


Figure 4.19. Signal Generator VHDL Module

update frequency of the display to 250 kHz. An LED selection state machine decodes an integer-based selection to one that enables only one seven segment module at a time. A process to convert the value of a four-bit register into the enable signals for the seven segments is needed to display the characters properly. Finally, the main process deconstructs a 16-bit hexadecimal number and sets the other processes accordingly to enable the correct seven segment module and its enabled segments.

4.14 Signal Generator

A signal generator module combines the functionality of the voltage digital-to-analog conversion module with the memory transfer module to create a multi-channel waveform generator that may be used for testing mixed-signal designs. A block diagram of this module is given in Figure 4.19.

A review of Section 4.9 shows that the DACs are controlled with a SPI through a 160-bit serial shift chain register. By storing 16-bit codes in memory, the DACs may be programmed by reading these memory locations sequentially and updating their outputs accordingly. Therefore, by computing a vector of digital inputs, an analog waveform may be realized at any frequency, offset, amplitude, and phase within the constraints of the digitally programmable interface and output range.

The execution controller initializes the DAC multiplexer and memory I/O multiplexer while waiting for the second instruction packet. The second and third instruction packets contain the start and end addresses for the memory locations containing the DAC codes. The first five memory locations in the specified memory range contain the channel addresses of the five DAC ICs. Therefore, for multi-channel signal generation, up to five channels may be specified with one channel per DAC chip.

This behavior is due to the 160-bit serial chain updating the five chips concurrently. Since only one channel per chip may be updated at a time, only one channel is allowed per chip during multi-channel signal generation. Additionally, since the last four channels of the fifth DAC chip provide biases to the injection and current ADC circuits, these channels may not be specified in the fifth memory location in the address range due to their required stability as a bias.

After receiving the ending address, the module sets the *sig_gen_enable* flag and and enters a waiting state until a termination packet is received. This termination packet is a single instruction packet and is implemented by the host controller using the loopback command. This is the only means of disabling signal generation and returning the execution controller to the idle state.

During the idle state, the DAC serial chain register is initialized to the no operation state for all five devices. Upon receiving the *sig_gen_enable* flag, the controller initializes the channel addresses for the five chips by reading the first five memory locations specified in the address range. Next, the data is loaded from the remaining addresses sequentially and is stored into the DAC serial chain register.

This processes occurs in blocks of five memory addresses per shift cycle. After loading the data into the serial shift chain register, the data is output via the serial peripheral interface at a frequency of 25 MHz. The module then reads the next five locations and stores them into the serial shift chain and repeats the process. This occurs until the end address is reached, after which the current address is reset to

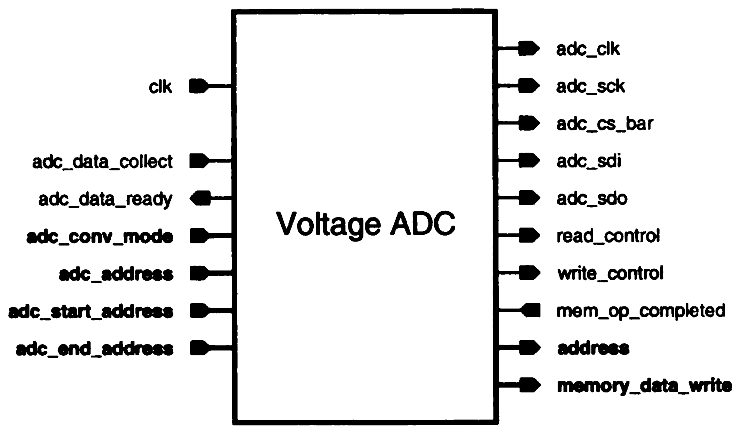


Figure 4.20. Voltage ADC VHDL Module

the beginning of the data loadable addresses. It is also during this state that the controller determines if the *sig_gen_enable* flag has been cleared, meaning that the termination packet has been received. The *sig_gen_complete* acknowledgment signal is set for one clock pulse and the module enters the idle state.

Thus, to construct multi-channel signal generation, the data for a single channel is stored in every fifth memory location. Routines have been devised in software to generate these data structures and store them in memory. An example function is given in Section E.17.

4.15 Voltage Analog-to-Digital Conversion

The board includes a 16-channel, 24-bit delta-sigma voltage mode analog-to-digital converter for data acquisition. This module controls the SPI communication protocol that both configures and receives the conversion result, as well as stores the received data in the FPGA board's SRAM.

Figure 4.20 shows the configuration of the voltage ADC VHDL module. The signals on the left-hand side connect to the execution controller whereas the other signals connect to the memory I/O module and external pins for the SPI and conver-

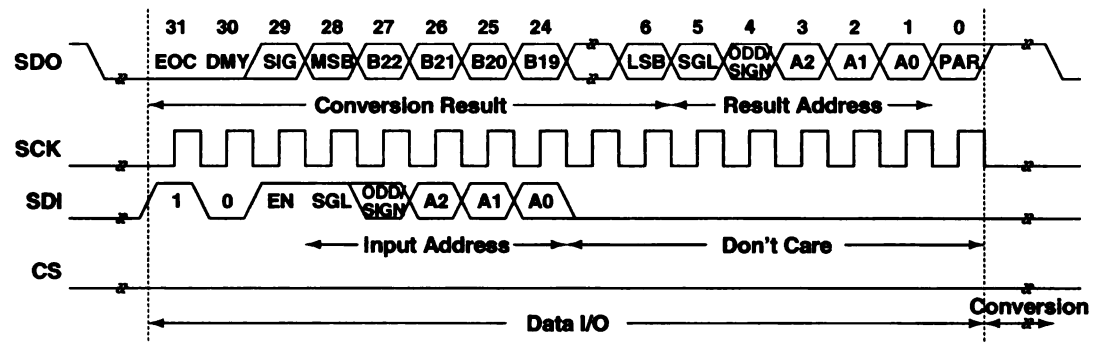


Figure 4.21. Voltage ADC Serial Peripheral Interface

sion clock.

During data acquisition, the channel number, number of samples, and conversion mode are specified. These are provided by three 16-bit packets to the execution controller, and are latched to the *adc_conv_mode*, *adc_address*, *adc_start_address*, and *adc_end_address* registers. As in the current ADC case, the memory addresses allow the data to be stored into the FPGA board's SRAM before being transferred over the communication bus, acting as a buffer to allow continuous data acquisition. Data is then transferred over the serial line via the block memory transfer module. The three most-significant bits of the memory addresses as well as the ADC channel number are obtained from the first instruction packet.

During the idle state, the module initializes all of the internal variables and external SPI signals and waits until the *adc_data_collect* flag is set. Next, it initializes the ADC's serial data input by setting the *adc_chain* register, which specifies the ADC channel address and the conversion clock mode for the next conversion.

The conversion clock mode specifies the speed and accuracy of the data converter. Four modes are selectable based on bits 6–7 of the first instruction packet. The implications of each of these modes is discussed further in Chapter 5, during the testing and validation of the test station's performance.

The module then shifts 32-bit data to and from the ADC, as shown in Figure 4.21.

The serial output read from the ADC corresponds to the previous conversion cycle; therefore, the first conversion after initiating analog-to-digital conversion is ignored. The ADC then goes into a waiting state for the serial output line to transition from high to low. This transition indicates the conversion is complete. The data is shifted into a 32-bit register where it is sliced into two 16-bit words for storage into two sequential memory cells in the FPGA development board's RAM. After writing to the memory, the current memory address is incremented and the process repeats for however many samples are specified.

The conversion clock rate is determined by the *adc_clk* signal. This ADC is capable of a conversion clock rate of 2 MHz, producing an output rate of over 97 samples per second. As in the current ADC module, the data may be stored anywhere in memory due to the use of specified address ranges at the input of the module. This allows the flexibility of filling the entire memory with samples if necessary or allowing space for other modules.

After reaching the ending addresses, the module sets the *adc_data_ready* flag to high for one clock cycle and then returns to the idle state.

4.16 Voltage DAC Multiplexer

The voltage digital-to-analog converters are shared between two modules, the single-channel, single update voltage DAC controller and the signal generator. The voltage DAC multiplexer allows multi-module access to the SPI. Figure 4.22 shows the pin configuration of this 2:1 multiplexer.

4.17 Digital Clock Manager

The Digital Clock Manager (DCM) module allows synthesis of a new clock frequency along with duty cycle and phase correction of existing clock sources [58]. Additionally,

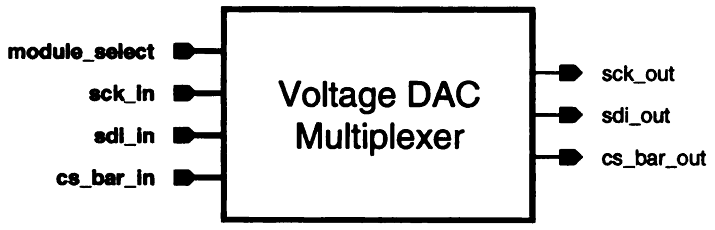


Figure 4.22. Voltage DAC Multiplexer VHDL Module

DCMs help to eliminate clock skew, thus improving performance of the overall system. Figure 4.23 shows the DCM module implemented on FPGA hardware.

The FPGA development board includes a 50 MHz external crystal oscillator that acts as input to the DCM. The DCM can be synthesized for a number of modes, but only buffering and duty cycle correction are used to improve fanout performance. Additionally, a PLL can multiply the clock to frequencies up to 280 MHz for the XC3S200FT256-4 and faster for parts with better speed grades. The DCM shown in Figure 4.23 has three outputs. Signal *clk0_out* shares the same frequency as *clkin_in*, but is buffered and has duty cycle correction. It also acts as the global clock to all other modules. Signal *clkfx_out* is realized as a 250 MHz clock source and *clkin_ibufg_out* is a buffered version of *clkin_in.*

The DCM is also capable of producing 90, 180, and 270 degree phase-shifted clock sources based on the source frequency and 180-degree phase-shifted sources of a doubled source frequency and the freely synthesizable (fx) frequency. Furthermore, it is possible to divide the clock. The DCM is preferable to using traditional clock dividers made with counters and comparators since these structures occupy slices whereas a DCM is a separate module existing in the periphery of the FPGA. This particular device contains eight DCMs.

Utilizing the PLL of the DCM module, it is possible to improve the speed of the overall system, thus reducing the baud rate error in the asynchronous serial communications and doubling the speed of the voltage DACs to their maximum SPI frequency

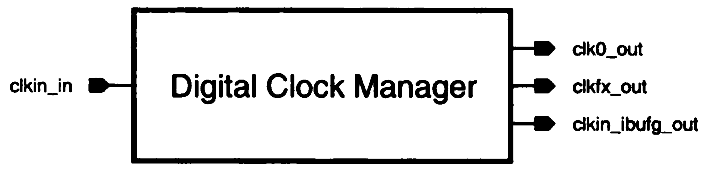


Figure 4.23. Digital Clock Manager Module

of 50 MHz. However, this would require adjusting all internal timing and counter sizes, which is not possible with existing slice utilization. A larger device such as the XC3S1000 would be required for any future expansion of the FPGA's synthesized functionality.

CHAPTER 5

Testing and Results

Before floating gate transistors may be programmed to currents in the nanoampere range, it is essential to test and calibrate the measurement circuits of the mixed-signal test station. This chapter validates the functionality of the voltage-mode ADCs and DACs, current-mode circuits, and floating-gate programming circuits. Next, the floating-gates are tested through a series of programming experiments detailed in Section 5.3.

The system controller detailed in Chapter 4 has been tested throughout its design and will not be discussed. Furthermore, it acts as the framework for the following experiments, which would not be possible without a robust digital interface.

5.1 Test Station Validation

5.1.1 Fowler-Nordheim Tunneling Pulse Response

Figure 5.1 shows the transient response of the gate driver output of the Fowler-Nordheim tunneling circuit. Since tunneling is either enabled or disabled by the least-significant bit of the corresponding instruction packet, the period of the tunneling pulse is defined by the speed of the host software to disable tunneling, which includes the overhead of making the tunneling function call and constructing the packet for serial transfer. A minimum tunneling pulse width of 139 μs is possible by sending

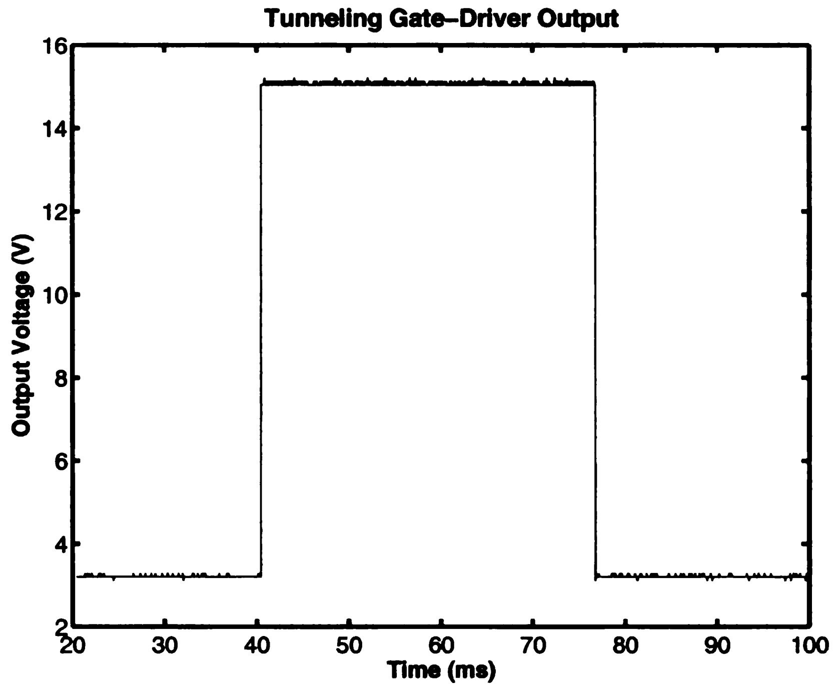


Figure 5.1. Tunneling Pulse

two instruction packets back-to-back.

5.1.2 Hot-Electron Injection Pulse Response

Figure 5.2 shows the minimum pulse width of the injection circuit for different biases of DAC channel #37. Unlike tunneling, which is a relatively slow process on the order of seconds (at 15 V), large injection currents can be generated with pulse widths on the order of microseconds. Thus, a logarithmic pulse scaling method has been implemented as discussed in Section 4.5. The OPA2743, which acts as a rail-to-rail comparator, has a slew rate of 10 V/μs and sets the minimum resolvable pulse width. Thus, the variable x in Equation (4.1) can be set to a minimum of six. Furthermore, it can be seen that the output of the MAX1681 does not have a direct negative correspondence to its input.

Figure 5.3 illustrates the logarithmic scaling of the injection pulse FPGA module for different values of x at a fixed input of 2 V to the MAX1681 voltage inverter. This

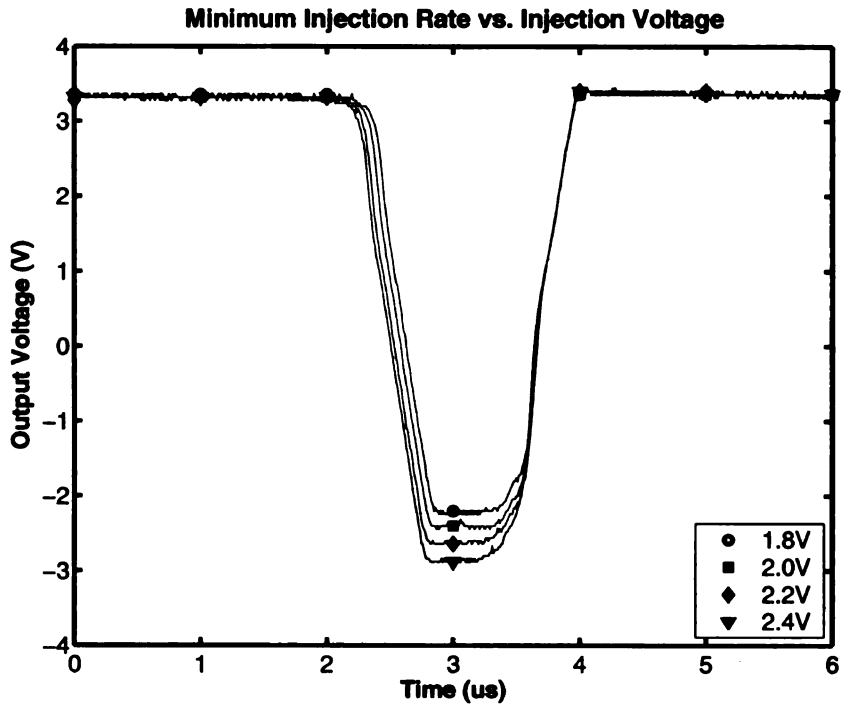


Figure 5.2. Minimum Injection Width

experiment also illustrates that the injection voltage has a slight settling and drift to its targeted amplitude.

5.1.3 Voltage-Mode ADC Linearity

The LTC2415-1 and LTC2418 have near identical performance specifications. The LTC2418, however, includes a front-end multiplexer for 16-channel data acquisition. Calibrating the LTC2415-1 will be especially important for accurate current measurement, which is essential for floating gate programming. These data converters have an external conversion clock source input F_0 that determines the conversion time. A maximum frequency of 2 MHz produces 97.5 samples per second.

Figure 5.4 shows the linearity of the data converter operating in the 2 MHz conversion clock mode. For this experiment, one channel of the voltage-mode DAC is connected to the first channel of the LTC2418. The DAC is a 16-bit converter and is guaranteed monotonic across its range of 0 to 4.096 V. First, the DAC's internal reg-

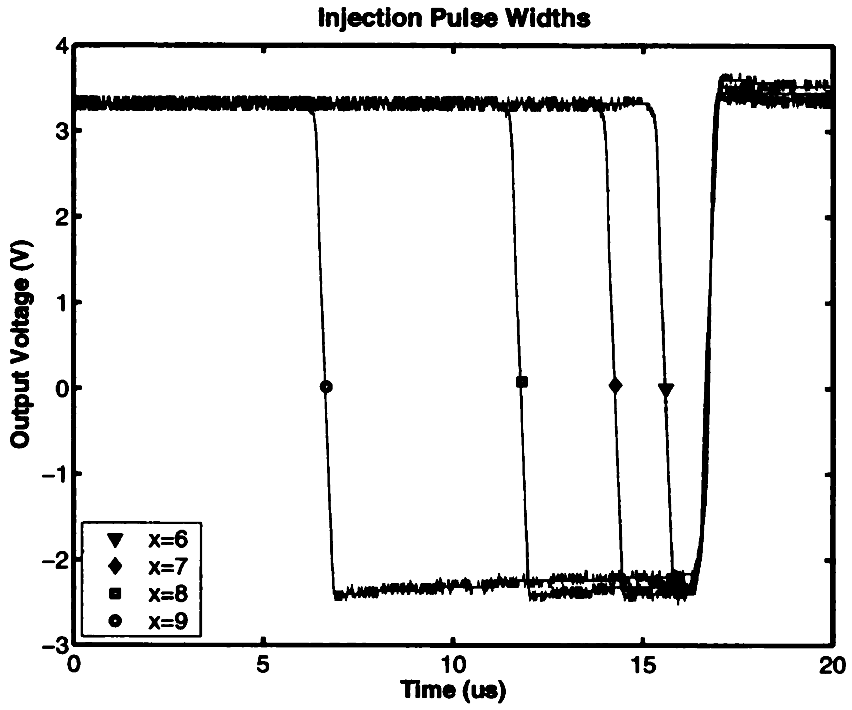


Figure 5.3. Injection Widths

ister is cleared, producing 0 V output, which is then sampled by the ADC. The DAC's register is incremented and this process is repeated spanning the entire resolution of the DAC. The deviation from ideal is plotted in millivolts.

Given that these are 24-bit data converters, a minimum of 16 bits is expected as the experiment is limited by the resolution of the digital-to-analog converters. However, Figure 5.4 shows a resolution of less than 8 bits. The DACs have a settling time of 2 μs, however the time between configuration of the DAC and the ADC sample is a minimum of 555.6 μs (for serial communications) without including setup time by the host. Therefore, this error is not due to any transients.

The LTC2418 datasheet shows that resolution is inversely proportional to sampling speed. In addition, the resolution deteriorates rapidly after 25 samples per second. Figures 5.5 and 5.6 show experimental results for 1 MHz and 400 kHz, respectively.

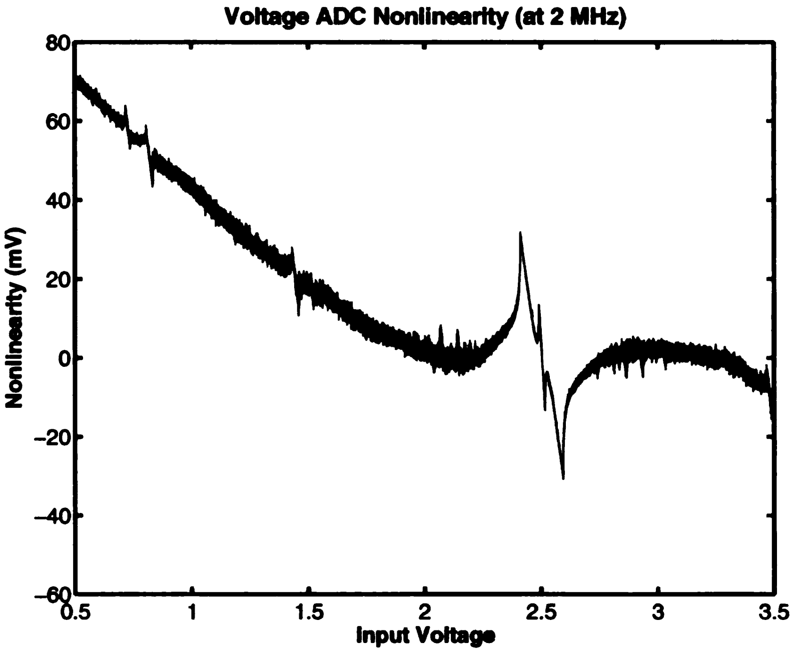


Figure 5.4. Voltage ADC Linearity at 2 MHz

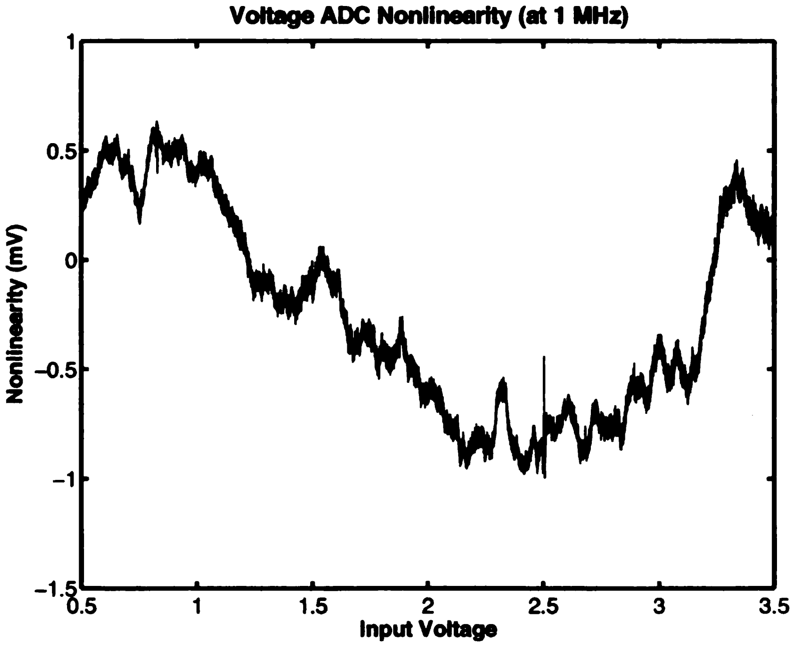


Figure 5.5. Voltage ADC Linearity at 1 MHz

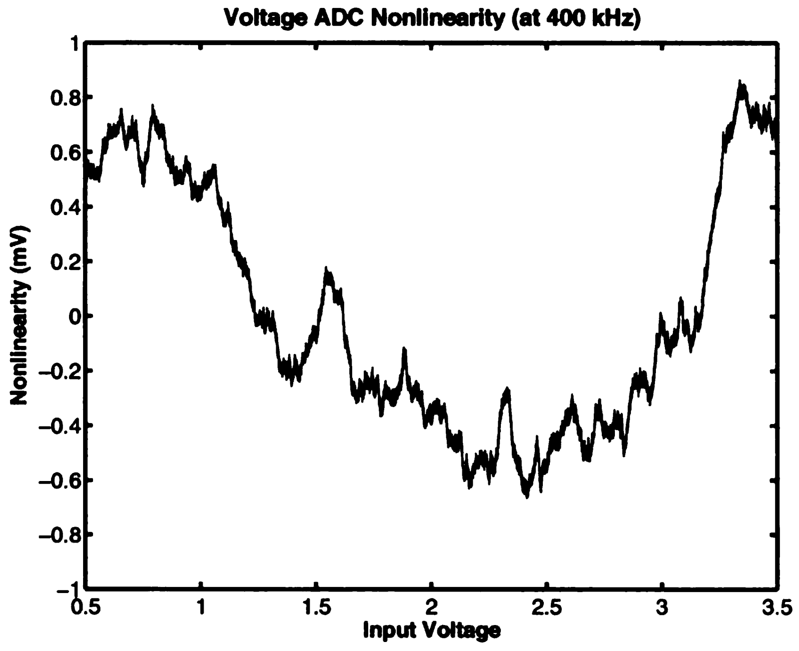


Figure 5.6. Voltage ADC Linearity at 400 kHz

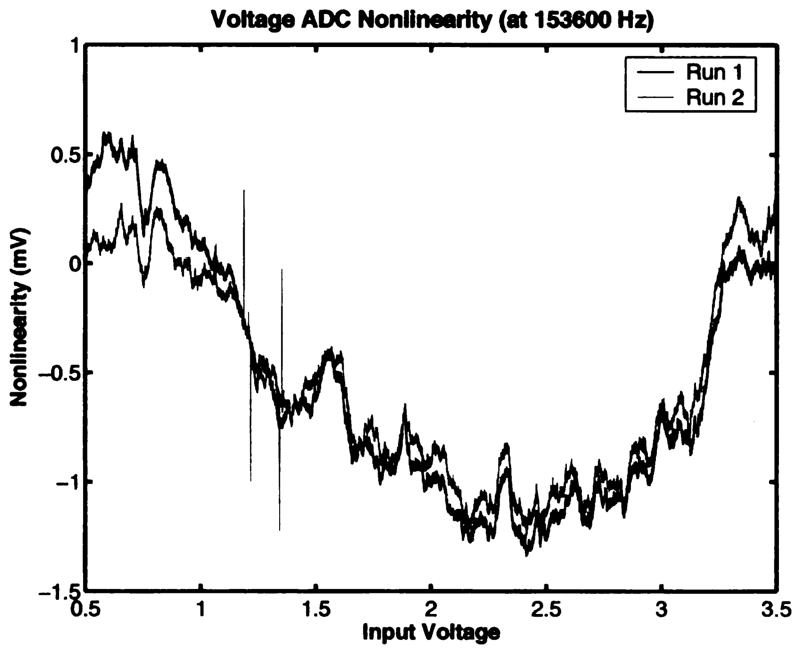


Figure 5.7. Voltage ADC Linearity at 153600 Hz

The converter has a notch filter designed to reject 60 Hz noise at its internal clock frequency when the F_0 pin is driven to logic low and rejects 50 Hz noise at logic high. The filter may be adjusted to a center frequency of $F_0/2560$ if provided an external conversion clock. This corresponds to an clock rate of 153.6 kHz for 60 Hz noise. Figure 5.7 shows the best case performance metric for the LT2418 at 6.2 samples/sec.

5.1.4 Signal Generator

The multi-channel signal generator module discussed in Section 4.14 has been tested through the example function generator script of Section E.17 in the appendices. Figures 5.8 through 5.11 show a 1 kHz waveform for the standard sine, triangle, sawtooth, and square waves with no offset or phase shift at two amplitudes: 1 and 0.5.

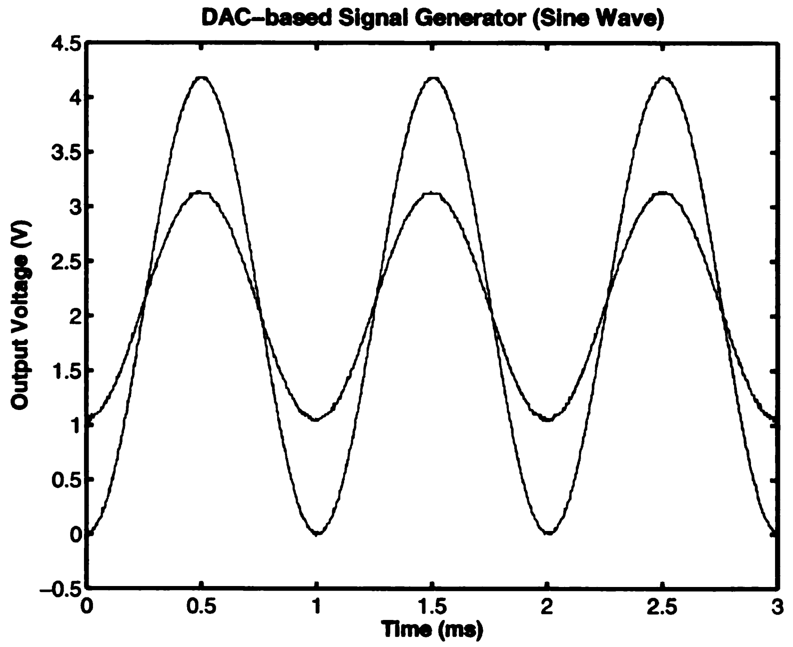


Figure 5.8. Signal Generator Sine Wave at 1 kHz

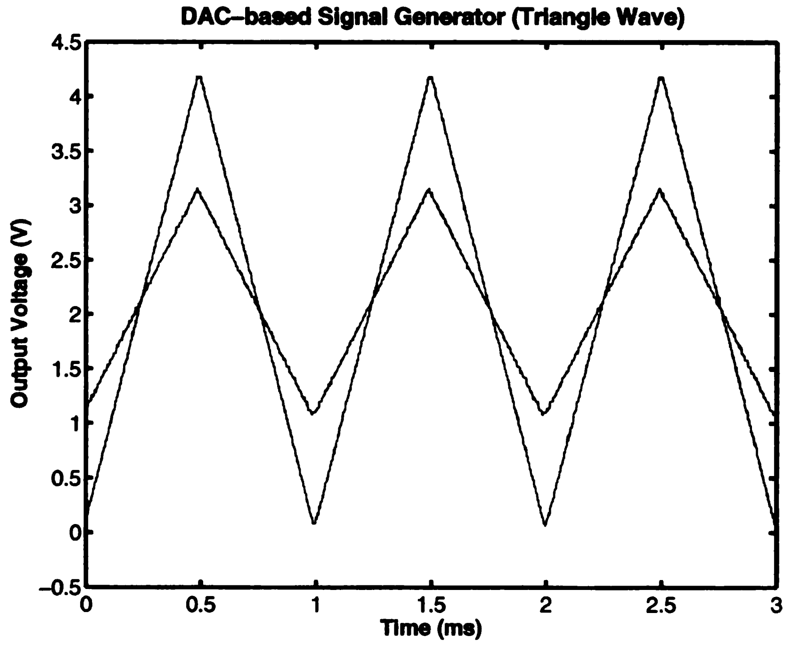


Figure 5.9. Signal Generator Triangle Wave at 1 kHz

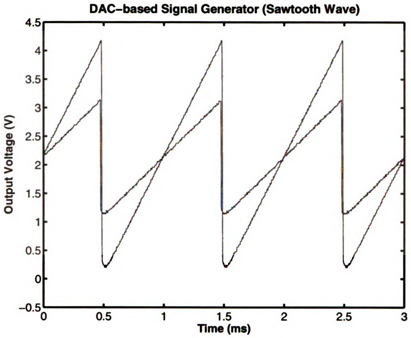


Figure 5.10. Signal Generator Sawtooth Wave at 1 kHz

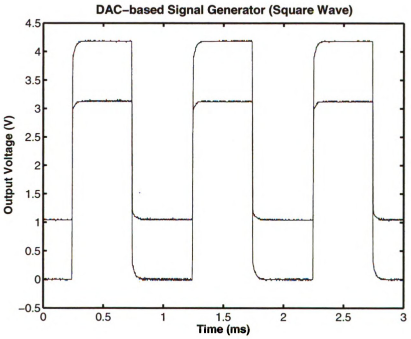


Figure 5.11. Signal Generator Square Wave at 1 kHz

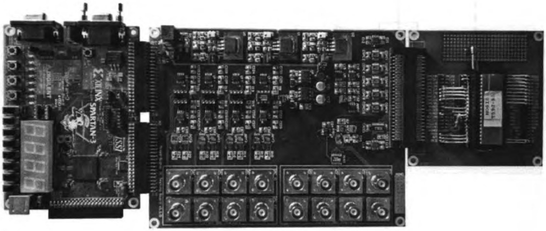


Figure 5.12. Floating Gate Programming Test Setup

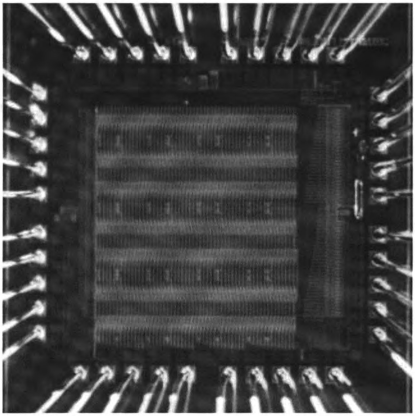


Figure 5.13. Analog SVM Chip Photomicrograph

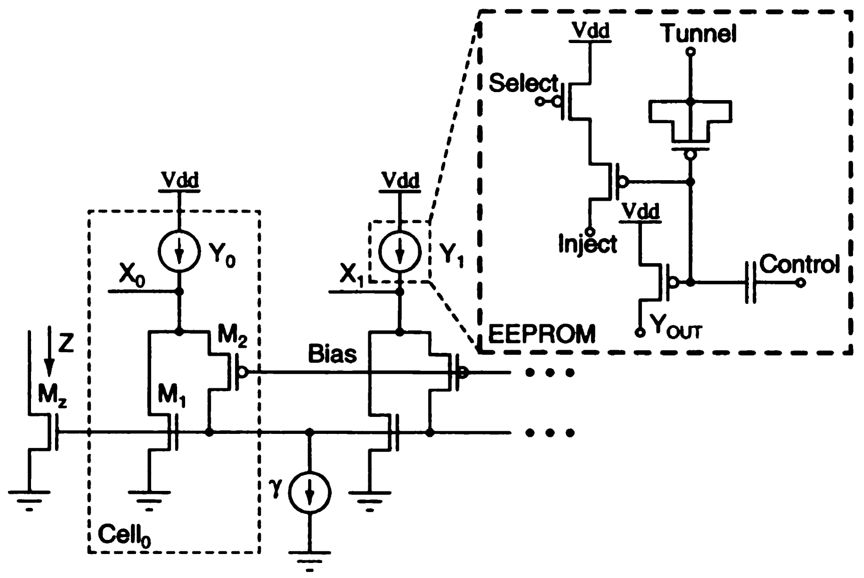


Figure 5.14. Support Vector Simplified Schematic

5.2 Overview of the Test Chip

An analog support vector machine (SVM) has been designed and fabricated using the AMI C5N process technology. This chip contains a 14-dimensional input space and 28 support vectors, which corresponds to 392 floating gates in the first stage of the SVM. The second phase of the algorithm contains an output stage with an additional 56 floating gate cells.

Without going into great detail on the SVM hardware, it is important to look at the transistor-level structure of the SVM for floating gate programming. The output currents from these cells will pass through the SVM to a common output node that is measured with the current ADC.

Figure 5.14 shows a simplified schematic of a training vector in the SVM's first stage. The floating gate cells are represented by independent current sources Y_0 and Y_1. X_0 and X_1 are represented as diode-connected transistors. The support vector output is given by Z.

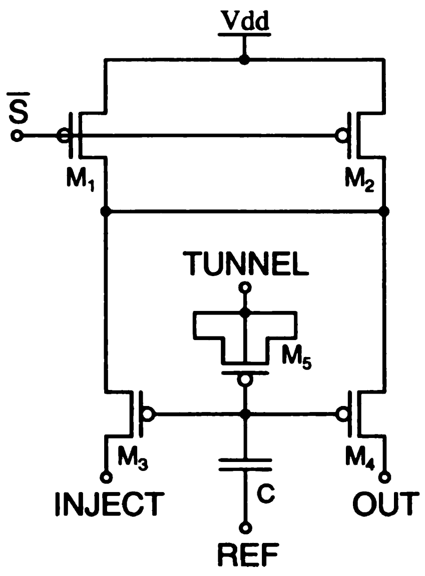


Figure 5.15. Floating Gate Cell Schematic

All ground connections meet at a common node that is separate from digital ground used for cell selection. This allows each cell to be individually selected to test the output current of individual floating gates or the performance of individual support vectors. The fabricated prototype includes cell selection transistors at the output of the X and Y inputs, above M_l and M_Z, and above the γ constant current sink.

Floating gate readout involves disabling the sourcing capability on the X_n transistors and passing all current through M_l. All other cells are disabled, including M_Z as well as M_γ for all support vectors. Additionally, this architecture requires the gate of the M_l transistor to be charged, otherwise it will not act as a switch and will limit the current flowing through the drain of the floating gate.

During every current readout or injection cycle, the M_2 transistor is enabled to charge the parasitic gate capacitance of M_l so it is completely enabled. M_2 must then be disabled, otherwise current may flow from adjacent cells, corrupting the reading. Additionally, the cascoded biasing transistor at the output of the floating gate cell is fully enabled to minimize effects of the early voltage.

Figure 5.15 shows the floating gate cell schematic as implemented on-chip. This basic topology was discussed in Chapter 2; however, note that M_1 through M_4 have been laid out as $6\mu m/3\mu m$ to minimize mismatch between the injection and readout transistors.

Cells are selected through a shift register controlled by the interface described in Section 4.12. The shift chain consists of 515 selection bits in the following order: 392 first stage cells, 28 output stage cells, six test cells, the remaining 28 output stage cells, five cells for an on-chip integrator controller, and 56 M_γ and M_Z selection cells. It is important to note that not every shift register stage contains a corresponding floating gate cell. Consequently, all experiments have been carried out on the first stage cells.

5.3 Floating Gate Testing Results

Figure 5.16 shows an array of 300 unequalized floating gate cells in their natural, post-fabrication state. With a 2 V control gate voltage, they vary from approximately 21 nA to 34.5 nA. Since the nearest integer to the maximum current read in the array is 35 nA, all cells will be equalized to this current.

5.3.1 Floating Gate Current Equalization

Figure 5.17 shows the same floating gate array programmed to 35 nA. Since the programming algorithm stops injecting once the cell current exceeds the targeted current, it can be seen that the average current slightly exceeds the targeted current. A histogram of the equalized floating gate currents is shown in Figure 5.18.

Next, the control gate voltage is increased to 2.2 V. This reduces the current in the floating gate cells by reducing the source-to-gate voltage of these capacitively-coupled pMOS transistors. Figures 5.19 and 5.20 show the measured currents in the floating gate cells and their histogram, respectively.

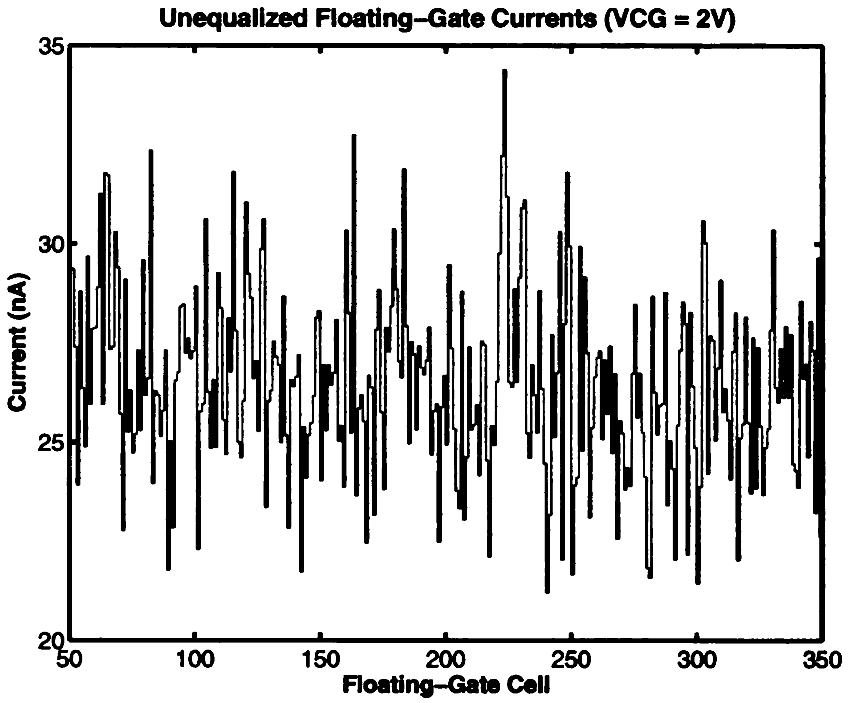


Figure 5.16. Unequalized Array of Floating Gates

5.3.2 Adaptive Injection Characteristics

Figure 5.21 shows the hot-electron injection characteristics for seven different targeted currents starting from the equalized array current of 35 nA. The algorithm used for this experiment is given in Section F.8.

Since injection current is a function of the initial current, the pulse width must be decreased as the initial current increases. The algorithm thus includes an injection width modifier that is incremented for different current ranges. The modifier is then subtracted from the default pulse width. The default pulse width is determined by the ΔI change between the previous injection cycle and targeted currents. As the measured current comes within a specified range of the targeted current, the injection pulse width is divided by two. This process is repeated until the current is within 250 pA of the targeted current.

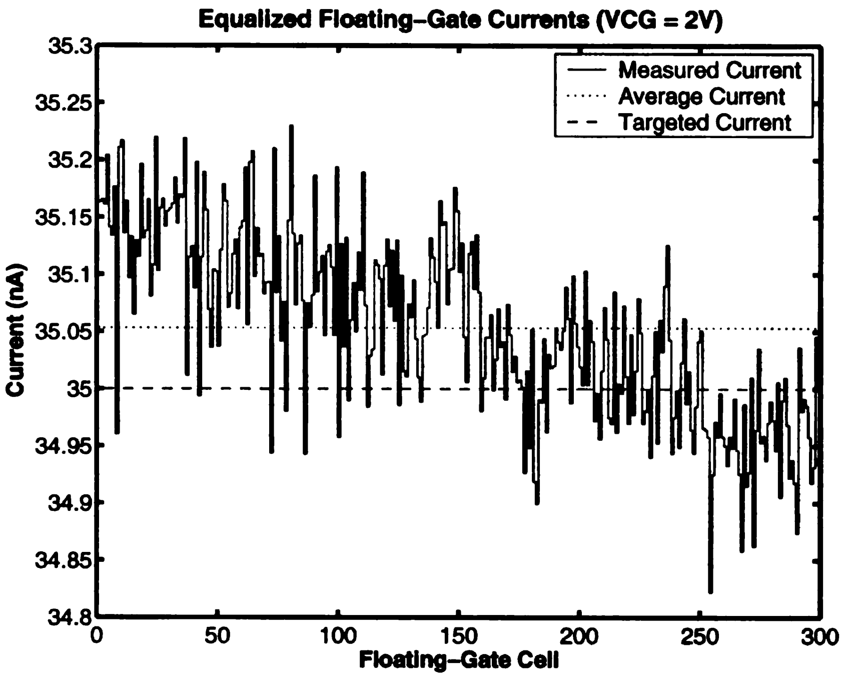


Figure 5.17. Equalized Array of Floating Gates

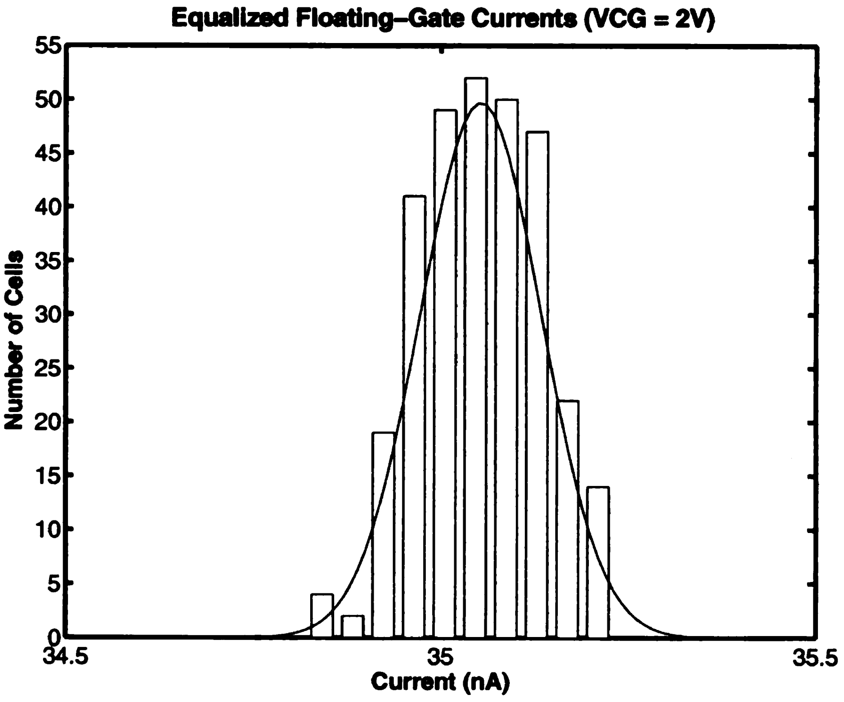


Figure 5.18. Floating Gate Equalization Accuracy Histogram

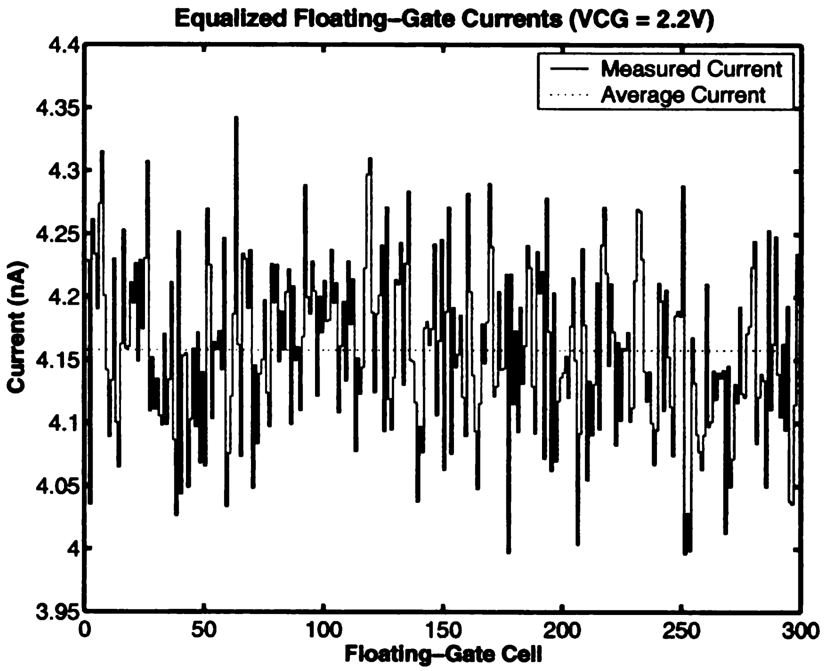


Figure 5.19. Equalized Array of Floating Gates at VCG = 2.2 V

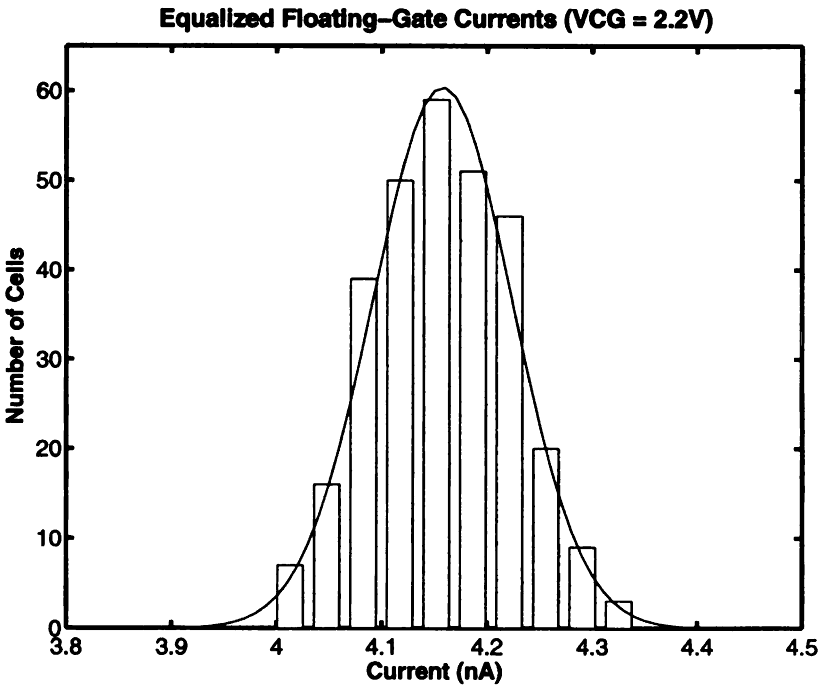


Figure 5.20. Floating Gate Equalization Accuracy Histogram at VCG = 2.2 V

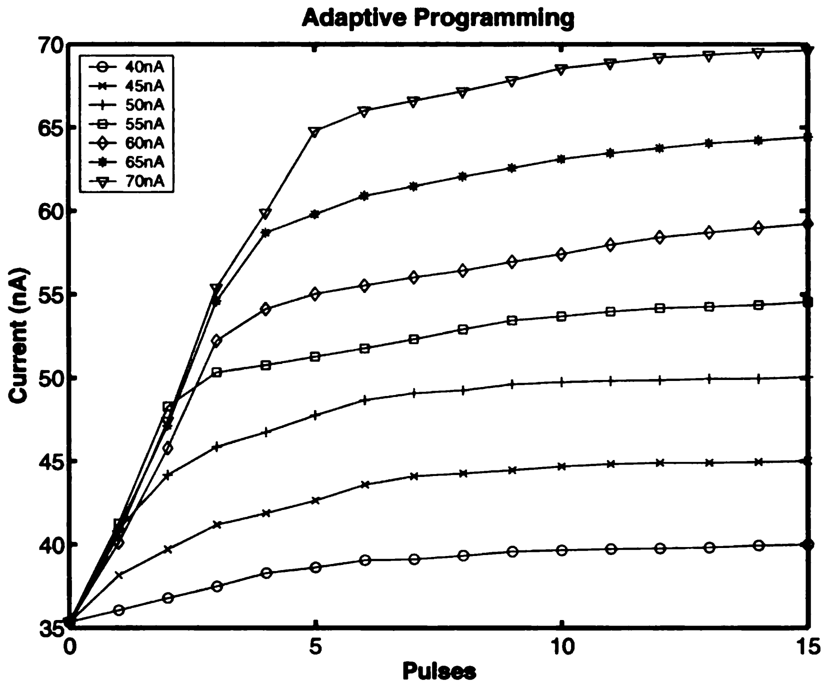


Figure 5.21. Adaptive Programming used for Hot-Electron Injection

It can be seen that the targeted current can be programmed within 2% accuracy in fewer than 16 pulses. Extrapolating this figure shows a worst case of 26 pulses for less than 0.5% accuracy. As compared to the state of the art injection algorithms that require accurate modeling of injection currents [34], this empirical model requires approximately twice as many pulses to achieve similar accuracy.

5.3.3 Effects of Injection on Threshold Voltage

Figure 5.22 shows the voltage to current characteristic for the standard pMOS transistor represented by X_n in Figure 5.15. These transistors have a width to length ratio of 6/3. This characteristic may be compared to that of the floating gate as shown in Figure 5.23. Here, the control gate voltage was set to 2 V and an array of floating gates was programmed in increments of 50 nA, starting at 50 nA.

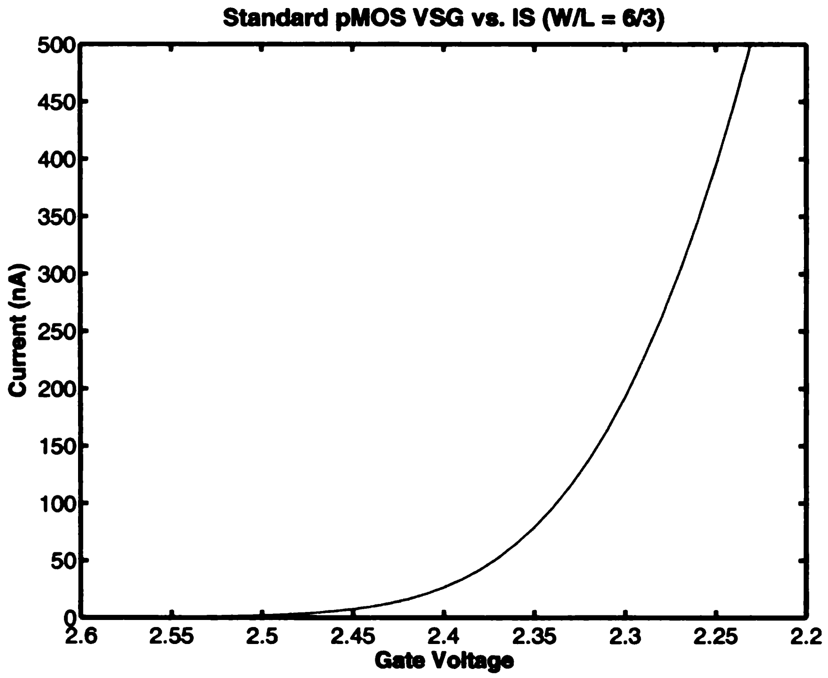


Figure 5.22. pMOS Input Stage V-I characteristics

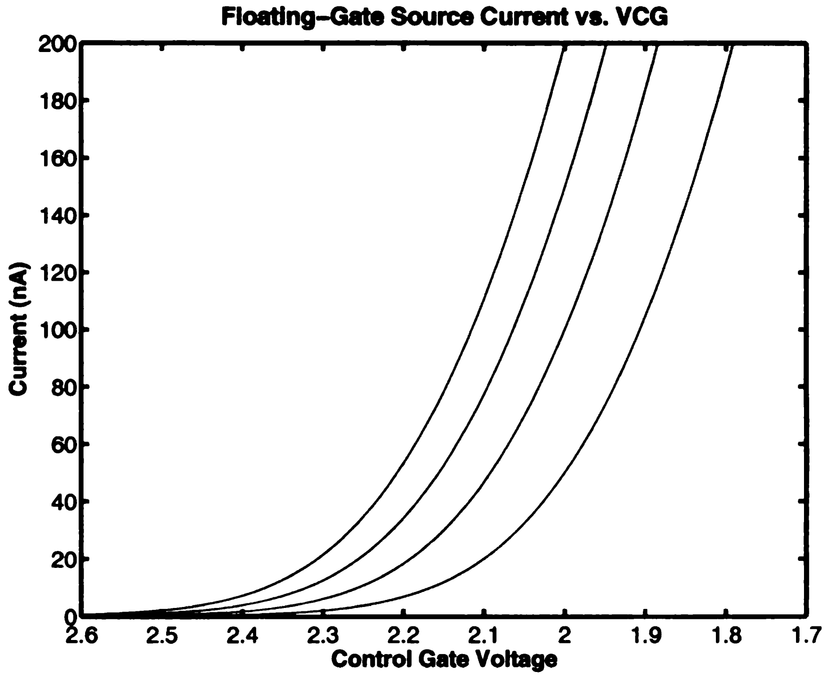


Figure 5.23. Floating Gate MOS Threshold Voltage Modulation

By sweeping the control gate voltage, the threshold voltages of the programmed floating gates may be observed. As more electrons are injected onto the floating node, the threshold voltage decreases. Furthermore, it can be seen that the voltage to current response of the floating gate is different from that of a standard pMOS transistor. This is due to different source-to-drain voltages across both transistors, as the floating gate cell includes an additional cascoded stage biased at 1.5 V.

5.3.4 Programmable Current Lookup Tables

Figures 5.24 and 5.25 demonstrate the ability to program a current-mode lookup table on-chip. The first figure illustrates a current ramp from the initial equalized current of 35 nA to 74 nA with a slope of 1 nA per cell. Figure 5.25 demonstrates a sine wave with an offset of 62 nA and an amplitude of 14 nA-pp. These offsets and amplitudes were chosen due to the initial state of the array, which had been equalized to 55 nA prior to programming.

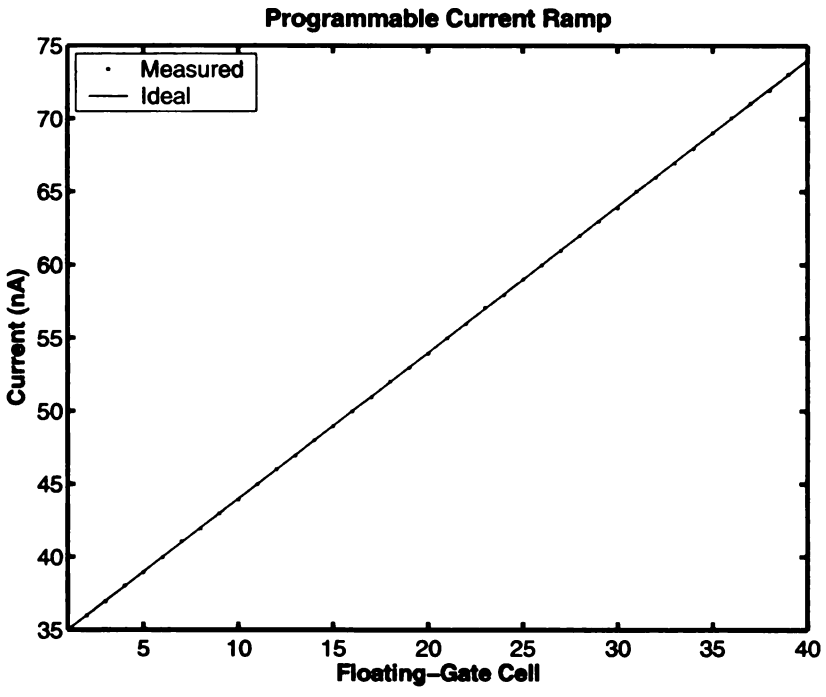


Figure 5.24. Floating Gate Current Ramp

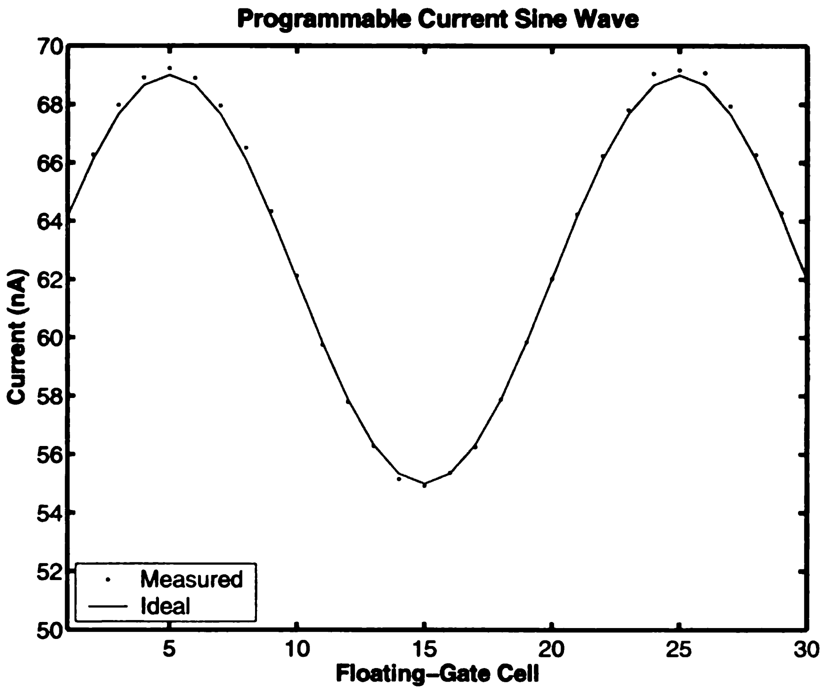


Figure 5.25. Floating Gate Current Sine Wave

CHAPTER 6

Conclusions

6.1 Accomplishments

Through this work, an automatic test unit (ATU) for topology and process-neutral analog floating gate programming and mixed-signal design testing has been fabricated and verified on a system-on-chip. This incorporated the design of a board-level system framework for precision voltage and current measurement and analog output, as well as high slew-rate injection and tunneling outputs. The system required the design of a high-speed controller realized in VHDL and synthesized on a Xilinx XC3S200. Furthermore, a software interface has been developed to provide a customizable API for future floating gate testing for analog computation in SoCs.

For the validation phase of this work, an analog support vector machine complete with floating gate transistors was designed and fabricated to test the accuracy of the ATU for performing hot-electron injection and tunneling functions, as well as off-chip current analog-to-digital conversion. Additionally, a novel floating gate programming algorithm has been developed through experimental observation, and has been used to successfully program floating gate cells to within 0.5% accuracy.

6.2 Suggestions For Future Work

This work is an ongoing effort to develop a rapid prototyping and verification system for mixed-signal ICs. Future iterations will improve on the features introduced in this work and further refine the methods developed herein. Several drawbacks to the existing system were identified during system validation and these items are discussed below.

6.2.1 Floating Gate Architectures

This work explored only one floating gate architecture that used a separate MOS device for programming and current output. A more efficient design would incorporate these two functions using the same floating gate MOS device. This ensures that both the injection and readout nodes have the same current sourcing characteristics. This is important for accurate modeling of the injection rate based on the existing control gate voltage and the charge stored on the floating node. Since mismatch between the injection and readout transistors is likely to occur, the $\Delta I/\Delta t$ where t represents the injection pulse period will not be constant across different cells given their output currents are equalized. Furthermore, any charge trapping that occurs in the gate oxide layer of the injection transistor will have a further effect on injection modeling, making it more difficult to characterize the current output transistor.

However, although it would appear this method would require less area per floating gate cell by eliminating the additional injection transistor, it would occupy a greater footprint due to two required multiplexing transistors. Furthermore, two additional multiplexing transistors should be added to the existing architecture in order to bypass the support vector machine for direct current measurements. The existing topology requires the floating gate current to pass through the support vector cells to the ground terminal. This creates testing complications since this current cannot be measured directly, or requires the SVM to be configured in an unstable state.

6.2.2 Board-Level Modifications

The floating gate test station was designed for the AMI C5N process, which requires a minimum of 15 V for Fowler-Nordheim tunneling and allows hot-electron injection between 4 V and 6.5 V. The tunneling and injection supplies were therefore designed for these voltages without giving consideration to more sophisticated fabrication processes.

A 0.25 μm process, for example, uses a 10 V tunneling voltage and 3.8 V V_{DS} for hot-electron injection. Although the amplitude of the injection output can be adjusted through the configuration of voltage DAC channel #37, it cannot be reduced beyond a certain threshold without the MAX1681 inverting DC-DC converter exceeding its input range. One solution is to modulate the test chip's supply voltage using one of the voltage DAC channels. Since each DAC channel is capable of sourcing up to 15 mA of current, these channels may be used directly to decrease the supply for injection at a lower V_{DS}.

Furthermore, digital potentiometers could be used at the output of the tunneling and injection supply nets to divide their output voltages. This method may be preferable for two reasons. First, as demonstrated in Chapter 5, there is not a direct correspondence between the output of the DAC channel and the injection supply output. Since the digital potentiometers are guaranteed to be monotonic regardless of their nominal values, the output may be accurately calculated based on a fixed output voltage. Second, the potentiometer would provide an additional load for these DC-DC converters, thus improving their efficiency.

Another potential improvement could come from modifying the topology of the current DAC to provide sinking as well as sourcing. This would allow current biasing from pMOS-based structures. One possible implementation could include inverting some of the channels to sink rather than source current. This may be done by using an inverting regulator that maintains a negative potential across the digital potentiometer in the existing topology. Both implementations may be combined by connecting both regulators to the same common ground path and using a switch to reverse the voltage polarity across the digital potentiometer.

6.2.3 Microcontroller-Based Test Station

The merits of an FPGA-based versus a microcontroller-based test station include high speed through parallel processing and access to off-chip RAM for data storage in real-time data acquisition experiments. Such performance cannot be matched by a general purpose microcontroller unit such as the PIC18F or dsPIC30/33 families by Microchip or the MSP430 family by TI due to limited I/O ports and peripheral speed. Even the hardware SPI and I^2C modules in these devices are only capable up to 2 Mbps whereas the FPGA is capable up to 125 Mbps when using the PLL module of the FPGA's digital clock manager.

However, these devices offer the advantage of being easily integrated onto the existing test station PCB through package availability such as DIP and SOIC, which stands in stark contrast to BGA and TQFP of modern FPGA devices. Additionally, some performance bottlenecks can be reduced by incorporating demultiplexers on the output of the part for chip selection in a SPI-only environment. This would allow individual part selection, eliminating daisy-chained configurations and reducing the number of shift cycles during serial communication with these devices.

The FPGA development board can accommodate a large number of parallel devices such as external SRAM due to high density packaging. Chips up to 2 MB are common as of 2007 and the current system integrates 512 kB of SRAM through two 256 kB devices. This cannot be matched directly by a microcontroller unit whose options for external memory include serial EEPROMs or parallel SRAM. In the case of SRAM, due to large data and address bus widths, multiplexers and latches would be required to interface with the microcontroller using a minimum number of pins, a partial speed penalty that makes real-time data acquisition more challenging. Additionally, microcontrollers have at most 30 kB of data RAM, which would limit the data buffer size before serially transferring the data for post processing.

An alternative to this problem would be to use the PIC18F2550/4550 family of USB microcontrollers capable of 12 million instructions per second (MIPS) [59]. USB can alleviate communication bottlenecks by providing up to 12 Mbps for full speed devices. Utilizing interrupt transfers, latency can be guaranteed [60], thus real-time data acquisition may occur through immediate transfer with very small buffers. Also, unlike the similar isochronous transfer mode, interrupt transfers include error checking, thus ensuring data transmission accuracy. Furthermore, the bus can supply power directly to the test station with up to 500 mA of current, well above the power requirements of the existing test station (70 mA) and thus eliminating the need for external power adapters.

In conclusion, one outcome of this work has been to create a modular test station. Through this, it is possible to make a single demo platform that combines both the functionality of the motherboard and daughter card containing the SoC under test. Taking into account the lessons learned in this design and the recommendations outlined above, it is now possible to make a completely integrated system which does not require external test and measurement equipment for chip testing.

APPENDICES

APPENDIX A

Support Vector Machine SoC

A.1 Layout

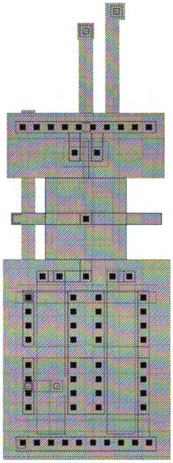


Figure A.1. Floating-Gate Transistor Layout

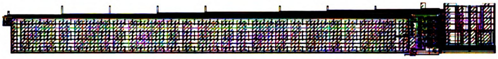


Figure A.2. Integrator Layout

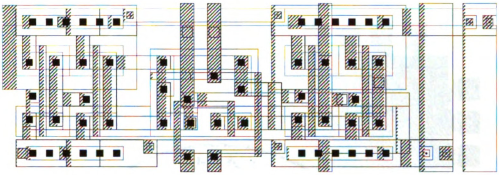


Figure A.3. Shift Register Layout

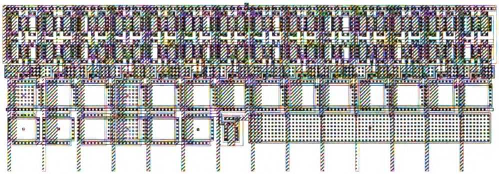


Figure A.4. Support Vector Layout

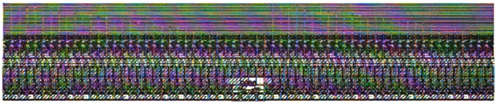


Figure A.5. Output Stage Layout with Floating Gates and Shift Register

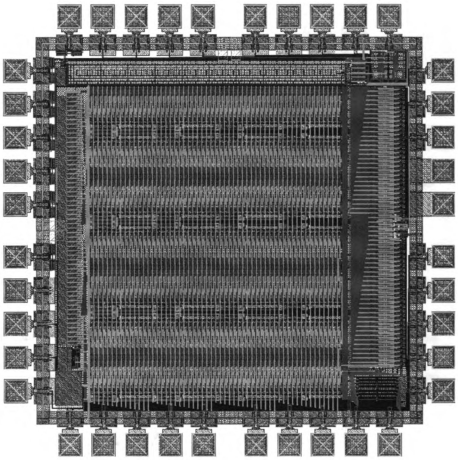


Figure A.6. Support Vector Machine Layout

A.2 Pad Frame

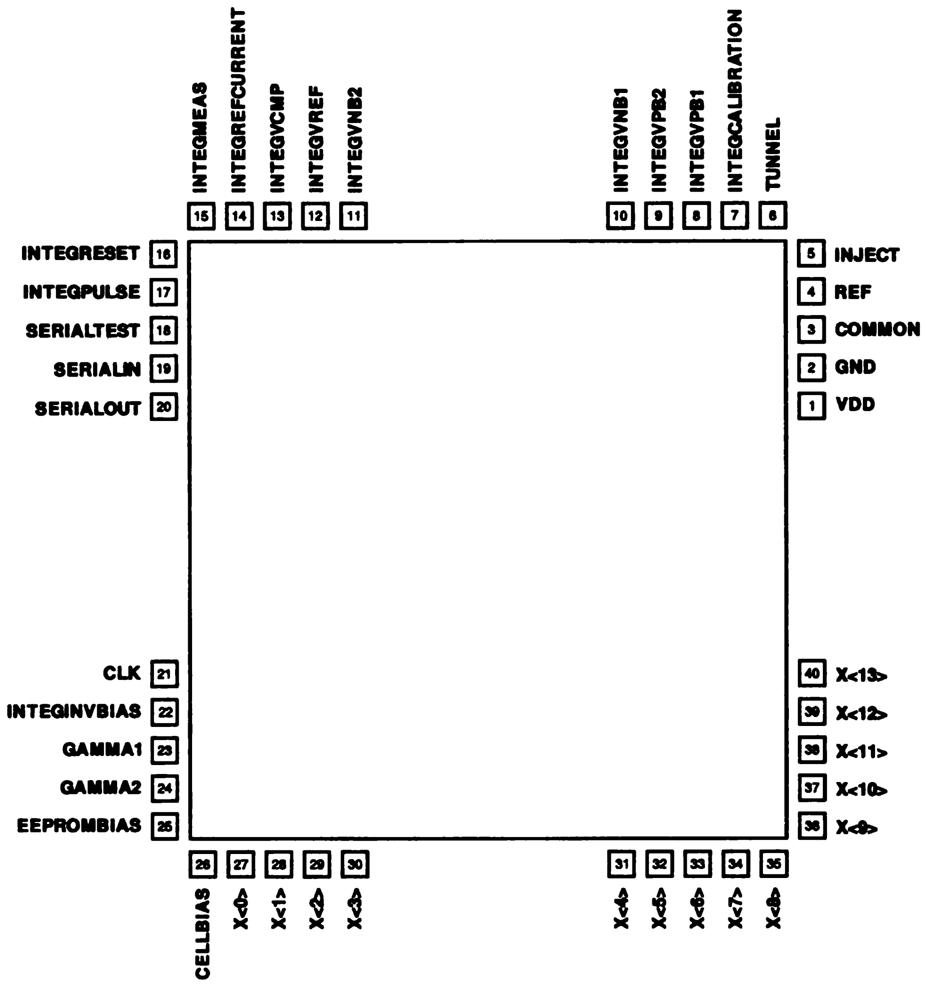


Figure A.7. Fabricated Prototype Pin Bonding Diagram

Pin	Name	Type	Description
1	VDD	Jumper	Supply
2	GND	Jumper	Ground
3	COMMON	Analog I/O	Aux Ground / Measurement
4	REF	Analog I/O	EEPROM Control Gate Reference
5	INJECT	Analog I/O	EEPROM Injection
6	TUNNEL	Analog I/O	EEPROM Tunneling
7	INTEGCALIBRATION	Analog I/O	Integrator Calibration Current
8	INTEGVPB1	Bias	Integrator Transconductor Bias
9	INTEGVPB2	Bias	Integrator Transconductor Bias
10	INTEGVNB1	Bias	Integrator Transconductor Bias
11	INTEGVNB2	Bias	Integrator Transconductor Bias
12	INTEGVREF	Bias	Integrator Reference Voltage
13	INTEGVCMP	Bias	Integrator Comparator Threshold
14	INTEGREFCURRENT	Bias	Integrator Input Sink
15	INTEGMEAS	Digital In	Integrator Measurement Control
16	INTEGRESET	Digital In	Integration Capacitor Reset
17	INTEGPULSE	Digital Out	Integrator Output Pulse
18	SERIALTEST	Digital Out	Shift Chain Cell #424 Out
19	SERIALIN	Digital In	Serial Shift Chain In
20	SERIALOUT	Digital Out	Serial Shift Chain Out
21	CLK	Digital In	Input Clock
22	INTEGINVBIAS	Bias	Integrator Output Inverter Bias
23	GAMMA1	Bias	Gamma #1 Bias
24	GAMMA2	Bias	Gamma #2 Bias
25	EEPROMBIAS	Bias	EEPROM Cascode Bias
26	CELLBIAS	Bias	SVM Cell Bias
27	X<0>	Analog I/O	Input Dimension #1
28	X<1>	Analog I/O	Input Dimension #2
29	X<2>	Analog I/O	Input Dimension #3
30	X<3>	Analog I/O	Input Dimension #4
31	X<4>	Analog I/O	Input Dimension #5
32	X<5>	Analog I/O	Input Dimension #6
33	X<6>	Analog I/O	Input Dimension #7
34	X<7>	Analog I/O	Input Dimension #8
35	X<8>	Analog I/O	Input Dimension #9
36	X<9>	Analog I/O	Input Dimension #10
37	X<10>	Analog I/O	Input Dimension #11
38	X<11>	Analog I/O	Input Dimension #12
39	X<12>	Analog I/O	Input Dimension #13
40	X<13>	Analog I/O	Input Dimension #14

Table A.1. Fabricated Prototype Pin Descriptions

APPENDIX B

Test Station Design Documentation

B.1 Test Station Parts List

Designator	Description	Manufacturer Part
A1	CONN HDR BRKWAY .100 80POS RT/A	4-103326-0
A2	CONN HDR BRKWAY .100 80POS RT/A	4-103326-0
A3	CONN RECEPT R/A 100POS 1.27MM	FX2-100S-1.27DS(71)
ADC1	IC ADC 16CH 24BIT DIFINPUT28SSOP	LTC2418CGN#PBF
ADC2	IC ADC 24BIT DIFFINPUT/REF16SSOP	LTC2415-1CGN#PBF
ADC #1	CONN JACK BNC VERT 50OHM PCB	227699-1
ADC #2	CONN JACK BNC VERT 50OHM PCB	227699-1
ADC #3	CONN JACK BNC VERT 50OHM PCB	227699-1
ADC #4	CONN JACK BNC VERT 50OHM PCB	227699-1
ADC #5	CONN JACK BNC VERT 50OHM PCB	227699-1
ADC #6	CONN JACK BNC VERT 50OHM PCB	227699-1
ADC #7	CONN JACK BNC VERT 50OHM PCB	227699-1
ADC #8	CONN JACK BNC VERT 50OHM PCB	227699-1
AMP0	IC OPAMP RRIO DUAL 12V 8-SOIC	OPA2743UAG4
AMP1	IC DUAL R-TO-R OP AMP 8-SOIC	TLC2252IDG4
AMP2	IC OPAMP RRIO DUAL 12V 8-DIP	OPA2743PAG4
AMP3	IC OPAMP RRIO DUAL 12V 8-DIP	OPA2743PAG4
AMP4	IC OPAMP RRIO DUAL 12V 8-DIP	OPA2743PAG4
AMP5	IC OPAMP RRIO DUAL 12V 8-DIP	OPA2743PAG4
C0	CAP TANTALUM 22UF 16V 20% SMD	F931C226MCC
C1	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
C2	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
C3	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
C4	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
C5	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K

Designator	Description	Manufacturer Part
C6	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
C7	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
C8	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
C9	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
C10	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
C11	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
C12	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
C13	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
C14	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
C15	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
C16	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
C17	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
C18	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
C19	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
C20	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
CADC1	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CB1	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CB2	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CB3	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CB4	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CB5	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CB6	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CB7	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CB8	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CB9	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CDAC1	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC2	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC3	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC4	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC5	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC6	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC7	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC8	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC9	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CDAC10	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CID0	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CID1	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
CID2	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CID3	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
CID4	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CID5	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA

Designator	Description	Manufacturer Part
CID6	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CID7	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
CID8	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CID9	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
CID10	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CID11	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
CID12	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CID13	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
CID14	CAP 4.7UF 25V CERAMIC F 1206	ECJ-3FF1E475Z
CID15	CAP TANTALUM 1UF 16V 20% SMD	F931C105MAA
CR1	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
CR2	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
CR3	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
CR4	CAP TANTALUM 10UF 16V 20% SMD	F931C106MBA
CR5	CAP 1UF 16V CERAMIC X7R 1206	ECJ-3YB1C105K
CT1	CAP 33UF 25V ELECT MZA SMD	EMZA250ADA330MF61G
CT2	CAP 33UF 25V ELECT MZA SMD	EMZA250ADA330MF61G
D0	LED RED CLEAR 1206 SMD	LTST-C150CKT
D1	LED RED CLEAR 1206 SMD	LTST-C150CKT
D2	LED RED CLEAR 1206 SMD	LTST-C150CKT
D3	LED RED CLEAR 1206 SMD	LTST-C150CKT
D4	DIODE SCHOTTKY 30V 1.5A NMP 2P	MA2Q70500L
D5	DIODE SCHOTTKY 30V 1.5A NMP 2P	MA2Q70500L
DAC1	IC DAC OCTAL R-R 16BIT 16SSOP	LTC2600CGN#PBF
DAC2	IC DAC OCTAL R-R 16BIT 16SSOP	LTC2600CGN#PBF
DAC3	IC DAC OCTAL R-R 16BIT 16SSOP	LTC2600CGN#PBF
DAC4	IC DAC OCTAL R-R 16BIT 16SSOP	LTC2600CGN#PBF
DAC5	IC DAC OCTAL R-R 16BIT 16SSOP	LTC2600CGN#PBF
DAC #1	CONN JACK BNC VERT 50OHM PCB	227699-1
DAC #2	CONN JACK BNC VERT 50OHM PCB	227699-1
DAC #3	CONN JACK BNC VERT 50OHM PCB	227699-1
DAC #4	CONN JACK BNC VERT 50OHM PCB	227699-1
DAC #5	CONN JACK BNC VERT 50OHM PCB	227699-1
DAC #6	CONN JACK BNC VERT 50OHM PCB	227699-1
DAC #7	CONN JACK BNC VERT 50OHM PCB	227699-1
DAC #8	CONN JACK BNC VERT 50OHM PCB	227699-1
JP0	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND
JP1	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND
JP2	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND
JP3	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND
JP4	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND

Designator	Description	Manufacturer Part
JP5	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND
JP6	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND
JP7	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND
JP8	SHORTING JUMPER GLD/NICKEL BLUE	929955-06-ND
JPW	CONN POWERJACK MINI .1" R/A PCMT	SC237-ND
L1	FERRITE CHIP 1000 OHM 200MA 0805	BLM21AG102SN1D
L2	POWER INDUCTOR 1.0mH 0.18A SMD	CDRH74NP-102MC
MUX1	IC SW OCTAL SER 2.7/5.5V 24TSSOP	ADG715BRUZ-ND
POT1	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARUZ100
POT2	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARU10
POT3	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARU10
POT4	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARU10
POT5	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARU10
POT6	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARU10
POT7	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARU10
POT8	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARU10
POT9	IC POT DIGITAL 128POS 14-TSSOP	AD7376ARU10
R0	RES 1.00K OHM 1/4W 1% 1206 SMD	ERJ-8ENF1001V
R1	RES 511 OHM 1/4W 1% 1206 SMD	ERJ-8ENF5110V
R2	TRIMPOT 2K OHM 4MM TOP ADJ SMD	3224W-1-202E
R3	RES 511 OHM 1/4W 1% 1206 SMD	ERJ-8ENF5110V
R4	RES 511 OHM 1/4W 1% 1206 SMD	ERJ-8ENF5110V
R5	TRIMPOT 2K OHM 4MM TOP ADJ SMD	3224W-1-202E
R6	RES 511 OHM 1/4W 1% 1206 SMD	ERJ-8ENF5110V
R7	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R8	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R9	RES 2.20M OHM 1/8W 1% 0805 SMD	MCR10EZHF2204
R10	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R11	RES 2.20M OHM 1/8W 1% 0805 SMD	MCR10EZHF2204
R12	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R13	RES 2.20M OHM 1/8W 1% 0805 SMD	MCR10EZHF2204
R14	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R15	RES 2.20M OHM 1/8W 1% 0805 SMD	MCR10EZHF2204
R16	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R17	RES 2.20M OHM 1/8W 1% 0805 SMD	MCR10EZHF2204
R18	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R19	RES 2.20M OHM 1/8W 1% 0805 SMD	MCR10EZHF2204
R20	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R21	RES 2.20M OHM 1/8W 1% 0805 SMD	MCR10EZHF2204
R22	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R23	RES 2.20M OHM 1/8W 1% 0805 SMD	MCR10EZHF2204

Designator	Description	Manufacturer Part
R24	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
R25	RES 511 OHM 1/4W 1% 1206 SMD	ERJ-8ENF5110V
RADC1	RES 2.20K OHM 1/8W 1% 0805 SMD	MCR10EZHF2201
RR1	RES 511 OHM 1/4W 1% 1206 SMD	ERJ-8ENF5110V
RR2	TRIMPOT 2K OHM 4MM TOP ADJ SMD	3224W-1-202E
RR3	RES 511 OHM 1/4W 1% 1206 SMD	ERJ-8ENF5110V
TUNNEL	IC MOSFET DRVR SGL HS 9A 8-DIP	UCC37322P
U0	IC REG POSITIVE 1.5A LDO TO-263	LM1086CS-ADJ/NOPB
U1	IC REG POSITIVE 1.5A LDO TO-263	LM1086CS-ADJ/NOPB
U2	IC REG POSITIVE 1.5A LDO TO-263	LM1086CS-ADJ/NOPB
U3	IC PREC REF LDO 4.096V 8-SOIC	LT1461ACS8-4#PBF
U4	IC REF LDO MICROPWR 2.5V 8SOIC	LT1461ACS8-2.5
U5	IC SW-CAP VOLT CONV 8-SOIC	MAX1681ESA
U7	IC SW-CAP VOLT CONV 8-SOIC	MAX1681ESA
U8	IC DC-DC CONV HI EFF 8-DIP	MAX762CPA+
U10	IC VOLT REFERENCE LDO 2.5V 8SOIC	REF192FSZ
U11	IC VOLT REFERENCE LDO 2.5V 8SOIC	REF192FSZ
U12	IC VOLT REFERENCE LDO 2.5V 8SOIC	REF192FSZ
U13	IC VOLT REFERENCE LDO 2.5V 8SOIC	REF192FSZ
U14	IC VOLT REFERENCE LDO 2.5V 8SOIC	REF192FSZ
U15	IC VOLT REFERENCE LDO 2.5V 8SOIC	REF192FSZ
U16	IC VOLT REFERENCE LDO 2.5V 8SOIC	REF192FSZ
U17	IC VOLT REFERENCE LDO 2.5V 8SOIC	REF192FSZ

Table B.1. Test Station Parts List

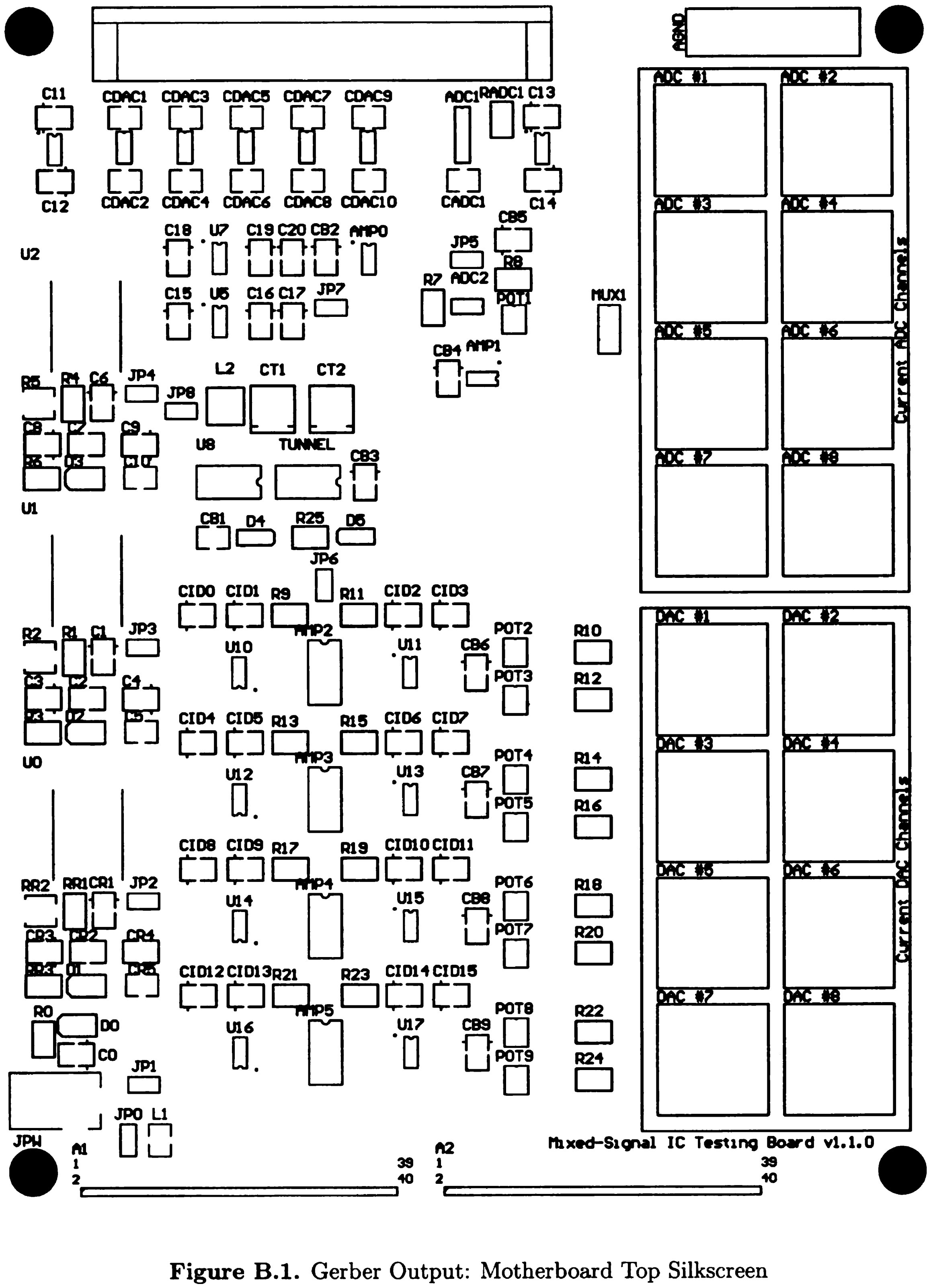


Figure B.1. Gerber Output: Motherboard Top Silkscreen

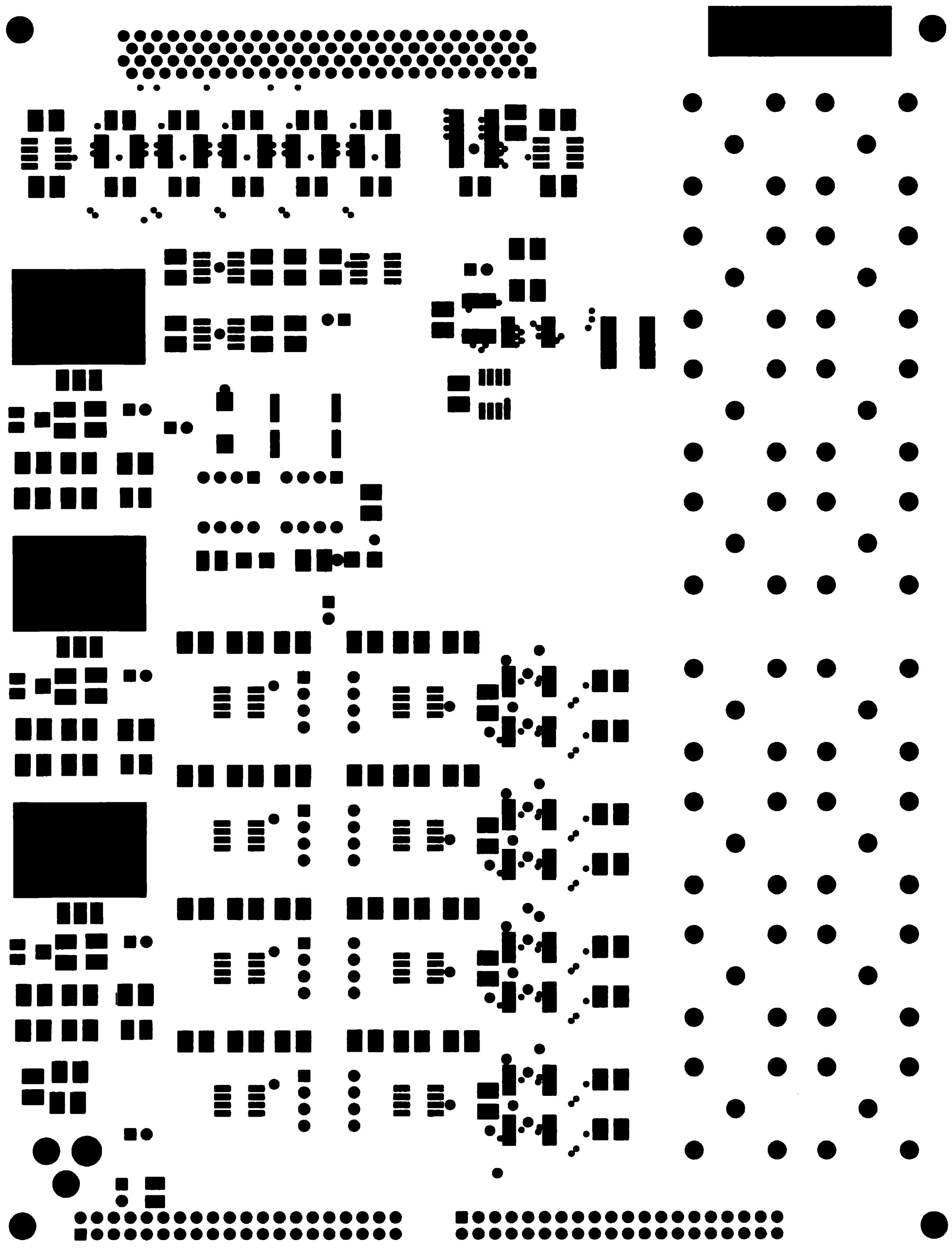


Figure B.2. Gerber Output: Motherboard Top Solder Mask

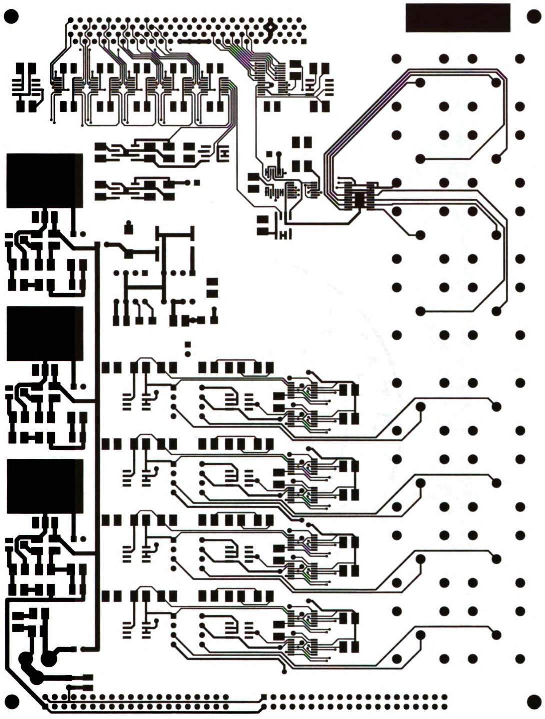


Figure B.3. Gerber Output: Motherboard Top Layer

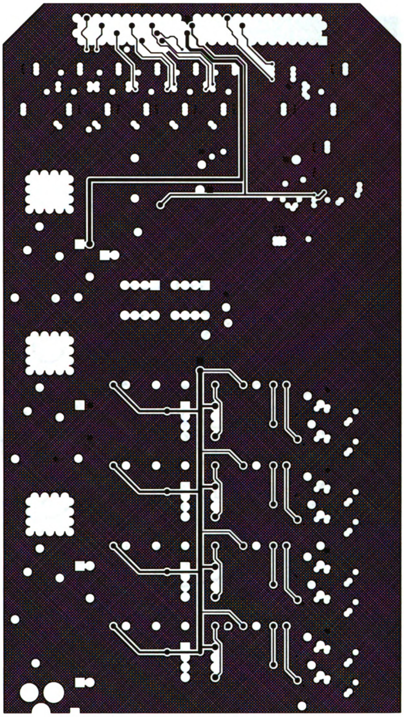


Figure B.4. Gerber Output: Motherboard Middle Layer 1

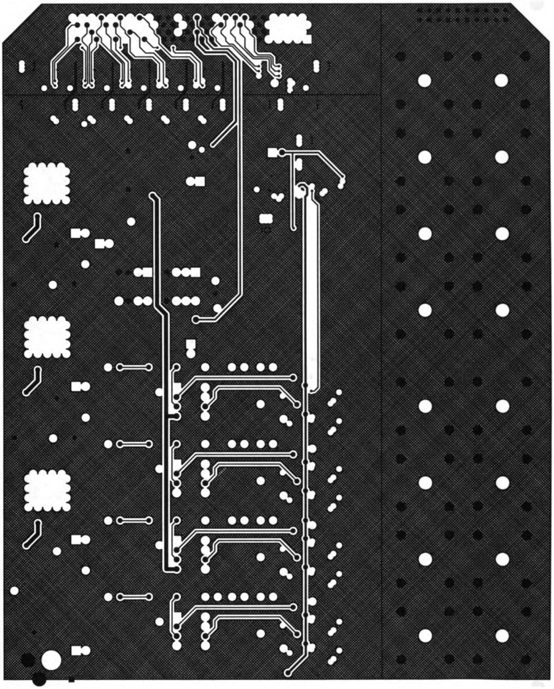


Figure B.5. Gerber Output: Motherboard Middle Layer 2

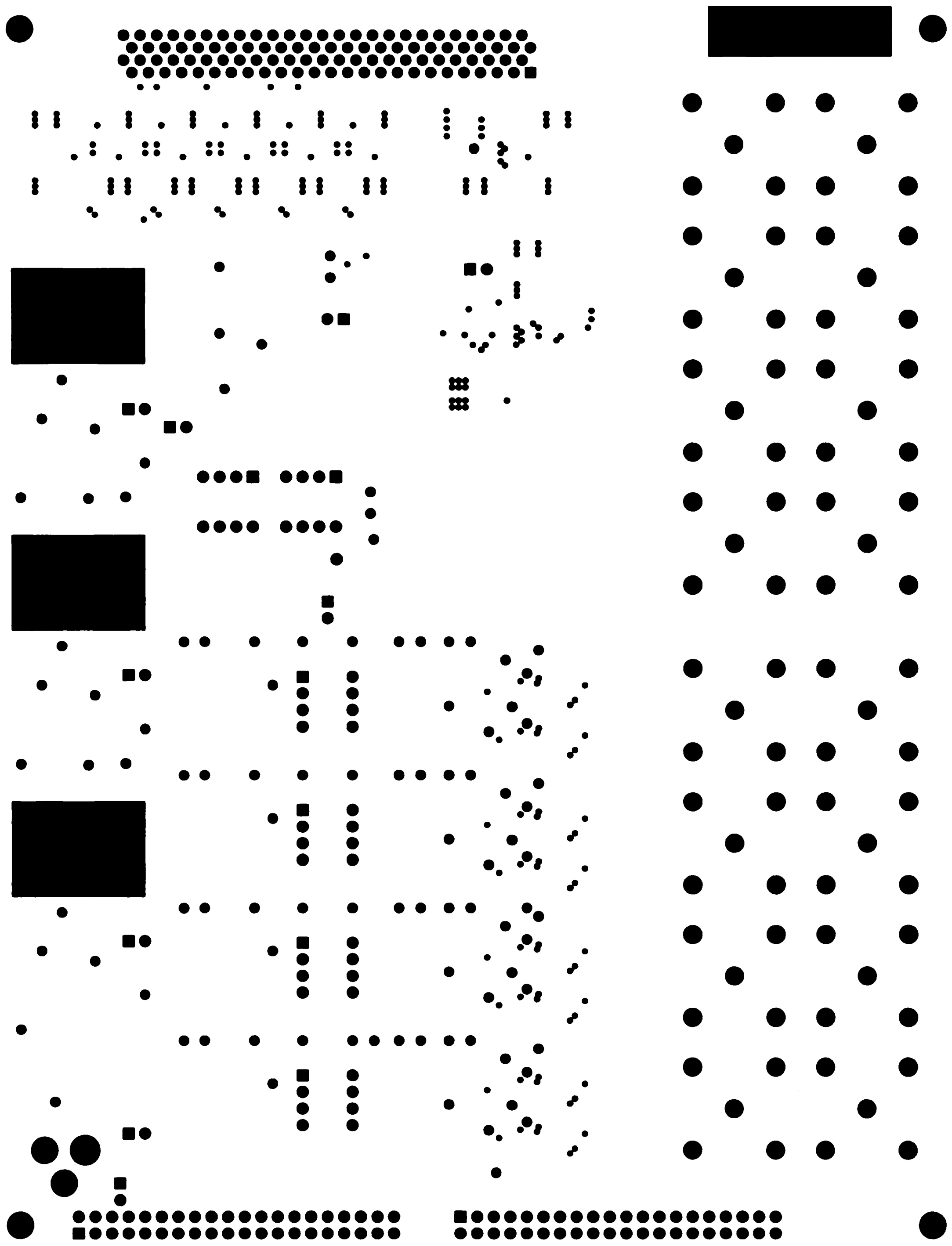


Figure B.7. Gerber Output: Motherboard Bottom Solder Mask

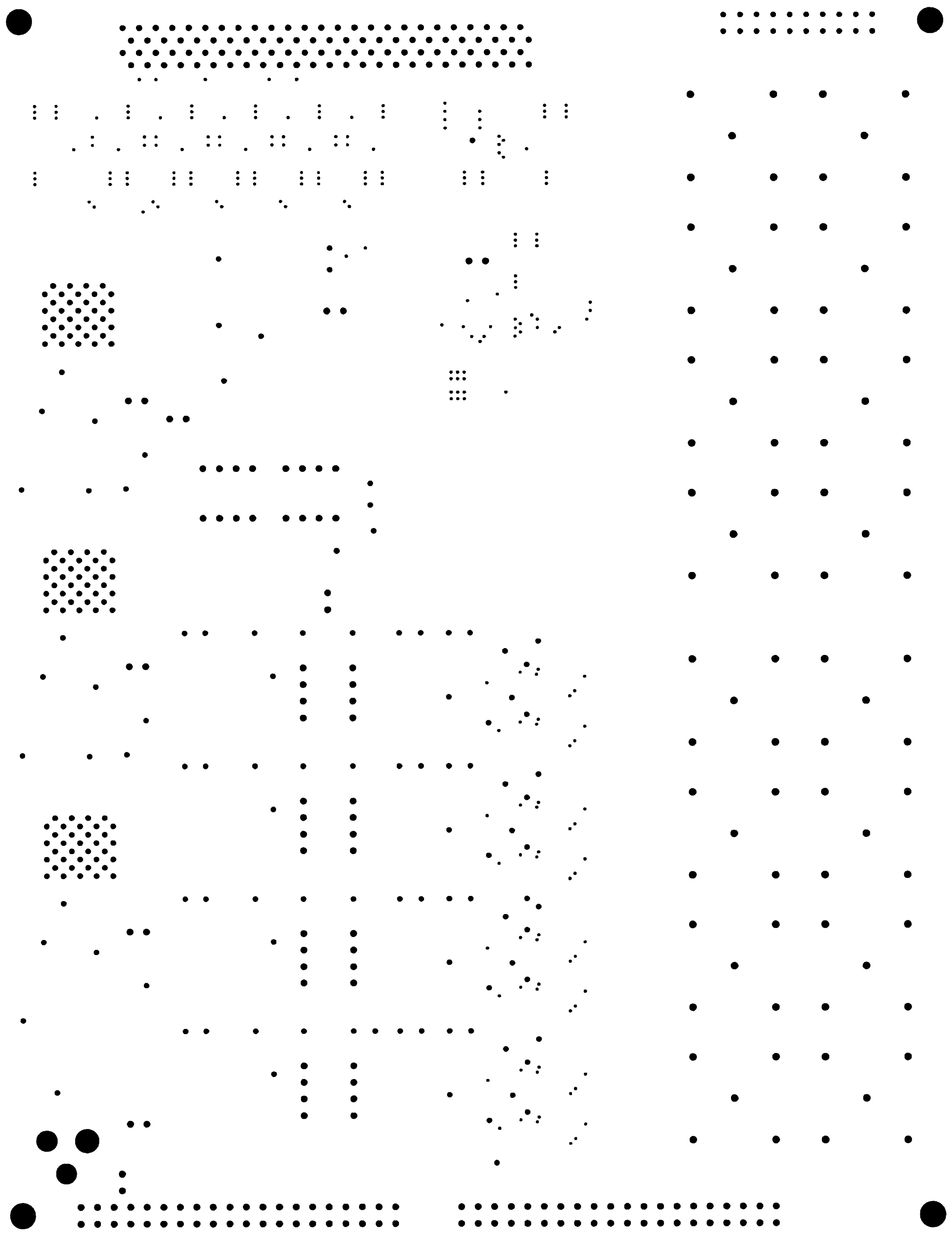


Figure B.8. Motherboard NC Drill Output

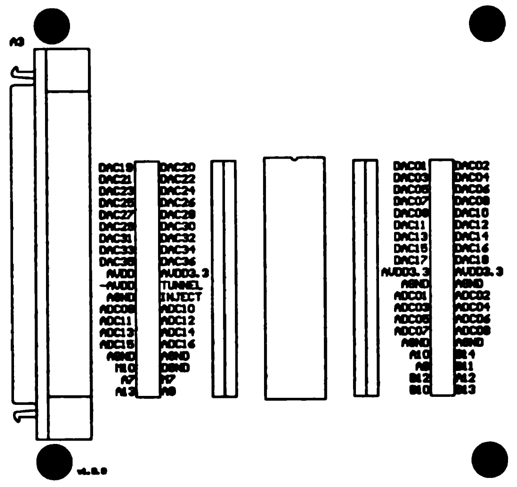


Figure B.9. Gerber Output: Daughterboard Top Silkscreen

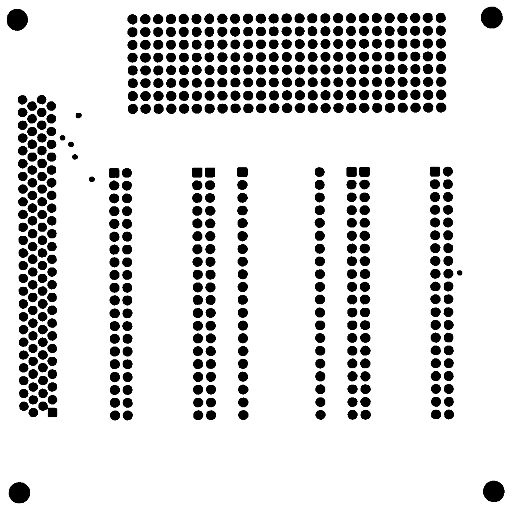


Figure B.10. Gerber Output: Daughterboard Top Solder Mask

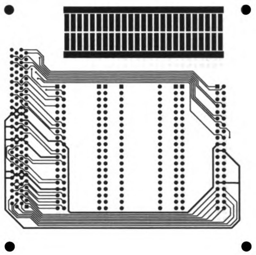


Figure B.11. Gerber Output: Daughterboard Top Layer

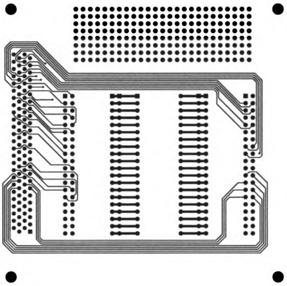


Figure B.12. Gerber Output: Daughterboard Bottom Layer

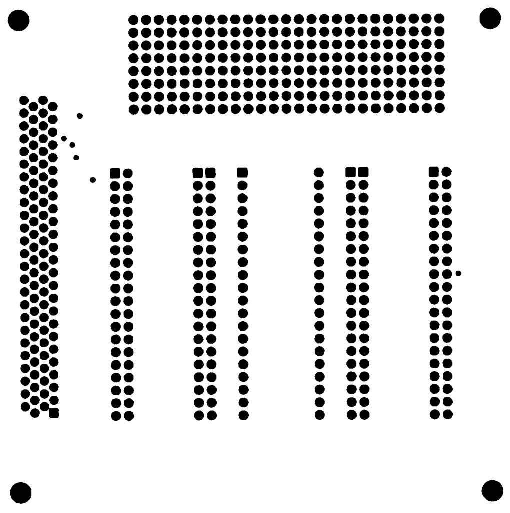


Figure B.13. Gerber Output: Daughterboard Bottom Solder Mask

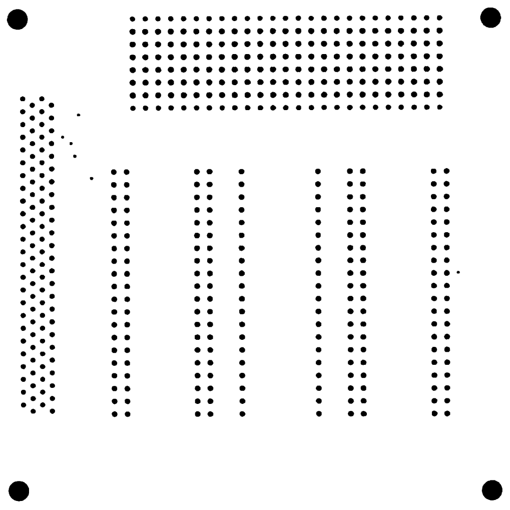


Figure B.14. Daughterboard NC Drill Output

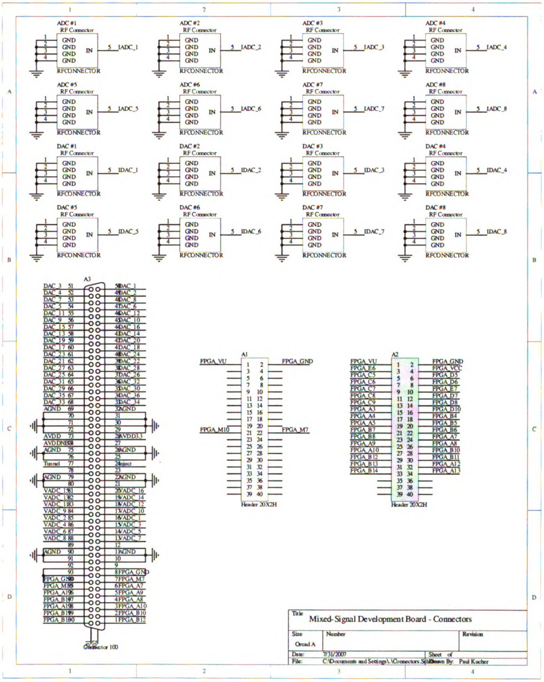


Figure B.15. Test Station Connectors

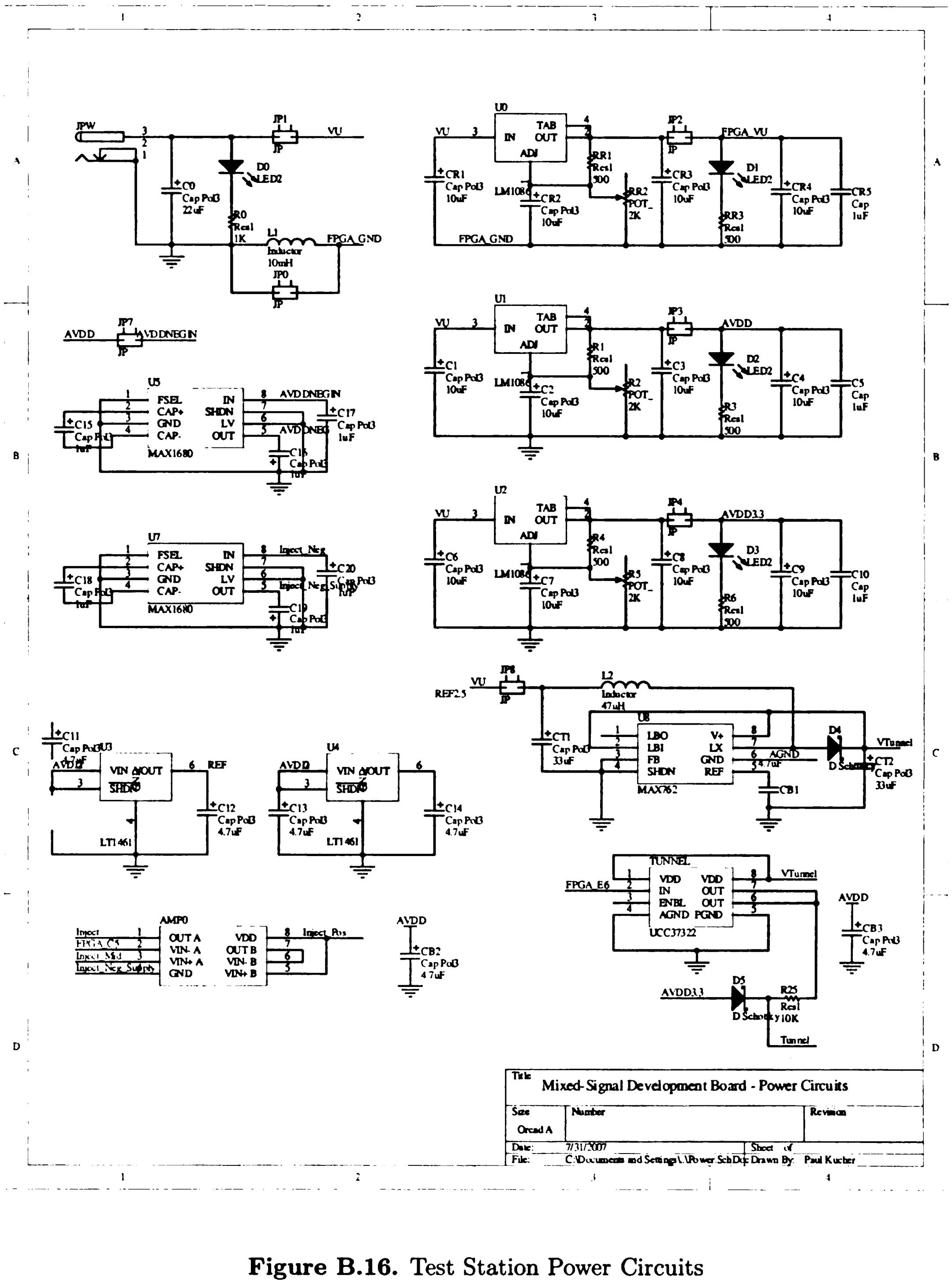


Figure B.16. Test Station Power Circuits

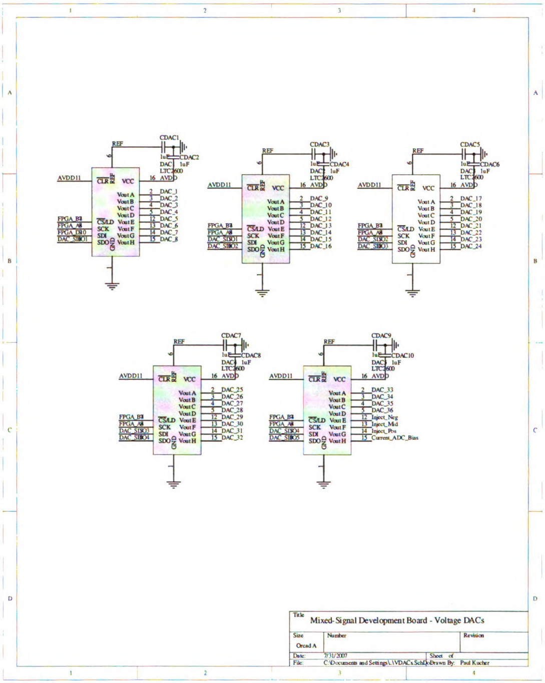


Figure B.17. Test Station Voltage Digital-to-Analog Converters

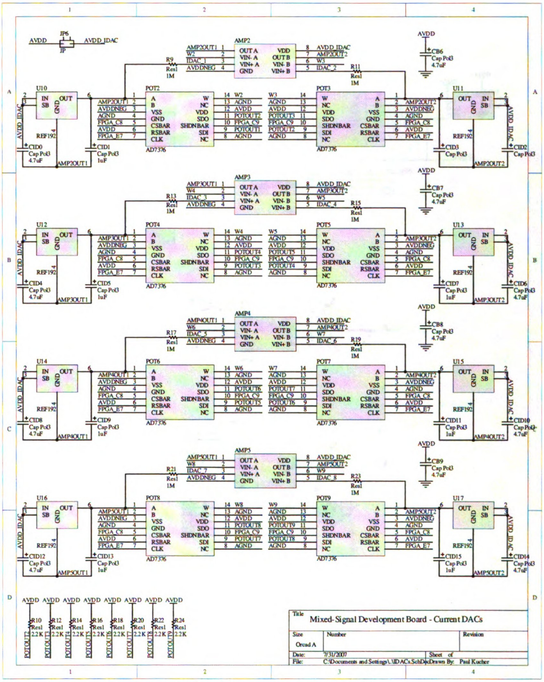


Figure B.18. Test Station Current Digital-to-Analog Converters

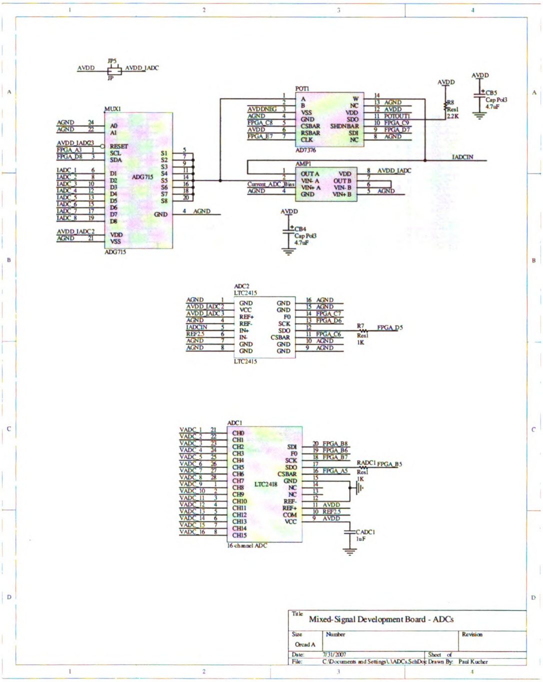


Figure B.19. Test Station Analog-to-Digital Converters

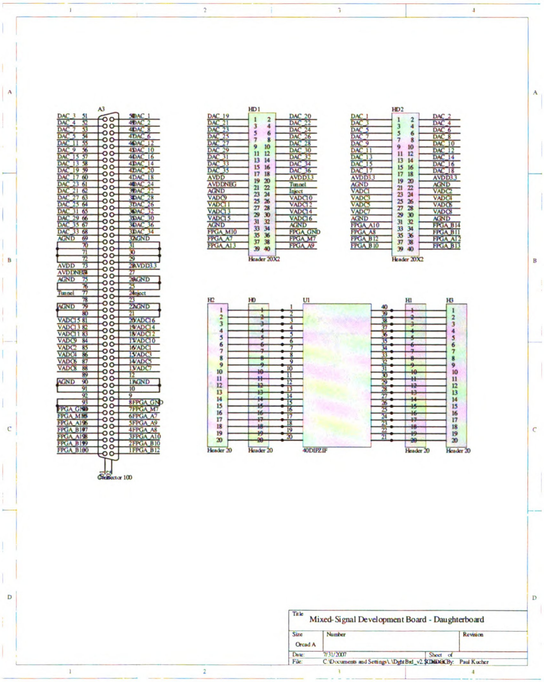


Figure B.20. Test Station Daughterboard

APPENDIX C

Test Station VHDL

C.1 Digital Clock Manager: dcm.vhd

```
------------------------------------------------------------------------
-- Author:        Paul R. Kucher
-- Module Name:   clkmgr - Behavioral
-- Modified:      2007-09-13
-- Description:   The digital clock manager for xc3s200-ft256-4.
------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.Vcomponents.ALL;

entity clkmgr is
Port(
    clkin_in:         in    std_logic;
    clkfx_out:        out   std_logic;
    clkin_ibufg_out:  out   std_logic;
    clk0_out:         out   std_logic
    );
end clkmgr;

architecture Behavioral of clkmgr is
   signal CLKFB_IN: std_logic;
   signal CLKFX_BUF: std_logic;
   signal CLKIN_IBUFG: std_logic;
   signal CLK0_BUF: std_logic;
   signal clk0_out1: std_logic;
   signal GND_BIT: std_logic;
   component BUFG
      port ( I : in    std_logic;
             O : out   std_logic);
   end component;

   component IBUFG
      port ( I : in    std_logic;
             O : out   std_logic);
   end component;

   -- Period Jitter (unit interval) for block DCM_INST = 0.15 UI
   -- Period Jitter (Peak-to-Peak) for block DCM_INST = 0.61 ns
   component DCM
      generic( CLK_FEEDBACK : string := "1X";
               CLKDV_DIVIDE : real := 2.0;
               CLKFX_DIVIDE : integer := 1;
               CLKFX_MULTIPLY : integer := 4;
               CLKIN_DIVIDE_BY_2 : boolean := FALSE;
               CLKIN_PERIOD : real := 10.0;
               CLKOUT_PHASE_SHIFT : string := "NONE";
               DESKEW_ADJUST : string := "SYSTEM_SYNCHRONOUS";
               DFS_FREQUENCY_MODE : string := "LOW";
               DLL_FREQUENCY_MODE : string := "LOW";
               DUTY_CYCLE_CORRECTION : boolean := TRUE;
               FACTORY_JF : bit_vector := x"C080";
               PHASE_SHIFT : integer := 0;
               STARTUP_WAIT : boolean := FALSE;
               DSS_MODE : string := "NONE");
      port ( CLKIN    : in    std_logic;
             CLKFB    : in    std_logic;
             RST      : in    std_logic;
             PSEN     : in    std_logic;
             PSINCDEC : in    std_logic;
             PSCLK    : in    std_logic;
             DSSEN    : in    std_logic;
             CLK0     : out   std_logic;
             CLK90    : out   std_logic;
             CLK180   : out   std_logic;
             CLK270   : out   std_logic;
             CLKDV    : out   std_logic;
             CLK2X    : out   std_logic;
             CLK2X180 : out   std_logic;
             CLKFX    : out   std_logic;
             CLKFX180 : out   std_logic;
             STATUS   : out   std_logic_vector (7 downto 0);
             LOCKED   : out   std_logic;
             PSDONE   : out   std_logic);
   end component;

begin
   GND_BIT <= '0';
   clkin_ibufg_out <= CLKIN_IBUFG;
   clk0_out <= CLKFB_IN;
   CLKFX_BUFG_INST : BUFG
      port map (I=>CLKFX_BUF,
                O=>clkfx_out);

   CLKIN_IBUFG_INST : IBUFG
      port map (I=>clkin_in,
                O=>CLKIN_IBUFG);

   CLK0_BUFG_INST : BUFG
      port map (I=>CLK0_BUF,
                O=>CLKFB_IN);

   CLK0_BUFG_INST1 : BUFG
      port map (I=>CLK0_BUF,
                O=>clk0_out1);

   DCM_INST : DCM
```

```
    generic map( CLK_FEEDBACK => "1X",
             CLKDV_DIVIDE => 2.0,
             CLKFX_DIVIDE => 1,
             CLKFX_MULTIPLY => 5,
             CLKIN_DIVIDE_BY_2 => FALSE,
             CLKIN_PERIOD => 20.000,
             CLKOUT_PHASE_SHIFT => "NONE",
             DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",
             DFS_FREQUENCY_MODE => "HIGH",
             DLL_FREQUENCY_MODE => "LOW",
             DUTY_CYCLE_CORRECTION => TRUE,
             FACTORY_JF => x"8080",
             PHASE_SHIFT => 0,
             STARTUP_WAIT => FALSE)
       port map (CLKFB=>CLKFB_IN,
                 CLKIN=>CLKIN_IBUFG,
                 DSSEN=>GND_BIT,
                 PSCLK=>GND_BIT,
                 PSEN=>GND_BIT,
                 PSINCDEC=>GND_BIT,
                 RST=>GND_BIT,
                 CLKDV=>open,
                 CLKFX=>CLKFX_BUF,
                 CLKFX180=>open,
                 CLK0=>CLK0_BUF,
                 CLK2X=>open,
                 CLK2X180=>open,
                 CLK90=>open,
                 CLK180=>open,
                 CLK270=>open,
                 LOCKED=>open,
                 PSDONE=>open,
                 STATUS=>open);

end Behavioral;
```

C.2 Instruction Decoder and System Controller: decode.vhd

```
--------------------------------------------------------------------------------
-- Author:          Paul R. Kucher
-- Module Name:     decode - Behavioral
-- Modified:        2007-09-13
-- Description:     This main module controls the entire system. It is responsible
--                  for decoding individual instructions coming from RS-232
--                  communications and enables the appropriate sub-modules
--                  to complete a given task.
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity decode is
generic(
    width:                  integer:=16;
    addr:                   integer:=18;
    depth:                  integer:=8
);
Port(
    clk:                    in  std_logic;

    serial_select:          out integer range 0 to 2;
    txd_ready:              out std_logic;
    txd_complete:           in  std_logic;
    rxd_complete:           in  std_logic;
    parallel_txd:           out std_logic_vector(15 downto 0);
    parallel_rxd:           in  std_logic_vector(15 downto 0);

    shift_start_flag:       out std_logic;
    shift_end_flag:         in  std_logic;

    leds:                   out std_logic_vector( 7 downto 0);
    led_data:               out std_logic_vector(15 downto 0);

    from_address:           out std_logic_vector(addr downto 0);
    to_address:             out std_logic_vector(addr downto 0);
    read_block:             out std_logic;
    write_block:            out std_logic;
    mem_data_in:            out std_logic_vector(15 downto 0);
    xfr_op_completed:       in  std_logic;
    module_select:          out integer range 0 to 5;

    vdac_select:            out integer range 0 to 2;
    sig_gen_enable:         out std_logic;
    sig_gen_complete:       in  std_logic;

    program_dac:            out std_logic;
    dac_programmed:         in  std_logic;
    dac_instruction:        out std_logic_vector(27 downto 0);

    adc_conv_mode:          out std_logic_vector(1 downto 0);
    adc_address:            out std_logic_vector(3 downto 0);
    adc_data_ready:         in  std_logic;
    adc_data_collect:       out std_logic;

    iadc_data_ready:        in  std_logic;
    iadc_data_collect:      out std_logic;

    inject_pulse:           out std_logic;
    pulse_injected:         in  std_logic;
    injection_pulse_width:  out std_logic_vector(7 downto 0);

    tunnel_pulse:           out std_logic;

    io_instruction:         out std_logic_vector(5 downto 0);
    io_update:              out std_logic;
    io_updated:             in  std_logic;
    io_output:              in  std_logic;

    digit_pot_number:       out std_logic_vector(4 downto 1);
    digit_pot_value:        out std_logic_vector(7 downto 1);
    digit_pot_update:       out std_logic;
    digit_pot_updated:      in  std_logic
);
end decode;

architecture Behavioral of decode is

begin

leds(7) <= '0'; leds(6) <= '0'; leds(5) <= '0'; leds(4) <= '0';
leds(3) <= '0'; leds(2) <= '0'; leds(1) <= '0'; leds(0) <= '0';

-- This is the instruction fetch and decode process. It takes in 16-bit
-- data from the RS232 bus and then decides whether to write to memory.
-- send data to the chip, read from memory. or send data back to the PC.
decode_and_execute: process( clk )
variable current_instruction: std_logic_vector(15 downto 0)
        := "0000000000000000";
variable execute_state: integer range 0 to 50 := 0;
variable inst_pointer, inst_pointer_last: integer range 0 to 100 := 0;
variable msb_addr, msb_addr2: std_logic_vector(2 downto 0) := "000";
variable read_or_write_block, new_instruction: std_logic := '0';
variable dac_command: std_logic_vector(11 downto 0);

begin
  if clk'event and clk = '1' then

    if rxd_complete = '1' then -- fetch instruction
      led_data <= std_logic_vector( parallel_rxd );
      current_instruction := parallel_rxd;
      inst_pointer := inst_pointer + 1;
    end if;

    if inst_pointer > inst_pointer_last then
      inst_pointer_last := inst_pointer;
      new_instruction := '1';
    else
      new_instruction := '0';
    end if;

    case execute_state is
      when 0 => -- Idle State
        if new_instruction = '1' then
          execute_state := 1;
        end if;
        txd_ready         <= '0';
        write_block       <= '0';
        read_block        <= '0';
        program_dac       <= '0';
        io_update         <= '0';
        adc_data_collect  <= '0';
        iadc_data_collect <= '0';
        digit_pot_update  <= '0';
        shift_start_flag  <= '0';
        module_select     <= 0;
        serial_select     <= 1;
        vdac_select       <= 1;
      when 1 => -- Decode Instruction
        if    current_instruction(15 downto 11) = "0000" then
          execute_state :=  3; -- Loopback
        elsif current_instruction(15 downto 12) = "0001" then
          execute_state :=  5; -- Memory Transfer
        elsif current_instruction(15 downto 12) = "0010" then
          execute_state :=  9; -- Voltage DAC
        elsif current_instruction(15 downto 12) = "0011" then
          execute_state := 11; -- Voltage ADC
        elsif current_instruction(15 downto 12) = "0100" then
          execute_state := 14; -- Current DAC
        elsif current_instruction(15 downto 12) = "0101" then
          execute_state := 16; -- Current ADC
        elsif current_instruction(15 downto 12) = "0110" then
          execute_state := 21; -- EEPROM Injection
        elsif current_instruction(15 downto 12) = "0111" then
          execute_state := 23; -- EEPROM Tunneling
        elsif current_instruction(15 downto 12) = "1000" then
          execute_state := 24; -- Digital IO
        elsif current_instruction(15 downto 12) = "1001" then
          execute_state := 26; -- Signal Generation
        elsif current_instruction(15 downto 12) = "1010" then
          execute_state := 30; -- Serial Shift Chain
        else  execute_state := 2;
        end if;
      when 2 => -- Reset Instruction Pointer
        inst_pointer_last := 0;
        inst_pointer := 0;
        execute_state := 0;
      when 3 => -- Transfer data in current instruction over RS-232
        parallel_txd <= current_instruction;
        txd_ready <= '1';
        execute_state := 4;
      when 4 => -- Transfer data continued
        txd_ready <= '0';
        if txd_complete = '1' then
          execute_state := 2;
        end if;
      when 5 => -- Memory Transfer (Block/Single Read & Write)
        module_select <= 2;
        serial_select <= 2;
        if new_instruction = '1' then -- Receive Starting Address
          from_address <= msb_addr & current_instruction;
          execute_state := 6;
        else
          msb_addr  := current_instruction(2 downto 0); -- 17-19th bits of
          msb_addr2 := current_instruction(5 downto 3); -- Address Reg.
          read_or_write_block := current_instruction(10);
        end if;
      when 6 =>
        if new_instruction = '1' then -- Receive Ending Address
          to_address <= msb_addr2 & current_instruction;
          execute_state := 7;
        end if;
      when 7 =>
        if read_or_write_block = '1' and new_instruction = '1' then
          write_block <= '1';
          mem_data_in <= current_instruction;
          execute_state := 8;
        elsif read_or_write_block = '0' then
          read_block <= '1';
          execute_state := 8;
        end if;
      when 8 =>
        write_block <= '0';
        read_block  <= '0';
        if xfr_op_completed = '1' then execute_state := 2;
        end if;
      when 9 => -- Voltage DAC
        if new_instruction = '1' then
          dac_instruction <= dac_command & current_instruction;
          program_dac <= '1';
          execute_state := 10;
        else
          dac_command := current_instruction(11 downto 0);
        end if;
      when 10 =>
```

```
          program_dac <= '0';
          if dac_programmed = '1' then execute_state := 3; end if;
        when 11 => -- Voltage ADC
          module_select <= 3;
          if new_instruction = '1' then -- Receive Starting Address
            from_address <= msb_addr & current_instruction;
            execute_state := 12;
          else
            msb_addr  := current_instruction(2 downto 0); -- 17-19th bits of
            msb_addr2 := current_instruction(5 downto 3); -- Address Reg.
            adc_address <= current_instruction(11 downto 8);
            adc_conv_mode <= current_instruction(7 downto 6);
          end if;
        when 12 =>
          if new_instruction = '1' then -- Receive Ending Address
            to_address <= msb_addr2 & current_instruction;
            adc_data_collect <= '1';
            execute_state := 13;
          end if;
        when 13 =>
          adc_data_collect <= '0';
          if adc_data_ready = '1' then
            current_instruction := "0000000000000000";
            execute_state := 3;
          end if;
        when 14 => -- Current DAC
          digit_pot_number <= current_instruction(11 downto 8);
          digit_pot_value  <= current_instruction(6 downto 0);
          digit_pot_update <= '1';
          execute_state := 15;
        when 15 =>
          digit_pot_update <= '0';
          if digit_pot_updated = '1' then
            current_instruction := "0000000000000000";
            execute_state := 3;
          end if;
        when 16 => -- Current ADC
          module_select <= 4;
          if new_instruction = '1' then -- Receive Starting Address
            from_address <= msb_addr & current_instruction;
            execute_state := 17;
          else
            msb_addr  := current_instruction(2 downto 0); -- 17-19th bits of
            msb_addr2 := current_instruction(5 downto 3); -- Address Reg.
            adc_address <= current_instruction(11 downto 8);
          end if;
        when 17 =>
          if new_instruction = '1' then -- Receive Ending Address
            to_address <= msb_addr2 & current_instruction;
            execute_state := 18;
          end if;
        when 18 =>
          if new_instruction = '1' then -- Receive Gain Update
            digit_pot_number <= "0000";
            digit_pot_value <= current_instruction(6 downto 0);
            if current_instruction(15) = '1' then -- Only Update on Request
              digit_pot_update <= '1';
              execute_state := 19;
            else
              iadc_data_collect <= '1';
              execute_state := 20;
            end if;
```

```
    end if;
  when 19 =>
    digit_pot_update <= '0';
    if digit_pot_updated = '1' then
      iadc_data_collect <= '1';
      execute_state := 20;
    end if;
  when 20 =>
    iadc_data_collect <= '0';
    if iadc_data_ready = '1' then
      current_instruction := "0000000000000000";
      execute_state := 3;
    end if;
  when 21 => -- EEPROM Injection
    injection_pulse_width <= current_instruction(7 downto 0);
    inject_pulse <= '1';
    execute_state := 22;
  when 22 =>
    inject_pulse <= '0';
    current_instruction := "0000000000000000";
    if pulse_injected = '1' then execute_state := 3; end if;
  when 23 => -- EEPROM Tunneling
    tunnel_pulse <= current_instruction(0);
    current_instruction := "0000000000000000";
    execute_state := 3;
  when 24 => -- Digital I/O
    io_instruction <= current_instruction(11 downto 10) &
      current_instruction(3 downto 0);
    io_update <= '1';
    execute_state := 25;
  when 25 =>
    io_update <= '0';
    if io_updated = '1' then
      current_instruction := "000000000000000" & io_output;
      execute_state := 3;
    end if;
  when 26 => -- Signal Generation
    vdac_select <= 2;
    module_select <= 5;
    if new_instruction = '1' then -- Receive Starting Address
      from_address <= msb_addr & current_instruction;
      execute_state := 27;
    else
      msb_addr  := current_instruction(2 downto 0); -- 17-19th bits of
      msb_addr2 := current_instruction(5 downto 3); -- Address Reg.
    end if;
  when 27 =>
    if new_instruction = '1' then -- Receive Ending Address
      to_address <= msb_addr2 & current_instruction;
      sig_gen_enable <= '1';
      execute_state := 28;
    end if;
  when 28 =>
    if new_instruction = '1' then -- Receive Stop Condition
      sig_gen_enable <= '0';
      execute_state := 29;
    end if;
  when 29 =>
    if sig_gen_complete = '1' then
      current_instruction := "0000000000000000";
      execute_state := 3;
    end if;
```

```
      when 30 => -- Program Serial Chain
        module_select <= 1;
        if new_instruction = '1' then -- Receive Starting Address
          from_address <= msb_addr & current_instruction;
          execute_state := 31;
        else
          msb_addr  := current_instruction(2 downto 0); -- 17-19th bits of
          msb_addr2 := current_instruction(5 downto 3); -- Address Reg.
        end if;
      when 31 =>
        if new_instruction = '1' then -- Receive Ending Address
          to_address <= msb_addr2 & current_instruction;
          shift_start_flag <= '1';
          execute_state := 32;
        end if;
      when 32 =>
        shift_start_flag <= '0';
        if shift_end_flag = '1' then
          current_instruction := "0000000000000000";
          execute_state := 3;
        end if;
      when others => execute_state := 0;
    end case;
  end if;
end process;

end Behavioral;
```

C.3 Multi-channel Digital Input/Output: digital_io.vhd

```
--------------------------------------------------------------------------------
-- Author:        Paul R. Kucher
-- Module Name:   digital_io - Behavioral
-- Modified:      2007-09-05
-- Description:   This module allows easy access to the FPGA's I/Os from the
--                remaining pins on the FPGA development board. From Matlab,
--                the direction of each pin may be set and data may be read
--                or written to each of these ports.
--
--                If the main decoder unit is modified to accommodate additional
--                functionality utilizing one of these I/Os, be sure remove the
--                pins from the UCF file accordingly.
--
--                Pins are currently assigned as follows:
--                digital_io(0)   FPGA Pin M10
--                digital_io(1)   FPGA Pin A7
--                digital_io(2)   FPGA Pin M7
--                digital_io(3)   FPGA Pin A13
--                digital_io(4)   FPGA Pin A9
--                digital_io(5)   FPGA Pin A10
--                digital_io(6)   FPGA Pin B14
--                digital_io(7)   FPGA Pin A8
--                digital_io(8)   FPGA Pin B11
--                digital_io(9)   FPGA Pin B12
--                digital_io(10)  FPGA Pin A12
--                digital_io(11)  FPGA Pin B10
--                digital_io(12)  FPGA Pin B13
--------------------------------------------------------------------------------
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity digital_io is
Port(
    clk:          in     std_logic; -- FPGA clock
    digital_ios: inout std_logic_vector(12 downto 0); -- FPGA I/O Pins

    instruction: in     std_logic_vector(5 downto 0);
    io_update:   in     std_logic; -- Flag to read instruction
    io_updated:  out    std_logic; -- Flag indicating instruction executed
    output:      out    std_logic  -- Output Data (If Applicable)
    );
end digital_io;

architecture Behavioral of digital_io is

begin

state_machine: process( clk )
variable io_state: integer range 0 to 7 := 0;
variable io_set: integer range 0 to 40 := 0;
begin
  if clk'event and clk = '1' then
    case io_state is
      when 0 => -- Initialization State
        digital_ios <= (digital_ios'range => 'Z');
        io_state := 1;
      when 1 => -- Idle State
        if io_update = '1' then
          io_state := 2;
        end if;
        io_set := 0;
        io_updated <= '0';
      when 2 => -- Make High Impedance
        if    instruction(3 downto 0) = "0000" then io_set :=  1;
        elsif instruction(3 downto 0) = "0001" then io_set :=  2;
        elsif instruction(3 downto 0) = "0010" then io_set :=  3;
        elsif instruction(3 downto 0) = "0011" then io_set :=  4;
        elsif instruction(3 downto 0) = "0100" then io_set :=  5;
        elsif instruction(3 downto 0) = "0101" then io_set :=  6;
        elsif instruction(3 downto 0) = "0110" then io_set :=  7;
        elsif instruction(3 downto 0) = "0111" then io_set :=  8;
        elsif instruction(3 downto 0) = "1000" then io_set :=  9;
        elsif instruction(3 downto 0) = "1001" then io_set := 10;
        elsif instruction(3 downto 0) = "1010" then io_set := 11;
        elsif instruction(3 downto 0) = "1011" then io_set := 12;
        elsif instruction(3 downto 0) = "1100" then io_set := 13;
        end if;
        io_state := 3;
      when 3 => -- Decode
        if instruction(5) = '1' then -- Set I/O
          if    instruction(3 downto 0) = "0000" then io_set := 14;
          elsif instruction(3 downto 0) = "0001" then io_set := 15;
          elsif instruction(3 downto 0) = "0010" then io_set := 16;
          elsif instruction(3 downto 0) = "0011" then io_set := 17;
          elsif instruction(3 downto 0) = "0100" then io_set := 18;
          elsif instruction(3 downto 0) = "0101" then io_set := 19;
          elsif instruction(3 downto 0) = "0110" then io_set := 20;
          elsif instruction(3 downto 0) = "0111" then io_set := 21;
          elsif instruction(3 downto 0) = "1000" then io_set := 22;
          elsif instruction(3 downto 0) = "1001" then io_set := 23;
          elsif instruction(3 downto 0) = "1010" then io_set := 24;
          elsif instruction(3 downto 0) = "1011" then io_set := 25;
          elsif instruction(3 downto 0) = "1100" then io_set := 26;
          end if;
          output <= instruction(4);
        else -- Read I/O
          if    instruction(3 downto 0) = "0000" then io_set := 27;
          elsif instruction(3 downto 0) = "0001" then io_set := 28;
          elsif instruction(3 downto 0) = "0010" then io_set := 29;
          elsif instruction(3 downto 0) = "0011" then io_set := 30;
          elsif instruction(3 downto 0) = "0100" then io_set := 31;
          elsif instruction(3 downto 0) = "0101" then io_set := 32;
          elsif instruction(3 downto 0) = "0110" then io_set := 33;
          elsif instruction(3 downto 0) = "0111" then io_set := 34;
          elsif instruction(3 downto 0) = "1000" then io_set := 35;
          elsif instruction(3 downto 0) = "1001" then io_set := 36;
          elsif instruction(3 downto 0) = "1010" then io_set := 37;
          elsif instruction(3 downto 0) = "1011" then io_set := 38;
          elsif instruction(3 downto 0) = "1100" then io_set := 39;
          end if;
        end if;
        io_state := 4;
      when 4 =>
        io_updated <= '1';
        io_state := 1;
      when others => io_state := 0;
    end case;

    case io_set is
      when  0 => output <= instruction(4);
      when  1 => digital_ios(0)  <= 'Z';
      when  2 => digital_ios(1)  <= 'Z';
      when  3 => digital_ios(2)  <= 'Z';
      when  4 => digital_ios(3)  <= 'Z';
      when  5 => digital_ios(4)  <= 'Z';
      when  6 => digital_ios(5)  <= 'Z';
      when  7 => digital_ios(6)  <= 'Z';
      when  8 => digital_ios(7)  <= 'Z';
      when  9 => digital_ios(8)  <= 'Z';
      when 10 => digital_ios(9)  <= 'Z';
      when 11 => digital_ios(10) <= 'Z';
      when 12 => digital_ios(11) <= 'Z';
      when 13 => digital_ios(12) <= 'Z';
      when 14 => digital_ios(0)  <= instruction(4);
      when 15 => digital_ios(1)  <= instruction(4);
      when 16 => digital_ios(2)  <= instruction(4);
      when 17 => digital_ios(3)  <= instruction(4);
      when 18 => digital_ios(4)  <= instruction(4);
      when 19 => digital_ios(5)  <= instruction(4);
      when 20 => digital_ios(6)  <= instruction(4);
      when 21 => digital_ios(7)  <= instruction(4);
      when 22 => digital_ios(8)  <= instruction(4);
      when 23 => digital_ios(9)  <= instruction(4);
      when 24 => digital_ios(10) <= instruction(4);
      when 25 => digital_ios(11) <= instruction(4);
      when 26 => digital_ios(12) <= instruction(4);

      when 27 => output <= digital_ios(0);
      when 28 => output <= digital_ios(1);
      when 29 => output <= digital_ios(2);
      when 30 => output <= digital_ios(3);
```

```
        when 31 => output <= digital_ios(4);
        when 32 => output <= digital_ios(5);
        when 33 => output <= digital_ios(6);
        when 34 => output <= digital_ios(7);
        when 35 => output <= digital_ios(8);
        when 36 => output <= digital_ios(9);
        when 37 => output <= digital_ios(10);
        when 38 => output <= digital_ios(11);
        when 39 => output <= digital_ios(12);
        when others => null;
      end case;
    end if;
  end process;

end Behavioral;
```

C.4 Digital Potentiometer Control: digital_pots.vhd

```
-----------------------------------------------------------------------------------
-- Author:          Paul R. Kucher
-- Module Name:     digital_pots - Behavioral
-- Modified:        2007-07-13
-- Description:     This module controls the nine digital potentiometers on the
--                  mixed-signal test board. It takes in the value and channel
--                  number and updates the chain accordingly
-----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity digital_pots is
Port(
    clk:                in  std_logic; -- FPGA Clock
    digit_pot_clk:      out std_logic; -- Potentiometer Serial Clock
    digit_pot_sdi:      out std_logic; -- Potentiometer Serial Data Input
    digit_pot_cs_bar:   out std_logic; -- Shift Enable
    digit_pot_shdn_bar: out std_logic; -- Shutdown Signal

    digit_pot_number:   in  std_logic_vector(4 downto 1); -- Pot Number
    digit_pot_value:    in  std_logic_vector(7 downto 1); -- 7-bit Pot Value
    digit_pot_update:   in  std_logic; -- Potentiometer Update Flag
    digit_pot_updated:  out std_logic  -- Digital Potentiometer Updated Flag
    );
end digital_pots;

architecture Behavioral of digital_pots is

begin

program_pots: process( clk )
variable count: integer range 0 to 127 := 0;
variable shift_count: integer range 0 to 70 := 0;
variable pot_state: integer range 0 to 2 := 0;
variable pot_chain: std_logic_vector(63 downto 1)
        := "100000010000001000000100000010000001000000100000010000001000000";
begin

if clk'event and clk = '1' then

if count > 64 then digit_pot_clk <= '1';
else digit_pot_clk <= '0';
end if;

case pot_state is
  when 0 => -- Idle State
    digit_pot_cs_bar <= '1';
    digit_pot_sdi <= '0';
    digit_pot_shdn_bar <= '1';
    digit_pot_updated <= '0';
    shift_count := 0;
    count := 0;
    pot_state := 1;
  when 1 =>
    if digit_pot_update = '1' then
      if    digit_pot_number = "0000" then -- IADC Potentiometer
        pot_chain( 7 downto  1) := digit_pot_value;
      elsif digit_pot_number = "0001" then -- IDAC Channel #1
        pot_chain(14 downto  8) := digit_pot_value;
      elsif digit_pot_number = "0010" then -- IDAC Channel #2
        pot_chain(21 downto 15) := digit_pot_value;
      elsif digit_pot_number = "0011" then -- IDAC Channel #3
        pot_chain(28 downto 22) := digit_pot_value;
      elsif digit_pot_number = "0100" then -- IDAC Channel #4
        pot_chain(35 downto 29) := digit_pot_value;
      elsif digit_pot_number = "0101" then -- IDAC Channel #5
        pot_chain(42 downto 36) := digit_pot_value;
      elsif digit_pot_number = "0110" then -- IDAC Channel #6
        pot_chain(49 downto 43) := digit_pot_value;
      elsif digit_pot_number = "0111" then -- IDAC Channel #7
        pot_chain(56 downto 50) := digit_pot_value;
      elsif digit_pot_number = "1000" then -- IDAC Channel #8
        pot_chain(63 downto 57) := digit_pot_value;
      end if;
      pot_state := 2;
    end if;
  when 2 =>
    if shift_count < 1 then
      digit_pot_sdi <= '0';
      digit_pot_cs_bar <= '0';
    elsif shift_count < 64 then
      digit_pot_sdi <= pot_chain(63);
    elsif shift_count < 65 then
      digit_pot_sdi <= '0';
      digit_pot_cs_bar <= '1';
    else
      digit_pot_updated <= '1';
      shift_count := 0;
      pot_state := 0;
    end if;

    count := count + 1;

    if count = 127 then
      count := 0;
      if shift_count < 64 and shift_count > 0 then
        pot_chain(63 downto 1) := pot_chain(62 downto 1) & pot_chain(63);
      end if;
      shift_count := shift_count + 1;
    end if;
  when others => pot_state := 0;
  end case;
end if;

end process;

end Behavioral;
```

C.5 Current ADC Control: iadc.vhd

```
--------------------------------------------------------------------------------
-- Author:          Paul R. Kucher
-- Module Name:     iadc - Behavioral
-- Modified:        2007-09-07
-- Description:     This module controls the 8 channel current ADC. It controls
--                  the single channel LTC 2415 ADC as well as the ADG 715
--                  analog multiplexer.
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity iadc is
generic(
    width:              integer := 16;
    addr:               integer := 18;
    depth:              integer := 8
);
Port(
    clk:                in    std_logic; -- FPGA clock
    iadc_clk:           out   std_logic; -- Conversion Clock (f0)
    iadc_sck:           out   std_logic; -- Serial Clock for ADC
    iadc_cs_bar:        out   std_logic; -- Serial Transfer Enable
    iadc_sdo:           in    std_logic; -- Serial Data Out (of ADC)
    iadc_scl:           out   std_logic; -- Serial Clock for Multiplexer
    iadc_sda:           inout std_logic; -- Serial I/O for Multiplexer

    iadc_address:       in    std_logic_vector(3 downto 0); -- Channel Address
    iadc_start_address: in    std_logic_vector(addr downto 0); -- St. Data Store
    iadc_end_address:   in    std_logic_vector(addr downto 0); -- End Data Store
    iadc_data_ready:    out   std_logic; -- Data Conversion Complete Flag
    iadc_data_collect:  in    std_logic; -- Control flag to initiate sampling

    read_control:       out   std_logic; -- Instruct module to read data
    write_control:      out   std_logic; -- Instruct module to write data
    mem_op_completed:   in    std_logic; -- Flag if memory operation completed
    address:            out   std_logic_vector(addr downto 0); -- Address to R/W
    memory_data_write:  out   std_logic_vector(width-1 downto 0) -- Data to write
    );
end iadc;

architecture Behavioral of iadc is
signal address_register: std_logic_vector(7 downto 0);
signal sdo_filtered: std_logic := '0';
signal sdo_sync: std_logic_vector(1 downto 0) := "00";
begin

address_encode: process( iadc_address )

```

```
begin
  case iadc_address is
    when "0000" => address_register <= "00000001";
    when "0001" => address_register <= "00000010";
    when "0010" => address_register <= "00000100";
    when "0011" => address_register <= "00001000";
    when "0100" => address_register <= "00010000";
    when "0101" => address_register <= "00100000";
    when "0110" => address_register <= "01000000";
    when "0111" => address_register <= "10000000";
    when others => address_register <= "00000000";
  end case;
end process;

sdo_sync_input: process( clk )
begin
  if clk'event and clk = '1' then
    sdo_sync <= sdo_sync(0) & iadc_sdo;
  end if;
end process;

-- Filter the incoming data for any glitches.
sdo_filter_input: process( clk )
variable count : integer range 0 to 7 := 5;
begin
  if clk'event and clk = '1' then
    if sdo_sync(1) = '1' and count < 5 then count := count + 1;
    elsif sdo_sync(1) = '0' and count > 0 then count := count - 1;
    end if;

    if count = 5 then sdo_filtered <= '1';
    elsif count = 0 then sdo_filtered <= '0';
    end if;
  end if;
end process;

state_machine: process( clk )
variable iadc_state: integer range 0 to 9 := 0;
variable count, start_count: integer range 0 to 500 := 0;
variable pause_count: integer range 0 to 50001 := 0;
variable adc_clk_count: integer range 0 to 100 := 0;
variable sdo_parallel: std_logic_vector(31 downto 0);
variable mux_chain: std_logic_vector(17 downto 0);
variable mux_count: integer range 0 to 127 := 0;
variable current_address: std_logic_vector(addr downto 0);
variable shift_count: integer range 0 to 40 := 0;
variable mux_shift_count: integer range 0 to 20 := 0;
variable conversion_complete: std_logic := '0';
--variable startup_count: integer range 0 to 15 := 0;
variable startup_complete: std_logic := '0';
begin
  if clk'event and clk = '1' then

--      if startup_count < 10 then
--        startup_count := startup_count + 1;
--        startup_complete := '0';
--      else
--        startup_complete := '1';
--      end if;

    if adc_clk_count > 50 then iadc_clk <= '1';
    else iadc_clk <= '0';
```

```
      end if;

      if adc_clk_count = 100 then adc_clk_count := 0;
      else adc_clk_count := adc_clk_count + 1;
      end if;

      if count > 200 then iadc_sck <= '1';
      else iadc_sck <= '0';
      end if;

      if (mux_count < 32 or mux_count > 96) and iadc_state = 2 then
        iadc_scl <= '0';
      elsif start_count > 20 and iadc_state = 1 then iadc_scl <= '0';
      else iadc_scl <= '1';
      end if;

      if pause_count < 50000 then
        pause_count := pause_count + 1;
      end if;

      if count = 400 or iadc_state /= 4 then count := 0;
      else count := count + 1;
      end if;

      if mux_count = 127 or iadc_state /= 2 then mux_count := 0;
      else mux_count := mux_count + 1;
      end if;

      case iadc_state is
        when 0 => -- Idle state
          read_control <= '0';
          write_control <= '0';
          iadc_cs_bar <= '1';
          shift_count := 0;
          mux_shift_count := 0;
          iadc_sda <= '1';
          iadc_data_ready <= '0';
          conversion_complete := '0';
          current_address := iadc_start_address;
          if iadc_data_collect = '1' and startup_complete = '1' then
            mux_chain := "100100000" & address_register & "0";
            start_count := 0;
            iadc_sda <= '0';
            iadc_state := 1;
          end if;
        when 1 => -- Multiplexer Write Start Condition
          if start_count > 45 then
            start_count := 0;
            iadc_state := 2;
          else
            start_count := start_count + 1;
          end if;
        when 2 => -- Update Multiplexer
          if mux_shift_count < 18 then
            if mux_count = 1 then
              iadc_sda <= mux_chain(17);
            elsif mux_count = 127 then
              mux_chain(17 downto 1) := mux_chain(16 downto 0);
              mux_shift_count := mux_shift_count + 1;
            end if;
          else
            iadc_sda <= '0';
          iadc_cs_bar <= '0';
          iadc_state := 3;
        end if;
      when 3 =>
        if start_count > 45 then
          iadc_sda <= '1';
          start_count := 0;
          iadc_state := 4;
        else
          start_count := start_count + 1;
        end if;
      when 4 => -- Shift Out ADC Data
        if shift_count < 32 then
          if count = 200 then
            sdo_parallel := sdo_parallel(30 downto 0) & sdo_filtered;
          elsif count = 390 then
            shift_count := shift_count + 1;
          end if;
        else
          pause_count := 0;
          iadc_state := 5;
        end if;
      when 5 => -- Conversion Waiting Period
        shift_count := 0;
        if sdo_filtered = '0' and pause_count = 50000 then -- Conv. Complete
          if conversion_complete = '0' then
            conversion_complete := '1';
            iadc_state := 4;
          else -- At least one conversion completed
            address <= current_address;
            memory_data_write <= sdo_parallel(31 downto 16);
            write_control <= '1';
            iadc_state := 6;
          end if;
        end if;
      when 6 =>
        write_control <= '0';
        current_address := current_address + 1;
        iadc_state := 7;
      when 7 =>
        if mem_op_completed = '1' then
          address <= current_address;
          memory_data_write <= sdo_parallel(15 downto 0);
          write_control <= '1';
          iadc_state := 8;
        end if;
      when 8 =>
        write_control <= '0';
        current_address := current_address + 1;
        iadc_state := 9;
      when 9 =>
        if mem_op_completed = '1' then
          if current_address > iadc_end_address then
            iadc_data_ready <= '1';
            conversion_complete := '0';
            iadc_state := 0;
          else
            iadc_state := 4;
          end if;
        end if;
      when others => iadc_state := 0;
    end case;

    if startup_complete = '0' then
      startup_complete := '1';
    end if;
  end if;
end process;

end Behavioral;
```

C.6 Floating-Gate Transistor Injection: injection.vhd

```
--------------------------------------------------------------------------------
-- Author:        Paul R. Kucher
-- Module Name:   injection - Behavioral
-- Modified:      2007-07-29
-- Description:   Hot electron injection is accomplished by providing a large
--                source to drain pulse to a PMOS transistor. This module
--                controls the pulse width that drives the external injection
--                circuit.
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity injection is
Port(
    clk:                   in  std_logic; -- FPGA clock
    inject:                out std_logic; -- Injection pulse

    inject_pulse:          in  std_logic; -- Inject pulse control bit
    pulse_injected:        out std_logic; -- Pulse operation completed flag
    injection_pulse_width: in  std_logic_vector(7 downto 0) -- P/W register
);
end injection;

architecture Behavioral of injection is

begin

injection_pulse: process( clk )

variable injection_state: integer range 0 to 2 := 0;
variable injection_pulse_count: std_logic_vector(7 downto 0);
variable injection_register, injection_compare: std_logic_vector(29 downto 0);

begin
  if clk'event and clk = '1' then
    case injection_state is -- injection pulse state machine
      when 0 => -- idle state
        pulse_injected <= '0';
        inject <= '0';
        injection_register := "000000000000000000000000000000";
        injection_compare  := "000000000000000000000000000001";
        injection_pulse_count := "00000001";
        if inject_pulse = '1' then injection_state := 1; end if;
      when 1 => -- pulse width setup
        if injection_pulse_width < injection_pulse_count then
          injection_state := 2;
        else
          injection_compare := injection_compare(28 downto 0) &
          injection_compare(29);
          injection_pulse_count := injection_pulse_count + "00000001";
        end if;
      when 2 => -- injection state
        if injection_register < injection_compare then
          inject <= '1';
          injection_register := injection_register +
          "000000000000000000000000000001";
        else -- return to idle state
          inject <= '0';
          pulse_injected <= '1';
          injection_state := 0;
        end if;
      when others => injection_state := 0;
    end case;
  end if;
end process;

end Behavioral;
```

C.7 Memory Transfer Control: memory_block_transfer.vhd

```
-----------------------------------------------------------------------------
-- Author:        Paul R. Kucher
-- Module Name:   memory_block_transfer - Behavioral
-- Modified:      2007-09-13
-- Description:   This module facilitates block memory transfers and indirectly
--                controls data to and from the memory by external sources.
-----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity memory_block_transfer is
generic(
    width:          integer := 16;
    addr:           integer := 18;
    depth:          integer := 8
);
Port(
    clk:            in  std_logic; -- FPGA clock

    from_address:   in  std_logic_vector(addr downto 0); -- Start Address
    to_address:     in  std_logic_vector(addr downto 0); -- End Address
    read_block:     in  std_logic; -- Read block flag
    write_block:    in  std_logic; -- Write block flag
    data_in:        in  std_logic_vector(15 downto 0); -- Data to write
    op_completed:   out std_logic; -- Memory operation complete flag

    txd_complete:   in  std_logic; -- Serial transfer complete flag
    txd_ready:      out std_logic; -- Serial data ready flag
    parallel_txd:   out std_logic_vector(15 downto 0); -- Data for serial

    read_control:   out std_logic; -- Instruct module to read data
    write_control:  out std_logic; -- Instruct module to write data
    mem_op_completed: in std_logic; -- Flag if memory operation completed
    address:           out std_logic_vector(addr downto 0); -- R/W Address
    memory_data_write: out std_logic_vector(width-1 downto 0); -- Write Data
    memory_data_read:  in  std_logic_vector(width-1 downto 0) -- Read Data
);
end memory_block_transfer;

architecture Behavioral of memory_block_transfer is

begin

memory_block: process( clk )

variable read_block_state: integer range 0 to 7 := 0;
variable write_block_state: integer range 0 to 3 := 0;
variable current_address: std_logic_vector(addr downto 0);

begin
  if clk'event and clk = '1' then

    if read_block_state = 0 and write_block_state = 0 then
      op_completed <= '0';
      read_control  <= '0';
      write_control <= '0';
      txd_ready <= '0';
    end if;

    case read_block_state is  -- Block Read State Machine
      when 0 =>
        if read_block = '1' then
          current_address := from_address;
          read_block_state := 1;
        end if;
      when 1 =>
        if to_address < current_address then
          op_completed <= '1';
          read_block_state := 0;
        else
          address <= current_address;
          read_control <= '1';
          read_block_state := 2;
        end if;
      when 2 =>
        read_control <= '0';
        read_block_state := 3;
      when 3 =>
        if mem_op_completed = '1' then
          read_block_state := 4;
        end if;
      when 4 =>
        parallel_txd <= memory_data_read;
        txd_ready <= '1';
        read_block_state := 5;
      when 5 =>
        txd_ready <= '0';
        read_block_state := 6;
      when 6 =>
        if txd_complete = '1' then
          current_address := current_address + "0000000000000000001";
          read_block_state := 1;
        end if;
      when others => read_block_state := 0;
    end case;
    case write_block_state is  -- Block Write State Machine
      when 0 =>
        if write_block = '1' then
          current_address := from_address;
          write_block_state := 1;
        end if;
      when 1 =>
        if to_address < current_address then
          txd_ready <= '1';
          op_completed <= '1';
          write_block_state := 0;
        else
          memory_data_write <= data_in;
          address <= current_address;
          write_control <= '1';
          parallel_txd <= data_in;
          write_block_state := 2;
        end if;
      when 2 =>
        write_control <= '0';
        if mem_op_completed = '1' then write_block_state := 3; end if;
      when 3 =>
        current_address := current_address + "0000000000000000001";
        write_block_state := 1;
      when others => write_block_state := 0;
    end case;
  end if;
end process;

end Behavioral;
```

C.8 Memory Controller: memory_io.vhd

```
----------------------------------------------------------------------------------
-- Author:        Paul R. Kucher
-- Module Name:   memory_io - Behavioral
-- Modified:      2007-09-13
-- Description:   This module is responsible for controlling the FPGA development
--                board's SRAM chips. This module controls both chips by adding
--                a 19th address bit to select each IC.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity memory_io is
generic(
    width:          integer := 16;
    addr:           integer := 18;
    depth:          integer := 8
);
Port(
    clk:            in    std_logic; -- FPGA clock
    ce1:            out   std_logic; -- Chip enable #1
    ub1:            out   std_logic; -- Upper byte enable #1
    lb1:            out   std_logic; -- Lower byte enable #1
    ce2:            out   std_logic; -- Chip enable #2
    ub2:                out   std_logic; -- Upper byte enable #2
    lb2:                out   std_logic; -- Lower byte enable #2
    oe:                 out   std_logic; -- Output enable
    we:                 out   std_logic; -- Write enable
    mem_address:        out   std_logic_vector(addr-1 downto 0); -- Addr. Bus
    mem_data1:          inout std_logic_vector(width-1 downto 0); -- Data Bus 1
    mem_data2:          inout std_logic_vector(width-1 downto 0); -- Data Bus 2

    read_control:       in    std_logic; -- Instruct module to read data
    write_control:      in    std_logic; -- Instruct module to write data
    mem_op_completed:   out   std_logic; -- Flag if memory operation completed
    address:            in    std_logic_vector(addr downto 0); -- Addr. to R/W
    memory_data_write:  in    std_logic_vector(width-1 downto 0); -- Write Data
    memory_data_read:   out   std_logic_vector(width-1 downto 0) -- Read Data
);
end memory_io;

architecture Behavioral of memory_io is

begin

-- This process controls whether the address is from the read
-- or write operation and sets the address bus accordingly.
-- It also sets the control signals from the memory and moves
-- data from the internal registers to the external memory and
-- vice versa.
memory_data: process( clk )

variable memory_state: integer range 0 to 3 := 0;
variable enable_mem1, enable_mem2: std_logic;

begin
  if clk'event and clk = '1' then
    case memory_state is
      when 0 =>
        if read_control = '1' then
          mem_address <= address( addr-1 downto 0 );
          we <= '1';
          oe <= '0';
          enable_mem1 := address(addr);
          enable_mem2 := not address(addr);
          memory_state := 1;
        elsif write_control = '1' then
          mem_address <= address( addr-1 downto 0 );
          we <= '0';
          oe <= '1';
          enable_mem1 := address(addr);
          enable_mem2 := not address(addr);
          memory_state := 2;
        else
          memory_state := 0;
          mem_data1 <= (mem_data1'range => 'Z');
          mem_data2 <= (mem_data2'range => 'Z');
          mem_address <= "000000000000000000";
          oe <= '1';
          we <= '1';
          enable_mem1 := '1';
          enable_mem2 := '1';
        end if;
        mem_op_completed <= '0';
      when 1 => -- read state
        if address(addr) = '0' then
```

```
          memory_data_read <= mem_data1;
        else
          memory_data_read <= mem_data2;
        end if;
        mem_op_completed <= '1';
        memory_state := 0;
      when 2 => -- write state
        if address(addr) = '0' then
          mem_data1 <= memory_data_write;
        else
          mem_data2 <= memory_data_write;
        end if;
        mem_op_completed <= '1';
        memory_state := 0;
      when others => memory_state := 0;
    end case;

    ce1 <= enable_mem1;
    ub1 <= enable_mem1;
    lb1 <= enable_mem1;

    ce2 <= enable_mem2;
    ub2 <= enable_mem2;
    lb2 <= enable_mem2;
  end if;
end process;

end Behavioral;
```

C.9 Memory Multiplexer: memory_mux.vhd

```
--------------------------------------------------------------------------------
-- Author:         Paul R. Kucher
-- Module Name:    memory_mux - Behavioral
-- Modified:       2007-09-13
-- Description:    This module multiplexes the input signals to the memory_io
--                 module. This is necessary since multiple modules need
--                 access to the memory buses. This module is controlled via
--                 the instruction execution controller.
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity memory_mux is
generic(
    width:                  integer := 16;
    addr:                   integer := 18;
    depth:                  integer := 8
);
Port(
    module_select:          in  integer range 0 to 5; -- Select

    read_control_in:        in  std_logic_vector(4 downto 0); -- Read Flag
    write_control_in:       in  std_logic_vector(4 downto 0); -- Write Flag
    address_in:             in  std_logic_vector(((addr+1)*5-1) downto 0);
    memory_data_write_in:   in  std_logic_vector((width*5)-1 downto 0);

    read_control_out:       out std_logic; -- Read Flag
    write_control_out:      out std_logic; -- Write Flag
    address_out:            out std_logic_vector(addr downto 0); -- Address
    memory_data_write_out:  out std_logic_vector(width-1 downto 0) -- Data
);
end memory_mux;

architecture Behavioral of memory_mux is

begin

multiplexer: process( module_select, read_control_in, write_control_in,
                      address_in, memory_data_write_in )

begin

case module_select is
  when 1 =>
    read_control_out <= read_control_in(0);
    write_control_out <= write_control_in(0);
    address_out <= address_in(addr downto 0);
    memory_data_write_out <= memory_data_write_in(width-1 downto 0);
  when 2 =>
    read_control_out <= read_control_in(1);
    write_control_out <= write_control_in(1);
    address_out <= address_in(((addr+1)*2)-1 downto addr+1);
    memory_data_write_out <= memory_data_write_in(width*2-1 downto width);
  when 3 =>
    read_control_out <= read_control_in(2);
    write_control_out <= write_control_in(2);
    address_out <= address_in(((addr+1)*3)-1 downto (addr+1)*2);
    memory_data_write_out <= memory_data_write_in(width*3-1 downto width*2);
  when 4 =>
    read_control_out <= read_control_in(3);
    write_control_out <= write_control_in(3);
    address_out <= address_in(((addr+1)*4)-1 downto (addr+1)*3);
    memory_data_write_out <= memory_data_write_in(width*4-1 downto width*3);
  when 5 =>
    read_control_out <= read_control_in(4);
    write_control_out <= write_control_in(4);
    address_out <= address_in(((addr+1)*5)-1 downto (addr+1)*4);
    memory_data_write_out <= memory_data_write_in(width*5-1 downto width*4);
  when others =>
    read_control_out <= '0';
    write_control_out <= '0';
    address_out <= "0000000000000000000";
    memory_data_write_out <= "0000000000000000";
end case;

end process;

end Behavioral;
```

C.10 Voltage DAC Controller: program_dacs.vhd

```
--------------------------------------------------------------------------------
-- Author:         Paul R. Kucher
-- Module Name:    program_dacs - Behavioral
-- Modified:       2007-09-13
-- Description:  Update one LTC2600 16-bit digital-to-analog converter
--               on the motherboard using the provided DAC instruction.
------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity program_dacs is
Port(
    clk:              in  std_logic; -- FPGA clock
    clr_bar:          out std_logic; -- CLR_BAR DAC pin
    cs_bar:           out std_logic; -- CS_BAR DAC pin
    sdi:              out std_logic; -- Serial Data In DAC pin
    sck:              out std_logic; -- Serial Clock DAC pin

    program_dac:      in  std_logic; -- Program DAC control flag
    dac_programmed:   out std_logic; -- Programming completed control flag
    dac_instruction:  in  std_logic_vector(27 downto 0) -- DAC instruction
);
end program_dacs;

architecture Behavioral of program_dacs is

begin

program_dacs: process( clk )
variable count: integer range 0 to 15 := 0;
variable program_state: std_logic_vector(1 downto 0) := "00";
variable dac_num: std_logic_vector(3 downto 0);
variable dac_chain: std_logic_vector(159 downto 0);
variable shift_count: integer range 0 to 180 := 0;
begin

clr_bar <= '1';

if ( clk'event AND clk='1' ) then

  if count > 0 then sck <= '1';
  else sck <= '0';
  end if;

  case program_state is
    when "00" =>
      if program_dac = '1' then
        program_state := "01";
      end if;
      cs_bar <= '1';
      dac_programmed <= '0';
    when "01" =>
      dac_num := dac_instruction(27 downto 24);
      program_state := "10";
    when "10" =>
      dac_chain :=
      "00000000111100000000000000000000" &
      "00000000111100000000000000000000" &
      "00000000111100000000000000000000" &
      "00000000111100000000000000000000" &
      "00000000111100000000000000000000";
      if    dac_num = "0001" then
        dac_chain(23  downto 0)  := dac_instruction(23 downto 0);
      elsif dac_num = "0010" then
```

```
        dac_chain(55  downto 32) := dac_instruction(23 downto 0);
      elsif dac_num = "0011" then
        dac_chain(87  downto 64) := dac_instruction(23 downto 0);
      elsif dac_num = "0100" then
        dac_chain(119 downto 96) := dac_instruction(23 downto 0);
      elsif dac_num = "0101" then
        dac_chain(151 downto 128) := dac_instruction(23 downto 0);
      end if;
      program_state := "11";
    when "11" =>
      if shift_count < 160 then
        sdi <= dac_chain(159);
        cs_bar <= '0';
      else
        sdi <= '0';
        cs_bar <= '1';
        dac_programmed <= '1';
        shift_count := 0;
        program_state := "00";
      end if;

      if count = 1 then
        shift_count := shift_count + 1;
        dac_chain(159 downto 1) := dac_chain(158 downto 0);
      end if;
    when others => program_state := "00";
  end case;

  if count = 1 or program_state /= "11" then count := 0;
  else count := count + 1;
  end if;

end if;
end process;

end Behavioral;
```

C.11 RS-232 Serial Controller: serial_io.vhd

```
------------------------------------------------------------------------
-- Author:          Paul R. Kucher
-- Module Name:     serial_io - Behavioral
-- Modified:        2007-08-06
-- Description:     This module is responsible for controlling the serial
--                  communications between the FPGA and the PC. It generates
--                  the appropriate baud rate and handles turning parallel into
--                  serial data for transmission and serial into parallel data
--                  for receiving. This module is currently configured for
--                  a 115200 bps baud rate, but may be adjusted by changing the
--                  counters in the baud clock generators.
------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity serial_io is
Port(
    clk:            in    std_logic; -- FPGA clock
    rxd:           in    std_logic; -- Physical receiver pin
    txd:           out   std_logic; -- Physical transmitter pin

    txd_ready:     in    std_logic; -- Data ready on parallel_txd to send
    txd_complete:  out   std_logic; -- Transmitted word control flag
    rxd_complete:  out   std_logic; -- Received word control flag
    parallel_txd:  in    std_logic_vector(15 downto 0); -- Transmit register
    parallel_rxd:  out   std_logic_vector(15 downto 0)  -- Received register
    );
end serial_io;

architecture Behavioral of serial_io is

signal baud_sample: std_logic;
signal rxd_state: std_logic_vector(4 downto 0) := "00000";
signal txd_state: std_logic_vector(4 downto 0) := "00100";
signal rxd_filtered, txd_temp: std_logic := '1';
signal rxd_sync: std_logic_vector(1 downto 0) := "11";
signal get_next_bit: std_logic := '0';

begin

-- This process divides the global clock down to the baud
-- rate for use in the RS232 serial communications.
-- 115200 bps (specifically 115207.373 bps) is the current baud rate.
baud_rate_set: process( clk )
variable count: integer range 0 to 500 := 0;
begin
  if clk'event and clk = '1' then
    count := count + 1;

    if count > 434 then count := 0;
    end if;

    if count = 1 then baud_sample <= '1';
    else baud_sample <= '0';
    end if;
  end if;
end process;

-- This process is what controls RS232 communications with Matlab. It will
-- read 16 bit data from the parallel_txd signal and sends it to Matlab
-- in two chunks of 8 bits each. 115200 baud is used as well as mark parity
-- and 2 stop bits.
txd_state_machine: process( clk )
begin
  if clk'event and clk = '1' then
    case txd_state is
      when "00100" => if txd_ready   = '1' then
        txd_state <= "00000"; end if; -- idle state
      when "00000" => if baud_sample = '1' then
        txd_state <= "01000"; end if; -- send start bit
      when "01000" => if baud_sample = '1' then
        txd_state <= "01001"; end if; -- bit 0
      when "01001" => if baud_sample = '1' then
        txd_state <= "01010"; end if; -- bit 1
      when "01010" => if baud_sample = '1' then
        txd_state <= "01011"; end if; -- bit 2
      when "01011" => if baud_sample = '1' then
        txd_state <= "01100"; end if; -- bit 3
      when "01100" => if baud_sample = '1' then
        txd_state <= "01101"; end if; -- bit 4
      when "01101" => if baud_sample = '1' then
        txd_state <= "01110"; end if; -- bit 5
      when "01110" => if baud_sample = '1' then
        txd_state <= "01111"; end if; -- bit 6
      when "01111" => if baud_sample = '1' then
        txd_state <= "00001"; end if; -- bit 7
      when "00001" => if baud_sample = '1' then
        txd_state <= "00010"; end if; -- parity bit
      when "00010" => if baud_sample = '1' then
        txd_state <= "00011"; end if; -- stop bit #1
      when "00011" => if baud_sample = '1' then
        txd_state <= "10000"; end if; -- stop bit #2
      when "10000" => if baud_sample = '1' then
        txd_state <= "11000"; end if; -- send 2nd start bit
      when "11000" => if baud_sample = '1' then
        txd_state <= "11001"; end if; -- bit 8
      when "11001" => if baud_sample = '1' then
        txd_state <= "11010"; end if; -- bit 9
      when "11010" => if baud_sample = '1' then
        txd_state <= "11011"; end if; -- bit 10
      when "11011" => if baud_sample = '1' then
        txd_state <= "11100"; end if; -- bit 11
      when "11100" => if baud_sample = '1' then
        txd_state <= "11101"; end if; -- bit 12
      when "11101" => if baud_sample = '1' then
        txd_state <= "11110"; end if; -- bit 13
      when "11110" => if baud_sample = '1' then
        txd_state <= "11111"; end if; -- bit 14
      when "11111" => if baud_sample = '1' then
        txd_state <= "10001"; end if; -- bit 15
      when "10001" => if baud_sample = '1' then
        txd_state <= "10010"; end if; -- parity bit
      when "10010" => if baud_sample = '1' then
        txd_state <= "10011"; end if; -- stop bit #1
      when "10011" => if baud_sample = '1' then
        txd_state <= "00100"; end if; -- stop bit #2
      when others  => txd_state <= "00100";
    end case;
  end if;
end process;

-- This process handles the parallel to serial conversion of the data in the
-- input register named 'parallel_txd'.
txd_shift: process( clk )
variable parallel_data : std_logic_vector(15 downto 0) := "0000000000000000";
begin
  if clk'event and clk = '1' then
    if baud_sample = '1' then
      if txd_state(3 downto 0) = "0000" then
        txd_temp <= '0'; -- start bit
        if txd_state(4) = '0' then -- load data
          parallel_data := parallel_txd;
        end if;
      elsif txd_state(3) = '1' then  -- send data bits
        txd_temp <= parallel_data(0);
        parallel_data(14 downto 0) := parallel_data(15 downto 1);
      else
        txd_temp <= '1'; -- idle is always high
      end if;

      if txd_state = "10011" then
        txd_complete <= '1'; -- send ack. bit
      end if;
    else
      txd_complete <= '0';
    end if;
  end if;
end process;

txd <= txd_temp;

-- Synchronize the receiver pin with the FPGA clock.
rxd_sync_input: process( clk )
begin
  if clk'event and clk = '1' then
    rxd_sync <= rxd_sync(0) & rxd;
  end if;
end process;

-- Filter the incoming data for any glitches.
rxd_filter_input: process( clk )
variable count: integer range 0 to 60 := 50;
begin
  if clk'event and clk = '1' then
    if rxd_sync(1) = '1' and count < 50 then count := count + 1;
    elsif rxd_sync(1) = '0' and count > 0 then count := count - 1;
    end if;

    if count = 50 then rxd_filtered <= '1';
    elsif count = 0 then rxd_filtered <= '0';
    end if;
  end if;
end process;

-- Baud rate generator for receiving.
rxd_next_bit: process( clk )
variable count: integer range 0 to 1000 := 0;
begin
  if clk'event and clk = '1' then
    if rxd_state(3 downto 0) = "0000" then
      if rxd_filtered = '0' then count := count + 1;
      else count := 0;
      end if;

      if count = 200 then
        get_next_bit <= '1';
        count := 0;
      else
        get_next_bit <= '0';
      end if;
    elsif count = 434 then
      get_next_bit <= '1';
      count := 0;
    else
      get_next_bit <= '0';
      count := count + 1;
    end if;
  end if;
end process;

-- This process manages the incoming command structure to the FPGA
-- over RS232 communications.
rs232_receive_sm: process( clk )
begin
```

```
  if clk'event and clk = '1' then
    case rxd_state is
      when "00000" => if get_next_bit = '1' then
        rxd_state <= "01000"; end if; -- start bit found?
      when "01000" => if get_next_bit = '1' then
        rxd_state <= "01001"; end if; -- bit 0
      when "01001" => if get_next_bit = '1' then
        rxd_state <= "01010"; end if; -- bit 1
      when "01010" => if get_next_bit = '1' then
        rxd_state <= "01011"; end if; -- bit 2
      when "01011" => if get_next_bit = '1' then
        rxd_state <= "01100"; end if; -- bit 3
      when "01100" => if get_next_bit = '1' then
        rxd_state <= "01101"; end if; -- bit 4
      when "01101" => if get_next_bit = '1' then
        rxd_state <= "01110"; end if; -- bit 5
      when "01110" => if get_next_bit = '1' then
        rxd_state <= "01111"; end if; -- bit 6
      when "01111" => if get_next_bit = '1' then
        rxd_state <= "00001"; end if; -- bit 7
      when "00001" => if get_next_bit = '1' then
        rxd_state <= "10000"; end if; -- stop bit
      when "10000" => if get_next_bit = '1' then
        rxd_state <= "11000"; end if; -- 2nd start bit found?
      when "11000" => if get_next_bit = '1' then
        rxd_state <= "11001"; end if; -- bit 8
      when "11001" => if get_next_bit = '1' then
        rxd_state <= "11010"; end if; -- bit 9
      when "11010" => if get_next_bit = '1' then
        rxd_state <= "11011"; end if; -- bit 10
      when "11011" => if get_next_bit = '1' then
        rxd_state <= "11100"; end if; -- bit 11
      when "11100" => if get_next_bit = '1' then
        rxd_state <= "11101"; end if; -- bit 12
      when "11101" => if get_next_bit = '1' then
        rxd_state <= "11110"; end if; -- bit 13
      when "11110" => if get_next_bit = '1' then
        rxd_state <= "11111"; end if; -- bit 14
      when "11111" => if get_next_bit = '1' then
        rxd_state <= "10001"; end if; -- bit 15
      when "10001" => if get_next_bit = '1' then
        rxd_state <= "00000"; end if; -- stop bit
      when others  => rxd_state <= "00000";
    end case;
  end if;
end process;

-- This process shifts in the 16-bit instruction / data from the PC and makes
-- it available on the signal 'parallel_rxd ' when 'rxd_complete ' is high.
rs232_shift_in: process( clk )
variable packet1, packet2 : std_logic_vector(7 downto 0);
begin
  if clk'event and clk = '1' then
    if get_next_bit = '1' and rxd_state(3) = '1' then
      if rxd_state(4) = '0' then -- shift data
        packet1 := rxd_filtered & packet1(7 downto 1);
      elsif rxd_state(4) = '1' then -- shift data
        packet2 := rxd_filtered & packet2(7 downto 1);
      end if;
    end if;

    if rxd_state = "10001" and get_next_bit = '1' then
```

```
      parallel_rxd <= packet2 & packet1;
      rxd_complete <= '1';
    else
      rxd_complete <= '0';
    end if;
  end if;
end process;

end Behavioral;
```

C.12 Serial I/O Multiplexer: serial_mux.vhd

```
----------------------------------------------------------------------------------
-- Author:          Paul R. Kucher
-- Module Name:     serial_mux - Behavioral
-- Modified:        2007-09-13
-- Description:     This module multiplexes the input signals to the serial_io
--                  module to allow direct communications back to the PC without
--                  having to pass data through the instruction decode and execute
--                  unit.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity serial_mux is
Port(
    module_select:    in    integer range 0 to 2; -- Select Line
    txd_ready_in:     in    std_logic_vector(2 downto 1); -- TXD Flag In
    parallel_txd_in:  in    std_logic_vector(32 downto 1); -- TXD Reg. In

    txd_ready_out:    out   std_logic; -- TXD Flag Out
    parallel_txd_out: out   std_logic_vector(16 downto 1) -- TXD Reg. Out
    );
end serial_mux;

architecture Behavioral of serial_mux is

begin

multiplexer: process( module_select, txd_ready_in, parallel_txd_in )

begin

case module_select is
  when 1 =>
    txd_ready_out <= txd_ready_in(1);
    parallel_txd_out <= parallel_txd_in(16 downto 1);
  when 2 =>
    txd_ready_out <= txd_ready_in(2);
    parallel_txd_out <= parallel_txd_in(32 downto 17);
  when others =>
    txd_ready_out <= '0';
    parallel_txd_out <= "0000000000000000";
end case;

end process;

end Behavioral;
```

C.13 Serial Shifting Controller: serial_shifter.vhd

```
----------------------------------------------------------------------------------
-- Author:         Paul R. Kucher
-- Module Name:    serial_shifter - Behavioral
-- Modified:       2007-09-05
-- Description:    This unit enables and disables the states of a generic serial
--                 shift register. The starting and stopping addresses specify
--                 where in memory the serial shifting states are stored and
--                 this module will then read the least significant bit
--                 from sequential memory locations and output them on the
--                 'serial_out' line.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity serial_shifter is
generic(
    width:              integer := 16;
    addr:               integer := 18;
    depth:              integer := 8
);
Port(
    clk:                in  std_logic; -- FPGA Clock (Pin T9)
    serial_clk:         out std_logic; -- Serial Clock (FPGA Pin A9)
    serial_in:          out std_logic; -- Serial Data To Chip (FPGA Pin E6)
    serial_out:         in  std_logic; -- Serial Data From Chip (FPGA Pin D5)

    shift_start_flag:   in  std_logic; -- Shifting operation begin flag
    shift_end_flag:     out std_logic; -- Shifting operation complete flag
    start_address:      in  std_logic_vector(addr downto 0); -- St. Address
    end_address :       in  std_logic_vector(addr downto 0); -- Ending Address

    read_control:       out std_logic; -- Instruct module to read data
    write_control:      out std_logic; -- Instruct module to write data
    mem_op_completed:   in  std_logic; -- Flag if memory operation completed
    address:            out std_logic_vector(addr downto 0); -- Address to R/W
    memory_data_write:  out std_logic_vector(width-1 downto 0); -- Write Data
    memory_data_read:   in  std_logic -- Read Data
);
end serial_shifter;

architecture Behavioral of serial_shifter is

begin

serial_shift: process( clk )
variable shift_state: integer range 0 to 7 := 0;
variable read_next: std_logic := '0';
variable startup_complete: std_logic := '0';
variable shift_out_period: integer range 0 to 200;
variable current_address, cell_count: std_logic_vector(addr downto 0);

begin

```

```
if clk'event and clk = '1' then

  case shift_state is
    when 0 =>
      shift_end_flag <= '0';
      serial_in <= '0';
      read_control  <= '0';
      write_control <= '0';
      memory_data_write <= "1111111111111111";
      address <= "0000000000000000000";
      if shift_start_flag = '1' then
        current_address := end_address;
        cell_count := end_address - start_address + 1;
        shift_state := 1;
      end if;
    when 1 =>
      if current_address < start_address then
        shift_end_flag <= '1';
        shift_state := 0;
      elsif read_next = '1' then
        address <= current_address;
        read_control <= '1';
        shift_state := 2;
      end if;
    when 2 =>
      read_control <= '0';
      shift_state := 3;
    when 3 =>
      if mem_op_completed = '1' then
        shift_state := 4;
      end if;
    when 4 => -- Read serial data back for verification
      serial_in <= memory_data_read;
      address <= current_address + cell_count;
      memory_data_write <= "000000000000000" & serial_out;
      write_control <= '1';
      shift_state := 5;
    when 5 =>
      write_control <= '0';
      shift_state := 6;
    when 6 =>
      if mem_op_completed = '1' then
        current_address := current_address - 1;
        shift_state := 7;
      end if;
    when 7 =>
      if shift_out_period = 0 then
        shift_state := 1;
      end if;
    when others => shift_state := 0;
  end case;

  if shift_out_period = 50 then read_next := '1';
  else read_next := '0';
  end if;

  if shift_out_period > 199 or shift_state = 0 then shift_out_period := 0;
  else shift_out_period := shift_out_period + 1;
  end if;

  if shift_out_period > 100 then serial_clk <= '0';
  else serial_clk <= '1';
```

```
    end if;
  end if;
  end process;

  end Behavioral;
```

C.14 Seven Segment Display: seven_segment.vhd

```
----------------------------------------------------------------------------------
-- Author:         Paul R. Kucher
-- Module Name:    seven_segment - Behavioral
-- Modified:       2007-08-06
-- Description:    This module controls the seven segment display on the FPGA
--                 development board.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity seven_segment is
Port(
    clk:       in    std_logic; -- FPGA clock
    digit_sel: out   std_logic_vector(3 downto 0); -- Digit selection
    digit_val: out   std_logic_vector(6 downto 0); -- Digit value

    led_data:  in    std_logic_vector(15 downto 0) -- Data for display
);
end seven_segment;

architecture Behavioral of seven_segment is

-- LED Timing
signal led_clk: std_logic;
signal led_index: integer range 1 to 4;
signal led_number: std_logic_vector(3 downto 0);

begin

-- Divide the clock for the 7-segment LED display.
led_timing: process( clk )
variable count_leds: integer range 0 to 200 := 0;
begin
if ( clk'event AND clk='1' ) then
  if count_leds < 100 then
    led_clk <= '0';
  else
    led_clk <= '1';
  end if;

  if count_leds > 199 then count_leds := 0;
  else count_leds := count_leds + 1;
  end if;
end if;
end process;

-- This process selects the digit to update.
digit_select: process( led_index )
begin
```

```
  case led_index is
    when 1 => digit_sel <= "1110";
    when 2 => digit_sel <= "1101";
    when 3 => digit_sel <= "1011";
    when 4 => digit_sel <= "0111";
    when others => null;
  end case;
end process;

-- For LED display write ucf file ( a,b,c,d,e,f,g for 7 segments)
-- Map the 7-bit code of the LED display to a 4-bit code 0-F
digit_value: process( led_number )
begin
    case led_number is
      when "0000" => digit_val <= "0000001";
      when "0001" => digit_val <= "1001111";
      when "0010" => digit_val <= "0010010";
      when "0011" => digit_val <= "0000110";
      when "0100" =>  digit_val <= "1001100";
      when "0101" =>  digit_val <= "0100100";
      when "0110" =>  digit_val <= "0100000";
      when "0111" =>  digit_val <= "0001111";
      when "1000" =>  digit_val <= "0000000";
      when "1001" =>  digit_val <= "0000100";
      when "1010" =>  digit_val <= "0001000";
      when "1011" =>  digit_val <= "1100000";
      when "1100" =>  digit_val <= "0110001";
      when "1101" =>  digit_val <= "1000010";
      when "1110" =>  digit_val <= "0110000";
      when "1111" =>  digit_val <= "0111000";
      when others =>  null;
    end case;
end process;

-- Set the value of the selected digit.
set_digit: process( led_clk )
variable count: integer range 0 to 5 := 0;
begin
  if led_clk'event and led_clk = '1' then
    count := count + 1;
    if count = 5 then count := 0;
    end if;

    case count is
    when 1 =>
      led_number <= led_data(3 downto 0);
      led_index <= 1;
    when 2 =>
      led_number <= led_data(7 downto 4);
      led_index <= 2;
    when 3 =>
      led_number <= led_data(11 downto 8);
      led_index <= 3;
    when 4 =>
      led_number <= led_data(15 downto 12);
      led_index <= 4;
    when others =>
      null;
    end case;
  end if;
end process;

end Behavioral;
```

C.15 Signal Generator: signal_gen.vhd

```
-------------------------------------------------------------------------------
-- Author:          Paul R. Kucher
-- Module Name:     signal_gen - Behavioral
-- Modified:        2007-09-13
-- Description:     This module controls high-speed digital-to-analog conversion
--                  for function/signal generation. It reads the values of the
--                  16-bit DACs from memory and loads them into the 160-bit serial
--                  shift chain.
-------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity signal_gen is
generic(
    width:              integer := 16;
    addr:               integer := 18;
    depth:              integer := 8
);
Port(
    clk:                in  std_logic; -- FPGA clock
    cs_bar:             out std_logic; -- CS_BAR DAC pin
    sdi:                out std_logic; -- Serial Data In DAC pin
    sck:                out std_logic; -- Serial Clock DAC pin

    sig_gen_enable:     in  std_logic;
    sig_gen_complete:   out std_logic;

    start_address:      in  std_logic_vector(addr downto 0); -- St. Data Store
    end_address :       in  std_logic_vector(addr downto 0); -- End Data Store

    read_control:       out std_logic; -- Instruct module to read data
    write_control:      out std_logic; -- Instruct module to write data
    mem_op_completed:   in  std_logic; -- Flag if memory operation completed
    address:            out std_logic_vector(addr downto 0); -- address to R/W
    memory_data_write:  out std_logic_vector(width-1 downto 0); -- Write Data
    memory_data_read:   in  std_logic_vector(width-1 downto 0) -- Read Data
);
end signal_gen;

architecture Behavioral of signal_gen is

begin

generate_signals: process( clk )
variable count: integer range 0 to 15 := 0;
variable program_state: integer range 0 to 5;
variable dac_chain: std_logic_vector(159 downto 0);
variable shift_count: integer range 0 to 180 := 0;
variable current_address , true_start: std_logic_vector(addr downto 0);
variable next_sequence: std_logic_vector(addr downto 0);
begin

if ( clk'event AND clk='1' ) then

if count > 0 then sck <= '1';
else sck <= '0';
end if;

if count = 1 or program_state /= 5 then count := 0;
else count := count + 1;
end if;

case program_state is
  when 0 => — Idle State
    cs_bar <= '1';
    sdi <= '0';
    read_control  <= '0';
    write_control <= '0';
    sig_gen_complete <= '0';
    memory_data_write <= "0000000000000000";
    dac_chain :=
      "00000000111100000000000000000000" &
      "00000000111100000000000000000000" &
      "00000000111100000000000000000000" &
      "00000000111100000000000000000000" &
      "00000000111100000000000000000000";
    if sig_gen_enable = '1' then
      program_state := 1;
      current_address := start_address;
      true_start := start_address + 5;
    end if;
  when 1 => — Setup Channels
    if current_address < true_start then
      address <= current_address;
      read_control <= '1';
      program_state := 2;
    else
      current_address := true_start;
      next_sequence := true_start + 5;
      program_state := 3;
    end if;
  when 2 =>
    read_control <= '0';
    if mem_op_completed = '1' then
      current_address := current_address + 1;
      dac_chain := dac_chain(127 downto 0) & "000000000011" &
        memory_data_read(3 downto 0) & "0000000000000000";
      program_state := 1;
    end if;
  when 3 => — Start Loading Data Sequence (DACs #1 – #5)
    if current_address < next_sequence then
      address <= current_address;
      read_control <= '1';
      program_state := 4;
    else
      program_state := 5;
    end if;
  when 4 => — Load Data Into Shift Chain
    read_control <= '0';
    if mem_op_completed = '1' then
      current_address := current_address + 1;
      dac_chain := dac_chain(127 downto 0) &
        dac_chain(159 downto 144) & memory_data_read;
      program_state := 3;
    end if;
    when 5 => -- Shift Out Data
      if shift_count < 160 then
        sdi <= dac_chain(159);
        cs_bar <= '0';
      else
        sdi <= '0';
        cs_bar <= '1';
        shift_count := 0;
        if current_address > end_address then
          current_address := true_start;
          next_sequence   := true_start + 5;
        else
          next_sequence := next_sequence + 5;
        end if;

        if sig_gen_enable = '1' then
          program_state := 3;
        else
          sig_gen_complete <= '1';
          program_state := 0;
        end if;
      end if;

      if count = 1 then
        shift_count := shift_count + 1;
        dac_chain := dac_chain(158 downto 0) & dac_chain(159);
      end if;
    when others => program_state := 0;
  end case;
end if;
end process;

end Behavioral;
```

C.16 Voltage ADC Controller: vadc.vhd

```
--------------------------------------------------------------------------------
-- Author:         Paul R. Kucher
-- Module Name:    vadc - Behavioral
-- Modified:       2007-08-23
-- Description:    This module controls the 16 channel, 24-bit Sigma Delta
--                 Analog-to-Digital converters used for voltage measurement.
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity vadc is
generic(
    width:          integer := 16;
    addr:           integer := 18;
    depth:          integer := 8
);
Port(
    clk:            in  std_logic; -- FPGA clock
    adc_clk:        out std_logic; -- Conversion Clock (f0)
    adc_sck:        out std_logic; -- Serial Clock
    adc_cs_bar:     out std_logic; -- Serial Transfer Enable
    adc_sdi:            out std_logic; -- Serial Data In (to ADC)
    adc_sdo:            in  std_logic; -- Serial Data Out (of ADC)

    adc_conv_mode:      in  std_logic_vector(1 downto 0); -- f0 Mode
    adc_address:        in  std_logic_vector(3 downto 0); -- Channel Address
    adc_start_address: in  std_logic_vector(addr downto 0); -- St. Data Store
    adc_end_address:    in  std_logic_vector(addr downto 0); -- End Data Store
    adc_data_ready:     out std_logic; -- Data Conversion Complete Flag
    adc_data_collect:   in  std_logic; -- Control flag to initiate sampling

    read_control:       out std_logic; -- Instruct module to read data
    write_control:      out std_logic; -- Instruct module to write data
    mem_op_completed:   in  std_logic; -- Flag if memory operation completed
    address:            out std_logic_vector(addr downto 0); -- Address to R/W
    memory_data_write: out std_logic_vector(width-1 downto 0) -- Write Data
    );
end vadc;

architecture Behavioral of vadc is

begin

state_machine: process( clk )
variable vadc_state: integer range 0 to 7 := 0;
variable count: integer range 0 to 50 := 0;
variable pause_count: integer range 0 to 50000 := 0;
variable adc_clk_count: integer range 0 to 350 := 0;
variable adc_clk_period: integer range 0 to 350 := 24;
variable adc_clk_high: integer range 0 to 165 := 12;
variable adc_chain, sdo_parallel: std_logic_vector(31 downto 0)
         := "01001111111111111111111111111111";
variable current_address: std_logic_vector(addr downto 0);
variable shift_count: integer range 0 to 40 := 0;
variable conversion_complete: std_logic := '0';
variable startup_complete: std_logic := '0';
begin
  if clk'event and clk = '1' then

    if adc_clk_count > adc_clk_high then adc_clk <= '1';
    else adc_clk <= '0';
    end if;

    if adc_clk_count = adc_clk_period then adc_clk_count := 0;
    else adc_clk_count := adc_clk_count + 1;
    end if;

    if count > 12 and vadc_state = 1 then adc_sck <= '1';
    else adc_sck <= '0';
    end if;

    if pause_count < 50000 then
      pause_count := pause_count + 1;
    end if;

    if count = 25 or vadc_state /= 1 then count := 0;
    else count := count + 1;
    end if;

    case vadc_state is
      when 0 => -- Idle State
        read_control <= '0';
        write_control <= '0';
```

```
          adc_cs_bar <= '0';
          adc_sdi <= '0';
          shift_count := 0;
          adc_data_ready <= '0';
          conversion_complete := '0';
          current_address := adc_start_address;
          if adc_data_collect = '1' and startup_complete = '1' then
            adc_chain := "1011" & adc_address(0) & adc_address(3 downto 1) &
              "000000000000000000000000";
            if    adc_conv_mode = "00" then -- 153600 Hz f0 w/60Hz freq. rej.
              adc_clk_high := 162;
              adc_clk_period := 325;
            elsif adc_conv_mode = "01" then -- 2 MHz f0
              adc_clk_high := 12;
              adc_clk_period := 24;
            elsif adc_conv_mode = "10" then -- 1 MHz f0
              adc_clk_high := 25;
              adc_clk_period := 49;
            else                            -- 400 kHz f0
              adc_clk_high := 75;
              adc_clk_period := 124;
            end if;
            vadc_state := 1;
            --adc_cs_bar <= '0';
          end if;
        when 1 => -- Shift In/Out State
          if shift_count < 32 then
            if count = 1 then
              adc_sdi <= adc_chain(31);
            elsif count = 14 then
              sdo_parallel := sdo_parallel(30 downto 0) & adc_sdo;
            elsif count = 25 then
              adc_chain(31 downto 1) := adc_chain(30 downto 0);
              shift_count := shift_count + 1;
            end if;
          else
            pause_count := 0;
            vadc_state := 2;
          end if;
        when 2 => -- Conversion Waiting Period
          shift_count := 0;
          if adc_sdo = '0' and pause_count > 49999 then -- Conversion Complete
            adc_chain := "1011" & adc_address(0) & adc_address(3 downto 1) &
              "000000000000000000000000";
            if conversion_complete = '0' then
              conversion_complete := '1';
              vadc_state := 1;
            else -- At least one conversion completed
              address <= current_address;
              memory_data_write <= sdo_parallel(31 downto 16);
              write_control <= '1';
              vadc_state := 3;
            end if;
          end if;
        when 3 =>
          write_control <= '0';
          current_address := current_address + 1;
          vadc_state := 4;
        when 4 =>
          if mem_op_completed = '1' then
            address <= current_address;
            memory_data_write <= sdo_parallel(15 downto 0);
          write_control <= '1';
          vadc_state := 5;
        end if;
      when 5 =>
        write_control <= '0';
        current_address := current_address + 1;
        vadc_state := 6;
      when 6 =>
        if mem_op_completed = '1' then
          if current_address > adc_end_address then
            adc_data_ready <= '1';
            conversion_complete := '0';
            vadc_state := 0;
          else
            vadc_state := 1;
          end if;
        end if;
      when others => vadc_state := 0;
    end case;

    if startup_complete = '0' then
      startup_complete := '1';
    end if;
  end if;
end process;

end Behavioral;
```

C.17 Voltage DAC Multiplexer: vdac_mux.vhd

```
--------------------------------------------------------------------------------
-- Author:         Paul R. Kucher
-- Module Name:    vdac_mux - Behavioral
-- Modified:       2007-07-15
-- Description:    This module multiplexes the output signals to the serial
--                 interface to the LTC2600 voltage digital-to-analog converters.
--                 Two separate modules control the DACs. An all-channel bias
--                 controller and a high-speed signal generation module.
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity vdac_mux is
Port(
    module_select: in integer range 0 to 2;

    sck_in:        in  std_logic_vector(2 downto 1);
    sdi_in:        in  std_logic_vector(2 downto 1);
    cs_bar_in:     in  std_logic_vector(2 downto 1);

    sck_out:       out std_logic;
    sdi_out:       out std_logic;
    cs_bar_out:    out std_logic
    );
end vdac_mux;

architecture Behavioral of vdac_mux is
```

```

begin

multiplexer: process( module_select , sck_in, sdi_in, cs_bar_in )

begin

case module_select is
  when 1 => -- Bias Controller
    sck_out    <= sck_in(1);
    sdi_out    <= sdi_in(1);
    cs_bar_out <= cs_bar_in(1);
  when 2 => -- Signal Generation
    sck_out    <= sck_in(2);
    sdi_out    <= sdi_in(2);
    cs_bar_out <= cs_bar_in(2);
  when others =>
    sck_out    <= '0';
    sdi_out    <= '0';
    cs_bar_out <= '1';
end case;

end process;

end Behavioral;
```

C.18 Top Module: top.vhd

```
--------------------------------------------------------------------------------
-- Author:        Paul R. Kucher
-- Module Name:   top - Behavioral
-- Modified:      2007-09-13
-- Description:   Top module for the Mixed-Signal Test Station. This module
--                is responsible for connecting the smaller components of
--                VHDL code.
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity top is
generic(
    width:      integer := 16;
    addr:       integer := 18;
    depth:      integer := 8
);
Port(
    clksrc:          in    std_logic; -- FPGA Pin T9
    serial_clk:      out   std_logic; -- FPGA Pin B10
    serial_in:       out   std_logic; -- FPGA Pin A9
    serial_out:      in    std_logic; -- FPGA Pin B13
    inject:          out   std_logic; -- FPGA Pin C5
    tunnel:          out   std_logic; -- FPGA Pin E6
    leds:            out   std_logic_vector(7 downto 0);
    rxd:             in    std_logic; -- FPGA Pin T13
    txd:             out   std_logic; -- FPGA Pin R13
    digit_sel:       out   std_logic_vector(3 downto 0);
    digit_val:       out   std_logic_vector(6 downto 0);
    ce1:                 out   std_logic; -- FPGA Pin P7
    ub1:                 out   std_logic; -- FPGA Pin T4
    lb1:                 out   std_logic; -- FPGA Pin P6
    ce2:                 out   std_logic; -- FPGA Pin N5
    ub2:                 out   std_logic; -- FPGA Pin R4
    lb2:                 out   std_logic; -- FPGA Pin P5
    oe:                  out   std_logic; -- FPGA Pin K4
    we:                  out   std_logic; -- FPGA Pin G3
    mem_address:         out   std_logic_vector(addr-1 downto 0); -- L5 - L3
    mem_data1:           inout std_logic_vector(width-1 downto 0); -- N7 - R1
    mem_data2:           inout std_logic_vector(width-1 downto 0); -- P2 - N1
    digital_ios:         inout std_logic_vector(12 downto 0); -- FPGA I/O Pins
    clr_bar:             out   std_logic; -- Unmapped
    cs_bar:              out   std_logic; -- FPGA Pin B4
    sdi:                 out   std_logic; -- FPGA Pin D10
    sck:                 out   std_logic; -- FPGA Pin A4
    adc_clk:             out   std_logic; -- FPGA Pin B6
    adc_sck:             out   std_logic; -- FPGA Pin B7
    adc_cs_bar:          out   std_logic; -- FPGA Pin A5
    adc_sdi:             out   std_logic; -- FPGA Pin B8
    adc_sdo:             in    std_logic; -- FPGA Pin B5
    iadc_clk:            out   std_logic; -- FPGA Pin C7
    iadc_sck:            out   std_logic; -- FPGA Pin D6
    iadc_cs_bar:         out   std_logic; -- FPGA Pin D5
    iadc_sdo:            in    std_logic; -- FPGA Pin C6
    iadc_scl:            out   std_logic; -- FPGA Pin D8
    iadc_sda:            inout std_logic; -- FPGA Pin A3
    digit_pot_clk:       out   std_logic; -- FPGA Pin E7
    digit_pot_sdi:       out   std_logic; -- FPGA Pin D7
    digit_pot_cs_bar:    out   std_logic; -- FPGA Pin C8
    digit_pot_shdn_bar: out   std_logic  -- FPGA Pin C9
);
end top;

architecture netlist of top is

component clkmgr is
Port(
    clkin_in:          in    std_logic;
    clkfx_out:         out   std_logic;
    clkin_ibufg_out: out   std_logic;
    clk0_out:          out   std_logic
    );
end component;

component digital_io is
Port(
    clk:           in    std_logic;
    digital_ios:   inout std_logic_vector(12 downto 0);
    instruction:   in    std_logic_vector(5 downto 0);
    io_update:     in    std_logic;
    io_updated:    out   std_logic;
    output:        out   std_logic
    );
end component;

component serial_io is
Port(
    clk:               in    std_logic;
    rxd:               in    std_logic;
    txd:               out   std_logic;
    txd_ready:         in    std_logic;
    txd_complete:     out   std_logic;
    rxd_complete:     out   std_logic;
    parallel_txd:     in    std_logic_vector(15 downto 0);
    parallel_rxd:     out   std_logic_vector(15 downto 0)
);
end component;

component serial_mux is
Port(
    module_select:    in    integer range 0 to 2;
    txd_ready_in:     in    std_logic_vector(2 downto 1);
    parallel_txd_in:  in    std_logic_vector(32 downto 1);
    txd_ready_out:    out   std_logic;
    parallel_txd_out: out   std_logic_vector(16 downto 1)
    );
end component;

component memory_block_transfer is
generic(
    width:              integer := 16;
    addr:               integer := 18;
    depth:              integer := 8
);
Port(
    clk:                in  std_logic;
    from_address:       in  std_logic_vector(addr downto 0);
    to_address:         in  std_logic_vector(addr downto 0);
    read_block:         in  std_logic;
    write_block:        in  std_logic;
    data_in:            in  std_logic_vector(15 downto 0);
    op_completed:       out std_logic;
    txd_complete:       in  std_logic;
    txd_ready:          out std_logic;
    parallel_txd:       out std_logic_vector(15 downto 0);
    read_control:       out std_logic;
    write_control:      out std_logic;
    mem_op_completed:   in  std_logic;
    address:            out std_logic_vector(addr downto 0);
    memory_data_write:  out std_logic_vector(width-1 downto 0);
    memory_data_read:   in  std_logic_vector(width-1 downto 0)
);
end component;

component memory_io is
generic(
    width:              integer := 16;
    addr:               integer := 18;
    depth:              integer := 8
);
Port(
    clk:                in    std_logic;
    ce1:                out   std_logic;
    ub1:                out   std_logic;
    lb1:                out   std_logic;
    ce2:                out   std_logic;
    ub2:                out   std_logic;
    lb2:                out   std_logic;
    oe:                 out   std_logic;
    we:                 out   std_logic;
    mem_address:        out   std_logic_vector(addr-1  downto 0);
    mem_data1:          inout std_logic_vector(width-1 downto 0);
    mem_data2:          inout std_logic_vector(width-1 downto 0);
    read_control:      in    std_logic;
    write_control:     in    std_logic;
    mem_op_completed:  out   std_logic;
    address:           in    std_logic_vector(addr downto 0);
    memory_data_write: in    std_logic_vector(width-1 downto 0);
    memory_data_read:  out   std_logic_vector(width-1 downto 0)
);
end component;

component memory_mux is
generic(
    width:                   integer := 16;
    addr:                    integer := 18;
    depth:                   integer := 8
);
Port(
    module_select:           in  integer range 0 to 5;
    read_control_in:         in  std_logic_vector(4 downto 0);
    write_control_in:        in  std_logic_vector(4 downto 0);
    address_in:              in  std_logic_vector(((addr+1)*5-1) downto 0);
    memory_data_write_in:    in  std_logic_vector((width*5)-1 downto 0);
    read_control_out:        out std_logic;
    write_control_out:       out std_logic;
    address_out:             out std_logic_vector(addr downto 0);
    memory_data_write_out:   out std_logic_vector(width-1 downto 0)
);
end component;

component seven_segment is
Port(
    clk:       in  std_logic;
    digit_sel: out std_logic_vector(3 downto 0);
    digit_val: out std_logic_vector(6 downto 0);
    led_data:  in  std_logic_vector(15 downto 0)
);
end component;

component program_dacs is
Port(
    clk:             in  std_logic;
    clr_bar:         out std_logic;
    cs_bar:          out std_logic;
    sdi:             out std_logic;
    sck:             out std_logic;
    program_dac:     in  std_logic;
    dac_programmed:  out std_logic;
    dac_instruction: in  std_logic_vector(27 downto 0)
);
end component;

component signal_gen is
generic(
    width:              integer := 16;
    addr:               integer := 18;
    depth:              integer := 8
);
Port(
    clk:                in  std_logic;
    cs_bar:             out std_logic;
    sdi:                out std_logic;
    sck:                out std_logic;
    sig_gen_enable:     in  std_logic;
    sig_gen_complete:  out std_logic;
    start_address:     in  std_logic_vector(addr downto 0);
    end_address:       in  std_logic_vector(addr downto 0);
    read_control:      out std_logic;
    write_control:     out std_logic;
    mem_op_completed:  in  std_logic;
    address:           out std_logic_vector(addr downto 0);
    memory_data_write: out std_logic_vector(width-1 downto 0);
    memory_data_read:  in  std_logic_vector(width-1 downto 0)
);
end component;

component vdac_mux is
Port(
    module_select: in integer range 0 to 2;

    sck_in:        in  std_logic_vector(2 downto 1);
    sdi_in:        in  std_logic_vector(2 downto 1);
    cs_bar_in:     in  std_logic_vector(2 downto 1);

    sck_out:       out std_logic;
    sdi_out:       out std_logic;
    cs_bar_out:    out std_logic
    );
end component;

component vadc is
generic(
    width:              integer := 16;
    addr:               integer := 18;
    depth:              integer := 8
);
Port(
    clk:                in  std_logic;
    adc_clk:            out std_logic;
    adc_sck:            out std_logic;
    adc_cs_bar:         out std_logic;
    adc_sdi:            out std_logic;
    adc_sdo:            in  std_logic;
    adc_conv_mode:      in  std_logic_vector(1 downto 0);
    adc_address:        in  std_logic_vector(3 downto 0);
    adc_start_address:  in  std_logic_vector(addr downto 0);
    adc_end_address:    in  std_logic_vector(addr downto 0);
    adc_data_ready:     out std_logic;
    adc_data_collect:   in  std_logic;
    read_control:       out std_logic;
    write_control:      out std_logic;
    mem_op_completed:   in  std_logic;
    address:            out std_logic_vector(addr downto 0);
    memory_data_write:  out std_logic_vector(width-1 downto 0)
    );
end component;

component iadc is
generic(
    width:               integer := 16;
    addr:                integer := 18;
    depth:               integer := 8
);
Port(
    clk:                 in    std_logic;
    iadc_clk:            out   std_logic;
    iadc_sck:           out    std_logic;
    iadc_cs_bar:        out    std_logic;
    iadc_sdo:           in     std_logic;
    iadc_scl:           out    std_logic;
    iadc_sda:           inout  std_logic;
    iadc_address:       in     std_logic_vector(3 downto 0);
    iadc_start_address: in     std_logic_vector(addr downto 0);
    iadc_end_address:   in     std_logic_vector(addr downto 0);
    iadc_data_ready:    out    std_logic;
    iadc_data_collect:  in     std_logic;
    read_control:       out    std_logic;
    write_control:      out    std_logic;
    mem_op_completed:   in     std_logic;
    address:            out    std_logic_vector(addr downto 0);
    memory_data_write:  out    std_logic_vector(width-1 downto 0)
    );
end component;

component digital_pots is
Port(
    clk:                in  std_logic;
    digit_pot_clk:      out std_logic;
    digit_pot_sdi:      out std_logic;
    digit_pot_cs_bar:   out std_logic;
    digit_pot_shdn_bar: out std_logic;
    digit_pot_number:   in  std_logic_vector(4 downto 1);
    digit_pot_value:    in  std_logic_vector(7 downto 1);
    digit_pot_update:   in  std_logic;
    digit_pot_updated:  out std_logic
    );
end component;

component injection is
Port(
    clk:                   in  std_logic;
    inject:                out std_logic;
    inject_pulse:          in  std_logic;
    pulse_injected:        out std_logic;
    injection_pulse_width: in  std_logic_vector(7 downto 0)
);
end component;

component serial_shifter is
generic(
    width:             integer := 16;
    addr:              integer := 18;
    depth:             integer := 8
);
Port(
    clk:               in  std_logic;
    serial_clk:        out std_logic;
    serial_in:         out std_logic;
    serial_out:        in  std_logic;
    shift_start_flag:  in  std_logic;
    shift_end_flag:    out std_logic;
    start_address:     in  std_logic_vector(addr downto 0);
    end_address:       in  std_logic_vector(addr downto 0);
    read_control:      out std_logic;
    write_control:     out std_logic;
    mem_op_completed:  in  std_logic;
    address:           out std_logic_vector(addr downto 0);
    memory_data_write: out std_logic_vector(width-1 downto 0);
    memory_data_read:  in  std_logic
);
end component;

component decode is
generic(
    width:                 integer := 16;
    addr:                  integer := 18;
    depth:                 integer := 8
);
Port(
    clk:                   in  std_logic;
    serial_select:         out integer range 0 to 2;
      txd_ready:             out std_logic;
    txd_complete:          in  std_logic;
    rxd_complete:          in  std_logic;
    parallel_txd:          out std_logic_vector(15 downto 0);
    parallel_rxd:          in  std_logic_vector(15 downto 0);
    shift_start_flag:      out std_logic;
    shift_end_flag:        in  std_logic;
    leds:                  out std_logic_vector( 7 downto 0 );
    led_data:              out std_logic_vector(15 downto 0);
    from_address:          out std_logic_vector(addr downto 0);
    to_address:            out std_logic_vector(addr downto 0);
    read_block:            out std_logic;
    write_block:           out std_logic;
    mem_data_in:           out std_logic_vector(15 downto 0);
    xfr_op_completed:      in  std_logic;
    module_select:         out integer range 0 to 5;
    vdac_select:           out integer range 0 to 2;
    sig_gen_enable:        out std_logic;
    sig_gen_complete:      in  std_logic;
    program_dac:           out std_logic;
    dac_programmed:        in  std_logic;
    dac_instruction:       out std_logic_vector(27 downto 0);
    adc_conv_mode:         out std_logic_vector(1 downto 0);
    adc_address:           out std_logic_vector(3 downto 0);
    adc_data_ready:        in  std_logic;
    adc_data_collect:      out std_logic;
    iadc_data_ready:       in  std_logic;
    iadc_data_collect:     out std_logic;
    inject_pulse:          out std_logic;
    pulse_injected:        in  std_logic;
    injection_pulse_width: out std_logic_vector(7 downto 0);
    tunnel_pulse:          out std_logic;
    io_instruction:        out std_logic_vector(5 downto 0);
    io_update:             out std_logic;
    io_updated:            in  std_logic;
    io_output:             in  std_logic;
    digit_pot_number:      out std_logic_vector(4 downto 1);
    digit_pot_value:       out std_logic_vector(7 downto 1);
    digit_pot_update:      out std_logic;
    digit_pot_updated:     in  std_logic
);
end component;

signal parallel_txd_in: std_logic_vector(32 downto 1);
signal txd_ready_in: std_logic_vector(2 downto 1);
signal serial_select: integer range 0 to 2;
signal txd_ready_out, txd_complete, rxd_complete: std_logic;
signal parallel_txd_out, parallel_rxd: std_logic_vector(15 downto 0);

signal led_data: std_logic_vector(15 downto 0);

signal from_address, to_address: std_logic_vector(addr downto 0);
signal read_block, write_block, xfr_op_completed, mem_op_completed: std_logic;
signal mem_data_in: std_logic_vector(15 downto 0);
signal memory_data_read: std_logic_vector(width-1 downto 0);

signal module_select: integer range 0 to 5;
signal read_control_in, write_control_in: std_logic_vector(4 downto 0);
signal address_in: std_logic_vector(((addr+1)*5-1) downto 0);
signal memory_data_write_in: std_logic_vector((width*5)-1 downto 0);
signal read_control_out, write_control_out: std_logic;
signal address_out: std_logic_vector(addr downto 0);
signal memory_data_write_out: std_logic_vector(width-1 downto 0);

signal program_dac, dac_programmed: std_logic;
signal dac_instruction: std_logic_vector(27 downto 0);

signal adc_conv_mode: std_logic_vector(1 downto 0);
signal adc_address: std_logic_vector(3 downto 0);
signal adc_data_ready, adc_data_collect: std_logic;

signal iadc_data_ready, iadc_data_collect: std_logic;

signal inject_pulse, pulse_injected: std_logic;
signal injection_pulse_width: std_logic_vector(7 downto 0);

signal io_instruction: std_logic_vector(5 downto 0);
signal io_update, io_updated, io_output: std_logic;

signal shift_start_flag, shift_end_flag: std_logic;

signal digit_pot_number: std_logic_vector(4 downto 1);
signal digit_pot_value: std_logic_vector(7 downto 1);
signal digit_pot_update, digit_pot_updated: std_logic;

signal vdac_select: integer range 0 to 2;
signal vdac_sck_in, vdac_sdi_in, vdac_cs_bar_in: std_logic_vector(2 downto 1);
signal sig_gen_enable, sig_gen_complete: std_logic;

signal clkin_ibufg_out, clk0_out: std_logic;
signal clkfx_out, clk: std_logic;

begin

clkmanager: clkmgr port map( clksrc, clkfx_out, clkin_ibufg_out, clk );

serial: serial_io port map( clk, rxd, txd, txd_ready_out, txd_complete,
    rxd_complete, parallel_txd_out, parallel_rxd );

ser_mux: serial_mux port map( serial_select, txd_ready_in,
    parallel_txd_in, txd_ready_out, parallel_txd_out );

memory_xfr: memory_block_transfer port map( clk, from_address, to_address,
    read_block, write_block, mem_data_in, xfr_op_completed, txd_complete,
    txd_ready_in(2), parallel_txd_in(32 downto 17), read_control_in(1),
    write_control_in(1), mem_op_completed,
    address_in(((addr+1)*2)-1 downto addr+1),
    memory_data_write_in(width*2-1 downto width), memory_data_read );

memory: memory_io port map( clk, ce1, ub1, lb1, ce2, ub2, lb2, oe, we,
    mem_address, mem_data1, mem_data2, read_control_out, write_control_out,
```

```
    mem_op_completed , address_out , memory_data_write_out , memory_data_read );

mem_mux: memory_mux port map( module_select , read_control_in , write_control_in ,
    address_in , memory_data_write_in , read_control_out , write_control_out ,
    address_out , memory_data_write_out );

display: seven_segment port map( clk, digit_sel , digit_val, led_data );

v_dac: program_dacs port map( clk, clr_bar , vdac_cs_bar_in(1), vdac_sdi_in(1),
    vdac_sck_in(1), program_dac , dac_programmed , dac_instruction );

sig_gen: signal_gen port map( clk, vdac_cs_bar_in(2), vdac_sdi_in(2),
    vdac_sck_in(2), sig_gen_enable , sig_gen_complete , from_address ,
    to_address , read_control_in(4), write_control_in(4), mem_op_completed ,
    address_in(((addr+1)*5)-1 downto (addr+1)*4),
    memory_data_write_in(width*5-1 downto width*4), memory_data_read );

v_dac_mux: vdac_mux port map( vdac_select , vdac_sck_in , vdac_sdi_in ,
    vdac_cs_bar_in , sck, sdi , cs_bar );

v_adc: vadc port map( clk, adc_clk , adc_sck , adc_cs_bar , adc_sdi , adc_sdo ,
    adc_conv_mode , adc_address , from_address , to_address , adc_data_ready ,
    adc_data_collect , read_control_in(2), write_control_in(2),
    mem_op_completed , address_in(((addr+1)*3)-1 downto (addr+1)*2),
    memory_data_write_in(width*3-1 downto width*2) );

i_adc: iadc port map( clk, iadc_clk, iadc_sck, iadc_cs_bar, iadc_sdo,
    iadc_scl, iadc_sda, adc_address , from_address , to_address ,
    iadc_data_ready , iadc_data_collect , read_control_in(3),
    write_control_in(3), mem_op_completed ,
    address_in(((addr+1)*4)-1 downto (addr+1)*3),
    memory_data_write_in(width*4-1 downto width*3) );

pots: digital_pots port map( clk, digit_pot_clk , digit_pot_sdi ,
    digit_pot_cs_bar , digit_pot_shdn_bar , digit_pot_number ,
    digit_pot_value , digit_pot_update , digit_pot_updated );

injection_control: injection port map( clk, inject, inject_pulse ,
    pulse_injected , injection_pulse_width );

input_output: digital_io port map( clk, digital_ios , io_instruction ,
    io_update , io_updated, io_output );

shifter: serial_shifter port map( clk, serial_clk, serial_in, serial_out ,
    shift_start_flag , shift_end_flag , from_address , to_address ,
    read_control_in(0), write_control_in(0), mem_op_completed ,
    address_in(addr downto 0), memory_data_write_in(width-1 downto 0),
    memory_data_read(0) );

main: decode port map( clk, serial_select , txd_ready_in(1), txd_complete ,
    rxd_complete , parallel_txd_in(16 downto 1), parallel_rxd ,
    shift_start_flag , shift_end_flag , leds, led_data, from_address ,
    to_address , read_block , write_block , mem_data_in , xfr_op_completed ,
    module_select , vdac_select , sig_gen_enable , sig_gen_complete ,
    program_dac , dac_programmed , dac_instruction , adc_conv_mode , adc_address ,
    adc_data_ready , adc_data_collect , iadc_data_ready , iadc_data_collect ,
    inject_pulse , pulse_injected , injection_pulse_width , tunnel ,
    io_instruction , io_update , io_updated , io_output , digit_pot_number ,
    digit_pot_value , digit_pot_update , digit_pot_updated );

end netlist;
```

C.19 Implementation Constraints File: top.ucf

```
# Implementation Constraints File
# Assignment of FPGA Pins
# Modified: 2007-09-13
NET "clksrc"              LOC = "T9"  ;
NET "mem_address<0>"      LOC = "L5"  ;
NET "mem_address<1>"      LOC = "N3"  ;
NET "mem_address<2>"      LOC = "M4"  ;
NET "mem_address<3>"      LOC = "M3"  ;
NET "mem_address<4>"      LOC = "L4"  ;
NET "mem_address<5>"      LOC = "G4"  ;
NET "mem_address<6>"      LOC = "F3"  ;
NET "mem_address<7>"      LOC = "F4"  ;
NET "mem_address<8>"      LOC = "E3"  ;
NET "mem_address<9>"      LOC = "E4"  ;
NET "mem_address<10>"     LOC = "G5"  ;
NET "mem_address<11>"     LOC = "H3"  ;
NET "mem_address<12>"     LOC = "H4"  ;
NET "mem_address<13>"     LOC = "J4"  ;
NET "mem_address<14>"     LOC = "J3"  ;
NET "mem_address<15>"     LOC = "K3"  ;
NET "mem_address<16>"     LOC = "K5"  ;
NET "mem_address<17>"     LOC = "L3"  ;
NET "mem_data1<0>"        LOC = "N7"  ;
NET "mem_data1<1>"        LOC = "T8"  ;
NET "mem_data1<2>"        LOC = "R6"  ;
NET "mem_data1<3>"        LOC = "T5"  ;
NET "mem_data1<4>"        LOC = "R5"  ;
NET "mem_data1<5>"        LOC = "C2"  ;
NET "mem_data1<6>"        LOC = "C1"  ;
NET "mem_data1<7>"        LOC = "B1"  ;
NET "mem_data1<8>"        LOC = "D3"  ;
NET "mem_data1<9>"        LOC = "P8"  ;
NET "mem_data1<10>"       LOC = "F2"  ;
NET "mem_data1<11>"       LOC = "H1"  ;
NET "mem_data1<12>"       LOC = "J2"  ;
NET "mem_data1<13>"       LOC = "L2"  ;
NET "mem_data1<14>"       LOC = "P1"  ;
NET "mem_data1<15>"       LOC = "R1"  ;
NET "mem_data2<0>"        LOC = "P2"  ;
NET "mem_data2<1>"        LOC = "N2"  ;
NET "mem_data2<2>"        LOC = "M2"  ;
NET "mem_data2<3>"        LOC = "K1"  ;
NET "mem_data2<4>"        LOC = "J1"  ;
NET "mem_data2<5>"        LOC = "G2"  ;
NET "mem_data2<6>"        LOC = "E1"  ;
NET "mem_data2<7>"        LOC = "D1"  ;
NET "mem_data2<8>"        LOC = "D2"  ;
NET "mem_data2<9>"        LOC = "E2"  ;
NET "mem_data2<10>"       LOC = "G1"  ;
NET "mem_data2<11>"       LOC = "F5"  ;
NET "mem_data2<12>"       LOC = "C3"  ;
NET "mem_data2<13>"       LOC = "K2"  ;
NET "mem_data2<14>"       LOC = "M1"  ;
NET "mem_data2<15>"       LOC = "N1"  ;
NET "ce1"                 LOC = "P7"  ;
NET "ce2"                 LOC = "N5"  ;
NET "lb1"                 LOC = "P6"  ;
NET "lb2"                 LOC = "P5"  ;
NET "ub1"                  LOC = "T4"  ;
NET "ub2"                  LOC = "R4"  ;
NET "oe"                   LOC = "K4"  ;
NET "we"                   LOC = "G3"  ;
NET "digit_val<0>"         LOC = "N16" ;
NET "digit_val<1>"         LOC = "F13" ;
NET "digit_val<2>"         LOC = "R16" ;
NET "digit_val<3>"         LOC = "P15" ;
NET "digit_val<4>"         LOC = "N15" ;
NET "digit_val<5>"         LOC = "G13" ;
NET "digit_val<6>"         LOC = "E14" ;
NET "digit_sel<0>"         LOC = "D14" ;
NET "digit_sel<1>"         LOC = "G14" ;
NET "digit_sel<2>"         LOC = "F14" ;
NET "digit_sel<3>"         LOC = "E13" ;
NET "leds<0>"              LOC = "K12" ;
NET "leds<1>"              LOC = "P14" ;
NET "leds<2>"              LOC = "L12" ;
NET "leds<3>"              LOC = "N14" ;
NET "leds<4>"              LOC = "P13" ;
NET "leds<5>"              LOC = "N12" ;
NET "leds<6>"              LOC = "P12" ;
NET "leds<7>"              LOC = "P11" ;
NET "txd"                  LOC = "R13" ;
NET "rxd"                  LOC = "T13" ;
NET "serial_clk"           LOC = "B10" ;
NET "serial_in"            LOC = "A9"  ;
NET "serial_out"           LOC = "B13" ;
NET "inject"               LOC = "C5"  ;
NET "tunnel"               LOC = "E6"  ;
NET "digital_ios<0>"       LOC = "M10" ;
NET "digital_ios<1>"       LOC = "A7"  ;
NET "digital_ios<2>"       LOC = "M7"  ;
NET "digital_ios<3>"       LOC = "A13" ;
#NET "digital_ios<4>"      LOC = "A9"  ;
NET "digital_ios<5>"       LOC = "A10" ;
NET "digital_ios<6>"       LOC = "B14" ;
NET "digital_ios<7>"       LOC = "A8"  ;
NET "digital_ios<8>"       LOC = "B11" ;
NET "digital_ios<9>"       LOC = "B12" ;
NET "digital_ios<10>"      LOC = "A12" ;
#NET "digital_ios<11>"     LOC = "B10" ;
#NET "digital_ios<12>"     LOC = "B13" ;
NET "digit_pot_clk"        LOC = "E7"  ;
NET "digit_pot_sdi"        LOC = "D7"  ;
NET "digit_pot_cs_bar"     LOC = "C8"  ;
NET "digit_pot_shdn_bar"   LOC = "C9"  ;
NET "cs_bar"               LOC = "B4"  ;
NET "sck"                  LOC = "A4"  ;
NET "sdi"                  LOC = "D10" ;
NET "adc_clk"              LOC = "B6"  ;
NET "adc_cs_bar"           LOC = "A5"  ;
NET "adc_sck"              LOC = "B7"  ;
NET "adc_sdi"              LOC = "B8"  ;
NET "adc_sdo"              LOC = "B5"  ;
NET "iadc_clk"             LOC = "C7"  ;
NET "iadc_cs_bar"          LOC = "C6"  ;
NET "iadc_sck"             LOC = "D6"  ;
NET "iadc_sdo"             LOC = "D5"  ;
NET "iadc_scl"             LOC = "A3"  ;
NET "iadc_sda"             LOC = "D8"  ;
```

APPENDIX D

MATLAB Toolbox Overview

Appendix E provides source listings for all MATLAB functions that control the test station. Each script is documented and syntax descriptions are available in the comments. Additionally, these descriptions are available in MATLAB via the help function in the form `help <function name>`.

These functions will only be visible in MATLAB if the toolbox has been properly added to the directory search path. This is accomplished by adding the full directory path in the `File` → `Set Path` dialog or by directly using the `path` function. In addition, ADC calibration data is stored in the same directory, and this path must be set in `FPGAInit.m`.

The board is controlled via the instructions described in Table 4.1, and all hardware-level communication is handled through the built-in file I/O functions such as `fopen`, `fwrite`, `fread`, and `fclose`. Serial objects are created with the `serial` function, which requires the COM port number as its argument. Consequently, this parameter must be modified for every installation. All other configuration options are set through this object and are given in Section E.1. Also, MATLAB writes to the serial port using an array of unsigned, 8-bit integers. Because the FPGA uses a 16-bit data word internally, MATLAB must read and write two 8-bit packets sequentially and reconstruct received data as a 16-bit unsigned integer.

The FPGA is initialized by calling `FPGAInit`. This function cannot be called a second time unless the serial port object is released by calling `fclose(s)`. The initialization script runs a sanity check on the FPGA to ensure that the memory transfer and loopback commands are working properly before initializing the test station. Also, the working directory is changed to the location of the toolbox. All other functions may be called following successful initialization. However, failure may result if the FPGA is not in its idle state or if the JTAG program bit stream is corrupted. Thus, ensuring that the FPGA has been reset prior to initialization will prevent either the FPGA or initialization scripts from entering a locked state.

Appendix F may act as an example toolbox for chip testing. These scripts were used in the floating gate experiments of Chapter 5. Here, `SVM2Init` acts as the system initialization routine and calls both `FPGAInit` as well as configures all of the on-chip biases and serial interface.

APPENDIX E

Test Station MATLAB Functions

E.1 FPGAInit.m

```
% This script initializes the FPGA for data acquisition. It opens the
% serial port for data transfer as well as runs some sanity checks on the
% FPGA's modules to ensure everything is working properly.

% Creates serial port object 's' at the desired settings.
cd('c:\Documents and Settings\Paul R. Kucher\Desktop\Matlab\FPGA Tools')
eval( sprintf(['load VADCCalibrationData']) ); % Voltage ADC Calibration Data

global s;
s = serial('COM2'); % Set to whatever COM port you are using.
s.Timeout = 5;
s.InputBufferSize = 600000; % A value greater than the maximum memory size
s.OutputBufferSize = 600000;
set(s,'BaudRate',115200,'Parity','none', 'StopBits', 1);
fopen(s);

random_number = round(rand(1)*1000);
FPGAWriteMemoryBlock(1,100,random_number);
read_data = FPGAReadMemoryBlock(1,100);
sum_of_zeros = sum(sum(read_data,1),2);

init_test  = FPGALoopback(5);
init_test2 = FPGALoopback(0);

if init_test ~= 5 || init_test2 ~= 0 || sum_of_zeros ~= random_number*100,
    fprintf('Sanity Check Failed! Check Setup.\n');
else
    fprintf('Initialization Successful!\n');
end;

% Initialize Potentiometers
FPGASetBiasCurrent( 0, 127 );
for channel=1:8,
    FPGASetBiasCurrent( channel, 127 );
end;

% Initialize Injection Circuit
FPGASetBias(37, 2);   % Injection Voltage
FPGASetBias(38, 1.5); % Threshold
FPGASetBias(39, 3.3); % Idle Voltage

% Initialize Current ADC
FPGASetBias(40, 1);
```

E.2 FPGALoopback.m

```
function output = FPGALoopback( intval );
% Syntax: output = FPGALoopback( intval )
%
% Loopback Test (sanity check). Takes in a value 0-255 and sends
% it to the FPGA, decodes it, and then serializes it back to Matlab to be
% read. This function is useful if you are writing a new instruction for
% the FPGA and you want to verify that the previous instruction has been
% decoded properly and the instruction fetch / decode process is at idle.

global s;

intval = uint8( intval );
fwrite(s, [intval hex2dec('00')],'sync');

while s.BytesAvailable <2,
end;

data = fread(s, 2);
output = data(1,:);
```

E.3 FPGADigitalIO.m

```
function output = FPGADigitalIO( pin_name, direction, value)
% Syntax: output = FPGADigitalIO( pin_name, direction, value)
%
% Generic digital I/O control of the FPGA's remaining I/O pins.
% Up to twelve channels are available if they have not been
% overridden in the FPGA's internal controller.

global s;

if nargin < 3,
    value = 0;
end;

if     strcmp(pin_name, 'M10'), channel = 0;
elseif strcmp(pin_name, 'A7'),  channel = 1;
elseif strcmp(pin_name, 'M7'),  channel = 2;
elseif strcmp(pin_name, 'A13'), channel = 3;
elseif strcmp(pin_name, 'A9'),  channel = 4;
elseif strcmp(pin_name, 'A10'), channel = 5;
elseif strcmp(pin_name, 'B14'), channel = 6;
elseif strcmp(pin_name, 'A8'),  channel = 7;
elseif strcmp(pin_name, 'B11'), channel = 8;
elseif strcmp(pin_name, 'B12'), channel = 9;
elseif strcmp(pin_name, 'A12'), channel = 10;
elseif strcmp(pin_name, 'B10'), channel = 11;
elseif strcmp(pin_name, 'B13'), channel = 12;
end;
```

```

if strcmp(direction, 'out'),
    dir_num = 8;
elseif strcmp(direction, 'in'),
    dir_num = 0;
else
    fprintf('You must specify the direction of data flow.');
    return;
end;

instruction = bitor( dir_num, bitshift( value, 2) );
instruction = bitor( hex2dec('80'), instruction );

try, fwrite(s, [channel instruction],'sync');
catch, fwrite(s, [channel instruction],'sync');
end;

while s.BytesAvailable < 2,
end;

data = fread(s,2);
output = bitshift(data(2,1),8) + data(1,1);
```

E.4 FPGAInjectPulse.m

```
function time = FPGAInjectPulse( width );
% Syntax: time = FPGAInjectPulse( width )
%
% Produce a pulse of -2V to the floating-gate transistor injection pin.
% The value of 'width' corresponds to the time the injection pulse is set.
% in the form time = (1/50e6)*2^(width+1) where 'time' is the return value
% in seconds.

global s;

max_pulse = hex2dec('FF');

if width > max_pulse,
    width = max_address;
elseif width < 0,
    width = 0;
end;

try, fwrite(s, [width hex2dec('60')],'sync');
catch, fwrite(s, [width hex2dec('60')],'sync');
end;

injection_width = 1/50e6*2^(width+1);

fprintf ( 'Injection width: %1.6f seconds.\n', injection_width );

pause(injection_width+1/57600*20);

while s.BytesAvailable < 2,
end;

fread(s,s.BytesAvailable);

time = injection_width;
```

E.5 FPGATunnel.m

```
function FPGATunnel( state );
% Syntax: FPGATunnel( state )
%
% Fowler-Nordheim Tunneling enable/disable function. The variable 'state'
% is specified as 1 if the tunneling voltage is desired and 0 if the 3.3V
% regulator supply is desired.

global s;

intval = uint8( state );
fwrite(s, [intval hex2dec('70')],'sync');

while s.BytesAvailable <2,
end;

data = fread(s, 2);
```

E.6 FPGASetBias.m

```
function status = FPGASetBias( channel_number , dac_value );
% Syntax: status = FPGASetBias( channel_number, dac_value );
%
% FPGASetBias sets a specific analog DC bias to one of the motherboard's
% five on-board DACs. The first parameter is the channel number (1-40) and
% the second is the specific voltage that the DAC will be set to (0 to 4.096V).

global s;

dac_num = floor((channel_number -1)/8) + 1;
channel_num = channel_number - (dac_num - 1)*8;

supply_range = 4.098;
dacbits = 16;

if (dac_num > 0) & (dac_num < 6) & (channel_num > 0) & (channel_num < 9)

    if dac_value > supply_range
        fprintf( ['Given DAC value = (%1.4f) has exceeded the higher '...
                'limit of 4.098V.\nDAC Value is corrected and assigned '...
                'with maximum supply range.\n'], dac_value );
        dac_value = supply_range;
    end;

    range_value = bitand(uint32((dac_value/supply_range)*((2^dacbits)-1)),...
        hex2dec('FFFF'));

    value_msbs = bitshift( bitand( range_value, hex2dec('FF00') ), -8);
    value_lsbs = bitand( range_value, hex2dec('00FF') );

    comm_msbs = bitor( hex2dec('20'), dac_num);
    comm_lsbs = bitor( hex2dec('30'), channel_num-1);

    try, fwrite(s, [comm_lsbs comm_msbs value_lsbs value_msbs],'sync');
    catch, fwrite(s, [comm_lsbs comm_msbs value_lsbs value_msbs],'sync');
    end;
```

```

    while s.BytesAvailable <2,
    end;
    fread(s,2);

    status = 1;
else
    fprintf( ['Either DAC number ( > 5) or channel number ( > 8)'...
             ' has exceeded the limits']);
    status = 0;
end;
```

E.7 FPGASetBiasCurrent.m

```
function FPGASetBiasCurrent( channel , value );
% Syntax: FPGASetBiasCurrent( channel, value );
%
% Bias current generator. The eight channels have a range of 0-7 (with the
% exclusion of the current ADC potentiometer, 1-8 otherwise) and have an
% integer 'value' from 0-127, representing the digital code of the
% potentiometer that controls the current amplitude.

global s;

value = uint8( value );

command = bitor(hex2dec('40'), channel);
fwrite(s, [value command],'sync');

while s.BytesAvailable <2,
end;

data = fread(s, 2);
output = data(1,:);
```

E.8 FPGAReadVoltage.m

```
function output = FPGAReadVoltage( channel , start_address , samples , clk_mode );
% Syntax: output = FPGAReadVoltage( channel, start_address , samples, clk_mode );
%
% FPGAReadVoltage allows voltage analog-to-digital conversion via
% the LTC2418 16-channel, 24-bit Delta-Sigma ADC. Channels are numbered
% 1-16 and are specified on the daughter board. The 'start_address'
% parameter sets the first address to begin storing the conversion
% result in memory. The variable 'samples' specifies the number of samples
% to be taken during the experiment.
%
% 'clk_mode' specifies the frequency of the ADC's conversion clock. It
% defaults to 2MHz when this optional parameter is unspecified. Other valid
% rates are 'internal' which provides the best accuracy and a 60 Hz notch
% filter to help reject light pickup noise, '400kHz' and '1MHz' modes are
% provided as a best compromise between speed and accuracy. Accuracy
% deteriorates rapidly after 1MHz as detailed in the LTC2418 datasheet.
%
% The output is a two-dimensional array that contains the channel from
% which the conversion result was obtained as well as the result itself.
```

```
%
% Last modified: 2007-09-13

global s fitted;

if nargin < 4,
     conv_clk_mode = 1;
elseif strcmp(clk_mode,'internal') == 1,
     conv_clk_mode = 0;
elseif strcmp(clk_mode,'400kHz') == 1,
     conv_clk_mode = 3;
elseif strcmp(clk_mode,'1MHz') == 1,
     conv_clk_mode = 2;
else % 2 MHz Conversion Clock
     conv_clk_mode = 1;
end;

channel = channel - 1;

end_address = start_address + 2*samples - 1;
data_total = end_address - start_address + 1;
start_address_read = start_address;
end_address_read = end_address;

max_address = hex2dec('7FFFF');

if start_address > end_address,
    fprintf('The ending address must be greater than the starting address!\n');
    return;
elseif (max_address - end_address) < ( end_address - start_address + 1),
    fprintf('The ending address is too close to the last address in RAM!\n');
    return;
elseif channel < 0 || channel > 15,
    fprintf('The specified channel number is invalid!\n');
    return;
end;

three_msbs1 = bitand( bitshift(start_address,-16), 7);
three_msbs2 = bitand( bitshift(end_address,-16), 7);

msbs = bitshift(conv_clk_mode,6) + bitshift(three_msbs2,3) + three_msbs1;

% Just set the address to allow a maximum number of bits.
start_address = uint16( bitand(start_address, 65535 ) );
end_address   = uint16( bitand(end_address, 65535 ) );

packet11 = bitand( start_address, hex2dec('00FF') );
packet21 = bitshift( bitand( start_address, hex2dec('FF00') ), -8);

packet12 = bitand( end_address, hex2dec('00FF') );
packet22 = bitshift( bitand( end_address, hex2dec('FF00') ), -8);

command = bitand(hex2dec('3F'),bitor(hex2dec('F0'),channel));

try, fwrite(s, [msbs command packet11 packet21 packet12 packet22],'sync');
catch, fwrite(s, [msbs command packet11 packet21 packet12 packet22],'sync');
end;
while s.BytesAvailable <2,
end;

fread(s,2);
data = FPGAReadMemoryBlock(start_address_read, end_address_read);
```

```

reference = 2.501;
supply    = 5.0209;

j=1;
for i=1:2:data_total,
    data_temp = bitor(bitshift(data(i),16),data(i+1));
    channel_returned = bitor(bitand(14,data_temp), ...
        bitshift(bitand(16,data_temp),-4));
    code = bitshift(bitand(536870848,data_temp),-6);
    measured_voltage = (bitget(data_temp,29))*bitget(data_temp,30)*supply ...
      + (1-bitget(data_temp,29))*bitget(data_temp,30) * ...
        (code/(2^23)*supply+reference) ...
      + (1-bitget(data_temp,30))*bitget(data_temp,29) * ...
        (reference-(1-code/(2^23))*supply);
    output(j,1) = channel_returned + 1;
    output(j,2) = measured_voltage; % - polyval(fitted,measured_voltage)+3e-3;
    j = j + 1;
end;
```

E.9 FPGAReadCurrent.m

```
function output = FPGAReadCurrent( channel, start_address, samples, gain );
% Syntax: output = FPGAReadCurrent( channel, start_address, samples, gain );
%
% Returns the voltage read by the LTC2415-1 data converter when doing I-V
% conversion. This function is called in FPGAEstimateCurrent, but gives
% better control over the conversion procedure such as which addresses to
% store conversion results and gain control.

global s;

update = 0;
if nargin < 4,
    update = 0;
    gain = 0;
else
    update = 8;
end;

channel = channel - 1;

end_address = start_address + 2*samples - 1;

data_total = end_address - start_address + 1;
start_address_read = start_address;
end_address_read = end_address;

max_address = hex2dec('7FFFF');

if start_address > end_address,
    fprintf('The ending address must be greater than the starting address!\n');
    return;
elseif (max_address - end_address) < ( end_address - start_address + 1),
    fprintf('The ending address is too close to the last address in RAM!\n');
    return;
elseif channel < 0 || channel > 15,
    fprintf('The specified channel number is invalid!\n');
    return;
```

```
end;

three_msbs1 = bitand( bitshift(start_address ,-16), 7);
three_msbs2 = bitand( bitshift(end_address ,-16), 7);

msbs = bitshift(three_msbs2 ,3) + three_msbs1;

% Just set the address to allow a maximum number of bits.
start_address = uint16( bitand(start_address , 65535 ) );
end_address   = uint16( bitand(end_address , 65535 ) );

packet11 = bitand( start_address , hex2dec('00FF') );
packet21 = bitshift( bitand( start_address , hex2dec('FF00') ), -8);

packet12 = bitand( end_address , hex2dec('00FF') );
packet22 = bitshift( bitand( end_address , hex2dec('FF00') ), -8);

command = bitand(hex2dec('5F'),bitor(hex2dec('F0'),channel));

try, fwrite(s, [msbs command packet11 packet21 packet12 ...
            packet22 update gain],'sync');
catch, fwrite(s, [msbs command packet11 packet21 packet12 ...
            packet22 update gain],'sync');
end;
while s.BytesAvailable <2,
end;

fread(s,2);

data = FPGAReadMemoryBlock(start_address_read ,end_address_read);

j=1;

for i=1:2:data_total,
    data_temp = bitor(bitshift(data(i),16),data(i+1));
    value = bitshift(bitand(1073741760,data_temp),-6);
    output(j) = 2*5.04*value/16777216-2.5;
    j = j + 1;
end;
```

E.10 FPGAEstimateCurrent.m

```
function output = FPGAEstimateCurrent( channel , samples );
% Syntax: output = FPGAEstimateCurrent( channel. samples );
%
% Calculate the measured current from the voltage-mode output of the
% current ADC circuit. The variable "reference" should be recalibrated
% upon startup for accurate current measurement.

global s currentadcoffset;

reference = 1.0258;
%reference = FPGAReadCurrent( 9.1.samples ); % Read from

measured = mean( FPGAReadCurrent( channel , 1, samples ) ); % Measure current

output = ( currentadcoffset - measured ) / 2.1955e6;

fprintf ( 'Measured Current:  %1.6fnA\n', output*1e9 );
```

E.11 FPGAReadMemory.m

```
function output = FPGAReadMemory( address );
% Syntax: output = FPGAReadMemory( address )
%
% Read from the FPGA data RAM by specifying a 19-bit address 'address' and
% return the value as 'output'.

global s;

max_address = hex2dec('7FFFF');

if address > max_address, % 19-bit addresses
    address = max_address;
end;

address1 = address;
address2 = address;

addresses = address2-address1+1;

three_msbs1 = bitand( bitshift(address1,-16), 7);
three_msbs2 = bitand( bitshift(address2,-16), 7);

msbs = bitshift(three_msbs2,3) + three_msbs1;

% Just set the address to allow a maximum number of bits.
address1 = uint16( bitand(address1, 65535 ) );
address2 = uint16( bitand(address2, 65535 ) );

packet11 = bitand( address1, hex2dec('00FF') );
packet21 = bitshift( bitand( address1, hex2dec('FF00') ), -8);

packet12 = bitand( address2, hex2dec('00FF') );
packet22 = bitshift( bitand( address2, hex2dec('FF00') ), -8);

try, fwrite(s, [msbs hex2dec('10') packet11 packet21 ...
            packet12 packet22],'sync');
catch, fwrite(s, [msbs hex2dec('10') packet11 packet21 ...
            packet12 packet22],'sync');
end;

while s.BytesAvailable < 2,
end;

data = fread(s,2);
output = bitshift(data(2,1),8) + data(1,1);
```

E.12 FPGAReadMemoryBlock.m

```
function output = FPGAReadMemoryBlock( address1, address2 );
% Syntax: output = FPGAReadMemoryBlock( address1, address2 )
%
% Read a block of the FPGA's memory by specifying two 19-bit addresses
% where address1 <= address2. The result is stored in the array 'output'.

global s;
```

```

max_address = hex2dec('7FFFF');

if address1 > max_address, % 19-bit addresses
    address1 = max_address;
end;

if address2 > max_address || address2 < address1, % 19-bit addresses
    address2 = address1;
end;

addresses = address2-address1+1;
bytes_to_read = 2*(addresses);

three_msbs1 = bitand( bitshift(address1,-16), 7);
three_msbs2 = bitand( bitshift(address2,-16), 7);

msbs = bitshift(three_msbs2,3) + three_msbs1;

% Just set the address to allow a maximum number of bits.
address1 = uint16( bitand(address1, 65535 ) );
address2 = uint16( bitand(address2, 65535 ) );

packet11 = bitand( address1, hex2dec('00FF') );
packet21 = bitshift( bitand( address1, hex2dec('FF00') ), -8);

packet12 = bitand( address2, hex2dec('00FF') );
packet22 = bitshift( bitand( address2, hex2dec('FF00') ), -8);

try, fwrite(s, [msbs hex2dec('10') packet11 packet21 ...
              packet12 packet22],'sync');
catch, fwrite(s, [msbs hex2dec('10') packet11 packet21 ...
              packet12 packet22],'sync');
end;

while s.BytesAvailable < bytes_to_read,
end;

data = fread(s,bytes_to_read);

addr_index = 1;
j = 1;
while j <= bytes_to_read,
    output(addr_index) = bitshift(data(j+1,1),8) + data(j,1);
    j = j+2;
    addr_index = addr_index + 1;
end;
```

E.13 FPGAWriteMemory.m

```
function FPGAWriteMemory( address, value );
% Syntax: FPGAWriteMemory( address, value )
%
% This function allows the value specified in 'value' to be written to the
% FPGA's on-board memory at the specified address given in 'address'.

global s;

max_address = hex2dec('7FFFF');
```

```
if address > max_address , % 19-bit addresses
    address = max_address;
end;

address1 = address;
address2 = address;

value = uint16(value);

addresses = address2-address1+1;

three_msbs1 = bitand( bitshift(address1,-16), 7);
three_msbs2 = bitand( bitshift(address2,-16), 7);

msbs = bitshift(three_msbs2,3) + three_msbs1;

% Just set the address to allow a maximum number of bits.
address1 = uint16( bitand(address1, 65535 ) );
address2 = uint16( bitand(address2, 65535 ) );

packet11 = bitand( address1, hex2dec('00FF') );
packet21 = bitshift( bitand( address1, hex2dec('FF00') ), -8);

packet12 = bitand( address2, hex2dec('00FF') );
packet22 = bitshift( bitand( address2, hex2dec('FF00') ), -8);

packet3 = bitand( value, hex2dec('00FF') );
packet4 = bitshift( bitand( value, hex2dec('FF00') ), -8);

try, fwrite(s, [msbs hex2dec('1F') packet11 packet21 ...
            packet12 packet22 packet3 packet4],'sync');
catch, fwrite(s, [msbs hex2dec('1F') packet11 packet21 ...
            packet12 packet22 packet3 packet4],'sync');
end;
while s.BytesAvailable<2,
end;

data = fread(s,2);
output = bitshift(data(2,1),8) + data(1,1);
```

E.14 FPGAWriteMemoryBlock.m

```
function output = FPGAWriteMemoryBlock( address1, address2, value );
% Syntax: output = FPGAWriteMemoryBlock( address1, address2, value )
%
% Write a block of data to the FPGA's Memory by specifying two 19-bit addresses
% where address1 <= address2, and the value to write.

global s;

max_address = hex2dec('7FFFF');

if address1 > max_address , % 19-bit addresses
    address1 = max_address;
end;

if address2 > max_address || address2 < address1, % 19-bit addresses
    address2 = address1;
```

```
end;

value = uint16(value);

addresses = address2-address1+1;

three_msbs1 = bitand( bitshift(address1,-16), 7);
three_msbs2 = bitand( bitshift(address2,-16), 7);

msbs = bitshift(three_msbs2,3) + three_msbs1;

% Just set the address to allow a maximum number of bits.
address1 = uint16( bitand(address1, 65535 ) );
address2 = uint16( bitand(address2, 65535 ) );

packet11 = bitand( address1, hex2dec('00FF') );
packet21 = bitshift( bitand( address1, hex2dec('FF00') ), -8);

packet12 = bitand( address2, hex2dec('00FF') );
packet22 = bitshift( bitand( address2, hex2dec('FF00') ), -8);

packet3 = bitand( value, hex2dec('00FF') );
packet4 = bitshift( bitand( value, hex2dec('FF00') ), -8);

try, fwrite(s, [msbs hex2dec('1F') packet11 packet21 ...
            packet12 packet22 packet3 packet4],'sync');
catch, fwrite(s, [msbs hex2dec('1F') packet11 packet21 ...
            packet12 packet22 packet3 packet4],'sync');
end;
while s.BytesAvailable <2,
end;

data = fread(s,2);
output = bitshift(data(2,1),8) + data(1,1);
```

E.15 FPGAWriteMemoryVector.m

```
function FPGAWriteMemoryVector(addresses, values)
% Syntax: FPGAWriteMemoryVector(addresses, values)
%
% Write the values in the vector 'values' to the addresses
% in vector 'addresses'. Note that both 'values' and
% 'addresses' must have the same length.

if length( addresses ) ~= length(values),
    fprintf('Vectors ''addresses'' and ''values'' are not of equal length!\n');
    return;
end;

total_values = length( addresses );

for i=1:total_values,
    FPGAWriteMemory(addresses(i),values(i))
end;
```

E.16 FPGASerialShift.m

```
function FPGASerialShift( start_address , end_address );
% Syntax: FPGASerialShift( start_address , end_address );
%
% This function acts as a generic , variable-width, high-speed serial shift
% register interface. By storing 1s and 0s into sequential memory addresses ,
% this function will instruct the FPGA to load the least-significant bit of
% each memory location and put it onto the serial data line.
%
% Note: The 'end_address' cannot be positioned less than the difference between
% the start and end addresses away from the last memory location in RAM.
% This is because the serial data shifted out from the chip (if present)
% will be shifted back into the FPGA and stored in the subsequent memory
% locations following the end address. Thus, the total amount of memory
% required for this module is 2*(end_address - start_address + 1).
%
% You may then read the appropriate memory locations to check that the
% desired serial chain operation is present.

global s;

max_address = hex2dec('7FFFF');

if start_address > end_address,
    fprintf('The ending address must be greater than the starting address!\n');
    return;
elseif (max_address - end_address) < ( end_address - start_address + 1),
    fprintf('The ending address is too close to the last address in RAM!\n');
    return;
end;

three_msbs1 = bitand( bitshift(start_address,-16), 7);
three_msbs2 = bitand( bitshift(end_address,-16), 7);

msbs = bitshift(three_msbs2,3) + three_msbs1;

% Just set the address to allow a maximum number of bits.
start_address = uint16( bitand(start_address, 65535 ) );
end_address   = uint16( bitand(end_address, 65535 ) );

packet11 = bitand( start_address, hex2dec('00FF') );
packet21 = bitshift( bitand( start_address, hex2dec('FF00') ), -8);

packet12 = bitand( end_address, hex2dec('00FF') );
packet22 = bitshift( bitand( end_address, hex2dec('FF00') ), -8);

try, fwrite(s, [msbs hex2dec('A0') packet11 packet21 ...
            packet12 packet22],'sync');
catch, fwrite(s, [msbs hex2dec('A0') packet11 packet21 ...
            packet12 packet22],'sync');
end;
while s.BytesAvailable <2,
end;
data = fread(s,2);
```

E.17 FPGAFunctionGenerator.m

```
function FPGAFunctionGenerator( channel, type, frequency, gain, offset );
% Syntax: FPGAFunctionGenerator( channel, type, frequency, gain, offset );
%
```

```
% This command sets up the waveforms necessary for function generation. The
% parameters may contain the following values:
%
%   channel:    1-4, 9-12, 17-20, 25-28, 33-36 are valid.
%   type:       'sine', 'square', 'sawtooth', and 'triangle' are valid.
%   frequency: < 5kHz gives acceptable performance.
%   gain:       0 < gain < 1
%   offset:     -.5 < offset < .5

global s signal_data;

if gain > 1,
    gain = 1;
elseif gain < 0;
    gain = 0;
end;

if offset > 0.5,
    offset = 0.5;
elseif offset < -0.5,
    offset = -0.5;
end;

% Compute the number of cycles to generate desired frequency
cycles = round((frequency^-1)/((20e-9)*21 + 160*2*20e-9))

dac_num = floor((channel-1)/8) + 1;
channel_num = channel - (dac_num - 1)*8;

if channel_num > 5,
    fprintf('The channel number (per DAC) cannot exceed 5!\n');
    return;
end;

if strcmp(type,'sine'),
    waveform = uint16(65535*(0.5 + offset) + ...
        65535*gain*(0.5*sin(2*pi*[1:cycles]/cycles)));
elseif strcmp(type,'square'),
    waveform([1:floor(cycles/2)])= uint16(65535*(0.5 + offset) + ...
        65535*gain*0.5*(ones(size([1:floor(cycles/2)]))));
    waveform([(floor(cycles/2)+1):cycles]) = uint16(65535*(0.5 + offset) - ...
        65535*gain*0.5*(ones(size([(floor(cycles/2)+1):cycles]))));
elseif strcmp(type,'sawtooth'),
    waveform = uint16( 65535*gain*[1:cycles]/cycles + ...
        0.5*65535*(1-gain) + offset*65535 );
elseif strcmp(type,'triangle'),
    waveform([1:floor(cycles/2)]) = ...
        65535*gain*[1:floor(cycles/2)]/floor(cycles/2) + ...
        0.5*65535*(1-gain) + 65535*offset;
    waveform([(floor(cycles/2)+1):cycles]) = ...
        65535*gain*[cycles-floor(cycles/2):-1:1]/floor(cycles/2) + ...
        0.5*65535*(1-gain) + 65535*offset;
else
    fprintf('Invalid waveform type. Setting values to zero.\n');
    waveform = zeros(1,cycles);
end;

FPGAWriteMemory( dac_num, channel_num-1 );

FPGAWriteMemoryVector([6-dac_num+5:5:(5*cycles+5)], waveform );
FPGASignalGen(1,5*cycles+5);
```

E.18 FPGASignalGen.m

```
function FPGASignalGen( start_address , end_address );
% Syntax: FPGASignalGen( start_address , end_address );
%
% This function initiates multi-channel function generation.
% It will send the start and end addresses where the waveforms
% are stored in memory and will enable the state machine
% in the FPGA that controls continuous reading of these memory
% locations.

global s;

max_address = hex2dec('7FFFF');

if start_address > end_address ,
    fprintf('The ending address must be greater than the starting address!\n');
    return;
elseif (max_address - end_address) < ( end_address - start_address + 1),
    fprintf('The ending address is too close to the last address in RAM!\n');
    return;
end;

three_msbs1 = bitand( bitshift(start_address ,-16), 7);
three_msbs2 = bitand( bitshift(end_address ,-16), 7);

msbs = bitshift(three_msbs2 ,3) + three_msbs1 ;

% Just set the address to allow a maximum number of bits .
start_address = uint16( bitand(start_address , 65535 ) );
end_address   = uint16( bitand(end_address , 65535 ) );

packet11 = bitand( start_address , hex2dec('00FF') );
packet21 = bitshift( bitand( start_address , hex2dec('FF00') ), -8);

packet12 = bitand( end_address , hex2dec('00FF') );
packet22 = bitshift( bitand( end_address , hex2dec('FF00') ), -8);

try, fwrite(s, [msbs hex2dec('90') packet11 packet21 ...
            packet12 packet22],'sync');
catch, fwrite(s, [msbs hex2dec('90') packet11 packet21 ...
            packet12 packet22],'sync');
end;
```

APPENDIX F

Floating Gate Testing Code

F.1 KeithleyInit.m

```
% Setup the Keithley 2400 SourceMeter for measurement.
% Creates serial port object 's2' at the desired settings.

global s2;
s2 = serial('COM6');
s2.Timeout = 5;
s2.InputBufferSize = 500000;
s2.OutputBufferSize = 500000;
set(s2,'BaudRate',57600,'Parity','none', 'StopBits', 1, 'Terminator','CR');
fopen(s2);

fprintf(s2, 'OUTP ON');
fprintf(s2, ':SENS:FUNC "CURR"');

if s2.BytesAvailable > 0,
    fread(s2,s2.BytesAvailable);
end;
```

F.2 KeithleyGetCurrent.m

```
function output = KeithleyGetCurrent( samples );
% Syntax: output = KeithleyGetCurrent( samples );
%
% Return the measured current as floating point value in units of Amperes.

global s2;

for i=1:samples,
    try, fprintf(s2, ':READ?');
    catch, fprintf(s2, ':READ?');
    end;
    while s2.BytesAvailable < 70,
    end;

    temp = fscanf(s2);
    output(i) = str2num( temp(15:27) );
```

```
end;
```

F.3 KeithleySetVoltage.m

```
function output = KeithleySetVoltage(voltage);
% Syntax: output = KeithleySetVoltage(voltage);
%
% Set the voltage of the SourceMeter.

global s2;

try, fprintf(s2, [':SOUR:VOLT:LEV:IMM:AMPL ' num2str(voltage)]);
catch, fprintf(s2, [':SOUR:VOLT:LEV:IMM:AMPL ' num2str(voltage)]);
end;
```

F.4 SVM2Init.m

```
% SVM Chip Version 2 Initialization Script
%
% Last modified: 2007-09-13

global ref integvpb1 integvpb2 integvnb1 integvnb2 integvref integvcmp;
global integrefcurrent inteqinvbias gamma1 gamma2 eeprombias cellbias;
global xin;

global currentadcoffset;

%Input and Bias Initial Values
ref             = 2.0;
%ref             = 2.2;
integvpb1       = 2.0;
integvpb2       = 1.4;
integvnb1       = 1.1;
integvnb2       = 1.7;
integvref       = 0.7;
%integvref       = 0;
integvcmp       = 1.0;
integrefcurrent = 0.65;
inteqinvbias    = 1;
gamma1          = 0.6;
gamma2          = 0.2;
eeprombias      = 1.5;
cellbias        = 1.5;
xin = 3.3*ones(1,14);

% Print the current state of the biases

fprintf ( 'REF             = %1.4f\n', ref );
fprintf ( 'INTEGVPB1       = %1.4f\n', integvpb1 );
fprintf ( 'INTEGVPB2       = %1.4f\n', integvpb2 );
fprintf ( 'INTEGVNB1       = %1.4f\n', integvnb1 );
fprintf ( 'INTEGVNB2       = %1.4f\n', integvnb2 );
fprintf ( 'INTEGVREF       = %1.4f\n', integvref );
fprintf ( 'INTEGVCMP       = %1.4f\n', integvcmp );
fprintf ( 'INTEGREFCURRENT = %1.4f\n', integrefcurrent );
fprintf ( 'INTEGINVBIAS    = %1.4f\n', inteqinvbias );
fprintf ( 'GAMMA1           = %1.4f\n', gamma1 );
fprintf ( 'GAMMA2           = %1.4f\n', gamma2 );
fprintf ( 'EEPROMBIAS       = %1.4f\n', eeprombias );
fprintf ( 'CELLBIAS         = %1.4f\n', cellbias );

FPGAInit;
FPGADigitalIO( 'M10', 'out', 1); % Integrator Measure
FPGADigitalIO( 'A7', 'out', 1);  % Integrator Reset

FPGAWriteMemoryBlock(1,5000,0);

%KeithleyInit;

FPGASetBias(33,integrefcurrent); % INTEGREFCURRENT (Pin 14)
FPGASetBias(32,integvcmp);       % INTEGVCMP (Pin 13)
FPGASetBias(31,integvref);       % INTEGVREF (Pin 12)
FPGASetBias(30,integvnb2);       % INTEGVNB2 (Pin 11)
FPGASetBias(29,integvnb1);       % INTEGVNB1 (Pin 10)
FPGASetBias(28,integvpb2);       % INTEGVPB2 (Pin 9)
FPGASetBias(27,integvpb1);       % INTEGVPB1 (Pin 8)
FPGASetBias(26,ref);             % REF (Pin 4)
FPGASetBias(18,integinvbias);    % INTEGINVBIAS (Pin 22)
FPGASetBias(17,gamma1);          % GAMMA1 (Pin 23)
FPGASetBias(16,gamma2);          % GAMMA2 (Pin 24)
FPGASetBias(15,eeprombias);      % EEPROMBIAS (Pin 25)
FPGASetBias(14,cellbias);        % CELLBIAS (Pin 26)

% Initialize Input Vectors

FPGASetBias(13, xin( 1));        % Pin 27
FPGASetBias(12, xin( 2));        % Pin 28
FPGASetBias(11, xin( 3));        % Pin 29
FPGASetBias(10, xin( 4));        % Pin 30
FPGASetBias( 9, xin( 5));        % Pin 31
FPGASetBias( 8, xin( 6));        % Pin 32
FPGASetBias( 7, xin( 7));        % Pin 33
FPGASetBias( 6, xin( 8));        % Pin 34
FPGASetBias( 5, xin( 9));        % Pin 35
FPGASetBias( 4, xin(10));        % Pin 36
FPGASetBias( 3, xin(11));        % Pin 37
FPGASetBias( 2, xin(12));        % Pin 38
FPGASetBias( 1, xin(13));        % Pin 39
FPGASetBias( 1, xin(14));        % Pin 40


FPGAReadCurrent(1,1,10);
currentadcoffset = mean(FPGAReadCurrent(9,1,20));
FPGAReadCurrent(1,1,20);
```

F.5 SVM2SelectCell.m

```
function SVM2SelectCell( cell )
% Syntax: SVM2SelectCell( cell )
%
% Select one of the internal 515 floating-gate cells.

FPGAWriteMemoryBlock(1,515,0);
FPGAWriteMemory(cell,1);
%FPGAWriteMemory(floor((cell-1)/14) + 488,1);
```

```
FPGASerialShift(1,515);
%FPGASerialShift(1,515);
pause(.5)
```

F.6 SVM2GetCurrent.m

```
function output = SVM2GetCurrent(cell)
% Syntax: output = SVM2GetCurrent(cell)
%
% Read the current from a floating-gate cell and return it in 'output'
% while using the on-board current ADC.
%
% Last modified: 2007-09-13

global eeprombias cellbias;

eeprombias      = 3.3;
cellbias        = 3.3;

FPGASetBias(15,eeprombias);        % EEPROMBIAS (Pin 25)
FPGASetBias(14,cellbias);          % CELLBIAS (Pin 26)

FPGAWriteMemoryBlock(1,5000,0);

SVM2SelectCell( cell );
pause(.5)
output = FPGAEstimateCurrent(1,20);

eeprombias      = 1.5;
cellbias        = 1.5;

FPGASetBias(15,eeprombias);        % EEPROMBIAS (Pin 25)
FPGASetBias(14,cellbias);          % CELLBIAS (Pin 26)
```

F.7 SVM2GetCurrents.m

```
% This script acquires floating-gate cell currents and stores them in the
% variable 'data'.

global cellbias;

for i=1:100,
    SVM2SelectCell(i);
    cellbias = 0;
    FPGASetBias(14,cellbias);          % CELLBIAS (Pin 26)
    cellbias = 3.3;
    FPGASetBias(14,cellbias);          % CELLBIAS (Pin 26)
  data(i) = FPGAEstimateCurrent(1,20);
    %data(i) = mean(KeithleyGetCurrent(5));
end;
```

F.8 SVM2SetCurrent.m

```
function injectionrate = SVM2SetCurrent( cell, value );
% Syntax: injectionrate = SVM2SetCurrent( cell, value );
%
% Set the specified cell to the targeted current where 'value' is
% given in Amperes.

global eeprombias cellbias s s2;

cellbias = 0;
FPGASetBias(14,cellbias);        % CELLBIAS (Pin 26)
cellbias = 3.3;
FPGASetBias(14,cellbias);        % CELLBIAS (Pin 26)

SVM2SelectCell(cell);
injmod = 0;
pulses = 0;
current = mean(KeithleyGetCurrent(5))
if current < 70e-9,  %Determine injection pulse width modifier
    injmod = 0;
elseif current < 100e-9,
    injmod = 1;
elseif current < 150e-9,
    injmod = 2;
else
    injmod = 4;
end;

injectionrate(pulses+1,1) = current;
injectionrate(pulses+1,2) = 0;

if current < value,
    while value > current,
        if current < value - 10e-9,
            pulsewidth = FPGAInjectPulse(22 - injmod);
        elseif current < value - 7e-9,
            pulsewidth = FPGAInjectPulse(21 - injmod);
        elseif current < value - 5e-9,
            pulsewidth = FPGAInjectPulse(20 - injmod);
        elseif current < value - 2e-9,
            pulsewidth = FPGAInjectPulse(19 - injmod);
        elseif current < value - 1e-9,
            pulsewidth = FPGAInjectPulse(18 - injmod);
        elseif current < value - .5e-9,
            pulsewidth = FPGAInjectPulse(17 - injmod);
        elseif current < value - .25e-9
            pulsewidth = FPGAInjectPulse(16 - injmod);
        else FPGAInjectPulse(15 - injmod);
        end;
        pulses = pulses + 1;
        current = KeithleyGetCurrent(1);
        injectionrate(pulses+1,1) = current;
        injectionrate(pulses+1,2) = pulsewidth;
        if current < 50e-9,
            injmod = 0;
        elseif current < 100e-9,
            injmod = 1;
        elseif current < 150e-9,
            injmod = 2;
        else
            injmod = 4;
        end;
    end;
```

```
end;
fprintf( 'Programmed current in %i pulses: %1.6f nA.\n', ...
    pulses, current*1e9 );
pause(0.5);
```

F.9 SVM2SetCurrents.m

```
% Script used to equalize an array of floating gates.

value = 35e-9;

for i=301:392,
  SVM2SetCurrent(i,value);
end;
```

F.10 SVM2InputSweep.m

```
% Last modified: 2007-09-05

global eeprombias cellbias;

eeprombias      = 3.3;
cellbias        = 3.3;
gamma1          = 2;

FPGASetBias(15,eeprombias);       % EEPROMBIAS (Pin 25)
%FPGASetBias(14,cellbias);         % CELLBIAS (Pin 26)
%FPGASetBias(17,gamma1);
SVM2SelectCell(1);

j = 1;
for i=3.3:-.01:1.75,
    FPGASetBias(14,0);          % CELLBIAS (Pin 26)
    FPGASetBias(14,3.3);          % CELLBIAS (Pin 26)
    FPGASetBias(13,i);
    sweepdata5(j) = mean(KeithleyGetCurrent(5));
    j = j + 1;
end;
eeprombias = 1.5;
FPGASetBias(15,eeprombias);
```

F.11 SVM2CurrentRampTest.m

```
% Script to set a targeted set of currents across the array specified in
% 'i' using the SVM2SetCurrent function. This script has also been used to
% configure other arbitrary waveforms on-chip.

global s s2;

for i=91:100,
    %SVM2SetCurrent(i,sin(((i)-90)/10*pi)*.5e-9+60e-9);
    SVM2SetCurrent(i,i*1e-9);
end;
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] C. Mead and M. Ismail, *Analog VLSI Implementation of Neural Systems*, Springer, 1989.

[2] A. H. Kramer, "Array-Based Analog Computation," IEEE Micro, Vol. 16, No. 5, Oct. 1996, pp. 4049.

[3] R. Genov, G. Cauwenberghs "Charge-Mode Parallel Architecture for Matrix-Vector Multiplication," IEEE T. Circuits and Systems II, vol. 48 (10), pp. 930-936, 2001.

[4] R. Sarpeshkar, C. Salthouse, J. Sit, M. W. Baker, S. M. Zhak, T. K. Lu, L. Turicchia, and S. Balster, "An Ultra-Low-Power Programmable Analog Bionic Ear Processor," IEEE Transactions on Biomedical Engineering, Vol. 52, No. 4, Apr. 2005, pp. 711-727.

[5] J. P. Lazzaro, S. Ryckebusch, M. A. Mahowald, and C. A. Mead, "Winner-Take-All Networks of O(N) Complexity," Caltech Computer Science Department Technical Report, Caltech-CS-TR-21-88, 1989.

[6] R. R. Harrison, J. A. Bragg, P. Hasler, B. A. Minch, and S. P. Deweerth, "A CMOS Programmable Analog Memory-Cell Array Using Floating-Gate Circuits," IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 48, No. 1, Jan. 2001.

[7] P. Hasler "Overview of Floating-Gate Devices, Circuits, and Systems," IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 48, No. 1, Jan. 2001.

[8] T. S. Hall, C. M. Twigg, J. D. Gray, P. Hasler, D. V. Anderson, "Large-Scale Field-Programmable Analog Arrays for Analog Signal Processing," IEEE Transactions on Circuits and Systems I: Regular Papers, Vol. 52, No. 11, Nov. 2005.

[9] E. Lee and P.G. Gulak, "A CMOS Field Programmable Analog Array," IEEE Journal of Solid State Circuits, Vol. 26, No. 12, Dec. 1991, pp. 1860-1867.

[10] V. Gaudet and P. G. Gulak, "Implementation Issues for High-Bandwidth Field-Programmable Analog Arrays," Journal of Circuits, Systems, and Computers Special Issue on Analog and Digital Arrays, World Scientific Publishing, Vol. 8, No. 5-6, 1998, pp. 541-558.

[11] M. Kucic, P. Hasler, J. Dugger, and D. V. Anderson, "Programmable and adaptive analog filters using arrays of floating-gate circuits," Proc. of 2001 Conference on Advanced Research in VLSI, Mar. 2001, pp. 148-162.

[12] L.R. Carley, "Trimming Analog Circuits Using Floating-Gate Analog MOS Memory," IEEE Journal of Solid-State Circuits, Vol. 24, No. 6, Dec. 1989, pp. 1569-1575.

[13] Y. L. Wong, M. H. Cohen, and P. A. Abshire, "A Floating-Gate Comparator With Automatic Offset Adaptation for 10-bit Data Conversion," IEEE Transactions on Circuits and Systems I: Regular Papers, Vol. 52, No. 7, Jul. 2005, pp. 1316-1326.

[14] T. S. Hall, P. Hasler, D. V. Anderson, "Field-Programmable Analog Arrays: A Floating-gate Approach," 12th International Conference on Field Programmable Logic and Applications. Montpellier, France. Sept. 2002.

[15] C. J. C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," Data Mining and Knowledge Discovery, Vol. 2, No. 2, Jun. 1998, pp. 121-167.

[16] E. Osuna, R. Freund, and F. Girosi, "Training Support Vector Machines: An Application To Face Dectection," Proc. Computer Vision and Pattern Recognition, 1997, pp. 130-136.

[17] A. K. Jain, *Fundamentals of Digital Image Processing*, Englewood, Cliffs, NJ: Prentice-Hall, 1989.

[18] A. Ganapathiraju, J. E. Hamaker, and J. Picone, "Applications of Support Vector Machines to Speech Recognition," IEEE Transactions on Signal Proc., Vol. 52, No. 8, Aug. 2004, pp. 2348-2355.

[19] V. Venkataramani, S. Chakrabartty, and W. Byrne, "Gini-Support Vector Machines for Segmental Minimum Bayes Risk Decoding of Continuous Speech," Computer Speech and Language, Vol. 21, No. 3, Jul. 2007, pp. 423-442.

[20] V. Wan and S. Renals, "Speaker Verication Using Sequence Discriminant Support Vector Machines," IEEE Transactions on Speech and Audio Processing, Vol. 13, No. 2, Mar. 2005, pp. 203-210.

[21] L. Wang, *Support Vector Machines: Theory And Applications*, Springer, 2005, pp. 370.

[22] C. E. Priebe, "Olfactory Classification via Interpoint Distance Analysis," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 23, No. 4, Apr. 2001, pp. 404-413.

[23] G. Xu, W. Tian, Z. Jin, and L. Qian, "Temperature Drift Modelling and Compensation for a Dynamically Tuned Gyroscope by Combining WT and SVM Method," Measurement Science and Technology, 18 (2007), IOP Publishing Ltd, pp. 1425-1432.

[24] P. C. Moster, "Gear Fault Detection and Classification Using Learning Machines," Sound and Vibration Magazine, Mar. 2004.

[25] X. Chen, R. Harrison, and Y. Zhang, "Genetic Fuzzy Fusion of SVM Classifiers for Biomedical Data," IEEE Congress on Evolutionary Computation, Sept. 2005, Vol. 1, pp. 654-659.

[26] B. Boser, I. Guyon, and V. Vapnik, "A training algorithm for optimal margin classifier," in Proc. 5th Annu. ACM Workshop Computational Learning Theory, 1992, pp. 144-52.

[27] P. Kucher and S. Chakrabartty, "An Energy-Scalable Margin Propagation-Based Analog VLSI Support Vector Machine," IEEE International Symposium on Circuits and Systems, 2007, 27-30 May 2007, pp. 1289-1292.

[28] S. Chakrabartty and G. Cauwenberghs, "Sub-Microwatt Analog VLSI Support Vector Machine for Pattern Classification and Sequence Estimation," Adv. Neural Information Processing Systems (NIPS'2004), Cambridge: MIT Press, 17, 2005.

[29] C. Mead, "Neuromorphic Electronic Systems," Proceedings of the IEEE, Vol. 78, No. 10, Oct. 1990, pp. 1629-1636.

[30] A. Bandyopadhyay, P. Hasler, and D. Anderson, "A CMOS Floating-Gate Matrix Transform Imager," IEEE Sensors Journal, Vol. 5, No. 3, Jun. 2005, pp. 455-462.

[31] Y. L. Wong and P. A. Abshire, "A 144x144 Current-Mode Image Sensor with Self-Adapting Mismatch Reduction," IEEE Transactions on Circuits and Systems I, Vol. 54, No. 8, Aug. 2007, pp. 1687-1697.

[32] W. D. Brown and J. E. Brewer, *Nonvolatile Semiconductor Memory Technology*, IEEE Press, 1998.

[33] AMI Semiconductor, "Process Specification Sheet: Process Technology 0.5μm," 2003.

[34] A. Bandyopadhyay, G. J. Serrano, and P. Hasler, "Adaptive Algorithm Using Hot-Electron Injecton for Programming Analog Computational Memory Elements Within 0.2% of Accuracy Over 3.5 Decades," IEEE Journal of Solid-State Circuits, Vol. 41, No. 9, Sept. 2006, pp. 2107-2114.

[35] V. Srinivasan, G. J. Serrano, J. Gray, and P. Hasler, "A Precision CMOS Amplifier Using Floating-Gate Transistors for Offset Cancellation," IEEE Journal of Solid-State Circuits, Vol. 42, No. 2, Feb. 2007, pp. 280-291.

[36] P. E. Allen and D. R. Holberg, *CMOS Analog Circuit Design, Second Edition*, Oxford University Press, New York, 2002.

[37] S. Shah and S. Collins, "A Model for Temperature Insensitive Trimmable MOSFET Current Sources," IEEE Transactions on Circuits and Systems II: Express Briefs, 2007.

[38] F. Adil, G. Serrano, and P. Hasler, "Offset Removal Using Floating-Gate Circuits for Mixed-Signal Systems," Proc. IEEE Southwest Symp. Mixed-Signal Design, Las Vegas, NV, Feb. 2003, pp. 190195.

[39] H. Spieler, "Radiation Detectors and Signal Processing - VII. Why Things Don't Work," University of Heidelberg, Oct. 2001.

[40] D. L. Jones, "PCB Design Tutorial," Rev. A, 29 Jun. 2004, pp. 17-18.

[41] Sunstone Circuits, "Material Specifications," Online, Accessed 4 Oct. 2007.

[42] National Semiconductor Corp., "LM1086 1.5A Low Dropout Positive Regulators," DS100948, Jun. 2005.

[43] Maxim, "MAX761/MAX762: 12V/15V or Adjustable, High-Efficiency, Low I_Q, Step-Up DC-DC Converters," 19-0201, Rev 0, Nov. 1993.

[44] Texas Instruments, Inc., "UCC27321, UCC27322, UCC37321, UCC37322 Single 9-A High Speed Low-Side MOSFET Driver with Enable," SLUS504C, Sept. 2002.

[45] Maxim, "MAX1680/MAX1681: 125mA, Frequency-Selectable, Switched-Capacitor Voltage Converters," 19-1247, Rev 0, Jul. 1997.

[46] Texas Instruments, Inc., "OPA743, OPA2743, OPA4743: 12V, 7MHz, CMOS, Rail-to-Rail I/O Operational Amplifiers," SBOS201, May 2001.

[47] Linear Technology Corporation, "LTC2600/LTC2610/LTC2620 Octal 16-/14-/12-Bit Rail-to-Rail DACs in 16-Lead SSOP," 2600fa, LT/TP 1103 1K Rev A, 2003.

[48] Linear Technology Corporation, "LT1461 Micropower Precision Low Dropout Series Voltage Reference Family," 1461f LT/LCG 0800 4K, 1999.

[49] Linear Technology Corporation, "LTC2414/LTC2418 8-/16-Channel 24-Bit No Latency $\Delta\Sigma$ ADCs," 241418fa, LT1105 Rev. A, 2005.

[50] P. Khairolomour, G. Leung, A. Li, "Precision Programmable Current Sources Use Digital Pots," Electronic Design, #9944, 31 Mar. 2005.

[51] Analog Devices, Inc., "REF19x Series: Precision Micropower, Low Dropout Voltage References," C00371-0-10/06(I), Rev I, 2006.

[52] Analog Devices, Inc., "AD7376: +30 V/±15 V Operation 128-Position Digital Potentiometer," Rev. B, Mar. 2007.

[53] Analog Devices, Inc., "ADG714/ADG715: CMOS, Low Voltage Serially Controlled, Octal SPST Switches," Rev. B, 2002.

[54] Texas Instruments, Inc., "TLC225x, TLC225xA Advanced LinCMOS Rail-to-Rail Very Low-Power Operational Amplifiers," SLOS176D, Mar. 2001.

[55] Linear Technology Corporation, "LTC2415/LTC2415-1 24-Bit No Latency $\Delta\Sigma$ ADCs with Differential Input and Differential Reference," sn2415 sn24151fs, LT/TP 0202 2K, 2001.

[56] Xilinx, Inc., "Spartan-3 FPGA Family: Complete Data Sheet," DS099 (v2.2), 25 May 2007.

[57] Xilinx, Inc., "Spartan-3 Starter Kit Board User Guide," UG130 (v1.1), 13 May 2005.

[58] Xilinx, Inc., "Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs," XAPP462 (v1.1), 5 Jan. 2006.

[59] Microchip Technology, Inc., "PIC18F2455/2550/4455/4550 Data Sheet," DS39632D, 30 Jan. 2007.

[60] Axelson, Jan, *USB Complete, Third Edition*, Lakeview Research LLC, pp. 28-29, 2005.