2007

An Analog Evolvable Hardware Device for Active Control

Saranyan A. Vigraham
Wright State University

Follow this and additional works at: http://corescholar.libraries.wright.edu/etd_all
Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation

This Dissertation is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu.
An Analog Evolvable Hardware Device for Active Control

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science and Engineering)

By

SARANYAN A. VIGRAHAM
M.S., WRIGHT STATE UNIVERSITY, 2003

2007
Wright State University
I HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER MY SUPERVISION BY Saranyan A. Vigraham ENTITLED An Analog Evolvable Hardware Device for Active Control BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy in Computer Science and Engineering.

__________________________________________________________
John Gallagher, Ph.D.
Dissertation Director

__________________________________________________________
Thomas Sudkamp, Ph.D.
Director, PhD Program,
Department of Computer Science and Engineering

__________________________________________________________
Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies

Committee on Final Examination

__________________________________________________________
John Gallagher , Ph.D.

__________________________________________________________
Michael Raymer , Ph.D.

__________________________________________________________
Mateen Rizki , Ph.D.

__________________________________________________________
Jack Jean , Ph.D.

__________________________________________________________
Mitch Wolff , Ph.D.

__________________________________________________________
Steven Garverick , Ph.D.
ABSTRACT

Vigraham, Saranyan . PhD., Computer Science and Engineering Ph.D Program, Department of Computer Science and Engineering, Wright State University, 2007 . An Analog Evolvable Hardware Device for Active Control.

The field of Evolvable Hardware (EH) has recently gained a lot of interest due to the novel methodology it offers for designing electrical circuits and machines. EH techniques involve configuring a reconfigurable hardware platform with the aid of learning engines such as evolutionary algorithms. The EH devices normally act as closed loop controllers with the capability of learning necessary control laws adaptively. Current EH practices have several shortcomings, which have restricted their use as reliable controllers. This dissertation will present an improved EH device based on behavioral reconfigurability that addresses the current open challenges in the field of analog Evolvable Hardware. This EH device is based on Continuous Time Recurrent Neural Network (CTRNN). The design and implementation of the CTRNN-EH device and a custom designed evolutionary learning engine will be presented in this work. In addition to answering the open challenges in the field of EH, this dissertation will also provide a novel programming circuitry to by which a VLSI CTRNN can be effectively programmed. Furthermore, a closed loop calibration scheme based on Evolutionary Algorithms is presented to address the effects of random offset variations in the CTRNN design.
Contents

1 Introduction .......................................................... 1
   1.1 Evolvable Hardware ............................................ 1
   1.2 Definitions used in the context of this dissertation ....... 2
   1.3 Structural vs. Behavioral Reconfigurability ................. 3
   1.4 Towards building a CTRNN-EH Device ....................... 4

2 Background: EA, CTRNNs, Analog Reconfigurable Hardware .... 6
   2.1 Evolutionary Algorithms ................................... 6
   2.2 Hardware EAs : The broader picture ....................... 9
   2.3 Hardware EAs : A Brief Survey ............................ 12
      2.3.1 Taxonomy .............................................. 12
      2.3.2 FPGA Implementations ............................... 13
      2.3.3 Non-FPGA Implementations ......................... 16
   2.4 Analog Reconfigurable Hardware .......................... 19
   2.5 Artificial Neural Networks ................................ 20
   2.6 CTRNNs ...................................................... 21
      2.6.1 Architecture of the CTRNN ......................... 22
      2.6.2 Configuring a CTRNN ............................... 24
      2.6.3 Modes of operation ................................ 24
   2.7 CTRNNs and Evolvable Hardware ............................ 25
      2.7.1 CTRNNs in Hardware .................................. 27
   2.8 Existing Analog VLSI CTRNN Implementations ............... 27
      2.8.1 Work of Gert Cauwenbergs ......................... 28
      2.8.2 Work of Bruce E Brown ............................. 29
      2.8.3 Work of Ryan J Kier ............................... 30
   2.9 Open Issues ................................................. 31

3 A Digital VLSI Evolutionary Engine for the CTRNN-EH Device ... 32
   3.1 Introduction ............................................... 32
   3.2 Minipop : EA Component of CTRNN-EH Device .............. 32
   3.3 Translating into hardware: Datapath ...................... 35
   3.4 Intricacies in evolving CTRNN configurations ............ 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1 The Memory Interface</td>
<td>41</td>
</tr>
<tr>
<td>3.5 Results: Simulation, Synthesis and Analysis</td>
<td>44</td>
</tr>
<tr>
<td>3.6 Conclusions</td>
<td>47</td>
</tr>
<tr>
<td>4 VLSI Implementation of the first generation 2x2 CTRNN</td>
<td>48</td>
</tr>
<tr>
<td>4.1 Block level architecture of a 2x2 CTRNN</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Design decisions</td>
<td>50</td>
</tr>
<tr>
<td>4.2.1 Budgetary</td>
<td>50</td>
</tr>
<tr>
<td>4.2.2 Circuit Implementation</td>
<td>50</td>
</tr>
<tr>
<td>4.3 Circuit implementation</td>
<td>51</td>
</tr>
<tr>
<td>4.3.1 Synapse and Summer</td>
<td>51</td>
</tr>
<tr>
<td>4.4 Sigmoidal Activation Function</td>
<td>54</td>
</tr>
<tr>
<td>4.5 Circuit diagram of a CTRNN row</td>
<td>55</td>
</tr>
<tr>
<td>5 A Memory Cell for Storing Programming Parameters in the Generation One CTRNN</td>
<td>64</td>
</tr>
<tr>
<td>5.1 Modifying Synapse and Sigmoid Cells</td>
<td>65</td>
</tr>
<tr>
<td>5.2 Charge Injection and Capacitive Feedthrough</td>
<td>67</td>
</tr>
<tr>
<td>5.2.1 Charge Injection</td>
<td>68</td>
</tr>
<tr>
<td>5.2.2 Capacitive Feedthrough</td>
<td>68</td>
</tr>
<tr>
<td>5.2.3 Reducing Capacitive Feedthrough and Charge Injection</td>
<td>68</td>
</tr>
<tr>
<td>5.3 Dummy switches and Differential Setup for Synapse Memory Cell</td>
<td>70</td>
</tr>
<tr>
<td>6 Programming Circuitry for the Generation One CTRNN</td>
<td>74</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>74</td>
</tr>
<tr>
<td>6.2 Digital to Analog Converter</td>
<td>74</td>
</tr>
<tr>
<td>6.2.1 Current steering DAC</td>
<td>75</td>
</tr>
<tr>
<td>7 Measured Results on the Generation one 2x2 CTRNN Device</td>
<td>82</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>82</td>
</tr>
<tr>
<td>7.2 Synapse and Sigmoid test</td>
<td>83</td>
</tr>
<tr>
<td>7.3 Digital to Analog Converter</td>
<td>85</td>
</tr>
<tr>
<td>8 Testing the Generation One CTRNN Device</td>
<td>90</td>
</tr>
<tr>
<td>8.1 Introduction</td>
<td>90</td>
</tr>
<tr>
<td>8.2 Test setup</td>
<td>90</td>
</tr>
<tr>
<td>8.3 Functional Validation of CTRNN</td>
<td>93</td>
</tr>
<tr>
<td>8.3.1 Extrinsic Verification</td>
<td>93</td>
</tr>
<tr>
<td>8.3.2 Finding the Tolerance of a CTRNN configuration in simulation</td>
<td>94</td>
</tr>
<tr>
<td>8.3.3 Intrinsic Verification and Testing</td>
<td>99</td>
</tr>
<tr>
<td>8.3.4 EA based calibration scheme</td>
<td>102</td>
</tr>
<tr>
<td>9 Generation 2 CTRNN Device</td>
<td>110</td>
</tr>
<tr>
<td>9.1 Introduction</td>
<td>110</td>
</tr>
<tr>
<td>9.2 Synapse design</td>
<td>111</td>
</tr>
<tr>
<td>9.3 Sigmoidal Activation Function</td>
<td>114</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Explanation of EA with an Example .............................................. 7
2.2 An Overview of CTRNN Operation ................................................. 23
2.3 An Overview of CTRNN configuration procedure .............................. 26

3.1 The Minipop algorithm ................................................................. 34
3.2 Datapath of the Minipop ............................................................... 36
3.3 The Memory Interface ................................................................. 42
3.4 A flow diagram of integrated architecture ....................................... 43
3.5 Simulations of One Max Problem using Minipop .............................. 44

4.1 Equivalent block diagram of a neuron in a CTRNN ............................. 49
4.2 Circuit diagram of a synapse .......................................................... 52
4.3 Synapse test circuit ................................................................. 56
4.4 Synapse simulation results ............................................................ 57
4.5 Synapse layout ......................................................................... 58
4.6 Circuit diagram of a sigmoid ......................................................... 59
4.7 Sigmoid test circuit ................................................................. 60
4.8 Sigmoid simulation results ............................................................ 61
4.9 Complete circuit diagram of a single row in a CTRNN ....................... 62
4.10 (a) Sigmoid Layout  (b)D2S layout  (c)PMOS mirror layout .............. 63

5.1 (a) Capacitor to store memory  (b) Switch transistor ......................... 65
5.2 Four memory cells connected in an an array-like fashion .................... 66
5.3 (a) Demonstration of charge injection  (b) Demonstration of capacitive Feedthrough 67
5.4 Dummy switch to cancel feedthrough and injection effects ................... 69
5.5 Memory cell with dummy switch ..................................................... 70
5.6 (a) Layout of memory cell with dummy switch  (b) Differential memory cell 72
5.7 Layout of 2x2 CTRNN with memory cell ......................................... 73

6.1 Basic current steering DAC .......................................................... 75
6.2 Binary weighted current mirror DAC .............................................. 77
6.3 Relation between neuron weights and programming voltages .............. 78
6.4 (a) IV Conversion circuit  (b) Simulation snapshot of DAC with IV conversion 80
6.5 Layout of the DAC ................................................................. 81
7.1 Layout of the padframe of generation one CTRNN ................................. 83
7.2 Testing the synapse and sigmoid circuits ........................................ 84
7.3 LM117 for current source .................................................................... 85
7.4 Measured results synapse .................................................................... 86
7.5 Measured results sigmoid .................................................................... 87
7.6 Testing the digital to analog converter ................................................. 88
7.7 Measured results sigmoid .................................................................... 89
7.8 Integral Nonlinearity ........................................................................... 89

8.1 Test setup ............................................................................................. 91
8.2 CTRNN time series for configurations in previous table ....................... 95
8.3 CTRNN behavior at the boundaries of tolerance sweep ....................... 96
8.4 CTRNN behavior at the boundaries of bias tolerance sweep .................. 97
8.5 CTRNN behavior at the boundaries of combined tolerance sweep .......... 98
8.6 Recorded and filtered output of CTRNN (configuration 1) ...................... 104
8.7 Recorded and filtered output of CTRNN (configuration 2) ...................... 105
8.8 Recorded and filtered output of CTRNN (configuration 3) ...................... 106
8.9 Configuration 1 HW and Sim outputs .................................................. 107
8.10 Configuration 2 HW and Sim outputs .................................................. 107
8.11 Configuration 3 HW and Sim outputs .................................................. 108
8.12 Scatter plot of the parameter perturbations on-chip .............................. 108
8.13 Configuration 1 after calibration ......................................................... 109
8.14 Configuration 2 after calibration ......................................................... 109

9.1 Architecture of four neuron arrangement .............................................. 111
9.2 Circuit diagram of a synapse ................................................................ 112
9.3 Synapse simulation results .................................................................. 114
9.4 Synapse layout ..................................................................................... 115
9.5 Circuit diagram of a sigmoid ............................................................... 116
9.6 Sigmoid test circuit .............................................................................. 117
9.7 Sigmoid simulation results ................................................................... 118

10.1 PCB layout for generation two device ................................................ 120
10.2 PCB layout for generation two device ................................................ 121
10.3 Programming circuitry for synapses .................................................... 122
10.4 Programming circuitry for synapses .................................................... 122
10.5 Measured results CTRNN: extrinsic 1 ................................................ 125
10.6 Measured results CTRNN: extrinsic 2 ................................................ 126
10.7 Measured results CTRNN: extrinsic 3 ................................................ 127
10.8 Measured results CTRNN: extrinsic 4 ................................................ 127
10.9 Sim vs filtered hardware output: config1 .......................................... 128
10.10 Sim vs filtered hardware output: config2 ......................................... 128
10.11 Sim vs filtered hardware output: config3 .......................................... 129
10.12 Sim vs filtered hardware output: config4 .......................................... 129
10.13 Measured results CTRNN: intrinsic 1 ............................................... 131
10.14 Measured results CTRNN: intrinsic 2 ............................................... 132
10.15 Measured results CTRNN: intrinsic 3 ............................................... 132
10.16 Measured results CTRNN: intrinsic 4 ............................................... 133
10.17 Sim vs filtered hardware output: configuration 1 . . . . . . . . . . . . . . . . . . . 133
10.18 Sim vs filtered hardware output: configuration 2 . . . . . . . . . . . . . . . . . . . 134
10.19 Sim vs filtered hardware output: configuration 3 . . . . . . . . . . . . . . . . . . . 134
10.20 Sim vs filtered hardware output: configuration 4 . . . . . . . . . . . . . . . . . . . 135

11.1 CTRNN-EH Device for Noise control . . . . . . . . . . . . . . . . . . . . . . . . 138
11.2 Current regulator circuit used for sensory inputs . . . . . . . . . . . . . . . . . . . 140
11.3 Block diagram showing control system . . . . . . . . . . . . . . . . . . . . . . . . 141
11.4 Phase inversion controllers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
List of Tables

2.1 CTRNN will help the proposed EH device achieve its desired functionality . . . . . 27
3.1 Table showing increase in genome length with increase in number of neurons . . . 40
3.2 The synthesis results of the minipop algorithm as used for one max problem . . . 45
3.3 Table indicating the chip area consumed by digital EA engine . . . . . . . . . . . . 46
8.1 Memory map . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
8.2 Some CTRNN configurations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
8.3 Microcontroller programming values for previous table . . . . . . . . . . . . . . 94
8.4 Some intrinsically evolved CTRNN configurations . . . . . . . . . . . . . . . . . 100
8.5 Actual parameter values found by fine tuning ideal CTRNN models . . . . . . . 101
10.1 Extrinsic Test: Configuration 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
10.2 Extrinsic Test: Configuration 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
10.3 Extrinsic Test: Configuration 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
10.4 Extrinsic Test: Configuration 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
10.5 Intrinsic Test: Configuration 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
10.6 Intrinsic Test: Configuration 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
10.7 Intrinsic Test: Configuration 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
10.8 Intrinsic Test: Configuration 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
I would like to extend my thanks to - John Gallagher for his continued support and close to infinite patience during my research and document writing, Steven Garverick for his guidance and support through out the research phase, Michael Raymer for his wonderful proof reading and document correction, Greg Kramer and Sanjay Boddhu for awesome brainstorming sessions, and all my other friends for showing understanding and patience during my PhD. This research would not have been possible without the aforementioned people and my family. In particular, my wife Tripti has shown extraordinary patience, consideration and understanding in all the phases of my research. She has been a source of inspiration during the ups and downs of this wonderful process. Special thanks to her.
Dedicated to
My mentor: John Gallagher
My parents: The Vigrahams
My wife: Tripti
Chapter 1.
Introduction

1.1 Evolvable Hardware

Evolvable Hardware (EH) is the practice of using evolutionary algorithms to design machines or electrical circuits. Evolvable hardware has recently generated significant research interest owing to its immense potential to generate novel circuits and architectures. Evolvable hardware has gained a lot of attention in the field of control system design [48, 25, 3]. The capability of finding a possible controller for a system using EH techniques is worthy of study. Several works exist in the literature where EH techniques were applied for space applications [48] and other engineering applications [50, 17, 57]. However, two challenging impediments have slowed the adoption of EH techniques for practical problems.

1. Evolutionary algorithms evolve circuit configurations based on how well the circuit meets one’s requirements (behavioral functionality). In practice, it is not uncommon to see circuit configurations that cannot be understood or explained. Also, when a
circuit configuration is evolved directly on reconfigurable hardware [31], device par-
asitics and electromagnetic interferences are falsely exploited making it impossible
to analyze the circuit.

2. An extension of above point is that the circuits evolved in hardware rarely work in
simulation and vice-versa. Owing to this, EH techniques have not been considered
entirely reliable.

1.2 Definitions used in the context of this dissertation

- Throughout the entire document, two terms are repeatedly used in the context of
  VLSI architectures used in Evolvable Hardware - Static and Dynamic. The defini-
tions are listed below:

  1. Dynamic architecture: Several EH practices involve evolving interconnects on
     a fabricated array of transistors [48]. In these circuits, prior to fabrication, it
     is impossible to determine every possible circuit configuration due to the ex-
     tremely large number of possible permutations. Even a simple circuit can have
different working versions on the device composed of different architectures or
signal paths. The shape a circuit might take is run time dependent or dynamic.
This is referred to as dynamic architecture.

  2. Static architecture: Prior to fabrication, the architecture of a circuit is well un-
derstood. The parasitics, noise and other critical circuit parameters are deter-
mined. No interconnects are evolved meaning the device architecture does not
change during evolution. The reconfigurability is achieved by programming the circuit parameters. Here, the reconfigurability is behavioral reconfigurability.

- **Behavioral reconfigurability**: Systems like neural networks are reconfigurable by changing their neural parameters. Changing the parameters of such a system helps one realize a different behavior. Such systems are termed behaviorally reconfigurable.

- **Structural reconfigurability**: Systems like analog arrays of transistors can be configured to form different circuits by changing switching connections. These systems are structurally reconfigurable.

### 1.3 Structural vs. Behavioral Reconfigurability

I believe the outstanding challenges in EH techniques can be overcome by employing behavioral reconfigurability instead of structural reconfigurability. This can be understood by the following illustration. Suppose a circuit is being evolved to function as some analog circuit on a reconfigurable hardware device like an Field Programmable Analog Array (FPAA). Using evolutionary algorithms, the FPAA is configured in different ways so that the required circuit can be realized. Functionally, we might find a circuit that works as desired, however when the circuit is ported back to the simulation software, a different behavior might be noted. In this case, the underlying device properties and electromagnetic effects would be causing the device to work as required. In practice it is much more complicated, but this simplification is representative of what happens. This phenomenon is more
visible in custom fabricated integrated circuits that are supposed to function as reconfigurable hardware e.g., Field Programmable Transistor Array (FPTA). Because the circuit’s architecture is dynamically being changed for evaluation after fabrication, hardware effects are possible due to permutations of circuit elements not observed in simulation. This problem can be resolved using behavioral reconfigurability.

The next point is if a circuit has static architecture (or is behaviorally reconfigurable), then how do we configure it in different ways? Neural network subsystems can be understood as behaviorally reconfigurable meaning that to change the block behavior we do not directly change the system architecture. We do this indirectly by changing its parameters. Logically speaking, systems like neural networks that are behaviorally reconfigurable might be better EH devices.

### 1.4 Towards building a CTRNN-EH Device

Continuous Time Recurrent Neural Networks (CTRNNs) [4] are dynamically expressive and behaviorally reconfigurable. These neural systems will be used to demonstrate the idea of behavioral reconfigurability in EH applications. Because CTRNNs have shown to be useful in control applications, I believe that a CTRNN-EH device will combine the benefits of both realms and thus yield a strong control device that can be analyzed.

As shown in chapter 2, CTRNNs have been successfully implemented in VLSI on three occasions, showing the engineering feasibility. However, these existing implementations in their current forms are not suitable for control applications. The reasons will be
clear during the detailed discussion in chapter 2. Hence, a VLSI implementation suitable
for EH applications is necessary. Further, for EH applications, an Evolutionary algorithm
acts as a learning engine. To configure the CTRNN module, a learning engine like an
EA is required. This EA will be implemented using digital VLSI techniques, making the
CTRNN-EH device a mixed signal VLSI device. The subsequent chapters will present the
implementation details and background material that is required to clearly understand the
presented work. Results demonstrating the efficacy of CTRNN-EH device for evolvable
hardware applications will be presented. While designing the CTRNN-EH device, several
new things in analog circuit design and calibration schemes were learned, which will be
presented in detail in the later sections of this document.
This dissertation details the analysis and construction of an Evolvable Hardware (EH) device, consisting of a Continuous Time Recurrent Neural Network (CTRNN) and an Evolutionary Algorithm (EA). In the context of this work, it is necessary to know about various existing hardware implementations of CTRNNs and EAs. This chapter will focus on giving a basic introduction to EAs and CTRNNs, and discussing their existing hardware implementations in the literature.

2.1 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are based on the Darwanian principles of natural evolution. A pool of possible solutions (usually encoded as bit-strings) is maintained. Biological operations like mutation and cross-over are applied. The philosophy of an EA lies in “survival
of the fittest”. The better individuals mate, reproduce and survive, while the inferior individuals are eliminated. The Figure 2.1 describes the basic architecture of an EA. This figure illustrates the flow diagram of an EA with a simple example, one-max problem. The One-max problem is a commonly-employed test problem to demonstrate the functionality of an EA. The problem involves finding the bitstring with the maximum number of ones. In this example, the goal is to find the four bit bitstring with most ones. The EA operates by cre-
ating an initial population at random. The initial population consists of random bitstrings that are considered as possible solutions. Here, the population size, as observed from the figure is four. After the initial population has been created, each individual must be assessed to determine quality. In this problem, the quality of a bitstring is determined by the number of ones it contains. Once this evaluation of quality or fitness has been performed, the next stage is to apply various genetic operators to various individuals to improve the overall fitness of the population. One common way to achieve this is to, pick the individuals from the population according to some selection scheme, perform the genetic operations like recombination and mutation on the selected strings and re-introduce them into the original population. Recombination generates new individuals from two parent individuals by combining their attributes together. Various types of recombination schemes exist in the literature [?] and are not discussed in this document. Mutation is a process performed to introduce variety in the population. A bit position is picked at random and is inverted [?].

Different forms of EAs exist in the literature. An early example is the Genetic Algorithm (GA), described by Goldberg [32]. From that time, numerous modifications to the simple EA have resulted. At present, the EA is considered as one of the most powerful search and optimization techniques. It has been extensively employed in a wide variety of applications including pattern recognition, artificial intelligence, robotics, electronic circuit design, space navigation and many other real world applications. For instance, EAs are used to mine data from large databases [13], classify complex data from medical applications [49], autonomously navigate robots [45], and to evolve novel electronic circuits.
adhering to stringent space and power constraints [50]. As mentioned earlier in the previous chapter, the goal of this work is to build an Evolvable Hardware device with EA and a reconfiguring engine on a single VLSI device. For achieving this goal, the EA has to be implemented in hardware and more precisely, minimal hardware. Having the EA in a minimalistic hardware implementation will reduce the size of the EH device, increase its applicability to real world problems with stringent size requirement, and render eventual implementation easier. Numerous researchers are interested in optimizing the EA for speed rather than size. In this work, the EAs will be configuring controllers that are interfaced to real world electro-mechanical devices that will be inherently much slower than the digital clock speeds on chip. Therefore, it makes no sense for us to worry about speeding up the EA. For this work, the only relevant concerns with respect to the EA are whether it works and how small and power efficient we can make it. Having discussed briefly the motivation for the hardware EAs, the next section will present the existing hardware EAs form literature.

2.2 Hardware EAs: The broader picture

The previous section describes the motivations for designing Hardware EAs (HEAs). The HEA research can be divided into two broad camps: speedup [23, 57] and size [57, 53, 21]. Researchers from the first camp are interested in making the EA run as fast as possible. The second group of researchers target size (and indirectly, power consumption). This group of researchers try to implement the EA in the fewest hardware components possible. The mo-
tivations for the choice are the end applications that the EA would be used in. It is more interesting and beneficial to reduce the size of the EA device so that it can be easily fit into the applications with size and power constraints. A small device is more prone to consume lesser power than a bigger device. Further, speedup can be obtained only to a certain extent because of the fitness evaluation bottleneck. An astute reader might observe the fact that the fitness evaluation time taken by an EA cannot be sped up faster than the real time it takes for evaluating the fitness. So, there is already an upper bound on the speedup that can be achieved. It might be thus, a reasonable goal to target just the minimum speedup necessary and emphasize on reducing the device size or area.

Different design choices exist while implementing EAs in hardware. The most common ones are Field Programmable Gate Arrays (FPGAs) and application specific Integrated Circuit (IC) designs. FPGA devices consist of a matrix of highly reprogrammable logic ICs. These devices combine the flexibility of software with the speed of hardware. They can serve as computational co-processors capable of performing a wide range of arithmetic and logic operations. FPGA implementations of EAs allow the programming and reprogramming of various genetic operational modules. For instance, the type of cross-over mechanisms, the rate at which mutation is performed, the number of evaluation cycles and other miscellaneous evolutionary details. The FPGA implementations offer the advantage of providing flexibility to test the EA for different combination of operators or different sets of parameters on any problem. Further, for computational intensive applications like data mining and pattern recognition, the FPGA implemented EAs can act as fast co-processors.
FPGA implementation also allows reuse of the device by loading it with a different kind of EA for a different task. While the advantages of FPGA implementation can be exploited to the fullest extent in the computational intensive applications requiring the services of an EA, this methodology is not at all suitable in applications with size and power constraints like the ones discussed previously.

Very Large Scale Integration (VLSI) techniques are popular for integrating numerous components (transistors) onto die (silicon) areas as small as a few $mm^2$. With the transistor sizes shrinking with every day (Moore’s law), it has become possible to build ultra-small devices. VLSI implementation of an EA is attractive for applications that require small hardware EAs. However, once a VLSI device has been fabricated, it is not possible to replace or modify the existing architecture. Such a device becomes “application specific”. These devices do not offer the flexibility that their FPGA counterparts offer. However, they have their own unique advantages.

In general, it can be said that there is no single right implementation technique applicable across all the applications. The hardware choice should be decided based on the end applications being addressed. For the work being discussed in this dissertation, VLSI implementation is more appropriate.
2.3 Hardware EAs: A Brief Survey

This section will provide a brief survey of the various HEA implementations that exist in the literature. The goal of this survey is to recognize the wide variety of HEA implementations.

2.3.1 Taxonomy

HEAs can be broadly classified into functional HEAs and data flow HEAs. Functional HEAs are the result of a direct mapping of the EA into hardware. They are usually decomposed into modules, each of which are designed to perform some function of the EA like mutation, crossover or selection. The underlying hardware modules are mirrored versions of their software counterparts. Functional HEAs can be optimized further to make use of hardware features like pipelines and parallelization. This class of HEAs are known as optimized functional HEAs. Functional HEAs are relatively easy to understand because of the behavioral similarity with the software EAs. These EAs however, only support coarse-grained parallelism. The data flow HEAs attempts to modularize the hardware in some manner convenient for efficient implementation by exploiting the underlying hardware structure on which the EA is being implemented. These can be achieved by modifying the EA operation at a higher level to increase its amenability to hardware implementations, or cleverly designing the HEA so that it exploits any inherent parallelism present by executing instructions out-of-order. This HEAs does not enjoy the same level of transparency as the function HEAs do. However, they possess a finer grained pipelining and parallelism.
2.3.2 FPGA Implementations

HGA: Hardware Based Genetic Algorithm

The HGA [46] is an optimized functional HEA that implements Goldberg’s simple genetic algorithm [56]. It has population memory to store the bit-coded genomes, and individual modules for various genetic operators like mutation, cross-over and roulette wheel selection. It was implemented on FPGA, and this served as a coprocessor for running the actual GA searches faster than it would take to run them normally on the host computer. The primary design goal was speedup. A brief description of the HGA is as follows:

The front end of the HGA system has a simple interface program running on a simple PC or workstation. A user can enter the parameters for the GA interactively through this interface. This is written into a memory that is shared with the back end. The user can specify the fitness function in some programming language like C or VHDL. The specification is translated into a hardware image and the FPGA is programmed to implement the fitness function. After the front end sends a “Go” signal which is detected by the back end, the GA is run based on the parameters already stored in the shared memory. The back end sends back a done signal which is detected by the front end and the final population is read from the shared memory.

The HGA can be classified as a functional HEA implementation. It uses coarse grain parallelism and internal module parallelization for achieving speedup. The selection, crossover and fitness modules were pipelined and the multiple selection modules were parallelized.
It was reported that the HGA required 6% of the clock cycles required by a software based GA and speedups of two to three orders of magnitude were deemed possible.

The speedups reported for the HGA relies primarily on the assumption that the objective function can be evaluated in one clock cycle. To keep this assumption valid for any complex fitness function, large amounts of digital hardware might be necessary. This will make a strong impact on the size of the HEA. However, if the objective fitness evaluation is done external to the device, this issue need not be considered.

**SPGA: SPLASH 2 Parallel GA**

The SPLASH 2 Parallel Genetic Algorithm (SPGA) [22] was proposed primarily to solve instances of the Traveling Salesman Problem and was targeted for implementation on a reconfigurable computer, SPLASH 2. The SPLASH 2 system consisted of a host computer and a collection of reconfigurable hardware sub-processor boards that are programmed through the use of VHDL. The SPGA implements a variation of the simple GA by encoding the order of visiting the cities as genomes rather than bit-strings. It uses problem-specific crossover and mutation operations.

The SPGA, like the HGA, targeted speedup. It employed pipelining and parallelization. This can be classified as an optimized functional HEA. The basic SPGA used four FPGAs, each of which served as a pipeline stage in the four-part pipeline scheme. Each FPGA had its own local memory. The operational details of the SPGA can be understood...
in detail from [22]. The SPGA was implemented on a real SPLASH 2 computer and an approximate speedup on one order of magnitude over a “state of the art” workstation was reported. This speedup might be attributed to the elimination of processing overhead and pipelining. The HEA size is reasonably small when one considers the traveling salesman problem as formulated by the authors. However, the scaling issues for the SPGA are unclear.

**Harware cGA and Variants**

The compact Genetic Algorithm (cGA) [27] is an EA initially proposed for theoretical studies and was later found to be amenable to compact hardware implementations. It employs a simulated population instead of a real population for reducing the memory requirements. The cGA maintains a probability vector giving information about the relative frequency of a bit being zero or one. Although, the cGA is a weak search algorithm, it is attractive for hardware implementation because the amount of space it requires is $O(G \log_2 N)$ as opposed to the $O(GN)$ or $O(N^2)$ (N is the number of genomes and G is the length of a genome) of other HEAs in literature. Apornthewan and Chongsttvatana [1] describes the cGA implementation in hardware. The cGA was modeled using Verilog, a hardware description language, and the design was implemented on an FPGA. Subsequent work by Gallagher et al., [19, 21] suggested modifications for the cGA enhancing its strength and search efficacy, and was demonstrated on FPGA implementation as well. The modified cGA was modeled using VHDL.
2.3.3 Non-FPGA Implementations

Gate Level EHW Chip

The EHW chip [30] is a good example of a functional HEA. The EHW chip integrates two Programmable Logic Arrays (PLAs) and a functional HEA onto a single, monolithic chip for learning PLA configurations. The two PLAs present on the chip are for parallelizing the candidate evaluations. There is very limited hardware optimization otherwise. This chip, primarily designed for evaluation purposes, has a speedup of 50 times the equivalent software implementation. The underlying hardware, in this case, is not an FPGA as it was with the above two cases. This EHW chip is a step towards designing HEAs on silicon for use applications requiring the services of an EA, while imposing space and power constraints on the design.

Ringed Genetic Algorithm

The Ringed Genetic Algorithm (RGA) [41] is a data flow HEA. It exploits the locality properties of hardware by embedding a customized EA into hardware in a specific fashion. Because the RGA relies on local interactions, it has the potential to remain simple even when the population size grows. The circuits evolved by the RGA are implemented on a reconfigurable array of interconnected cells. Cell configurations can be specified using bit streams. These bit streams are the chromosomes created during the operational cycle of the GA due to the result of the genetic operations - crossover and mutation. It is not clear that the RGA has ever been implemented, and hence, it is not possible to make approximations
about size and speed.

**Systolic Array Genetic Algorithm**

Systolic arrays have an array of processing elements that are only locally connected and communicate directly only with nearby neighbors. Input data is introduced at the edges of the array in some orderly fashion and propagated to subsequent cells until the desired computation is performed. The systolic arrays have inherent parallelism due to their pipelined structure. Because communication is local, signal degradation can be avoided. Power consumption can be minimized with a clever design.

Systolic arrays offer attractive solutions for implementing HEA on Silicon. However, they are not generic designs and may be highly inflexible. But, if one considers that the on-chip EAs are usually application specific, systolic arrays may be of interest. To date, only one research group has proposed systolic array implementation of a GA. In [6], the authors proposed a systolic array version of a simple GA. The design was based on a seven segment “macro pipeline” of systolic arrays in which each segment corresponded to a specific GA operation. The implementation was performed in C and relied heavily on loop optimization techniques like loop unrolling. The segments were integrated together to give a complete systolic genetic algorithm. This design had high fine-grain parallelism and can be categorized as a synthesized data flow implementation. The most interesting aspect of such an implementation was the removal of dependance of the performance of the EA on the genome size. From the literature, it appears that this design was never implemented on real hardware.
We have described the various HEAs existing in the literature and presented their advantages and shortcomings. However, none of the existing HEAs mentioned above are the best choices for implementing as the reconfiguring engine on the EH device being designed. This will be discussed in detail in chapter four, which will also provide a design and implementation of a completely functional custom designed EA engine suitable for EH applications.
2.4 Analog Reconfigurable Hardware

Traditionally, analog reconfigurable hardware are defined as analog circuits/programmable arrays that can be dynamically configured in a field of operation. Two major forms of traditional analog reconfigurable hardware exist - Field Programmable Analog Arrays and Field Programmable Transistor Arrays [11, 12]. Field programmable analog arrays (FPAA) have been defined as following. This definition has been used from [11].

A Field Programmable Analog Array (FPAA), built in CMOS technology, contains uncommitted operational amplifiers, switches, and banks of programmable switched capacitors (S/C) and can be used to build filters for analog signals as well as a large number of diverse analog applications. The parameters of a given application, such as a filter, are functions of the capacitor values. The chip is divided into 20 identical, configurable analog blocks (CABs), each composed of an operational amplifier, five capacitor banks, and switches that can be used to interconnect the cell components and determine their operation. There are both static and dynamic CMOS switches. The static switches are used to determine the configuration of cell components and inter-cell connections. These switch settings are determined once during the programming phase of an application after which they remain unchanged. The dynamic switches are associated with capacitors and are switched periodically during the circuit operation changing the effective function of capacitors as typically exploited in switched capacitor (S/C) circuits. Both static and dynamic switches are electronically
controlled and thus the functionality of each CAB, the capacitor sizes, and
the interconnections between CABs are programmable. As a result many
diverse circuit architectures can be implemented.

The FPTA has been described as follows [12]

The FPTA chip architecture consists of an 8x8 matrix of re-configurable cells. The
chip can receive 96 analog/digital inputs and provide 64 analog/digital
outputs. Each cell is programmed through a 16 bits data bus/9 bits address
bus control logic, which provides an addressing mechanism to download
the bit-string of each cell. A total of 5000 bits is used to program the whole
chip. An array of 16x8 photodetectors, distributed within the cells, is also
integrated on chip. It is the first chip integrating reconfigurable processing
circuitry with sensing. The FPTA-2 cell consists of 14 transistors con-
nected through 44 switches and it is able to map different building blocks
for analog processing, such as two and three stages OpAmps, logarithmic
photodetectors, or Gaussian computational circuits.

2.5 Artificial Neural Networks

Artificial Neural Networks (ANNs) are networks of interconnected artificial neurons. The
artificial neurons are computation models inspired by biological neurons. Different types
of neural networks exist in the literature [44]. ANNs can be broadly classified into stateless
and stateful neural networks. In stateless neural networks, the activations of the output
neurons are static functions of their inputs. These kinds of networks are appropriate for tasks like pattern recognition [44]. The stateful neural networks allow feedback loops, meaning that the neuron’s output behavior is also a function of its previous state. They are more attractive for real world control tasks that are causal. Among these neural networks, three sub-categories exist - discrete, discrete-continuous and continuous. Discrete models are those in which the neuron activation as well as the time step in which it increments, are discrete or integer values. In continuous models the activation and increment times can be real. Discrete-continuous are a combination of both the discrete and continuous models. A popular continuous neural network is the Hopfield’s neural network [33]. In this work, we will consider a variant of Hopfield’s neural network. The reasons and motivation for this choice will be made clear in the subsequent sections.

2.6 CTRNNs

Continuous Time Recurrent Neural Networks (CTRNNs) are models of Hopfield’s continuous neurons [33] with unconstrained weight matrices. Each CTRNN neuron state can be described by the following mathematical equation

$$\tau \frac{dy_i}{dt} = -y_i + \sum_{j=1}^{N} w_{ji} \sigma(y_j + \theta_j)$$

where $y_i$ is the state of each neuron, $\tau$ is its time constant, $w_{ji}$ is the strength of the connection from $j^{th}$ neuron to $i^{th}$ neuron, $\theta$ is a bias term and $\sigma(x) = 1/(1 + e^{-x})$ is the standard logistic activation function.
CTRNNs have been proven to be capable of approximating any smooth dynamics in principle, when provided with a sufficient number of neurons [14]. This property of CTRNNs makes it most attractive in the context of this work. Theoritically, CTRNNs can learn any necessary control law related to a real world problem when they are provided with sufficient neurons. While it is not possible to predict the exact, required number of neurons mathematically, it has been observed that even for relatively small number of neurons, interesting dynamical laws are possible. Various researchers have employed CTRNNs for control applications in different engineering domains [5, 15, 16, 51, 20]. In these works, four neuron networks were demonstrated to be necessary and sufficient for addressing the control problems at hand. Hence, for similar applications, it might not be incorrect to infer that, it is essential to have atleast four neuron networks, for effectively learning a desired control law.

### 2.6.1 Architecture of the CTRNN

Figure 2.2 describes the architecture and the parameters of a CTRNN. The figure shows a five neuron fully interconnected CTRNN capable of receiving a sensory input. Each neuron of the neural network has 8 parameters (i.e., 5 weights, 1 time constant, 1 bias and 1 external sensory weight). Each neuron consists of three computational stages. The first stage is the summer, where all the weighted inputs are added. For making this summation temporal, a low pass filter is provided at the second stage. The output of the low pass filter is the state of the neuron. The final stage of the computation is the sigmoid unit that takes as input the state (with a bias applied to it) and provides an output that lies between 0 and
Figure 2.2: An Overview of CTRNN Operation
1. This neural output indicates the strength of the firing provided by the particular neuron; 0 means no firing and 1 means maximum firing.

2.6.2 Configuring a CTRNN

Appropriate configurations for a CTRNN are the set of parameters that make the CTRNN perform the desired function. For instance, if a CTRNN is being used to suppress vibrations, a correct configuration will be the parameters of the CTRNN necessary for it to suppress the vibrations. There are many ways to reconfigure a CTRNN. In this work, we choose the Evolutionary Algorithm as the reconfiguring engine for the CTRNN. The reason for this choice is that it is possible to configure a CTRNN with an EA by specifying only an objective function. There are many instances of real world problems for which one knows whether the system is under control but does not know how to design a controller. One can specify the objective function as merely the state of the system (controlled or uncontrolled), and the EA can evolve a configuration for the CTRNN that depends on just the observed state. In the instances where a different learning engine is employed, it can be very difficult to configure the CTRNN depending on only the observed state of the controlled system.

2.6.3 Modes of operation

There are two modes of evolving CTRNNs:

- Extrinsic Evolution: Functional configurations for the CTRNN are evolved in simu-
lation. Later, the CTRNN device is configured to resemble the evolved configuration, and used for the desired application.

- **Intrinsic Evolution**: CTRNN configurations are evolved for a particular application, while in operation. For instance, if a CTRNN is used to actively suppress oscillations inside a device, then the CTRNN configuration that can suppress the oscillations are evolved while the oscillating device is active or operating. In extrinsic evolution, the conditions are simulated, and CTRNNs are evolved under these simulated conditions.

Figure 2.3 illustrates the configuration procedure of a CTRNN, in a step by step fashion. Usually, a subset of neurons in the network serve as output neurons. All the neurons receive the sensory input for their processing. In this figure, neuron one serves as the output neuron of a CTRNN that is being evolved to generate rhythmic motor signals.

## 2.7 CTRNNs and Evolvable Hardware

The reconfigurable hardware of an EH device provides the control signals to a system when the device acts as a closed loop feedback controller. This being the case, the efficacy of the EH controller depends to a great extent on the “controlling capabilities” of the reconfigurable hardware. Because CTRNNs have been demonstrated to be capable of learning control laws, they are a reasonable choice for being the reconfigurable hardware in the EH control devices. Hence, in this work CTRNNs will be used as the reconfigurable hardware for the EH device being designed. Appropriately, this device will be referred to as a CTRNN-EH device in the rest of this document.
An Illustration of how the CTRNN is trained by an EA to perform the desired function.

An Evolutionary algorithm is used to evolve the parameters of the CTRNN. The performance of the CTRNN when it is configured with the evolved parameters is monitored and new configurations are evolved till the desired performance is achieved. The above illustration shows the process of evolving a CTRNN to generate a rhythmic motor pattern.

Figure 2.3: An Overview of CTRNN configuration procedure
<table>
<thead>
<tr>
<th>Property of CTRNN</th>
<th>Desired property of EH control device</th>
<th>Goal Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can approximate a wide variety of smooth dynamics</td>
<td>Should learn to be an appropriate controller</td>
<td>YES</td>
</tr>
<tr>
<td>Can be reconfigured by the on-board EA</td>
<td>Should adapt to changes in the environment</td>
<td>YES</td>
</tr>
<tr>
<td>Amenable to post-evolution analysis</td>
<td>Should be possible to successfully reverse engineer its principle of operation</td>
<td>YES</td>
</tr>
</tbody>
</table>

Table 2.1: CTRNN will help the proposed EH device achieve its desired functionality

### 2.7.1 CTRNNs in Hardware

As is the case with the EA, it is necessary to build the CTRNN in hardware. While different choices of hardware exist [18, 35, 7], a Very Large Scale Integration (VLSI) implementation is desired. The CTRNNs are amenable to analog VLSI implementations [35, 7, 8] in that they can be implemented in small silicon area. As mentioned previously, if a compact hardware implementation of an EA is realized, the CTRNN-EH device can be made compact, which will enhance its applicability to many real world control problems with stringent area and power requirements.

### 2.8 Existing Analog VLSI CTRNN Implementations

While there are numerous hardware neural network implementations in the literature, there are only three Continuous Time Recurrent Neural Networks (in the form mentioned in
this document) in the literature. Unfortunately, none of the three existing CTRNN implementations, in their current form, are suitable for the desired control applications. A brief summary of the existing CTRNN implementations, with their respective advantages and disadvantages, is discussed in the subsequent sections of this chapter.

2.8.1 Work of Gert Cauwenbergs

This work implements and characterizes an analog recurrent neural network chip incorporating local provisions for online learning of continuous dynamics. A stochastic perturbative algorithm for error descent learning was proposed. Using this algorithm it was possible to obtain a modular and scalable VLSI architecture for learning functions. It was also possible to retain neural parameters for longer durations by the means of a refresh circuitry. This device was used to train oscillatory behavior by means of error descent in parameter space.

Brief Analysis

This work appears to be the first attempt to incorporate a learning engine with a VLSI recurrent neural network on a single chip. The major drawback seems to be that the time constants are fixed. The time constants governs the dynamics of the network, providing first order low pass filtering in the evolution of the neuron state variables [4]. This means that the network dynamics are already restricted and hence, in my opinion, will not find much use in real world control applications. This implementation also makes use of refresh circuitry for retaining the network parameters for a longer period of time. Using refresh circuitry, in general, is considered a tradeoff between the learning accuracy and the increased power
dissipation in the required refresh circuitry. This network was trained for oscillatory behavior. The limitation observed for this test was the requirement of periodicity in the input and target signals during the learning process.

### 2.8.2 Work of Bruce E Brown

The primary goal of this work was to design and implement a CTRNN in analog VLSI so that it can be used to generate locomotion patterns for controlling a micro-robot. This work was the first of its kind in terms of designing a CTRNN for control applications. The analog VLSI design for the CTRNN was demonstrated through the design of a 2 x 2 oscillator (2 neuron CTRNN). Additionally, the interfaces of the sensor and actuator circuits were presented.

**Brief Analysis**

Because our interest in implementing a CTRNN VLSI device is derived from its efficacy in various real time control applications, this work is more interesting than the previously mentioned one. This work can be considered as first real CTRNN implementation. The fundamental building blocks were clearly illustrated. The time constants were determined by external capacitors. While some consider this as a drawback, this implementation offers a user the flexibility of determining the response of a CTRNN. Further, for the real world applications with varying dynamics, programmable time constants may be a useful option. This network is useful in terms of understanding the underlying basic building blocks implementing various neural computation stages. This network cannot be employed for real
world control applications in its current form because of its limited number of neurons (2x2). A 2 neuron network can act best as an oscillator and this was the behavior that was demonstrated in this work. Because the goal of this work was to describe the locomotion control characteristics of a micro-robot with a minimum number of synapses, a 2 x 2 neuron that acted as an oscillator was sufficient. Further, because the CTRNN circuitry implemented oscillation, it did not accept any external inputs. So, sufficient results are not present to verify the behavior of the circuit when sensory inputs are presented to it. This also means that this circuit cannot be used as a closed loop controller. Simulation results were presented comparing the analog VLSI implementation with its software counterpart. The negligible inaccuracies observed in the analog VLSI implementation were attributed to inaccuracies in the analog VLSI multiplication.

2.8.3 Work of Ryan J Kier

This goal of this work was to design and implement a pattern generator in analog VLSI. For these purposes, a four neuron fully interconnected CTRNN with no external sensory inputs was designed and implemented in analog VLSI. A novel synapse implementation - Multiplying Digital to Analog Convertor (MDAC) was proposed.

Brief Analysis

This seems to be an extension of Brown’s work on CTRNN. Similar to above, a CTRNN with an oscillatory behavior was designed. This time, a four neuron oscillator has been designed. However, this work has a novel synapse design. The synapse circuitry was
implemented as MDAC, which was a novel approach. Further, the low pass filter was included on chip. A capacitor array was provided for eliminating off chip low pass filter components. However, there was a critical error reported in the design of the low pass filter due to which, if the chip is run for more time, the bias values decay and change the dynamics of the CTRNN.

2.9 Open Issues

The following are the open issues that can be inferred from the discussions staged in this chapter.

1. There are no existing hardware CTRNN implementations (VLSI) that are suitable for real time control applications in their current form.

2. Existing hardware EA implementations in the literature have specific shortcomings making them unattractive for use as the reconfiguring engine in a CTRNN-EH device.
Chapter 3.

A Digital VLSI Evolutionary Engine for
the CTRNN-EH Device

3.1 Introduction

This chapter presents the Evolutionary Algorithm (EA) that is suitable for Silicon implementation. Discussions about the datapath, hardware implementation and the space requirements are presented in detail. The operation of the EA is demonstrated on a simple test problem. Due to budgetary constraints, this device could not be fabricated on Silicon at this time.

3.2 Minipop : EA Component of CTRNN-EH Device

While many hardware EAs exist [19, 46] in literature, not all are suitable for implementing as the digital EA engine for the CTRNN-EH devices. It is because of the constraints on
the area the digital EA engine can consume. Of the existing hardware EA implementations, *CGA family of hardware EAs [21, 54], which are extensions to the compact genetic algorithm [27], come close to being suitable for using as the digital EA engine in the CTRNN-EH device. The *CGA family of algorithms are space-saving hardware EAs as demonstrated in [21] suitable for evolving CTRNN parameters. They realize their space-savings by maintaining a simulated population than a real population. This is done by using a probability vector representing the probability associated with a bit position of a genome being one. Normally, a 16-bit fixed point representation of the probability is chosen. The memory storage required for a *CGA member is the (genome length * fixed-point precision) where as, for the standard population based genetic algorithm, it is (genome length * total members in the population). When the population size becomes large, the memory requirement increases linearly for a canonical genetic algorithm while it increases logarithmically for a *CGA member. More details about this can be found in [27, 21]. The *CGAs for the reasons mentioned above, are attractive members for using as the digital EA engine in the CTRNN-EH device. A recent work in the literature [37] proposed a new algorithm that was suitable for evolving CTRNN parameters - minipop. This algorithm derives its inspiration from the Micro-GA [34] by maintaining a small population. The small population results in significant space savings. As mentioned previously, the *CGA family algorithms offer significant space savings when a large population is being maintained. The work [36, 37] demonstrates that, the minipop algorithm is adept at evolving CTRNN parameters effectively even in noisy environments. This forms the motivation for using minipop as the digital EA component of the CTRNN-EH device. This section will describe the minipop algorithm and in the later sections, space savings realized by using the minipop algorithm.
Figure 3.1 illustrates the standard Minipop algorithm. The search mechanism of the minipop algorithm is propelled by mutation and hypermutation. When each evaluation completes, the best four members of the evaluation form the population for the next evaluation. There is a hypermutation tournament introduced in each evaluation to navigate the algorithm towards the best possible solution in the entire search space. If the fitness
landscape has large areas of flat plateus and the algorithm gets caught in one such plateu, mutation may not be sufficient to steer the algorithm out. This is overcome by the hyper-mutation tournament.

Initially, population size is fixed at N. For all our purposes, we used a value of 4 for N. Using a population size of 4 makes a compact hardware implementation possible without losing the algorithm efficacy. This is demonstrated in a later section. The initial population is generated at random (lines 2-5). Fitness scores of all the members of the population is evaluated and stored. Later, all the members of the population are mutated individually and their fitness scores are evaluated (lines 9-10). If the mutant’s fitness is better than its parent’s fitness, it replaces the parent in the population for the next evaluation (lines 11-14). After this step, hypermutation tournament is conducted. A completely random individual is generated and evaluated against the worst member in the population. If this hypermutant is better than the worst member in the current population, it replaces the worst member (lines 16-23). The best individual of the final four is returned as the best solution till the current evaluation. The process repeats until the number of evaluations reaches maximum value.

### 3.3 Translating into hardware: Datapath

Figure 3.2 shows a datapath for the Minipop algorithm with population size of 4. A population size of 4 has been previously demonstrated to be sufficient for a variety of CTRNN benchmarks [37]. The proposed Minipop architecture contains 4 random number generators for generating population of candidate solutions, 1 random number generator for generating hypermutant, multiplexors for routing the proper data bits, registers to store the
population, mutant and their respective scores, two custom logic blocks for incorporating the tournament results, best and worst candidates. The datapath operates in the following manner:

Figure 3.2: Datapath of the Minipop

1. Initially, four 8-bit individuals are generated in parallel by the random number generators (RNGs). These individuals are routed to the 32-bit bus via the two to one
multiplexors. The select line of the multiplexor is initially set to route the random individuals to the bus. Once the routing has been completed, it is reset so that, it routes the individuals from the genome bus from the subsequent evaluations. A random individual for the hypermutation tournament is also generated in parallel. This individual is not a part of the four individuals generated and is sent to the fitness evaluation module directly for computing its score.

2. The four 8-bit individuals generated in step 1 are loaded into the population register and mutation module. The mutation module is a customized logic module that performs bitwise mutation. A dedicated random number generator is present for performing mutation. This RNG generates one random number on each rising clock edge. This is compared with the current mutation rate and mutation is performed accordingly on each bit. Hence, it takes 32 rising clock edges to finish mutating all the four individuals. All the random number generators have been implemented using linear feedback shift.

3. After the mutation has been performed, the 32 bit mutated string is loaded into the mutation register. Now, the population register and mutation registers contain the population and mutated population respectively. It is essential to store both these information as Minipop algorithm compares each individual with its mutated version and propagates the fitter one to the next evaluation. Because there are four individuals, four tournaments have to be conducted ideally. But since we chose to represent all the four 8-bit individuals as one 32-bit string, one tournament between the 32-bit population string and the 32-bit mutant string would suffice.
4. The register contents are routed to an external fitness evaluation (FEV) module. This FEV module is located off-chip and it returns a 32-bit string comprising of eight 4-bit scores corresponding of each of the eight evaluated individuals. These scores are loaded into the allotted score registers. After the FEV evaluates the presented individuals, the random individual generated for the hypermutation tournament is also evaluated and stored in a score register.

5. At this juncture, all the individuals including the hypermutant have been evaluated and are ready for subsequent processing. The 32-bit string representing the scores of the 8 individuals (hypermutant, excluded) is sent to the comparison logic block. This block is a customized logic block that takes a 32-bit score string as its input and outputs a 12-bit string encoding the entire information about the tournament results, best and worst candidates. As an illustration, consider a scenario where 1st and 3rd original strings are better and 2nd and 4th mutated strings are better. The first four bits of the 12-bit output will be 0101 representing the fact that, in the second and fourth tournament, mutated version was the winner. Out of these four, if the candidate corresponding to the first bit position is the best and the candidate corresponding to the 3rd bit position is the worst, the complete output of the logic block will be 0101-0010-1000. This representation is very convenient form of storing the complete information and logic for the required information storage can be minimized. In addition to this, 12-bit output, this logic block also gives the 4-bit score of the worst individual for the next stage.

6. After the scores have been evaluated and the four better individuals determined, they
have to be ranked according to their fitness. This ranking is necessary to determine the champion and the worst member in the population. After the worst member has been determined, hypermutation tournament has to be conducted where the worst member of the population is evaluated against a random bitstring. The random bitstring replaces the worst member if it has a better fitness score than the worst member. The reorder logic block does the two operations - ranking and hypermutation. This custom logic block takes 32-bit original population, 32-bit mutated population, 4-bit loser score, 4-bit random hypermutant score, 8-bit random hypermutant individual, and the 12-bit cmplogic output as its inputs. It reorders the final four members of the population based on the information stored in the cmplogic output. After the first stage of reorder, it conducts hypermutation tournament and replaces the worst individual with the random bitstring if the random bitstring is better. Considering the same illustration used above, the first stage of reorder will yield a 32-bit string of \(<i1><m2><m4><i3>\). \(i1\) corresponds to individual 1, \(m2\) corresponds to mutated individual 2, \(m4\) corresponds to mutated individual 4 and \(i3\) corresponds to individual 3. This means that individual 1 is the best individual and is the current champion. Individual 3 has the worst score among the four. Hypermutation is conducted and if the hypermutant score is better than score of individual 3, the random bitstring takes the slot of individual 3. The final output in this case would be \(<i1><m2><m4><\text{random}>\). 

7. The first 8 bits of the reorder logic output correspond to the current champion.
3.4 Intricacies in evolving CTRNN configurations

The datapath proposed in the previous section is suitable for evolving 8 bit genomes. Our goal is to evolve the parameters for upto eight neuron fully interconnected CTRNN. Each neuron of a N-neuron fully interconnected CTRNN has 1 time constant, 1 bias, N weight connections including the self connection and 1 external sensory input. The table 1 indicates the various genome lengths needed to encode all the parameters of the CTRNN. It can be seen that, the genome length increases significantly even when the number of neurons are increased by 1. It is not practical to increase the hardware units to comply with the increase in number of bits. So an obvious solution seems to be to use a memory module for storing the neural parameters. If a memory module exists for storing the neural parameters, the same datapath presented above can be used for evolving the parameters of a CTRNN with the desired number of neurons. The datapath operation will be however, split into two because of the multiplexing involved.

1. In the first stage of the datapath operation “all” the parameters of the CTRNN which, in other words, the complete genome, is generated, and loaded into the memory. FEV
reads the genomes from the memory, evaluates them and writes the score back to the memory.

2. In the second stage, the parameters and the fitness scores stored in the memory are used for tournaments, ranking and reordering.

Memory Requirements

Minipop algorithm necessitates storage of 9 genomes (4 individuals, 4 mutated individuals and 1 hypermutant) at a single time. The memory requirement increases by a significant factor when one moves towards evolving 704 bit genomes for a CTRNN with 8 neurons. A maximum of 1K byte memory is necessary when one intends to store nine 704-bit genomes (792 bytes), programming mode details and other necessary evolutionary details. It is worthwhile to note that the *CGA algorithm with 16-bit probability precision requires about 1.4K just to store one genome information. To store atleast 2 genomes for conducting tournaments, it requires about 3K memory. This is the best case memory requirement when a *CGA algorithm is used as the digital EA component of the CTRNN-EH device.

3.4.1 The Memory Interface

Fig 3.3 is the logic diagram of the minipop datapath with an on-board memory of 1K bytes. This logic diagram should be understood in the following manner:

This datapath is exactly similar to the one shown for minipop algorithm except for two
changes - addition of control module and a 1K byte memory. A static memory map is used for storing the neural parameters, fitness scores and evolutionary details in the memory. For instance, for an eight neuron system, first $88 \times 4 = 352$ bytes correspond to the initial four members of population, second 352 bytes correspond to the mutated version of individuals followed by 88 bytes representing the hypermutant. The error scores and the other evolutionary details like tournament information, generations, generations taken for the first best solution, etc., are stored in the subsequent memory locations. The memory is byte accessible and has 10 address lines. It has bidirectional data lines (eight) and the mode of operation (write or read) is selected by the R/W pin being set high or low. The control module raises the $\text{req\_eval}$ line when all the neural parameters are loaded. The external FEV, loads the bits from the static memory locations in a serial fashion, evaluates them and
writes back the error scores to the assigned memory locations. It raises the `eval_done` line and the control module gives the appropriate signal to the cmplogic module to start its function. Till the fitness scores are evaluated, the second part of the logic diagram stays inactive (cmplogic and reorder blocks). After the reorder gives the ranked outputs, the cycle continues.

The figure 3.4 illustrates a possible microcontroller flow diagram for this integrated architecture.

![Flow Diagram](image)

Figure 3.4: A flow diagram of integrated architecture
3.5 Results: Simulation, Synthesis and Analysis

In this section, the working of the minipop algorithm is demonstrated followed by the synthesis results and the analysis of the observed results. The complete algorithm has been modeled in Verilog HDL and simulated in Cadence Simvision. The figure 3.5 gives a snapshot of the simulation results of the minipop algorithm. The clock speed for the simulation was set to 20 MHz. The algorithm was tested for various mutation rates ranging from 0 to 100. The simulation shown in the figure has a mutation rate of 25%. The simulation window shows the output at each stage of the datapath - population and mutation registers, cmplogic and reorder logic. The champion at the end of each evaluation is also shown. As it can be seen, for the simulation shown in the figure, the best solution is found in 0.0135 ms. This was just a test problem used to verify the operation of the hardware version of this algorithm. Rigorous testing is underway where parameters for networks as large as 8 neurons are being evolved.

Figure 3.5: Simulations of One Max Problem using Minipop
Synthesis Results and Analysis

The synthesis results for the minipop algorithm for the one max problem (genome length 8) will be first discussed. A discussion about the chip area this design will consume will be provided. Following that, an observation about the increase in chip area when memory and control logic are added, will be provided. An overall estimate of the chip in the presence of digital EA engine and a CTRNN is provided.

<table>
<thead>
<tr>
<th>Module</th>
<th>Wireload</th>
<th>Cell Area</th>
<th>Total Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>NONE</td>
<td>420838.56</td>
<td>420838.56</td>
</tr>
<tr>
<td>cmplogic</td>
<td>NONE</td>
<td>33953.76</td>
<td>33953.76</td>
</tr>
<tr>
<td>eightBitReg_0</td>
<td>NONE</td>
<td>10193.76</td>
<td>10193.76</td>
</tr>
<tr>
<td>eightBitReg_1</td>
<td>NONE</td>
<td>10193.76</td>
<td>10193.76</td>
</tr>
<tr>
<td>eightBitReg_2</td>
<td>NONE</td>
<td>10193.76</td>
<td>10193.76</td>
</tr>
<tr>
<td>eightBitReg_3</td>
<td>NONE</td>
<td>10193.76</td>
<td>10193.76</td>
</tr>
<tr>
<td>eightBitReg_4</td>
<td>NONE</td>
<td>10193.76</td>
<td>10193.76</td>
</tr>
<tr>
<td>eightBitReg_5</td>
<td>NONE</td>
<td>10193.76</td>
<td>10193.76</td>
</tr>
<tr>
<td>eightBitReg_6</td>
<td>NONE</td>
<td>10193.76</td>
<td>10193.76</td>
</tr>
<tr>
<td>eightBitReg_7</td>
<td>NONE</td>
<td>10193.76</td>
<td>10193.76</td>
</tr>
<tr>
<td>lfsr_0</td>
<td>NONE</td>
<td>19476.00</td>
<td>19476.00</td>
</tr>
<tr>
<td>lfsr_1</td>
<td>NONE</td>
<td>19476.00</td>
<td>19476.00</td>
</tr>
<tr>
<td>lfsr_2</td>
<td>NONE</td>
<td>19476.00</td>
<td>19476.00</td>
</tr>
<tr>
<td>lfsr_3</td>
<td>NONE</td>
<td>19476.00</td>
<td>19476.00</td>
</tr>
<tr>
<td>lfsr_4</td>
<td>NONE</td>
<td>19476.00</td>
<td>19476.00</td>
</tr>
<tr>
<td>lfsr_5</td>
<td>NONE</td>
<td>19476.00</td>
<td>19476.00</td>
</tr>
<tr>
<td>mutation</td>
<td>NONE</td>
<td>48944.16</td>
<td>48944.16</td>
</tr>
<tr>
<td>mux_0</td>
<td>NONE</td>
<td>2741.76</td>
<td>2741.76</td>
</tr>
<tr>
<td>mux_1</td>
<td>NONE</td>
<td>2741.76</td>
<td>2741.76</td>
</tr>
<tr>
<td>mux_2</td>
<td>NONE</td>
<td>2741.76</td>
<td>2741.76</td>
</tr>
<tr>
<td>mux_3</td>
<td>NONE</td>
<td>2741.76</td>
<td>2741.76</td>
</tr>
<tr>
<td>register_0</td>
<td>NONE</td>
<td>38196.00</td>
<td>38196.00</td>
</tr>
<tr>
<td>register_1</td>
<td>NONE</td>
<td>38196.00</td>
<td>38196.00</td>
</tr>
<tr>
<td>reorder</td>
<td>NONE</td>
<td>51671.52</td>
<td>51671.52</td>
</tr>
<tr>
<td>AWMUX_32_1</td>
<td>NONE</td>
<td>8172.00</td>
<td>8172.00</td>
</tr>
</tbody>
</table>

Table 3.2: The synthesis results of the minipop algorithm as used for one max problem
The table 2 shows the area report when the minipop algorithm is synthesized for the one max problem. This is the exact datapath indicated in figure 3.2. On-board memory was not required for this problem as only 8 bit genomes were evolved. All the individual modules were implemented in Verilog HDL and optimized for area. Earlier, we mentioned the use of information encoded logic for minimizing the area requirements of the chip. A look at the area consumed by the cmplogic module and reorder logic module would justify this claim. The cmplogic module is the most complicated module in the design and it consumes just 8% of the total design area. The reorder logic module which uses the information provided by the cmplogic module and the information present in various registers of the system, consumes about 12% of the design area. Further, they help in reducing the complete area of the digital EA engine and thus in process, the CTRNN-EH device.

<table>
<thead>
<tr>
<th>Memory (Kbytes)</th>
<th>Digital Logic Area (mm²)</th>
<th>Memory Area (mm²)</th>
<th>Total Chip Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.4208</td>
<td>0</td>
<td>0.4208</td>
</tr>
<tr>
<td>0.25</td>
<td>0.4472</td>
<td>1.08</td>
<td>1.5272</td>
</tr>
<tr>
<td>0.50</td>
<td>0.4472</td>
<td>2.20</td>
<td>2.6472</td>
</tr>
<tr>
<td>0.75</td>
<td>0.4472</td>
<td>3.40</td>
<td>3.8472</td>
</tr>
<tr>
<td>1.0</td>
<td>0.4472</td>
<td>4.53</td>
<td>4.9772</td>
</tr>
</tbody>
</table>

Table 3.3: Table indicating the chip area consumed by digital EA engine

Table 3 indicates the increase in the area of the chip when the control logic and memory modules are included in the design. These values were obtained by synthesizing the design for different memory sizes. Cadence Physically Knowledgable Synthesis tool was used to synthesize the design. As noted from the table, a presence of 1K memory which is
sufficiently large enough to store the information for evolving configurations for an eight neuron fully interconnected CTRNN, consumes about 5 $mm^2$ of Silicon area.

### 3.6 Conclusions

In this chapter, a design for the digital part of the CTRNN-EH control device was presented. The functionality of the EA was shown on a simple test problem. The intent when designing this algorithm was to fabricate this design along with the analog CTRNN design. However, due to budgetary constraints, the EA was not fabricated.
Chapter 4.

VLSI Implementation of the first generation 2x2 CTRNN

This chapter will present a circuit implementation of the first generation 2x2 CTRNN device. A detailed discussion of concept to circuit architecture will be presented. Simulation results will be shown for justifying design decisions. Detailed results from the silicon sample will be provided in a later chapter dedicated to results.

4.1 Block level architecture of a 2x2 CTRNN

The previous chapter presented a mathematical equation for the CTRNN of the form

\[ \tau \frac{dy_i}{dt} = -y_i + \sum_{j=1}^{N} w_{ji} \sigma(y_i + \theta_i) + I \]  (4.1)

where \( y_i \) is the state of each neuron, \( \tau \) is its time constant, \( w_{ji} \) is the strength of the connection from \( j^{th} \) neuron to \( i^{th} \) neuron, \( \theta \) is a bias term, \( I \) is the sensory input, and
\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

is the standard logistic activation function. It is possible to derive a block level implementation from the above equation. It can be deduced from the equation that a summation stage and an integration stage are required. There is a weighed multiplication of the neuron outputs that are presented to the summer. Because the change in the state of a neuron depends on the current neuron stage, the integration is leaky. This equation can be thus represented easily with the block diagram shown in Figure 4.1.

The output of the neuron \( \sigma(y + \theta) \) is fed back to the first stage. The first stage, which can be considered as the input stage implements the weighted summation of inputs. The second stage is the integrator stage. As mentioned above, the feedback path of \( y_i \) implements the leaky integration. The sensory input I is also added at this junction. The output of the leaky integrator is passed to the final stage implementing the neuron activation function.
4.2 Design decisions

4.2.1 Budgetary

The first generation CTRNN silicon implementation had two main purposes. First, evaluating the feasibility of an analog VLSI implementation of a programmable CTRNN, and second, understanding the effects of various factors like noise and external data acquisition interfaces on the device. Keeping these factors in mind, it was desired to tapeout more than one version of the CTRNN device. The first device would be used to derive knowledge about the correct design and implementation and the second device would be the final prototype. Further, if the first device were to arrive fully functional from the foundry, a digital core could be added to the second device for autonomously programming the CTRNN. Hence the technology decisions were governed, at least in part, by budget allowances. MOSIS fabricates tiny sized commercial chips for 6500 dollars in AMI 0.6\(\mu\) technology. This includes the packaging costs. The overall budget available was 17000 dollars, meaning two generations could be fabricated. Remaining money was allocated to PCB etching and other test setup.

4.2.2 Circuit Implementation

The various sub-blocks in a neuron architecture can be implemented in subthreshold analog VLSI. Multipliers, integrators and summers can be built using relatively few transistors. Also, subthreshold VLSI offers the advantages of low power consumption and small size. However, there are disadvantages like susceptibility to noise and device mismatch, which
would necessitate careful floorplanning, design and layouts.

### 4.3 Circuit implementation

This section will detail each block in the neuron architecture and present its analog VLSI implementation.

#### 4.3.1 Synapse and Summer

The first stage of the design is a synapse. The synapse takes an input and multiplies it with a stored weight. This weight is normally an interconnection weight (presented by $w_{ij}$ in equation 4.1). The synapse output goes to a summing stage along with outputs from other synapses in the network. If the output quantities are current, then the summer can be simply implemented as a junction using Kirchoff’s current law. In the block diagram shown the summer is highlighted. The synapse is embedded within the summer. Here, I will discuss the synapse primarily as it is challenging to implement. The summer is implemented directly by combining the output junctions of the synapses. In the first generation CTRNN device, the synapse has been implemented by a simple two tailed differential pair.

In the subthreshold operating region, this simple three transistor circuit allows one to implement an analog multiplier. This design was effectively used by Brown [7]. Also, since this circuit is effectively an operational transconductance amplifier (OTA), it provides a current output while allowing us to define our input parameter weight as a voltage. This scheme lets us realize a summer directly, as mentioned above. The transfer characteristics
of this circuit are given by the following equation

\[ I_1 - I_2 = I_b \tanh \left( \frac{(V_w - V_{ref})}{2\eta V_{th}} \right) \]  \hspace{1cm} (4.2)

The transistor N2 is the input transistor. This generates the bias current \( I_b \). The output of the neuron(s), which is current (shown later), is converted to voltage using diode connected load, and is provided as the input to the synapse cell at N2. \( V_w \) is the voltage stored on the gate of transistor N0. This voltage represents the interconnection weight or programming parameter with respect to this cell. It can be seen that the output current, which is a differential value, is equivalent to the current generated by input voltage and a constant term set by the interconnection weight. However, this constant term is a hyperbolic tangent depending on the interconnection weight and thus, non linear. In the normal sense, this
circuit is of limited use as this is not a linear analog multiplier. Hence, to make this circuit useful for the CTRNN implementation, we should only consider its operation in its linear region. For proper functioning of the synapse, the transistors N0 and N1 must operate in subthreshold region. An experiment was conducted to determine the range of input current that will allow the operation of these transistors in the desired region. An upper limit of 320nA was determined. To be in safe operating regions, the maximum current was fixed to be 200nA. This was the upper limit set to ensure that synapse was acting as a linear multiplier.

Figure 4.3 illustrates the test setup used to test the synapse behavior. An input current was sourced into the transistor N2 and the output current was observed. This output current was converted to a single ended voltage value by passing over a resistor. Simulations were performed to check the linearity of the synapse by sweeping the input current from 0 to its maximum value of 200nA. Also, the weight value stored on the gate of transistor N0 was varied to see the effect on the slope of the output voltage. Figure 9.3 shows the simulation result for this experiment. It can be seen that the synapse behaves entirely linearly over the operating range. The variation of the weight value results in the change of slope of the output voltage. The mapping between the analog voltage value on the gate of N0 is given by

\[ V_w = 2\eta V_{th} \tanh^{-1} \left( \frac{w_{ij}}{w_{max}} \right) \quad (4.3) \]

where \( V_w \) is the voltage on gate of N0, \( V_{th} \) is the thermal voltage, \( w_{ij} \) is the raw voltage value between [-15,15] and \( w_{max} \) has a value of 16. It should be noted that while the maximum raw voltage value is 16, the weights are restricted to +/-15. This is because the
inverse hyperbolic tangent of $1/-1$ is not defined. $\eta$ is the non ideality factor and is process dependent. A value of 1.5 is assumed for the simulations.

The layout of the synapse is shown in Figure 9.4. The input transistor’s aspect ratio is 6/120. This is broken into three fingers for better matching. The transistors N0 and N1 have sizes of 24/3 respectively.

### 4.4 Sigmoidal Activation Function

The activation function of the CTRNN neuron is a tanh sigmoid, as was discussed in chapter 2. The implementation of the sigmoid circuit is identical to the synapse circuitry. While we were concerned about the linear operating range of the synapse circuitry, for the sigmoid circuit there is no such restriction. However, the primary difference between the sigmoid circuitry and the synapse circuitry is that the input nodes change. While transistor N0 stored a constant weight in synapse circuit, here it takes an input current through the integrator. The transistor N2 which was the input transistor in synapse acts as a constant current source. The gate voltage for this transistor is provided by on-chip bias circuitry. While the transistor N0 receives the input from the synapse and integrator, the transistor N1 gets the bias value of the neuron as its gate voltage. The purpose of the bias voltage $\theta$ is to vary the orientation of the tanh curve along the x-axis. The test setup for a sigmoid simulation is presented in Figure 9.6. Figure 9.7 presents the simulation results when the value of $\theta$ is changed while sweeping along the input voltage of N0. The x-axis is a differential voltage between gates of transistors N0 and N1.
The conversion for $\theta$ in terms of the analog gate voltage of transistor N1 is given by

$$V_{\theta} = 2\eta\theta \quad (4.4)$$

It can be seen from the results that when $\theta$ value is varied, the sigmoid shifts along the x-axis. The layout for the sigmoid cell is shown in Figure 4.10. This layout, as one can expect, is similar to the layout of the synapse cell.

### 4.5 Circuit diagram of a CTRNN row

The circuit diagram of one row in a 2x2 CTRNN architecture is presented in Figure 4.9. The synapse and sigmoid circuit have been discussed in the previous sections. There is some equally important support circuitry in addition to the synapse and sigmoid. Of these, the first is D2S (differential to single ended) converter. The purpose of this circuit is to convert the differential output of the synapse to a single ended value. The two transistors N7 and N8 act as cascode devices that help maintain the transistors N0 and N1 of the synapse at the same potential. The integrator is an external RC circuit. While the final aim is to have an onchip integrator, a decision was made to leave it outside the chip in the present work, in the interest of reducing area and complexity.

The positive output of the sigmoid circuit is fed back to the input stage and the negative output is routed to the ground. This can be seen from the two current mirrors at the output stage. The layouts for the D2S and the current mirror are shown in Figure 4.10.
Figure 4.3: Synapse test circuit
Figure 4.4: Synapse simulation results
Figure 4.5: Synapse layout
Figure 4.6: Circuit diagram of a sigmoid
Figure 4.7: Sigmoid test circuit
Figure 4.8: Sigmoid simulation results
Figure 4.9: Complete circuit diagram of a single row in a CTRNN
Figure 4.10: (a) Sigmoid Layout    (b) D2S layout    (c) PMOS mirror layout
Chapter 5.

A Memory Cell for Storing Programming Parameters in the Generation One CTRNN

In the previous chapter, we saw the circuit implementations of the individual building blocks of a CTRNN. The programming parameters, including interconnection weights $w_{ij}$ and neuron biases $\theta$ were shown as gate voltages at the synapse and sigmoid cells, respectively. In the test circuits that were shown, these parameters were programmed by ideal DC voltage sources. However in reality this is not the case. Programming the CTRNN through external voltage sources is not a feasible option. This would necessitate a significant increase in pin count (depending on the neuron architecture). Also this option is not elegant to an analog circuit designer. The background chapter presents several on-chip analog memory schemes from the literature. For this work, a capacitive memory scheme
similar to the one presented used by Cauwenberg [8] was implemented.

## 5.1 Modifying Synapse and Sigmoid Cells

The programming parameters are stored as voltages on capacitors. For storing the interconnection weights, the synapse cells were modified to include capacitors for storing programming weights. This can be seen from Figure 5.1 (a). Figure 5.1 (b) provides an extension to the basic memory cell. An AND gate and a switch transistor are found in addition to the capacitor. The purpose of the switch transistor is straightforward. When the switch is ON, the capacitor will be programmed to the voltage value found on the net $V_{w}$. When the switch is turned off, this value will be stored in the capacitor as charge. The purpose of the AND gate can be understood from a look at Figure 5.2, which shows the four synapses connected to a common data line. All the programming parameters (only four are

![Figure 5.1: (a) Capacitor to store memory (b) Switch transistor](image)
Figure 5.2: Four memory cells connected in an array-like fashion

shown here) are connected in an array-like structure. This allows a modular programming approach. All the cells are connected by the data line. Depending on the address bits, only one memory cell will be switched on. The capacitor on that cell will be programmed to the value on the data or voltage line. This explains the purpose of the AND gate, which maintains the ON/OFF state of a memory cell.
5.2 Charge Injection and Capacitive Feedthrough

Capacitive memory storage is easy to implement but has several disadvantages, including charge injection, capacitive feed through and leakage. Leakage is a common phenomena and requires periodic refresh of stored values. In this work, an external refresh circuitry is implemented and will be discussed in detail in a later chapter.

Figure 5.3: (a) Demonstration of charge injection (b) Demonstration of capacitive Feedthrough
5.2.1 Charge Injection

Charge injection can be understood from Figure 5.3 (a). When the MOSFET switch is on and $V_{DS}$ is small, a charge results from under the gate oxide due to the inverted channel. When the MOSFET turns off, the charge is injected in the capacitor and into $V_{in}$. Since, $V_{in}$ is assumed to be a low impedance source driven node, the injected charge has no effect on it. However, the charge injected into CLOAD changes the voltage stored in it.

5.2.2 Capacitive Feedthrough

Figure 5.3 (b) shows the capacitances between the gate/drain and gate/source of the MOSFET when it operates in the triode region (true in the MOSFET switch). When the gate clock signal $\phi$ turns high, the clock signal feeds through the gate/drain and gate/source capacitances. As the switch is turned on, the input signal, $v_{in}$ is connected to the load capacitor via the switch. There is no effect of capacitive feedthrough here and CLOAD is charged to $v_{in}$. However, when the clock goes low or NMOS turns off, a capacitive voltage divider exists between the source and load capacitance. As a result, a portion of the clock signal appears across CLOAD. This is the effect of clock feedthrough.

5.2.3 Reducing Capacitive Feedthrough and Charge Injection

Many methods exist in the literature to reduce the effects of charge injection and clock feedthrough. Dummy switches are one of the most widely employed approaches [2]. This can be understood from Figure 5.4. Dummy switch is a NMOS transistor with its source
and drain shorted. This is placed in series with the switch transistor. When the switch transistor turns off, half the channel charge is injected toward the dummy switch. Dummy switch turns on with a slight delay inducing a charge equivalent to $\frac{C_{ox}}{2}$. But since, the dummy switch size is half the switch transistor size, the charge induced by the dummy transistor will cancel the charge injected by the switch transistor. Thus, the value on the capacitor is preserved.
5.3 Dummy switches and Differential Setup for Synapse Memory Cell

Figure 5.5 presents the complete memory cell schematic with the dummy cell to cancel capacitive feedthrough and injection effects. The layout of this design is shown in Figure 5.6 (a).

The synapse is a differential pair based design and is sensitive to voltage differences on the gates of N0 and N1. Because of a switching circuit, intermediate voltage spikes might occur that will change the differential voltage momentarily resulting in incorrect behavior. Also, the any unmodeled and unforeseen effects of leakage, charge injection and
feedthrough will result in unpredictable results. This is hard to debug after fabrication. Hence, a differential memory cell approach was carried out. Here, identical memory cell circuitry is created on both the sides of the differential amplifier. The bias voltage on gate of N1 is also stored on a capacitor, which is connected to the same memory cell setup. Any variations of droop will be caught on either side of the differential amplifier ensuring its correct behavior. This setup is indicated in Figure 5.6 (b).
Figure 5.6: (a) Layout of memory cell with dummy switch (b) Differential memory cell
Figure 5.7: Layout of 2x2 CTRNN with memory cell
Chapter 6.

Programming circuitry for the

Generation One CTRNN

6.1 Introduction

In the previous chapter, we saw the designs for memory cell and the 2x2 CTRNN. This chapter continues the discussion of implementation, presenting the design for the programming circuitry needed to digitally program the memory cells to the right neural parameters. This chapter will present discussion on the type of digital to analog converter (DAC) used and some support circuitry to make it work correctly.

6.2 Digital to Analog Converter

The 2x2 CTRNN contains six programmable parameters of which four are interconnection weights $w_{ij}$ and two are neuron bias values, $\theta_i$. These values have to be externally
programmed by an user or autonomously by a learning engine residing either onchip or offchip. For the ease of programming interfaces and scalability, digitally programming the CTRNN is most desirable. However, because the neural parameters are stored as analog values on the capacitive memory cells, a digital to analog converter is required. Several choices of digital to analog converters exist in the literature [2, 26]. In this work, a DAC based on the current steering principle or binary weighted current mirror scheme was employed.

6.2.1 Current steering DAC

A generic current steering DAC is shown in the figure 6.1. This circuit consists of a set of

![Basic current steering DAC](image)

Figure 6.1: Basic current steering DAC
current sources of unit value. The output current $i_{out}$ is generated by the combination of switches set by the digital code. For instance a code of 0000 0000 would route complete current to ground and thus, the output current would be zero (ideally). When the digital input code is binary, a binary weighted current scheme is employed. Here, the unit current sources are replaced by binary weighted values. For instance, in this figure, D0 would contain a current source of value I, D1 would have 2I, D2 would have 4I and so on.

The advantage of the current steering DAC is its high current drive capability. No output buffers are necessary to drive resistive loads and the circuit has a fast operation. However, the accuracy of current steering DAC is dependent on the matching that can be obtained. Binary weighted current mirrors are prone to severe fabrication mismatches and careful layout is necessary. Another problem that is associated with this architecture is the error due to the switching. Since the current sources are in parallel, if one of the current sources is switched off and the other switched on, a transient error could occur. Output will have a momentary spike.

Figure 6.2 shows the current steering DAC employed in this work. It is made up of binary weighted current mirrors. A current is sourced through the transistor P0, which is the input transistor. The size of P0 is $2^8$ times the size of P9. The outputs of the PMOS transistors are connected to a simple switch circuitry, one per transistor composed of minimum length PMOS transistors. The purpose of the switch circuitry is to route the output current of the respective PMOS transistor (DAC transistor) to either the positive or negative net. For instance, if the LSB B0 is high, the current through P9 is routed to I+. A value of 0000 0001 would mean that I+ has a contributing current from P9 where as, I- has contributions
from all the other binary-weighted PMOS transistors.

Figure 6.2: Binary weighted current mirror DAC

The programming parameters (interconnection weights and neuron biases) are voltage parameters. Hence, a current to voltage converter is required. A simple resistor would do this job. However, from Chapter 3, the relation between raw weight value and equivalent analog voltage can be seen by the following equation:

$$V_w = 2\eta V_{th} tanh^{-1}\left(\frac{w_{ij}}{w_{max}}\right)$$

(6.1)

Figure 6.3 shows the relation between the weight and converted voltage. It can be seen that the relation is non-linear as it depends on an inverse hyperbolic tangent function. Hence,
Figure 6.3: Relation between neuron weights and programming voltages

A resistive load for I-V conversion would not suffice. A non-linear load exhibiting a trend similar to one seen in figure 6.3 would be required. The non-linear relationship between $w_{ij}$ and $V_w$ can be attributed to the transfer characteristics of the differential pair synapse. Hence, a similar load to the DAC would convert its output current to the desired voltage level. Figure 6.4 (a) presents one such circuit.

This circuit serves a dual purpose. It provides an input current to the DAC and converts the output current of the DAC to an equivalent programming voltage. It is easy to see that the second part of the circuit, shown in Figure 6.4 (a), mirrors the synapse and sigmoid cell. All the transistor sizes are also the same. The transistor N3 generates an input current of 200nA that is provided to P0. Transistor N4 is included for better matching and for maintaining N2 and N3 in the same potential. Figure 6.4 (b) provides a simulation snapshot of the DAC with reference circuitry. It can be seen from the figure that the output of the
combined circuitry (DAC and IV) is non-linear, similar to the expected trend. Spikes can be seen during transition edges. This is expected to happen in a current steering DAC, as mentioned previously. The layout of the complete DAC with reference/conversion circuitry and switches is shown in Figure 6.5.

The DAC consumes a significant portion of the onchip area due to the large binary weighted PMOS transistor.
Figure 6.4: (a) IV Conversion circuit (b) Simulation snapshot of DAC with IV conversion
Figure 6.5: Layout of the DAC
Chapter 7.

Measured Results on the Generation one 2x2 CTRNN Device

7.1 Introduction

This chapter will present the measured results on the individual neuron and DAC blocks from the first generation CTRNN device. We will start with a snapshot of the padframe indicating various blocks. Following it, all the testable blocks will be considered individually and measured results will be provided. Figure 7.1 provides a snapshot of the padframe layout of generation one CTRNN device. It can be seen that this device has two binary weighted current mirrors of eight-bit resolution each, a 2x2 programmable CTRNN, test circuitry for synapse and sigmoid, bias circuitry for providing on-chip control currents, and voltages and a switched capacitor resistor for testing large on-chip resistor feasibility. The implementation of the S-CR has directly been adopted from Yu’s work[58].
Figure 7.1: Layout of the padframe of generation one CTRNN

7.2 Synapse and Sigmoid test

The pins corresponding to the synapse and sigmoid test are shown in Figure 7.2. The test setup is very similar to the ones shown in chapter 3. Here, a test current is sourced externally into the device and the output voltages are observed. This setup maintains similarity with the cadence simulations done to verify the VLSI designs. The input current is generated using discrete components, LM117 in this case. A picture of the circuit setup of LM117 as current source is given in Figure 7.3
Figure 7.2: Testing the synapse and sigmoid circuits

Figure 7.4 shows the measured results of the synapse circuit. The x-axis is the current sweep in nA and y-axis presents the output current. It can be seen that the synapse behaves linearly as expected from the simulation results. The synapse cell was programmed to different programming weights and the observed output indicates that the synapse design is correct. Similarly, Figure 7.5 shows the measured results of the sigmoid cell. Here too, it is seen that the synapse behaves as expected. Here, the output is indicated in current units and input is a differential voltage similar to the metrics used in simulation.
Figure 7.3: LM117 for current source

7.3 Digital to Analog Converter

In Figure 7.1, two binary weighted current mirror DACs are visible. The purpose of having two DACs was to have one for testing. Having a separate DAC for testing does not necessitate the decoupling of the primary DAC from the CTRNN while testing. Figure 7.6 shows the related pins used for testing the DAC. As seen from the figure, there are five address pins that are used to select a combination of row and column cells from a total of two rows and three columns. The digital code is put on the data pins and the output is observed by a data sweep. Here, a microcontroller was used to sweep the data values from 0 to 255. The results are shown in Figure 7.7, which also presents a head-to-head comparison of measured DAC behavior with expected behavior. The DAC output seen in this figure is devoid of spikes. The spikes were filtered from the DAC using MATLAB. It can be seen
Figure 7.4: Measured results synapse

that the measured DAC behavior closely follows the expected behavior. At the extreme programming values a deviation can be noted. To measure how significant the deviation is, an integral nonlinearity (INL) plot was created. This is shown in Figure 7.8. INL can be calculated by taking the deviations of the DAC output from its ideal desired behavior. For a current steering DAC, it is harder to achieve acceptable values of INL [2]. It can be seen that the worst case INL values of +/-0.8 LSB are observed at the minimum and maximum values of digital input, which corresponds to the weight values at the extreme ends. In the weight ranges between [-10,10], the DAC has an INL of less than 0.5, which is acceptable. The DNL of the DAC is not shown. However, from figure 14, we can observe that hardware DAC behavior closely follows the ideal DAC behavior. Slight deviations are noted at the extreme points of the curve. Hence, DNL is not a problem for this DAC. Also, because the
Figure 7.5: Measured results sigmoid

DAC behavior is inherently non linear, DNL cannot be determined directly. For determining the DNL, change in voltage values with respect to digital inputs must be determined from the ideal curve and a comparison with hardware output must be performed.
Figure 7.6: Testing the digital to analog converter
Figure 7.7: Measured results sigmoid

Figure 7.8: Integral Nonlinearity
Chapter 8.

Testing the Generation One CTRNN Device

8.1 Introduction

This Chapter will present the test results on the generation one CTRNN device. Functionality of the device will be verified for extrinsic and intrinsic experiments. The next section presents a description of the test setup.

8.2 Test setup

Figure 8.1 illustrates the conceptual block diagram of the test setup utilized for testing the CTRNN first and second generation devices. The test setup shown is more appropriate to
intrinsic device testing, as was explained in detail in Chapter 2. Simply stated, intrinsic experiments are closed loop experiments where an EA-based learning engine autonomously configures the CTRNN device. During normal device testing in the extrinsic mode, the CTRNN will be manually programmed by the user. Here, the setup remains the same, except that the desktop PC will act as an interface for entering the neuron parameters. It will not run the EA learning engine.

The CTRNN device (generation one or two) is programmed by an 8056 microcontroller. The microcontroller serves the dual purposes of acting as a programmable interface and also a refresh circuit. The neural parameters are stored in the flash memory of the mi-
crocontroller and a code resides on it to periodically refresh the data stored in the memory cells. Ideally, this task will be done by a learning engine on-chip with a static RAM to store the neural parameters or memory map. Due to budgetary constraints, the learning engine could not be fabricated on the same device as a CTRNN. Hence, a microcontroller and desktop PC are being used to perform the tasks that would be performed by an EA.

The 8056 receives the neural parameters from the desktop PC via serial port. Based on the memory map, the CTRNN device is programmed. This memory map is shown in Table 8.1. The output of the CTRNN device is recorded using National Instruments data acquisition cards. The desktop PC also acts as a data collection and processing device. The values from the NI-DAC are recorded in the PC for analysis and post processing.

<table>
<thead>
<tr>
<th>Address</th>
<th>Cell</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101</td>
<td>R1-C1</td>
<td>w11</td>
</tr>
<tr>
<td>00110</td>
<td>R2-C1</td>
<td>w12</td>
</tr>
<tr>
<td>01001</td>
<td>R1-C2</td>
<td>w21</td>
</tr>
<tr>
<td>01001</td>
<td>R2-C2</td>
<td>w22</td>
</tr>
<tr>
<td>10001</td>
<td>R1-C3</td>
<td>$\theta_1$</td>
</tr>
<tr>
<td>10010</td>
<td>R2-C3</td>
<td>$\theta_2$</td>
</tr>
</tbody>
</table>

Table 8.1: Memory map
8.3 Functional Validation of CTRNN

8.3.1 Extrinsic Verification

In extrinsic verification, the CTRNN device is programmed to exhibit some known behavior. Owing to decades of research on CTRNN at Wright State University, Case Western Reserve University, and other locations, a vast library containing thousands of CTRNN configurations is available. In this work, random configurations taken from these libraries were used for functional verifications. In extrinsic verification, the accuracy of the CTRNN device in exhibiting predicted behavior was tested. The time constants of the neurons on the chip were fixed as 1mS via external RC circuits. Table 8.2 presents some neuron oscillator configurations.

<table>
<thead>
<tr>
<th>Config</th>
<th>$w_{11}$</th>
<th>$w_{12}$</th>
<th>$w_{21}$</th>
<th>$w_{22}$</th>
<th>$\theta_1$</th>
<th>$\theta_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.98</td>
<td>10.38</td>
<td>-5.47</td>
<td>9.58</td>
<td>-8.17</td>
<td>-2.11</td>
</tr>
<tr>
<td>2</td>
<td>6.99</td>
<td>-10.62</td>
<td>9.87</td>
<td>7.44</td>
<td>5.57</td>
<td>-6.95</td>
</tr>
<tr>
<td>3</td>
<td>4.19</td>
<td>6.786</td>
<td>-3.85</td>
<td>7.55</td>
<td>3.51</td>
<td>-0.87</td>
</tr>
</tbody>
</table>

Table 8.2: Some CTRNN configurations

Table 8.3 shows the equivalent programming voltages and microcontroller hex codes for loading these configurations into the CTRNN device. The expected behavior of the three configurations is shown in Figure 8.2. When the CTRNN device was programmed using the setup shown in Figure 8.1, the device did not exhibit oscillatory behavior. The following tests were sequentially carried out:
### Table 8.3: Microcontroller programming values for previous table

<table>
<thead>
<tr>
<th>Config</th>
<th>$w_{11}$</th>
<th>$w_{12}$</th>
<th>$w_{21}$</th>
<th>$w_{22}$</th>
<th>$\theta_1$</th>
<th>$\theta_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9f</td>
<td>cc</td>
<td>5c</td>
<td>c4</td>
<td>19</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>25.15mV</td>
<td>60.38mV</td>
<td>$-27.8$mV</td>
<td>53.9mV</td>
<td>$-318$mV</td>
<td>$-82.5$mV</td>
</tr>
<tr>
<td>2</td>
<td>ae</td>
<td>2f</td>
<td>c7</td>
<td>b1</td>
<td>c4</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>36.5mV</td>
<td>$-62.4$mV</td>
<td>56.1mV</td>
<td>39.3mV</td>
<td>217.4mV</td>
<td>$-271.3$mV</td>
</tr>
<tr>
<td>3</td>
<td>9a</td>
<td>ac</td>
<td>67</td>
<td>b2</td>
<td>53</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>20.9mV</td>
<td>35.3mV</td>
<td>$-19.1$mV</td>
<td>39.9mV</td>
<td>$-136.9$mV</td>
<td>$-34.2$mV</td>
</tr>
</tbody>
</table>

8.3.2 Finding the Tolerance of a CTRNN configuration in simulation

The first test was to find out whether the CTRNN configurations were tolerant to parameter fluctuations. There is a high possibility that the current mirror based DAC and the differential amplifier designs, which are highly susceptible to random mismatches, might not be getting accurate programming codes. When a programming code is being sent, it might be interpreted as a slightly higher or lower voltage due to mismatches and the device might not work. The configurations in the tables were individually considered and their parameters were swept for tolerance studies.

**Tolerance to weights**

The configurations presented in Table 8.2 were swept around the evolved weight values for weight tolerance study. Figure 8.3 shows the CTRNN behavior (in simulation) of configuration 1 when the weight values are swept around +/-0.3 of their original values. What
Figure 8.2: CTRNN time series for configurations in previous table

this figure reveals is interesting. A qualitative analysis can be made that during parameter sweep, the CTRNN behavior maintains oscillatory. However, there is a randomness in the way the parameter variations affect the behavior. For this particular configuration, the CTRNN is more tolerant on the negative boundary than on the positive one, where there is a significant change in the CTRNN behavior. From the circuit point of view, this gets even more interesting. A change in the parameter value by 0.3 implies an equivalent change of 1.5mV. At weight values around 0 (or in other words, smaller weight values) this is a significant change. The tanh curve varies non-linearly and values close to zero are more sensitive to change in voltage than the higher parameter values. So, even this
small change of 1.5mV would translate to a bigger change when the parameter values are close to zero. Hence, it seems probable that the differential amplifier mismatches or charge injection might be inducing unforeseen parameter changes that causes incorrect parameter values.

**Tolerance to Bias**

It was seen previously that the CTRNN behavior is tolerant to small changes in weight values. Figure 8.4 reveals a parameteric sweep when the weights are kept constant and the biases are swept to +/-0.3 of their original values. Again, as in the previous case, it
Figure 8.4: CTRNN behavior at the boundaries of bias tolerance sweep

can be seen that this particular CTRNN configuration is more sensitive to variations on the positive side of the sweep. Figure 8.5 shows the CTRNN behavior in simulation when both the bias and weights are swept. This tolerance study is more realistic as both bias values and weight parameters are being programmed on-chip. The tolerance studies indicate that the CTRNN oscillatory behavior changes quantitatively for small perturbations around the parameter values. However, because the CTRNN device did not exhibit any sort of oscillatory behavior when programmed for the configurations in Table 8.2, two scenarios seem plausible:

- The CTRNN device may be getting erroneous programming values which might be
Figure 8.5: CTRNN behavior at the boundaries of combined tolerance sweep attributed to nonlinear characteristics. The error values for small weight and bias parameters might be getting amplified because of this nonlinear behavior.

- The worst case scenario is that the CTRNN device might not be working because of faulty design. However, this does not seem likely because all the individual circuit component/blocks were tested on-chip and these worked as expected, as previously noted.
8.3.3 Intrinsic Verification and Testing

The device functionality and the errors due to mismatch/charge injection was verified using intrinsic evolution experiments. In intrinsic evolution, CTRNN configurations are evolved in closed loop using Evolutionary Algorithms. Using the EA presented in Chapter 4 and the closed loop setup shown in 8.1, intrinsic experiments were conducted. CTRNN oscillatory configurations were evolved directly on-chip. The fitness function of the EA was defined simply to reward any oscillatory behavior.

Experimental Details

The EA parameters are given as follows:

- Mutation = 5%
- Genome length = NEURONS * NEURONS + 1 * NEURONS (4 weights and 2 biases)
- No. of Evaluations = 500

For the above parameters, the average time for one experiment was 12 hours. Thirty five chips were fabricated. Five chips were either damaged during testing or arrived faulty from the foundry. Five intrinsic experiments were run on each of 20 random chips sampled from the working lot of 30. Every one of the twenty chips had at least two working oscillator configurations evolved. As expected the CTRNN design was functionally correct and was not faulty. Hence, the problem with the extrinsic experiments can be confidently attributed to errors due to charge injection or mismatches after fabrication due to internal offsets causing the programmed values to be perturbed.
Experimental Results

Table 8.4 presents three oscillatory configurations evolved on-chip. These were randomly taken from the 100 experiments conducted.

<table>
<thead>
<tr>
<th>Config</th>
<th>$w_{11}$</th>
<th>$w_{12}$</th>
<th>$w_{21}$</th>
<th>$w_{22}$</th>
<th>$\theta_1$</th>
<th>$\theta_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>-12</td>
<td>13</td>
<td>5</td>
<td>3</td>
<td>-13</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7</td>
<td>-7</td>
<td>9</td>
<td>-3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>-12</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td>-13</td>
</tr>
</tbody>
</table>

Table 8.4: Some intrinsically evolved CTRNN configurations

The outputs of these configurations are shown in Figures 8.6, 8.7 and 8.8 respectively. The equivalent filtered outputs are shown in the same figures.

It can be seen that proper oscillatory CTRNN configurations could be evolved on-chip, which means that the functionality and design of the CTRNN is correct. The next step is to determine whether these configurations, as evolved on chip, oscillate in simulation. This will provide a clear explanation of the fabrication errors. The CTRNN was programmed (in simulation) to the configurations provided in Table 8.4.

Testing HW evolved configurations on ideal simulation

Figures 8.9, 8.10 and 8.11, provide the head to head comparison for hardware recorded output (filtered) and simulated output for the same configurations. It can be seen that quantitatively, the hardware-measured configuration differs from its respective expected
behavior. However, this trend confirms something that was previously suspected. In the previous section, a tolerance study was presented where CTRNN configurations were swept to +/-0.3 around their original values. This study was performed in simulation using ideal CTRNN models. The behavior observed there coincides with the behavior observed here. It is evident that the hardware CTRNN is subject to unmodeled errors that might be due to mismatches or charge injection of capacitors in memory cells.

An EA based error calculation scheme was employed to determine the exact effects of charge injection/mismatches. From the previous section about tolerance studies, it is evident that the hardware CTRNN configurations have been subject to random perturbations. So, the idea is to find out how much they are perturbed around their original values so that a calibration scheme can be designed. The ideal CTRNN was fine tuned with the help of an EA so that its output matches the measured hardware output. This experiment would give an overview about the parameter errors. It would also present if the error is localized and quantifiable. Table 8.5 presents the results of this experiment on the intrinsically evolved configurations of Table 8.4.

<table>
<thead>
<tr>
<th>Config</th>
<th>$w_{11}$</th>
<th>$w_{12}$</th>
<th>$w_{21}$</th>
<th>$w_{22}$</th>
<th>$\theta_1$</th>
<th>$\theta_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.1</td>
<td>-11.85</td>
<td>13.2</td>
<td>5.1</td>
<td>3.1</td>
<td>-12.9</td>
</tr>
<tr>
<td>2</td>
<td>0.3</td>
<td>7.1</td>
<td>-6.85</td>
<td>9.2</td>
<td>-2.85</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>4.2</td>
<td>-11.9</td>
<td>14.25</td>
<td>12.15</td>
<td>3.1</td>
<td>-12.9</td>
</tr>
</tbody>
</table>

Table 8.5: Actual parameter values found by fine tuning ideal CTRNN models

Figure 8.12 shows a scatter plot of the parameter perturbations across the entire 20
tested chips from the fabricated lot.

It can be seen that all the experiments reveal errors on the positive side of the axis. It has also been observed that a CTRNN configuration was more sensitive to positive valued errors. While this cannot be generalized, the scatter plot allows us to make some useful observations.

- The errors have qualitatively the same behavior (i.e. they range from [0.0, 0.3] raw value). This would translate to around 1.5mV error using the equation 4.1 from Chapter 4.

- This would suggest that the factor affecting the neural parameter is more of a localized problem than random effect.

- Charge injection or random mismatches might be contributing to the offsets. Since the values are randomly scattered, mismatches might be the actual cause of the observed errors.

### 8.3.4 EA based calibration scheme

This section will present a formalized calibration procedure based on the aforementioned discussion and experiments. In this calibration scheme, it was attempted to cancel the error. The calibration steps are as follows:

1. An oscillator configuration is evolved intrinsically as previously mentioned.
2. The evolved oscillator configuration is loaded in the ideal CTRNN model and its behavior is simulated.

3. A parametric sweep around the evolved oscillator configuration is conducted so that the simulated behavior matches with measured data.

4. A lookup table is created with parameters and their respective errors.

5. When the CTRNN device is programmed, a compensation is artificially injected based on this lookup table. This completes the calibration.

Figures 8.13 and 8.14 show the behavior of the CTRNN device after calibration. It can be seen that the calibration is effective in cancelling offsets induced either due to charge injection or mismatches. The quality of match is only as good as the calibration scheme. The current calibration scheme is not perfect due to communication latencies and random offset errors and hence a 100% match is not seen.
Figure 8.6: Recorded and filtered output of CTRNN (configuration 1)
Figure 8.7: Recorded and filtered output of CTRNN (configuration 2)
Figure 8.8: Recorded and filtered output of CTRNN (configuration 3)
Figure 8.9: Configuration 1 HW and Sim outputs

Figure 8.10: Configuration 2 HW and Sim outputs
Figure 8.11: Configuration 3 HW and Sim outputs

Figure 8.12: Scatter plot of the parameter perturbations on-chip
Figure 8.13: Configuration 1 after calibration

Figure 8.14: Configuration 2 after calibration
Chapter 9.

Generation 2 CTRNN Device

9.1 Introduction

The Generation two CTRNN device had four neurons that were fully interconnected. These neurons were arranged in grid like fashion as shown in 9.1. The figure presents the general architecture of the arrangement found in the Generation two device. This is a simple illustration of what is found on silicon and does not provide the entire interconnection details and information about the support circuitry. It can seen from the figure that there are 16 synapses in total representing various interconnection weights. Four sigmoidal activation blocks provide the respective neural outputs that are fed back to the input stages. Sensory inputs are provided to all the synapses along a column. This is because the synapses across a column correspond to the same neuron.
9.2 Synapse design

Because the design of individual blocks are similar to that found in Generation one device, the descriptions are similar to those already discussed. Nevertheless, these descriptions are provided again for continuity. Similar to the first generation CTRNN device, the synapse has been implemented by a simple two tailed differential pair. In the subthreshold operating region, this simple three transistor circuit allows one to implement an analog multiplier. This design was effectively used by Brown [7]. Also, since this circuit is effectively an operational transconductance amplifier (OTA), it provides a current output while allowing us to define our input parameter weight as a voltage. This scheme lets us realize a summer
directly, as mentioned above. The transfer characteristics of this circuit are given by the following equation

$$I_1 - I_2 = I_b \tanh \left( \frac{(V_w - V_{ref})}{2\eta V_{th}} \right)$$ \hspace{1cm} (9.1)

The transistor N2 is the input transistor. This generates the bias current $I_b$. The output of the neuron(s), which is current (shown later), is converted to voltage using diode connected load, and is provided as the input to the synapse cell at N2. $V_w$ is the voltage stored on the gate of transistor N0. This voltage represents the interconnection weight or programming parameter with respect to this cell. It can be seen that the output current, which is a differential value, is equivalent to the current generated by input voltage and a constant term set by the interconnection weight. However, this constant term is a hyperbolic tangent
depending on the interconnection weight and thus, non linear. In the normal sense, this
circuit is of limited use as this is not a linear analog multiplier. Hence, to make this circuit
useful for the CTRNN implementation, we should only consider its operation in its linear
region. For proper functioning of the synapse, the transistors N0 and N1 must operate in
subthreshold region. An experiment was conducted to determine the range of input current
that will allow the operation of these transistors in the desired region. An upper limit of
320nA was determined. To be in safe operating regions, the maximum current was fixed
to be 200nA. This was the upper limit set to ensure that synapse was acting as a linear
multiplier. Figure 9.3 shows the simulation result for this experiment. It can be seen that
the synapse behaves entirely linearly over the operating range. The variation of the weight
value results in the change of slope of the output voltage. The mapping between the analog
voltage value on the gate of N0 is given by

\[ V_w = 2\eta V_{th} \tanh^{-1}\left(\frac{w_{ij}}{w_{max}}\right) \tag{9.2} \]

where \( V_w \) is the voltage on gate of N0, \( V_{th} \) is the thermal voltage, \( w_{ij} \) is the raw voltage
value between [-15,15] and \( w_{max} \) has a value of 16. It should be noted that while the
maximum raw voltage value is 16, the weights are restricted to +/-15. This is because the
inverse hyperbolic tangent of 1/-1 is not defined. \( \eta \) is the non ideality factor and is process
dependent. A value of 1.5 is assumed for the simulations.

The layout of the synapse is shown in Figure 9.4. The input transistor’s aspect ratio
is 6/120. This is broken into three fingers for better matching. The transistors N0 and N1
have sizes of 24/3 respectively.
9.3 Sigmoidal Activation Function

The activation function of the CTRNN neuron is a tanh sigmoid, as was discussed in chapter 2. The implementation of the sigmoid circuit is identical to the synapse circuitry. While we were concerned about the linear operating range of the synapse circuitry, for the sigmoid circuit there is no such restriction. However, the primary difference between the sigmoid circuitry and the synapse circuitry is that the input nodes change. While transistor N0 stored a constant weight in synapse circuit, here it takes an input current through the integrator. The transistor N2 which was the input transistor in synapse acts as a constant current source. The gate voltage for this transistor is provided by on-chip bias circuitry. While the transistor N0 receives the input from the synapse and integrator, the transistor
N1 gets the bias value of the neuron as its gate voltage. The purpose of the bias voltage $\theta$ is to vary the orientation of the tanh curve along the x-axis. The test setup for a sigmoid simulation is presented in Figure 9.6. Figure 9.7 presents the simulation results when the value of $\theta$ is changed while sweeping along the input voltage of N0. The x-axis is a differential voltage between gates of transistors N0 and N1.
The conversion for $\theta$ in terms of the analog gate voltage of transistor N1 is given by

$$V_{\theta} = 2\eta\theta \quad (9.3)$$

It can be seen from the results that when $\theta$ value is varied, the sigmoid shifts along the x-axis. The layout for the sigmoid cell is shown in Figure 4.10. This layout, as one can expect, is similar to the layout of the synapse cell.
Figure 9.6: Sigmoid test circuit
Figure 9.7: Sigmoid simulation results
Chapter 10.

Test Results on Generation Two CTRNN Device

10.1 Introduction

This chapter will present the test results on the generation two CTRNN device. As mentioned in the previous chapter, the generation two device is a 4x4 neuron device. It was also mentioned in the previous chapter that the programming efforts using the improved programming cell circuitry did not work correctly. An alternate programming circuitry external to the IC was provided as a backup in the generation two device, anticipating this problem. This chapter will describe the details about this programming circuitry and measured results at the block level. The test setup of the hardware-in-loop system remains similar to that described in chapter 7.
10.2 Test setup

A test PCB was fabricated for the generation two device. The PCB layouts are described in figures 10.1 and 10.2. As seen, the second generation CTRNN device was packaged in a 40 pin DIP package. This was chosen for an easy testing interface. The PCB hosted DS1803 potentiometers for externally programming the on-chip CTRNNs. The programming circuit designs were motivated from the work published in [?] and more details can be found this work. The generation two CTRNN IC functioned on a supply voltage of \( V_{DD} = 2.2 \text{V} \). From the design discussions presented in the previous chapter, it is clear that the weight values follow a tanh function. The bias voltage \( V_{bias} \), which is the reference volt-
Figure 10.2: PCB layout for generation two device

...age for the synapse cells, comes from an external bias circuit, which was also described in the previous chapter. Because the synapse cells are differential amplifier based designs, the programming voltages must be differential to the bias reference voltage. Hence, the external programming circuits for the neuronal synapses are provided a reference voltage value that is equivalent to the bias voltage generated by the bias circuitry. This voltage can be sensed via the external I/O pins and, if inconsistent with the expected value, can be corrected by adjusting the input current that is sourced to the bias circuit. The programming circuitry is shown in Figure 10.3. The programming circuitry for the neuron biases ($\theta$) is similar to the synapse programming circuitry. The only difference is in the reference...
Figure 10.3: Programming circuitry for synapses

generation is required to match the voltage generated by the internal bias circuitry. This is shown in Figure 10.4.

Figure 10.4: Programming circuitry for synapses
10.3 Extrinsic evolution experiments

Extrinsic evolution experiments involve loading pre-evolved configurations to the CTRNN device and observing its functionality. Here, the idea is to determine whether the functionality observed in software simulation holds on hardware. In other words, does the hardware function as expected from mathematical models. This is one of the current open challenges in the field of Evolvable Hardware. The CTRNN was tested on ten randomly-chosen hardware configurations. The configurations are listed in Tables 10.1, 10.2, 10.3, 10.4.

<table>
<thead>
<tr>
<th>Config</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>θ</th>
<th>τ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.4</td>
<td>-11.58</td>
<td>-12.07</td>
<td>3.15</td>
<td>1.54</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>-1.41</td>
<td>-0.45</td>
<td>9.41</td>
<td>-0.20</td>
<td>-2.86</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>13.71</td>
<td>3.98</td>
<td>-0.188</td>
<td>-2.95</td>
<td>-6.39</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>-5.58</td>
<td>-12.28</td>
<td>5.20</td>
<td>1.15</td>
<td>12.32</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 10.1: Extrinsic Test: Configuration 1

<table>
<thead>
<tr>
<th>Config</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>θ</th>
<th>τ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.87</td>
<td>-13.96</td>
<td>-6.59</td>
<td>3.38</td>
<td>5.48</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>1.01</td>
<td>5.84</td>
<td>-2.8</td>
<td>3.95</td>
<td>-3.68</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>-0.85</td>
<td>-6.86</td>
<td>-4.48</td>
<td>7.69</td>
<td>-7.96</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>7.62</td>
<td>-1.54</td>
<td>11.17</td>
<td>-1.41</td>
<td>-4.44</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 10.2: Extrinsic Test: Configuration 2

The figures 10.5, 10.6, 10.7 and 10.8 show the measured results from the hardware for
Table 10.3: Extrinsic Test: Configuration 3

<table>
<thead>
<tr>
<th>Config</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>θ</th>
<th>τ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.98</td>
<td>0.13</td>
<td>-8.12</td>
<td>13.18</td>
<td>-7.69</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>-6.66</td>
<td>12.24</td>
<td>9.01</td>
<td>4.02</td>
<td>13.74</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>-7.47</td>
<td>12.74</td>
<td>-3.03</td>
<td>-6.92</td>
<td>8.86</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>-7.53</td>
<td>-7.62</td>
<td>5.23</td>
<td>-0.36</td>
<td>8.34</td>
<td>21</td>
</tr>
</tbody>
</table>

The software simulation results for each configuration was compared with filtered hardware data. The output recorded from the chip was subjected to low pass filtering in Matlab to filter out the noise. The figures 10.9, 10.10, 10.11 and 10.12 show the resultant plots.

The hardware output matches with the expected simulation output. In other words, the hardware behaves as determined by its mathematical model. This is a very significant result regarding reliability of the hardware. It can be seen that there are slight deviations in the form of wiggles at the peaks and crests of the waveform. These wiggles can be
eliminated further by decreasing the filter frequency. However, some of the wiggles are due to AC coupling in the system and cannot be completely eliminated. As is the case with any hardware, 100% matching with the ideal mathematical models is impractical.

### 10.4 Intrinsic Evolution Experiments

In the intrinsic evolution experiments, configurations are directly evolved on CTRNN hardware using the hardware-in-loop test setup demonstrated previously. Here, an evolutionary algorithm runs on the desktop PC and evolves configurations, which are loaded into the CTRNN device via the 8056 microcontroller. Over a hundred experiments were conducted. Four experiments were conducted on 25 samples from the fabricated lot of 40. Here, four
randomly chosen configurations out of the dataset of hundred experiments are presented. Tables 10.5, 10.6, 10.7 and 10.8 present the configurations evolved in hardware.

The figures 10.13, 10.14, 10.15 and 10.16 provide the hardware measured results of the CTRNN device for the aforementioned configurations. These oscillatory configurations were loaded into the software simulation consisting of ideal mathematical models of the neural network. The figures 10.17, 10.18, 10.19 and 10.20 provide the comparison between filtered hardware output and software simulation output.
Figure 10.7: Measured results CTRNN: extrinsic 3

Figure 10.8: Measured results CTRNN: extrinsic 4
Figure 10.9: Sim vs filtered hardware output: config1

Figure 10.10: Sim vs filtered hardware output: config2
Figure 10.11: Sim vs filtered hardware output: config3

Figure 10.12: Sim vs filtered hardware output: config4
<table>
<thead>
<tr>
<th>Config</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>θ</th>
<th>τ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.82</td>
<td>-3.9</td>
<td>-5.03</td>
<td>-9.9</td>
<td>6.47</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>9.65</td>
<td>10.56</td>
<td>-12.23</td>
<td>5.2</td>
<td>0.64</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>10.09</td>
<td>-8.98</td>
<td>9.54</td>
<td>-6.12</td>
<td>6.45</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>3.02</td>
<td>10.02</td>
<td>-7.99</td>
<td>6.90</td>
<td>-7.50</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 10.5: Intrinsic Test: Configuration 1

<table>
<thead>
<tr>
<th>Config</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>θ</th>
<th>τ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.14</td>
<td>-8.73</td>
<td>12.84</td>
<td>-7.35</td>
<td>-2.44</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>5.76</td>
<td>11.52</td>
<td>2.75</td>
<td>-0.49</td>
<td>-10.75</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>13.92</td>
<td>-5.91</td>
<td>-6.62</td>
<td>13.62</td>
<td>-7.31</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>-8.52</td>
<td>12.45</td>
<td>4.27</td>
<td>-11.64</td>
<td>-2.21</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 10.6: Intrinsic Test: Configuration 2

<table>
<thead>
<tr>
<th>Config</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>θ</th>
<th>τ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.63</td>
<td>-3.92</td>
<td>9.03</td>
<td>7.69</td>
<td>-5.75</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>-4.62</td>
<td>5.45</td>
<td>-11.91</td>
<td>-8.59</td>
<td>12.35</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>-6.20</td>
<td>2.99</td>
<td>3.6</td>
<td>-8.86</td>
<td>-1.45</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>-6.67</td>
<td>-3.08</td>
<td>10.74</td>
<td>-1.42</td>
<td>1.53</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 10.7: Intrinsic Test: Configuration 3
<table>
<thead>
<tr>
<th>Config</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>θ</th>
<th>τ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.39</td>
<td>10.35</td>
<td>-2.51</td>
<td>-5.79</td>
<td>-5</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>1.38</td>
<td>-7.5</td>
<td>-2.96</td>
<td>-7.18</td>
<td>9.6</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>6.58</td>
<td>-12.04</td>
<td>10.79</td>
<td>10.54</td>
<td>-4.91</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>8.78</td>
<td>-1.19</td>
<td>-8.22</td>
<td>5.07</td>
<td>-1.15</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 10.8: Intrinsic Test: Configuration 4

Figure 10.13: Measured results CTRNN: intrinsic 1
Figure 10.14: Measured results CTRNN: intrinsic 2

Figure 10.15: Measured results CTRNN: intrinsic 3
Figure 10.16: Measured results CTRNN: intrinsic 4

Figure 10.17: Sim vs filtered hardware output: configuration 1
Figure 10.18: Sim vs filtered hardware output: configuration 2

Figure 10.19: Sim vs filtered hardware output: configuration 3
Figure 10.20: Sim vs filtered hardware output: configuration 4
11.1 Introduction

In this chapter, the CTRNN device functionality will be demonstrated on a real world problem. There were two sets of experiments that were conducted to verify the functionality of CTRNN device on a control problem. In the first experiment, a closed loop vibration control of a mathematical combustion chamber model was attempted. This model is described in detail in [28, 29, 51]. Also, a successful demonstration of CTRNN control for this problem was demonstrated in [51, 55]. From these previous results, CTRNN devices are theoretically suitable for controlling vibrations in turbine jet engines. The following section will describe the details of this experiment and observations. This experiment was unsuccessful as the CTRNN device could not stabilize the instability in the mathematical
Active noise control will be demonstrated where the CTRNN devices were used to evolve phase inverting circuits that can be used to generate anti-noise waveforms, which are useful in mitigating mechanical noise and vibrations [52].

11.2 Active Noise Control

11.2.1 Introduction

Active noise cancellation is modification of a sound field by electro-acoustic means. Usually, the procedure involves driving a speaker to create a sound field that is a mirror image of the noise field so that the offending noise is dampened [43, 10]. While such inversion methods can be directly applied for periodic noises, it becomes increasingly problematic for non-periodic noises. Active noise control has been successfully attempted in a number of instances for several applications ranging across different engineering disciplines [47, 9]. With increasing interest in automotive and military applications, the need for sophisticated noise control devices is growing [9, 40, 39]. Anti-noise devices, as they are popularly known, range from specially made headsets for combat pilots to dampen surrounding unwanted noise disturbances [40], to anti-noise devices for suppressing rotor noises in jet aircrafts [39]. Recent successes in noise cancellation have been impressive. However, there are still many challenges to be met. One category of challenges involves reducing the size and power consumption of these systems, another category involves constructing systems that can cancel non-periodic noise [38]. Here, we introduce the concept of Active Noise
Control using Continuous Time Recurrent Neural Network Evolvable Hardware (CTRNN-EH). One can view such control scheme as Mechatronic control systems that have dynamic recurrent neural networks as their core component as in [42].

**Figure 11.1: CTRNN-EH Device for Noise control**

### 11.2.2 Principle

The CTRNN-EH device acts like a traditional feedback control device by monitoring sensory inputs presented from the plant and providing actuator efforts to the plant (shown in fig 11.1). While this explanation is sufficient to understand the high-level operation of the device, further explanation is necessary to appreciate the ways in which the device may be used. One use of the CTRNN-EH device is to instantiate a known controller methodology. Classic example is to invert the sound field for periodic low frequency disturbances. For such applications, the ENC is pre-programmed or pre-configured either using a configuring engine or via external programming pins. The configuring engine does not come to play
under the normal operating conditions. This is because, if the goal of the CTRNN-EH device is to invert the periodic sound field and this remains the controlled environment, the pre-programmed configuration will suffice. This is a simple case and can be applied to similar applications. Another use of the CTRNN-EH device is to allow it to self-configure for more challenging applications. Consider the examples of trying to suppress vibrations in combustion chamber of turbine jet engines, engine rotor blades, non-periodic low frequency disturbances, etc. In these cases and across many qualitatively similar applications, the configuring engine plays a very significant part. In these applications there are obvious challenges. Some of them are - lack of complete domain knowledge to design an effective controller, changes in the dynamics of the system due to mechanical factors and ageing, random noise, etc. For such applications, the CTRNN-EH device goes into self-training mode where the configuring engine configures the CTRNN controller based on the observed behavior of the system. Here, the functionality of CTRNN-EH VLSI device will be tested on the first of the aforementioned illustrations. Periodic anti-noise signals will be evolved on the CTRNN-EH hardware.

11.3 Sensory circuit

In the previous experiments, oscillators were evolved on the CTRNN device. For evolving oscillators, external sensory inputs are not necessary. However, for sensing the output of the engine model and providing a control signal for stabilization, the CTRNN must be able to consider the information presented to it. Internally, the architecture of the CTRNNs used in the CTRNN device allows sensory information in the form of currents. However,
this current must be weighted before being presented to the device. For this purpose, an external current regulator circuit using LM317 was designed. By varying the supply voltage on the input pin of LM317, the output current can be regulated. The maximum value of the weighted current is comparable with neuron output currents. Hence, using the output channels of the data aquisition device, the supply voltage of this circuit was controlled so that output current of this circuit varied from 0 to 200nA. The details of the current regulator circuit, block diagram of the complete system and the experimental setup is presented in 11.2,11.3, and ?? . The current regulator circuit is taken from the LM317 datasheet and shows a constant supply voltage resulting a constant regulated current. For the purposes of our experiments, this was changed.

Figure 11.2: Current regulator circuit used for sensory inputs


11.4 Experimental results

The CTRNN devices failed to suppress the vibrations in the mathematical engine models.

Upon investigating, the following reasons were found out to be prominent:

1. The noise in the CTRNN device output had to be filtered before sending into the engine model as actuation signals. This was hard to do at run time. The high frequency noise was causing the control efforts to fail.

2. The sensory inputs provided by the external regulator circuit was not perfectly linear. High resolution of current division was not possible because the currents were in the low range of nano amperes. Hence, an accurate sensory input could not be provided for the CTRNN device.

The above issues are mostly associated with the experimental setup rather than the device itself. The fact that the engine model is running on a software simulation and the CTRNN
device is trying to stabilize that model in real time poses a lot of communication and latency
issues. Hence, the mode of experiment was changed. The primary motivation was to test
the functionality of CTRNN device while actively monitoring its sensory inputs, which
would enable its use in closed loop control. Hence, any experiment that will help achieve
this goal is satisfactory. In a previous work [52], CTRNN controllers were demonstrated
for active noise control. There, it was shown that CTRNN controllers can actively invert
the phase of an input signal for generating equivalent anti-noise. Using the same principle,
a second experimental setup was constituted for testing the capability of CTRNN devices
to generate phase invertors.

11.5 Active noise controllers: Phase inverting CTRNNs

In [52], phase inverting CTRNN controllers were demonstrated to be able active noise con-
trol devices. There, it was demonstrated that CTRNN were capable of generating anti-noise
waveforms for both periodic and non-periodic signals. In this work, the possibility of gen-
erating phase inverted controllers real time with the CTRNN devices will be tested. This
experiment will serve as an excellent testbed for evaluating the functionality of CTRNN
device with reference to sensory inputs. The block diagram for this experiment is similar
to one shown above for engine control experiments. Here, the CTRNN device monitors
the input waveforms presented as sensory inputs and outputs current waveforms that are
an exact phase inverted versions of the presented inputs. For the phase inversion exper-
iments, the experiments were intrinsically evolved. The evaluation function for the EA
was changed to capture phase inverted controllers. A reference waveform was fed through
the data acquisition card to the sensory input pin. This input was current. The evaluation function used was a minimization function that added the input and output waveforms and calculated a least mean square error forcing strict phase inversion. Five experiments were conducted and the CTRNN device successfully evolved phase inverting circuits in four of these experiments. Figure 11.4 shows the two best phase invertors. Reference waveform in indicated in the same plot for a head-to-head comparison. It must be noted that the y-scales in the normalized units of zero to one.

It can be seen from the figure that the input waveform does not need to be sinusoidal to evolve good phase inverters. As a matter of fact, any periodic and non-periodic phase inverters can be evolved. This figure shows a damped sinusoidal and a pure sinusoidal input waveforms used for experiments.
Figure 11.4: Phase inversion controllers
Chapter 12.

Conclusions

12.1 CTRNN-EH Overview

In this dissertation, the design, implementation and testing of a fully functional CTRNN-EH device was presented. CTRNN-EH devices are inspired from the idea of using Continuous Time Recurrent Neural Networks for Evolvable hardware applications. Unlike the existing EH practices, where reconfigurable hardware like FPGA and FPTA are used in combination with evolutionary algorithms, a dynamically reconfigurable neural network is employed in this work. In the traditional EH practices, the transistor arrays or gate arrays are configured dynamically using evolutionary practices. Because several permutations of interconnects are possible on these devices, they more often than not, behave unpredictably and not amenable to any sort of analysis. Thus, EH devices evolved using these mechanisms have not found engineering acceptance. CTRNNs on the other hand, do not exhibit such problems. They have a static architecture after fabrication, meaning that no intercon-
nections or signal paths are changed. The reconfigurability is achieved using programming the CTRNN parameters. Hence, prior to fabricating this device, the reconfigurability can be thoroughly analyzed using extensive simulations. Also, this will allow us the possibility of ensuring the fidelity between hardware performance and mathematical simulations, which was not possible in other reconfigurable hardware devices.

12.2 Summary of this work

With the above motivation, design of a CTRNN based evolvable hardware device was undertaken. The primary objectives of this design were

1. Predicatability of operation.

2. Comparable performance between measured hardware results and mathematical simulations.

3. Feature to accept external sensory inputs allowing the device to be used in closed loop control applications.

4. Incorporate a learning engine on the same device as the CTRNN.

5. Minimize the area and power of both the neural network and evolutionary algorithms.

There were three existing CTRNN implementations in the literature [35, 7, 8]. None of these three designs were suitable for EH application in their present form. However, the design presented in [7] was elegant and a low power design. There was no programming
circuitry for this design. The CTRNN was 2x2 (two neurons and fully interconnected) and
programmed externally via I/O pins. This design was modified to support higher number of
neurons (upto four) and digital programming capability was added. A novel programming
scheme was designed for digitally programming the CTRNN device. The device could be
programmed using 8 data lines and 5 address lines. The generation one CTRNN device
was fabricated. This device had a 2x2 architecture with a programming capability. The
programming circuit was implemented using binary weighted current mirrors.
Intrinsic and Extrinsic experiements were conducted on this device. Initially, the device was
programmed with known configurations to evaluate the functionality. In these extrinsic ex-
periments, it was observed that the CTRNN device does not behave as expected. For all the
oscillator configurations that were loaded into the device, none exhibited oscillatory behav-
ior. However, all the individual neuron blocks and the digital to analog converters exhibited
correct functionality when tested. Logically presuming that the problems might be associ-
ated with mismatches, intrinsic experiments were run on generation one CTRNN devices.
Using an EA, oscillators were evolved on the CTRNN devices. Oscillatory configurations
were successfully evolved on the generation one CTRNN devices. However, when those
configurations were ported back to mathematical models, the oscillators were not observed.
Using evolutionary algorithms, a closed loop calibration scheme was designed where the
neuron parameters were fine tuned. It was observed that after fine tuning, the hardware and
mathematical simulations were identical to each other. The initial inconsistencies between
the hardware measured results and mathematical simulations could be attributed to random
mismatches or charge injection issues in the CTRNN design. However, with calibration,
these issues could be addressed and the full functionality of the CTRNN could be observed.
The first generation 2x2 CTRNN device was successful and a lot of information was learnt about the programming capabilities and neural network design itself. For the second fabrication run, the neural network size was increased to four. A 4x4 CTRNN was fabricated with an alternate programming circuitry based on PMOS ladder networks. This was another novel attempt at integrating the programming circuitry within the existing synapse circuits. However, this programming circuit failed to work. The reasons are not clearly known. Hence, an external programming array using digital potentiometers was constructed. The 4x4 CTRNN worked accurately for both intrinsic and extrinsic configurations. The next set of experiments were conducted to verify the device’s capability to process sensory inputs accurately. For evaluating the sensory inputs, closed loop control problem was chosen. A mathematical model of engine combustion chamber exhibiting thermoacoustic instability was running on a desktop PC. Using the CTRNN device, suppressions of the vibrations in the combustion model were attempted. However, these experiments were not successful and the CTRNN device could not suppress the vibrations. The primary reason was that the experimental setup was unfair and had to contend with a lot of communication latency and noise. Hence, an alternate experiment was constructed. Here, using a similar closed loop setup, phase inversion circuits were evolved on the CTRNN device. Periodic signals were input via the sensory input of the CTRNN device and controllers were evolved that generated phase inverted signals of the input. This experiment was successful and the accurate processing of sensory inputs was verified.
12.3 Goals achieved in this work

In the work presented in this dissertation, two working versions of CTRNN device was presented. Further, a novel programming circuitry based on current mirrors and capacitive storage was provided. A fully functional digital implementation for minipop evolutionary algorithm was presented which was optimized for area and power. Unfortunately, due to budgetry constraints, a complete SOC with the digital EA and the CTRNN could not be fabricated. The closed loop functionality of CTRNN devices were demonstrated and it was shown that sensory inputs were being accurately interpreted and processed.

12.4 Contributions of this dissertation

The following are the primary contributions of this work to the field of Evolvable hardware

1. First analog evolvable hardware in the literature that is based on behavioral reconfigurability.

2. First analog evolvable hardware device where Intrinsic configurations behave accurately when extrinsically tested and vice-versa.

3. First analog evolvable hardware device where Software to hardware fidelity is accurately maintained.

4. First analog evolvable hardware device where hardware behavior can be predicated beforehand using software/mathematical simulations.
The following are the primary contributions to the field of analog/mixed signal VLSI

1. A novel programming scheme based on capacitive memory for programming continuous time recurrent neural network

2. First CTRNN device reported in literature with full connectivity combined with sensory processing capability.

3. EA based calibration scheme is novel for cancelling differential amplifier offsets, mismatches and charge injection effects.

12.5 Future work

Obvious extension is to integrate the CTRNN and the digital EA on a single device. This would allow a complete SoC capability. Better programming schemes can be investigated that would eliminate the need for external calibrations. The PCB design employed for this work was conservative and the goal was to minimize the noise. A compact design can be created for better portability and testing. The design employed for the CTRNN is scalable upto 8 neurons. However, the routing and noise issues might dominate as the number of neurons increase. This certainly is an area to work upon where other design approaches are investigated to easily scale the number of neurons while not increasing noise and complexity.
Bibliography


Appendix A.

Matlab code for filtering

\[
\begin{align*}
d &= 1.0; & \text{\% duration} \\
n &= \text{size}(\text{ts}); & \text{\% number of samples}
\end{align*}
\]

\[
\begin{align*}
\text{while 1} & \\
\text{clc;} & \\
v &= \text{input('lowpass(1), highpass(2), bandpass(3), notch(4) ? ');} & \\
\text{if } v \geq 1 & \text{ \& } v \leq 4; & \\
\text{break} & & \\
\text{end} & \\
\text{end} &
\end{align*}
\]

\[
\begin{align*}
\text{while 1} & \\
\text{clc;} & \\
v &= \text{input('lowpass(1), highpass(2), bandpass(3), notch(4) ? ');} & \\
\text{if } v \geq 1 & \text{ \& } v \leq 4; & \\
\text{break} & & \\
\text{end} & \\
\end{align*}
\]

\[
\begin{align*}
\text{lf} &= 10; & \text{\% lowest frequency} \\
\text{hf} &= 4000; & \text{\% highest frequency} \\
\text{lp} &= \text{lf} \times d; & \text{\% ls point in frequency domain} \\
\text{hp} &= \text{hf} \times d; & \text{\% hf point in frequency domain}
\end{align*}
\]

\[
\begin{align*}
\text{while 1} & \\
\text{clc;} &
\end{align*}
\]
switch v
    case 1
        a = [‘LOWPASS’];
        filter = zeros(1, n); % initialize by 0
        filter(1, 1 : lp) = 1; % filter design in real number
        filter(1, n - lp : n) = 1; % filter design in imaginary number
    case 2
        a = [‘HIGHPASS’];
        filter = ones(1, n); % initialize by 1
        filter(1, 1 : hp) = 0; % filter design in real number
        filter(1, n - hp : n) = 0; % filter design in imaginary number
    case 3
        a = [‘BANDPASS’];
        filter = zeros(1, n); % initialize by 0
        filter(1, lp : hp) = 1; % filter design in real number
        filter(1, n - hp : n - lp) = 1; % filter design in imaginary number
    case 4
        a = [‘NOTCH’];
        filter = ones(1, n);
        filter(1, lp : hp) = 0;
        filter(1, n - hp : n - lp) = 0;
    end

% do filter
s = fft(ts'); % FFT
s = s .* filter; % filtering
s = ifft(s); % inverse FFT
s = real(s);

plot(ts);
figure;
plot(s);
q=s';