UNIVERSITY OF CALIFORNIA

Los Angeles

FPGA Implementation of Network Optimization for Flash ADC Calibration

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Electrical Engineering

By

Yuta Toriyama

2011

The thesis of Yuta Toriyama is approved.

_____

Babak Daneshrad

_____

Chih-Kong Ken Yang

_____

Dejan Marković, Committee Chair

University of California, Los Angeles

2011

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Professor Dejan Marković. His faith in my ability to succeed has brought me here today. I also wish to thank Professor Chih-Kong Ken Yang and Professor Babak Daneshrad for being on my thesis committee. Their helpful and thoughtful comments are definitely appreciated.

I am also grateful for my many group members. Richard Dorrance, Henry Chen, Kevin Dwan, and Qian Wang have all been helpful in class and in research. We have made it through tough times together. The other members of the group, Cheng Wang, Vaibhav Karkare, Sarah Gibson, Rashmi Nanda, Fengbo Ren, Tsung-Han Yu, Fang-Li Yuan, Chia-Hsiang Yang, and Vicki Wang were great engineers, mentors, and friends, all of whom were very helpful when I need any kind of assistance.

I sincerely thank my mother, father, and sister for their never-ending support. There is always so little I can do for them but so much they do for me. This thesis is a result of their endurance through all of the trouble I have given them since I was born, and therefore to them this thesis is dedicated.

ABSTRACT OF THE THESIS

FPGA Implementation of Network Optimization for Flash ADC Calibration

by

Yuta Toriyama

Master of Science in Electrical Engineering

University of California, Los Angeles, 2011

Professor Dejan Marković, Chair

Low-power and low-area flash ADC architectures are extremely susceptible to offset variations, and a calibration technique to alleviate this effect is necessary. An FPGA implementation of a calibration algorithm for a probabilistic flash ADC architecture is presented as a yield optimizing solution. Given a fabricated set of comparator thresholds, the calibration will optimize for the lowest quantization noise for a known input signal probability density function. The hardware implementation is optimized for speed and benefits not only from parallelism but also from the proximity of the FPGA board to the actual chip, as opposed to an ordinary computer. The presented FPGA implementation, including the formulation of a linear program and solving it via the simplex algorithm, is an integral part of guaranteeing a low power and low area flash ADC to be functional, providing a solution with time savings relative to any existing methods that could potentially be used to calibrate the flash ADC.

CHAPTER I

Introduction

1.1 Motivation

Modern applications drive many of the current innovations in circuit design. Increasingly many applications place stringent requirements on power and area, and Analog to Digital Converter (ADC) architectures are no exception. For example, one such application is data converters on implantable neural devices. The primary concern of the digitizer on an implantable chip is to minimize the power density and to minimize the area consumption, while achieving the fastest possible sampling rate. This allows for the maximum number of data channels to be digitized given the limited power and area budget. A flash ADC architecture as a choice of digitization is appealing because of its inherent speed of operation. However, a flash ADC with low power and area cannot be built in a straightforward manner because of increased CMOS variability. Existing solutions to this problem use a combination of design techniques such as large device sizing or redundancy to improve the yield.

In this work, an optimized system-level hardware implementation of a calibration technique for a redundancy flash ADCs is presented. The technique includes the generation of a network representation of a flash ADC and a corresponding linear programming problem that is solved via the simplex method [1]. This technique takes

advantage of design tradeoffs to improve the yield. The implementation on an FPGA, which will be called the "network solver," formulates the linear programming problem and solves it to optimize the flash ADC, allowing for a seamless integration of the system as a whole.

1.2 Flash ADCs with Large Offsets

An extremely important consideration in the design of modern chips is the power density. For the example of an implantable neural chip, a power density of 800 $\mu W/mm^2$ has been shown to damage brain cells [2], making it necessary to minimize the power density to a level much less than this upper limit. Also, chips often have limited sources of energy for operation, for example in battery operated mobile devices. Maximizing the battery lifetime is another concern which leads to the necessity of minimizing the power consumption of chip circuitry. Furthermore, modern circuits must strive to take up as little area as possible to drive down their costs. This, however, is in direct opposition to the need for larger devices to mitigate variation in analog circuits and is therefore problematic. Large offsets in flash ADCs arise from these various factors, and these offsets only become worse with technology and supply voltage scaling.

The effects of large variation on yield, due to low power and small area, are further investigated. In an ideal flash ADC, the thresholds at which the comparators toggle, $V_{i,ideal}$, are evenly spaced between $V_{1,ideal} = 0$ and $V_{N,ideal} = V_{FS}$, the full scale voltage. Realistically, however, each threshold, V_i, is a random variable whose value is

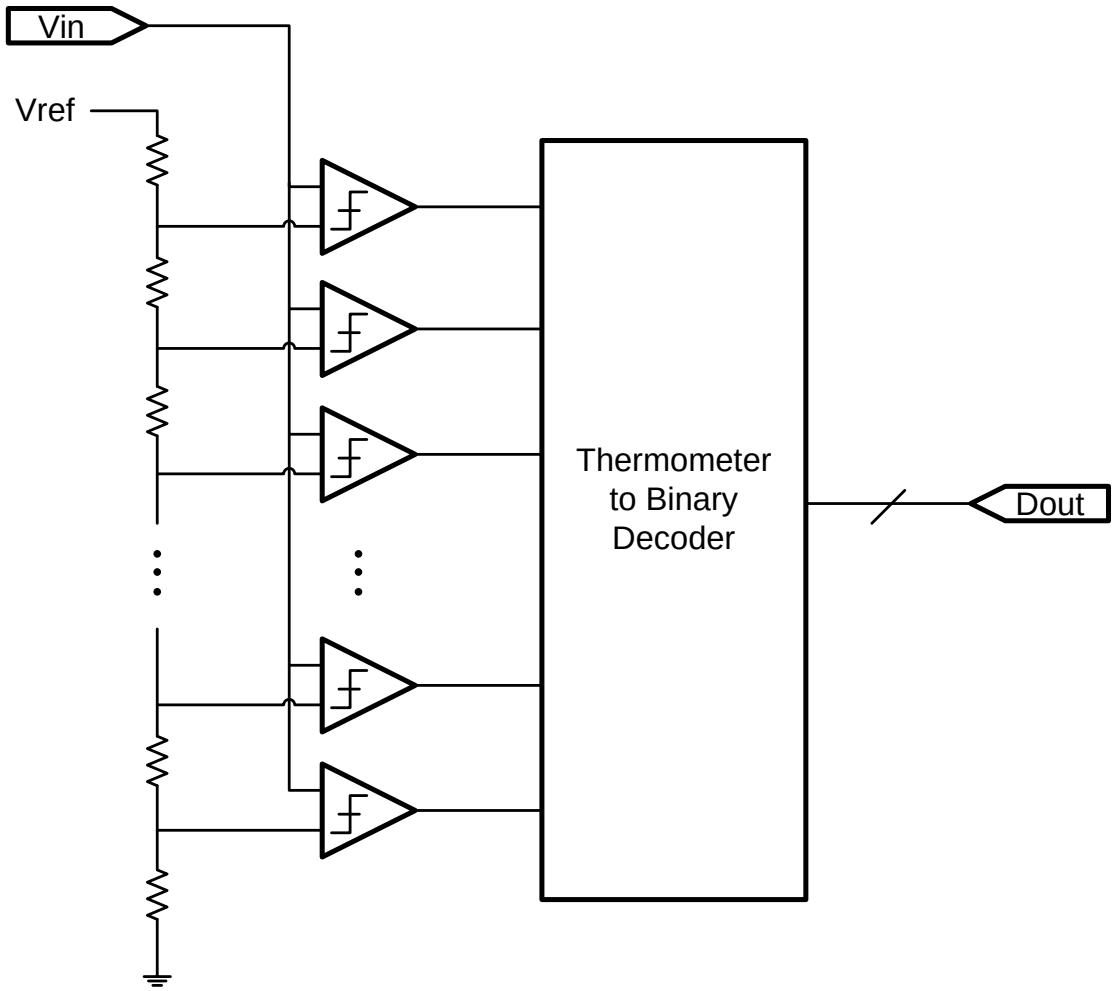


Figure 1.1: Classic Flash ADC Architecture

only known after fabrication (V_1 and V_N are assumed to be 0 and V_{FS}, respectively).

Each of these random variables can be assumed to be independent and identically distributed (i.i.d.) Gaussian random variables. This is because the variation of the threshold values at which each comparator switches is a cumulative effect of many contributing factors, such as resistive ladder mismatch and comparator input offset [3]. If the variance of these random variables is small, the LSB and thus the effective number of bits (ENOB) of the data converter is limited by this variation, since

the variation determines how finely the analog voltage levels are divided.

However, once variations start to grow, it is possible for the comparators to output a non-thermometer coded value, due to the fact that the thresholds at which each comparator switches are no longer in order. For single errors, bubble correction logic can be placed between the comparators and the encoder to ensure a thermometer code input to the encoder. However, beyond a single error, the flash ADC fails to function properly. Thus, the yield of the flash ADC can be directly correlated with the distribution of the i.i.d. Gaussian random variables representing the thresholds of the ADC.

The yield for a flash ADC with no correction mechanisms can be defined as the probability that the random threshold values are monotonically increasing. The monotonic increase ensures that the flash ADC will always operate correctly (although it is possible to imagine extremely rare scenarios for which the ENOB of digitization is greatly sacrificed). Mathematically, this probability can be written as:

$$\begin{aligned} \text{Yield}_N &= p(V_1 < V_2 < \cdots < V_N) \\ &= p(V_1 < V_2)p(V_2 < V_3 | V_1 < V_2) \cdots p(V_{N-1} < V_N | V_1 < \cdots < V_{N-1}) \end{aligned} \tag{1.1}$$

Some calculate this by assuming that each probability is independent, and that the yield is the product of the probabilities of consecutive threshold values being in a monotonic order [4]. However, this is an approximation and becomes inaccurate for random variables with large variances.

The yield probability as a function of standard deviation is shown in Figure 1.2

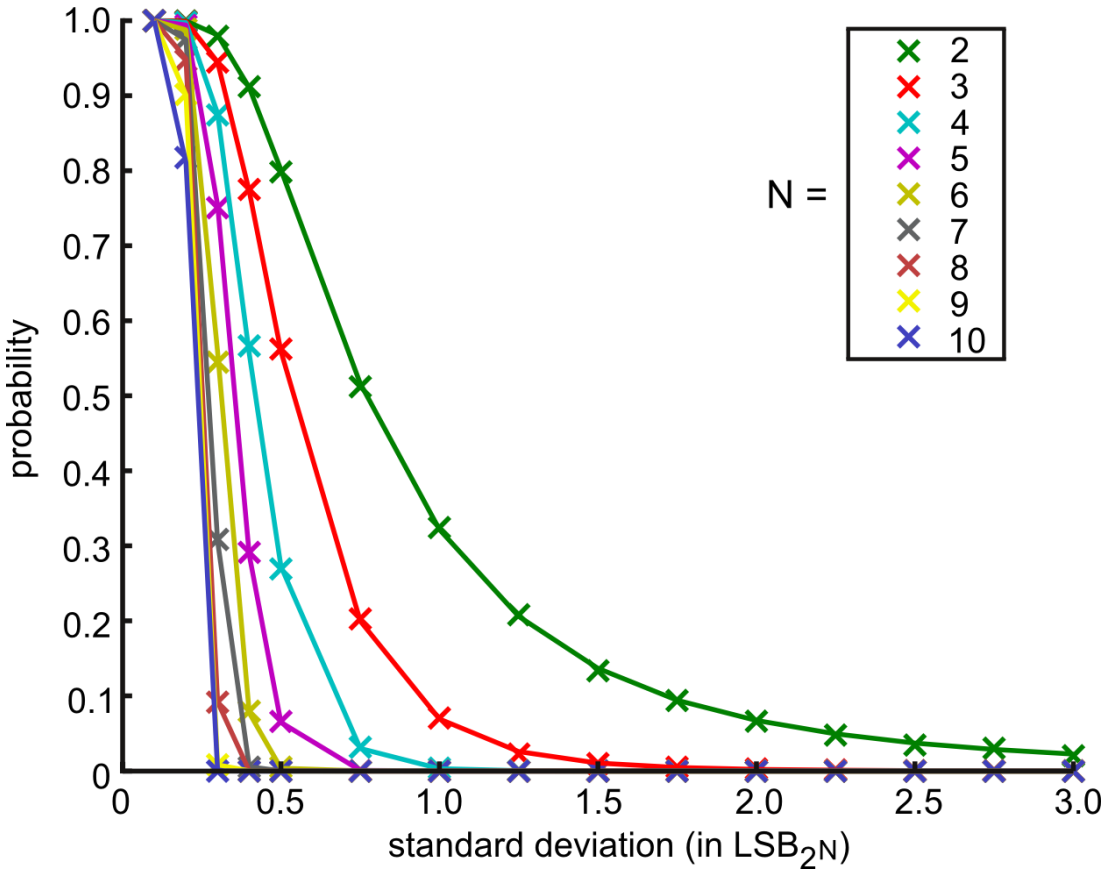


Figure 1.2: Yield vs. Standard Deviation for N-bit Flash ADCs [5]

[5]. The yield drops off very quickly, especially for flash ADCs with larger number of bits.

1.3 Previous Work

ADCs that have been implemented on power and area limited designs, such as implantable neural chips, are generally of a different architecture, such as that of a SAR ADC, to decrease the power consumption [6]. However, these ADCs cannot operate at

very fast frequencies, and thus many of them must be placed in parallel to achieve the required data conversion rate, leading to a penalty in area.

A popular technique to mitigate the problematic yield of flash ADCs is redundancy [7] [8]. While placing redundancy in the comparators of flash ADCs can improve the yield, an increasing amount of redundancy is required for an increasing amount of variation, leading to a corresponding cost in power and area. Furthermore, redundancy often complicates the architecture, because the comparator outputs are no longer thermometer codes; complicated logic structures such as Wallace adders become necessary, further increasing the power and area consumptions. Other techniques such as offset compensation [9], while effective in mitigating variation, remain costly to implement in terms of power and area, making these techniques unattractive for implantable neural chips.

In terms of the network solver calibration technique, not much previous work has been found. Part of the requirement of the network solver is to have an implementation of the simplex algorithm that can handle linear programming (LP) problems with on the order of 10,000 variables. A hardware implementation of the simplex algorithm has been presented by Bayliss et al. [10] but does not meet the required problem size to target a reasonably-sized flash ADC, nor is it optimized for the particular problem type, as will be discussed later. Furthermore, a simplex solver, whether in hardware or software, only completes half of the task; the generation of the linear programming problem is necessary in order to meet the needs of the work in this thesis. Thus, so far the only potential solution to functionally implementing the

calibration technique resides in the software domain, causing unwanted delays in interfacing with the chip and the time to achieve the optimum calibration.

1.4 Organization of Thesis

This thesis outlines the flow of attaining the solution to the problem of achieving a low power and area flash ADC architecture via the implementation of the network solver in hardware. Many design choices have been made in order to optimize the hardware implementation so that a clear advantage over other solutions is attained. Chapter 2 presents the network solver yield optimization technique, including the formulation of the network representation of the flash ADC and solving the linear programming problem. Chapter 3 describes the network solver implementation methodology in detail, outlining the functionality along with the design decisions and optimizations. Finally, Chapter 4 presents the obtained results and concludes the thesis.

CHAPTER II

Linear Programming Problem Formulation

It is clear that fabrication of low power and low area flash ADCs without regard to the effects of variation results in unacceptably low yield. This chapter sheds light into the possibility of improving this yield by formulating a linear programming optimization problem.

2.1 Network Matrix and Cost Vector Formulation

The yield of the flash ADC drops drastically due to the necessity of all thresholds to be in monotonically increasing order. This leads directly to the idea of selectively turning on and off comparators such that the resulting ladder contains only monotonically increasing threshold values. This probabilistic approach, while degrading the granularity of quantization from the ideal case, can guarantee a thermometer coded output and thus functionality of data conversion for any distribution of thresholds. Put another way, this can be seen as a redundancy technique but with power to only the necessary comparators. However, unlike traditional redundancy architectures, a complicated digital logic block after the output of the comparators is not necessary because it is always guaranteed to be thermometer coded.

Each comparator can be seen as corresponding to a particular threshold value at which the output of that comparator will flip; this threshold value is the instance of the Gaussian random variable discussed in the previous chapter. This encompasses all of the physical variations occurring due to many factors such as input offset and resistor ladder offset, and characterizes it as a single number. The optimum choice of comparators to turn on (or off) is defined to be the one such that the signal to quantization noise ratio (SQNR), for a given input probability density function, is maximized. If the input signal characteristics are unknown or are not well characterized, SQNR can be maximized for a uniformly distributed input probability density function (pdf). In this manner, the problem of improving flash ADC yield can be transformed into the problem of optimally choosing comparator thresholds from a given random set.

An important abstraction in obtaining the optimum calibration method for this flash ADC is to view the thresholds as the nodes of a directed graph. For any arbitrary flash ADC, a corresponding directed graph can be created by forming an edge from one threshold node, V_i, to another, V_j, if $i < j$ and $V_i < V_j$. That is, an edge points from one node to another only if the threshold increases and is also supposed to increase.

A monotonic path from the lowest threshold to the highest can be traversed through the directional graph by following the edges from start to finish. Taking an edge from V_i to V_j corresponds, in the flash ADC, to turning off the comparators with indices between i and j. Thus, any path in this directed graph from node 1 to node N corresponds to a choice in comparators guaranteeing monotonicity and thus functionality of the flash ADC.

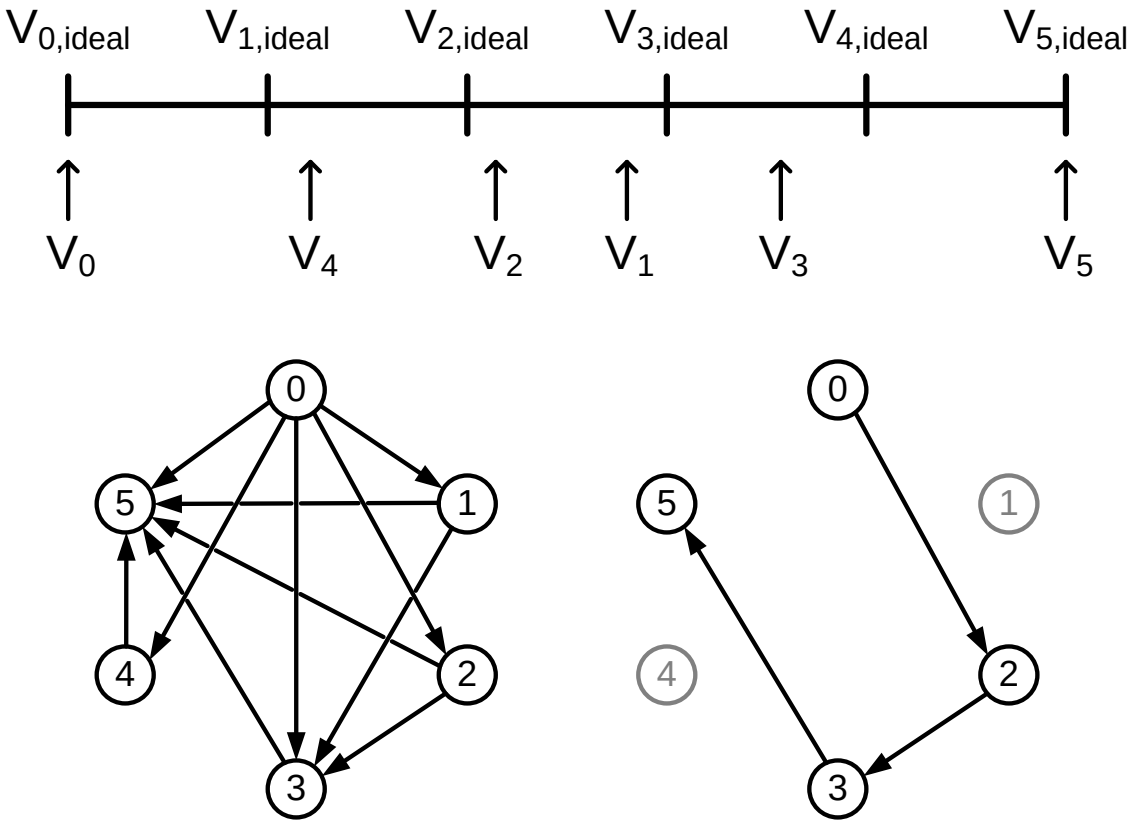


Figure 2.1: Example distribution of thresholds (top), a corresponding graph (lower left), and a graphical representation of a possible monotonic path (lower right)

An example of a distribution of thresholds is depicted in Figure 2.1. The ideal thresholds are shown as vertical bars on the horizontal line, but the actual thresholds that are fabricated are shown by the arrows. First, to generate a graph from this set of thresholds, all of the nodes are drawn. From node 0, edges extend to every other node because all of the other thresholds are higher than V_0. From node 1 edges only extend out to nodes 3 and 5 because V_3 and V_5 are higher than V_1, while V_2 and V_4 are lower,

when ideally all of these nodes should be higher. This process is repeated for all nodes until the entire graph is completed, as shown in the lower left of Figure 2.1. From this complete graph, any monotonic path from V_0 to V_5 can be extracted by traversing through the graph from node 0 to node 5, as shown in the lower right of Figure 2.1.

For each edge realized in this graph, a corresponding cost can be assigned. The cost incurred by choosing an edge is defined to be the quantization noise power contribution of the edge. The quantization noise power contribution of an edge from V_i to V_j is:

$$C_{ij} = \int_{V_i}^{V_j} \left(V_{in} - V_{i,ideal}\right) p(V_{in}) dV_{in} \tag{2.1}$$

where $p(V_{in})$ is the pdf of the input signal. This formulation thus allows for optimization for non-uniform input pdfs. This is beneficial for many applications, as with the running example of implantable neural chips. In terms of algorithm performance, a non-uniform quantization can actually outperform a uniform quantization [11]. Thus, calibration techniques should be flexible to allow for maximizing the performance as a function of the input signal pdf.

2.2 LP Transformation and the Simplex Algorithm

Any directed graph without self loops, as is the case with a graph generated from a flash ADC, can be represented as a node-edge incidence matrix. The incidence matrix A of a directed graph is an $N \times M$ matrix where the number of rows N is the number of nodes, and the number of columns M is the number of edges. For each edge k

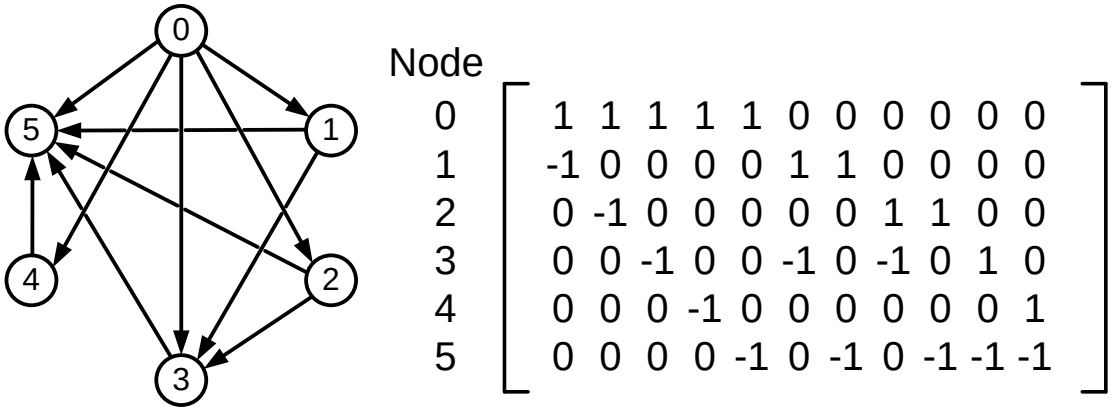


Figure 2.2: Matrix formulation from a given graph

leaving node i and entering node j, element a_{ki} is 1, element a_{kj} is -1, and a_{nm} is 0 otherwise.

It is intuitive to see that a monotonic path from node 1 to node N corresponds to a set of columns in this matrix A that sum up to the vector $[1\ 0\ 0\ \ldots\ 0\ 0\ -1]^T$. While there are many possible paths, there is one that is optimal in the sense that the sum of the costs incurred by each edge taken is minimized. These can be used as the foundation for creating an LP problem to find the optimal such path. In general, an LP is an optimization problem with the following format: choose the $M \times 1$ vector x so as to minimize the objective function

$$Z = c^T x \tag{2.2}$$

where c is an $M \times 1$ vector of costs, given the constraints

$$Ax \leq b \tag{2.3}$$

where A is an $N \times M$ matrix and b is an $N \times 1$ vector so that the vector x is given a set of N linear constraints.

The LP is formulated as follows: choose x so as to maximize the objective function $Z = c^T x$, where c is the vector of quantization noise power contribution associated with each arc column in the matrix A, such that $Ax = b$ and $x = \{0,1\}$, where x is an $M \times 1$ vector indicating which edges of the graph are chosen, and $b = [1\ 0\ \ldots\ 0\ -1]^T$. This formulation by itself, however, is a binary integer program (BIP) because of the binary constraints on each x_i. A BIP is a subset of integer linear programs (ILP), which in general is more complicated to solve than a regular LP. However, because of the method with which matrix A was formulated, it can be seen that A is totally unimodular. Namely, all elements in A are 0 or ± 1, and there are two nonzero elements in each column of which one is positive and the other is negative [12]. The total unimodularity of A guarantees that by replacing $x = \{0,1\}$ with $x \geq 0$, an optimal binary solution for which each value in x is equal to zero or one can be obtained by solving the problem as a standard form LP [13]. To be strictly precise, the very last row, or the very last equality constraint, must be negated so that the right hand side of the constraint is non-negative, but this does not affect our problem.

A standard form LP can be efficiently solved by a well known algorithm called the simplex algorithm. The reader is directed to Appendix A for a brief review of the simplex algorithm. Phase 1 of the simplex algorithm is to find an initial basic feasible solution. In general, for LPs with inequality constraints, the origin ($x = 0$) is a feasible cornerpoint and can be used as an initial basic feasible solution. However, because the constraint in the formulated LP for flash ADC calibration is comprised of equalities, the origin does not meet those constraints and thus extra steps must be taken in order to

initially search for a feasible solution. Phase 2 of the simplex algorithm is to iterate through the possible feasible solutions, decreasing the objective function value each time, until an optimum solution is found. The search for an initial solution can be implemented as another optimization problem for which the steps in Phase 2 can be conducted with a separate objective function. To summarize, starting from a graph representation of the flash ADC, a standard form LP has been formulated which can be solved to guarantee functionality and optimality of the ADC.

CHAPTER III

Hardware Realization

So far the problem formulation and solution for creating a flash ADC with guaranteed monotonic yield have been presented. In this chapter the solution is extended to include the implementation so that it may be presented as a complete entity from which the proposed flash ADC can be calibrated.

3.1 Justification of Implementation in Hardware

There exist many software packages, both commercial and open source, that solve LPs efficiently via the simplex method with many complicated optimizations in performance. However, for the purposes of this work, a hardware implementation of the search for the minimum cost path is desirable.

The main reason is the speed at which the solution is found. The implementation solution is targeted to solve a network generated from a flash ADC consisting of 2^{10} = 1024 comparators initially, equaling 1024 nodes in the directed graph representation. This equates to 1024 equality constraints in the LP. In addition, in the directed graph that is formulated for the flash ADC, the number of edges can explode drastically, because there can be edges between any two of the 1024 nodes in the graph. In the worst case, if all 1024 thresholds in the flash ADC are ideal and monotonic, then the

number of edges in the corresponding graph is 523,776. The number of edges in the graph translates into the number of variables in the LP, making the LP very large and thus slowing down any software solution. Because the size of the LP can be problematic in hardware as well, a threshold on the maximum cost for an edge allowed can be placed when generating the graph from the input thresholds. Even with this technique at hand, however, the LP remains very large. As an extreme example, MATLAB, running on a single thread 2GHz Intel Xeon processor with 4GB of RAM, using the optimization toolbox built-in function `bintprog`, is unable to solve an LP of the same type even a hundredth of this size because it runs out of memory (while the machine is a dual processor, quad core server with 16Gb of total RAM, the OS is a 32-bit Windows and MATLAB 2007 is single threaded, so it is assumed that MATLAB itself uses this much of the machine specs).

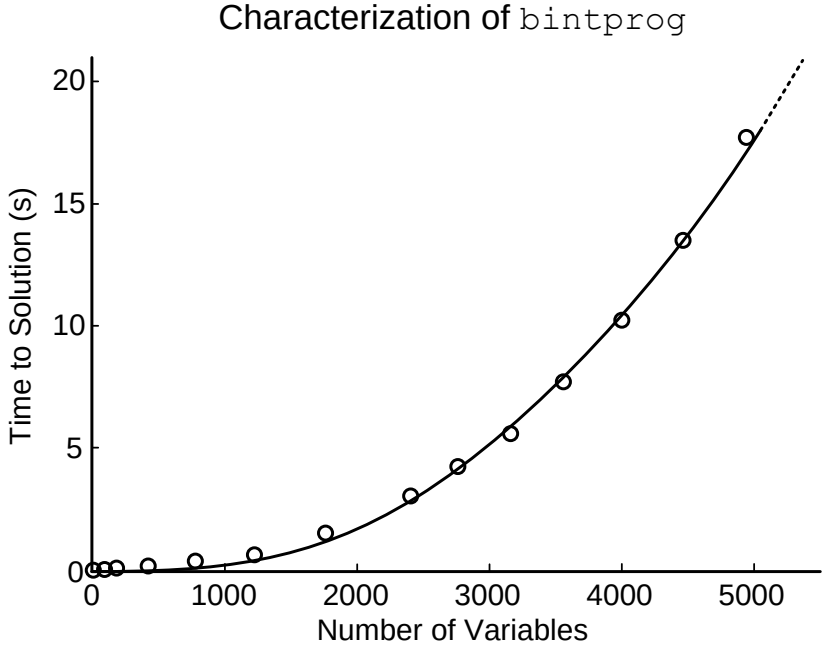


Figure 3.1 Characterization of speed to solution of the MATLAB `bintprog` *function*

Figure 3.1 shows the time the MATLAB function `bintprog` takes to finish various sized binary integer programming problems. The x-axis gives the number of variables in the LP, or the number of columns in the tableau, and the y-axis gives the time it took for the function to compute the answer, in seconds. The circles are the data points, and the curve is a fitted polynomial of degree 2. The rightmost circle corresponds to the maximum size problem that `bintprog` is able to solve. Since the curve fits well, it can be seen that if memory were not a constraint, the time for MATLAB to compute the answer will grow quadratically with the number of columns in the simplex tableau.

Another important but often overlooked reason for implementing this solution in hardware is the interfacing aspect between the actual flash ADC chip and the network solver implementation. For a software solution, there must exist a means of communication between the chip and the computer. Furthermore, after the data has been communicated to the computer, some software must first generate the network graph and the associated LP to feed in as an input to a simplex solver, which has a non-negligible time overhead. MATLAB, running on the machine stated above, takes half an hour just to generate the LP from a given set of thresholds, showing that a more optimal solution is necessary.

Therefore, it is of interest to us to implement the network solver on hardware, namely, an FPGA board. An ASIC solution is unnecessary in that only one implementation is necessary for all fabricated flash ADCs, and solving a large LP can quite possibly take up too much silicon area. However, while a hardware solution will

seemingly solve these problems, it is not trivial to implement such a solution. The main problem lies in the serial and iterative nature of the simplex algorithm. Hardware implementations of algorithms and calculations in general are sped up by taking advantage of parallelism and pipelining. However, an iteration of the simplex algorithm cannot begin until the previous iteration has completed, since a new entering basic variable cannot be chosen until the simplex tableau has been updated. As an example, Bayliss et al. [10] obtains increased throughput in their FPGA simplex algorithm implementation by pushing multiple LPs in the pipeline to solve, which heavily limits the size of solvable LPs making this unsuited for calibrating a sizeable flash ADC.

Speed improvements in the simplex method on hardware can still be expected, however, due to the calculation efficiency benefit of hardware over software. Also, a large speed improvement can be expected in the interfacing with the chip and setting up the LP for simplex execution before finally returning the correct solution. This is not a negligible point, in that the solution to the LP problem generates, as binary basic variables, the edges to be traversed for an optimal solution. However, what is actually needed to calibrate the flash ADC are the nodes through which the optimal path traverses. While the software solution of the LP must be reinterpreted to generate the list of comparators to turn on or off, a custom design of the calibration mechanism allows the output to already be in the desired format to feed into the flash ADC.

3.2 Functional Description and Design Considerations

The hardware to be designed must take in a vector of thresholds or nodes as an input and output as the solution which nodes to traverse in order to calibrate the given flash ADC. In the design of the stated FPGA implementation, it is important to consider not only what speed improvements can be made but also what resource optimizations can be made in order to make the solution scalable to larger LPs, which in the future may lead to being able to calibrate a larger flash ADC for more levels of quantization and possibly expand the application space.

Knowing that a large simplex tableau must be stored in order to perform the simplex algorithm, care must be taken in the choice of the method of storage. Thus, important characteristics of the tableau are analyzed in order to take advantage of them in the implementation.

An obvious observation of the starting simplex tableau is that it is sparse, because only two nonzero elements exist per column. However, this does not remain true throughout the simplex method. While the tableau can still be thought to be relatively sparse, it is difficult to analyze exactly how sparse the tableau will be, especially because this can change from problem to problem. Furthermore, a storage methodology that takes advantage of the sparseness of the tableau, for example by storing the coordinates of nonzero entries only, may lead to possible slowdowns in the computation time, due to the need to search through the entire memory of storage for every operation that needs to be conducted.

On the other hand, to store the elements of the simplex tableau, only two bits per element are necessary, because even throughout the Gaussian eliminations of the simplex method the elements in the tableau are guaranteed to be only zeros and positive or negative ones, again, due to the total unimodularity of the initial matrix. Thus, the decision has been made to store the simplex tableau in a relatively straightforward fashion, with coordinates of the matrix corresponding to address bits of the memory. An external 4MB quad data rate (QDR) SRAM is used to store the tableau, allowing for up to 2^{14} columns in the matrix or edges in the graph. Because the SRAM is a QDR, each address corresponds to two 32-bit words that are read out or written to in two consecutive cycles. Each word in the memory stores 16 elements, and consecutive words store rows of the simplex tableau. Thus the upper ten bits of a memory address represent the vertical coordinate *y* of the element in the tableau. The lower nine bits of the memory address, one bit to indicate which word of the address, and 4 bits to indicate which bit field of the word, correspond to the horizontal coordinate *x* of the element in the tableau.

Since it is plausible that more edges will exist in the incidence matrix, as

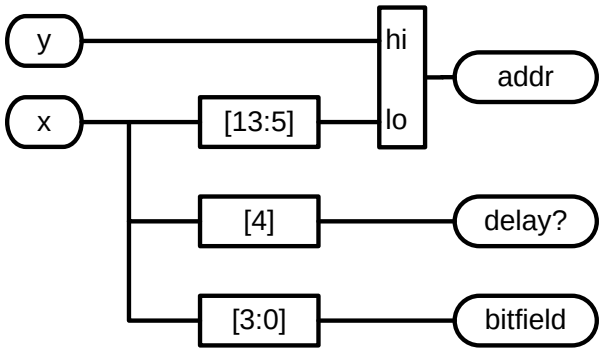


Figure 3.2: Coordinates of tableau mapping to address/word bit fields in memory

mentioned previously, a threshold maximum value for the cost of each arc is placed as a heuristic in order to eliminate edges that have too high of a cost and thus will most likely not be traversed. The cost vector is not stored in a separate memory.

An interesting and important optimization can be made with respect to setting up the Phase 1 LP. In general, with the addition of artificial variables r_n to find the initial feasible solution, the constraints are rewritten to look like:

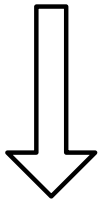
$$\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_m x_m + r_n = \beta_n \tag{3.1}$$

and the initial objective function of Phase 1 is to minimize *W*:

$$W = \sum_{i=1}^{m} r_i \tag{3.2}$$

where α_i are the elements in the *A* matrix, and β is the corresponding element in the *b* vector. This is so that when the Phase 1 cost is minimized, that equates to setting all of the artificial variables equal to zero, leading to an initial feasible solution, if one exists. Thus the simplex tableau is the original *A* matrix concatenated with the identity matrix. However, as the tableau has been set up as-is, it is not in proper form, because while the artificial variables are the initial basic variables with non-zero values, their corresponding columns have two non-zero entries (one in the constraints and one in the cost vector). To reduce the simplex tableau to a proper form, all of the constraint rows must be subtracted from the Phase 1 cost vector. Because the initial form of the simplex tableau is the node-edge incidence matrix and the last constraint row has been negated, this summation of all of the rows in the initial simplex tableau results in the Phase 1 cost vector equaling twice the last row. Furthermore, once the artificial variables leave

0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	W
c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	0	0	0	0	0	0	Z
1	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1
-1	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0
0	-1	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0
0	0	-1	0	0	-1	0	-1	0	1	0	0	0	0	1	0	0	0
0	0	0	-1	0	0	0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	0	1	0	1	0	1	1	1	0	0	0	0	0	1	1



0	0	0	0	-1	0	-1	0	-1	-1	-1	W’
c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	Z
1	1	1	1	1	0	0	0	0	0	0	1
-1	0	0	0	0	1	1	0	0	0	0	0
0	-1	0	0	0	0	0	1	1	0	0	0
0	0	-1	0	0	-1	0	-1	0	1	0	0
0	0	0	-1	0	0	0	0	0	0	1	0
0	0	0	0	1	0	1	0	1	1	1	1

Figure 3.3: Optimization with manipulation of artificial variables

the basis and become zero-valued, they will never re-enter the basis, because this would violate the convex nature of the feasible region. Each iteration of the simplex algorithm reduces the value of the minimization objective function, and this can only be done in Phase 1 by forcing the positively-valued artificial variables to zero. Thus, it is not necessary to store the last *N* columns of the simplex tableau that had been concatenated to the original matrix to include the artificial variable constraints. An example of this reduction of the tableau is shown in Figure 3.3.

Therefore, it is shown that the first N steps of Phase 1 are deterministic, and thus the LP can simply be set up to be at that iteration to begin with, saving on time as well as memory. In addition, because the actual value of the Phase 1 cost vector is not of any significance, the entire vector can be divided by two to simplify the Gaussian eliminations to occur in Phase 1, as well as save on memory. This optimization is possible due to the specific problem formulation at hand for the flash ADC.

Once the Phase 1 LP has been set up as above, the simplex method can be carried out. The same operations are conducted during Phase 1 and Phase 2 because of the introduction of artificial variables. This method, as opposed to other methods to find an initial solution like the big-M method, integrates well with the FPGA design in this thesis, since a simplex solver is already required to be implemented; the only necessary addition is a second cost vector containing the cost vector for the Phase 1 artificial variables.

To find the entering basic variable, the minimum value in a large vector must be found. A serial search will take a long time, but a tree structure requires a memory structure where each element can be tapped out, leading to a non-straightforward implementation. A tradeoff can be taken here so as to implement the search with basic RAM elements but does not take as long to find the solution. For example, the RAM used to store the non-basic variables can be split into two RAMs, and a serial search through each RAM can take place simultaneously, allowing the minimum value to be found in half the time and with little additional logic and latency.

The minimum ratio test can be implemented in a simple manner. In a general LP, the minimum ratio test requires divisions and comparisons. However, again because of the total unimodularity of the tableau, the minimum ratio to be found actually is in the same row as the minimum value in the vector *b* for which the corresponding element in the pivot column is a positive one. This is because for the selected entering basic variable, an exiting basic variable for which the entering basic variable will have a zero or negative coefficient cannot constrain the entering basic variable to a maximum limit to increase the objective function value. Simple, small logic suffices to conduct the minimum ratio test for this particular type of LP.

Gaussian elimination within the tableau can be parallelized, because more than one two-bit element is stored at each address. Each addressable word is 32 bits, so 16 columns worth of Gaussian elimination is conducted in parallel. The operation for the Gaussian elimination is simplified as well, because every element is zero or positive or negative one, and also the pivot element is guaranteed to be positive one.

3.3 Architectural Description

The architecture of the design can be split into control logic, computational logic, and memory, in a somewhat similar fashion to a microprocessor design. Roughly speaking, memory feeds into computational logic blocks, which calculate the necessary data, and the control logic directs the correct data into the correct memories. The computational logic can be further divided into stages of the simplex algorithm, which

run one at a time and in a loop until the simplex algorithm reaches an optimum solution, all of which is controlled by the control logic as well.

A conceptual pictorial description of the architecture is shown in Figure 3.4. These pieces of the architecture will be discussed in detail. In any digital design, consideration of the available hardware is critical. The reader is directed to Appendix B for details on the design environment, including both the software and hardware infrastructure utilized to design the network solver.

The control logic is centered on a counter, which indicates the stage of the algorithm currently is executing. This information feeds into each of the computational logic stages as well as the multiplexers that control the inputs into all memory blocks. The memory block contents are all reset before the computational logic blocks begin executing. Each stage in this algorithm corresponds to a sequential logic stage, and their functionality and implementation considerations will be addressed in the following subsections. The flow chart showing the algorithm divided into the stages implemented in this architecture is depicted in Figure 3.5.

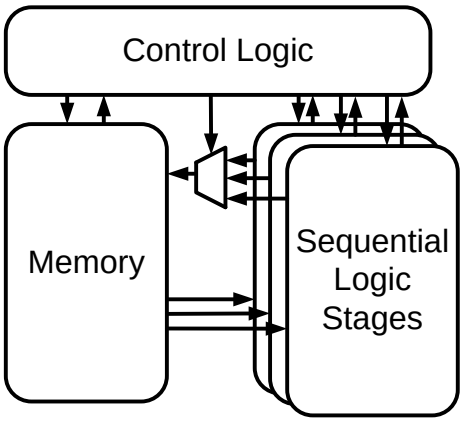


Figure 3.4: Conceptual block diagram of overall architecture

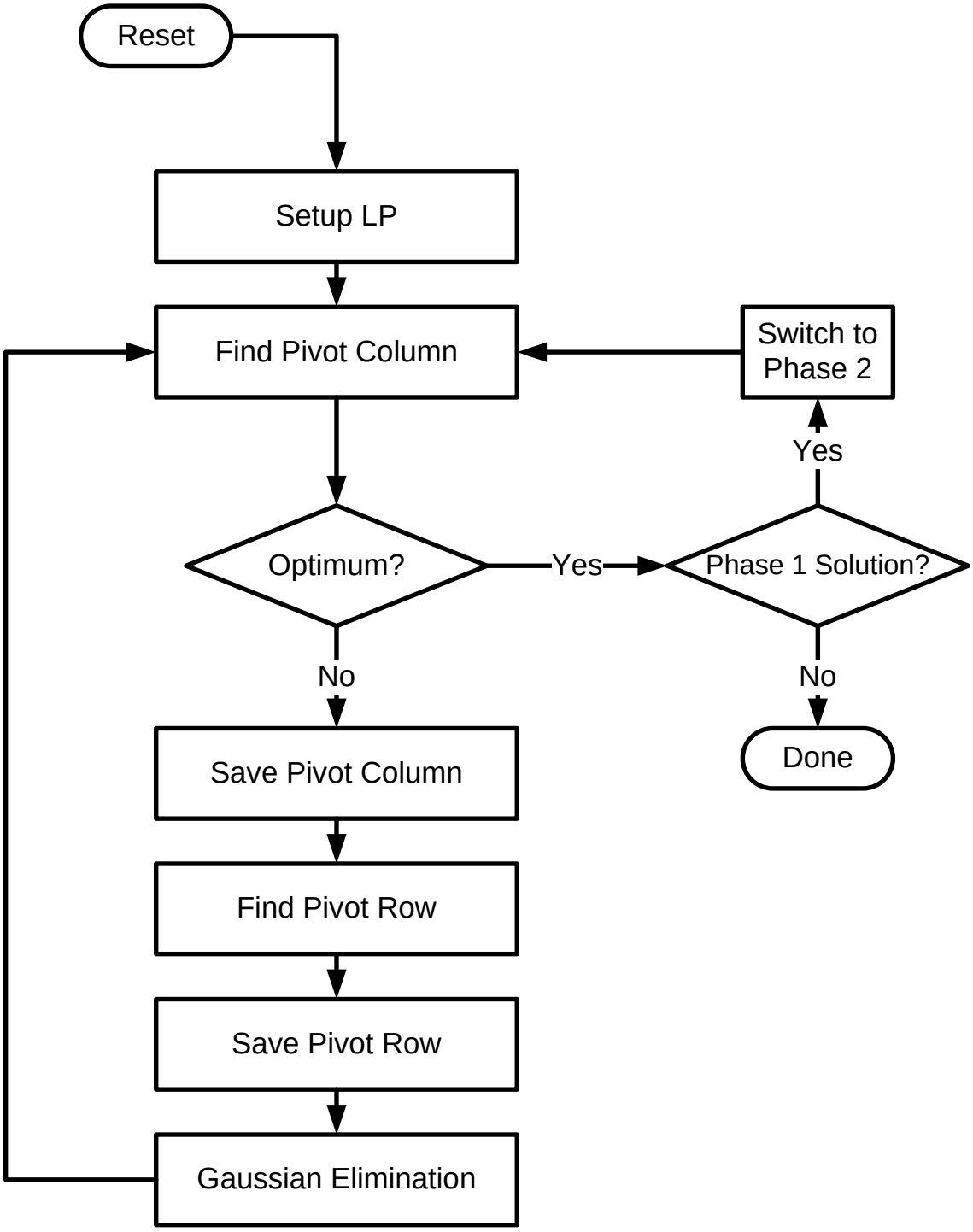


Figure 3.5: Flow chart of simplex logic architecture

3.3.1 Setup LP

The "Setup LP" stage reads out threshold values of the input vector from a memory and generates the *A* matrix, the two cost vectors, the *b* vector, and the list of basic and non-basic variables. All of these are generated and stored in memory in parallel, saving time. The parallelism incurred here requires little overhead, because the stored data are not dependent on each other. The input threshold vector is stored in a dual-port RAM, from which two thresholds are read out at a time, and the corresponding cost associated with a potential arc between these two nodes is calculated. Each potential edge is checked for validity, and subsequently stored as a column in the incidence matrix which becomes the starting point for the simplex tableau. With each column stored, a corresponding cost is calculated via equation (2.1) and then stored as the cost vector, which is the topmost row in a simplex tableau. In the implementation of

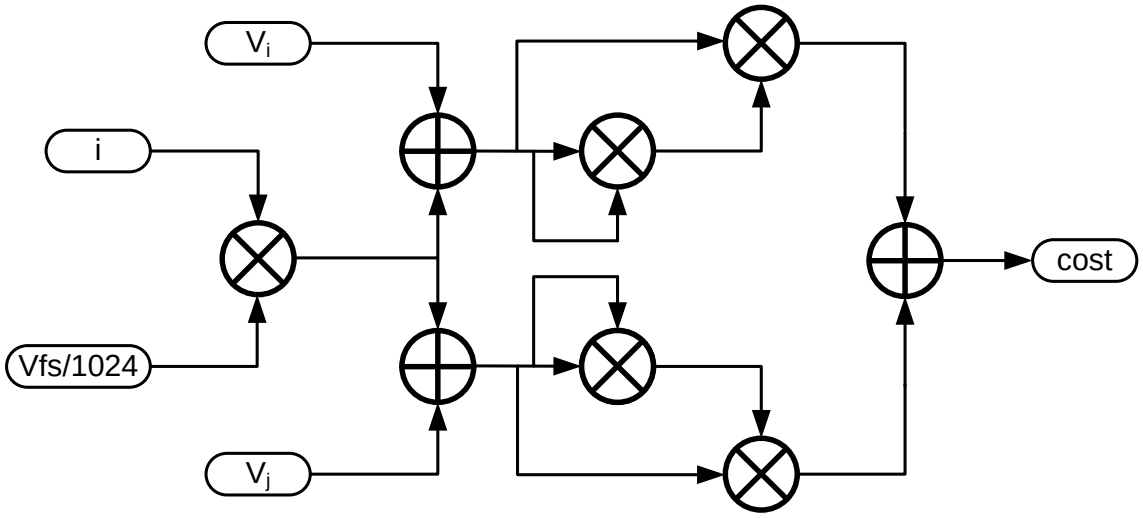


Figure 3.6: Cost calculator implemented with adders and multipliers (delay elements not shown)

this cost calculation, shown in Figure 3.6, the integral is evaluated assuming a uniform pdf for the input signal, although implementing a cost calculator for any pdf is possible.

Of the elements created in this stage, the *A* matrix is stored in the external SRAM, while the others are stored in the block RAM within the FPGA. The particular SRAM on the board incurs a ten-cycle latency, which must be taken into consideration in design. As mentioned previously, 16 elements of the incidence matrix are stored in each 32-bit word of this SRAM, allowing the large LP to be stored at all and easing the execution of parallelism later during Gaussian elimination. Since this "Setup LP" step only occurs once, it is only important to make sure that it does not take up a significant

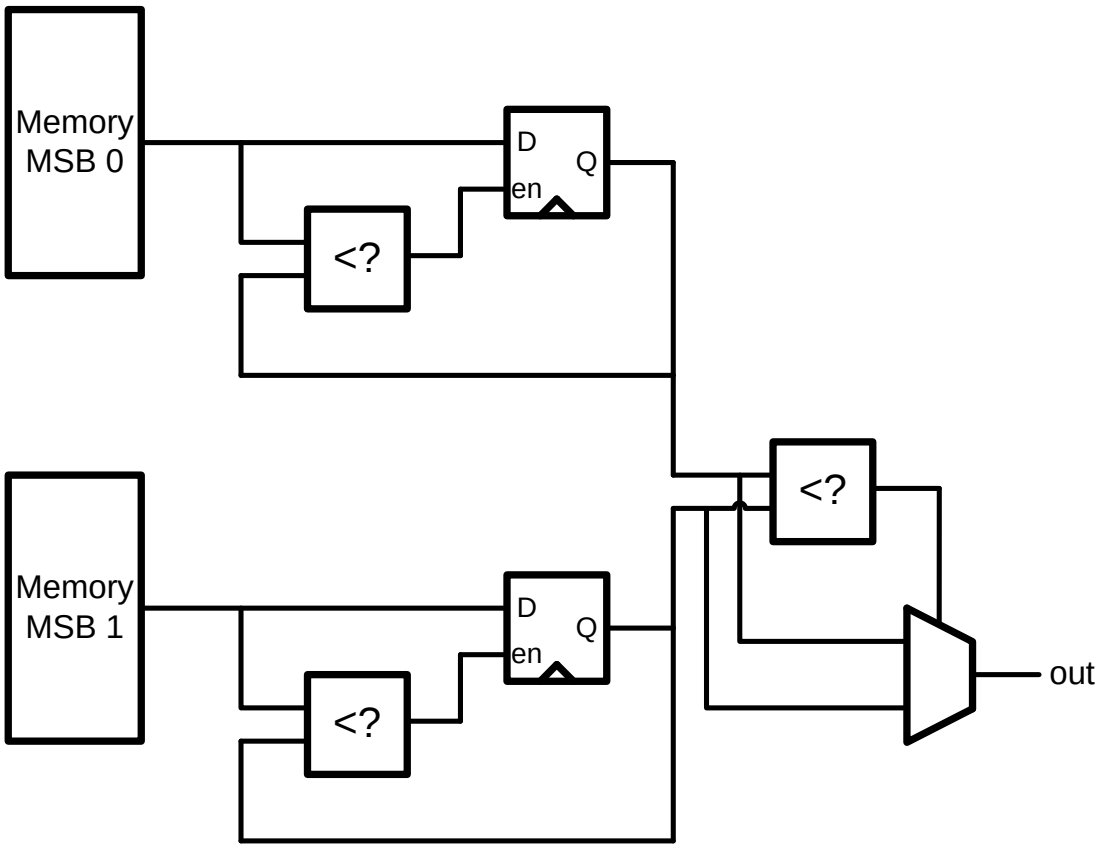


Figure 3.7: Simplified block diagram of architecture to find a minimum

number of clock cycles.

3.3.2 Find Pivot Column

The "Find Pivot Column" stage searches for the minimum value within the cost vector, and returns this minimum value as well as its location, which becomes the location of the pivot column. In general, the logic to finding a minimum can be optimized for speed by generating comparators in a tree structure. However, since the cost vector is 2^{14} long and must be stored in block RAM, a complete binary tree structure is infeasible. Since a completely serial search takes many cycles to complete, the cost vector memory is divided into subsections or sub-memories, for which the minimum within each is found then compared against each other to finally obtain the global minimum in fewer clock cycles than a serial search. This architecture is illustrated in Figure 3.7. The two memory subblocks are looped through simultaneously, which reduces the search time by half.

3.3.3 Check for Optimality

The "Check for Optimality" stage detects whether the current solution is the optimal solution to the LP and indicates to the control logic what to do next. If the solution is non-optimal, the simplex method is carried on. If the solution is optimal based on the artificial variables cost vector, then the cost vector is switched to the original objective function and the simplex method starts over to solve the Phase 2 LP, because a basic feasible solution has been found. If the solution is optimal based on the

objective function, then the simplex algorithm is completed, and the optimum solution can be calculated by reading out the *b* vector and the list of basic variables.

3.3.4 Save Pivot Column

The "Save Pivot Column" stage extracts the pivot column, saved in the external SRAM, and stores it into block RAM in the FPGA for ease of access to the pivot column. Having this step eases the execution of Gaussian elimination later by not needing to read out the pivot column values from the external SRAM for each row, because when compared to the SRAM, the BRAM has reduced latency and reduced control signal complexity.

3.3.5 Find Pivot Row / Save Pivot Row

The "Find Pivot Row" stage executes the minimum ratio test to determine the exiting basic variable. As mentioned before, the minimum ratio test has been optimized to take advantage of the nature of this particular LP, so a fairly simple search through the *b* vector suffices to find the minimum ratio. While it seems reasonable to execute a similar optimization here as that in the "Find Pivot Column" stage to find a minimum value, it is actually an unnecessary complication in the design, since the *b* vector is only a sixteenth of the size of the cost vector.

The "Save Pivot Row" stage extracts the pivot row from the matrix *A* and stores it into block RAM, a similar procedure as executed in the "Save Pivot Column" stage. This local copy of the pivot row also helps speed up the process of Gaussian elimination,

again because two accesses to different addresses in the SRAM in a single clock cycle are not possible.

3.3.6 Gaussian Elimination

Finally, the "Gaussian Elimination" stage turns the pivot element in the simplex tableau into the single 1 within its column by Gaussian elimination. Since each element in the tableau is two bits wide and each word in a memory is 32 bits, it is natural to perform 16 subtractions in parallel to execute the row reduction. Also, rows for which the corresponding element in the pivot column is zero are skipped, speeding up the process of updating the large tableau.

3.4 Benchmarking and Comparison

The design has been implemented on a board with a Xilinx V5SX95T FPGA, and runs at 160MHz. The design resource usage is summarized in Table 3.1 below. The total memory usage including the external SRAM is approximately 33Mbits.

Device	Number Utilized / Number Available	Percent Utilized
DSP48E	62/640	9%
RAMB36_EXP	62/244	25%
Slice Registers	10107/58880	17%
Slice LUTs	10884/58880	18%

Table 3.1 Summary of Resource Utilization of design on FPGA

In the end it is important to see whether the hardware solution truly provided what was aimed for. Besides verifying the functionality of the FPGA design, an attempt to make fair comparisons between performances of existing hardware and software solutions has been made. However, because this particular hardware solution is targeted for such a specific application, only vaguely related existing solutions have been found to be worthy of comparison. Thus the differences in performance and differences in the respective application spaces are also considered.

Rough performance comparisons against software solutions can be made by considering software that implements the network solver piece by piece. MATLAB is used as the software language with which many comparisons are made, because it is the language used in the hardware design and test environment. However, because of the relative inefficiency of MATLAB in terms of computational time, a freely available software package called lp_solve, software optimized for solving LPs, is considered for comparison. lp_solve is run on a 1.3GHz Intel Pentium SU2700 processor with 4GB of RAM. The entire network solver is divided into two sections as means for comparison: the network generation, and the simplex algorithm for the LP solution.

Table 3.2 summarizes the comparison of this work to existing software. The test vector input size is 1024 nodes, with an upper threshold placed on the edge cost to limit the total number of edges to fit on the FPGA design. The data transfer time to and from the respective solvers are not considered.

	This Work	MATLAB	lp_solve
Network Generation	0.419s	1839.39s (4390x)	Not Implemented
Simplex Algorithm	1.357s	Out of Memory	14.17s (10.5x)

Table 3.2: Computational Time Comparison and Relative Improvement of FPGA

To make the comparison between this work and MATLAB complete, a speed comparison between this work and the MATLAB `bintprog` function has been made for problem sizes that could be handled by `bintprog`. Figure 3.8 shows, on a logarithmic scale, the time it takes for each of these methods to solve BIPs of the same size. The black lines show the same points as plotted on Figure 3.1. Furthermore, Figure 3.9 shows the performance of the FPGA solution on a linear scale. As can be seen from this plot, the time the FPGA solution takes to solve a BIP is linear with the number of variables in the problem, whereas it was seen from Figure 3.1 that the software solution takes quadratic time. This divergence in performance can also be seen in Figure 3.8. Thus, not only is the FPGA a faster solution, but also the difference between performances will only increase as the flash ADC size is increased.

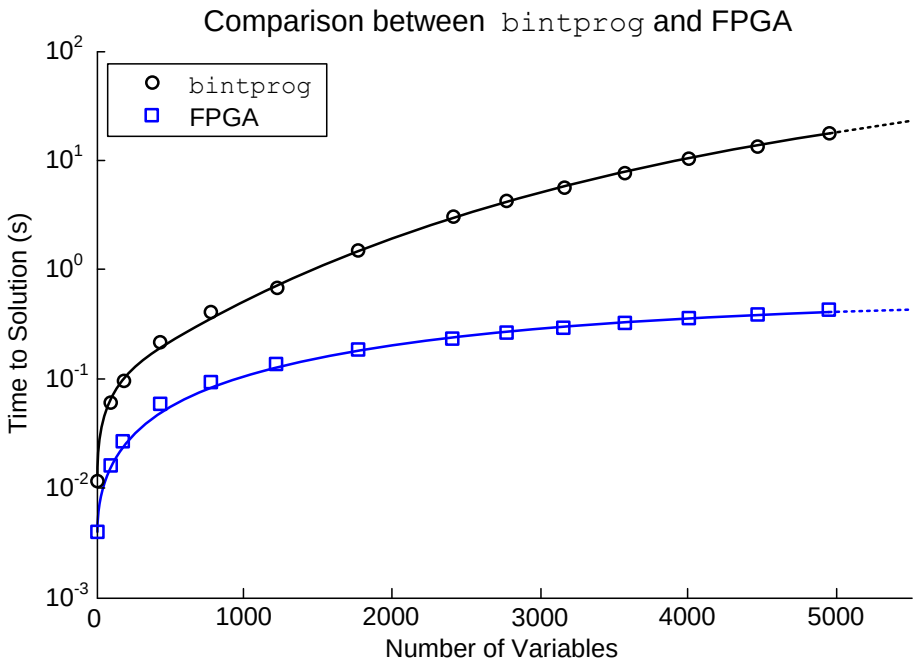


Figure 3.8 Comparison of Performance between MATLAB and FPGA

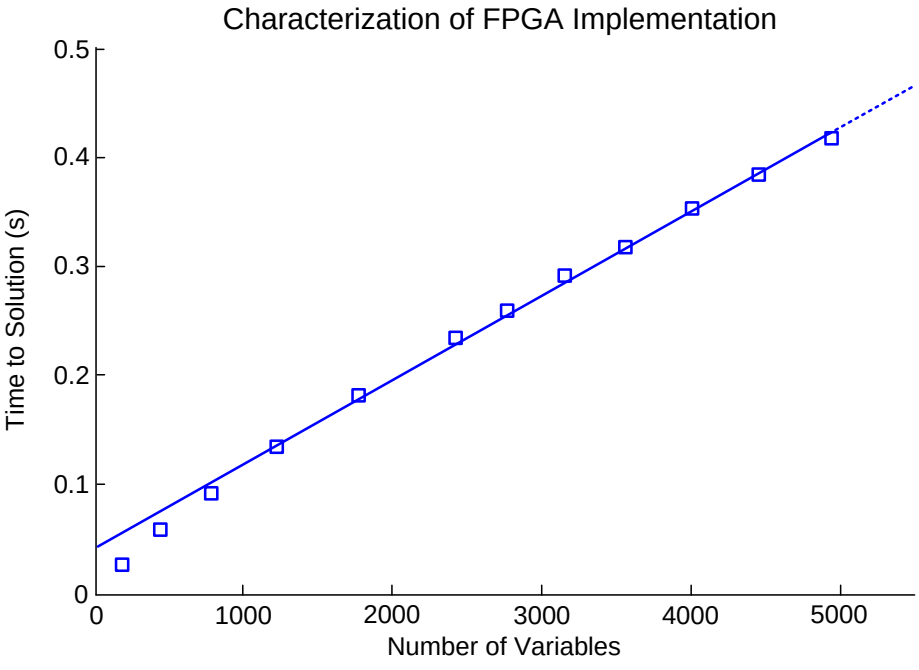


Figure 3.9 Characterization of FPGA Performance for Small BIPs

As can be seen from Table 3.2 and Figures 3.8 and 3.9, the hardware implementation has tremendous benefits, in terms of performance, even when looking at the solver parts individually. While a tenfold improvement in speed is relatively low in terms of comparison between hardware and software in general (owed largely to the iterative nature of the simplex algorithm), it is still an important speedup in the larger perspective, where the cost per chip is heavily dependent on the test time per chip.

The only hardware comparisons that can be made are against other implementations that have demonstrated somewhat similar functionality. Bayliss, et. al. [10] have implemented the simplex algorithm on the FPGA, but their design is not applicable to the flash ADC problem for several reasons. The foremost reason is that the maximum size of the LP that their design can solve is limited to 751 constraints and 751 variables, whereas the flash ADC problem contains 1024 constraints and subsequently many more variables. Also, their design requires multiple LPs to be processed through their pipeline in order to achieve their claimed maximum performance, and this cannot benefit us in the flash ADC calibration application. Furthermore, because they only claim the time it takes for their design to complete a single iteration of the simplex method, rather than the total time it takes to solve an entire LP, the comparison has been made on their basis. This is because the number of iterations of the simplex algorithm that is required to solve an LP is a function of both the problem size and the solver implementation. Table 3.3 highlights the main comparisons and differences.

	This Work	[10]
Types of LPs solved	Binary Integer	Standard-Form Linear
Problem size (constraints x variables)	1024 x 16384	751 x 751
Time per Iteration	0.31 ms	9.84 ms(*)
Hardware Used	Virtex 5SX95T + 32Mb SRAM	Virtex 4VFX140
Pipeline Multiple LPs	No	Yes

Table 3.3: Comparison of Hardware Implementations. (*) Time per iteration has been extrapolated from the quoted data to meet our problem size

CHAPTER IV

Conclusions and Future Work

To conclude, a summary of the main contributions of this work are listed, and possible direction for future research are discussed.

4.1 Summary of Research Contributions

- The network solver has been implemented in hardware which allows for calibration of the redundant flash ADC architecture.
- Both the LP formulation and the simplex algorithm have been analyzed in depth to perform speed optimizations at both the algorithmic and architectural level. Thus, vast improvements in the time-to-solution of the network solver have been achieved in the hardware implementation.
- This implementation not only shows a significant improvement in speed over software, but also shows increasing benefits with increasing problem sizes. Thus this solution will scale well with technology and will be applicable for calibrating larger redundant flash ADCs.

4.2 Future Work

Currently a chip that implements the described configurable flash ADC is being worked on, in order to demonstrate the system as a whole. A possible direction for future work also includes the evaluation of hardware implementations of other algorithms, such as Dijkstra's algorithm, that software LP solvers implement to optimize their performance. This could lead to a possible speed up in the calibration process. Also, once the flash ADC has been verified as a standalone data converter, integration into an entire system to demonstrate system performance is a promising research direction.

APPENDIX A

The Simplex Algorithm

A brief overview of the simplex algorithm is provided here. Some of the terminology used in the text of this thesis is also explained here. While this appendix focuses on the concepts used in this thesis, more complete and in-depth literature on the simplex method is widely available.

A.1 The LP Setup and the Simplex Tableau

The simplex algorithm is an efficient and simple way to solve a linear programming optimization problem. The steps are often visualized in a tableau form, which allows for hand calculations of smaller linear programs (LPs) and also gives an intuitive feel for what occurs in computational engines that implement the simplex algorithm. For an LP to maximize the objective function:

$$Z = c^T x \tag{A.1}$$

given the constraints:

$$Ax \leq b \tag{A.2}$$

the tableau is formulated as:

$$\begin{bmatrix} -c^T & Z \\ A & b \end{bmatrix} \tag{A.3}$$

Each column corresponds to a variable in the LP, and each row corresponds to a basic

variable, which is simply the set of variables that are nonzero. If this particular set of values for the variables is a feasible corner point, which it must be for all iterations of the simplex method, then that solution is called the basic feasible solution. In the tableau, all of the columns that correspond to the basic variables are all zeros except for a single 1 at the row which corresponds to that basic variable. The actual value of the basic variable can be read off from the rightmost column in the tableau. This remains true throughout the simplex iterations. Because the simplex method works by searching through basic feasible solutions to optimize the cost function, the tableau can be updated to reflect the new set of basic variables that correspond to a more optimal feasible solution. In each iteration, the non-basic variable that most positively affects the cost function obtains a nonzero value and thus becomes a basic variable, while the basic variable that most limits the value of the entering non-basic variable becomes zero and thus a non-basic variable. There are said to be the entering and exiting variables, respectively, since they enter or exit the basis.

A.2 Steps Through the Simplex Method

The main steps in the simplex algorithm are to check for optimality, to select the entering and exiting variables, and to update the tableau to reflect the new basis. Because of the convex nature of an LP, an iteration of these steps is guaranteed to arrive at the optimum solution after enough time.

The entering basic variable corresponds to the column with the most negative value of c in the top row of the tableau. This is because if the coefficient for that non-

basic variable in the objective function row is negative, then allowing that variable to have a positive value will have a positive effect on the objective function, allowing the new basis to be more optimal. If there are no negative coefficients in that row, then the optimum solution has been reached, and the simplex algorithm terminates. The column which corresponds to the entering basic variable is called the pivot column.

To find the exiting basic variable, the minimum ratio test (MRT) is performed. For the MRT, first the elements in the rightmost column are divided by the elements in the pivot column. Of these ratios, the minimum positive value is found and the associated row is the row corresponding to the exiting basic variable. This is because the variable with the minimum ratio most limits the growth of the value of the entering basic variable. This row in the tableau found by the MRT is called the pivot row, and the element in the tableau in both the pivot column and pivot row is called the pivot element.

The tableau must now be updated to reflect the new basis. This is done through a series of Gaussian operations on the tableau. The pivot element is made equal to 1 by dividing that row by the pivot element. In this way the rightmost element in that row is updated to reflect the value of the new basic variable that has entered the basis. Then the pivot element is made the single 1 in that column by performing Gaussian elimination. These steps are repeated until the optimum basis has been found.

A.3 Finding an Initial Basic Feasible Solution

So far it has been assumed that the simplex algorithm already starts at a basic

feasible solution. This in general is trivial for LPs with inequality constraints ($Ax < b$), because the origin can be a starting point. However, for some LPs with equality constraints ($Ax = b$), as is the LP that is discussed in this thesis, the origin is not in the feasible region and thus cannot serve as a starting point for the simplex algorithm. In this case, a set of variables called artificial variables are added to the set of constraints, all of which initially have nonzero values to begin with to satisfy the equality constraints when the other variables are non-basic. Then, the values of the artificial variables are forced to zero by optimizing another LP, which is to minimize the cost function defined to be the sum of the values of the artificial variables. If a feasible solution exists, then this optimization will drive all of the artificial variables out of the basis, making the objective function for this LP equal to zero. Thus, the simplex algorithm is performed twice, once to search for an initial basic feasible solution, and once again to search for the optimum solution after an initial feasible solution has been found. These two steps of the LP are referred to as the Phase 1 and Phase 2 LPs, respectively.

APPENDIX B

MATLAB / Simulink FPGA Design Environment

The environment with which the network solver implementation presented in this thesis is implemented is described here. The accessibility of this infrastructure has been a key driver for the success of the FPGA implementation.

B.1 Available Hardware

The board used for the design in this thesis is the ROACH (Reconfigurable Open Architecture Computing Hardware), a standalone FPGA processing board developed by the CASPER research group at UC Berkeley [14]. Besides the Xilinx V5SX95T FPGA and two 2M × 18-bit QDRII+ SRAMs, a separate PowerPC runs Linux and provides a Linux interface for controlling the board, including programming the FPGA and communicating data between the FPGA and a computer. The KATCP protocol allows MATLAB to use the Ethernet interface to communicate via the PowerPC to the FPGA [15]. Thus, as long as there is an internet connection, not only is the FPGA programmed with a bitstream through MATLAB, digital data can be written into and read from the FPGA through MATLAB as well, allowing excellent flexibility in testing the design on the FPGA. For example, MATLAB scripts can be written to test very large test cases or multiple test cases with ease.

Figure B.1: The ROACH Board

B.2 MATLAB/Simulink Design Environment

Hardware design for FPGAs is typically done through hardware description languages (HDL) such as Verilog or VHDL, but this can be time consuming and bug-prone. Xilinx provides an alternative tool based on Simulink that will generate the HDL based on a block level description of the functionality of the digital design. The blocks are specified in a Simulink style, making the design process intuitive. Signals can be viewed using the Simulink scope block, making the debugging process fairly simple as well. Signal bitwidth specifications can be automated, which is especially helpful for implementations of arithmetic.

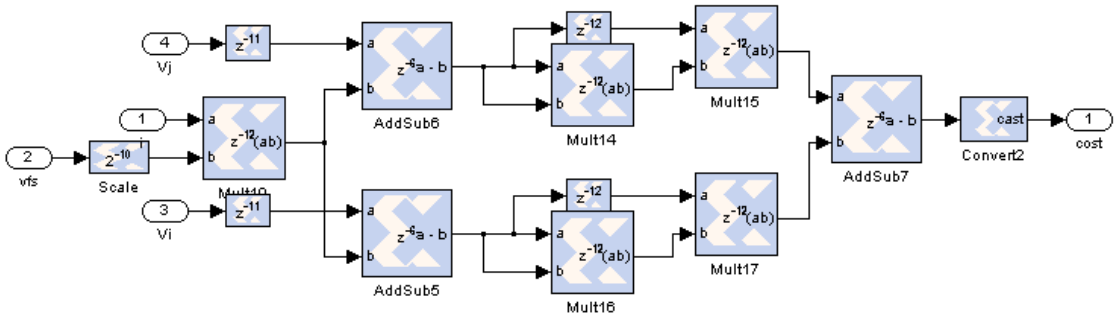


Figure B.2: Example Simulink Xilinx Blockset Design

This intimacy of the hardware design flow with MATLAB offers a high-level design procedure with ease of design and test, especially for large, computation-heavy designs. This design environment has been key in making the work in this thesis a success.

REFERENCES

[1] J. W. Chinneck, "Practical Optimization: A General Introduction," [Online]. Available: http://www.sce.carleton.ca/faculty/chinneck/po.html

[2] T. Sees, H. Harasake, G. Saidel, and C. Davies, "Characterization of tissue morphology, angiogenesis, and temperature in adaptive response of muscle tissue to chronic heating," *Lab Investigation*, vol. 78, pp. 1553-1562, 1998.

[3] V. Wang, K. Agarwal, S. R. Nassif, K. J. Nowka, and D. Markovic, "A Simplified Design Model for Random Process Variability," *IEEE Transactions on Semiconductor Manufacturing*, pp. 8-16, Feb. 2009.

[4] M. P. Flynn, C. Donovan, and L. Sattler, "Digital Calibration Incorporating Redundancy of Flash ADCs," *Trans. on Circuits and Systems II*, 50(5):205-213, May 2003.

[5] V. L. W. Limketkai, "Design of Flash ADCs with Large Offsets using Redundant Comparators." Ph.D. dissertation, University of California, Los Angeles, 2011.

[6] R. R. Harrison, et al., "A Low-Power Integrated Circuit for a Wireless 100-Electrode Neural Recording System," *Solid-State Circuits, IEEE Journal of* , vol.42, no.1, pp.123-133, Jan. 2007.

[7] D. Daly and A. Chandrakasan, "A 6b 0.2-0.9V Highly Digital Flash ADC with Comparator Redundancy," *International Solid-State Circuits Conference*, pp. 554-555, Feb. 2008.

[8] Skyler Weaver, et al., "A 6b Stochastic Flash Analog-to-Digital Converter Without Calibration or Reference Ladder," *IEEE Asian Solid-State Circuits Conference*, pp. 373-376, Nov. 2008.

[9] Ji-Jon Sit and Rahul Sarpeshkar, "A Micropower Logarithmic A/D with Offset and Temperature Compensation," *Solid-State Circuits, IEEE Journal of*, pp. 308-319, Feb. 2004.

[10] S. Bayliss, C. S. Bouganis, G. A. Constantinides, W. Luk, "An FPGA Implementation of the Simplex Algorithm," *Field Programmable Technology, IEEE International Conference on*, pp.49-56, Dec. 2006.

[11] S. Gibson, V. Wang, D. Markovic, “Effects of Quantization on Neural Spike Sorting,” to be published *IEEE International Symposium on Circuits and Systems*, May 2011.

[12] A. Schrijver, “Combinatorial Optimization: Polyhedra and Efficiency,” Springer, 2003.

[13] M. S. Bazaraa, J. J. Jarvis, “Linear Programming and Network Flows,” Wiley, 1977.

[14] “ROACH,” [Online]. Available: http://casper.berkeley.edu/wiki/ROACH

[15] “KATCP,” [Online]. Available: http://casper.berkeley.edu/wiki/KATCP