CASCADED DIGITAL REFINEMENT FOR INTRINSIC EVOLVABLE HARDWARE

by

VIGNESH THANGAVEL
B.E. (Hons.) Birla Institute of Technology and Science, Pilani, 2013

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

Spring Term
2015

Major Professor: Ronald F. DeMara

ABSTRACT

Intrinsic evolution of reconfigurable hardware is sought to solve computational problems using the intrinsic processing behavior of System-on-Chip (SoC) platforms. SoC devices combine capabilities of analog and digital embedded components within a reconfigurable fabric under software control. A new technique is developed for these fabrics that leverages the digital resources' enhanced accuracy and signal refinement capability to improve circuit performance of the analog resources' which are providing low power processing and high computation rates. In particular, Differential Digital Correction (DDC) is developed utilizing an error metric computed from the evolved analog circuit to reconfigure the digital fabric thereby enhancing precision of analog computations. The approach developed herein, Cascaded Digital Refinement (CaDR), explores a multi-level strategy of utilizing DDC for refining intrinsic evolution of analog computational circuits to construct building blocks, known as Constituent Functional Blocks (CFBs). The CFBs are developed in a cascaded sequence followed by digital evolution of higher-level control of these CFBs to build the final solution for the larger circuit at-hand. One such platform, Cypress PSoC-5LP was utilized to realize solutions to ordinary differential equations by first evolving various powers of the independent variable followed by that of their combinations to emulate mathematical series-based solutions for the desired range of values. This is shown to enhance accuracy and precision while incurring lower computational energy and time overheads. The fitness function for each CFB being evolved is different from the fitness function that is defined for the overall problem.

I dedicate this work to my family who constantly supported me and believed in me even in the face of complete loss of hope of further progress, only to steer me to turn it around to what it is now. I also dedicate this work to my advisor and guru, Dr. Ronald F. DeMara for being patient and tolerant of my quirks, for giving the right push time and again, streamlining the occasional eccentricity into worthwhile products and for imbibing in me a spirit of professionalism for life! I also dedicate this work to my best friends, Hariharan Ramakrishnan and Arvind Ranganathan for providing constant motivation and to help me take it easy just as much as I needed both.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and sincere thanks to my advisor Dr. Ronald F. DeMara, for providing me with extensive guidance, astute insights and tremendous support in every leg of research, without which this thesis would not have been possible. I would like to extend special thanks to Dr. Zixia Song for being in my thesis committee, brainstorming and adding a new mathematically richer perspective to the thesis to help alter its course for the better. I would also like to thank Dr. Kalpathy Sundaram for serving as my thesis committee member and for the constant encouragement to pursue excellence in research. I would like to take this opportunity to thank my research partner Mr. Steven Pyle, for providing brilliant suggestions and brainstorming ideas at crucial moments to ensure successful development of well-motivated research. I would also like to extend my thanks to all members of the UCF Computer Architecture Lab team for their encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE: INTRODUCTION

Technology scaling and the criticality of issues like power wall and dark silicon as elaborated in [1] are magnified in pure digital frameworks for computation, thereby compromising the enhancement in computational efficiency otherwise sought and expected. One viable candidate to navigate through this labyrinth of deeply interconnected problems is the emerging field of cooperative analog-digital computations which leverages the complementary benefits of both as demonstrated successfully in the signal processing domain in [2]. Design of analog circuits to perform computations, though attractive, is highly non-trivial and is fraught with issues of scalability and flexibility to perform a wide range of computations with added noise issues. They however handle continuous ranges very cost-effectively and interface well with real world data. Pure digital design on the other hand is very systematic and offers great computational versatility and robustness, but is prohibitively costly and complex to handle continuous data ranges and requires lot of conversions to interface well with real world data.

Significance and Context of the Work

While analog and digital computations have their own pros and cons, it is distinctly visible that these are complementary as well and can be exploited intelligently if present and available for reconfiguration, ideally on the same platform with complete interconnection of both types of resources. PSoC 5LP is one platform that offers reconfigurable analog and digital fabrics at a low cost for developing low power applications. Though it doesn't offer much interconnection between every unit of the analog and digital fabrics, respectively, it does offer reconfigurable resources in both domains under the software control of a single embedded ARM Cortex-M3 core. However, due to the limited memory and silicon availability on chip, their reconfigurable use for bigger and complex applications is limited especially by parallelizability of processes. In spite of these limitations, this platform provides an impressive variety of resources considering its cost and ability to emulate general purpose computations.

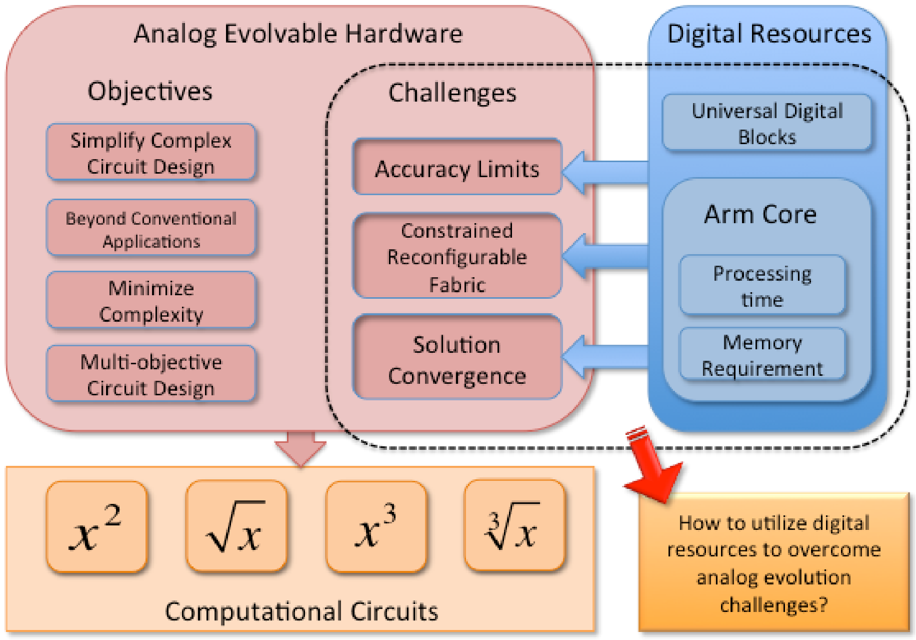

Analog Evolvable Hardware
Objectives
Simplify Complex Circuit Design
Beyond Conventional Applications
Minimize Complexity
Multi-objective Circuit Design
Challenges
Accuracy Limits
Constrained Reconfigurable Fabric
Solution Convergence
Digital Resources
Universal Digital Blocks
Arm Core
Processing time
Memory Requirement
x^2
$\sqrt{x}$
x^3
$\sqrt[3]{x}$
Computational Circuits
How to utilize digital resources to overcome analog evolution challenges?

Figure 1: Paradigm of Analog – Digital Computations [35]

Motivation

Evolutionary algorithms (EAs) are a well-known bio-inspired meta-heuristic to find solutions to complex multi-objective problems especially where the fitness function being evaluated for quality of solutions is multi-modal and a population of good solutions is desired as [3] elaborates. Genetic Algorithm (GA) in particular, are a very popular class of EAs that mimic

Darwinian evolution and the process of natural selection to pick candidate solutions from populations of configurations. As [3] explains, GA's implicit parallelism and the uncanny ability of finding building blocks inspire one to think of new ways to solve problems that differ from the tried and tested classical approaches which tend to incur more computational costs in many settings. One main characteristic of solutions developed by GAs is the inherent approximations that make them suitable for the application at hand to the desired degree without compromising on accuracy where necessary. The paradigm of approximate circuit synthesis as outlined in [4], introduces the notion of a very effective practical tool of functional approximation to achieve objectives of lower computational overheads. Also, intrinsic evolution helps exploit device properties beyond simulations and design abstractions and assumptions otherwise used which often face risk of violation in practical circuits as the pioneering work in [5] elaborates. Though [5] predicts limited applications such as signal processing for intrinsic evolution, it could possibly be applied to a broader range of scenarios and inspire better approaches to perform computations. These qualities encouraged the cascaded approach developed herein, which starts by evolving building blocks to the solution at hand with desired levels of tunable accuracies, followed by the process of combining them appropriately, all intrinsically on chip.

Platform and Resources

To develop techniques for intrinsic evolution of the digital fabric to improve upon the outputs of the analog fabric, a clear understanding of PSoC's capabilities and limitations is necessary. The following section outlines the same for PSoC 5LP System-on-Chip platform.

The major systems interoperating in PSoC 5LP devices are the Digital System, Analog System, Memory System, CPU System, Program/Debug Section, Clocking System, and Power Management System. These systems are equipped with the following features as listed in the PSoC architecture TRM [6]:

- ARM Cortex-M3 CPU with a nested vectored interrupt controller and a high-performance DMA controller
- Several types of memory elements including SRAM, flash, and EEPROM
- System integration features, such as clocking, a feature rich power system, and versatile programmable inputs and outputs
- Digital system that includes configurable universal digital blocks (UDBs) and specific function peripherals, such as CAN and USB
- Analog subsystem that includes configurable switched capacitor (SC) and continuous time (CT) blocks, up to 20-bit Delta Sigma converters, 8-bit DACs that can be configured for 12-bit operation, more than one SAR ADC, comparators, PGAs, and more
- Programming and debug system through JTAG, serial wire debug (SWD), and single wire viewer (SWV)

PSoC’s Reconfigurable Digital Fabric

PSoC’s digital fabric consists of PLDs or programmable logic devices organized in pairs inside the universal digital blocks or UDBs which can all be interconnected with one another in a desired fashion and hence constituting a reconfigurable digital fabric. There are 24 UDBs or 48

PLDs in total. The routing is not easily reconfigurable programmatically, involving too many register writes for each configuration and in the absence of clear documentation or support, routing is practically fixed based on the schematic configuration at boot time by the IDE. In addition to combinational logic, PSoC 5LP also has sequential logic in each UDB consisting of ALU, registers, FIFO and comparators. However, the datapath isn't very flexible to support dynamic reconfiguration and is hence not explored in this work.

Contributions of Thesis

1. A novel refinement technique implemented with a small amount of digital logic and memory to improve the accuracy of computations performed by analog circuits significantly.
2. Reducing chromosome lengths and introducing a new paradigm of piecewise evolution in cascaded stages followed by higher level evolution for control of these pieces.
3. An exploration in the mathematical decomposition and techniques of approximation of functions using variations of Laurent and Puiseux series to develop newer and better adjustments that can improve accuracy of calculations with available resources significantly.

Organization of Thesis

This thesis is organized into eight chapters. First the significance and context of this thesis is enumerated upon, motivation explained and the platform and resources used in building this work is described in the introductory chapter. Next, related works are explored, paradigms of multi-stage and piecewise evolution introduced and the rationale behind choice of mode of crossover, mutation implementation and elitism and selection procedures are justified based on literature in this chapter on literature review. In chapter 3, the Programmable System on Chip platform (PSoC) and the resources available on it for dynamic reconfiguration are explored. In Chapter 4, Differential Digital Correction (DDC) technique developed to evolve various powers of x is elaborated on. In Chapter 5, the Coefficient Prediction (CP) technique developed to predict coefficients of various powers of x evolved in DDC is elaborated on. In chapter 6, the Cascaded Digital Refinement (CaDR) technique is reviewed and the nature of solutions obtained is discussed to improve on the scheme. In Chapter 7, Results of the work done are summarized, starting with comparison with other works, evolution of various powers of x using DDC and their use in CPGA to evolve solutions to sine and cosine of x. The coefficients obtained are also discussed. In Chapter 8, Conclusions about the significance of results, improvements and suggestions for future work and use of Built in Self-Test (BIST) based fitness functions to improve system reliability are discussed.

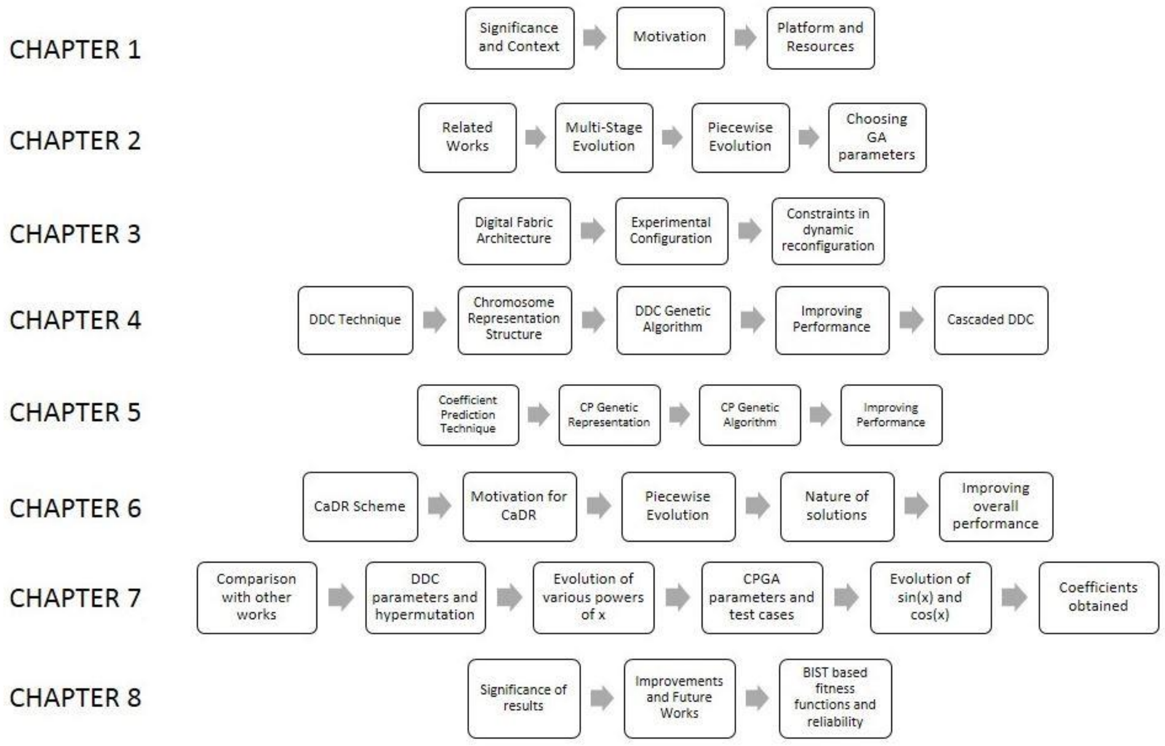
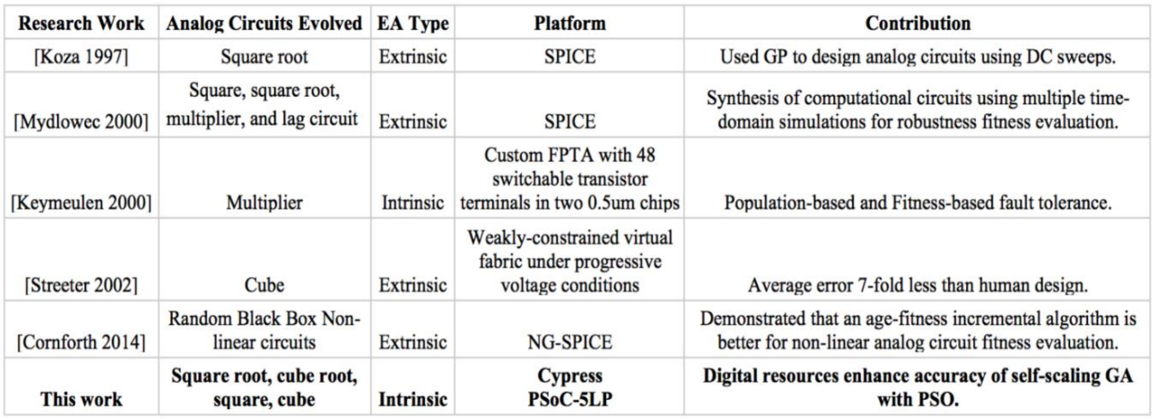

CHAPTER 1
Significance and Context
Motivation
Platform and Resources
CHAPTER 2
Related Works
Multi-Stage Evolution
Piecewise Evolution
Choosing GA parameters
CHAPTER 3
Digital Fabric Architecture
Experimental Configuration
Constraints in dynamic reconfiguration
CHAPTER 4
DDC Technique
Chromosome Representation Structure
DDC Genetic Algorithm
Improving Performance
Cascaded DDC
CHAPTER 5
Coefficient Prediction Technique
CP Genetic Representation
CP Genetic Algorithm
Improving Performance
CHAPTER 6
CaDR Scheme
Motivation for CaDR
Piecewise Evolution
Nature of solutions
Improving overall performance
CHAPTER 7
Comparison with other works
DDC parameters and hypermutation
Evolution of various powers of x
CPGA parameters and test cases
Evolution of sin(x) and cos(x)
Coefficients obtained
CHAPTER 8
Significance of results
Improvements and Future Works
BIST based fitness functions and reliability

Figure 2: Organization of Thesis

CHAPTER TWO: LITERATURE REVIEW

Related Works

A variety of Evolutionary Algorithms (EAs) have been used to realize novel electronic circuit designs intrinsically on reconfigurable fabrics. Numerous innovative works have contributed to the literature of which only a few are highlighted in Table 1 relating to analog and hybrid analog-digital domains. For example, Koza et al. demonstrated an approach to automatic synthesis of analog circuits using Genetic Programming (GP) which included crossover and lowpass filters at various frequencies, an amplifier, a source identification circuit, a CC (cube-root), a time-optimal controller circuit, a voltage reference circuit, and a temperature-sensing circuit, all extrinsically using DC sweeps for fitness evaluation [16]. Mydlowec and others followed the path of Koza, evolving other CCs extrinsically [17, 18], in some cases using multiple time domain simulations to improve robustness.



Research Work	Analog Circuits Evolved	EA Type	Platform	Contribution
[Koza 1997]	Square root	Extrinsic	SPICE	Used GP to design analog circuits using DC sweeps.
[Mydlowec 2000]	Square, square root, multiplier, and lag circuit	Extrinsic	SPICE	Synthesis of computational circuits using multiple time-domain simulations for robustness fitness evaluation.
[Keymeulen 2000]	Multiplier	Intrinsic	Custom FPTA with 48 switchable transistor terminals in two 0.5um chips	Population-based and Fitness-based fault tolerance.
[Streeter 2002]	Cube	Extrinsic	Weakly-constrained virtual fabric under progressive voltage conditions	Average error 7-fold less than human design.
[Cornforth 2014]	Random Black Box Non-linear circuits	Extrinsic	NG-SPICE	Demonstrated that an age-fitness incremental algorithm is better for non-linear analog circuit fitness evaluation.
This work	**Square root, cube root, square, cube**	**Intrinsic**	**Cypress PSoC-5LP**	**Digital resources enhance accuracy of self-scaling GA with PSO.**

Figure 3: Collection of Related Works [35]

Keymeulen et al. demonstrated intrinsic EHW on Field Programmable Analog Arrays (FPAAs) for population-based and fitness-based evolution of fault-tolerant analog circuits [30].

Later, Streeter et al. [29] also showed that GP was able to iteratively evolve circuits that could be attached to computational circuits to refine their performance. In [26] EAs were used to evolve four analog CCs as well as two digital circuits using analog components. In [23] swarming algorithms such as PSO were used to evolve analog circuit sizing. Recently, Cornforth et al. evolved non-linear circuits by utilizing a strategic fitness evaluation scheme without necessarily optimizing them for area. They were able to show that a variety of stimuli can extrinsically evolve nonlinear analog circuits, which conform to randomly generated black-box circuits, demonstrating the strength of the method.

While several previous works in analog CC design using EAs have involved simulation, recent Programmable System on Chip (PSoC) devices providing reconfigurable analog fabric, digital logic, and ARM cores enable new capabilities. Analog fabrics allow rapid evolution, but are limited by precision and/or accuracy, which may be refined with evolved digital circuits. The ARM core on the PSoC allows on-chip execution of EAs such as the GAs developed herein.

The work done by Shin and Hitoshi [32] describes the paradigm of multi-stage evolution to increase evolutionary pressure to manage circuit area more efficiently. This work also shows the advantages of using multi-stage evolution with regards to power, memory usage and better conformation to stringent accuracy requirements. Every succeeding stage of evolution leverages the results of the previous stage to evolve solutions that satisfy more strict accuracy and quality requirements than the previous stages, thereby supporting the fact that evolution in stages is desirable for design of complex circuits. This supports the idea of cascaded evolution stage followed by coefficient prediction developed herein.

Work done by Kazadi, et.al [33], on the other hand explores piecewise evolution to evolve pieces of a circuit in discrete functional stages and breakdown of the problem into independent output units. They have shown that piecewise evolution that involves relatively random breakdown of a problem into independent units doesn’t guarantee any improvements over overall evolution of the whole circuit as one piece. However, their work also mentions, “What would seem to be the correct line of future work in this area would be the investigation of viable ways of breaking down complex problems into simple connected parts”. Hence, for problems where it is possible to achieve breakdown into simple connected stages, the paradigm of piecewise evolution is feasible. In the approach developed herein, it is mathematically shown that such breakdown is achieved for any mathematical function and hence the idea of cascaded evolution of powers of *x* followed by their combination is expected to be advantageous and better than the evolution of the complete solution function directly using all resources in one single stage. Additional reasons for using a flavor of piecewise evolution in multiple stages are the memory and fabric constraints which encourage this approach taken here.

Evolutionary Algorithms

Evolutionary algorithms rely on the building block hypothesis and questions about granularity of the building blocks used to build the candidate solutions are of prime importance when designing the chromosome representation and GA for the problem at hand. The most important considerations in the design of GAs are the genetic representation, fitness function and its complexity, selection scheme for the crossover function, mutation rate and adaptability of its rates and functionality to overcome stasis.

Various selection schemes are suggested for the crossover function. Selection schemes that favor crossover only among the fitter individuals result in premature convergence and though suitable for evolution in small search spaces, it is quite ill-behaved for larger search spaces. This was verified in this work, by starting with the evolution of 2:4 decoders which would converge to a decent solution (12 out of 16 outputs correct) within the first 100 generations and not improve at all owing to excessive presence of certain genes only, whereby at least one other important gene was always discarded in the selection process. Fitness proportional selection scheme is better and selects individuals for crossover with probability proportional to their respective fitness. This approach too suffers from some inadequacies. The work done by Annie S. Wu, et.al [12] elaborates on the commonly used selection schemes for crossover and introduces a fitness uniform selection scheme which preserves fitness diversity across generations and hence virtually guarantees good solutions with a well regulated convergence rate. This approach though, excellent for non-deceptive problems, doesn't perform well for deceptive problems. Tournament selection [34] is one popular scheme which is capable of producing good solutions with optimal convergence rates. In this work, as it is difficult to identify if the problem is deceptive or not, a variation of binary tournament selection with ideas inspired by fitness uniform selection scheme have been used.

With regards to fitness functions, the major drawback is with the complexity of evaluation and hence several approaches to tackle these problems have been presented. The paper by Haddow [11] provides an overview of the field of Evolvable Hardware (EHW) and pinpoints to the divide and conquer strategy that aims at simplifying the fitness evaluation space. Modularization and development of fitness functions on its basis are able to overcome scalability challenges in EHW to a very good extent, but determining modularity and determining fitness functions for the genes desirable to be retained is a highly involved and almost always a non-trivial task. This work ensures modularity and utilizes individual fitness functions for the same to ensure simplification of the computation process to enhance computational accuracy.

Royal Road functions [8] and experiments on them [9, 13] have revealed that, in general, it is best to combine building blocks that represent the finest granularity available for manipulation than coarser blocks owing to the fact that premature convergence has a very high probability of occurring in the latter. This is found to be essential in preserving genetic diversity as identified in [13] for fixed populations begin evolved. Thus, in order to determine series expansion to approximate a solution function, it is desirable to determine coefficients for each power of the independent variable being combined, separately than to attempt to find coefficients for groups of these terms. Thus, in this work, the coefficient prediction algorithm attempts to find coefficients for each power in consideration and combines them to obtain a decent approximation of the solution function. The coefficients are weights and CaDR may be viewed as a GA handling weights and develops on some of the ideas presented in [22] on weighted GA for multi-objective optimization.

CHAPTER THREE: PROGRAMMABLE SYSTEM ON CHIP

PLD Architecture

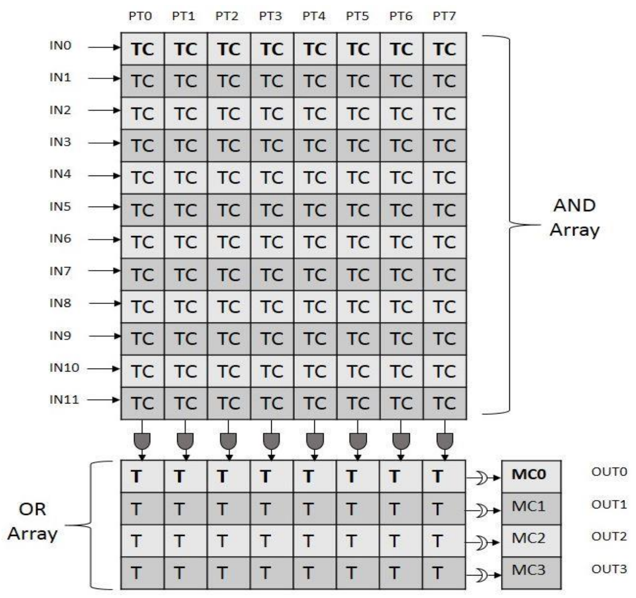


Figure 4: Simplified PSoC PLD Architecture adopted from [6]

Each PLD consists of an AND array with 12 input terms (IT) producing 8 product terms (PT) and an OR array that can programmatically access the 8 PTs to produce 4 output terms accessible either as combinational or registered outputs based on configuration written into each of the macro-cells (MCs) that control the nature of the output. This architecture is referred to as 12C4, where 12 stands for the number of input terms, C indicates that the PTs are constant and are accessible to the OR array and 4 indicates the number of output terms emerging from the OR array. Each input term in the AND array may be asserted as true or complement input, thereby contributing to the corresponding product in that fashion. Each term in the OR array may or may not be asserted, thereby allowing different logic combinations based on product terms arriving from the AND array.

Macrocell Architecture

The outputs produced by the logic combinations in AND and OR arrays appear in macrocells that have carry in and carry out signals and provisions for XOR feedback, set/reset select, true or inverted registered output and output bypass that can be configured to have combinational or registered outputs. Configuring Macrocells to produce pure combinational outputs requires setting constant to logic 0 (this produces the output from the OR gate as is) and the output bypass for the macrocell under consideration to logic 1 (this multiplexes out the combinational output only).

Experimental Configuration

LUT instantiation and wiring

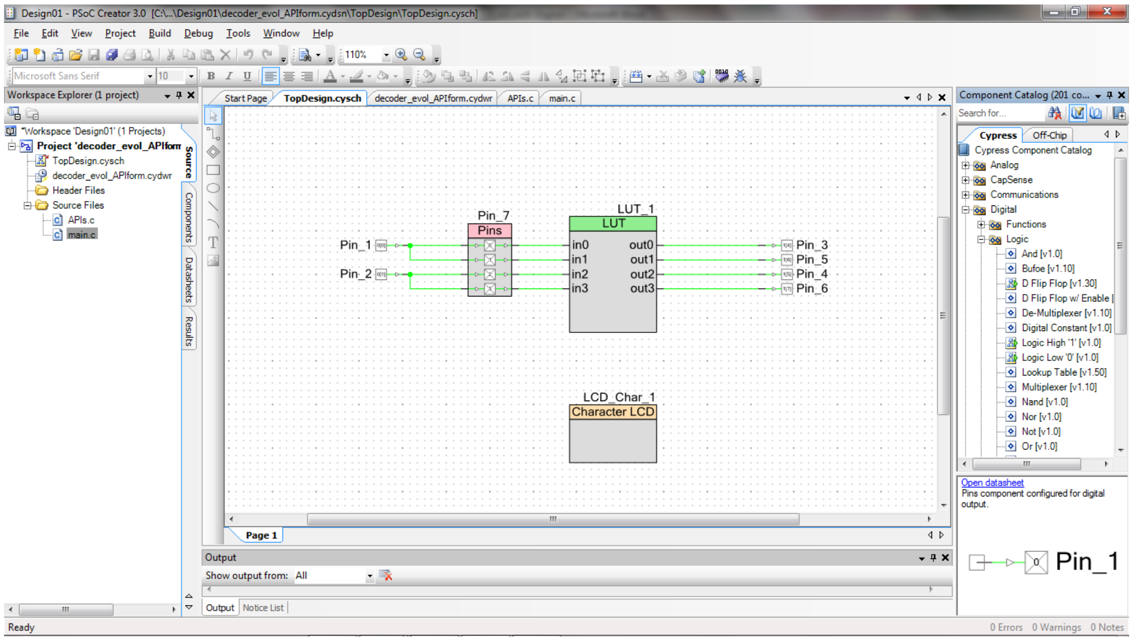
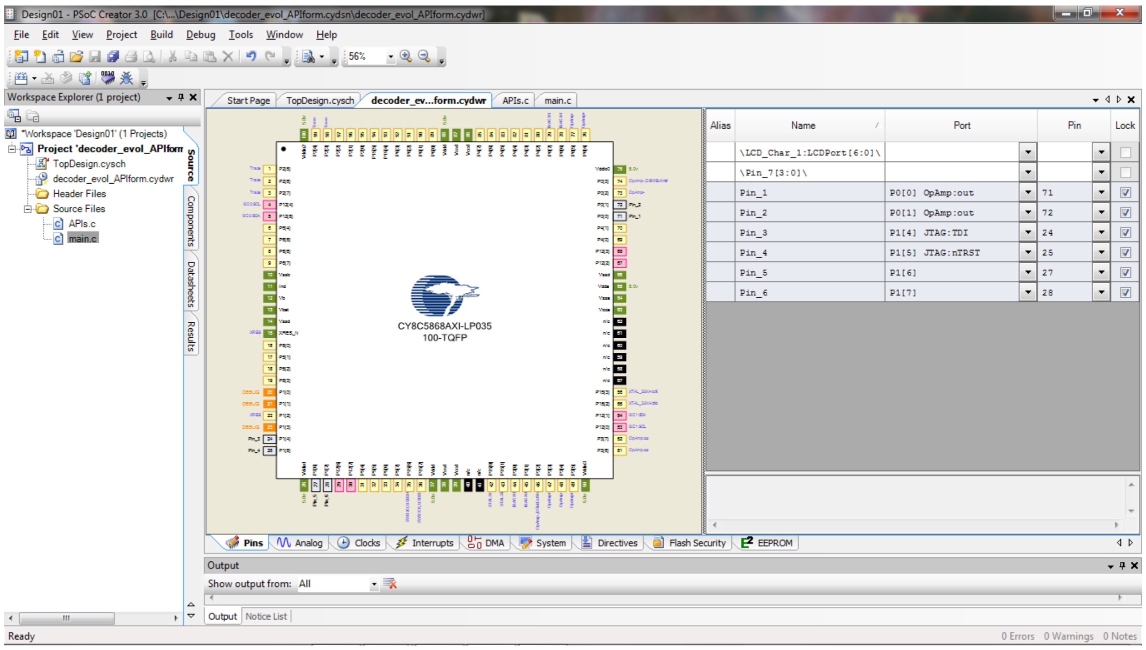


Figure 5: Schematic showing instantiated resources and connections

PSoC has no LUTs in hardware and the LUTs instantiated in the schematic are implemented in the PLDs. As illustrated in the above figure, a bus of pins maybe created anywhere in the schematic to connect input and output pins to different LUTs in the schematic as desired. This can be done only in the beginning, while boot-loading the code to the PSoC device though. The total number of LUTs that can instantiated for use is limited by the design space available in the schematic editor.

Force Directives – Pins



Design01 - PSoC Creator 3.0 [C:\...\Design01\decoder_evol_APIform.cydsn\decoder_evol_APIform.cydwr]
File Edit View Project Build Debug Tools Window Help
Workspace Explorer (1 project)
Workspace 'Design01' (1 Projects)
Project 'decoder_evol_APIform'
TopDesign.cysch
decoder_evol_APIform.cydwr
Header Files
Source Files
APIs.c
main.c
Start Page
TopDesign.cysch
decoder_ev...form.cydwr
APIs.c
main.c
CY8C5868AXI-LP035
100-TQFP
Pins
Analog
Clocks
Interrupts
DMA
System
Directives
Flash Security
EEPROM
Output
Show output from: All
Output
Notice List
Ready
0 Errors 0 Warnings 0 Notes

Alias	Name	Port	Pin	Lock
	\LCD_Char_1:LCDPort[6:0]\			☐
	\Pin_7[3:0]\			☐
	Pin_1	P0[0] OpAmp:out	71	✓
	Pin_2	P0[1] OpAmp:out	72	✓
	Pin_3	P1[4] JTAG:TDI	24	✓
	Pin_4	P1[5] JTAG:nTRST	25	✓
	Pin_5	P1[6]	27	✓
	Pin_6	P1[7]	28	✓

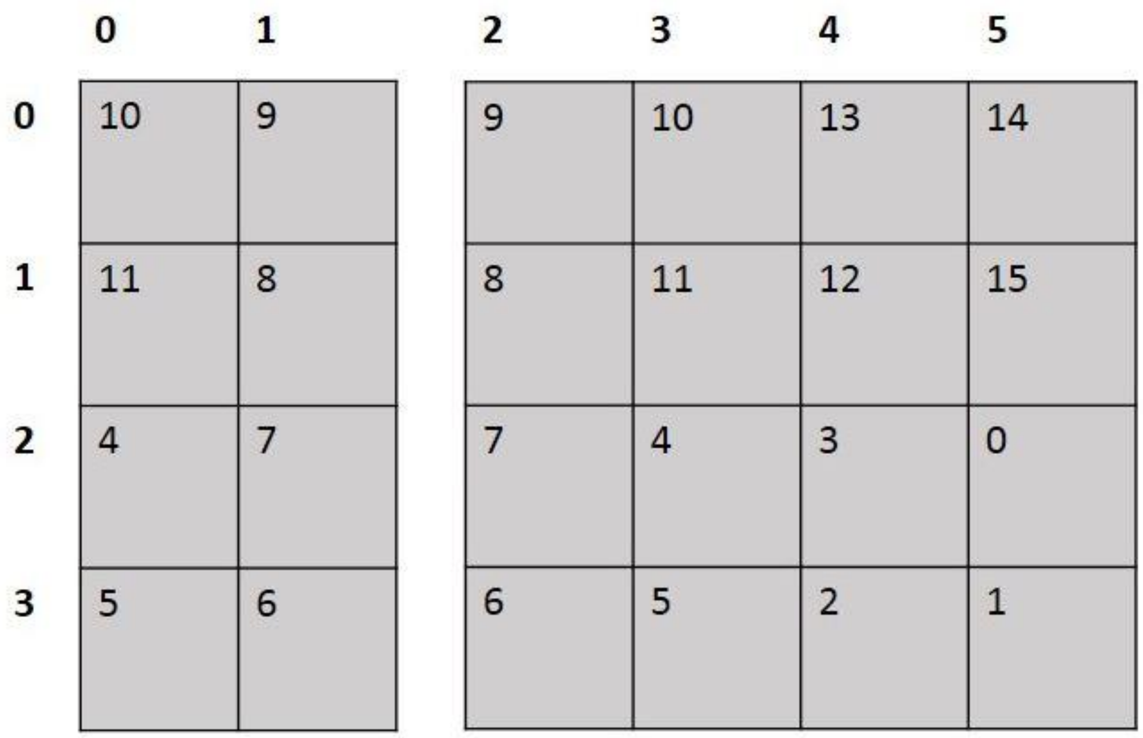
Figure 6: Design Wide Resources file showing pin assignment

For the simple design shown in the schematic in figure 5, specifying pin usage and is done as in the figure 6. For each pin used in the schematic, its placement may be directed in the .cydwr file which is also configured for many other components that may be used.

Forcing resource utilization and implementation of LUTs in the PLD space

Placement directives, such as force directives may be used to force placement of LUT implementation to any specific PLD, which is analogous to the PROHIBIT command in the Xilinx User Constraint File [7]. Thus, through register writes, the logic function of each LUT implemented in each PLD is reconfigured in runtime. Hence logic configurability of each PLD is achieved through code eventually.

LUTs in the schematic are implemented in the PLDs. The number of input signals/bits available to each LUT in the schematic is restricted to a maximum of 5 by the IDE. The number of output signals/bits that can be forced out of a single LUT is 8. Each PLD however, only produces 4 output bits. In order to force the implementation of a LUT (instantiated in the schematic), the output bits of the LUT need to be placed in the PLD of interest, 4 at a time for each PLD. This is done by using the ForceSignal directive to place the corresponding output signal names to the PLD of interest. This imposes the restriction that each PLD can only accept 5 input bits and produce 4 output bits, since each LUT has an input width of only 5 bits. Hence, in the design, each LUT with 4 output bits maps to a single PLD. UDBs are identified by a pair of indices which indicate the row and column number of the UDB in the following ordering grid:

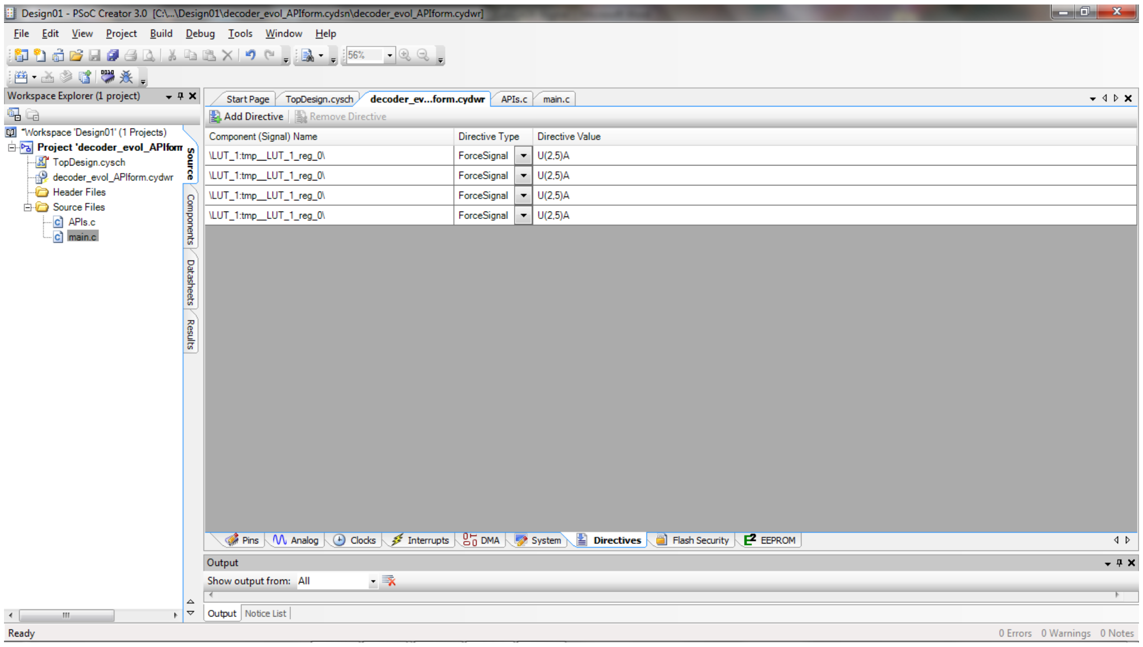


	0	1	2	3	4	5
0	10	9	9	10	13	14
1	11	8	8	11	12	15
2	4	7	7	4	3	0
3	5	6	6	5	2	1

Figure 7: UDB mapping scheme within PSoC build files adopted from [36]

The above chart indicates how UDBs are indexed in PSoC. There are two banks of UDBs available on PSoC – bank 0 and bank 1. Each UDB has a corresponding row and column number as indicated above, that uniquely identifies it irrespective of bank number. Each such UDB has two PLDs that are indicated by A and B. The general syntax is thus U(row_number, column_number)PLD_index, where PLD_index could be A or B depending on whether PLD 0 or PLD 1 is being referenced. Thus, for instance, if we are to reference PLD 0 (or A) of UDB 0, it would be written as U(2, 5)A. This information is used in forcing signals that arrive as outputs of LUTs to specific PLDs. In order to do so, the correct signal name is to be obtained from the report file generated when the project is built. Thus when attempting to find the signal name (also called signal alias in the report), the project with a single LUT in the schematic is built without forcing directives. This randomly places the signals across different PLDs in the fabric. From the report file, the signal alias name is identified and then used for forcing corresponding signals. The general format of a signal alias is as follows:

\LUT_x:tmp__LUT_x_reg_y\. Here x represents the number(/name) of the LUT as it appears in the schematic when instantiated and y represents the output bit/signal number arriving from that LUT. To force a signal to a particular PLD, two components are necessary– signal/alias name, UDB and PLD index obtained by referring to the chart above and following the referencing rule as detailed above. In addition to these, the force signal directive is to be chosen from the dropdown to force one output's placement in a specific PLD of choice. The following example shows how forcing is done for LUT_1 instantiated in the schematic:



Design01 - PSoC Creator 3.0 [C:\...\Design01\decoder_evol_APIform.cydsn\decoder_evol_APIform.cydwr]
File Edit View Project Build Debug Tools Window Help
Workspace Explorer (1 project)
*Workspace 'Design01' (1 Projects)
Project 'decoder_evol_APIform
TopDesign.cysch
decoder_evol_APIform.cydwr
Header Files
Source Files
APIs.c
main.c
Start Page
TopDesign.cysch
decoder_ev...form.cydwr
APIs.c
main.c
Add Directive
Remove Directive
Pins Analog Clocks Interrupts DMA System Directives Flash Security EEPROM
Output
Show output from: All
Output Notice List
Ready
0 Errors 0 Warnings 0 Notes

Component (Signal) Name	Directive Type	Directive Value
\LUT_1:tmp__LUT_1_reg_0\	ForceSignal	U(2,5)A
\LUT_1:tmp__LUT_1_reg_0\	ForceSignal	U(2,5)A
\LUT_1:tmp__LUT_1_reg_0\	ForceSignal	U(2,5)A
\LUT_1:tmp__LUT_1_reg_0\	ForceSignal	U(2,5)A

Figure 8: Design Wide Resources file showing force directives for PLD placement

In order to add additional PLDs with the restrictions as mentioned above it is required to extend these placement directives, to all the PLDs of interest, keeping in mind the placement of all pins instantiated in the schematic and the way they are interconnected in the schematic as well. Following a similar procedure, the signal names are identified and are forced to the corresponding PLDs of interest.

Constraints in Digital Configuration of 1 PLD:

Several constraints were encountered in the configuration of a single PLD of a specific UDB which implemented a single LUT placed in the schematic. The major constraints were as follows:

Number of input terms that can simultaneously enter a PLD

Only 5 inputs could be fed to a single LUT and use of force directives wasn't able to achieve placement of inputs to a PLD, but only the placement of output signals to a specific PLD. Thus, only the corresponding input signals would arrive at the input end of the PLD. Though, there are 12 inputs available in a PLD, when placement is forced by the user, only as many inputs as an LUT supports can be accommodated in a PLD. This gives a constraint of being able to support only 5 input terms out of the 12 input terms available in the PLD architecture.

Selection of input lines used by the PLD in implementing the LUT

The input lines that are actually used by the PLD are randomly chosen out of the 12 input lines available to the PLD at design time, i.e. when the boot-loader programs the PSoC device.

This constraint is made more severe by the fact that the other input lines are in high Z state and hence if one of those lines are asserted in the OR array, then the corresponding output from the macrocell is also in high Z state. Thus, in order to avoid driving outputs to high Z state, the actual input lines initialized for use by the schematic need to be identified after the schematic configuration is built from the boot loaded code. These input lines are referred to as active lines and custom code was developed to identify these active lines by searching in the address space corresponding to the PLD(s) being used.

Storing the configuration:

PSoC's g++ compiler doesn't support Boolean type variable and hence several bit manipulation and type conversion operations had to be performed, to carefully extract out configuration from each register and store the same in the chromosome for that individual.

Writing back configurations to the chromosomes:

Following the problem with inactive input lines and their potential to drive output to high Z state if asserted, reconfiguration of the PLD also involves writing only to the active lines identified. This restriction is limited only to the AND array, while the OR array has no such restrictions when 4 output terms are derived from the same. It has to be noted that if there are fewer product terms than 8 arriving at each line of the OR array, the other product terms if asserted later can result in high Z output for that line. Thus special care is taken to evolve the whole PLD with non-coding sequences participating in evolution, but only the configuration of the active lines being written back in both arrays. This constraint provides an opportunity to test the effect of evolution of introns and their effect in convergence of the overall GA as and when required.

Writing back configuration to the corresponding registers:

Owing to the limitations of g++ compiler and the possibility of driving outputs to high impedance, writing back configuration to the registers, involved bit manipulations and type conversions and writing only if the corresponding active line were asserted in the chromosome.

CHAPTER FOUR: DIFFERENTIAL DIGITAL CORRECTION

Differential Digital Correction Technique

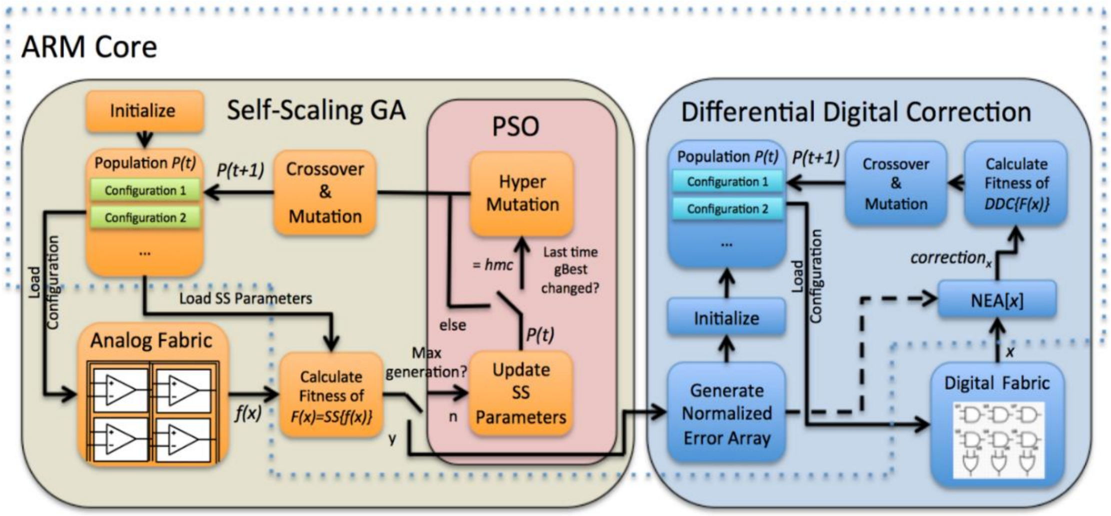


Figure 9: DDC and Self-Scling GA schematic for analog-digital evolution [35]

The solutions obtained from analog evolution can rapidly approximate the desired solutions, although their accuracy is limited and susceptible to imprecision. The results of the analog stage are obtained and written to files. These files are then included in DDC code as header files for subsequent digital evolution. The Differential Digital Correction (DDC) technique selects a correction factor for each test input from a Normalized Error Array (NEA), which contains fractions of the maximum analog error. In order to build the NEA, the deviations of the analog outputs from the oracle are obtained and compared to obtain the maximum deviation or the maximum analog error. Following this, the NEA is constructed by generating fractions of the maximum analog error upto 256 levels with positive and negative correction factors adjusted to be able to provide correction to the case corresponding to maximum error as well, as shown below:

```
      for (i=0;i<256;i++){
              if (i < 128)
                  NEA[i] = -((float)(127-i)/127.0) *
winner.maxDeviation;
              else
                  NEA[i] = ((float)(i-127)/128.0) *
winner.maxDeviation;
          }
```

This DDC utilizes correction factors quantized to 256 levels and hence evolves PLDs to produce an 8-bit mapping for the 256 test inputs to reduce the error at each test point to the best possible extent. The DDC fitness is evaluated as follows:

$$\textit{for } i := 0 \textit{ to } 255$$
$$\textit{fitness} \mathrel{+}= \textit{oracle}[i] - \textit{out}_{\textit{analog}}[i] - \textit{NEA}\ [D]$$

The NEA containing correction factor elements called normalized differences is indexed by the instantaneous 8-bit output D, to realize the 8-bit output mapped to one of the 256 values which provides a correction factor of proportional magnitude. The D values so obtained are stored as the LUT configurations corresponding to the test inputs for which they appear. The fitness function thus represents cumulative error which is minimized through a GA which intrinsically evolves the digital fabric.

Representation Scheme and Chromosome Structure:

In order to evolve a single PLD, its configuration bits are encoded into a chromosome which genetic operators can act upon. The configuration bits consist of the following: active lines, AND array parameters and OR array parameters. Input lines that are active, i.e. not in high impedance state, in the actual implementation are determined from the register configuration immediately after booting and are marked as active lines; other lines cannot be written. There are 12 input lines for the AND array and four output lines for the OR array.

Active Lines:

Input lines that are active in the actual implementation are determined from the register configuration immediately after booting and are marked as active lines. There are 12 input lines for the AND array and 4 output lines for the OR array. Hence we have a total of 16 active lines depending on the usage of resources determined from the schematic at the time of booting. Active lines are copied as is for all individuals generated for a particular PLD and remain, for all practical purposes, a constant gene of the chromosome.

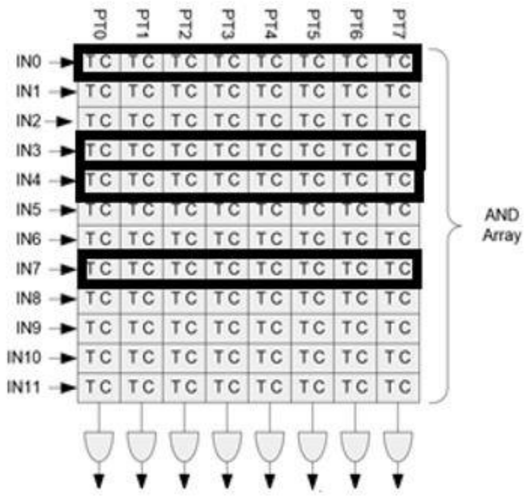


Figure 10: Example of selection of active lines by IDE for logic implementation. Adopted from [6]

AND array parameters:

For each input line that is active, the input bit can be asserted as true or complement input in each of the product terms that will be calculated for each column in the array. There are 12 input lines with 8 columns each corresponding to the 8 product terms. For each column corresponding to an input line, the bit being considered can be asserted as true (0) or complement (1) and are encoded only for active lines in the same fashion in the chromosome. In order to encode the configuration, the configurations of the active input lines alone are to be recorded and hence up to eight such 32-bit integers store the configuration from the corresponding registers for two PLDs constituting a Universal Digital Block (UDB) in the fabric.

Initially, the bits of every input line in the AND array were considered and encoded for evolution. This resulted in larger evolution times with more time and resources being spent on introns or unexpressed genes. Additionally it also posed a greater risk of performing erroneous register writes. This process was found to be wasteful and hence introns were pruned out in encoding and subsequently in evolution. The current structure for evolution contains only the extrons and is hence much less resource and time intensive.

OR array parameters:

Like the AND array, for the OR array, each of the product terms arriving from the AND array may be asserted (1) or not asserted (0) and is encoded likewise in the chromosome. We have four such 16-bit integers that store the configuration from the corresponding registers. Hence we have an array containing 32 values for the OR array.

The chromosome for each PLD is encoded as a structure with three arrays: AL (active lines), AND array parameters and OR array parameters representing configuration bits as described above and effectively describing contents of one UDB.

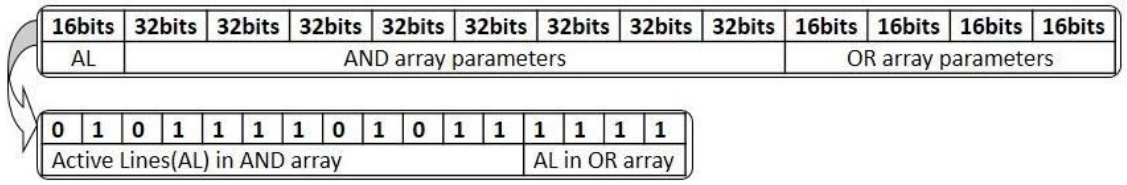


Figure 11: Chromosome representation for each UDB evolved in DDC

<u>DDC Genetic Algorithm</u>

Generation of random individuals and the population size:

Individuals with the chromosome structure described above are randomly generated initially. The configuration of each of these individuals is programmed and the output is tested against various inputs applied to the board to evaluate fitness. The elite individuals are chosen for crossover with other individuals and their crossover probabilities are assigned based on their fitness. With regards to successful generation of random individuals, it was realized that seeding the random number generator appropriately was an important step. Seeding very often or too less often resulted in poor randomness and hence in the diversity of the populations in the initial or subsequent generations, accordingly as the case was observed to be. Also, a population of sufficiently large size was necessary to ensure that the search space was large enough not to lead the GA to a local extremum. Various population sizes were attempted given the memory limitations of the fabric. A population size of 80 was deemed suitable for evolution.

Elitism and selection of individuals for crossover:

An elitism of 2 is implemented where the two best fit individuals don’t undergo mutation but have a high chance of crossing over with other individuals. In order to select individuals for crossover, binary tournament selection is implemented. Two adjacent individuals in the population array are checked for their fitness and the fitter one is chosen and stored in a separate array containing winners of the tournament in every such pair of individuals compared.

Crossover:

The crossover operation is performed separately in each iteration for the AND arrays and OR arrays owing to their inherent functional differences. Crossover operation entails choosing a random crossover point (single point crossover) and copying the configuration bits from one end to the crossover point, respectively from each parent to generate the configuration bit-stream for the offspring. Crossover points were chosen to be at the boundaries between the eight 32 bit numbers in the AND array and the boundaries between four 16 bit numbers in the OR array respectively. Tournament selection is done with a tournament size of two and constitute 40 of the total 80 individuals per generation. Single-point crossover is then performed between one of these 40 fitter individuals and another individual randomly chosen from the whole population. The individuals so generated replace a fraction of the individuals in the population, while the remainder are from the fitter half of the population selected by binary tournament selection. The fraction of individuals that would be replaced with new individuals is a parameter tunable at the beginning of evolution. The fittest individual is marked to be preserved intact while other individuals undergo mutation.

Various schemes have been attempted for crossover. The first (and very ineffective scheme) involved choosing a class of elite individuals –fittest, second, third and fourth fittest individuals- and preforming crossover only between them to generate 6 new individuals which would replace the other 6. This resulted in a lack of genetic diversity and very quick convergence to a solution that wouldn't improve any further owing to very similar genes.

To improve performance, the crossover pattern was modified to pick the fittest and the second and perform crossover with probabilities of 100% and 50% and with a probability of 30% with the rest. This resulted in a slight increase in performance, but wasn't encouraging enough. The probabilities of crossover were then modified to 10%, 10% and 80% respectively for the crossover with the fittest, second and the rest, respectively. Again, the performance improved but only marginally.

The probabilities of crossover were modified to see improvements in performance and justifiably, a proportional search scheme was chosen for implementation. This scheme assigns a fixed probability of crossover for each individual based on its relative fitness value among all individuals. The probability of being chosen for crossover is essentially the ratio of the fitness of that individual to the maximum possible fitness that can be achieved. Individuals so chosen produce new individuals whose fitness is evaluated and subsequent replacement of less fit individuals in the older generation occurs. New individuals so generated replace the older individuals while the fittest and the second fittest individuals of each generation are retained. This approach resulted in good improvements in performance.

Considering the success of the proportional search scheme above and the need for fitness diversity while still finding the problem of evolution not entirely classifiable as non-deceptive, the fitness uniform selection scheme as discussed in Dr. Wu’s work [12] couldn’t be adopted as is. Tournament selection was selected to offer better fitness diversity, while the scheme for replacement of individuals involving maintaining all or a fraction of the fitter individuals selected in the tournament tries to emulate the advantages of the proportional search scheme and possibly overcoming the problem if deception where it appears. Maintenance of an elitism of 2 helped with debugging and code development, especially when concerns of losing the best individual appeared.

Mutation:

Mutation is implemented as a simple bit flip with a finite probability of occurring at any position on the chromosome. All but the two most elite individuals undergo simple bit flip mutation with a default mutation rate of 0.1% per bit in the chromosome. In order to deal with a potential stasis condition several schemes were attempted. The mutation rate was ramped up by 1 (or a small increment) percent every few hundred iterations in a fixed fashion. The number of iterations after which the rise occurred was tuned to determine a suitable empirical number, but this approach failed to produce good results for all the different circuits being evolved. Unusually high mutation rates were also used in hopes of improving the performance.

Mutation rate was observed to be a crucial factor in the performance of the DDCGA with adaptive mutation being very useful in overcoming stasis. Other mutation altering strategies discussed before weren’t as effective in improving performance. A condition of stasis is detected and reported if the best fitness achieved hasn’t changed in 50 iterations using a single control bit.

The difference between average fitness and best fitness achieved is compared and encoded likewise in the stasis information to decide whether mutation should be enabled at the default rate or at an incremented rate, where increments in steps of 0.01 improved performance. Faster rates destroyed good solutions and slower rates couldn't help maintain enough diversity to search for better solutions.

Improving performance of the GA and overcoming stasis conditions:

In order to improve the performance of the GA, initially parameters were tuned using different crossover parameters. Choosing boundaries that match with the chromosome structure better proved to be beneficial. Also, choosing a random point for crossover instead of a fixed point on the chromosome helped improve the search space. Initially, evolution was attempted with crossover at mid-point only. This was followed by attempts within a fixed set of points. These experiments produced poor results as the genetic diversity was severely restricted. Completely random crossover points with no strictly defined boundaries were then attempted and it was noticed that good solutions disappeared very soon after their appearance owing to extreme variations. A fair balance was achieved when crossover boundaries matched with the chromosome definitions introducing randomness within reasonable limits.

One other technique to add to genetic diversity involved creating and introducing some random individuals into the gene pool after a few hundred iterations when a potential stasis like situation was expected. This helped improve the randomness and hence the search space in some cases. Unlike mutation, however, the randomness introduced by addition of new individuals easily went beyond control when more than 2 individuals were introduced at a high frequency (less than

50 generations). Very controlled addition of a single random individual resulted in performance mostly at par with the case where such an addition weren’t being made. Like in the case of mutation, it is expected that performance could improve if addition of random individuals were performed on demand with an adaptive flavor to resemble what is popularly called hyper mutation. This approach is still under investigation.

Evolving individuals by building chromosomes only form the extrons resulted in quicker evolution times and gave better scope for scalability and extension to multi-PLD chromosome. Evolution with active lines alone results in better utilization of memory and is largely used. Evolution with introns is attempted as an additional exercise to study their benefits.

PLD allocation and utilization:

Differential Digital Correction GA (DDCGA) reconfigures the digital fabric. The DDCGA evolves four PLDs of two chromosome sets per individual: one each for the AND and OR arrays, and one set of individuals for each pair of four PLDs, given constraints of the digital fabric. Use of fewer PLDs was found to provide insufficient resources for evolution, while the use of a larger number of PLDs didn’t produce better results than with 4 PLDs. Also, considering the memory constraints of PSoC, parallel evolution of more PLDs introduced issues with randomness and loss of tractability owing to insufficient memory to have a sufficiently large population for all PLDs to evolve, thereby resulting in a drop in performance once the population size was affected by the resource allocation for all PLDs for each step in evolution.

Cascaded DDC:

Following the success of DDC in evolving powers of *x* such as square, cube, square root and cube root, a few other powers of *x* were evolved too. These included the analog computational circuits for fourth root and fourth powers of *x* which were then refined by DDC. Circuits to compute zeroth and first powers of *x* were trivial and required very short evolution times to obtain corresponding analog computation circuits, followed by short DDC stages to refine them. Thus, different powers of *x* starting from fourth root through fourth power of *x* were evolved in a cascaded fashion, one following the other and the arrays containing the outputs of each of these stages were stored separately for the coefficient prediction stage to act on the same.

CHAPTER FIVE: COEFFICIENT PREDICTION

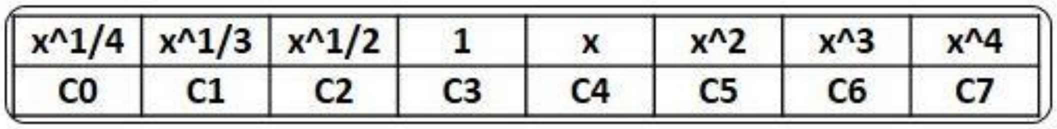
Coefficient Prediction Technique

Following the cascaded DDC stage, the coefficient prediction (CP) stage attempts to predict coefficients of the evolved circuits that can be combined through addition to approximate the available range of functional values of the independent variable, *x*, to best represent the composite function in terms of the independent variable. Coefficients are randomly assigned initially to each of the eight powers of the independent variable, evolved by cascaded stages of DDC starting from the most accurate to the least accurate power (smallest to largest). The coefficient prediction phase first performs range scaling of the computed oracle by dividing all the values in it by a scaling factor that is determined from overflow considerations in calculations involving multiplication of the largest coefficient assignable and the largest value of biggest power of *x* used and the sum of all such products for all powers of *x* that would not result in an overflow in the results computed. Once range scaling is performed, CPGA randomly initializes a population with random coefficients for each power of *x* and performs fitness evaluations, selects good individuals, performs single-point crossover at a randomly determined point to exchange genetic material followed by mutation to evolve a suitable set of coefficients that act as weights in combining different powers of *x* to produce functions that approximate the solution function as closely as possible given the number of iterations after which it is terminated. The chromosome for *x*^0 function is trivial and hence in order to reduce overall evolution time, it isn’t evolved, but the values are directly written to the corresponding array.

Coefficient Prediction Genetic Representation

Chromosome Structure:

The chromosome is a simple array of 8 floating point numbers, each of which is within the range determined after range scaling. Bitwise manipulations of the chromosome aren’t performed from code in any step in the CPGA. Each chromosome is equivalent to set of coefficients for all the data points for all powers of *x*. The following is the chromosome for one individual:



x^1/4	x^1/3	x^1/2	1	x	x^2	x^3	x^4
C0	C1	C2	C3	C4	C5	C6	C7

Figure 12: Chromosome representation for coefficients evolved in CP

The chromosome and the corresponding correlations between the coefficients and various powers of *x* are as indicated above. Each of the coefficients C0 through C7 in the above chromosome is within the scaled range obtained after performing range scaling.

Coefficient Prediction Genetic Algorithm

CPGA Algorithm Flow

Range Scaling:

- Determine largest value of the largest power of *x* from cascaded DDC
- Determine the range of values for each coefficient being evolved from the following relation:
- C_range*A_largest*8 = range(float32)
- Or C_range = range(float32)/(A_largest*8)

Initialize Population:

- Randomly pick values using rand()%C_range for all the coefficients C0, C1, C2, C3, C4, C5, C6, C7

- Initialize 80 individuals or more with randomly chosen coefficients

  <u>*Dynamic Range Adaptation:*</u>
- Check if best fit individual has a total error greater than 100
- Halve C_largest and re-initialize population with new coefficients in smaller range except first 10 individuals
- Continue narrowing search space by reducing C_largest till a comfortable range is found
- *Range forcing*: Further narrow range by halving C_largest for half the population after error goal is achieved

  <u>*Fitness Evaluation:*</u>
- Store the values of {oracle[i]-(output[i]-NEA[D_array[i]]} in a separate array called results _DDC_C0 for x^(1/4) and likewise for other powers: results _DDC_C1, results _DDC_C2, results _DDC_C3, results _DDC_C4, results _DDC_C5, results _DDC_C6, results _DDC_C7 for C1 through C7 as well respectively
- Multiply C0* results _DDC_C0 and store in array S0
- Multiply C1* results _DDC_C1 and store in array S1
- Multiply C2* results _DDC_C2 and store in array S2
- Multiply C3* results _DDC_C3 and store in array S3
- Multiply C4* results _DDC_C4 and store in array S4
- Multiply C5* results _DDC_C5 and store in array S5
- Multiply C6* results _DDC_C6 and store in array S6
- Multiply C7* results _DDC_C7 and store in array S7
- Add S0+S1+S2+S3+S4+S5+S6+S7 = ΣS
- Error = S- ΣS
- Fitness = sum(Error)

  <u>*Crossover:*</u>
- Select elite 2
- Binary tournament selection
- Single point crossover with tunable number of individuals to be replaced in the population and fitter individuals pool as in DDC

  <u>*Mutation:*</u>
- With probability of mutation Pm, add rand(-1, 1) (is this the best way to mutate them??) to each gene of the chromosome
- Leave out elite 2 unaltered

  <u>*Continue GA:*</u>
- When stasis is detected adapt mutation rates
- Introduce hypermutation as introduction of random individuals if necessary on demand to improve genetic diversity

Terminate and Present Solution:

- Terminate when maximum number of iterations is reached (or if the fitness goal is achieved)
- Print coefficients and error of the best fit individual obtained

Range Scaling

As in indicated in algorithm flow above, the coefficients C0 through C7 are all floating point numbers. However, each is multiplied by the corresponding power of *x* to calculate the solution at every test point. Consider the following calculation of the solution at a test point i:

```
Sol[i] = C0*x^(1/4)[i] + C1*x^(1/3)[i] + C2*x^(1/2)[i] + C3 +
        C4*x^[i] + C5*x^2[i] + C6*x^3[i] + C7*x^4[i]
```

For the above calculation at a test point i, it is seen that the sum of products must be a value within the range of Sol[i], which is a 32 bit floating point number. In order to ensure that all coefficients have the same scaled range for easy scalability of the final solution obtained, the scaling factor, A_largest is determined by dividing overall range of values of float32 by 8*x^4[255]. Coefficients are randomly picked within this scaled range to initialize the population for evolution.

Generation of random individuals and the population size:

Individuals with the chromosome structure described above are randomly generated initially. The configuration of each of these individuals is programmed and the output is tested against various inputs applied to the board to evaluate fitness. The elite individuals are chosen for crossover with other individuals and their crossover probabilities are assigned based on their fitness. With regards to successful generation of random individuals, it was realized that seeding the random number generator appropriately was an important step. Seeding very often or too less often resulted in poor randomness and hence in the diversity of the populations in the initial or subsequent generations, accordingly as the case was observed to be. Also, a population of sufficiently large size was necessary to ensure that the search space was large enough not to lead the GA to a local extremum. Various population sizes were attempted given the memory limitations of the fabric. A population size of 80 was deemed suitable for evolution.

Dynamic Range Adaptation:

When arbitrarily large coefficients are chosen for evolution, the total error is observed to be arbitrarily large as well and the search space in this case is too large for a GA to be able to navigate through in a fixed number of generations. Mutations and crossovers aren't sufficient to sufficiently alter the coefficients to be able to reduce the search space and obtain fitter individuals. Thus a technique to dynamically alter the range of coefficients defining the chromosome is used. Every iteration where the total error is larger than 100, the range of the coefficients chosen for all but the first ten individuals is halved. Crossover and mutation are then performed. This results in production of fitter individuals or otherwise depending on the nature and characteristics of the solution function. Range adaptation reduces the range of exploration of coefficients to a more likely range for all powers of *x* involved. Range forcing is also attempted when the total error is less than 100, in which case the coefficients of a quarter of the individuals is halved. Range forcing hasn't been able to produce significant results for sin(*x*) but may produce better results for other functions. Dynamic range adaptation is essential in order to ensure meaningful evolution of coefficients and spans the first few iterations in CPGA.

<u>Elitism and selection of individuals for crossover:</u>

An elitism of 2 is implemented where the two best fit individuals don’t undergo mutation but have a high chance of crossing over with other individuals. In order to select individuals for crossover, binary tournament selection is implemented. Two adjacent individuals in the population array are checked for their fitness and the fitter one is chosen and stored in a separate array containing winners of the tournament in every such pair of individuals compared.

<u>Crossover:</u>

The crossover operation is performed by choosing a random crossover point between two of the eight coefficients in the chromosome. Crossover operation entails choosing a random crossover point (single point crossover) and copying the configuration bits from one end to the crossover point, respectively from each parent to generate the coefficients defining the offspring. Tournament selection is done with a tournament size of two and constitute 40 of the total 80 individuals per generation. Single-point crossover is then performed between one of these 40 fitter individuals and another individual randomly chosen from the whole population. The individuals so generated replace a fraction of the individuals in the population, while the remainder are from the fitter half of the population selected by binary tournament selection. The fraction of individuals that would be replaced with new individuals is a parameter tunable at the beginning of evolution. The fittest individual is marked to be preserved intact while other individuals undergo mutation.

Following the success of tournament selection for DDC and for reasons concerning deceptiveness of the problem at hand and offering fitter individuals a higher chance to pass on their genes, as discussed in the crossover section of DDC, binary tournament selection was chosen for implementation.

Mutation:

Mutation is implemented in two flavors. The first and more prevalent flavor (also called normal flavor here) in the population is the addition or subtraction of a small random number generated between -4 and 4. All but the two most elite individuals undergo this form of mutation with a default mutation rate of 0.1% per bit in the chromosome. In order to deal with a potential stasis condition, the mutation rate was ramped up at a rate of 0.01% every iteration when under stasis, but the effects weren’t significant with respect to evolution of better solutions. In order to consider the possibility of a sign flip and its effect on the solution, a small number of individuals were randomly chosen from the population and a sign flip was performed for each coefficient of the chosen individual with the same probability of 0.1%. This flavor of mutation was less prevalent in the population and the number of individuals chosen for sign flip mutation was fixed at a maximum of 10.

Mutation rate was observed to be a crucial factor in the performance of the CPGA with adaptive mutation being very useful in overcoming stasis. Other mutation altering strategies discussed before weren’t as effective in improving performance. A condition of stasis is detected and reported if the best fitness achieved hasn’t changed in 50 iterations using a single control bit. The difference between average fitness and best fitness achieved is compared and encoded likewise in the stasis information to decide whether mutation should be enabled at the default rate or at an incremented rate, where increments in steps of 0.01 improved performance. Faster rates destroyed good solutions and slower rates couldn't help maintain enough diversity to search for better solutions.

Improving performance of the GA and overcoming stasis conditions:

In order to improve performance first different crossover boundaries were experimented with. Choosing boundaries that match with the chromosome structure better proved to be beneficial. Next, in order to pull the GA out of local minima for total error, the implementation of the first flavor or the normal flavor of mutation was changed based on stasis information. When stasis was detected, the size of the numbers added to or subtracted from the coefficient under consideration was ramped by proportional to the number of iterations of the GA elapsed. The increase in size was managed within a small range to prevent the GA from entering into a state of random search.

Scaling back to original range

Following the evolution of the final solution to the problem, the best fit individual's coefficients needn't be multiplied by the scaling factor, 8*A_largest, where A_largest is the biggest term in the largest power of *x* evolved. The coefficients were observed to be fairly small for most test functions attempted. The final solution contains the coefficients most closely approximating the solution function for the differential equation at hand.

CHAPTER SIX: CASCADED DIGITAL REFINEMENT

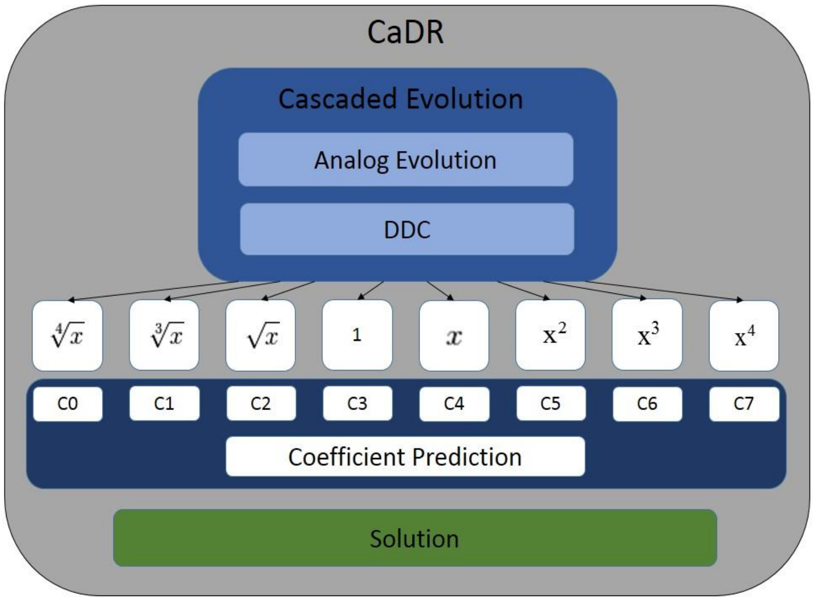
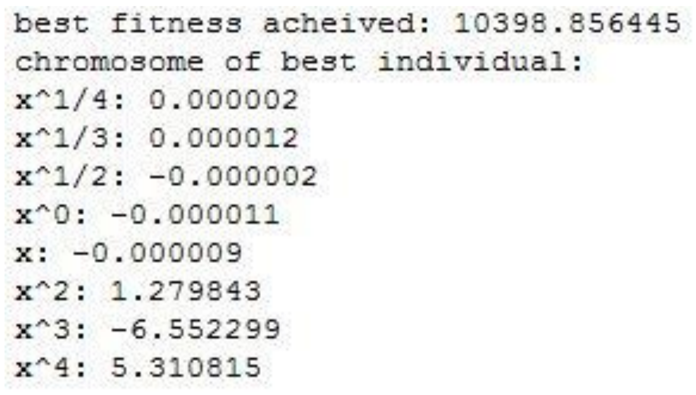


Figure 13: CaDR scheme for solving approximating solution function to a DE

Cascaded Digital Refinement (CaDR) technique essentially uses DDC technique in a cascaded succession, on the analog evolved solution, to evolve the powers of the independent variable, *x*, that would be used to approximate the solution of the ordinary differential equation whose solution is also a function of the independent variable *x*, by combining them with appropriate coefficients which are determined by the Coefficient Prediction GA (CPGA). DDC and CP together constitute the CaDR technique to numerically approximate the best solution to a given ODE with a known solution. The underlying assumption is that mathematical power series expansions such as Taylor series hold for any function of the independent variable *x* that is continuous and differentiable at very point. Though power series expansions typically use integral powers of *x*, it is conjectured here that use of fractional powers of *x* along with integral powers of *x* and predicting coefficients for all of them using CPGA would result in better solution to the ODE than mere integral or fractional powers of the independent variable alone. Given the accuracy with which DDC evolves different powers of *x*, it was noted that smaller powers of *x* were more accurately evolved owing to the precision inherently available from the analog section which the DDC further improves upon. This motivates use of fractional powers of *x* (less than 1) alone, but this conjecture isn't verified to produce closer approximations to the power series expansion of a function than complete use of integral powers alone. It is understandable that several more terms would be required to produce a decent approximation of the solution function of the ODE, when more fractional powers are in use, especially as for higher values of the independent variable *x*. Thus, in order to reduce error arising both due to the inherent error in larger integral powers of *x* evolved and also to reduce the possible errors for higher values of *x* when evolved using fractional powers alone, a suitable compromise between the two is attempted to produce maximum possible accuracy of CaDR, where the accuracy of CP stage is expected to depend on that of the building blocks produced in the cascaded DDC stages.

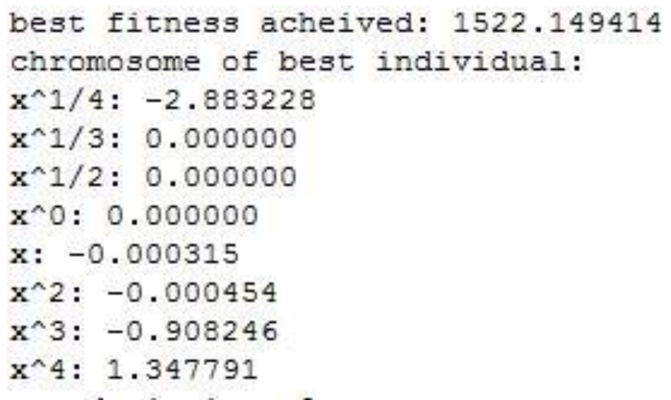
Evolving fractional powers from fourth root to square root of *x* followed by integral powers from square to fourth power of *x* via DDC is proposed for the first stage, namely cascaded DDC stage of CaDR. The circuit configurations to produce zeroth power of *x* is trivial and hence known. The second stage of CaDR, namely CP stage is expected to evolve coefficients of the powers of *x* considered, that would approximate the solution function to the ODE to the desired level of accuracy. To approximate the solution function with combinations of powers of *x* mentioned above, 8 different coefficients need to be determined. In order to proceed with the same, CP stage first considers issues of data overflow and hence implements range scaling to ensure that the coefficients may be randomly initialized within appropriate value ranges, to be able to combine these powers with the weights of their coefficients and to ensure that the respective errors of each individual in the population may be evaluated against the oracle. Once initialized, CPGA performs binary tournament selection, crossover between one individual from the fitter half and the other randomly selected from the population. Mutation operation is then performed on these individuals with suitable probabilities and fitness/error evaluation at each of the 256 data points starting from 0 and incremented in steps of 0.016 for the independent variable *x* are performed. The best two individuals of each generation are retained without undergoing mutation and the number of individuals to be replaced after crossover is a tunable parameter. Once the termination condition is reached, the coefficients of the best fit individual are taken, scaled back to the original size and the solution function is evaluated against the originally computed oracle to determine performance error.

CaDR essentially emulates one possible Puiseux series expansion of a mathematical function of the independent variable x involving the powers discussed above. At the end of CPGA, the coefficients for the best fit individual produced by CPGA are printed out. These coefficients point to the possible combinations of evolved powers of x with varying degrees of error in each of them. Thus CaDR solves a multi-objective optimization problem of determining an appropriate set of weights that can be used to combine results of approximate computations of the mathematical building blocks to compute arbitrarily complex functions. It is interesting to note that a combination of integral and fractional powers of independent variable may be used to compute its higher powers. Also, several mathematical functions may be constructed using CaDR and their results stored and used to compute more complex functions involving them and integral and possibly fractional powers.



```
best fitness acheived: 10398.856445
chromosome of best individual:
x^1/4: 0.000002
x^1/3: 0.000012
x^1/2: -0.000002
x^0: -0.000011
x: -0.000009
x^2: 1.279843
x^3: -6.552299
x^4: 5.310815
```

Figure 14: Coefficients evolved to approximate fifth power of *x*



```
best fitness acheived: 1522.149414
chromosome of best individual:
x^1/4: -2.883228
x^1/3: 0.000000
x^1/2: 0.000000
x^0: 0.000000
x: -0.000315
x^2: -0.000454
x^3: -0.908246
x^4: 1.347791
```

Figure 15: Coefficients evolved to approximate fourth power of *x*

CaDR can also be used to improve upon the accuracy of powers of x evolved by DDC by combining small fractions of the other powers evolved. As shown in figure 15, the coefficient for the fourth power is close to 1, but owing to the error in the DDC stage, other powers of x are also involved. The performance of CPGA for the fourth power compares to that for evolution through

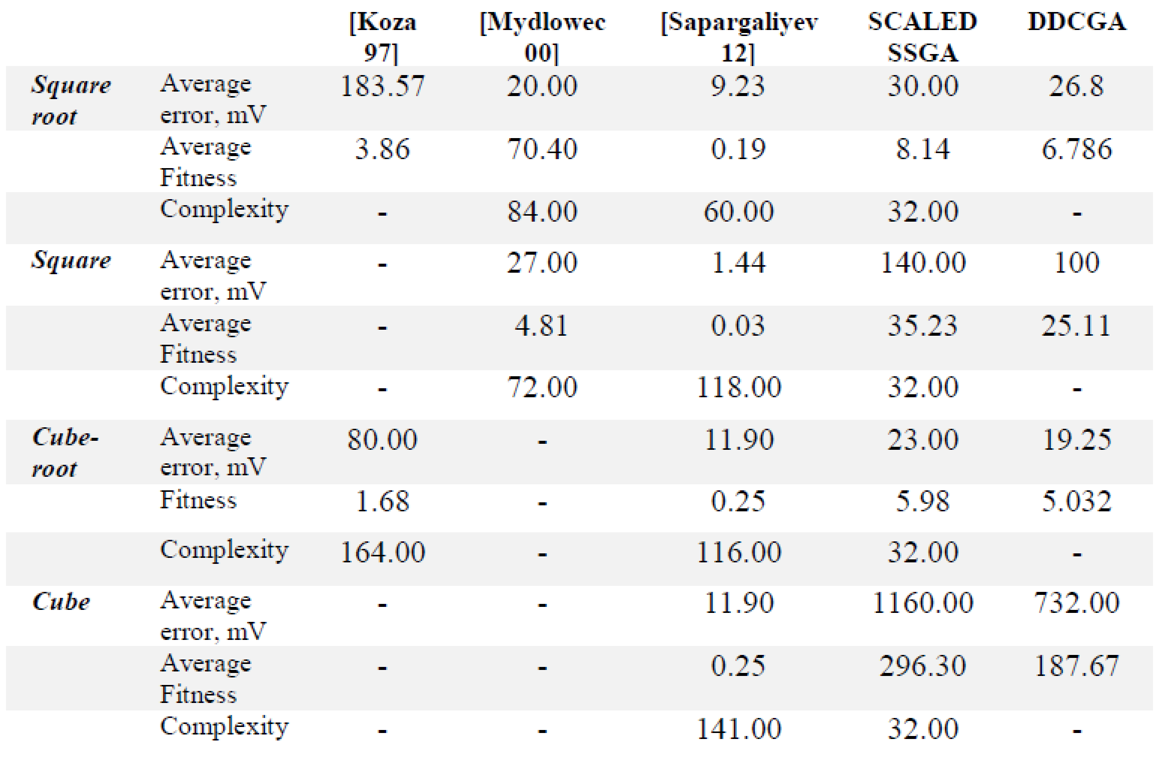
DDC without hypermutation. Though, this approach may not guarantee better solutions for smaller powers of x, it performed notably to produce results for a few larger powers of x such as the fifth power. As shown in figure 14, the coefficients to approximate the fifth power of *x* are largely combinations of second, third and fourth powers respectively.

Several interesting observations can be made about the mathematical nature of the coefficients produced to approximate the solution. They are automatically identified and differ depending on the accuracy of the powers of x involved but largely indicate the approximate values needed to approximate a given function with a limited number of powers of x. CPGA was performed only for 500 iterations. With further evolution it may be possible to approximate an arbitrary function to an arbitrary degree of accuracy. The coefficients predicted by CaDR can serve as *mathematical tools to study the nature of Puiseux series expansions* of known functions to begin with. With further investigation, it may be possible to develop formulas or algorithms to predict coefficients to compute a given arbitrary function with various approximate powers of x.

CaDR, being capable of developing solutions to several functions can be used to store values of computed functions. With addition of extra coefficients and an increase in the size of the chromosome for CPGA, these computed results may also be assigned weights and combined with other powers of x to enable possible quicker and more accurate computation of bigger functions of x as the case may be. The powers of x evolved needn't be consecutive powers of x. Though, mathematically convergence of such series may not be guaranteed, several possible combinations exist and may be explored to develop better techniques to build computing techniques to best exploit available resources and computing power adaptively and achieve desired performance. In this sense, multiple cascaded runs of DDC and CP can be used to produce better digital refinement of solution. Thus cascaded digital refinement technique can be recursive in application to achieve desired objectives of accuracy and ability to perform more mathematically involved computations. The possibilities in this direction are limited by the ability to produce accurate formulations of different building blocks.

CHAPTER SEVEN: RESULTS

Several parameters affected the performance of the GAs developed in CaDR. The most important factors that affected GA performance were population size, crossover technique and number of individuals replaced every generation, initial mutation rate, rate of increment of mutation rate and it’s responsiveness to stasis condition when detected, introduction of random individuals and the seeds used for generation of initial population and for choosing crossover point.



		[Koza 97]	**[Mydlowec 00]**	**[Sapargaliyev 12]**	**SCALED SSGA**	**DDCGA**
Square root	Average error, mV	183.57	20.00	9.23	30.00	26.8
	Average Fitness	3.86	70.40	0.19	8.14	6.786
	Complexity	-	84.00	60.00	32.00	-
Square	Average error, mV	-	27.00	1.44	140.00	100
	Average Fitness	-	4.81	0.03	35.23	25.11
	Complexity	-	72.00	118.00	32.00	-
Cube-root	Average error, mV	80.00	-	11.90	23.00	19.25
	Fitness	1.68	-	0.25	5.98	5.032
	Complexity	164.00	-	116.00	32.00	-
Cube	Average error, mV	-	-	11.90	1160.00	732.00
	Average Fitness	-	-	0.25	296.30	187.67
	Complexity	-	-	141.00	32.00	-

Figure 16: Collection of comparative performance of DDC against previous works [35]

Compared to the results of the previous works in Figure 16, the square-root and cube-root CCs evolved with SSGA achieved an average error of 30mV and 23mV, respectively, and performed better than Koza et al. The square-root CC evolved in this paper performed marginally better than Mydlowee et al. [26], with an average error of 20mV, but the square CC did not outperform. All test cases performed worse than in [18], but considering their work evolved CCs extrinsically without device constraints, this is understandable. It is interesting to note that DDC improved accuracy of all circuits to various degrees. The greatest reduction in average error was seen for the cube circuit where a reduction from 1160mV to 732mV yielded a 36.89 percent reduction in error on average. Likewise, DDC improved accuracy by reducing average error in square, square root and cube-root CCs by 28.57, 10.67 and 16.3 percent respectively, on average. Some general trends were observed with regards to error reduction by DDC. For all CCs, the error reduction was larger when SSGA performance was worse than average, thus providing for a stabilizing effect to maintain accuracy within reasonable bounds. Performance for individual cases depended on error distribution of SSGA output. As shown in Figure 17, SSGA evolves four islands of populations in parallel to produce a best fit individual with a total error of 296.3. The DDC then evolves the digital fabric to correct errors and reduce it to 169.48. As far as the authors are aware, this is the first realization of intrinsic evolution of analog CCs on a commercial PSoC device utilizing a compact fabric of 4 SC op-amp Blocks rather than an unlimited number of resistors and BJTs. With an addition of only four PLDs, significant accuracy improvements were also achieved.

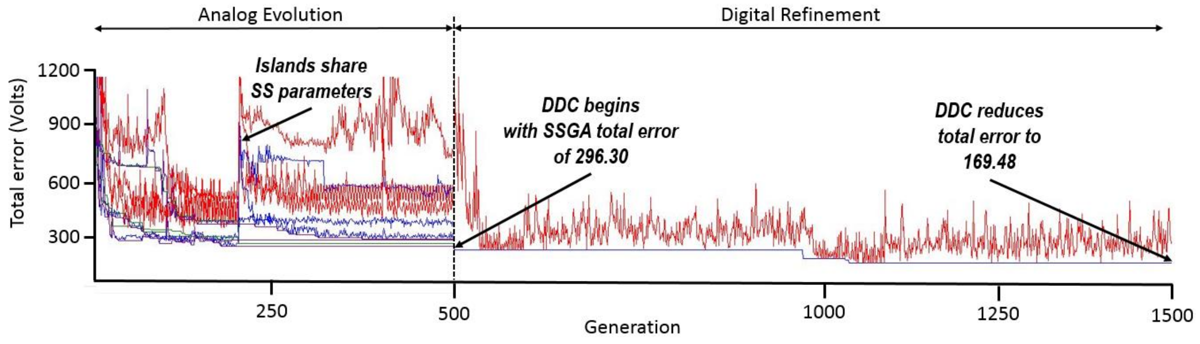


Figure 17: Evolution of cube circuit and its refinement using DDC

To save evolution time, the results of analog evolution of each circuit were stored in separate header files which were then included individually as DDC was used to refine them one after the other in a cascaded fashion. The results for fourth root, cube root, square root, square, cube and fourth power circuits are as indicated in figures 18 through 29. Evolution was performed for 1000 iterations in DDC and the scale in all these figures is Generation*2/3. In order to obtain the best performance for each of these circuits, a few parameters required change, while the style and flavor of mutation remained largely the same. Hypermutation in DDC is implemented as the introduction of one random individual every few hundred generations of evolution. Hypermutation rate is defined in this context as the interval in terms of the number of generations when a random individual is introduced to the population pool. Hypermutation rates of 100, 200 and 300 were largely experimented with occasional use of 150 and other rates for better results for individual circuits evolved in DDC. From the trends observed in the CCs evolved for computations of various powers of *x*, it is seen that larger powers of *x* tend to have bigger errors in computation owing to the larger errors and correspondingly larger maximum deviation propagated from the analog stage. Individual analyses of each of the evolved circuits are presented.

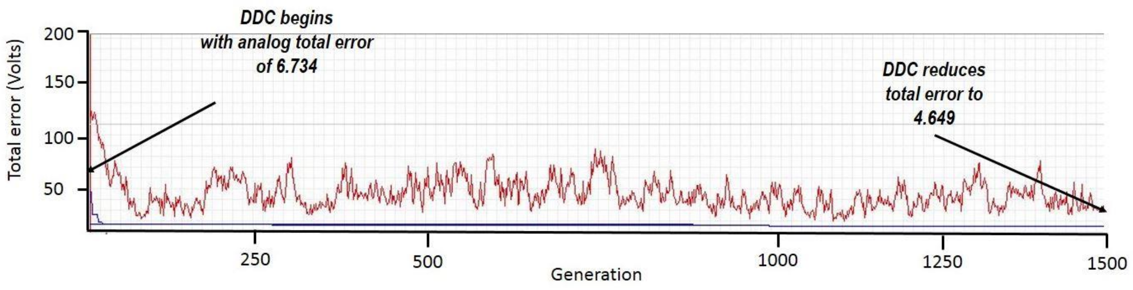


Figure 18: DDC Evolution of Fourth Root circuit without hypermutation

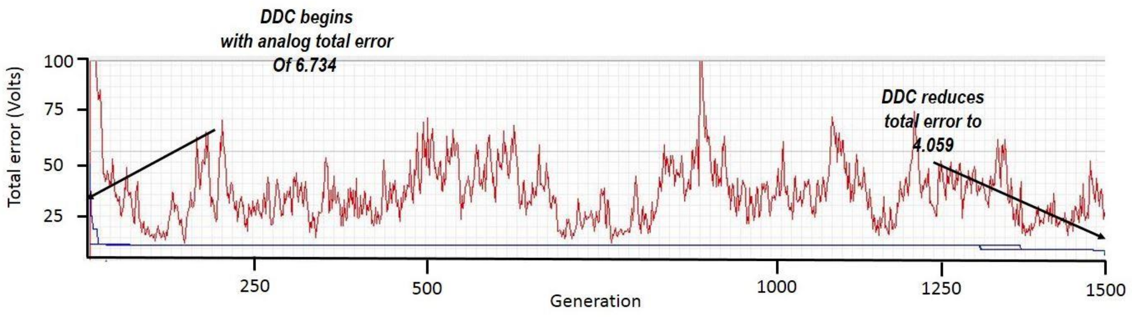


Figure 19: Best case DDC evolution of Fourth Root circuit with a hypermutation rate of 300

Fourth root circuit's digital evolution phase is as presented in Figure 18 without hypermutation and Figure 19 for the best case evolved with a hypermutation rate of 300, respectively. Analog evolution phase ends with a total error of 6.734 as compared to the oracle computed. DDC then refines this circuit to produce configuration of PLDs that reduce the total error to 4.649 in the first case and to 4.059 in the second, thus enhancing the accuracy of computation by 31% and 39% respectively, for the fourth root case. The effect of hypermutation is significantly visible in the evolution of fourth root circuit as can be seen in the difference in total error for fourth root circuit with the best case hypermutation rate of 300 over the other case. More frequent hypermutations resulted in poorer performance. The effects of multiple introducing multiple individuals for hypermutation can be explored.

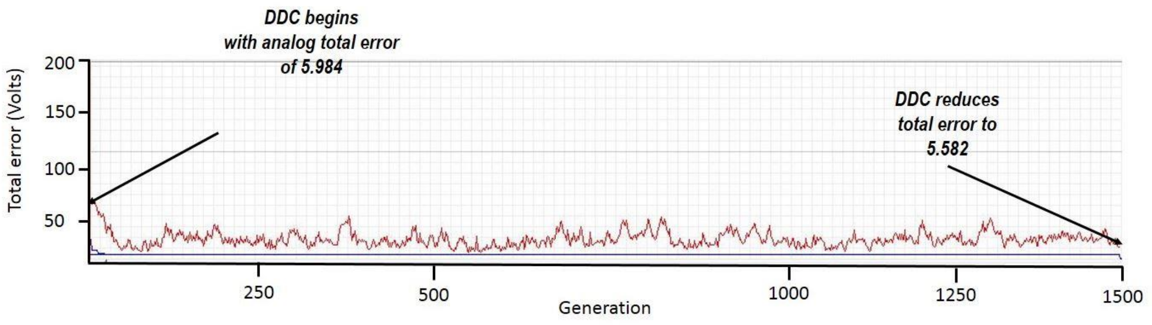


Figure 20: DDC Evolution of Cube Root circuit without hypermutation

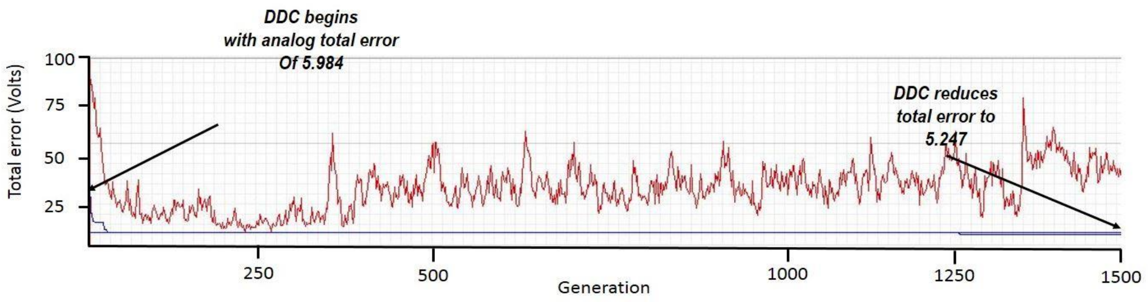


Figure 21: Best case DDC evolution of Cube Root circuit with a hypermutation rate of 300

Cube root circuit's digital evolution phase is as presented in Figure 20 without hypermutation and Figure 21 for the best case evolved with a hypermutation rate of 300, respectively. Analog evolution phase ends with a total error of 5.984 as compared to the oracle computed. DDC then refines this circuit to produce configuration of PLDs that reduce the total error to 5.582 in the first case and to 5.247 in the second, thus enhancing the accuracy of computation by 6.7% and 12.3% respectively, for the cube root case. The effect of hypermutation is significantly visible in the evolution of cube root circuit as can be seen in the difference in total error for cube root circuit with the best case hypermutation rate of 300 over the other case. More frequent hypermutations resulted in a decrease in performance. However, other rates close to 300 may improve performance.

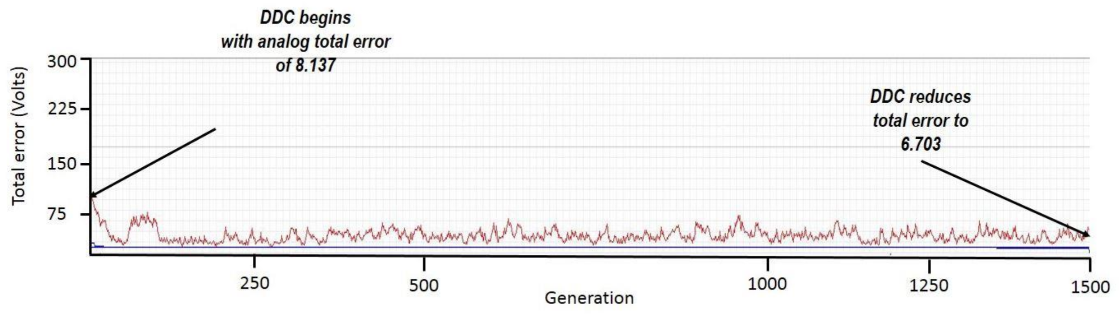


Figure 22: DDC Evolution of Square Root circuit without hypermutation

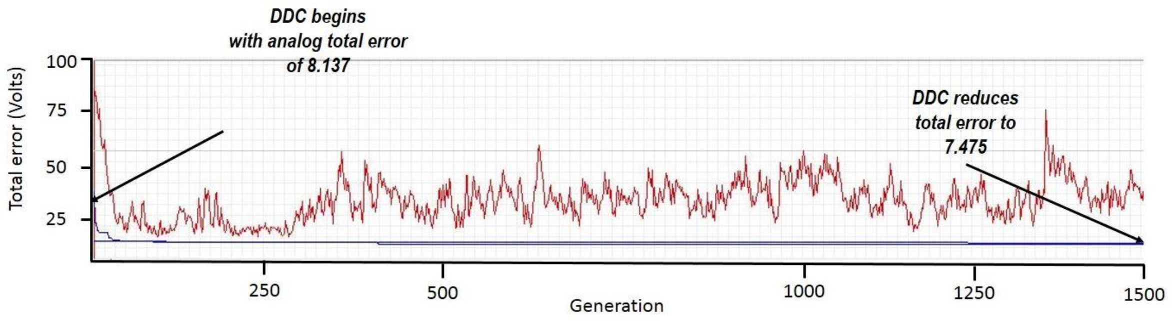


Figure 23: Best case DDC evolution of Square Root circuit with a hypermutation rate of 200

Square root circuit's digital evolution phase is as presented in Figure 22 without hypermutation and Figure 23 for the best case evolved with a hypermutation rate of 200, respectively. Interestingly slower hypermutation rates like 300 or above resulted in identical performance to the case without hypermutation. Analog evolution phase ends with a total error of 8.137 as compared to the oracle computed. DDC then refines this circuit to produce configuration of PLDs that reduce the total error to 6.703 in the first case and to 7.475 in the second, thus enhancing the accuracy of computation by 17.6% and 8.1% respectively, for the square root case. The effect of hypermutation is significantly visible in the evolution of square root circuit as can be seen in the difference in total error for square root circuit with the best case hypermutation rate of 200 over the other case. More frequent hypermutations resulted in a decrease in performance. However, other rates between 250 and 300 may improve performance

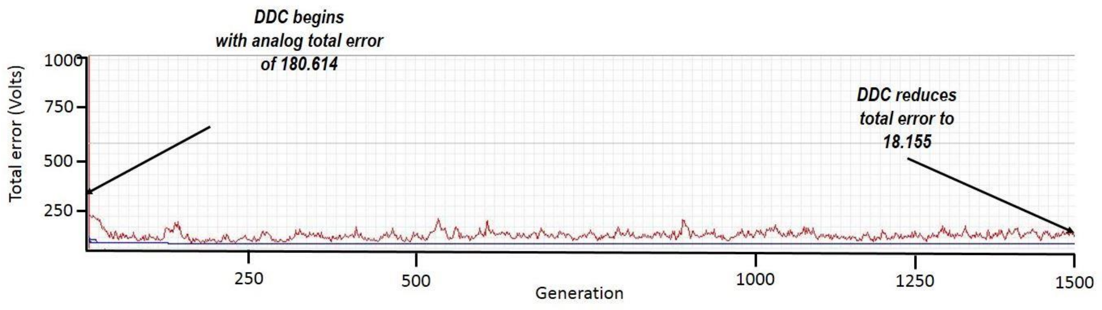


Figure 24: DDC Evolution of First Power circuit without hypermutation

First power circuit's digital evolution phase is as presented in Figure 24. Analog evolution phase ends with a total error of 180.614 as compared to the oracle computed. DDC then refines this circuit to produce configuration of PLDs that reduce the total error to 18.155, thus enhancing the accuracy of computation by a whopping 89.9%for this case. The reason for such difference is the difference in treatment of the problem as viewed by the analog and digital fabrics respectively. It is observed that the performance of this circuit is almost identical for all hypermutation rates used and is a trivial evolution case for digital fabric. Also zeroth power of *x*, being trivial is not evolved through DDC as it doesn't warrant the need for the same.

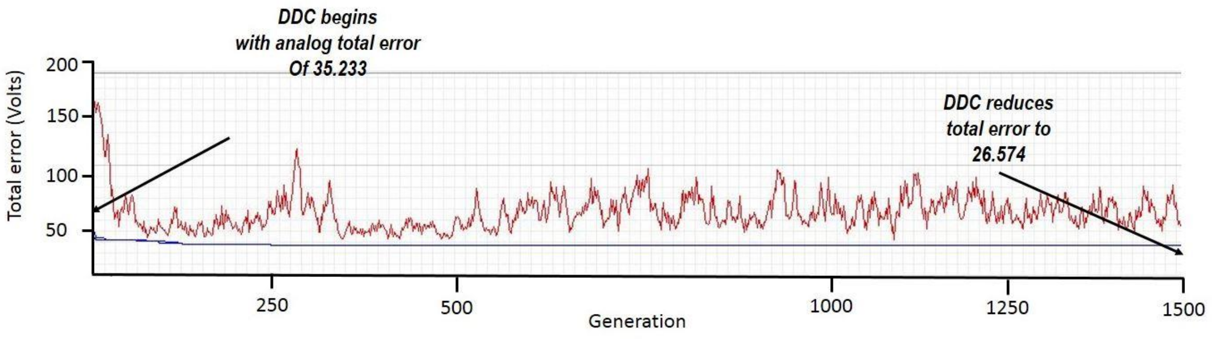


Figure 25: DDC Evolution of Square circuit without hypermutation

Square circuit's digital evolution phase is as presented in Figure 25 without hypermutation. The best evolved case with hypermutation was the one with a rate of 200 and its performance matched very closely to the above case at 26.876. Interestingly slower hypermutation rates like 300 or faster ones like 100 resulted in poorer performance to the 200 rate case which was marginally worse than the case without hypermutation. Analog evolution phase ends with a total error of 35.233 as compared to the oracle computed. DDC then refines this circuit to produce configuration of PLDs that reduce the total error to 26.574, thus enhancing the accuracy of computation by 24.6% for the square case. The effect of hypermutation is visible in the evolution of squar circuit as can be seen in the difference in total error for square circuit with the best case hypermutation rate of 200 over the other cases. Other rates of hypermutation between 150 and 250 may improve performance

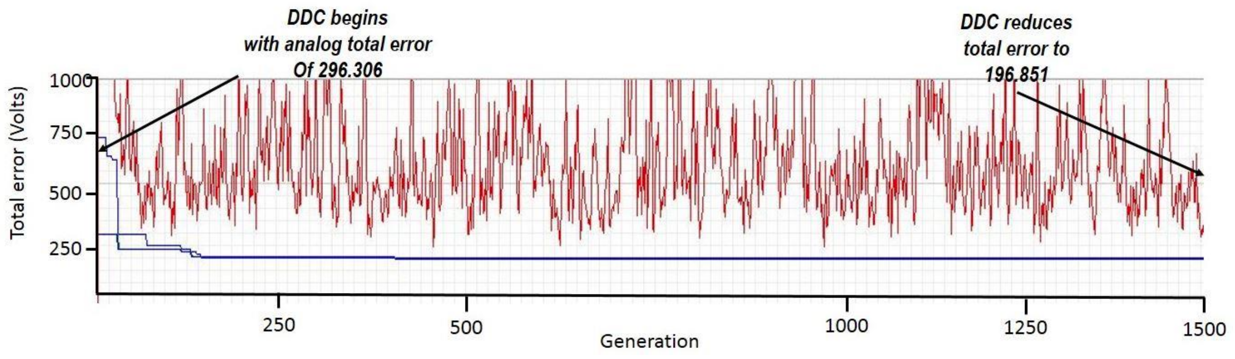


Figure 26: DDC Evolution of Cube circuit without hypermutation

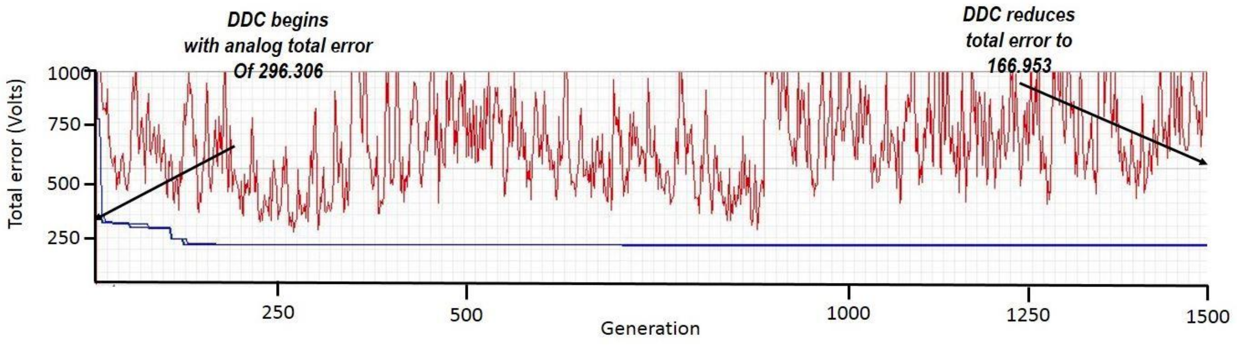


Figure 27: DDC evolution of Cube circuit with a hypermutation rate of 200

Cube circuit's digital evolution phase is as presented in Figure 26 without hypermutation and as in Figure 27 for a better evolution case with a hypermutation rate of 200. The best evolved case with hypermutation was the one with a rate of 150 and its performance came close to the case with a rate of 200 at a total error of 161.82. Interestingly slower hypermutation rates like 300 or higher resulted in poorer performance. Also hypermutation rates faster than 100 degraded performance by a bit but still managing to perform better than the case without hypermutation. Analog evolution phase ends with a total error of 296.306 as compared to the oracle computed. DDC then refines this circuit to produce configuration of PLDs that reduce the total error to 196.851 and 166.953, thus enhancing the accuracy of computation by 33.5% and 43.6% respectively, for the cube case. The effect of hypermutation is visible in the evolution of cube circuit as can be seen in the difference in total error for cube circuit with the best case hypermutation rate of 150 over the other cases. Other rates of hypermutation between 100 and 200 are expected to performance.

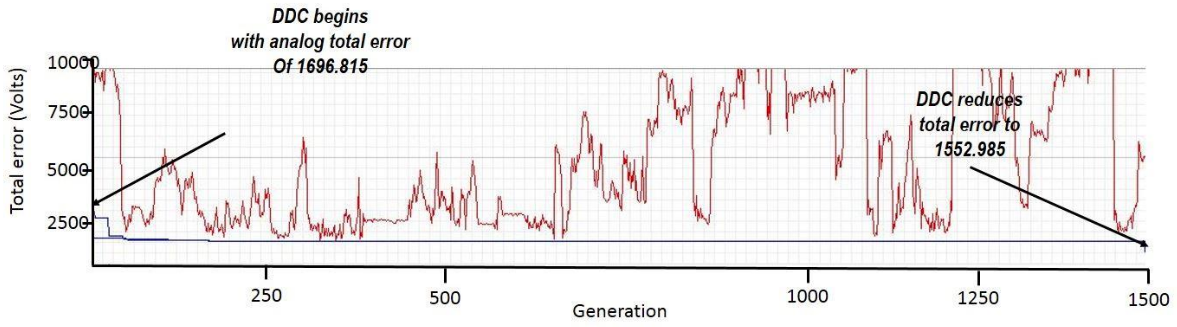


Figure 28: DDC Evolution of Fourth Power circuit without hypermutation

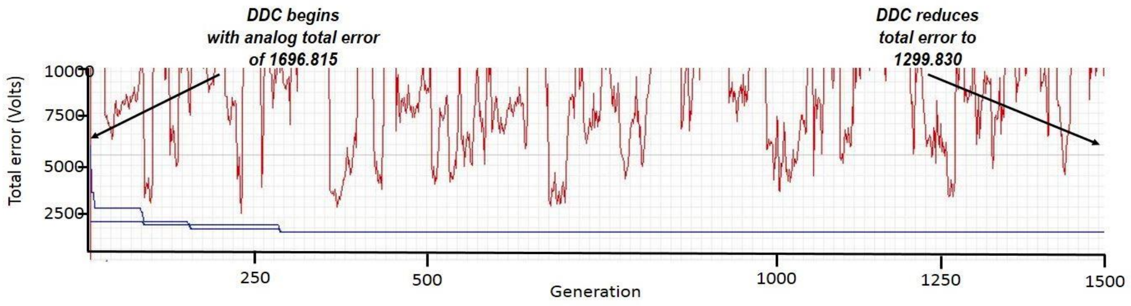


Figure 29: DDC evolution of Fourth Power circuit with a hypermutation rate of 100

Fourth power circuit's digital evolution phase is as presented in Figure 28 without hypermutation and as in Figure 29 for the best evolution case with a hypermutation rate of 100. Performance notably increased for faster hypermutation rates than 150 and slower hypermutation rates like 300 or higher resulted in poorer performance. Also hypermutation rates faster than 100 seemed to improve performance by a bit but rates as fast as 50 degraded performance. Analog evolution phase ends with a total error of 1696.815 as compared to the oracle computed. DDC then refines this circuit to produce configuration of PLDs that reduce the total error to 1552.985 and 1299.830 respectively, thus enhancing the accuracy of computation by 8.4% and 23.3% respectively, for the fourth power case. The effect of hypermutation is best visible in the evolution of fourth power circuit as can be seen in the difference in total error for fourth power circuit with and without hypermutation. Other rates of hypermutation less than 150 need to be explored to find better performing solutions

Overall certain trends are clearly visible in the evolution of individual powers of *x* using DDC. Faster hypermutation helps with evolution of larger powers and slower hypermutation or the lack of it helps with evolution of smaller powers of *x*. Further exploration of various hypermutation rates would be necessary to ascertain the best performance ranges for other intermediate powers of *x* some of which have been developed in this work. Work on other fractional powers of *x* greater than 1 is being pursued and offers richer possibilities with regards to Puiseux series expansions of functions and possible enhancements in accuracy when used in concert with CPGA.

To save evolution time, the results of DDC for each circuit were stored in separate header files which were then included together for CPGA to use the same to predict their coefficients. The functions, sin(*x*) and cos(*x*) were chosen as the test functions and CPGA predicted coefficients to combine the fourth root, cube root, square root, zeroth power, first power, square, cube and fourth power circuits to approximate the functions sin(*x*) and cos(*x*) in separate runs. Results for evolution of these coefficients are as indicated in figures 30 through 34. Evolution was performed for 500 iterations in CP and the scale in all these figures is Generation*3. In order to obtain the best performance for each of these circuits, a few parameters required change, while the style and flavor of mutation remained largely the same. Range forcing essentially implemented dynamic range adaptation technique of reducing the range of coefficients to half the previous range even when the total error is less than 100. For the functions sin(*x*) and cos(*x*) attempted, this was unable to produce better results. Hence dynamic range adaptation alone was used with adaptive mutation techniques used in their respective flavors as discussed in CPGA.

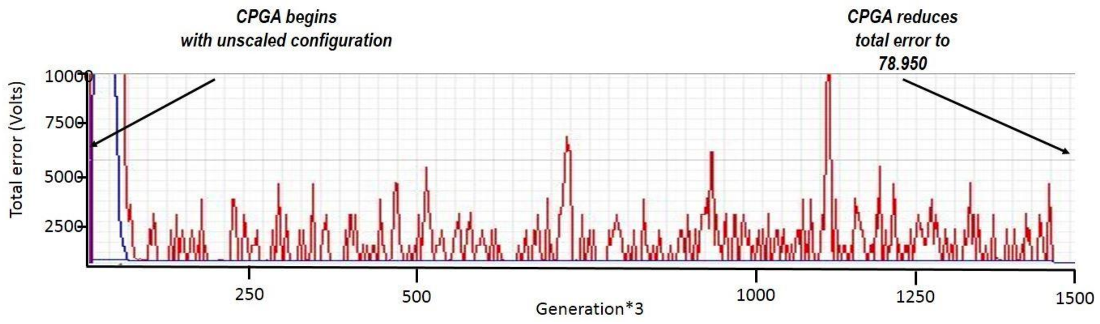


Figure 30: CPGA evolution of coefficients for sin(*x*) without range forcing

The result of evolution of coefficients for sin(*x*) without range forcing is as shown in figure 30. First range scaling is performed and hence a steep slope is observed in fitness curve. The largest value allowed for each coefficient was found to be in the range of 60000 and correspondingly the smallest was in the range of -60000. Each coefficient is initialized to a value falling within the bracket mentioned. In every following iteration, the bracket was halved down and all but the first 10 individuals were reinitialized to random values, if the best fit individual in the previous bracket had a total error greater than 100. This was continued till a suitable range for coefficients was detected and the total error for the best fit case was small enough to continue evolution. Starting from a random search space with a huge total error (>10000000), CPGA reduced the total error to 78.950 in 500 generations.

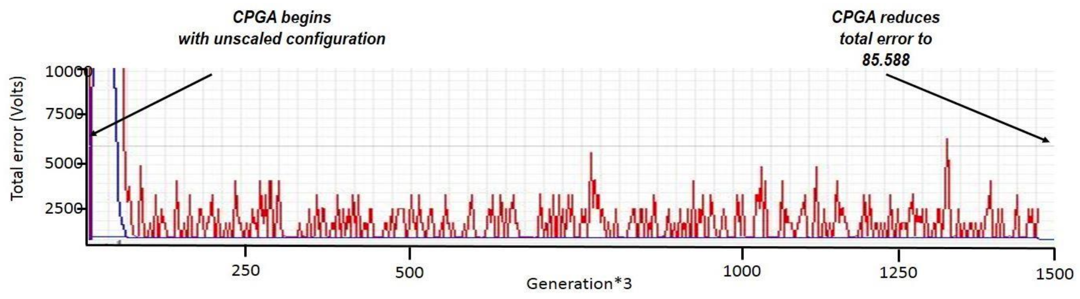


Figure 31: CPGA evolution of coefficients for sin(*x*) with range forcing for error<100

When range forcing was done for the case of error smaller than 100, the performance was as observed in figure 31. Range forcing forced the bracket of coefficients for half the population to half the previous bracket when CPGA first produced a set of coefficients that could reduce the error to just below 100. The reason for poorer performance is that a good set of individuals for evolution is removed by range forcing and hence the remaining individuals aren't able to produce better solutions with limited diversity in the desired range. As discussed before, dynamic range adaptation first halved the bracket from (-60000, 60000) till the fitness reached 100 and then continued reduction of bracket for a half the population instead. The coefficients so predicted resulted in a total error of 85.588 after evolution through 500 generations.

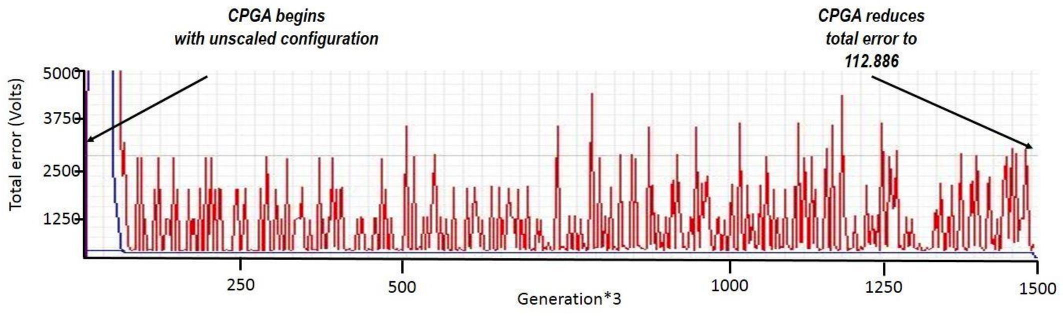
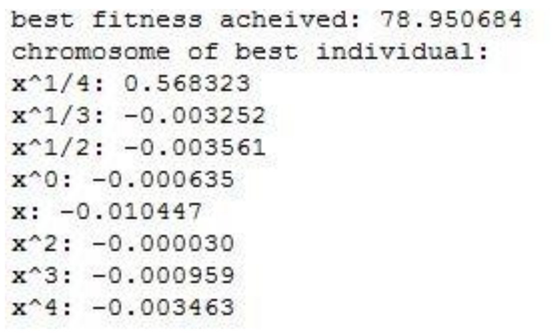


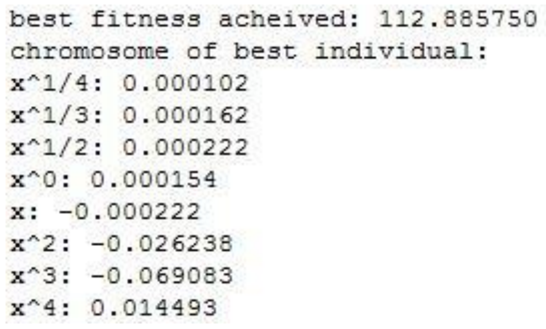
Figure 32: CPGA evolution of coefficients for cos(x) without range forcing

Evolution for cos(x) is as shown in figure 32. Following the failure of range forcing to produce better results for sin(x), range forcing wasn't attempted for this case. It is interesting to note that the performance of CPGA to predict coefficients for sin(x) is better than the same for cos(x). CPGA was employed with no changes to mutation rates and flavor and this resulted in prediction of coefficients that reduced the fitness from more than 10000000 in a random search space to 112.886 in 500 generations.



```
best fitness acheived: 78.950684
chromosome of best individual:
x^1/4: 0.568323
x^1/3: -0.003252
x^1/2: -0.003561
x^0: -0.000635
x: -0.010447
x^2: -0.000030
x^3: -0.000959
x^4: -0.003463
```

Figure 33: Coefficients evolved to approximate sine of *x*



```
best fitness acheived: 112.885750
chromosome of best individual:
x^1/4: 0.000102
x^1/3: 0.000162
x^1/2: 0.000222
x^0: 0.000154
x: -0.000222
x^2: -0.026238
x^3: -0.069083
x^4: 0.014493
```

Figure 34: Coefficients evolved to approximate cosine of *x*

The coefficients evolved for sine and cosine of *x* are as shown in figures 33 and 34 respectively. For sine of *x* it is seen that the fourth root is the major contributor in approximating the function. Likewise for cosine of *x* it is seen that cube and the fourth power of *x* are the major contributors.

Evolution of coefficients for random polynomials involving upto fifth power of *x* and combination of powers of *x* and sine or cosine of *x* were also attempted. It was observed that dynamic range adaptation required modifications in implementation to allow CPGA to explore a certain bracket well enough before narrowing down to a smaller bracket. Also, the accuracy of the individual powers of *x* evolved by DDC had a significant impact on accuracy. In order to improve the performance of CaDR, smaller and more accurate powers need to be evolved through DDC and implementation of CPGA needs tuning to prevent premature narrowing of the search space when predicting coefficients. Dynamic storage of newly computed results and variable size chromosome for CPGA could result in better accuracy for bigger computational circuits. This, however is bound by the runtime memory and on-board RAM available at the platform's disposal.

CHAPTER EIGHT: CONCLUSION AND FUTURE WORKS

CaDR - a multilevel strategy to evolve solution functions of differential equations by combining various powers of the independent variable to produce power series expansion is developed herein. CaDR considers overflow issues and performs range scaling to predict solutions within appropriate range of values represented as 32 bit floating point numbers. This work builds on the SCALER technique built in [35], a two-method approach to scale, translate, and refine evolved analog computational circuits using evolved digital resources. Once analog computational circuits are evolved, DDC then evolves a precise digital error compensation circuit to compensate for analog aberrations while extending its accuracy and precision. To effect the same in a cascaded fashion, without heavy memory overheads, the outputs from the evolved analog circuits are stored in header files that are then included in the code for the implementation of CaDR to produce the final solution. To demonstrate the CaDR approach, a simple differential equation, namely *dy/dx = cos(x)* was defined as the problem to be solved, whose solution, namely sin(x) was pre-computed and used as the oracle against which evolution of solution function was done. CP was successfully able to predict coefficients for the eight powers, fourth root through fourth power of the independent variable to approximate the sine and cosine functions of the independent variable.to a reasonable level of accuracy. The novel hybrid analog-digital design that is evolved leverages the relative advantages of both circuit domains.

The precision of DDC can further be improved by using values that can satisfy a 16-bit mapping instead of the 8-bit mapping used here. Alternatively, an approach to intrinsically generate potentially faster or perhaps energy-conserving hybrid analog-digital computational circuits for repetitive scientific or restricted energy applications is being investigated.

As an extension to the above work done to compute solution to a simple differential equation, computations of solutions to more complex higher order, ordinary differential equations may be performed by effecting appropriate range scaling and utilization of solutions of simpler functions with higher level coefficient prediction algorithms that can combine them to quickly yield the desired solution. CaDR can be utilized to generate a library/suite of commonly used mathematical functions such as trigonometric functions which can then be combined with various powers of *x* with adaptive coefficient prediction GA. Mathematically speaking, CaDR attempts to produce a truncated Puiseux series or generalized form of power series with fractional exponents to approximate functions. A natural extension is to evolve various fractional powers of *x* using cascaded DDC and then using CP to predict coefficients and thereby develop expansions that can calculate solution functions more accurately. Also, CaDR could benefit by evolution of solutions using a larger series than the 8 powers of *x* used in this work. Accuracy concerns determine the effectiveness of such extensions.

Reliable Evolution Control through Periodic BIST Based Fitness Adjustments

In the face of increasingly adaptive hardware with longer evolution times and/or adaptation conditions, more energy efficient, less time and resource intensive, online self-testing mechanisms are necessary to ensure that evolution is controlled to achieve best results while remaining alert to the possibility of failures both during and after evolution.

As an extension to the work done in CaDR, EvolBIST – a technique to ensure reliability in evolution time with periodic insertion of online BIST functions in the fitness function of the main operational GA to evolve circuits that are failure-aware both during and after evolution is proposed. Typical reliability check mechanisms employ BIST functions to fully evolved circuits and use modular redundancy to maintain performance. EvolBIST proposes a mechanism for slow evolution of redundant resources which are periodically written with the best fit configuration obtained from the main GA. Redundant resources are picked from well spread out areas of the fabric to ensure that this pool has a very small probability of being affected by the same type of faults as other resources. Their outputs are compared to that of the main circuit during the first step of BIST and presence/absence of fault is immediately identified in the main circuit by a mismatch condition. If no fault is identified, the redundant resource pool starts evolution, while BIST is done on the main circuit. In order to perform BIST, the configuration store of the redundant resources is updated with those of the last step in evolution for the main circuit, before BIST. When the redundant circuit has evolved to a level where a decent level of performance is deliverable, subsequent BIST runs on the main circuit are accompanied by maintenance of performance by the redundant circuit through muxes, which are assumed to be golden. While there are no faults, both the main and redundant circuits continue evolution at their respective rates and perform BIST alternately keeping the performance at desired levels. If under fault, BIST is performed to identify the faulty resource while redundant resource pool is in standby till the faulty element in the main circuit is identified and swapped with the corresponding resource from the redundant pool. The faulty element is then evolved with the redundant resources at the slower rate demarcated for the redundant pool.

After evolution is complete, fault information from this redundant pool is used to enhance BIST of working elements. This also gives an opportunity to detect multiple faults after evolution, as information about faults that have occurred at evolution time is then available. If further adaptation of the main circuit or its parts is necessary, EvolBIST ensures that evolution proceeds with available information about fault propagation in the resources and also gives an opportunity to re-use faulty resources to avoid performance degradation beyond a certain level.

APPENDIX A: SAMPLE DDC EVOLUTION CODE

Sample code illustrating the DDC evolution of cube circuit:

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF Vignesh Thangavel.
 *
 * ========================================
  */


//#include <root_fourth.h>
//#include <cubert.h>
//#include <sqrt.h>
//#include <root_third.h>
//#include <square.h>
#include <cube.h>
//#include <x4.h>
//#include <x5.h>
//#include <x6.h>
//#include <x7.h>
//#include <x8.h>
//#include <oracle.h>


//#define start_seed 27
#define start_seed 4
#include <ga_def.h>
#include <project.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
#include <LCD.h>
#include <UART_1.h>
#define BEGIN_DET_IT 0x40010000
#define BEGIN_DET_ORT 0x40010030
#define END_DET_ITORT 0x40010038
#define BEGIN_DET_IT2 0x40010280
#define BEGIN_DET_ORT2 0x400102B0
#define END_DET_ITORT2 0x400102B8

//#define BEGIN_DET_IT3 0x40010200
//#define BEGIN_DET_ORT3 0x40010230
//#define END_DET_ITORT3 0x40010238
#define BEGIN_DET_IT3 0x40010400
#define BEGIN_DET_ORT3 0x40010430
```

```
#define END_DET_ITORT3 0x40010438

//#define BEGIN_DET_IT4 0x40010080
//#define BEGIN_DET_ORT4 0x400100B0
//#define END_DET_ITORT4 0x400100B8
#define BEGIN_DET_IT4 0x40010680
#define BEGIN_DET_ORT4 0x400106B0
#define END_DET_ITORT4 0x400106B8

#define N_digi 80
#define G 1500 //1500 fits in one screen

/*******GradMut 0 means we start with 0.1% and 1 means we start with
0.01%****/

#define N3_init 1 //square (also cube) circuit - 1, square (and cube)root - 1
or 2, start with upto 10 for GradMut 1
#define N4_init 1 //square (also cube) circuit - 1, square (and cube)root - 1
or 2, start with upto 10 for GradMut 1
#define N3_incr 0.1 //square circuit (also cube) - 0.1 with GradMut 0, square
(and cube)root - 1 or 2 with GradMut 1??
#define N4_incr 0.1 //square circuit (also cube)- 0.1 with GradMut 0, square
(and cube)root - 1 or 2 with GradMut 1??
#define Ulim 1000   //square circuit(also cube) - 50, square (and cube)root -
1 or 2
#define Llim 50     //square circuit (also cube) - 10, square (and cube)root
- 1 or 2
#define GradMut 0 //if 0 mutation rate increments by 0.1 every time, but
actually changes probability only when it is a whole number [rand()%1000] and
if 1, mutation rate increments in gradual smaller steps rand()%10000
#define Xovertype 1 //if 0, fixed tournament with replacement of half the
individuals and if 1, random tournaments of 2
#define N_repl 70 //better to change if Xovertype is 1, but it can be changed
even otherwise to see if there are better results
struct ClsIndividual{
      uint16 active_lines;
      uint32 IN_PT[8];
      uint16 ORT_PT[4];
};
struct ClsIndividual Individuals[N_digi], Individuals2[N_digi],
new_individual, new_individual2, fitter_indi[N_digi/2],
fitter_indi2[N_digi/2];

struct ClsIndividual fittest_indi, fittest_indi2, fresh_indi, fresh_indi2;

struct ClsIndividual *Individualsp;
struct ClsIndividual *Individuals1p;
struct ClsIndividual *indp1;
struct ClsIndividual *ind1p1;
struct ClsIndividual *indp2;
struct ClsIndividual *ind1p2;
struct ClsIndividual *new_individualp;
struct ClsIndividual *new_individual2p;
int main()
{
//    goldenIndividual* bestAnalog = AGA_DDA(400,100,20);
//    CyPins_ClearPin(Pin_6_0);
//    CyPins_ClearPin(Pin_7_0);
//    CyPins_ClearPin(Pin_8_0);
//    CyPins_ClearPin(Pin_9_0);
//    CyPins_ClearPin(Pin_10_0);
//    CyPins_ClearPin(Pin_11_0);
//    CyPins_ClearPin(Pin_12_0);
//    CyPins_ClearPin(Pin_13_0);

      UART_1_Start();
      char string[80];
      LCD_Start();
      LCD_WriteControl(LCD_CLEAR_DISPLAY );
      /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    /* CyGlobalIntEnable; */ /* Uncomment this line to enable global
interrupts. */

      /***************************************************PHASE 1: detection
of logic used*****************************************************/
      //LCD_PrintString("P1 ");
      uint32 i=0, j=0, temp=0;
      uint16 const_reg=0, bypass=0, bypass2=0, const_reg2=0, iteration1 = 0,
temp1=0;
      uint8  stasis = 0, pos_array[8] = {0}, y_array[256] = {0}, x=0, m=0,
zf=0, z=0, z1=0, z2=0, y=0, r=0, D=0, out[8] = {0}, N1=100, N2=1, c2 = 0,
c3=0, c4=0, c5=0, D_array[256] = {0};
      uint32 iteration = 0, it_max=0;
      float32 fitness[N_digi]={0}, fitness_avg = 0, fitness_max_prev = 1000,
fitness_max[2]={20000, 20000}, fitness_last=0, fit_t_max=1000, c1=0,  c = 0,
difference[256] = {0}, tot_diff=0, tot_oracle=0, net_diff=0, N3=0, N4=0;
      for(i=0; i<8; i++) {pos_array[i] = 1<<i;}
//    for(i=0; i<256; i++){y_array[i] = 255-i;}
      Individualsp = &Individuals[0];
      for(i=BEGIN_DET_IT, j=0; i<BEGIN_DET_ORT; i += 4, j++)
      {temp = CY_GET_REG32(i);
      if(temp == 0)
      {Individualsp->active_lines += 0*(1<<j);}
      else
      {Individualsp->active_lines += 1*(1<<j);
      Individualsp->IN_PT[m] = temp;
      m++;}
      }

      temp=0; m=0;
      for(i=BEGIN_DET_ORT; i<END_DET_ITORT; i +=2, j++)
      {temp1 = CY_GET_REG16(i);
            if(temp1 == 0)
      {Individualsp->active_lines += 0*(1<<j);}
      else
      {Individualsp->active_lines += 1*(1<<j);
      Individualsp->ORT_PT[m] = temp1;
      m++;}
      }

      const_reg = CY_GET_REG16(CYREG_B0_P0_U0_MC_CFG_CEN_CONST);
      bypass = CY_GET_REG16(CYREG_B0_P0_U0_MC_CFG_BYPASS);

//    sprintf(string, "%x %x %x %x %x %x %x %x %x %x %x %x %x\n\n\r",
Individuals[0].IN_PT[0], Individuals[0].IN_PT[1], Individuals[0].IN_PT[2],
Individuals[0].IN_PT[3], Individuals[0].IN_PT[4], Individuals[0].IN_PT[5],
Individuals[0].IN_PT[6], Individuals[0].IN_PT[7], Individuals[0].ORT_PT[0],
Individuals[0].ORT_PT[1], Individuals[0].ORT_PT[2], Individuals[0].ORT_PT[3],
Individuals[0].active_lines);
//    UART_1_PutString(string);

      temp=0; temp1=0; m=0;


      Individualsp = &Individuals2[0];
      for(i=BEGIN_DET_IT2, j=0; i<BEGIN_DET_ORT2; i += 4, j++)
      {temp = CY_GET_REG32(i);
      if(temp == 0)
      {Individualsp->active_lines += 0*(1<<j);}
      else
      {Individualsp->active_lines += 1*(1<<j);
      Individualsp->IN_PT[m] = temp;
      m++;}
      }

      temp=0; m=0;
      for(i=BEGIN_DET_ORT2; i<END_DET_ITORT2; i +=2, j++)
      {temp1 = CY_GET_REG16(i);
            if(temp1 == 0)
      {Individualsp->active_lines += 0*(1<<j);}
      else
      {Individualsp->active_lines += 1*(1<<j);
      Individualsp->ORT_PT[m] = temp1;
      m++;}
      }

      const_reg2 = CY_GET_REG16(CYREG_B0_P1_U1_MC_CFG_CEN_CONST);
      bypass2 = CY_GET_REG16(CYREG_B0_P1_U1_MC_CFG_BYPASS);
//    sprintf(string, "%x %x %x %x %x %x %x %x %x %x %x %x %x\n\n\r",
Individuals2[0].IN_PT[0], Individuals2[0].IN_PT[1], Individuals2[0].IN_PT[2],
Individuals2[0].IN_PT[3], Individuals2[0].IN_PT[4], Individuals2[0].IN_PT[5],
Individuals2[0].IN_PT[6], Individuals2[0].IN_PT[7],
Individuals2[0].ORT_PT[0], Individuals2[0].ORT_PT[1],
Individuals2[0].ORT_PT[2], Individuals2[0].ORT_PT[3],
Individuals2[0].active_lines);
```

```
//	UART_1_PutString(string);
	temp1=0; m=0;
	//temp=0;

	            temp1=0; m=0;
	temp=0;

	Individualsp = &fresh_indi;
	for(i=BEGIN_DET_IT3, j=0; i<BEGIN_DET_ORT3; i += 4, j++)
	{temp = CY_GET_REG32(i);
	if(temp == 0)
	{Individualsp->active_lines += 0*(1<<j);}
	else
	{Individualsp->active_lines += 1*(1<<j);
	Individualsp->IN_PT[m] = temp;
	m++;}
	}

	temp=0; m=0;
	for(i=BEGIN_DET_ORT3; i<END_DET_ITORT3; i +=2, j++)
	{temp1 = CY_GET_REG16(i);
	      if(temp1 == 0)
	{Individualsp->active_lines += 0*(1<<j);}
	else
	{Individualsp->active_lines += 1*(1<<j);
	Individualsp->ORT_PT[m] = temp1;
	m++;}
	}
	temp1=0; m=0;
	temp=0;

	Individualsp = &fresh_indi2;
	for(i=BEGIN_DET_IT4, j=0; i<BEGIN_DET_ORT4; i += 4, j++)
	{temp = CY_GET_REG32(i);
	if(temp == 0)
	{Individualsp->active_lines += 0*(1<<j);}
	else
	{Individualsp->active_lines += 1*(1<<j);
	Individualsp->IN_PT[m] = temp;
	m++;}
	}

	temp=0; m=0;
	for(i=BEGIN_DET_ORT4; i<END_DET_ITORT4; i +=2, j++)
	{temp1 = CY_GET_REG16(i);
	      if(temp1 == 0)
	{Individualsp->active_lines += 0*(1<<j);}
	else
	{Individualsp->active_lines += 1*(1<<j);
	Individualsp->ORT_PT[m] = temp1;
	m++;}
	}
```

```
      /********************PHASE 2: Invoke GA to act on the entire PLD
***************************/
      /******************************create
individuals***************************/
      //LCD_PrintString("P2 ");
      srand(start_seed%65535);

      for(z = 1; z < N_digi; z++)
      {Individualsp = &Individuals[z];
      Individuals1p = &Individuals2[z];
      Individuals[z] = Individuals[0]; //ensures that the active lines are
copied as is
      Individuals2[z] = Individuals2[0];
      for(m=0; m<8; m++)
      {
      for(y = 0; y < 32; y++)
      {
      //srand((rand()%65535));
      r = rand()%2; //generate a random number - 0 or 1
      Individualsp->IN_PT[m] += r*(1<<(y));
      r = rand()%2; //generate a random number - 0 or 1
      Individuals1p->IN_PT[m] += r*(1<<(y));
      }
      }
      for(m=0; m<4; m++)
      {for(y=0; y < 16; y++)
      {
//    if(!(((Individuals[0].ORT_PT[m]>>y)&1)==0) &&
(((Individuals[0].ORT_PT[m]>>(y+1))&1)==0) &&
(((Individuals[0].ORT_PT[m]>>(y+2))&1)==0) &&
(((Individuals[0].ORT_PT[m]>>(y+3))&1)==0))
//    {
      //srand((rand()%65535));
      r = rand()%2;
      Individualsp->ORT_PT[m] += r*(1<<(y));
      r = rand()%2;
      Individuals1p->ORT_PT[m] += r*(1<<(y));
//    }
      }
      }
//    sprintf(string, "%x %x %x %x %x %x %x %x %x %x %x %x %x\n\r",
Individualsp->IN_PT[0], Individualsp->IN_PT[1], Individualsp->IN_PT[2],
Individualsp->IN_PT[3], Individualsp->IN_PT[4], Individualsp->IN_PT[5],
Individualsp->IN_PT[6], Individualsp->IN_PT[7], Individualsp->ORT_PT[0],
Individualsp->ORT_PT[1], Individualsp->ORT_PT[2], Individualsp->ORT_PT[3],
Individualsp->active_lines);
//    UART_1_PutString(string);
      }
      /***************************writing back the configuration to
corresponding registers and fitness calculation************************/
      m=0;
      while(fitness_max[0] > 0.5 && iteration1 != G)
      {
            srand(labs(rand()+iteration1)%65535);
            for(z=0; z<N_digi; z++)
            {     m=0;
            Individualsp = &Individuals[z];
            for(j=0, i=BEGIN_DET_IT; i<BEGIN_DET_ORT; j++, i += 4)
            {if((((Individualsp->active_lines)>>j)&1) != 0)
            {CY_SET_REG32(i, Individualsp->IN_PT[m]);
            m++;
            }
            }
            m=0;
            for(i=BEGIN_DET_ORT, j=12; i<END_DET_ITORT; j++, i += 2)
            {if((((Individualsp->active_lines)>>j)&1) != 0)
            {CY_SET_REG16(i, Individualsp->ORT_PT[m]);
            m++;}
            }

            m=0;
            Individuals1p = &Individuals2[z];
            for(j=0, i=BEGIN_DET_IT2; i<BEGIN_DET_ORT2; j++, i += 4)
            {if((((Individuals1p->active_lines)>>j)&1) != 0)
            {CY_SET_REG32(i, Individuals1p->IN_PT[m]);
            m++;
            }
            }
            m=0;
            for(i=BEGIN_DET_ORT2, j=12; i<END_DET_ITORT2; j++, i += 2)
            {if((((Individuals1p->active_lines)>>j)&1) != 0)
            {CY_SET_REG16(i, Individuals1p->ORT_PT[m]);
            m++;}
            }
            m=0;


            /****fitness calcuation***/
      CY_SET_REG16(CYREG_B0_P0_U0_MC_CFG_BYPASS, 0x0055);
      CY_SET_REG16(CYREG_B0_P0_U0_MC_CFG_CEN_CONST, 0x0000);
      CY_SET_REG16(CYREG_B0_P1_U1_MC_CFG_BYPASS, 0x0055);
      CY_SET_REG16(CYREG_B0_P1_U1_MC_CFG_CEN_CONST, 0x0000);

      //CyDelay(250);
      D=0;
      for(i=0; i< 255; i++)
      {for(m=0; m<8; m++)
      {switch(m)
      {
      case 0:
      if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_1_0);}
      else {CyPins_ClearPin(Pin_1_0);}
      break;
      case 1:
      if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_1_1);}
      else {CyPins_ClearPin(Pin_1_1);}
      break;
      case 2:
      if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_1_2);}
      else {CyPins_ClearPin(Pin_1_2);}
      break;
      case 3:
      if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_1_3);}
      else {CyPins_ClearPin(Pin_1_3);}
      break;
      case 4:
      if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_2_0);}
      else {CyPins_ClearPin(Pin_2_0);}
      break;
      case 5:
      if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_2_1);}
      else {CyPins_ClearPin(Pin_2_1);}
      break;
      case 6:
      if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_2_2);}
      else {CyPins_ClearPin(Pin_2_2);}
      break;
      case 7:
      if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_2_3);}
      else {CyPins_ClearPin(Pin_2_3);}
      break;
      }
      }
      out[0] = CyPins_ReadPin(Pin_3_0);
      out[1] = CyPins_ReadPin(Pin_3_1);
      out[2] = CyPins_ReadPin(Pin_3_2);
      out[3] = CyPins_ReadPin(Pin_3_3);
      out[4] = CyPins_ReadPin(Pin_4_0);
      out[5] = CyPins_ReadPin(Pin_4_1);
      out[6] = CyPins_ReadPin(Pin_4_2);
      out[7] = CyPins_ReadPin(Pin_4_3);
      D=0;
      for(m=0; m<8; m++)
      {if(out[m] == 16) {out[m] = 1;}
            D += out[m]*pos_array[m];}
      //D = D/16;
      difference[i] = fabsf(outputs[i] - norm_diff[D]);

//      if(z==0){if(iteration==0){sprintf(string, "%d %f %f %f\n\r", D,
bestAnalog->norm_diff[D], bestAnalog->outputs[i], bestAnalog->oracle[i]);
//      UART_1_PutString(string);}
//      }
```

```
      fitness[z] += fabsf(oracle[i] - (difference[i]));
      D=0;}
      iteration++;}
            tot_oracle =0; tot_diff =0;
//          for(i=0; i<256; i++){sprintf(string, "difference vs oracle %f
%f\n", difference[i], bestAnalog->oracle[i]);
//          UART_1_PutString(string);
//          tot_oracle += bestAnalog->oracle[i];
//          tot_diff += difference[i];}
//          sprintf(string, "net fitness %f sum of oracle %f sum of
differences %f\n", fabsf(tot_oracle - tot_diff), tot_oracle, tot_diff);
//          UART_1_PutString(string);
            fitness_avg = 0;
      for(z=0; z<N_digi; z++)
      {
            if(fitness[z] < fitness_max[0])
            {
            Individuals[1] = Individuals[0];
            Individuals[0] = Individuals[z];
            Individuals2[1] = Individuals2[0];
            Individuals2[0] = Individuals2[z];
            fitness_max[1] = fitness_max[0];
        fitness_max[0] = fitness[z];
            }
            else if(fitness[z] == fitness_max[0]){}
            else if(fitness[z] > fitness_max[0]  && fitness[z] <=
fitness_max[1])
            {fitness_max[1] = fitness[z];
            Individuals[1] = Individuals[z];
            Individuals2[1] = Individuals2[z];
            }
            fitness_avg = fitness_avg +fitness[z];
      }
      if(iteration1%50 == 0){
      if(fitness_max[0] == fitness_max_prev)
      {stasis = 1;}
      else {stasis = 0;}
      fitness_max_prev = fitness_max[0];}

      c = fitness_max[0];
      LCD_WriteControl(LCD_CLEAR_DISPLAY );
      LCD_PrintString("M_F ");
      LCD_PrintNumber(c);
      LCD_PrintString(" G ");
      LCD_PrintNumber(iteration1);
      fitness_avg = fitness_avg/N_digi;
      LCD_Position(1,0);
      LCD_PrintNumber(fitness_avg);
      sprintf(string, "%f, %f, %f, %f\n\r",  fitness_max[1], fitness[0],
fitness_avg, c);
      UART_1_PutString(string);
      if(iteration1 == 0) {LCD_WriteControl(LCD_CLEAR_DISPLAY );
```

```
            c = fitness[0];
      LCD_PrintNumber(c);
      CyDelay(1000);
      }
      if(fitness[z] > 1 || iteration1 < G) {it_max = iteration1;}


      /****************************crossover*************************/
      //crossover between each pair of the elites chosen for crossover
      //crossover at half point
      //individuals 0 and 1
            j=0;
            N1 = rand()%8;
            N2 = rand()%4;
            z2=2;
            m=0;
            fit_t_max=1000;

            for(c4=1,c5=1; c4<40; c4++, c5++)
            {for(c3=c4*2; c3<(c4+1)*2; c3++)
            {if(fitness[c3]<fit_t_max)
            {fit_t_max = fitness[c3];
            Individualsp = &Individuals[c3];
            Individuals1p = &Individuals2[c3];}
            }
            fitter_indi[c5] = *Individualsp;
            fitter_indi2[c5] = *Individuals1p;
            fit_t_max=1000;}

            fitter_indi[0] = Individuals[0];
            fitter_indi2[0] = Individuals2[0];

            if(iteration1%200 == 0)
      {     for(z = 1; z < N_digi; z++)
      {Individualsp = &Individuals[z];
      Individuals1p = &Individuals2[z];
      Individuals[z] = Individuals[0]; //ensures that the active lines are
copied as is
      Individuals2[z] = Individuals2[0];
      for(m=0; m<8; m++)
      {
      for(y = 0; y < 32; y++)
      {
      //srand((rand()%65535));
      r = rand()%2; //generate a random number - 0 or 1
      Individualsp->IN_PT[m] = 0;
      Individualsp->IN_PT[m] += r*(1<<(y));
      r = rand()%2; //generate a random number - 0 or 1
      Individuals1p->IN_PT[m] = 0;
      Individuals1p->IN_PT[m] += r*(1<<(y));
      }
      }
      for(m=0; m<4; m++)
      {for(y=0; y < 16; y++)
      {
//    if(!(((Individuals[0].ORT_PT[m]>>y)&1)==0) &&
(((Individuals[0].ORT_PT[m]>>(y+1))&1)==0) &&
(((Individuals[0].ORT_PT[m]>>(y+2))&1)==0) &&
(((Individuals[0].ORT_PT[m]>>(y+3))&1)==0))
//    {
      //srand((rand()%65535));
      r = rand()%2;
      Individualsp->ORT_PT[m] = 0;
      Individualsp->ORT_PT[m] += r*(1<<(y));
      r = rand()%2;
      Individuals1p->ORT_PT[m] = 0;
      Individuals1p->ORT_PT[m] += r*(1<<(y));
//    }
      }
      }
//    sprintf(string, "%x %x %x %x %x %x %x %x %x %x %x %x %x\n\r",
Individualsp->IN_PT[0], Individualsp->IN_PT[1], Individualsp->IN_PT[2],
Individualsp->IN_PT[3], Individualsp->IN_PT[4], Individualsp->IN_PT[5],
Individualsp->IN_PT[6], Individualsp->IN_PT[7], Individualsp->ORT_PT[0],
Individualsp->ORT_PT[1], Individualsp->ORT_PT[2], Individualsp->ORT_PT[3],
Individualsp->active_lines);
//    UART_1_PutString(string);
      }
      }

            z2=N_digi-N_repl; z=0;
            while(z2<N_digi)
            {if (Xovertype == 1){zf = rand()%(N_digi/2)+1;}
            if (Xovertype == 0){zf = z;}
            indp1 = &fitter_indi[zf];
            ind1p1 = &fitter_indi2[zf];
            z1 = rand()%N_digi+1;
            indp2 = &Individuals[z1];
            ind1p2 = &Individuals2[z1];
            if((indp1->IN_PT != indp2->IN_PT || indp1->ORT_PT != indp2-
>ORT_PT) || (ind1p1->IN_PT != ind1p2->IN_PT || ind1p1->ORT_PT != ind1p2-
>ORT_PT))
            {indp2 = &Individuals[z1];
            ind1p2 = &Individuals2[z1];}
            else{z1++;
            indp2 = &Individuals[z1];
            ind1p2 = &Individuals2[z1];}
                  new_individual.active_lines = Individuals[0].active_lines;
//to copy the active lines as is
                  new_individual2.active_lines =
Individuals2[0].active_lines;
                  new_individualp = &new_individual;
                  new_individual2p = &new_individual2;
                  j=0;
                  for(c2=0; c2<N1; c2++){new_individualp->IN_PT[c2] = indp1-
>IN_PT[c2];
                  new_individual2p->IN_PT[c2] = ind1p1->IN_PT[c2];}
                  j=0;
                  for(c2=N1; c2<8; c2++){new_individualp->IN_PT[c2] = indp2-
>IN_PT[c2];
                  new_individual2p->IN_PT[c2] = ind1p2->IN_PT[c2];}

                  j=0; c2=0;

                  for(c2=0; c2<N2; c2++){new_individualp->ORT_PT[c2] = indp1-
>ORT_PT[c2];
                  new_individual2p->ORT_PT[c2] = ind1p1->ORT_PT[c2];}

                  j=0;

                  for(c2=N2; c2<4; c2++){new_individualp->ORT_PT[c2] = indp2-
>ORT_PT[c2];
                  new_individual2p->ORT_PT[c2] = ind1p2->ORT_PT[c2];}
                  Individuals[z2] = *new_individualp;
                  Individuals2[z2] = *new_individual2p;
                  z2++; z++;
                  m=0;
            }

            for(z=2; z<N_digi-N_repl; z++){Individuals[z] = fitter_indi[z];
            Individuals2[z] = fitter_indi2[z];}

            m=0;
      /**************check these out**************/
      //crossover at boundaries of IT_PT or ORT_PT
      //crossover at random point
      //UART_PutArray(Individuals[fittest[0]].IN_PT[i],1);
      /*****************************mutation*******************************/
      // mutation on New Indidivual
      m=0; j=0;
      //srand((rand()%65535));
      if(fitness_max[0] > 0 || iteration1 < G)
      {for(z=2; z<N_digi; z++)
      {Individualsp = &Individuals[z];
      Individuals1p = &Individuals2[z];
      N3=N3_init; N4=N4_init;
      if(fitness_avg-fitness_max[0]>Ulim && stasis==0){N3=0; N4=0;}
      if(fitness_avg-fitness_max[0]<Llim && stasis==1){N3+=N3_incr;
N4+=N4_incr;}
      for(m=0; m<8; m++)
      {for(i=0; i<32; i++)
      {
      if(GradMut == 0) {x=rand()%1000;}
      if(GradMut == 1) {x=rand()%10000;}
      j=1<<i;
      if(x<N3){if(((Individualsp->IN_PT[m]>>i)&1)==1){Individualsp->IN_PT[m]
-= j;}
      else if(((Individualsp->IN_PT[m]>>i)&1)==0){Individualsp->IN_PT[m]
+=j;}
      if(((Individuals1p->IN_PT[m]>>i)&1)==1){Individuals1p->IN_PT[m] -= j;}
      else if(((Individuals1p->IN_PT[m]>>i)&1)==0){Individuals1p->IN_PT[m]
+=j;}}
      }
      }
      for(m=0; m<4; m++)
      {for(i=0; i<16; i++)
      {
      if(GradMut == 0) {x=rand()%1000;}
      if(GradMut == 1) {x=rand()%10000;}
      j=1<<i;
      if(x<N4){if(((Individualsp->ORT_PT[m]>>i)&1)==1){Individualsp-
>ORT_PT[m] -= j;}
      else if(((Individualsp->ORT_PT[m]>>i)&1)==0){Individualsp->ORT_PT[m]
+=j;}
      if(((Individuals1p->ORT_PT[m]>>i)&1)==1){Individuals1p->ORT_PT[m] -=
j;}
      else if(((Individuals1p->ORT_PT[m]>>i)&1)==0){Individuals1p->ORT_PT[m]
+=j;}}
      }
      }
      }
      }
      for(z=0; z<N_digi; z++) {fitness[z] = 0;
      //Individualsp = &Individuals[z];
//    if(iteration1%10==0 || iteration1==1 || iteration1==2){sprintf(string,
"%x %x %x %x %x %x %x %x %x %x %x %x %x\n\r", Individualsp->IN_PT[0],
Individualsp->IN_PT[1], Individualsp->IN_PT[2], Individualsp->IN_PT[3],
Individualsp->IN_PT[4], Individualsp->IN_PT[5], Individualsp->IN_PT[6],
Individualsp->IN_PT[7], Individualsp->ORT_PT[0], Individualsp->ORT_PT[1],
Individualsp->ORT_PT[2], Individualsp->ORT_PT[3], Individualsp-
>active_lines);
//    UART_1_PutString(string);}
      }

      iteration1++;
      }
//    for(i=0; i<256; i++){sprintf(string, "difference vs oracle %f %f\n",
difference[i], bestAnalog->oracle[i]);
//    UART_1_PutString(string);}
      /****************PHASE 3: Writing back the configuration to the
registers - storing successful configurations in SRAM for each completed GA
run********************/
            m=0;
            Individualsp = &Individuals[0];
            Individuals1p = &Individuals2[0];
            for(j=0, i=BEGIN_DET_IT; i<BEGIN_DET_ORT; j++, i += 4)
            {if((((Individualsp->active_lines)>>j)&1) != 0)
            {CY_SET_REG32(i, Individualsp->IN_PT[m]);
            m++;}
            }
            m=0;
            for(i=BEGIN_DET_ORT, j=12; i<END_DET_ITORT; j++, i += 2)
            {if((((Individualsp->active_lines)>>j)&1) != 0)
            {CY_SET_REG16(i, Individualsp->ORT_PT[m]);
            m++;}
            }
            m=0;

            CY_SET_REG16(CYREG_B0_P0_U0_MC_CFG_BYPASS, 0x0055);
            CY_SET_REG16(CYREG_B0_P0_U0_MC_CFG_CEN_CONST, 0x0000);
            for(j=0, i=BEGIN_DET_IT2; i<BEGIN_DET_ORT2; j++, i += 4)
            {if((((Individuals1p->active_lines)>>j)&1) != 0)
            {CY_SET_REG32(i, Individuals1p->IN_PT[m]);
            m++;}
            }
            m=0;
            for(i=BEGIN_DET_ORT2, j=12; i<END_DET_ITORT2; j++, i += 2)
            {if((((Individuals1p->active_lines)>>j)&1) != 0)
            {CY_SET_REG16(i, Individuals1p->ORT_PT[m]);
            m++;}
            }
            m=0;
      CY_SET_REG16(CYREG_B0_P1_U1_MC_CFG_BYPASS, 0x0055);
      CY_SET_REG16(CYREG_B0_P1_U1_MC_CFG_CEN_CONST, 0x0000);
      //Output final configuration to the UART and for storing away in the
computer
      /* Place your application code here. */
            /******testing*******/
      fitness_last=0; D=0;
      for(i=0; i< 255; i++)
      {for(m=0; m< 8; m++)
      {switch(m){
      case(0):
      {if (((i>>m)&1) == 1) {CyPins_SetPin(Pin_1_0);}
      else {CyPins_ClearPin(Pin_1_0);}
      break;}
      case(1):
      {if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_1_1);}
      else {CyPins_ClearPin(Pin_1_1);}
      break;}
      case(2):
      {if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_1_2);}
      else {CyPins_ClearPin(Pin_1_2);}
      break;}
      case(3):
      {if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_1_3);}
      else {CyPins_ClearPin(Pin_1_3);}
      break;}
      case(4):
```

```
      {if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_2_0);}
      else {CyPins_ClearPin(Pin_2_0);}
      break;}
      case(5):
      {if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_2_1);}
      else {CyPins_ClearPin(Pin_2_1);}
      break;}
      case(6):
      {if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_2_2);}
      else {CyPins_ClearPin(Pin_2_2);}
      break;}
      case(7):
      {if(((i>>m)&1) == 1) {CyPins_SetPin(Pin_2_3);}
      else {CyPins_ClearPin(Pin_2_3);}
      break;}
      }
      }
      out[0] = CyPins_ReadPin(Pin_3_0);
      out[1] = CyPins_ReadPin(Pin_3_1);
      out[2] = CyPins_ReadPin(Pin_3_2);
      out[3] = CyPins_ReadPin(Pin_3_3);
      out[4] = CyPins_ReadPin(Pin_4_0);
      out[5] = CyPins_ReadPin(Pin_4_1);
      out[6] = CyPins_ReadPin(Pin_4_2);
      out[7] = CyPins_ReadPin(Pin_4_3);
      D=0;
      for(m=0; m<8; m++)
      {if(out[m]!=0){out[m] = 1;}
      D += (out[m])*pos_array[m];}
    D_array[i] = D;
//    if(y_array[i] > (i - D))
//    {fitness_last += (y_array[i] - (i - D));}
//    else {fitness_last += ((i - D) - y_array[i]);}
      fitness_last += fabsf(oracle[i] - (outputs[i] - norm_diff[D]));
      D=0;}
      /******testing*******/
       //c3 = fitness_max[3],
      LCD_WriteControl(LCD_CLEAR_DISPLAY );;
      LCD_PrintString(" ");
      LCD_PrintNumber(c);
      LCD_PrintString(" ");
      LCD_PrintNumber(c1);
      LCD_PrintString(" ");
      LCD_PrintNumber(it_max);
      CyDelay(1000);
      LCD_PrintNumber(fitness_last);
      CyDelay(1000);
      sprintf(string, "Max fitness %f Generations %ld Average fitness
%f\n\r", fitness_last, it_max, fitness_avg);
      UART_1_PutString(string);
      //UART_1_Stop();
      //LCD_PrintString("Phase 3 ");
    float32 averageError_analog, averageError_digital, totalError_analog,
totalError_digital;
//    for(;;)
//    {

//        for(i=0; i<255; i++){
//            sprintf(string,"%f,%f\n\r", outputs[i], outputs[i] -
norm_diff[D_array[i]]);
//            UART_1_PutString(string);
//        }
//        averageError_analog = 0;
//        totalError_analog = 0;
//        for(i=0;i<255;i++){
//            totalError_analog += fabs(outputs[i] - oracle[i]);
//        }
//        averageError_analog = totalError_analog/255;
//
//        averageError_digital = 0;
//        totalError_digital = 0;
//        for(i=0;i<255;i++){
//            totalError_digital += fabs(oracle[i] - (outputs[i] -
norm_diff[D_array[i]]));
//        }
//        averageError_digital = totalError_digital/255;

            uint32 analog_fitness, digital_fitness, AvgA_fit, AvgD_fit,
max_dev;
            for(i=0; i<256; i++){
      sprintf(string, "%f %d %f\n\r", oracle[i], D_array[i], outputs[i]-
norm_diff[D_array[i]]);
      UART_1_PutString(string);
      analog_fitness += fabsf(oracle[i] - outputs[i]);
      digital_fitness += fabsf(oracle[i] - (outputs[i] -
norm_diff[D_array[i]]));
      difference[i] = fabsf(oracle[i] - (outputs[i] -
norm_diff[D_array[i]]));}
      AvgA_fit = analog_fitness/256; AvgD_fit = digital_fitness/256;
      for(i=0; i<255; i++) {if(difference[i]>max_dev) {max_dev =
difference[i];}}
      sprintf(string, "max deviation: %f\n", max_dev);
      UART_1_PutString(string);
      sprintf(string, "Analog: %f Digital:%f Average_Analog:%f
Average_Digital:%f % Imp_overall %f % Imp_average %f\n\r", analog_fitness,
digital_fitness, AvgA_fit, AvgD_fit, ((analog_fitness -
digital_fitness)/analog_fitness)*100, ((AvgA_fit - AvgD_fit)/AvgA_fit)*100);
      UART_1_PutString(string);
//    CyPins_SetPin(Pin_6_0);
//    CyPins_SetPin(Pin_7_0);
//    CyPins_SetPin(Pin_8_0);
//    CyPins_SetPin(Pin_9_0);
//    CyPins_SetPin(Pin_10_0);
//    CyPins_SetPin(Pin_11_0);
//    CyPins_SetPin(Pin_12_0);
//          }
    return 0;
}

      /* [] END OF FILE */
```

APPENDIX B: SAMPLE CPGA CODE

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF Vignesh Thangavel.
 *
 * ========================================
*/
//#include <project.h>
#include <root_fourth.h>
#include <root_third.h>
#include <sqrt.h>
#include <x.h>
#include <x2.h>
#include <x3.h>
#include <x4.h>

//#include <ga_def.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <LCD.h>
#include <UART_1.h>

#define N_digi 80
#define start_seed 50
#define G 500 //1500 fits in one screen

#define N3_init 1
#define N3_incr 0.1
#define Ulim 1000
#define Llim 50
#define GradMut 0
#define Xovertype 1
#define N_repl 60

struct CPGAIndividual{
      float coeff[8];
};
struct CPGAIndividual Individuals[N_digi], new_individual,
fitter_indi[N_digi/2], fittest_indi, fresh_indi;
struct CPGAIndividual *Individualsp;
struct CPGAIndividual *indp1;
struct CPGAIndividual *indp2;
struct CPGAIndividual *new_individualp;
//struct CPGAIndividual *Individuals1p;
//struct CPGAIndividual *ind1p1;
//struct CPGAIndividual *ind1p2;
//struct CPGAIndividual *new_individual2p;
```

```
float A_largest, C_largest, C_largest_new;
int main()
{     uint32 i=0, j=0;
      uint16 iteration1 = 0;
      uint8  stasis = 0, x=0, m=0, zf=0, z=0, z1=0, z2=0, N1=100, c2 = 0;
      uint8 c3=0, c4=0, c5=0, sel_indi=0;
      float fitness[N_digi]={0}, fitness_avg = 0, fitness_max_prev = 1000,
fitness_max[2]={2000000000, 20000000000}, fitness_last=0, fit_t_max=1000;
      float c = 0, difference[256] = {0}, N3=0, oracle_CPGA[256]={0},
result[256] = {0};
      float DDC_zero[256] = {1};
      float fitness_max_prev_1=0, avg_fitness_prev=2000000000;
//    float32 resultf = 0;

      /*********Range Scaling**************/
      A_largest = 277.102600;
      C_largest = 0x7F7FFFF/(8*A_largest);

      /**********Initialize I/O and compute oracle*************/
      UART_1_Start();
      char string [100];
      LCD_Start();
      LCD_WriteControl(LCD_CLEAR_DISPLAY );
      srand(start_seed);
      for(i=0; i<256; i++)
      {
//    oracle_CPGA[i] = oracle_cube[i];
      oracle_CPGA[i] = (float)sin((double)oracle_first[i]);
      }
      sprintf(string, "A_largest :%f C_largest %f oracle_fourth[255]
:%f\n\r", A_largest, C_largest, oracle_fourth[255]);
      UART_1_PutString(string);

      Individualsp = &Individuals[0];
      for(j=0; j<8; j++)
      {Individualsp->coeff[j] = 0;}
      sprintf(string, "Individual 0: %f\n\r", Individualsp->coeff[j]);
      UART_1_PutString(string);
//
//    sprintf(string, "Individual 0: %f\n\r", Individualsp->coeff[j]);
//    UART_1_PutString(string);
      /*******Random Initialization of Individuals*******/

      for(z=1; z<N_digi; z++)
      {     for(j=0; j<8; j++)
            {Individualsp = &Individuals[z];
            Individualsp->coeff[j] =
(((float)rand()/(float)(RAND_MAX))*(2*C_largest))-(C_largest);
            }
      }
      C_largest_new = C_largest;
      sprintf(string, "random number update: %f\n\r",
Individuals[1].coeff[5]);
      UART_1_PutString(string);

      /********Fitness Evaluation of Individuals********/
m=0;
while(fitness_max[0] > 0.5 && iteration1 != G)
{
      for(z=0; z<N_digi; z++)
      {Individualsp = &Individuals[z];
      for(i=0; i<256; i++){result[i] = 0;}
            for(j=0; j< 8; j++)
            {switch(j){
                  case(0):
                  {     for(i=0; i<256; i++)
                        {result[i] += Individualsp-
>coeff[j]*DDC_fourthrt[i];} //done
//                            sprintf(string, "result %lu : %f\n\r", i,
result[i]);
//                            UART_1_PutString(string);
                  break;}
                  case(1):
                  {     for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_cubert[i];}
//done
//                            sprintf(string, "result %lu : %f\n\r", i,
result[i]);
//                            UART_1_PutString(string);
                  break;}
                  case(2):
                  {     for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_sqrt[i];}
//done
//                            sprintf(string, "result %lu : %f\n\r", i,
result[i]);
//                            UART_1_PutString(string);
                  break;}
                  case(3):
                  {     for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_zero[i];}
//done
//                      sprintf(string, "result %lu : %f\n\r", i, result[i]);
//                            UART_1_PutString(string);
                  break;}
                  case(4):
                  {     for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_one[i];}
//done
//                      sprintf(string, "result %lu : %f\n\r", i, result[i]);
//                            UART_1_PutString(string);
                  break;}
                  case(5):
                    {      for(i=0; i<256; i++)
                           {result[i] += Individualsp->coeff[j]*DDC_sq[i];}
//done
//                         sprintf(string, "result %lu : %f\n\r", i, result[i]);
//                                UART_1_PutString(string);
                    break;}
                    case(6):
                    {      for(i=0; i<256; i++)
                           {result[i] += Individualsp->coeff[j]*DDC_cube[i];}
//done
//                         sprintf(string, "result %lu : %f\n\r", i, result[i]);
//                                UART_1_PutString(string);
                    break;}
                    case(7):
                    {      for(i=0; i<256; i++)
                           {result[i] += Individualsp->coeff[j]*DDC_fourth[i];}
//done
//                         sprintf(string, "result %lu : %f\n\r", i, result[i]);
//                                UART_1_PutString(string);
                    break;}
                    }
             }
             for(i=0; i<256; i++)
             {fitness[z] += fabsf(oracle_CPGA[i] - result[i]);}
//           sprintf(string, "fitness %d : %f\n\r", z, fitness[z]);
//           UART_1_PutString(string);
      }

      /*****************Elitism and selection of fitter
individuals***************/
      fitness_avg = 0;
      for(z=0; z<N_digi; z++)
      {
             if(fitness[z] < fitness_max[0])
             {
             Individuals[1] = Individuals[0];
             Individuals[0] = Individuals[z];
             fitness_max[1] = fitness_max[0];
        fitness_max[0] = fitness[z];
             }
             else if(fitness[z] == fitness_max[0]){}
             else if(fitness[z] > fitness_max[0]  && fitness[z] <=
fitness_max[1])
             {fitness_max[1] = fitness[z];
             Individuals[1] = Individuals[z];
             }
             fitness_avg = fitness_avg +fitness[z];
      }
      if(iteration1%50 == 0){
      if(fitness_max[0] == fitness_max_prev)
      {stasis = 1;}
      else {stasis = 0;}
      fitness_max_prev = fitness_max[0];}

//    if(iteration1%5 == 0){
//          if(avg_fitness_prev > fitness_avg)
//          {fitness_max_prev_1 = fitness_max[0];}
//    }
//
//    avg_fitness_prev = fitness_avg;
      /**************Print stuff****************/
      c = fitness_max[0];
      if(c>100) {
            C_largest_new = C_largest_new/2;
            for(z=10; z<N_digi; z++)
      {     for(j=0; j<8; j++)
            {Individualsp = &Individuals[z];
            Individualsp->coeff[j] =
(((float)rand()/(float)(RAND_MAX))*(2*C_largest_new))-(C_largest_new);
            }
      }
      }

//    if(c>10 && c<100) {
//          C_largest_new = C_largest_new/2;
//          for(z=(3*N_digi)/4; z<N_digi; z++)
//    {     for(j=0; j<8; j++)
//          {Individualsp = &Individuals[z];
//          Individualsp->coeff[j] =
(((float)rand()/(float)(RAND_MAX))*(2*C_largest_new))-(C_largest_new);
//          }
//    }
//    }

      if(c<20) {
            C_largest_new = C_largest_new/2;
            for(z=N_digi-10; z<N_digi; z++)
      {     for(j=0; j<8; j++)
            {Individualsp = &Individuals[z];
            Individualsp->coeff[j] =
(((float)rand()/(float)(RAND_MAX))*(2*C_largest_new))-(C_largest_new);
            }
      }
      }

      LCD_WriteControl(LCD_CLEAR_DISPLAY );
      LCD_PrintString("M_F ");
      LCD_PrintNumber(c);
      LCD_PrintString(" G ");
      LCD_PrintNumber(iteration1);
      fitness_avg = fitness_avg/N_digi;
      LCD_Position(1,0);
      LCD_PrintNumber(fitness_avg);
      sprintf(string, "%f, %f, %f, %f\n\r", fitness_max[1], fitness[0],
fitness_avg, c);
      UART_1_PutString(string);
      if(iteration1 == 0) {LCD_WriteControl(LCD_CLEAR_DISPLAY );
            c = fitness[0];
      LCD_PrintNumber(c);
      CyDelay(1000);
      }
//    if(fitness[z] > 1 || iteration1 < G) {it_max = iteration1;}

      /****************Crossover***************/
      j=0;
            N1 = rand()%8;
            z2=2;
            m=0;
            fit_t_max=1000;

            for(c4=1,c5=1; c4<40; c4++, c5++)
            {for(c3=c4*2; c3<(c4+1)*2; c3++)
            {if(fitness[c3]<fit_t_max)
            {fit_t_max = fitness[c3];
            Individualsp = &Individuals[c3];}
            }
            fitter_indi[c5] = *Individualsp;
            fit_t_max=1000;}

            fitter_indi[0] = Individuals[0];

            z2=N_digi-N_repl; z=0;
            while(z2<N_digi)
            {if (Xovertype == 1){zf = rand()%(N_digi/2)+1;}
            if (Xovertype == 0){zf = z;}
            indp1 = &fitter_indi[zf];
            z1 = rand()%N_digi+1;
            indp2 = &Individuals[z1];
            if(indp1->coeff != indp2->coeff)
            {indp2 = &Individuals[z1];}
            else{z1++;
            indp2 = &Individuals[z1];}
                  new_individualp = &new_individual;
                  j=0;
                  for(c2=0; c2<N1; c2++){new_individualp->coeff[c2] = indp1-
>coeff[c2];}
                  j=0;
                  for(c2=N1; c2<8; c2++){new_individualp->coeff[c2] = indp2-
>coeff[c2];}

                  j=0; c2=0;
                  Individuals[z2] = *new_individualp;
                  z2++; z++;
                  m=0;
            }
            for(z=2; z<N_digi-N_repl; z++){Individuals[z] = fitter_indi[z];}

      /****************Mutation****************/
            m=0; j=0;
      //srand((rand()%65535));
      if(fitness_max[0] > 0 || iteration1 < G)
      {for(z=2; z<N_digi; z++)
      {Individualsp = &Individuals[z];
      N3=N3_init;
      if(fitness_avg-fitness_max[0]>Ulim && stasis==0){N3=0;}
      else if(fitness_avg-fitness_max[0]<Llim && stasis==1){N3+=N3_incr;}
      else if(fitness_avg-fitness_max[0]>Ulim && stasis==1){N3+=N3_incr;}
      for(m=0; m<8; m++)
      {
      if(GradMut == 0) {x=rand()%1000;}
      if(GradMut == 1) {x=rand()%10000;}
//    j=1<<i;
      if(x<N3){if(stasis==0){Individualsp->coeff[j] +=
(((float)rand()/(float)(RAND_MAX))*8)-4;}
      if(stasis==1){Individualsp->coeff[j] +=
(((float)rand()/(float)(RAND_MAX))*(8+((iteration1)/50)))-
(4+((iteration1)/50));}
      }
      }
      }
      for(i=0; i<10; i++)
      {sel_indi = rand()%N_digi;
      if(sel_indi !=0 && sel_indi !=1)
      {Individualsp = &Individuals[sel_indi];
      for(j=0; j<8; j++)
      {x=rand()%1000;
      if(x<N3) {Individualsp->coeff[j] = (Individualsp->coeff[j])*(-1);}
      }
      }
      }
      }

      for(z=0; z<N_digi; z++) {fitness[z] = 0;}

      iteration1++;
}
      /*****Add a small random number to the existing value and do sign bit
flip mutation for a very few individuals - upto 5*******/

            /****************Final fitness evlatuation and
error******************/
      Individualsp = &Individuals[0];
      for(i=0; i<256; i++){result[i] = 0;}
            for(j=0; j< 8; j++)
            {switch(j){
                  case(0):
                  {       for(i=0; i<256; i++)
                        {result[i] += Individualsp-
>coeff[j]*DDC_fourthrt[i];}
                  break;}
                  case(1):
                  {       for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_cubert[i];}
                  break;}
                  case(2):
                  {       for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_sqrt[i];}
                  break;}
                  case(3):
                  {       for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_zero[i];}
                  break;}
                  case(4):
                  {       for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_one[i];}
//done
                  break;}
                  case(5):
                  {       for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_sq[i];}
                  break;}
                  case(6):
                  {       for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_cube[i];}
//done
                  break;}
                  case(7):
                  {       for(i=0; i<256; i++)
                        {result[i] += Individualsp->coeff[j]*DDC_fourth[i];}
                  break;}
                  }
            }
            for(i=0; i<256; i++)
            {fitness_last += fabsf(oracle_CPGA[i] - result[i]);}

            sprintf(string, "best fitness acheived: %f\n", fitness_last);
            UART_1_PutString(string);
            sprintf(string, "chromosome of best individual:\nx^1/4:
%f\nx^1/3: %f\nx^1/2: %f\nx^0: %f\nx: %f\nx^2: %f\nx^3: %f\nx^4: %f\n\r",
Individuals[0].coeff[0], Individuals[0].coeff[1], Individuals[0].coeff[2],
Individuals[0].coeff[3], Individuals[0].coeff[4], Individuals[0].coeff[5],
Individuals[0].coeff[6], Individuals[0].coeff[7]);
            UART_1_PutString(string);
      uint32 digital_fitness=0, AvgD_fit=0, max_dev=0;
      for(i=0; i<256; i++) {if(difference[i]>max_dev) {max_dev =
difference[i];}}
      sprintf(string, "max deviation: %d\n", (int)max_dev);
      UART_1_PutString(string);
```

```
	sprintf(string, "Digital:%f Average_Digital:%f \n\r",
(double)digital_fitness, (double)AvgD_fit);
	UART_1_PutString(string);

	return 0;
}

	/* [] END OF FILE */
```

LIST OF REFERENCES

[1] Michael B. Taylor. “Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse,” *Proc. of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 1131-1136.

[2] P. Hasler and D.V. Anderson, "Cooperative analog-digital signal processing," *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol.4, pp.IV-3972 - IV-3975, 13-17 May 2002.

[3] David Beasley, R.R. Martin, and D.R. Bull, *An overview of genetic algorithms: Fundamentals*, University Computing,Volume 15, No. 2, 1993. Pg 58-68.

[4] Sekanina, L.; Vasicek, Z., "Approximate circuit design by means of evolvable hardware," *Evolvable Systems (ICES),2013 IEEE International Conference on* , vol., no., pp.21,28, 16-19 April 2013.

[5] Adrian Thompson, "Silicon Evolution," *Proceedings of the First Annual Conference on Genetic Programming*, MIT press, 1996.

[6] *PSoC 5LP Architecture TRM*, Cypress Semiconductor, San Jose, CA, 2014, pp. 299.

[7] R. Al-Haddad, R. Oreifej, R. A. Ashraf, and R. F. DeMara, "Sustainable Modular Adaptive Redundancy Technique Emphasizing Partial Reconfiguration for Reduced Power Consumption," *International Journal of Reconfigurable Computing*, vol. 2011, Article ID 430808, 25 pages, 2011.

[8] Mitchell, M., Forrest, S., & Holland, J.H. (1992), “The royal road for genetic algorithms: Fitness landscapes and GA performance,” *Proceedings of the First European Conference on Artificial Life*. Cambridge, MA: MIT Press/Bradford

Books.

[9] T. Jansen and I. Wegener. *Real royal road functions: where crossover provably is essential.* Discrete Applied Mathematics, 149(1-3):111–125, 2005

[10] Lohn, J. D., G. L. Haith, S. P. Columbano, and D. Stassinopoulos (1999), "A comparison of dynamic fitness schedules for evolutionary design of amplifiers," *Proc. 1st NASA/DoD Workshop of Evolvable Hardware*, pp. 87-92.

[11] Haddow, Pauline C. and Tyrrell, Andy M., "Challenges of evolvable hardware: past, present and the path to a promising future," *Genetic Programming and Evolvable Machines*, vol 12, no 3, 2011. pp 183-215.

[12] Ruben Ramirez-Padron, Feras Batarseh, Kyle Heyne, Annie S. Wu, and Avelino Gonzalez, "On the performance of fitness uniform selection for non-deceptive problems," *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10).* ACM, New York, NY, USA, Article 32, 6 pages.

[13] Tobias Storch, Ingo Wegener, *Real royal road functions for constant population size,* Theoretical Computer Science, Volume 320, Issue 1, 12 June 2004, Pages 123-134, ISSN 0304-3975

[14] S. Sethumadhavan, R. Roberts, Y. Tsividis, "A Case for Hybrid Discrete-Continuous Architectures," *Computer Architecture Letters*, vol. 11, no.1, pp. 1-4, Jan.-June 2012.

[15] Theodore W. Cornforth and Hod Lipson. *Reverse-Engineering Nonlinear Analog Circuits with Evolutionary Computation.* Unconventional Computation and Natural Computation. Springer International Publishing, 2014. pp. 105-116.

[16] John R. Koza, F.H. Bennett, III, D. Andre, Martin A. Keane, and F.Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming," *IEEE Transactions on Evolutionary Computation*, vol.1, no.2, pp.109-128, Jul 1997.

[17] Yanfeng Jiang, Jiaxin Ju, Xiaobo Zhang, Bing Yang, "Automated analog circuit design using Genetic Algorithms," *ASID 2009. 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication*, pp. 223 - 228, 20-22 Aug. 2009.

[18] Yerbol A. Sapargaliyev and Tatiana G. Kalganova. "Open-ended evolution to discover analogue circuits for beyond conventional applications," *Genetic Programming and Evolvable Machines*, 13.4 (2012): pp. 411-443.

[19] Sangwook Suh, Arindam Basu, Craig Schlottmann, Paul E. Hasler, John R. Barry, "Low-Power Discrete Fourier Transform for OFDM: A Programmable Analog Approach," , *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 2, pp. 290-298, Feb. 2011.

[20] A. Senn, A. Peter, J. G. Korvink, "Analog circuit synthesis using two-port theory and genetic programming," *AFRICON, 2011*, pp.1-8, 13-15 Sept. 2011

[21] Glenn ER Cowan, Robert C. Melville, and Y. Tsividis. "A VLSI analog computer/digital computer accelerator," *Solid-State Circuits, IEEE Journal of*, 41.1 (2006): 42-53.

[22] Ismai, F.S.; Yusof, R., "A new self organizing multi-objective optimization method," *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on* , vol., no., pp.1016,1021, 10-13 Oct. 2010

[23] Russ C. Eberhart and James Kennedy, "A new optimizer using particle swarm theory," *Proceedings of the sixth international symposium on micro machine and human science*. Vol. 1. 1995.

[24] Samrat L. Sabat, K. Shravan Kumar, and Siba K. Udgata. "Differential evolution and swarm intelligence techniques for analog circuit synthesis," *NaBIC 2009 World Congress on Nature & Biologically Inspired Computing, IEEE*, 2009.

[25] Forrest H. Bennett III, et al. "Evolution by Means of Genetic Programming of Analog Circuits that Perform Digital Functions," *GECCO*. 1999.

[26] William Mydlowec and John Koza, "Use of time-domain simulations in automatic synthesis of computational circuits using genetic programming," *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, Las Vegas, Nevada*, 2000.

[27] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn, "The island model genetic algorithm: On separability, population size and convergence," *Journal of Computing and Information Technology* 7 (1999): 33-48.

[28] Monica Sam, Sanjay K Boddhu, Kayleigh E. Duncan, John C.Gallagher, "Evolutionary strategy approach for improved in-flight control learning in a simulated Insect-Scale Flapping-Wing Micro Air Vehicle," *Evolvable Systems (ICES), 2014 IEEE International Conference on* , pp.211,218, 9-12 Dec. 2014

[29] Matthew J. Streeter, Martin A. Keane, and John R. Koza. "Iterative Refinement Of Computational Circuits Using Genetic Programming," *GECCO*, 2002.

[30] D. Keymeulen, R. Zebulum, Y. Jin, A. Stoica,, "Fault-tolerant evolvable hardware using field-programmable transistor arrays," *IEEE Transactions on Reliability*, vol. 49, no. 3, pp. 305-316, Sept. 2000.

[31] *PsoC 5LP Registers TRM*, Cypress Semiconductor, San Jose, CA, 2014, pp. 299.

[32] Shin Ando, Mitsuru Ishizuka, and Hitoshi Iba. 2003. *Evolving analog circuits by variable length chromosomes*. In Advances in evolutionary computing, Ashish Ghosh and Shigeyoshi Tsutsui (Eds.). Springer-Verlag New York, Inc., New York, NY, USA 643-662.

[33] S. Kazadi, Y. Qi, I. Park, N. Huang, P. Hwu, B. Kwan, W. Lue, and H. Li, "Insufficiency of piecewise evolution," *in 3rd NASA/DoD Workshop on Evolvable Hardware*, D. Keymeulen et al. (eds.), IEEE Computer Society Press: Los Alamitos, CA, 2001, pp. 223–231.

[34] Miller, B. L., & Goldberg, D. E. (1995). *Genetic algorithms, tournament selection, and the varying effects of noise* (IlliGAL Report No. 95007). Urbana: University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory

[35] Steven D. Pyle, Vignesh Thangavel, Stephen M. Williams, and Ronald F. DeMara, "Self-Scaling Evolution of Analog Computation Circuits with Digital Accuracy Refinement" submitted to *2015 NASA/ESA Conference on Adaptive Hardware and Systems*.

[36] *AN84810 - PSoC® 3 and PSoC 5LP Advanced DMA Topics*. Cypress Semiconductor, San Jose, CA, 2014, pp. 299