University of Central Florida

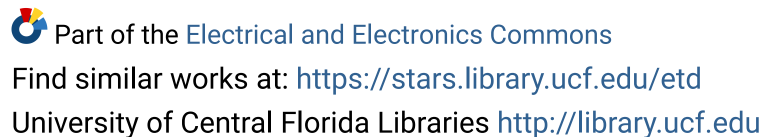
STARS

Electronic Theses and Dissertations, 2004-2019

2012

Simulation Study Of A Gpram System: Error Control Coding And Connectionism

Steven E. Schultz
University of Central Florida



Part of the Electrical and Electronics Commons
Find similar works at: https://stars.library.ucf.edu/etd
University of Central Florida Libraries http://library.ucf.edu

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Schultz, Steven E., "Simulation Study Of A Gpram System: Error Control Coding And Connectionism" (2012). *Electronic Theses and Dissertations, 2004-2019*. 2242.
https://stars.library.ucf.edu/etd/2242

University of Central Florida
STARS
Showcase of Text, Archives, Research & Scholarship

SIMULATION STUDY OF A GPRAM SYSTEM: ERROR CONTROL CODING AND CONNECTIONISM

by

STEVEN E. SCHULTZ
B.S. University of Central Florida, 2009

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2012

Major Professor:
Lei Wei

© 2012 by STEVEN E. SCHULTZ

ABSTRACT

A new computing platform, the General Purpose Reprsentation and Association Machine is studied and simulated. GPRAM machines use vague measurements to do a quick and rough assessment on a task; then use approximated message-passing algorithms to improve assessment; and finally selects ways closer to a solution, eventually solving it. We illustrate concepts and structures using simple examples.

To my parents, who taught me the most important lessons in my life: to work hard, do my best, and treat people kindly. To my girlfriend, Nathalia, who has given me immeasurable support and love. To my advisor, Dr. Wei, for having vision, passion, and courage to pursue knowledge. To everyone I do not know personally that has inspired me.

ACKNOWLEDGMENTS

This work would not be possible without the vision of my advisor, Dr. Lei Wei nor would it be possible without his patience and guidance. Again, thank you.

TABLE OF CONTENTS

LIST OF FIGURES ix

LIST OF TABLES x

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 LITERATURE REVIEW 4

CHAPTER 3 GPRAM THEORY 10

3.1 Introduction 10

3.2 Principles 12

3.3 Simple Illustration 13

3.4 Error-Control Coding 15

3.5 Basic Functions 17

3.6 Connections vs. Tasks: Prioritzing 19

CHAPTER 4 PROBLEM DEFINITION AND SIMPLE PROTOTYPE STUDY . . 22

4.1 Introduction 22

4.2 Unit Cells . . . 24

4.3 Rules . . . 25

4.3.1 Combination Rules . . . 27

4.3.2 Rule Dependency . . . 27

4.4 Complexity . . . 28

4.5 Connection Matrices . . . 30

4.6 Desired Outcomes . . . 31

4.6.1 Butterworth Filter . . . 31

4.6.2 Fast Fourier Transform . . . 31

4.6.3 Simple Feedback Control . . . 33

4.7 Results of the Simple Prototype Study . . . 34

CHAPTER 5 GPRAM SIMULATIONS AND LEARNING FROM PERFECT CODEWORDS . . . 36

5.1 (7,4) Hamming Code with Perfect Codewords . . . 36

5.2 (7,4) Hamming Code with Imperfect Codeword . . . 37

5.3 (10,5) LPDC Code with Perfect Codeword . . . 40

5.4 Effects of Error in Codewords . . . 41

5.5 Simulation Discussion . . . 41

CHAPTER 6 CONCLUSION 43

LIST OF REFERENCES 45

LIST OF FIGURES

Figure 2.1: An example of a neuron with function inputs f, x_i, weights w_i and output. 6

Figure 3.1: Block diagram of a simple GPRAM system. 15

Figure 3.2: Generated codewords from (7,4) Hamming Code 16

Figure 4.1: An example of a unit cell. 25

Figure 4.2: A four-unit network and its corresponding connection matrix. 30

Figure 4.3: Idealized outcome for Butterworth filter structure from simple prototype study. 32

Figure 4.4: Standard butterfly diagram of FFT with n = 4. 32

Figure 4.5: Simple feedback control system. The threshold detector outputs a signal to the next unit to increment or decrement. 34

Figure 4.6: Simple prototype software running and converging to the simple feedback control system solution. 35

Figure 5.1: A typical run from the main simulation program window. 42

LIST OF TABLES

Table 3.1: Tasks and corresponding sensor and action values 17

Table 4.1: Comparison of Vague and Rigorous Procedures 23

Table 4.2: Fundamental Rule Types 26

Table 4.3: Hybrid Rule Types 27

Table 4.4 Sample Network Complexities 29

Table 5.1: Connection Representation in a two-dimensional matrix. 37

Table 5.2: Results from Simulation of GPRAM with (7,4) Hamming Code 40

CHAPTER 1

INTRODUCTION

Current machine intelligence is intelligent only on the surface. It is still very much a mindless machine that executes written, fixed code and is bound by the arbitrarily-set limits of its human author and creator. What makes something intelligent is its ability to adapt to new environments and new problems. One may be able to classify the process of evolution as intelligent, where features of a system not useful to the survival of the system are deleted from future iterations (generations). Human beings are intelligent for several reasons. An important feature of human intellect is the use of tools to do work that would otherwise be impossible. Because it is not completely obvious how a problem will be solved, humans can devise a multitude of solutions. It is this fuzziness or uncertainty in the completion of a task that allows human intelligence to really shine: finding multiple solutions to the same problem.

Using the human brain as a guide in how we design the General Purpose Representation and Association Machine (GPRAM) system is helpful. The brain, at a high level, can be thought of as a general-purpose computing machine because it can handle a very broad range of tasks, fairly well, and some extremely well. Let it be clear that the study of this thesis is not a study to mimic the exact function and mechanisms of the human brain. Any talk

of the subject and comparisons to the neural system of human beings is merely for design inspiration and point-of-reference. This is a study of how to take lessons learned from the studies of the human brain and apply them to a new kind of computing. In this thesis, two types of general-purpose computing are studied and simulated.

In designing a general-purpose machine, there are two major approaches. Suppose a design has a set of rules and all operations and tasks are done using this fixed set of rules. There are many, many tasks that can be done since a task can have a multitude of instructions and of presumably any length. This is what is known as the precise approach. The rules are fixed in form and number; thus no rule can change, new rules cannot be added and existing rules cannot be removed. The precise approach is what is used in microprocessors, for instance. The instructions of a microprocessors instruction set are fixed and tied to the physical attributes of the processor. New instructions cannot be added to the instruction set, only higher-level instructions can be made with existing instructions. Existing instructions cannot be altered to do something different. This low-level instruction set is commonly known as assembly language. As the level of abstraction goes up, higher-level languages become possible. Computer programming languages like C and Java utilize assembly instructions to realize each of their respective instruction sets and functions.

The focus of this thesis, however is the second approach. Imagine a machine or system that can adjust its instructions to fit a particular need, or when it senses other instructions are no longer needed. This sort of system with utmost flexibility is desirable since the exact use of our system (hence, general-purpose machine) is not known. This system, thus, should

have the ability to learn. For example, instead of an addition instruction having only two operands, the system might change its format to accept three operands. This approach of building in flexibility will be called the versatile approach. Attributes of a machine designed with the versatile approach include: the ability to make quick, rough estimates and decisions and the ability to find solutions, however inexact, for a wide-range of problem types.

One of the two systems under study in the versatile approach is a simple prototype which uses unit cells as building blocks for higher-level functions. The cells act like mathematical functions (input and output with some operation done on the input) but can be connected to other cells in a network. The network can rearrange itself and change the functions of its unit cells, thus giving a very flexible problem-solving platform. The second system is the General Purpose Representation and Association Machine (GPRAM) from [Wei12]. The GPRAM uses low-density parity check codes and other error-control coding techniques in the hopes to uncover solutions to problems that would otherwise be unseen by trained scientists and engineers [Wei12] . The difficulties in building a system using the versatile approach will be illustrated in both systems.

It will be verified whether the methods presented achieve the attributes for this new computing platform. This work spreads multiple disciplines including artificial intelligence and error-control coding with the use of low density parity check codes. I hope to highlight the difficulties in realizing these designs, but also show the advantages that would come from having such systems.

CHAPTER 2

LITERATURE REVIEW

The basis of this thesis work stretches multiple disciplines. Building intelligence and flexibility into machines has been a work-in-progress since before the electronic digital computer. An intelligent machine has been defined as, any machine that can accomplish its specific task in the presence of uncertainty and variability in its environment.[Nil10] Nilsson provides an even broader definition, any machine that possesses the quality that enables an entity to function appropriately and with foresight in its environment. The dream of automation has been alive since the times of the ancient Greeks. Aristotles syllogisms can thought of as a type of logical reasoning argument that is the basis of predicting behavior. In the wake of Pascals Calculating machine, Charles Babbages Difference Engine, and numerous other mechanical calculators, the theory behind machine intelligence was still, for a long time, a very early work in progress [Rob90]. The technology to make such theory realizable was still years and years away.

It was not until the 1930s that Alan Turing showed that any form of computation could be described digitally [Nil10]. His a(utomatic)-machine or Turing machine was an abstract, mathematical device that can simulate the logic of any computer algorithm. It is a simple device, consisting of four parts: a tape, a head, table, and a state register. The tape holds

the input to the algorithm; the data to be manipulated. The head is a device that reads the data from the tape and changes the value at that position depending on what is read from the table. The table tells the head what to write based on the current symbol being read from the tape and the current state held in the state register. The head advances from symbol to symbol on the tape. Turings theoretical work formed the foundation of computer science. The idea was born that a machine can receive symbols from a finite alphabet, manipulate them in an unsupervised way and give a corresponding output [Teu02].

Connectionism aims to explain human intelligence and the brain by artificial neural networks. McCulloch and Pitts published the first formal paper on neural networks in 1943. By assuming that a neuron conformed to the restraints that it had a finite threshold, was binary, and was inhibitory or excitory and caused a delay of one cycle, they proved that any logical expression could be constructed by a network of these neurons [Teu02].

An artificial neural network is an information processing system which consists of simple and interconnected elements. They aim to emulate the structure of the human brain and its ability to learn from experience. The neuron being the basic element; it acts as a threshold or detector in parallel with other neurons to determine if a signal is present or not. Larger blocks of these ANNs can be built in a manner similar to how logic gates can be used to build advanced digital circuits. The real advantage is that they can be built to detect varying levels of signals not just binary ON or OFF [Teu02].

Modern artificial neural networks have their foundation in the beginning of the 1960s. The term “softcomputing from Lotfi Zadeh, referred to systematically integrating fuzzy logic

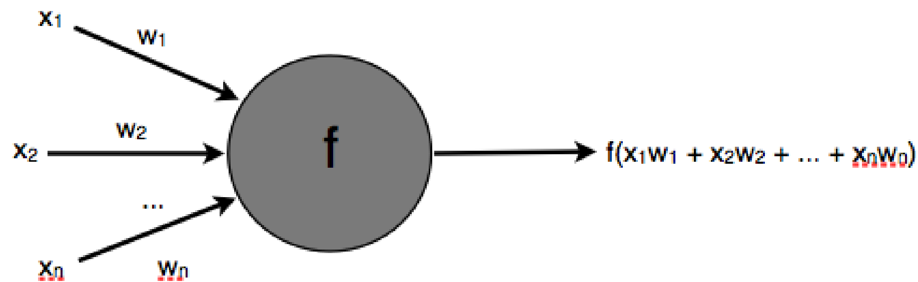


Figure 2.1: An example of a neuron with function inputs f, x_i, weights w_i and output.

techniques. Zadeh was the first to do such a thing and in 1992 coined the term [For01]. Soft-computing today is generally known as the usage of fuzzy logic, neural networks, evolutionary algorithms, and non-linear distributed systems to perform some computation or learning operation. Fuzzy logic is based on the premise that human thinking can be broken up into key elements that are not numbers but rather indicators of fuzzy sets. A fuzzy set is a class of objects in which the transition between inclusion and exclusion is gradual. Every element can be long to any set within some degree of certainty or significance. The more specific term neurofuzzy networks is used to signify a set of fuzzy rules in a neural network. One could train the neuro-fuzzy network with an algorithm to iteratively change parameters and weights.

Learning, in a very general sense, is the ability of a system to adapt to an environment. In varying educational settings, learning can refer to an individual's ability to understand new concepts, the ability to apply gained knowledge to new problems, or simply the ability to retain and recall facts. However, in neural networks, learning refers to the search of those parameters to the network that can optimize the predetermined function [Nil10].

These parameters include the weights of the inputs, the thresholds, and timing. Learning in the artificial intelligence realm is typically split into supervised learning and unsupervised learning. One can deduct that supervised learning uses some external agent to modify the learning system parameters. Nonsupervised neural networks are generally used for solving problems of classification. For unsupervised learning neural networks, an algorithm tends to group input data with common characteristics. Sometimes there can be a training phase for these types of algorithms and systems. Thus, the training data sets the weights and parameters and when new input is received, a somewhat close solution would follow [For01].

For the simple prototype study later on in this thesis, learning is kept as unsupervised as possible since the decisions to turn on or off rules is part of the actual system, not outside of the system (i.e, a user). The simple prototype study's type of learning is closest to reinforcement learning; it learns actions by doing those actions that garner the highest reward without being explicitly told what actions to take.

The Neuron Doctrine posited by the Spanish neuroanatomist Santiago Ramon y Cajal, proposed that living cells called neurons together with their interconnections were fundamental to what the brain does [Nil10]. The discovery that the brain was made of much smaller individual units along with the size, shape and functional specialization was revolutionary. What I borrow from this area of human anatomy is the basis for the Simple Prototype Study: individual cells that perform certain functions connected in a network to build much a more advanced structure.Human thinking is associative. We compare new input from our senses with memories similar (or opposite) to those experiences.

Perceptrons are a linear classification algorithm that can be considered a type of soft-computing. Proposed by Frank Rosenblatt, perceptrons take in an input, multiply it by some factor and then a collection of these weighted inputs is summed to give an output [For01]. In terms of artificial neural networks, perceptron algorithms are singlelayer or multilayer. Multilayer perceptrons form a more complicated network with usually one or more hidden layers that exist between the inputs and outputs. Later in this thesis, a hierarchal structure for the GPRAM will be studied and the similarities between such system with a multilayer perceptron network will become very clear.

Tanner graphs are bipartite graphs that are used to represent constraints and construct codes in error-control coding. A Tanner graph utilizes recursive techniques to make longer codes. Since it is bipartite there are two sets of nodes; one set the variable nodes, the other, the parity-check nodes. Tanner graphs are used to represent many codes, not just LDPC.

Low-density parity check codes were first devised by Gallager in the 1960s but did not find widespread use until the early 1990s. Low-density parity check codes are linear block codes that are constructed via a sparse bipartite graph (Tanner graph). The codes are formed by a sparse paritycheck matrix which is usually randomly generated. The sparse matrix is valid if it fits within the sparsity constraints. The paritycheck matrix is then used to form the generator matrix, from which all codewords are formed by multiplying with every information word. Messages can decoded iteratively with belief propagation. This is a method of using past-received bits to decode ambiguous or unknown parts of the message. Information is passed between variable and parity-check nodes in the form of log-likelihoods.

Nodes are updated every iteration. This information basically tells the decoding process what the node's likely value will be based upon current values and past values.

The GPRAM is first described by Wei [Wei12]. A previous paper on amorphous communications describes a communications system based on statistics and random pulse waveforms. The messages are embedded in the statistics of these random pulse waveforms. The implications of this being that the decoding can detect between minute differences in waveforms, much like a human eye can detect very small differences in similar objects. Error-control coding, as an application, attempts to lower the bit error rate during a communication transmission. The GPRAM utilizes LPDC because of their ties with belief propagation and their decoding schemes match nicely with how neurons operate.

CHAPTER 3

GPRAM THEORY

3.1 Introduction

The GPRAM (General Purpose Representation and Association Machine) is founded on the principle that any information-processing system can be split into two parts: one of which makes quick, rough estimates; the other which does the precise task [Wei12]. The GPRAM focuses on the first part and will further split this process into two stages: group design and individual specification. Group design refers to preserving common features in the group. Individual specification is the process of narrowing down the features of the system to focus on specific tasks. This use of group design and individual specification will be known from here on as the versatile approach. The hope of the GPRAM design is to find rules and structures which will give a group of good systems, but not necessarily perfect systems as conventional designs strive to achieve. Conventional designs want an optimized design for a very specific task. Several lessons from error-control coding that influence the design of the GPRAM. The averaging of performance of randomly-constructed long codes can approach the Shannon limit, so there must exist exist many good codes (not perfect, but good). There are many codes randomly constructed on Tanner graphs

that are near optimal given that the graphs have few, small loops. An important lesson directly applied to the GPRAM is that information can be obtained and passed between sub-graphs at low complexity. The decoding process can be implemented iteratively. The structure can become very robust (higher tolerance for noise and errors) as well. Pearl's belief algorithm and iterative decoding share some common links. Pearl's belief algorithm is one of the key tools to process information for Bayesian networks. Some Fourier transform and Kalman filtering problems and solutions to them can be unified under the lowdensity graph representation and iterative decoding. Operations of codes on graphs and iterative decoding can be divided into three stages: repetition, random permutation, and non-linear operation. This is very similar to how neurons work. To restate, the goal is to design a system that can be flexible and generalpurpose like the human brain. If the GPRAM does not know which tasks need to be solved, then how do we know which representation or association is good or bad? If we cannot determine which one needs to be eliminated, then we need to search over all orders, at least as many as possible. This is often impractical. Many scientists have been searching for the precise mathematical models and coding principles of the human brain. Very often, their models do not fit well with what we have observed. When witnessing some of nature's decision-making systems it can be noticed that a machine with vague computation and approximation is sufficient to make quick estimates at a certain confidence. Once we accept the concepts of vagueness and approximation, it opens many doors to new designs.

We begin with a short treatment of the foundation of the GPRAM system, then a walk-

through example will be completed to give the reader a clearer idea of how a GPRAM works.

3.2 Principles

The hope of the GPRAM is to use vagueness to our advantage in the discovery of new principles. Principles that are part of the human brain and nature are often clouded by details so minute that the bigpicture is lost. According to Fox and Raichle most of our knowledge about the functions of the brain is obtained from studying a minor portion of brain activity. What is hoped to be uncovered is how the GPRAM can help us understand how the brain works.

This leads to four major principles of GPRAM theory.

- Principle 1: Split informationprocessing into two parts. The outer part (global) will handle interactions between global and local portions. The inner part (local) will preserve a higher degree of resolution.

- Principle 2: Treat each object in the outer part or the global region as one of many samples of an object; one of many representations of the same object. In addition, freely associate representations with little constraint.

- Principle 3: Communication between individual portions of the GPRAM are essential for effective (and efficient) GPRAM design.

- Principle 4: The GPRAM must have variation capabilities to keep generality and stability.

These four principles have been distilled from the possibilities of what it is we want the GPRAM to do and how we want it to operate. These principles will guide the design.

3.3 Simple Illustration

In this example, we are illustrating the versatile approach of the GPRAM so that the user can have a greater appreciation for what is being done. Consider two symbols A and B; A has four possible values $[0, 1, 2, 3]$; B has three $[0, 1, 2]$. I define three tasks:

1. Indicate true if $A > 1$ and $B > 1$.
2. Indicate true if $A = 0$ and $B = 0$.
3. Indicate true if $A \geq 1$ and $B > 1$.

Four representation cases will be used: two to represent the precision approach, two to represent the versatile approach.

Case 1 maps $A = [0, 1, 2, 3]$ to $[x_1, x_2] = [00, 01, 10, 11]$ and B to $[y_1, y_2] = [00, 01, 10]$. Task 1 is performed easily as $x_1 \wedge y_1$. However, it becomes more difficult for Task 2 and Task 3. This precise approach is efficient for very specific tasks. If the number of values increases it becomes very difficult to use the versatile approach.

Case 2 maps $A = [0, 1, 2, 3]$ to $[x_1, x_2, x_3, x_4] = [1000, 0100, 0010, 0001]$ and B to $[y_1, y_2, y_3] = [100, 010, 001]$. Task 2 can be performed by one operation $(x_1 \wedge y_1)$ but for the two other tasks, two operators are required (OR and NOT).

Case 3 maps $A = [0, 1, 2, 3]$ to $[x_1, x_2, x_3, x_4] = [0010, 1011, 1001, 1101]$ and B to $[y_1, y_2, y_3] = [000, 011, 101]$. Looking at x_4, we can see it duplicates x_1 and can be deleted. This deletion is a feature of using the versatile approach. We can use this redundancy to our advantage. Thus, we can perform Task 3 with one operation $x_1 \wedge y_1$, but more than one operation will be needed for Task 1 and Task 2.

Case 4 maps $A = [0, 1, 2, 3]$ to $[x_1, x_2, x_3, x_4] = [0100, 1011, 1011, 1011]$ and B to $[y_1, y_2, y_3] = [000, 000, 101]$. This is a poor case because of the repetition of mappings. This is a proper example of the perils of the versatile approach. In general, when the number of possible values becomes very large, the probability of choosing a poor representation mapping becomes smaller.

When using the versatile approach one may discover some simple solutions to perform a specific task. Complexity saving for these particular cases is minor. This example highlights to fixtures of GPRAM theory and design: constantly search for simple approximation and discover new ways of representation.

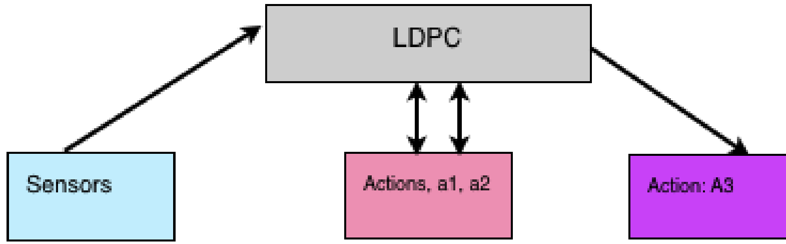


Figure 3.1: Block diagram of a simple GPRAM system.

3.4 Error-Control Coding

In error control coding, it is common to use a (7,4) Hamming code which is produced by a generator matrix. Information words, i, and codewords, c, are related to one another by this generator matrix

$$\mathbf{c} = \mathbf{i}\mathbf{G} \tag{3.1}$$

where **c** is the vector of codewords, **i** is the information bits vector (**i** = (i1, i2, i3, i4) and **G** is the generator matrix. The vector **i** contains all possible information words.

For the GPRAM simulation, the (7,4) Hamming code implementation consists of seven variable nodes, which correspond to each bit of a codeword and three parity check nodes. External to the GPRAM system is two sensors, and three actions. Sensors and the two actions connect to variable nodes and their values change, therefore the sensors are inputs

$$
\begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \\ W_7 \\ W_8 \\ W_9 \\ W_{10} \\ W_{11} \\ W_{12} \\ W_{13} \\ W_{14} \\ W_{15} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}
$$

Figure 3.2: Generated codewords from (7,4) Hamming Code

as well as two of the actions. The third action is connected to a variable node but is an output.

$$
G = [I|P] = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{3.2}
$$

$$
H = [P^t|I] = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}
$$

Codewords are generated by multiplying each element in **i** with every column in **G**. Each element in **i** is multiplied by its counterpart in the column of **G** and then summed using modulo-2 addition over all elements of i. Each codeword is produced by taking the 1 x 4 matrix of **i**, multiplying by the 4 x 7 matrix of **G**, and producing a 1 x 7 matrix of codeword bits. Each 1 x 7 matrix is a codeword. In this case there would be sixteen codewords. Once the codewords are generated, initial sensor values are given.

Table 3.1 shows the tasks and the values for s_1, s_2, a_1, a_2, and a_3.

Table 3.1: Tasks and corresponding sensor and action values

	Inputs	Output	Special Function
Tasks	s_1,s_2, a_1, a_2	a_3	*Switch*
1	1 1 0 1	0	ON
2	1 0 0 1	1	ON
3	1 1 1 0	0	OFF
4	0 0 0 1	0	ON
5	0 0 1 0	0	OFF

3.5 Basic Functions

The simulation program then searches for a codeword with variable node values that match the sensor values. The codeword search begins by randomizing the list of codewords. This is done to prevent the system from selecting a codeword from the beginning of the list more often than the bottom half or the middle. Once a list of random codewords is made, the

codewords are tried for validity. A codeword is valid if the connections made along with the current values of the inputs result in the parity check nodes all being zero. The first check for validity is if the variable nodes that are connected tos_1 and s_2 are correct for the codeword in question. For instance, if v_1 is connected to s_1 and v_6 connected to s_2, if the If they match, then we continue. If not, then that codeword is skipped and the next word on the randomized list becomes the current codeword to be checked for validity. Once the list of sixteen codewords is exhausted (if none of them match) then connections are erased and reconnected, and codewords are randomized again and the process repeats. Supposing the parity check nodes are all zero, meaning, a valid codeword is found, the simulation then records this connection scheme and codeword as valid for that task.

What makes this method difficult is that the connections are fixed from the beginning of the codeword search until every possible codeword is checked. This creates a lot of wasted time in searching for the correct codeword. An alternate method would be to change connections as the parity bits are checked. For a parity check node that is a “1 (an error), the simulator program will check which variable node(s) is at fault and change one, or some, or all connection(s) to an unconnected variable node. Then, the parity check nodes are checked again. If there is still not a valid connection, the other variable node is connected and tried. Following this, if the new connection is unsuccessful as well, connections are made again for all nodes and the process repeats. Suppose if v_2 is connected to s_1 and s_1 is “1 and therefore v1 is “1. Further supposing that s_1 must be a “1 for the particular task and that it is parity

check node 1 that is giving the error. Either v_1, v_2, v_3, or v_5 must be changed for the first parity check node to be 0.

3.6 Connections vs. Tasks: Prioritzing

In designing the system, there is a choice that is made between connections staying fixed or task(s) staying fixed. In Connection-priority simulation, the connections between the variable nodes and the sensors/actions remain fixed until a valid codeword is found. It is therefore unknown which task will arise when this prioritization is made. In Table 3.1, there are five tasks listed. If the task produced does not correspond to those in Tasks 1-5, it is labeled unknown or irrelevant. Imagine a carpenter having a selection of tools to cut wood. Some tools are better suited for the task depending on the type of cut and the type of wood. All of the tools will cut the wood, but only specific ones will actually be right for the job. These irrelevant tasks that are not described in Tasks 1-5 are like those carpentry tools not right for the job. They fit the main requirement of the parity check nodes being valid, but do not constitute an action. Only one action is performed or not performed as output in the first case (the simple case with the (7,4) Hamming code), so the result is simple. However, in a more complicated system with multiple actions to be performed, giving priority to connections can give some interesting results. When prioritizing connections, the task to be done is unknown. A major question is: with randomizing connections, will more tasks

be completed than other tasks? Are some connections more optimal for certain tasks than others? Is one task more likely to happen over others?

With a task-priority simulation, the task to be completed is known before the start of the simulations. Connections and codewords are tested until the task is done. Within task-priority simulation there are two options: 1. make new connections after each unsuccessful attempt. or 2. keep the connections the same until all codewords are exhausted for those connections. For option 2, connections are only re-established between variable nodes and sensors/actions when all possible codewords are found to be invalid for that task. The connections are thus fixed for the entirety of the codeword search process. Instead of priority, the term fixed could also be used as it may be a bit less misleading. In addition, the option for codeword-priority or codeword-fixed simulation would mean that one codeword is tried for multiple connection schemes until the parity-check nodes are valid. The codeword is chosen first and then connections are made from the sensors and actions to the variable nodes. If the parity-check nodes are invalid, the codeword is kept the same and connections are re-established. How this re-establishment occurs is discussed later on. When the parity-check nodes are found to be valid in a codeword-priority simulation, the task done is then recorded like it is in other types of simulation. So far, connection-prioritized, task-prioritized, and codeword-prioritized have been discussed. Hybrid prioritization is applying two of the three to the same simulation. Connection and task fixed simulation hold the connections and desired task constant while codewords are tried. Connection and codeword prioritized simulations hold the connections and codeword fixed while different tasks are

tried. Codeword and task prioritized simulations fix the codeword and task while connections are tried.

CHAPTER 4

PROBLEM DEFINITION AND SIMPLE PROTOTYPE STUDY

4.1 Introduction

A simple prototype can be built to demonstrate the versatile approach. In this prototype study, functions and routines are built from unit cells that have: input connections, perform a function that produce output, output connections and control inputs. By arranging these unit cells in certain configurations, higher-level functions can be built. These configurations will be known as networks. Rules govern how many units are part of the system, how the units are connected, and what functions the units are performing. These rules act as a type of DNA or map to the system structure and function. Rules are turned off and on like genes, consequently creating different systems. The higher-level functions to be realized with these unit cells are a 4-point FFT, a second-order Butterworth filter, and a simple control system/feedback regulator. These goals were chosen because we need some goal for such an open system. It is necessary to know how to shape the system into doing something desirable or useful. These three test goals were chosen in particular because they are typical electrical engineering functions, that can show off the potential usefulness of a system. Certain sets of rules will be better suited for one routine, while a certain subset will be better suited

for another. The challenge here is one that was previously stated: how is something built for versatility, but still able to focus on a single function? This portion serves to contrast with typical precise approach design. It will be shown whether or not this method gives any decent or conclusive results.

Table 4.1: Comparison of Vague and Rigorous Procedures

Vague Procedures	*Rigorous Procedures*
Steps may be in multiple sequences.	Exact steps.
Objects/words may have multiple definitions	Exact definitions.
May not be optimal in any regard.	May be much closer to optimal.
Adaptive.	May not be adaptive or be much less flexible.
Good for several types of conditions and environments.	Constructed for certain conditions or environments.

One of the main themes of this thesis work is the rejection of rigorous, precise procedures and the acceptance of vagueness. This simple prototype study illustrates the differences between the two very well. A routine or function made from vague procedures may have steps to completion in different sequences where as the function performed by the precise approach will have exact steps. The vague approach will not give an optimal solution where the precise function may give an optimal or very optimal solution. This is where the trade-offs occur between the two approaches are most visible. The precise approach can produce a very accurate solution but can only do so one way. The vague approach would be able to adapt and change to get a nearby solution.

4.2 Unit Cells

The basic building block of a network is the unit cell. Each cell has a basic function: addition, subtraction, multiplication, division, or time delay. Unit cells have inputs and outputs. Control lines are used to change the function of a unit cell or to turn it on or off. The unit cells available input and output connections is dependent on the function that is selected, for instance a unit cell with a squaring function can only take in one input.

The available fundamental functions for unit cells are: arithmetic functions (addition, subtraction, division, multiplication), threshold detection, time delay, increment and decrement. Addition function requires at least two inputs (of either positive or negative sign) as well as the subtraction and multiplication functions. The division function can only have two inputs and if the divisor is 0, the unit cell will output a null signal (different from a zero signal). If any unit cells receive a null signal as input they will behave as if there is no signal at that input (i.e, a unit cell performing addition will not be able to use that input and if it is only one of two inputs, will not add and will thus also output a null signal). One can see that a null signal can propagate throughout the entire network. In the software simulations done, null signals were not found too often because the randomized test data rarely produced zero.

Unit cells receive input from the environment or from other unit cells.

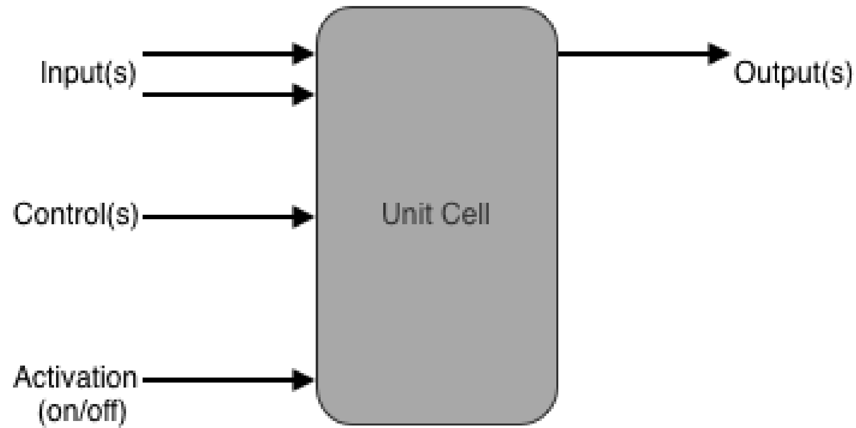


Figure 4.1: An example of a unit cell.

4.3 Rules

Direct determination of connections and functions is done by the rules. These rules allow units to connect, disconnect, power on and off, and change function. Three fundamental types of rules exist: function, connection, and timing. Function rules alter the function of a unit or units. Connection rules change how units are connected in relation to each other. Timing rules set time constraints on functions and connections. The term fundamental is used to describe a set of rules that change units in solely those ways just listed. Fundamental function rules have no connection or timing requirements, fundamental connection rules have no function or timing requirement, and so forth.

Hybrid rules are combination of two or more of the fundamental rule types. These can be implemented to give more flexibility and simplicity in the overall number of rules. Hybrid rules can be function-connection, function-timing, timing-connection, or based on all three

Table 4.2: Fundamental Rule Types

Fundamental RuleType	*Example*
Function	Unit A must perform addition on its inputs.
Connection	If unit A is not connected to unit B, then unit A can connect to unit C.
Timing	Unit A must connect to unit B after the initial 5 time units.

fundamental types: function-connection-timing. Thus, the total number of rule types is seven: three fundamental and four hybrid. Generating rules for use in the simulation is done by substituting in different values into rules. In this case the rules are stored as functions that return values depending if the rules are followed or violated. This design allows the simulation to be more simply coded and also allows for the generation of multiple rules of the same type. Used values for rules that did not lead to a desired result can be recorded and not tried again. Rules are made as functions in the C language. The arguments to these functions are the units, functions and timing constraints for the rule. A rule may be Unit A must perform addition on its inputs. The function prototype may look like: `int fund_function(int Unit, int Operation)`, with the variables `Unit` and `Operation` being numerical codes for their corresponding unit and operation. In the main portion of the code or from another function, this `fund_function` may be called over and over and include other unit and operation numbers. A more thorough discussion of how the system was implemented in code is given later in this chapter.

Table 4.3: Hybrid Rule Types

Hybrid RuleType	*Example*
Function-Connection	Unit A must be connected to Unit B if Unit As function is addition.
Connection-Timing	Unit A must be connected to Unit B within the first initial ten time units.
Function-Timing	Unit A must be performing addition on even-numbered time units.
Function-Connection-TIming	Unit A must do addition on its inputs and connect to Unit B if at least 10 time units have passed.

4.3.1 Combination Rules

Rules can reference other rules. For instance, Rule 2 may state: If Rule 1 is not true, Rule 3 will also not be true. It can be said that Rule 3 is dependent on Rule 1. When Rule 1 is found to not be true it will be turned off and because of the dependency found in Rule 2, Rule 3 will also be turned off. Rules that do this will be called combination rules.

4.3.2 Rule Dependency

Rules can form a chain of dependencies. Thus, if a rules is found to be untrue (or violated) it can cancel or turn off many other rules. How many rules are truly independent?

4.4 Complexity

As can be seen by the last rule Hybrid Rule type, rules can be complex. Multiple units can be named in the Function-Connection-Timing rule to create more specificity. For instance, Unit A and Unit B must be adding their inputs and connected to Unit C or Unit D if at least 10 time units have passed. Complexity is something typically measured in algorithm studies and computer science. How can the complexity of one of these networks be measured? Some parameters to consider are: number of units, number of connections per unit (average connections), and number of functions available per unit. The complexity measures are a value calculated with the assumption that all parameter values are equally likely for all units. In doing the simulations, units with substantially higher parameter values (a much higher number of inputs for example) than other units may occur. This unit may skew the complexity value of the network. Thus, a higher-order statistical measure (variance for example) would need to be included in the complexity measure. Working with this assumption that each unit will have the same parameter range for each parameter allows for a rather simple calculation of the complexity; only averages are used.

$$A^D \times (B + C + E) = X \tag{4.1}$$

The complexity value equation, where: A is the number of units in the network, B is the number of inputs per unit, C is the number of outputs per unit, D is the number of functions per unit, and E is the number of control lines per unit. The equal-parameter assumption

implies that we have uniform units that can be set in a way that each are identical. There would be no special units that perform only one function or one kind of function. The complexity value of a network shown below can be calculated as follows. Each unit has two available functions and one control line. Units A and B have one input connections and two output connections each. Units C and D have two input connections and one output connections each. Thus, putting those values into the formula:

$$4^2 \times (1.5 + 1.5 + 1) = 64 \tag{4.2}$$

gives the complexity score. The network has a complexity value of 64. This score is only relevant when measured against other networks, below is a table with some sample networks and their respective scores.

Table 4.4: Sample Network Complexities

Type of Network	Complexity Score
Butterworth Filter ($n = 2$) network	61440 (A = 4, B = $\frac{7}{4}$, C = 1, D = 7, E = 1)
FFT (4-point)	$1.07x10^9$ (A = 16, B = 2, C = 1, D = 7, E = 1)
Simple Feedback Control	8.485 (A = 2, B = 1, C = 1.5, D = 1.5, E = 0.5)

Table 4.4 shows how quickly the complexity of a network can grow given more options in functions and number of units. The three examples in this table correspond to the three functions listed later in this section.

4.5 Connection Matrices

In the simulation program, the network connections are represented by an adjacency matrix. The columns of the matrix represent the outgoing connections of that unit specified by the column number. When a 1 is an column i and row j, that means there is a connection outgoing from unit i incoming to unit j.

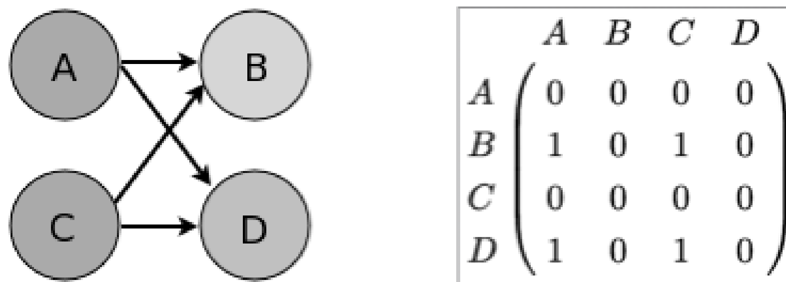


Figure 4.2: All other arrows except for those representing connections are excluded for simplicity.

The matrix shown in Figure 4.2 would have the subsequent network connections. What this allows the software author (and the user) to do is easily see the entire network in a single matrix. Loops within networks can also be searched for easily since patterns are easily identifiable. Unit A outgoing to Unit B and Unit B outgoing to Unit A is found by seeing that a 1 is in both (0,1) and (1,0). This can be generalized to a rule: if (row i, col j) and (row i+1, col j-1) are both 1 than a loop exists. Loops involving more than two units can be checked just as easily.

4.6 Desired Outcomes

This section gives idealized examples for what the outcomes of the simple prototype structures should be.

4.6.1 Butterworth Filter

We define a second-order Butterworth filter ($n = 2$) as having the transfer function:

$$H(s) = \frac{1}{s^2 + x\sqrt{2} + 1} \tag{4.3}$$

Using the unit cells, one implementation can be shown as follows. The input coming from the environment is either the value of the frequency of the signal (10 Hz, 120 MHz, etc) or the input can be a raw signal and the frequency be found by a frequency detector subsystem also made by unit cells. We will discuss and show the former and simpler implementation. The output to the environment is the attenuation of the input signal.

4.6.2 Fast Fourier Transform

The second tested outcome for the simple prototype study is the Fast Fourier Transform with $n = 4$ (4 point FFT). Again, to reemphasize, the Butterworth filter, FFT and feedback

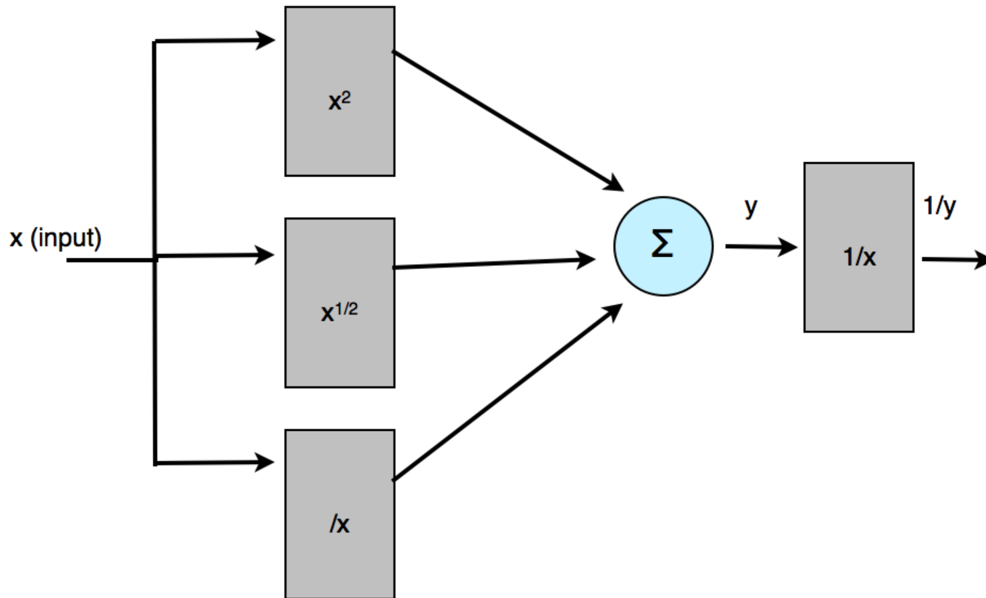


Figure 4.3: Idealized outcome for Butterworth filter structure from simple prototype study.

control system are desired outcomes only meant to illustrate the difficulty in making an open-ended system. By reducing the possible solution number to three it becomes easier to illustrate and much easier to implement for that sole purpose. I could have selected many more different solutions, however, the idea is just to illustrate.

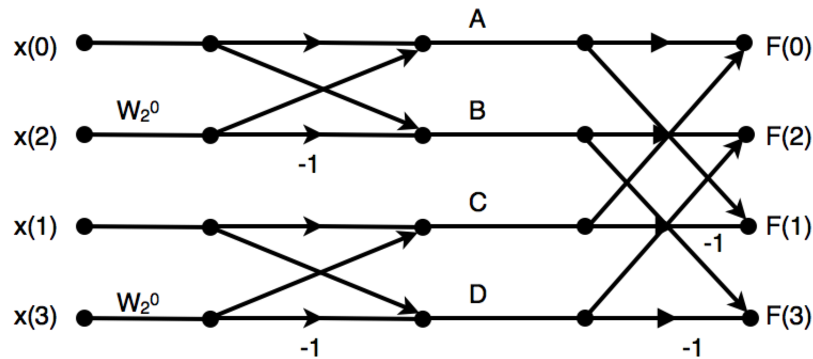


Figure 4.4: Standard butterfly diagram of FFT with n = 4.

The diagram in 4.4 is shown with intermediate states $A, B, C, andD$. They correspond to

$$A = x(0) + W_2^0 x(2) \tag{4.4a}$$

$$B = x(0) - W_2^0 x(2) \tag{4.4b}$$

$$C = x(1) + W_2^0 x(3) \tag{4.4c}$$

$$D = x(1) - W_2^0 x(3) \tag{4.4d}$$

Generally, the Discrete Fourier Transform is found by

$$F(n) = \sum_{k=0}^{N-1} x(k) e^{\frac{-j2\pi kn}{N}} \tag{4.5}$$

where $W_N = e^{\frac{-j2\pi}{N}}$. One can see that individual unit cells could easily take the place of the arithmetic functions in the FFT.

4.6.3 Simple Feedback Control

The third solution we will be looking for is a simple feedback control system. This consists of an input, a threshold detection and an increment or decrement unit. As input is received the threshold detector detects if it is above or below the reference signal. If it is equal to the reference, the incremented-decrementer unit will decrement. This decision is arbitrary.

Input will occur every five time units, therefore the output has time to settle to the reference value.

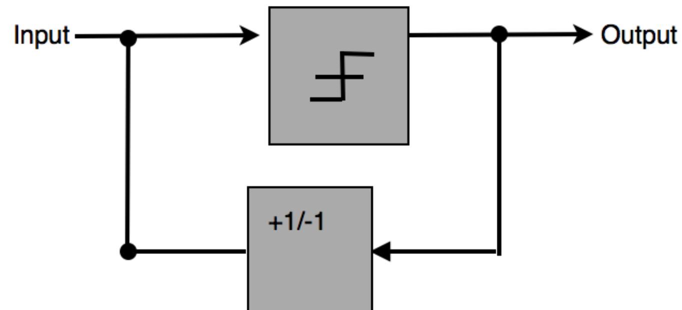


Figure 4.5: Simple feedback control system. The threshold detector outputs a signal to the next unit to increment or decrement.

4.7 Results of the Simple Prototype Study

The results of the simple prototype study were rather inconclusive in that rarely was there a time when any of the three available solutions were reached. Tailoring the rules needed to get to a solution was the most difficult part of this. The solution of the least complexity was the one in which the simulation program actually converged. This was the simple feedback control system solution.

Figure 4.6 shows the results of the software. The output will oscillate between 4 and 5 since this is what the threshold was set to.

```
Time = 0        System Input[0] = 1
B: ON   Mode: 0
B Output = 2    Total: 2
MODE CONTROL = 2

Time = 1        System Input[1] = -1
B: ON   Mode: 0
B Output = 0    Total: 2
MODE CONTROL = 2

Time = 2        System Input[2] = -1
A: ON
B: ON
Connection: A -> B
Mode: 0
A output = 0    B output = 1
Total = 3
MODE CONTROL = 3

Time = 3        System Input[3] = -1
A: ON
B: ON
Connection: A -> B
Mode: 0
A output = 0    B output = 1
Total = 4
MODE CONTROL = 4

Time = 4        System Input[4] = -1
A: ON
B: ON
Connection: A -> B
Mode: 1
A output = 0    B output = -1
Total = 3
MODE CONTROL = 3

Time = 5        System Input[5] = 1
A: ON
B: ON
Connection: B -> A
Mode: 0
A output = 3    B output = 2
Total = 6
MODE CONTROL = 6
```

Figure 4.6: Simple prototype software running and converging to the simple feedback control system solution.

CHAPTER 5

GPRAM SIMULATIONS AND LEARNING FROM PERFECT CODEWORDS

5.1 (7,4) Hamming Code with Perfect Codewords

The simulations of the GPRAM were done in the C language. The parity matrix, stored in a separate file, was read by the program. An identity matrix was concatenated with the parity matrix to form the generator matrix, from which the codewords are generated. Connections are then randomly chosen so that each s_1, s_2, a_1, a_2, and a_3 is connected to one of the seven variable nodes. The remaining two variable nodes are unconnected and are set to zero.

One issue of this simulation is how to represent the connections of the variable nodes and sensors/actions/switches. The first case has the connections were stored in a matrix named selection. This matrix has five elements representing s_1, s_2, a_1, a_2, and a_3. Each element holds a number 1-7 representing one of the seven variable nodes. A second way of representing the connections is with a two-dimensional matrix where the rows represent the sensors/actions and the columns represent the variable nodes. For the (7,4) Hamming code case, this comes to a 5 x 7 matrix. Where there is a connection, a “1” is stored. An

example is if s_1 and v_2 are connected, there would be a "1" stored at `connections[0][1]`. This is an easier way from a programming and software writing standpoint because it allows an easy way to not only see what is connected to what, but what is not connected. When searching for a new valid connection it is important to know what is not connected. This facilitates the search by limiting the number of words to try [Lei12].

Table 5.1: Connection Representation in a two-dimensional matrix.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
s_1	0	1	0	0	0	0	0
s_2	1	0	0	0	0	0	0
a_1	0	0	0	1	0	0	0
a_2	0	0	0	0	1	0	0
a_3	0	0	1	0	0	0	0

5.2 (7,4) Hamming Code with Imperfect Codeword

This section will illustrate the simulations of the GPRAM system with (7,4) Hamming code with imperfect codewords. What we introduce now is noise to our system to have a better grasp of how robust and versatile the GPRAM really is.

Before establish any connection between v_s and $s1, s2, a1, a2, a3$, we input ? as log-likelihood ratio (LLR) of variable node j, where ?, ?is an independent Gaussian noise of zero mean and variance of, and l is iteration number. The noise at each iteration is different and

$$v_{ji}^l = \begin{cases} u_j^l & l = 0 \\ u_j^l + \sum\limits_{\substack{k=1 \\ k \neq i}}^{d_v} u_{kj}^{l-1} & l \neq 0 \end{cases} \tag{5.1}$$

$$\tanh \frac{u_{ij}^l}{2} = \prod_{\substack{k=1 \\ k \neq j}}^{d_c} \tanh \frac{v_{ik}^l}{2} \tag{5.2}$$

$$x_{ji}^l = \begin{cases} 1 & v_j^l > 0 \\ 0 & v_j^l < 0 \end{cases} \tag{5.3}$$

independent. We then use the sum-product algorithm to update the message as conventional iterative decoding, i.e.,

where v_{ji} is the LLR from variable node j to check node i, u_{ij} is the LLR from check node i to variable node j, d_v and d_c are degrees of variable and check nodes, respectively. At the end of each of iteration, we make decision of each variable node as

where $v_j^l = u_j^l + \sum_{k=1}^{d_v} u_{kj}^{l-1}$. After a number of iterations, (say Ia), we make connections based on comparison among x_j^l's and values of $s1, s2, a1, a2, a3$ and v_s are established, for those nodes connected to input signals $s1, s2, a1, a2, a3$ the input is $u_j^l = \frac{2y_j^l}{\sigma_a^2}$ as log-likelihood ratio (LLR) of variable node j, where $y_j^l = a_j^l + n_j^l$ and $a_j^l = (2z-1)B$, where z denotes the input logic and B is a constant. The first step is to initialize the system by setting all v_{ji} and u_{ij} to zero. Pick up Task 1, so $s_1\ s_2 = 11$. Since no connection has been established, we have ?for $j = 1,,7$. We update ? according to the two equations above for each iteration. After 30 iterations, we get a 30 by 7 table of x_k^j. Each column is corresponding to one variable

node. We count the numbers of 1 in each column, and connect s_1 s_2 to two columns with largest numbers of 1 in the table, i.e., two locations of variable nodes where s_1 s_2 connect. In one of individual system, s_1 s_2 has connected to v_1 v_2. Next, a_1 a_2 will connect to another two variable nodes. First, we initialize the system by setting all v_{ji} and u_{ij} to zero, but maintain the connections established in Step 1. Pick up Task 1, so $s_1s_2a_1a_2 = 1101$. Now, we have ? for two variable nodes connected to s_1s_2, and ?for the remaining variable nodes. Again, run for 30 iterations and generate a 30 by 7 table of x_j^l ?. Within the remaining 5 unconnected variable nodes, we connect a_1 to the column which has the most of 0; and a_2 to the column which has the most of 1. Again, one of individual GPRAM connects a_1 to v_6 and a_2 to v_7. In this third step, a_3 will connect to one variable node selected from the remaining unconnected nodes. We once again initialize the system by setting all v_{ji} and u_{ij} to zero, but maintain the connections established in Steps 1 and 2. This time, instead of inputting one task (in Steps 1 and 2), we input two tasks. That is, during the first 10 iterations, we have $s_1, s_2, a_1, a_2, a_3] = = [1, 1, 0, 1, 0]$ (Task 1), and then switch to $s_1, s_2, a_1, a_2, a_3 = [1, 0, 0, 1, 1]$ (Task 2). We have $y_j^l = (2z_j^l - 1)B + n$?for four variable nodes connected to $s_1, s_2, a_1, a_2,$ and $y_j^l = n$?for the remaining three variable nodes. After 30 iterations, we generate 30 by 7 table of x_j^l ?. Within the remaining 3 unconnected variable nodes, we connect a_3 to the column which has the largest number of iterations that a_3 is equal to x_j^l?. Again, one of individual GPRAM connects a_3 to v_5. Therefore, s_1, s_2, a_1, a_2, a_3 has all successfully connected to the variable nodes. So during each trial, messages(i.e., some forms of a posterior probability) are passing between variable nodes and check nodes

many times. What's more, due to the noise, s_1, s_2, a_1, a_2 will connect to different locations although in the same connection. Besides, for those unused variable nodes, they are free to connect other sensors or action nodes.

Now, we would like to see what happens when we perform a task with the (7,4) Hamming code. The success of the trials were output to a separate file so we could see what connections were made. The simulations were run for 1000 trials.

Table 5.2: Results from Simulation of GPRAM with (7,4) Hamming Code

$s_1 s_2 a_1 a_2$	a_3	Good (¿ 90%)	Middle (70% - 90%)	Bad (¡ 70%)
1101	1	2	18	46
	0	97	81	54
1001	1	99	80	32
	0	1	19	68
1110	1	85	85	51
	0	14	15	48
0001	1	2	64	54
	0	97	34	45

5.3 (10,5) LPDC Code with Perfect Codeword

Simulations were done with a (10,5) LPDC with perfect codewords. In this case we have no noise injected and every node value is 1 or 0. These simulations gave much better results as the tasks done faster and more accurately. The initial connections of the GPRAM were the only real limiting factor to performance. More optimization could be done to keep well-performing connections schemes and discard ones that gave poor performance.

However, such an optimization would be troublesome to implement because of the law of diminishing returns. The extra computation time and power needed to do this would not have a significant benefit to the system. Longer codes increase the probabilities of connections and the number of available Longer codes will give less desirable results.

5.4 Effects of Error in Codewords

Mainly, the speed at which the tasks are completed is the largest discrepancy between error codewords and perfect codewords; which is a by-product of the inaccuracies caused by the noise. Noise is something that must be dealt with if a GPRAM is ever to be fully realized. The most important aspect for later contributors and designers is how to limit the noise in the message-passing between the check nodes and variable nodes. Another aspect that my be considered later on is how does noise change affect different levels of the hierarchy?

5.5 Simulation Discussion

One major outcome of this work is the software testbed that came from doing the simulations. The software was written with scalability in mind. The user of the software can enter in any size block code for their simulation in an external text file that is read by the software.

Trials were recorded in an external file as a log of what was done during the simulation. The GPRAM simulations showed that longer block codes, make for worse connections.

```
We are in initialize now
V1 -> A2
V2 -> A1
V3 -> S2
V4 -> S1
V5: Unconnected
V6: Unconnected
V7 -> A3
The beginning of the while loop
S1: v4
S2: v3
A1: v2
A2: v1
A3: v7

Randomized Codeword 0 = Real Codeword 9
Assigning Valid Codeword
        Codeword: W8    S1 = 1  S2 = 0
C1 = 0
C2 = 1
C3 = 0

C_2 invalid
Randomized Codeword 1 = Real Codeword 7
Codeword: 6 - Invalid codeword
Randomized Codeword 2 = Real Codeword 5
Codeword: 4 - Invalid codeword
Randomized Codeword 3 = Real Codeword 3
Codeword: 2 - Invalid codeword
Randomized Codeword 4 = Real Codeword 16
Codeword: 15 - Invalid codeword
Randomized Codeword 5 = Real Codeword 14
Codeword: 13 - Invalid codeword
Randomized Codeword 6 = Real Codeword 12
Assigning Valid Codeword
        Codeword: W11   S1 = 1  S2 = 0
C1 = 0
C2 = 0
C3 = 0

All clear
+++++++++++++++++ ----> 1
```

Figure 5.1: A typical run from the main simulation program window.

One challenge of the simulations was waiting for valid connections since the random number generators sometimes get stuck on the same number for hundreds or thousands of choices.

CHAPTER 6

CONCLUSION

New, novel ways of looking at artificial intelligence are needed. Even if work can be done to disprove a theory or method it is useful to the scientific community. From connectionism to Turing machines to the GPRAM, artificial intelligence research is varied and diverse.

The simple prototype study done in this thesis has shown the difficulty and futility in creating such an open-ended system. What is gained from this portion is that boundaries, rules, and conditions need to exist within a system in order to produce something useful. Our bodies are constantly receiving feedback from the outside environment and from within. Without this feedback, our physiological systems would work unchecked and perhaps unpredictable. However, there are certain rules, chemical and physical, that set the groundwork by which all of our biochemical processes operate. The simple prototype study showed that many systems can be made for many purposes not just by simply changing the structure and function of the individual blocks of the system but by the input of the system. Rules that change the function of unit cells and the structure of the network were used to try and find one of three desirable outputs. Having only three test outputs is a lot like looking for only three specific articles of clothing in a randomlychosen house: it is incredibly unlikely to find what you are looking for. One may ask if our system was general enough? Was it broad

enough or flexible enough? The answer is a certain “no” for all of those questions. However, I believe the problem is not just in the design of a system but in the testing. If we were to accept more outputs as valid, our system would eventually appear to be a “good” system.

GPRAM is a new idea in the very early stages of development. Utilizing hierarchy and associations seems like a good way to build a system that can be brain-like since this is what comes natural to human beings. For the future, a hardware implementation of a GPRAM system can be developed. Currently, work is being done for an FPGA implementation, but the hope is to have an ASIC implementation. The use of a GPRAM could be used in a variety of applications. The best example could be when a user needs an estimate or rough prediction. The GPRAM could then narrow down possibilities and then calculate a more exact answer for the user. Some challenges to GPRAM design in the future is code design. Will any codes be optimal for a wide-variety of uses?

Looking into the future is not easy. Developing new technology and theories is difficult and at times, completely discouraging. However, it is most important. This thesis represents that bluesky thought that has one dreaming. It is utterly satisfying to go into the unknown.

LIST OF REFERENCES

[For01] Luigi Fortuna. *Soft Computing*. Springer-Verlag, 1st edition, 2001.

[Lei12] Huihui Li Lei Wei, Steven Schultz. "General purpose representation and association machine part 3: Prototype study using LDPC codes." pp. 1–5, March 2012.

[Nil10] N.J. Nilsson. *The quest for artificial intelligence: A history of ideas and achievements*. Cambridge University Press, 2010.

[Rob90] Barry Edelson Robert I. Levine, Diane E. Drang. *AI and expert systems : a comprehensive guide, C language*. McGraw-Hill, 2 edition, 1990.

[Teu02] Christof Teuscher. *Turing's connectionism : an investigation of neural network architectures*. Springer-Verlag, 2002.

[Wei12] Lei Wei. "General purpose representation and association machine part 1: Introduction and illustrations." In *Southeastcon, 2012 Proceedings of IEEE*, pp. 1 –5, March 2012.