

DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

**HARDWARE BASED SIMULATOR FOR
COUPLED ARRAYS OF PENDULUMS**

by
Gökhan BAŞ

February, 2017

İZMİR

HARDWARE BASED SIMULATOR FOR COUPLED ARRAYS OF PENDULUMS

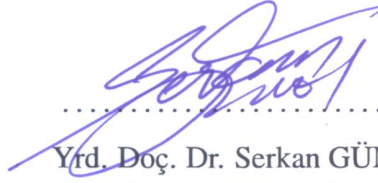
**A Thesis Submitted to the
Graduate School of Natural And Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of
Science in Electrical Electronics Engineering**

**by
Gökhan BAŞ**

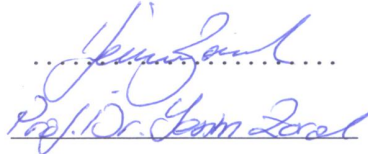
**February, 2017
İZMİR**

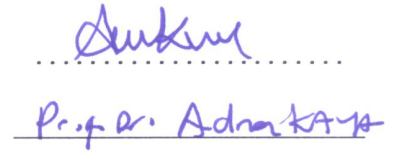
M.Sc THESIS EXAMINATION RESULT FORM


We have read the thesis entitled "**HARDWARE BASED SIMULATOR FOR COUPLED ARRAYS OF PENDULUMS**" completed by **GÖKHAN BAŞ** under supervision of **YRD. DOÇ. DR. SERKAN GÜNEL** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


.....
Yrd. Doç. Dr. Serkan GÜNEL

Supervisor


.....
Prof. Dr. Emin Zoral
Jury Member


.....
Prof. Dr. Adnan Kaya
Jury Member


.....
Prof. Dr. Emine İlknur CÖCEN
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would like to devote this work to my father Halil İbrahim, my mother Türkan and my wife Seher.

Gökhan BAŞ



HARDWARE BASED SIMULATOR FOR COUPLED ARRAYS OF PENDULUMS

ABSTRACT

In this thesis, we have investigated coupled identical dynamical systems embedded in a field programmable gate array in order to accelerate solution of associated ordinary differential equation. The numerical solutions of ODE requires iterations that form an algebraic loop. This study shows that the parallel computing ability of field programmable gate array becomes economical after a certain number of coupled nodes.

Keywords: Network, Physical systems, Coupling, Ordinary differential equation on field programmable gate array.

KUPLAJLI SARKAÇ DİZİSİNİN DONANIM TABANLI BENZETİMİ

ÖZ

Bu tezde birden çok özdeş dinamik sistemin sahada programlanabilir kapı dizisi üzerinde gerçekleştirilmesi çalışıldı. Adi türevli denklemlerin çözümleri sahada programlanabilir kapı dizisi ile hızlandırılabilir. Adi türevli denklemlerin sayısal çözümleri cebirsel döngü içeren iterasyonlar gerektirir. Bu çalışma aynı sistemin kişisel bilgisayarlara nazaran kaç düğümden sonra FPGA çözümünün daha hızlı olacağını da öngörmektedir.

Anahtar kelimeler: Fiziksel sistem ağları, Kuplaj, Alanda programlanabilir kapı dizisi üzerinde adi türevli denklem.

CONTENTS

	Page
THESIS EXAMINATION RESULT FORM.....	ii
ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZ.....	v
LIST OF FIGURES	viii
LIST OF TABLES.....	ix
 CHAPTER ONE – INTRODUCTION.....	 1
1.1 Introduction to Networks	2
1.2 Field Programmable Gate Arrays.....	4
1.2.1 Special DSP48A1 slices for Xilinx Spartan 6	4
1.2.2 Hardware Description Languages (HDLs)	5
1.2.3 Selection of Arithmetic Coding.....	6
 CHAPTER TWO – FPGA IMPLEMENTATION OF FUNCTIONS.....	 8
2.1 Maintaining a Design Server	8
2.2 Calculation of Sinusoidal Functions in FPGA.....	8
2.3 The CORDIC Algorithm.....	9
2.4 The CORDIC Sine and Cosine	12
2.4.1 Four Quadrant Converging Design	12
2.4.2 Design Summary	14
2.4.3 Design Criteria	15
 CHAPTER THREE – ON CHIP DYNAMICAL SYSTEMS SOLVING ALGEBRAIC LOOPS.....	 17
3.1 Initialization of The ODE States	18
3.2 A Random Number Generator for FPGA	18

3.3 Case 1: Analytic Solution is Available	20
3.4 Case 2: Analytic Solution is not Available	22
3.5 Considering Multiple Linear Dynamical Systems	25
3.6 Multiple Dynamical Systems with a Nonlinearity	28
3.6.1 Implementation of Difference Equations with Trigonometric Functions	29
3.7 FPGA Implementation of Star Network	30
3.7.1 Comparison with Sequential Programming	30
CHAPTER FOUR – CONCLUSION.....	36
REFERENCES.....	37
APPENDICES	40
A.1 $F = 31$ Bit $N = 20$ Iteration CORDIC Block With Algebraic Loop	40
A.2 $F = 31$ Bit $N = 20$ Iteration CORDIC Unrolled Design.....	44
A.3 $F = 31$ Bit $N = 20$ Iteration CORDIC Test Bench.....	47
A.4 $I + F = 24$ Bit Random Number Generator Module	48
A.5 $I + F = 24$ Bit Absolute Value Calculator.....	50
A.6 $I + F = 24$ Bit Single Pendulum Module	51
A.7 $I + F = 24$ Bit Six Node Coupled Pendulum Solver Module.....	54
A.8 MATLAB™ Code For Comparison	59
A.9 C Code For Comparison IEEE754-32	62
A.10 C Code For Comparison $I = 2, F = 29$ Signed Fixed Point.....	67

LIST OF FIGURES

	Page
Figure 1.1 Five different network examples	2
Figure 1.2 FPGA structure.....	3
Figure 1.3 Logic cell structure.	3
Figure 2.1 The rotation.....	10
Figure 2.2 Unrolled sine cosine CORDIC processor.....	11
Figure 2.3 Sequential CORDIC rotations amounts $\arctan 2^{-i}$	12
Figure 2.4 Output waveforms and error plots.	16
Figure 3.1 The algebraic loop of the ODE.....	17
Figure 3.2 Unrolled $x_{i+1} = F_{\Delta t}(x_i)$ structure.....	17
Figure 3.3 32-bit LFSR schematic.	19
Figure 3.4 Output waveform of the 32-bit LSFR random number generator.	19
Figure 3.5 ODE with analytic solution and its implementation	20
Figure 3.6 Unrolled feed forward circuitry to compute $1 - e^{\alpha}$	21
Figure 3.7 Iterative circuit of the dynamical system of Figure (Figure 3.5).	23
Figure 3.8 Unrolled iterative circuit of the dynamical system of Figure (Figure 3.5).....	23
Figure 3.9 Iterative circuit of the dynamical system of Figure (Figure 3.5).	24
Figure 3.10 Two independent RC model implemented in parallel	25
Figure 3.11 Two coupled RC model implemented in parallel (I).....	26
Figure 3.12 Two coupled RC model implemented in parallel (II)	28
Figure 3.13 Two node synchronization.	32
Figure 3.14 Four node star and six node star synchronization.....	33

LIST OF TABLES

	Page
Table 1.1 32 bit IEEE754 standard coding examples	6
Table 2.1 Unrolled design vs. algebraic loop design using DSP48A1 or not	14
Table 2.2 Unrolled design vs. algebraic loop design using pipelining or not	15
Table 3.1 Comparison of design summaries.....	31
Table 3.2 Comparison of design summaries 16 bit	34
Table 3.3 Mean time needed for one iteration to be completed on PC.....	34



CHAPTER ONE

INTRODUCTION

Use of Field Programmable Gate Arrays (FPGA) in solving computation problems are encountered in many different fields of the science and engineering where fast computations are required. In those cases, specific and dedicated hardware is developed to achieve a faster solution. During the solution of a computational problem, instead of using a general purpose microprocessor with a central processing unit (CPU), FPGA implementation of the problem introduces parallel computing improvements in the target arithmetic. For example, implemented finite impulse response image filtering on an FPGA circuitry works much faster compared to the one implemented on a CPU that works sequential than parallel (Nelson, 2000).

In this thesis, we use FPGA to implement a dedicated Ordinary Differential Equation (ODE) system solver for linearly interconnected network of rigid physical models.

When compared to the analog computing solutions, FPGA implementation requires discretization of the problem, if the problem is continuous by nature. This situation has both advantages and disadvantages. The first well-known advantage is that discrete systems are much more robust to noise that is compared to analog systems. Secondly, the used elements in an analog computer have tolerances which are another source of error. On the other hand, the disadvantage is that discretized systems have quantization error. Specific to the ODE problems, the quantization error is multiplied in the difference equation of the ODE, where calculation of evolution of system states take place (Mathews, 1992). Sometimes this error can undergo a rather spread mapping of the difference equation and the obtained solution becomes intolerably erroneous compared to the exact solution, though the difference equation is Lipschitz continuous. When the solution of the ODE is analytic, the solution is exact and the difference equation is used once to obtain the final state of the system and the solution can be converted to a fully feed-forward circuit as a desired situation. For most of the ODE systems, an analytic solution does not exist in general and the solution comes in the form of iterations of the difference equation.

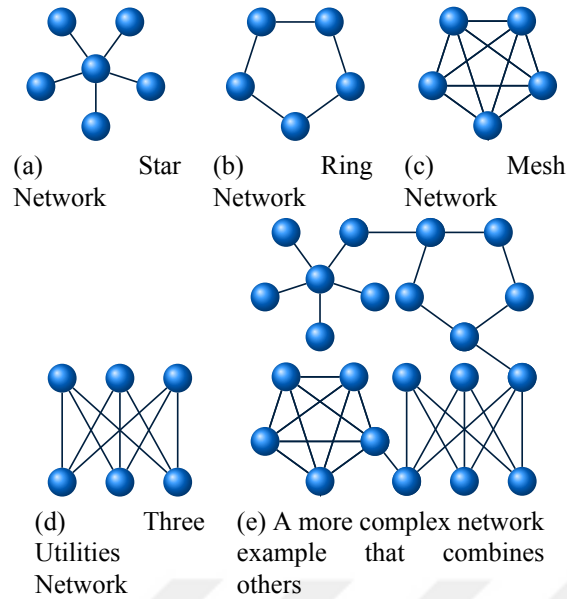


Figure 1.1 Five different network examples

1.1 Introduction to Networks

A network is a structure that consists of interconnected elements, where the nodes represent the elements of the system and edges represent the interactions between them (Caldarelli, 2012). The world wide web and its own dynamics can also be represented as interconnected computer nodes that leads to a network. The biological systems, for example a community of ants, can be described as network that is interconnected by means of their internal communication by pheromone (Dorigo, 1992). Furthermore, in living cells molecules radiate communication signals to each to sustain tissular life (Ernst, 2004). All can be modeled mathematically and in the end these examples leads to network of nodes and vertices. Usually, networks are presented by graphs. Some network graph examples are given in Figure (Figure 1.1). This study aims to design systems as an analogy to such networked systems by means of Field Programmable Gate Arrays (FPGAs).

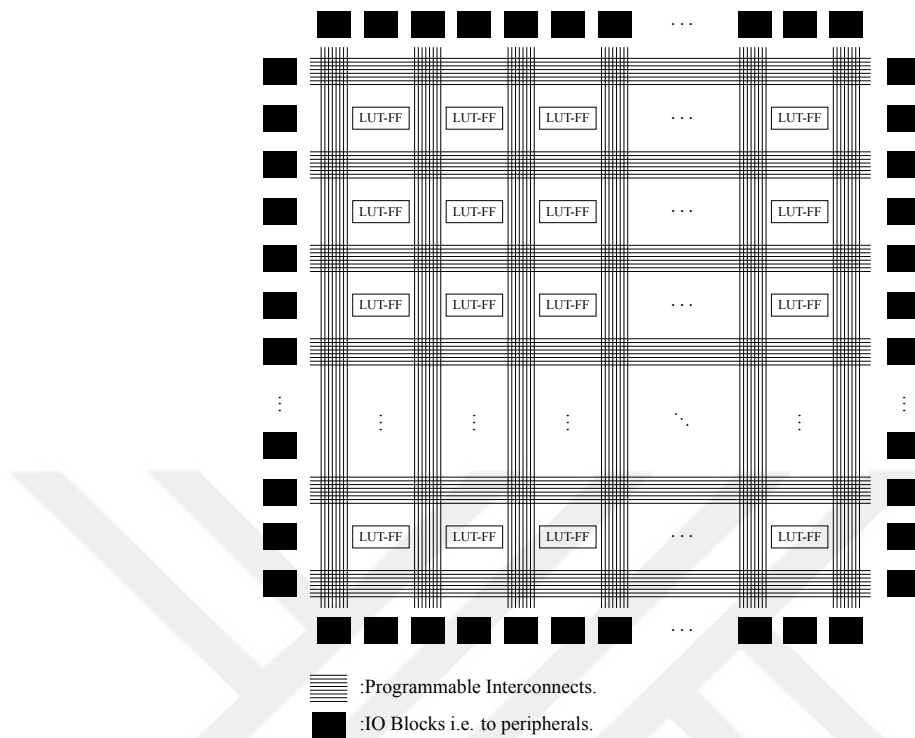


Figure 1.2 FPGA structure: LUT-FF pairs stands for logic cells that is consisted of Lookup Table and Flip Flop pairs. The inside of a LUT-FF is as shown in Figure (Figure 1.3).

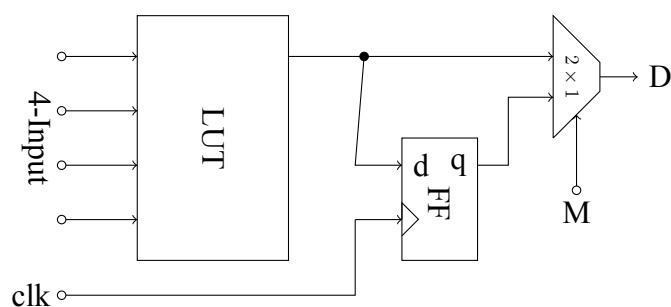


Figure 1.3 Logic cell structure, 4-Input, clk, M and D can be routed to other LUT-FF by programmable interconnects shown in Figure (Figure 1.2), The inside of the LUT and FF, the initial values are determined after power on reset this is also programmable. Furthermore M determines whether this LUT-FF pair is used as a combinational LUT circuit or a combinational feed FF circuit.

1.2 Field Programmable Gate Arrays

FPGAs are semiconductor devices which are an array of configurable logic cells connected via programmable interconnects. The logic cells form the main structure of FPGAs. A logic cell consists of one Look-up Table (LUT), one D-flip flop and one 2×1 multiplexer. Figure (Figure 1.3) shows a logic cell with four input LUT. LUTs are actually small random access memories that fulfill logic operations. By the combination of thousands of logic cells, complex and large functions can be implemented.

The programming interface used to create logic functions is called Hardware Description Language (HDL). FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks.

There are several companies¹ with many different configuration boards² commercially available with different input output (IO) units on-board. Character LCD, Analog digital converter, digital analog converter, LED lines, switch lines, buttons, video graphics array output jack, AC97 controller, general purpose IO etc, are some of available optional peripherals.

1.2.1 Special DSP48A1 slices for Xilinx Spartan 6

The DSP48A1 slices are special slices that support many independent functions, including multiplier, multiplier-accumulator (MACC), pre-adder/subtractor followed by a MACC, multiplier followed by an adder. The architecture also supports connecting multiple DSP48A1 slices to form wide math functions, DSP filters, and complex arithmetic without the use of general FPGA logic (Xilinx™, 2014).

¹Altera and Xilinx

²Virtex, Spartan and Stratix series etc.

1.2.2 Hardware Description Languages (HDLs)

Digital circuit design is advancing rapidly in recent few decades. At the very beginning of this phenomenon the first digital circuits was consisting of vacuum tubes. After the invention of transistor in 1947 those circuits become in solid state transistors. As the years past, integrated circuits (ICs) are invented and much more complex digital circuits are produced and those ranging from small scale integration (SSI), medium scale integration (MSI), large scale integration (LSI), very large scale integration (VLSI). According to the Moore's Law, the amount of integration is doubled every two years. During this improvement computer technology is also improved and Computer Aided Design (CAD) has become a key point of IC design. With the aid of computer designers started to design at the gate level designs which is one above of the transistor level. CAD is also used in placing circuit templates and routing of the signals in the design. At the gate level design, use of different CAD commands of different designers make those commands become standardized.

At last but not least, HDLs provides many beneficial features to the designer,

1. Designs depending on HDLs made abstracted and they do not have to be transferred on to an FPGA or IC production flow,
2. Designs that are done with HDLs can be easily simulated before production and most of the design faults at the gate level can easily be detected and fixed immediately before production.
3. Many manufacturer builds HDL libraries and cores that makes designers' job easier.

Verilog is an implementation of HDL which has a syntax much alike to C programming language. In this thesis, I preferred Verilog.

Table 1.1 32 bit IEEE754 standard coding examples.

N	$S \underbrace{e_7 e_6 e_5 e_4 e_3 e_2 e_1 e_0}_E \underbrace{m_{22} m_{21} \dots m_0}_M$
0.000000	00000000000000000000000000000000
1.000000	00011111110000000000000000000000
1.500000	00011111111000000000000000000000
-2.000000	10100000000000000000000000000000
-3.000000	10100000001000000000000000000000
4.000000	00100000010000000000000000000000
6.666666	00100000011010101010101010101010
8.000000	00100000100000000000000000000000
10.000000	00100000100100000000000000000000
11.000000	00100000100110000000000000000000

1.2.3 Selection of Arithmetic Coding IEEE754 or Signed 2's Complement

In digital systems where arithmetic operations are carried out, there should be a way of holding the information. This is commonly done either by coding the values into a binary signed 2's complement system or using IEEE754 floating point standard. There are 16 bit (half-precision), 32 bit (single precision), 64 bit (double precision), 128 bit (quadruple precision), and 256 bit (octuple precision) implementation of this standard.

In our design, we selected to use signed 2's complement coding. Since Xilinx tools do not provide a debugging interface for IEEE754 numbers, we would have to build an interpreter or transfer the numbers into the computer to interpret them. Additionally, the resultant arithmetic circuitry of ODE solver will be too large compared to the signed 2's complement one. Furthermore signed 2's complement coding is also primarily supported in both Verilog and VHDL as well as other HDLs at the software level. Besides that Xilinx ChipScope™ is able to deal with signed 2's complement numbers and plot them.

The IEEE754-32 standard it is widely used for coding in computer systems where the most significant bit of the number is sign bit S the trailing 8 bits denotes exponent E and the remaining 23 bits stands for mantissa M and the number N is coded as follows;

$$N = (-1)^S \times 2^{E-127} \times 1.M \quad (1.1)$$

Fixed point numbers include an integer part I and a fraction part F in addition to the sign bit S , e.g. signed $I = 3$, $F = 28$ fixed point numbers are defined as follows:

$$N = S \underbrace{b_2 b_1 b_0}_{I=3} \underbrace{b_{-1} \cdots b_{-28}}_{F=28}, \quad (1.2)$$

where the most significant S is the sign bit, $b_2 b_1 b_0$ is the integer part I and trailing $b_{-1} b_{-2} \cdots b_{-28}$ is the fraction part F , i.e.

$$N = (-1)^S \sum_{i=-28}^2 b_i 2^i. \quad (1.3)$$

In this thesis, coupled identical dynamical systems are implemented on an FPGA kit with different data encoding and different number of nodes in the system. The measurements are done with ChipScope™. ChipScope™ is a programmable measuring environment that picks measurements from the debugged circuit on FPGA to the computer side via the download cable. The second investigation is whether we accelerate the solution of ordinary differential equation on FPGA although numerical solutions with iterations that are in the form of algebraic loops. Third investigation is that after how many nodes we can compete with a personal computer solution of the same problem. We will also show that the parallel computing ability of FPGA becomes economical after a certain number of nodes when compared to sequential programming.

CHAPTER TWO

FPGA IMPLEMENTATION OF FUNCTIONS

2.1 Maintaining a Design Server for both Local and Remote Access

The FPGA development kits are electronic boards that can be easily get damaged during a transportation or set up and remove operation due to static electric. In order to use the board from a remote location, we set up a design server that has a public Internet Protocol (IP) address. Then we installed Xilinx tools as well as a Secure Shell (SSH) server software and a Virtual Network Computation (VNC) server software on the server. The result is remotely accessible design and debug environment. However if the programmed circuitry uses the peripherals such as buttons or switches you can not use them remotely, nevertheless there is a design core of Xilinx called ChipScope™Virtual Input Output (VIO) that emulates both switches and buttons during running the programmed circuitry. Besides from gathering information from the board via computer ChipScope™has this VIO core emulates input and output from the computer side. Since we can not build a ODE solver circuit from a totally isolated from environment there must be a signal that initializes states of the ODE system, this operation is done by using VIO core.

2.2 Calculation of Sinusoidal Functions in FPGA Environment

In designing ODE solvers in FPGA environment, we have to think about advanced calculations of functions due to the fact that many nonlinear continuous or discrete systems, especially physical models consists of them. The trigonometric functions i.e. $f(x) = \sin x$ or $f(x) = \cos x$ are widely used ones.

A digital implementation of sine and cosine functions can be in the form of a look up table. This is the fastest available method, however this implementation uses too much resources when compared to other methods and $(-\pi, \pi]$ domain has to be

sampled/discretized in addition to using discrete numbers which is not desired in ODE solutions.

The Coordinate Rotational Digital Computation (CORDIC) algorithm does not need such a discretization (Arbaugh, 2004). It iteratively calculates the sine and cosine value of the input angle with multiply and add (MAD) operations.

We both tested those iterations in the form of a feed forward unrolled design and the one that forms an algebraic loop. The algebraic loop structure that uses less resources than unrolled method where the internal states are to be hold in update registers, which results in set up and hold times and need of a clock that slows the design.

In the unrolled design, we tested the algorithm with fixed point data encoding with 31 bits denoting the fractional part for 20 iterations. The combinational path length is proportional with carried out number of iterations. The combinational path works feeding the angle θ in radians and the result appears at the output stage as $\cos \theta$ and $\sin \theta$ values. The one that contains an algebraic loop differs on the point of internal states $\cos \theta_i$ and $\sin \theta_i$ are kept in update registers which are updated sequentially without conveying the middle steps of the iterations to a next stage of a new layer of forwarding hardware; which in turn helps to save the resources indeed. At the same time throughput is reduced due to repeating set up and hold times of the registers in the loop.

2.3 The CORDIC Algorithm

Consider the rotation of a point in two dimensional coordinate system as shown in Figure (Figure 2.1), Equation (2.1) denotes the rotation of the point $[x_{n-1}, y_{n-1}]^T$ to $[x_n, y_n]^T$ with the one-to-one onto linear transformation matrix as shown in Figure (Figure 2.1).

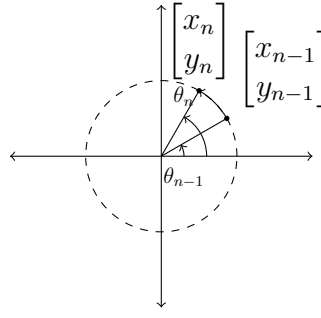


Figure 2.1 The rotation

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix} \quad (2.1)$$

The rotation matrix in Equation (2.1) can be re-written as in the following form by making use of the trigonometric identity $\sin \theta = \frac{1}{\sqrt{1+\tan^2 \theta}}$ and $\cos \theta = \frac{\tan \theta}{\sqrt{1+\tan^2 \theta}}$ which substituted in Equation (2.1) to yield:

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \frac{1}{\sqrt{1+\tan^2 \theta}} \begin{bmatrix} 1 & -\tan \theta \\ \tan \theta & 1 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix}. \quad (2.2)$$

If the rotations going to be carried out are considered using the transformation $\tan \theta_n = 2^{-n}$ for each rotation, the following equation set can be written.

$$\begin{bmatrix} x_n \\ y_n \\ \theta_n - \theta_{n-1} \end{bmatrix} = \frac{1}{\sqrt{1+2^{-2n}}} \begin{bmatrix} 1 & -\sigma_{n-1}2^{-n} & 0 \\ \sigma_{n-1}2^{-n} & 1 & 0 \\ 0 & 0 & \sqrt{1+2^{-2n}}\sigma_{n-1} \end{bmatrix} \begin{bmatrix} x_{n-1} \\ y_{n-1} \\ \arctan 2^{-n} \end{bmatrix} \quad (2.3)$$

$$\begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \sigma_n = \begin{cases} 1, & \theta_n < \theta \\ -1, & \theta_n > \theta \\ 0, & \text{otherwise} \end{cases} \quad (2.4)$$

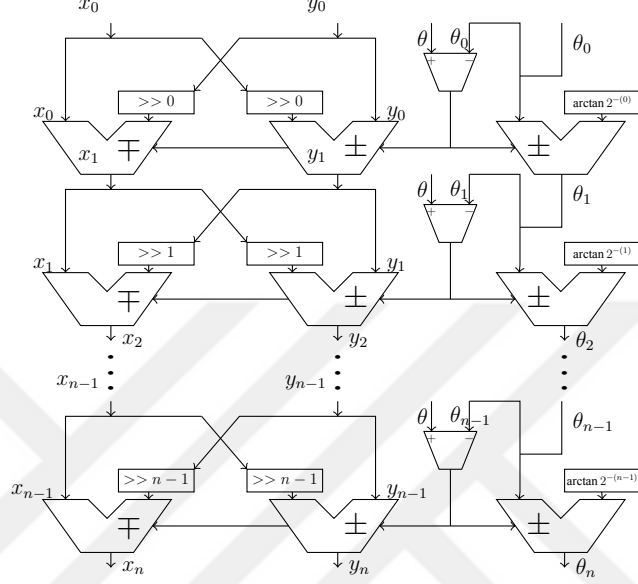


Figure 2.2 N iteration unrolled sine cosine CORDIC processor, at the last layer a K_N scaling is needed to make sure that $x_n = \cos \theta_n$, and $y_n = \sin \theta_n$.

Equation (2.3) and (2.4) gives the whole algorithm. Beginning from the initial condition $[x_0, y_0, \theta_0]^T = [1, 0, 0]$, each θ_n value is compared to the input angle θ to predict the right direction for the next turn with a smaller angle $\arctan 2^{-n}$. Since there exist no calculator for an arc-tangent function, $\arctan 2^{-n}$ is simply embedded in a logical table. The resultant angle θ_n is within the neighborhood of the input angle θ with a maximum error of $\pm \arctan 2^{-N}$ for N iterations. However, the scaling factor $K_N = \prod_{n=0}^N \frac{1}{\sqrt{1+2^{-2n}}}$ has to be taken into account to obtain a correct answer. As in the case of the $\arctan 2^{-n}$ numbers, that factor can be easily called from a look up table circuitry as well. The algorithm is called as a shift-add (SAD) algorithm since calling the next iteration requires 2^{-n} scaling which is continued with an addition of x_{n-1} and y_{n-1} .

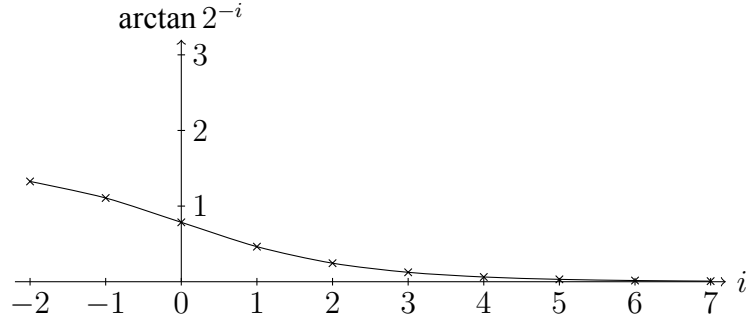


Figure 2.3 Sequential CORDIC rotations amounts $\arctan 2^{-i}$

2.4 The CORDIC Sine and Cosine: Implementation

The space-time trade off always stands for an implementation issue for digital systems especially that contain algorithms including iterations. The problem here is how to implement this digital iteration sequence with the fastest way or in a minimum number of resources or both. In this thesis, timing constraints are assumed to be critical, hence the implementation is optimized accordingly.

In the Figure (Figure 2.2), the implementation of the Equations (2.3) is seen with constant shifters and cascaded adders. At the output stage, it is necessary to scale the obtained x_n and y_n with K_N . The implementation considered here employs a $F = 31$ bit fractional fixed point numbers as internal states θ_n , x_n and y_n .

2.4.1 Four Quadrant Converging Design

The required outcome is to compute $\sin \theta$ and $\cos \theta$. However the $\arctan 2^{-i}$ angle is $\pi/4$ radians for $i = 0$. Which means an infinite number of diminishing addition for state θ_n even in the same direction i.e. $\sigma_n = 1 \forall n$ or $\sigma_n = -1 \forall n$ yields the following

upper and lower bounds;

$$\theta_n = \lim_{N \rightarrow \infty} \sum_{i=0}^N \sigma_n \arctan 2^{-i} \quad (2.5)$$

$$\max \theta_n = \sum_{i=0}^{\infty} \arctan 2^{-i} = 1.7433\text{rad}, \quad (2.6)$$

$$\min \theta_n = - \sum_{i=0}^{\infty} \arctan 2^{-i} = -1.7433\text{rad} \quad (2.7)$$

which means that this type of design only calculates the input angle spanning $\pm 1.7433\text{rad}$ therefore total $(-\pi, \pi]$ range is not covered. To deal with this problem the initial rotation angle θ_1 should be selected as $\sigma_0 \arctan 2^2$ to cover the desired input range, i.e.

$$\max \theta_n = \sum_{i=-2}^{\infty} \arctan 2^{-i} = 4.1763\text{rad} \quad (2.8)$$

$$\min \theta_n = - \sum_{i=-2}^{\infty} \arctan 2^{-i} = -4.1763\text{rad} \quad (2.9)$$

Selection of starting angle by two iterations earlier causes a wide range as shown in Equation (2.8), which means that total rotation range is increased by an unnecessary amount of

$$\underbrace{\pm 4.1763\text{rad}}_{\text{available}} - \underbrace{\pm \pi\text{rad}}_{\text{required}} = \underbrace{\pm 1.0347\text{rad}}_{\text{redundant}},$$

though number of iterations are increased. This requires more resources and time. To meet with complete coverage with minimizing requirements needs changing the initial conditions. This can be compensated by selecting initial angle $\theta_0 = \pm \pi/2$ therefore the resultant coverage will be $\pm 1.5708\text{rad} + \pm 1.7433\text{rad} = \pm 3.3141\text{rad}$ which is slightly above $\pm \pi\text{rad}$. The initial condition of Equation (2.4) becomes;

$$[x_0, y_0, \theta_0]^T = \begin{cases} [0, 1, \frac{\pi}{2}]^T, & \theta > 0 \\ [0, -1, -\frac{\pi}{2}]^T, & \theta < 0 \end{cases} \quad (2.10)$$

Therefore no additional iterations are required.

Table 2.1 Design summaries for unrolled design vs. algebraic loop design with using DSP48A1 or not.

Resource summary for algebraic loop design:	
Number of Slice LUTS	1180/27288 (%4)
Number of DSP48A1	8/58 (%13)
Timing summary for algebraic loop design:	
Minimum Period	12.44ns
Maximum combinational path delay	No path found
Resource summary for algebraic loop design (no DSP48A1):	
Number of Slice Registers	160/54576 (%0)
Number of Slice LUTS	2150/27288 (%7)
Number of DSP48A1 slices	-
Timing summary for algebraic loop design (no DSP48A1):	
Minimum Period	12.235ns
Maximum combinational path delay	No path found
Resource summary for unrolled design:	
Number of Slice LUTS	2444/27288 (%8)
Number of DSP48A1 slices	12/58 (%20)
Timing Summary for unrolled design:	
Minimum Period:	No path Found
Maximum combinational path delay	106.58ns
Resource summary for unrolled design (no DSP48A1):	
Number of Slice LUTS	2950/27288 (%14)
Number of DSP48A1 slices	-
Timing Summary for unrolled design: (no DSP48A1)	
Minimum Period:	No path Found
Maximum combinational path delay	97.437ns

2.4.2 Design Summary

Two different case implementations, namely iterative and unrolled case is depicted in Table (Table 2.1) and (Table 2.2) for Spartan™6 xc6slx45-3csg484 chip.

Here we can, easily measure how fast the unrolled design is; $20 \times 12.44 = 248.8\text{ns}$ is required time for the algebraic loop for 20 iterations and this amount is about 2.5 times slower than unrolled timing which is 106.58ns. With pipelining this measure is approximately same with unrolled case i.e. $20 \times 5.732 = 114.64 \approx 106.58\text{ns}$. The fastest available solution for $N = 20$ iterations is using the IP core without pipelining. Pipelining increases the rate of throughput with high clocking rates of the circuit in the return of a little bit slowing down when compared the no pipelining situation of

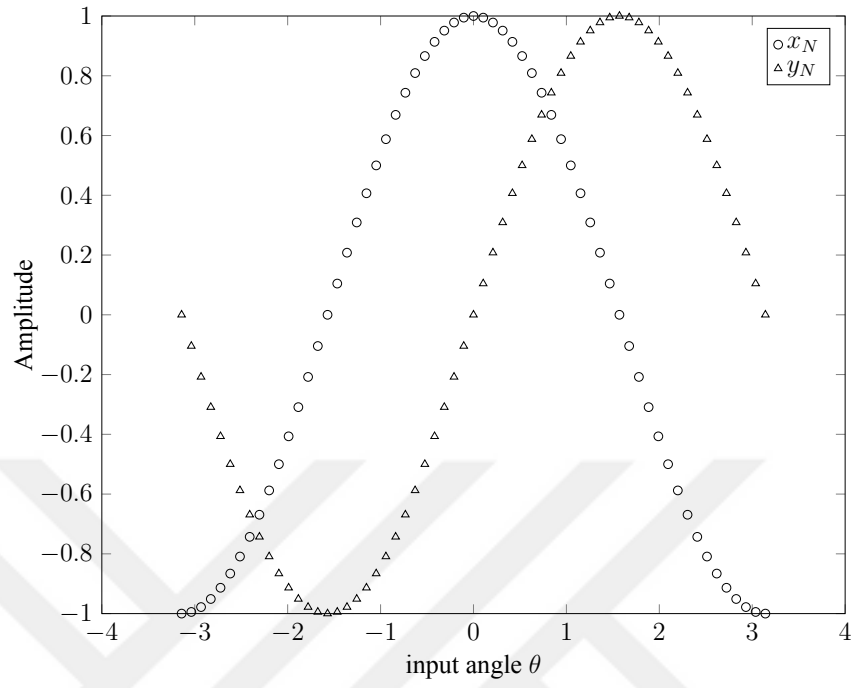
Table 2.2 Design summaries for unrolled design vs. algebraic loop design with using register pipelining or not.

Resource summary for unrolled CORDIC IP: (with pipelining)	
Number of Slice Registers	2860/54576(%5)
Number of Slice LUTS	2978/27288 (%8)
Number of DSP48A1 slices	-
Timing summary for unrolled CORDIC IP (with pipelining):	
Minimum Period	5.732ns
Maximum combinational path delay	No path found
Resource summary for unrolled CORDIC IP: (without pipelining)	
Number of Slice Registers	112/54576(%0)
Number of Slice LUTS	2975/27288 (%8)
Number of DSP48A1 slices	-
Timing summary for unrolled CORDIC IP (without pipelining):	
Minimum Period	81.342ns
Maximum combinational path delay	No path found

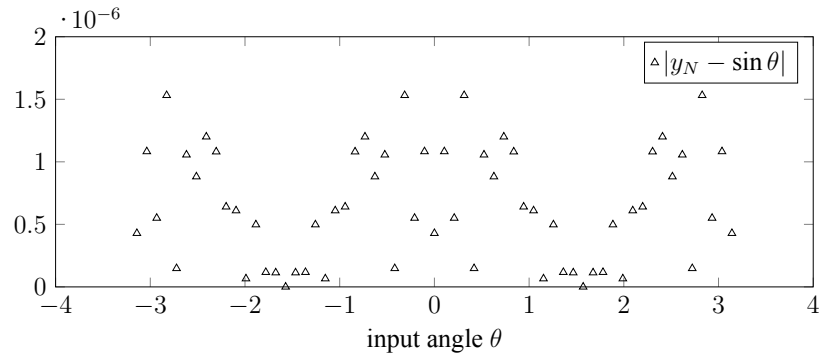
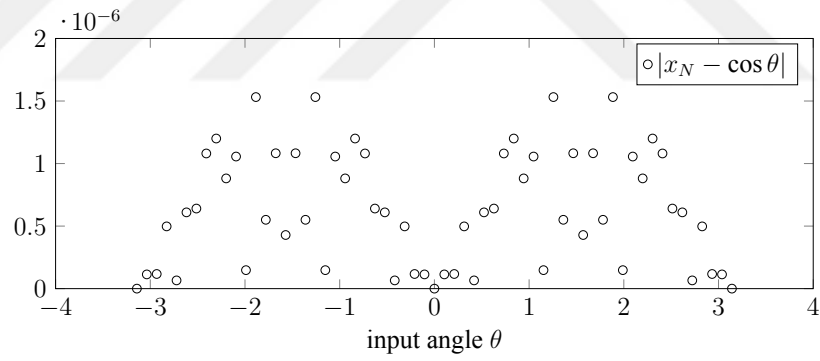
the same implementation. Maximum clocking rate is for pipelined architecture is $1/5.732\text{ns} \approx 174\text{MHz}$

2.4.3 Design Criteria: Calculation of The Bound of Error for N Iterations.

In order assess the accuracy of the implementations, we checked the error bounds. Amount of error is based absolutely on the minimum angle of turn with N iterations. Which is simply $|\arctan 2^{-19}| < |2 \times 10^{-6}|$ in our case, that explains the error margin in plot in Figure (Figure 2.4b).



(a)



(b)

Figure 2.4 Output waveforms and error plots.

CHAPTER THREE

ON CHIP DYNAMICAL SYSTEMS SOLVING ALGEBRAIC LOOPS

Every differential equation does not have an analytic solution. However the solutions can be approximated by iterations. Every iteration that contains feedback is called as an algebraic loop. Technically, the numerical solution of a differential equation that is in the form

$$\dot{x} = f(x, t)$$

is based on an algebraic loop that is shown in Figure (Figure 3.1). The main question arises from this point of view is: "Can we avoid this loop to obtain a parallel in parallel out expression?" as shown in Figure (Figure 3.2). There are two ways to do so and they have limitations that is a consequence cost of synthesized parallelism.

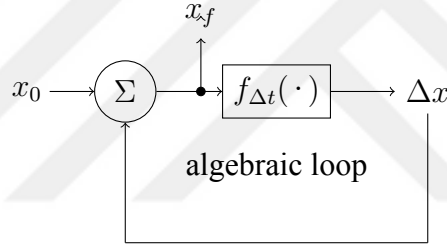


Figure 3.1 Iterative structure of approximate solution covered with algebraic manipulations that is called an algebraic loop of the ODE.

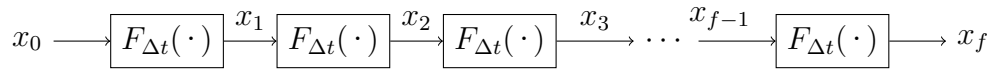


Figure 3.2 Unrolled $x_{i+1} = F_{\Delta t}(x_i)$ structure.

1. If there is an analytic solution to $\dot{x} = f(x, t)$ the whole algebraic loop becomes a single algebraic equation that can be realized as a single high resolution look up table or MAD operations. For example an e^x in the solution can be obtained by adding the Maclaurin series expansion terms up to a desired resolution and those terms are only a combination of MAD operations. In most of the time seemingly complex algebraic terms are reduced to MAD operations.
2. $\dot{x} = f(x, t)$, $t \in [t_0, t_f]$, the discretization of time domain yields finite number of iterations for a finite Δt iterations. That can be formulated as an iterative

function $F_{\Delta t}(\cdot)$ and that is,

$$x_{i+1} = F_{\Delta t}(x_i).$$

Consequently the loop of Figure (Figure 3.1) can be unrolled to complete a feed forward parallel path as shown in Figure (Figure 3.2).

3.1 Initialization of The States For Prior to Running The ODE System

In order to solve the ODE system, one has to supply the initial states to the solver. This can be done by using VIO core through PC connection to the FPGA kit. Although VIO core is easily programmed for initialization, there is a restriction that user have to input 8×32 bits manually from this ChipScopeTM interface. Instead of this, a single initialize command can be issued via an IO interface on-board or via VIOTMCore to perform randomly generated system states where the error dynamics of them diminishes after synchronization. We used the fact that the initial values are to be random and non-zero at the initialization.

3.2 A Random Number Generator for FPGA

We utilized a Linear Feedback Shift Register (LFSR) for generation of randomized states for the ODE system. In this circuitry, output of a serial in parallel shift register is feedback to itself via xor operations of several output bits. Namely, we built a random 32 bit LFSR for each state of the dynamical system and the LFSR's feedback consisted of XORing 0th, 1st, 21st and 31st bits. The random generated output of one of them at initialization is shown in Figure (Figure 3.4).

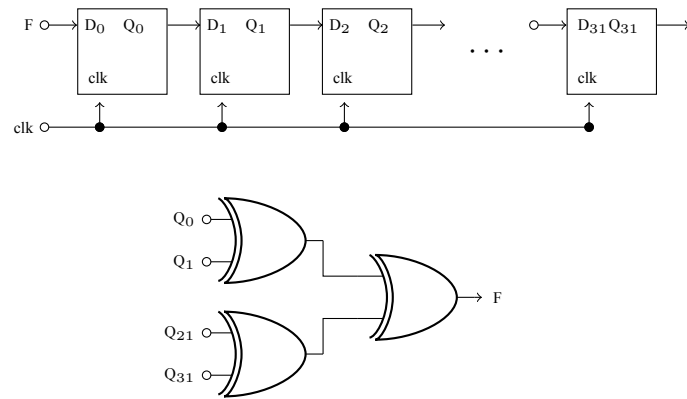


Figure 3.3 32-bit LFSR schematic.

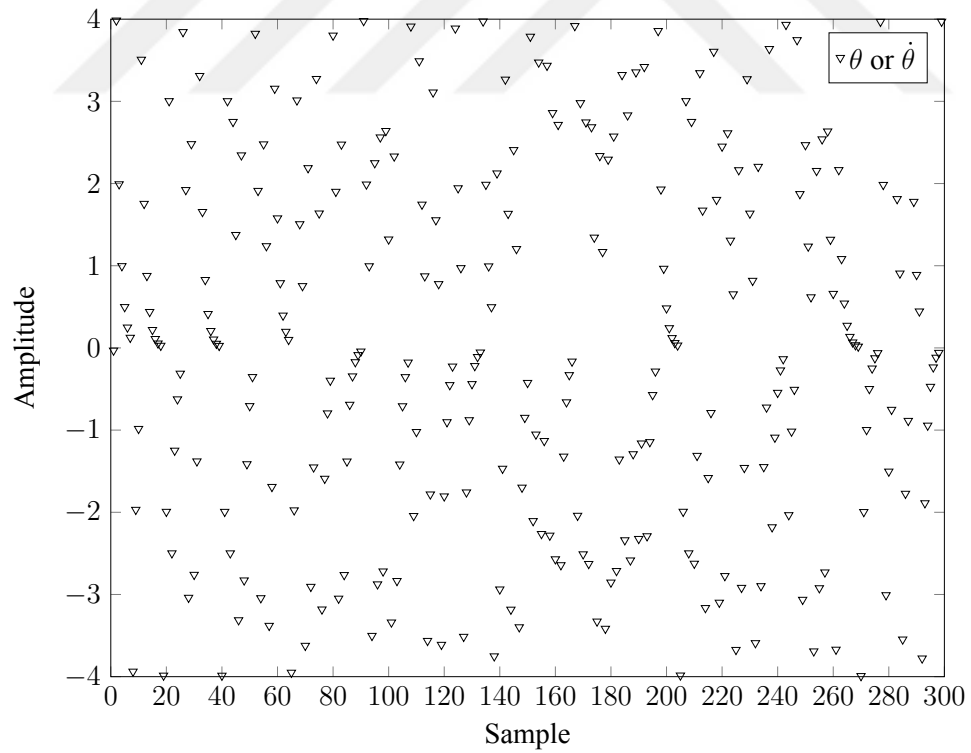


Figure 3.4 Output waveform of the 32-bit LFSR random number generator.

3.3 Case 1: A Single Dimensional Dynamical System That Has an Analytic Solution

Consider a RC circuit of Figure (Figure 3.5). The solution is well-known to satisfy the following equation;

$$V(t) = \frac{dv_C}{dt}RC + v_C. \quad (3.1)$$

The solution of this ODE comes with the multiplication with integration constant $e^{\frac{t}{RC}}$,

$$v_C(t_f) = V(1 - e^{-\frac{t_f}{RC}}). \quad (3.2)$$

The solution is given in Equation (3.2). Whence the circuitry of the parent equation that is an unsolved ODE reduces to single monolithic feed forward logical circuitry. This is shown in Figure (Figure 3.5). The block $1 - e^{-\frac{t_f}{RC}}$ can be calculated as a wholly integrated look-up table or this expression can be calculated with an unrolled circuitry that iterates Maclaurin series to maintain a feed forward structure. The $N = 9$ partial

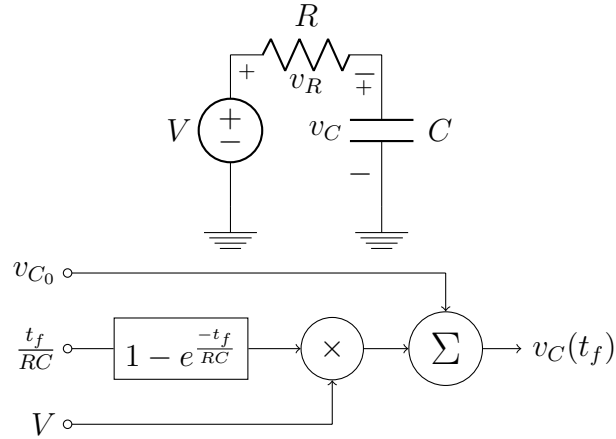


Figure 3.5 A single dimensional dynamical system that has an analytic solution and its feed forward implementation.

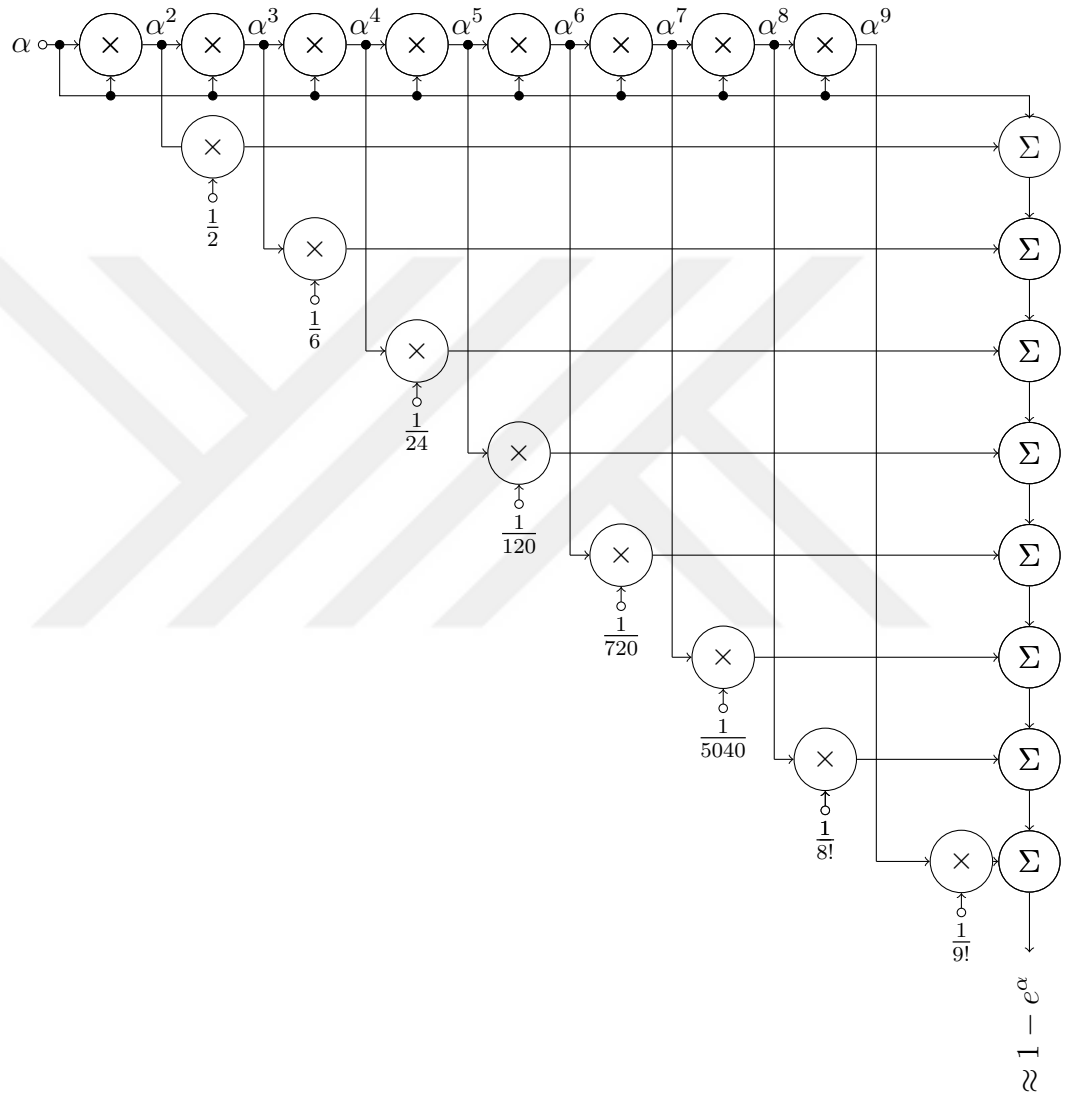


Figure 3.6 Unrolled feed forward circuitry to compute $1 - e^\alpha$ for the circuitry of Figure (Figure 3.5).

series expansion of a function $F(x)$ is as follows:

$$F(x) = \sum_{n=0}^{\infty} \frac{F^{(n)}(x_0)}{n!} (x - x_0)^n \bigg|_{x_0=0} \Rightarrow F(x) = \sum_{n=0}^{\infty} \frac{F^{(n)}(0)}{n!} x^n \quad (3.3)$$

$$F(\alpha) = 1 - e^\alpha = 1 - \frac{\alpha^0}{0!} - \frac{\alpha^1}{1!} - \frac{\alpha^2}{2!} + \text{Higher Order Terms}, \quad (3.4)$$

$$\approx -\alpha - \frac{\alpha^2}{!2} - \frac{\alpha^3}{!3} - \frac{\alpha^4}{!4} - \frac{\alpha^5}{!5} - \frac{\alpha^6}{!6} - \frac{\alpha^7}{!7} - \frac{\alpha^8}{!8} - \frac{\alpha^9}{!9}. \quad (3.5)$$

3.4 Case 2: A Single Dimensional Dynamical System Assuming No Analytic Solution At All

If an analytic solution does not exist, the Equation (3.1) can be discretized to a difference equation by assuming $\dot{v}_C \Delta t \approx v_C[k+1] - v_C[k]$, where k is an integer variable that is denoted to belong to discrete time domain and Δt denotes the time interval between two consecutive discrete times k and $k+1$. After using this Euler expansion that approximates the derivative, the Equation (3.1) becomes,

$$\frac{v_C[k+1] - v_C[k]}{\Delta t} = \frac{V - v_C[k]}{RC} \quad (3.6)$$

$$v_C[k+1] = (V - v_C[k]) \frac{\Delta t}{RC} + v_C[k]. \quad (3.7)$$

There exist better discretization techniques such as Runge Kutta (RK) method. They require more internal stages leading much more complex circuitry and a larger algebraic loop path to be encountered. The Figure (Figure 3.7) shows the algebraic loop of Equation (3.7).

Furthermore the algebraic loop of Euler approximated ODE can be unrolled to complete m finite number of steps as indicated in Figure (Figure 3.8) to solve capacitor voltage v_C after $m\Delta t$ seconds of exposure to V bias. Alternatively the structure in Figure (Figure 3.9) can be used for calculations at each cycle.

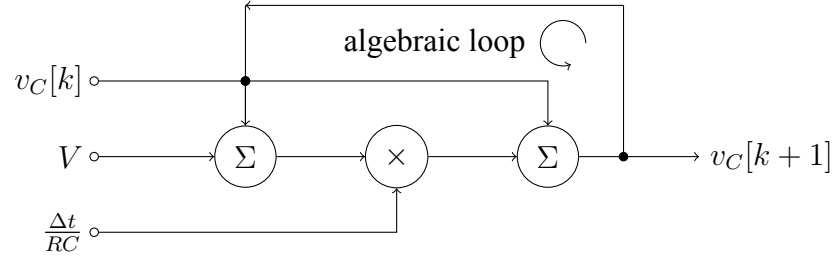


Figure 3.7 Iterative circuit of the dynamical system of Figure (Figure 3.5) that solves Explicit Euler discretization where $v_C[k]$ denotes initial condition of capacitor voltage, $\frac{\Delta t}{RC}$ denotes a resolution based scale constant and V denotes capacitor bias voltage.

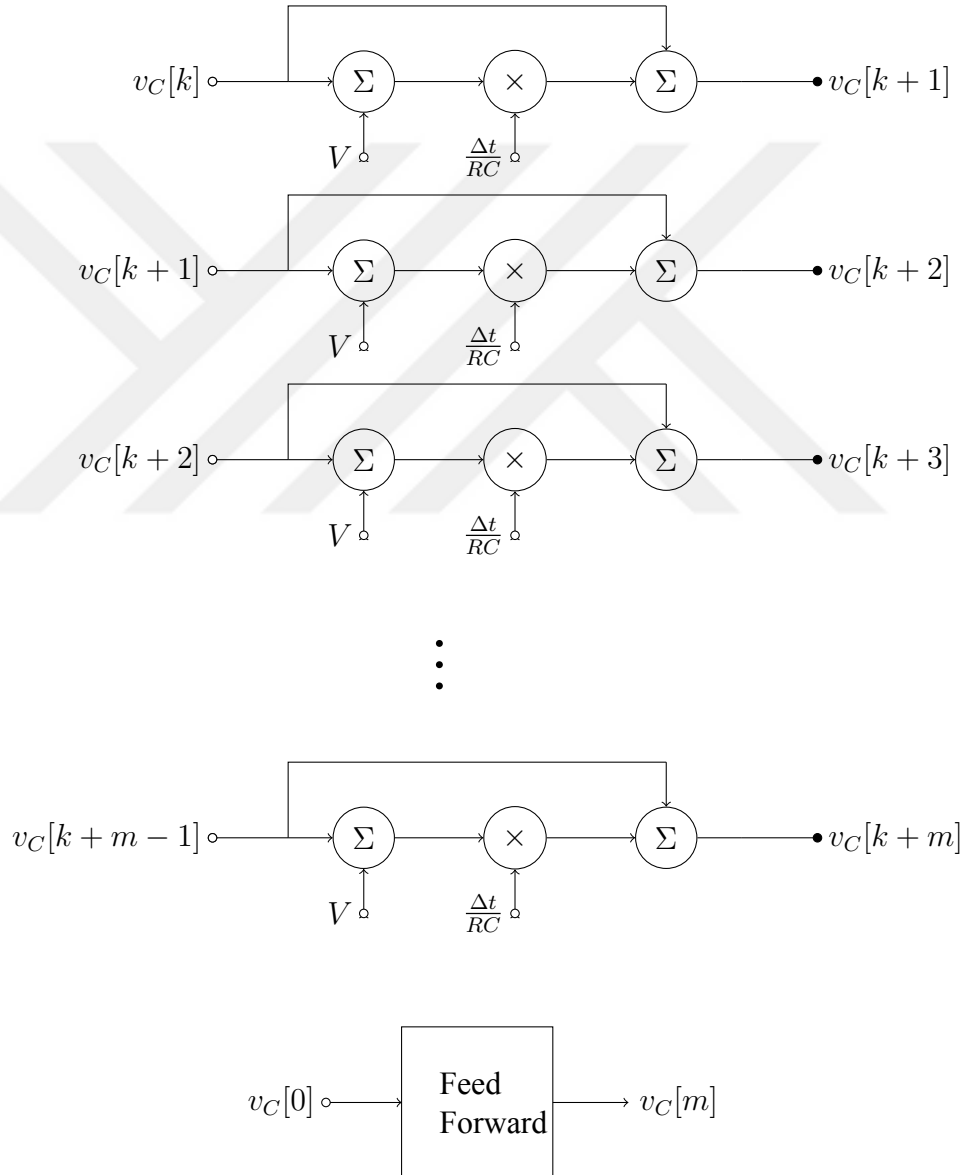


Figure 3.8 Unrolled iterative circuit of the dynamical system of Figure (Figure 3.5) that solves explicit Euler discretization where $v_C[k]$ denotes initial condition of capacitor voltage, $\frac{\Delta t}{RC}$ denotes a resolution based scale constant and V denotes capacitor bias voltage. This iteration can be written as $v_C[k+1] = F(v_C[k])$.

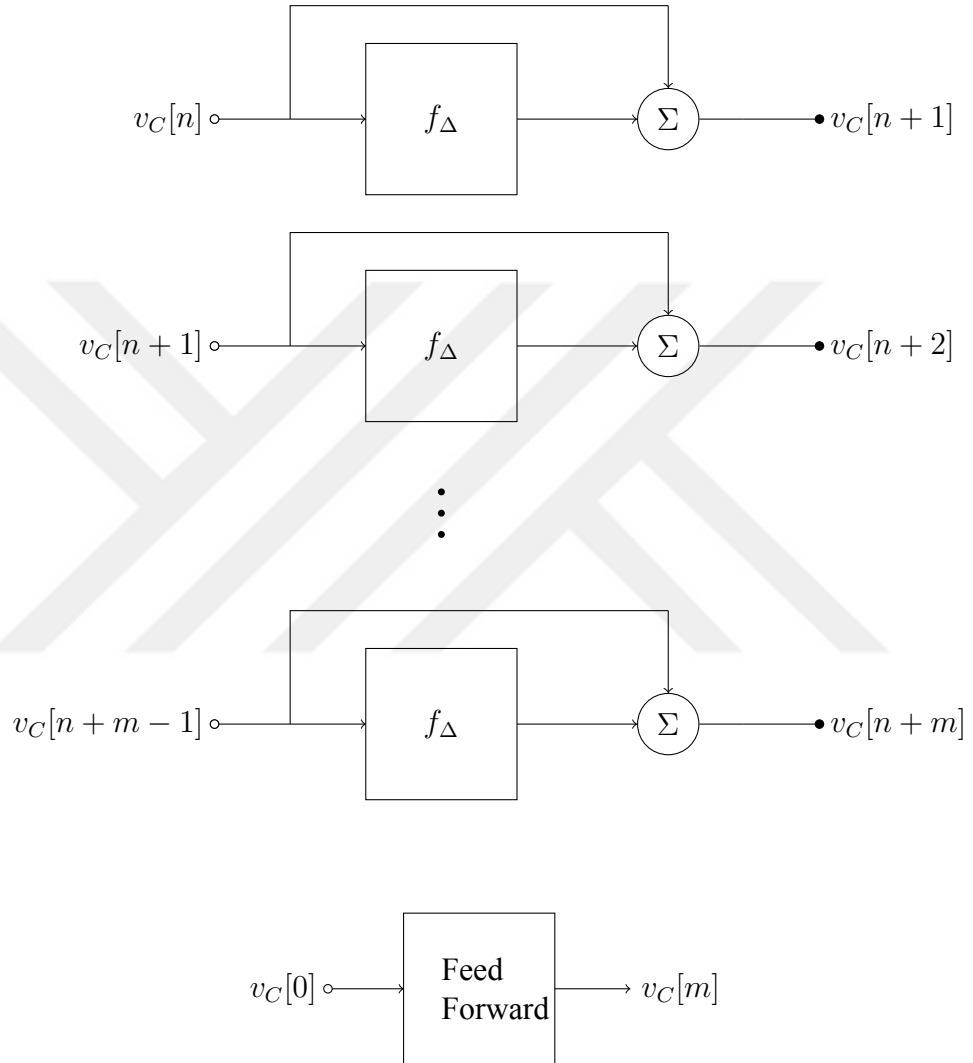


Figure 3.9 Iterative circuit of the dynamical system of Figure (Figure 3.5) that solves explicit Euler discretization where $v_C[n]$ denotes initial condition of capacitor voltage, This iteration can be written as $v_C[n+1] = f_{\Delta}(v_C[n]) + v_C[n]$.

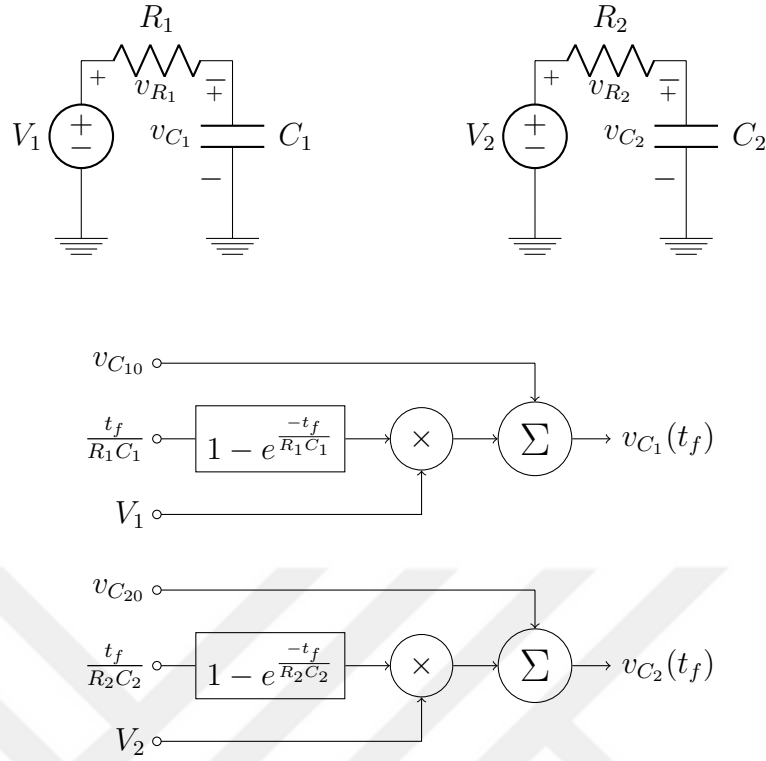


Figure 3.10 Two independent RC circuitry model that is to be implemented in parallel in a logical circuitry.

3.5 Considering Multiple Linear Dynamical Systems

Let us investigate the case of multiple systems, say we have more than one identical systems. The first assumption is there exist no coupling between them as shown in Figure (Figure 3.10), which means they are separated, but they are in the same logical circuitry as shown in Figure (Figure 3.10). The solution is the same with Equation (3.2) and the solution provided by a parallel in parallel out single circuitry. Next, we can establish a linear coupling between them, we can introduce Figure (Figure 3.11). The solution for this system can be written in the following equations;

$$R_1 C_1 \frac{dv_{C_1}}{dt} + v_{C_1} = v_{C_2} - v_{C_1} \quad (3.8)$$

$$R_2 C_2 \frac{dv_{C_2}}{dt} + v_{C_2} = v_{C_1} - v_{C_2} \quad (3.9)$$

$$\begin{bmatrix} \frac{dv_{C_1}}{dt} \\ \frac{dv_{C_2}}{dt} \end{bmatrix} = \begin{bmatrix} \frac{-2}{R_1 C_1} & \frac{1}{R_1 C_1} \\ \frac{1}{R_2 C_2} & \frac{-2}{R_2 C_2} \end{bmatrix} \begin{bmatrix} v_{C_1} \\ v_{C_2} \end{bmatrix} \quad (3.10)$$

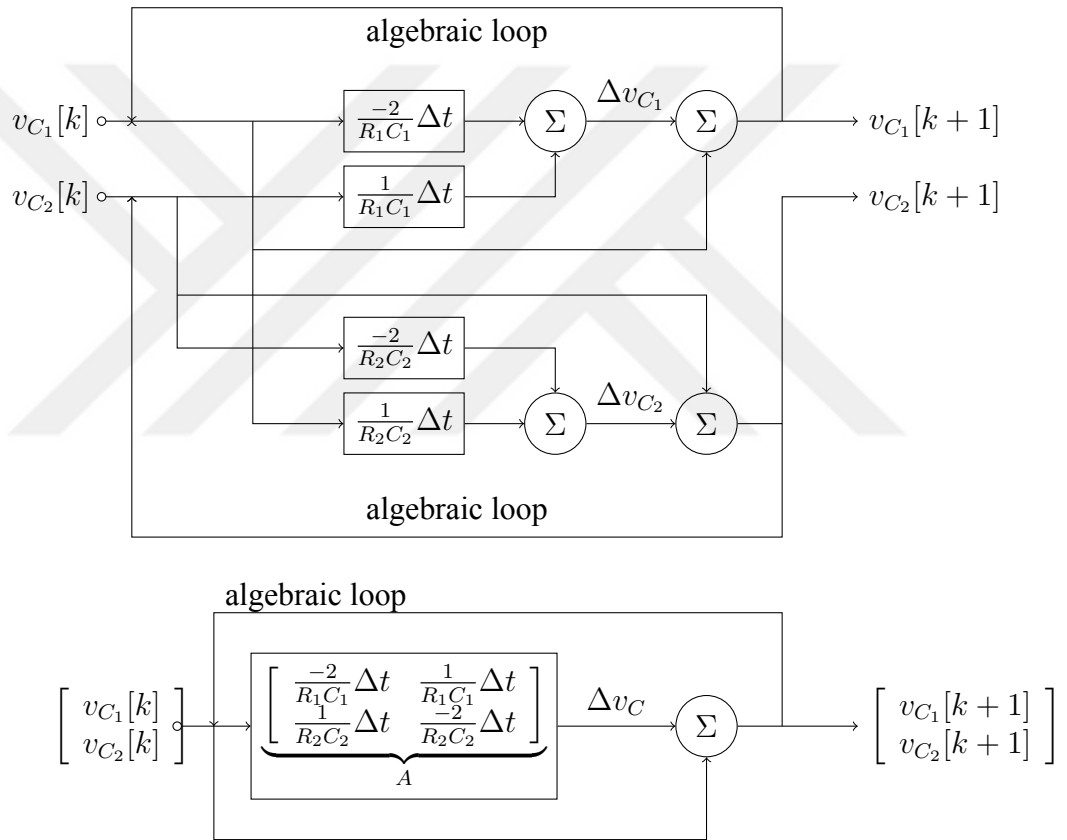
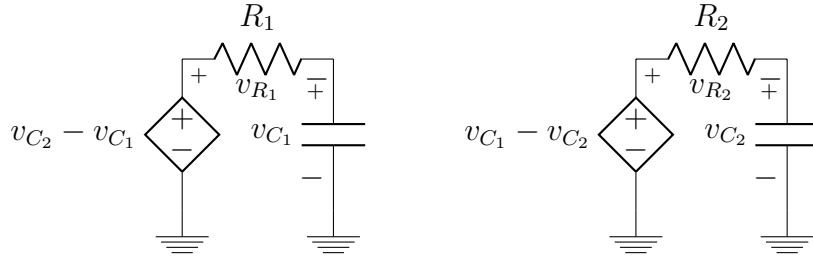


Figure 3.11 Two linearly coupled RC circuitry that is to be implemented in parallel in a logical circuitry with the same unrolling as shown in Figure (Figure 3.9). The matrix product can be implemented in parallel since every row of matrix A is multiplied with the initial value $v_C[k]$. The multiplication of each row with the initial vector and summation of the dot product can be depicted as same with other rows. The length of combinational path for each MAD operation is the same with any other so the matrix product can be implemented in parallel.

Let,

$$\frac{d}{dt} \begin{bmatrix} v_{C_1} \\ v_{C_2} \end{bmatrix} = \begin{bmatrix} \frac{-2}{R_1 C_1} & \frac{1}{R_1 C_1} \\ \frac{1}{R_2 C_2} & \frac{-2}{R_2 C_2} \end{bmatrix} x = \frac{A}{\Delta t} x \quad (3.11)$$

$$\frac{dx}{dt} = \frac{A}{\Delta t} x \quad (3.12)$$

$$e^{\frac{A}{\Delta t} t} \frac{dx}{dt} - e^{\frac{A}{\Delta t} t} \frac{A}{\Delta t} x = 0 \quad (3.13)$$

$$\frac{d}{dt} (e^{\frac{A}{\Delta t} t} x) = 0 \quad (3.14)$$

$$\int_{t=0}^{t_f} d(e^{\frac{A}{\Delta t} t} x(t)) = 0 \quad (3.15)$$

$$e^{\frac{A}{\Delta t} t_f} x(t_f) - x(0) = 0 \quad (3.16)$$

$$x(t_f) = e^{\frac{A}{\Delta t} t_f} x(0). \quad (3.17)$$

The coupled RC circuit described with Equation (3.17) with input states can be implemented as shown in Figure (Figure 3.12). If we just do not want to solve Equation (3.12) analytically we can discretize this equation according to (Chen, 1998) in the following manner;

$$\frac{dx}{dt} = \lim_{\Delta t \rightarrow 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} = \frac{A}{\Delta t} x(t) \quad (3.18)$$

$$x(t + \Delta t) = x(t) + Ax(t) \quad (3.19)$$

$$x((k + 1)\Delta t) = (I + A)x(k\Delta t), \quad k \in \mathbb{N} \quad (3.20)$$

$$x[k + 1] = (A + I)x[k]. \quad (3.21)$$

This Euler approximation of Equation (3.21) is implemented in Figure (Figure 3.11). The result is similar with Figure (Figure 3.7) apart from matrix multiplication and bias voltages. The matrix multiplication in Figure (Figure 3.11) is seen to be parallel in row multiplications of the matrix, that is, both two rows of the matrix is multiplied and summed with the states at the same part of processing time.

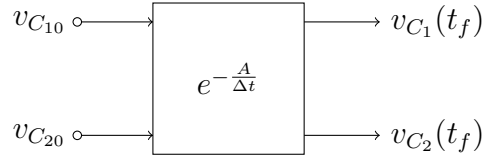


Figure 3.12 Two linearly coupled RC circuitry model that is to be implemented in parallel in a logical circuitry with parallel in parallel out states.

3.6 Considering Multiple Dynamical Systems Including a Non-linearity

We selected out nonlinear dynamical ODE system as the model of a unity period undamped pendulum in this case where the modeled rigid body equation of motion is defined as in the following equation;

$$\frac{d^2\theta}{dt^2} = -\sin\theta. \quad (3.22)$$

The trigonometric function $\sin\theta$ is the source of non-linearity that we want to include. The explicit Euler discretized equation set can be defined as:

$$\theta[k+1] = \omega[k]\Delta t + \theta[k] \quad (3.23)$$

$$\omega[k+1] = -\sin\theta[k]\Delta t + \omega[k]. \quad (3.24)$$

For two pendulum system, the unity coupling coefficient linearly coupled network system dynamics can be written as;

$$\frac{d\theta_1}{dt} = \omega_1 \quad (3.25)$$

$$\frac{d\omega_1}{dt} = -\sin\theta_1 \quad (3.26)$$

$$\frac{d\theta_2}{dt} = \omega_2 + \theta_1 - \theta_2 \quad (3.27)$$

$$\frac{d\omega_2}{dt} = -\sin\theta_2 + \omega_1 - \omega_2. \quad (3.28)$$

The Euler kind discretization of this system yields the following discrete equation set;

$$\theta_1[k + 1] = \omega_1[k]\Delta t + \theta_1[k] \quad (3.29)$$

$$\omega_1[k + 1] = (-\sin \theta_1[k])\Delta t + \omega_1[k] \quad (3.30)$$

$$\theta_2[k + 1] = (\omega_2[k] + \theta_1[k] - \theta_2[k])\Delta t + \theta_2[k] \quad (3.31)$$

$$\omega_2[k + 1] = (-\sin \theta_2[k] + \omega_1[k] - \omega_2[k])\Delta t + \omega_2[k]. \quad (3.32)$$

The right hand side of the difference equation of ODE system that is formed by Equations (3.23,3.24) is

$$f_{\Delta_1}(\cdot) = \begin{bmatrix} \omega[k] \\ -\sin \theta[k] \end{bmatrix} \Delta t \quad (3.33)$$

and the difference equation of ODE system that is consist of Equations (3.29, 3.30, 3.31, 3.32) can be defined as;

$$f_{\Delta_2}(\cdot) = \begin{bmatrix} \omega_1[k] \\ -\sin \theta_1[k] \\ \omega_2[k] + \theta_1[k] - \theta_2[k] \\ -\sin \theta_2[k] + \omega_1[k] - \omega_2[k] \end{bmatrix} \Delta t. \quad (3.34)$$

3.6.1 FPGA Implementation of Difference Equations Consisting of Trigonometric Functions

The implementation the difference equation functions of Equations (3.33) and (3.34) includes sinus operation and this is the bottleneck of the this FPGA implementation. Since implementation of Figure (Figure 2.2) is consisted of MAD operations, the longest combinational path that occurs in Equation (3.34) is this path i.e. consider the calculation of $-\sin \theta_1[k] + \theta_2[k] - \theta_1[k]$ the $\sin \theta_1$ operation have to be completed before the addition and subtraction operations. Therefore, the speed and parallel computation is limited to here. However, the other part of the equation where $-\sin \theta_2[k] + \theta_1[k] - \theta_2[k]$ is calculated is implemented in parallel, i.e. both $\sin \theta_1$ and

$\sin \theta_2$ calculations work in parallel.

The implemented FPGA solvers' ChipScope™ plots are captured. In Figure (Figure 3.13b) the error norm $e[k]$ of all of the states in equation set described with Equations (3.30), (3.31), (3.32) is absolutely normed with respect to Equation (3.29) and summed. The $e[k]$ is defined as;

$$e[k] = \sum_{i=1}^2 |\theta_i - \theta_{i_0}| + \sum_{i=1}^2 |\omega_i - \omega_{i_0}| \quad (3.35)$$

3.7 Analysis of FPGA Implementation of Star Network of Identical Pendulums

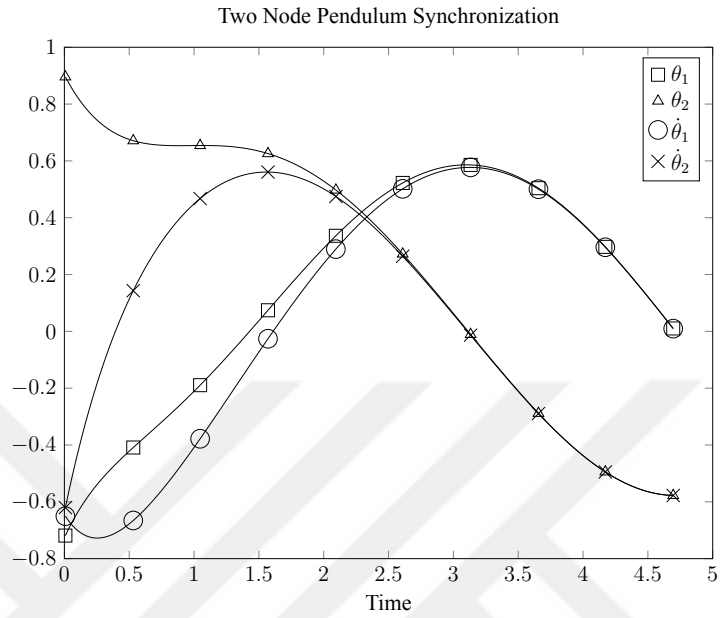
Used resources are analyzed here for two, three, four, five, and six identical pendulum of networks of star connection. The most costly equipment here is sinus block where CORDIC algorithm is used. That can be seen when we compare the Tables (Table 2.1), (Table 2.2) and (Table 3.1). We can have an idea of how much resource is consumed for $1 - e^\alpha$ at $O(\alpha^{10})$ resolution when compared to CORDIC block at $N = 20$ resolution on Spartan 6. Additionally, we can compare them with two, three, four, five and six node pendulum system that has a CORDIC resolution of $N = 22$ iterations for 24 bit resolution. What we want to conclude from the Table (Table 3.1) is that minimum period of the system is approximately 85ns for star coupling. This amount reduced to 55ns for 16 bit implementations. The 16 bit implementation of six node star network runs but granular error becomes visible at this resolution as shown in Figure (Figure 3.14b) which is not desired.

3.7.1 Comparison with Sequential Programming

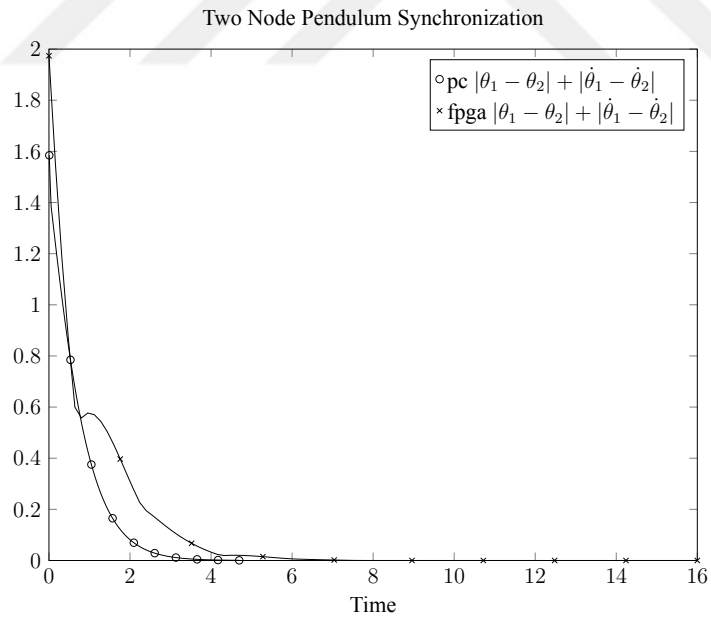
As we can see in the Table (Table 3.3) MATLAB™ `ode45(...)` and `ode23(...)` on IEEE754-32 floating numbers are depicted. Floating numbers have great range of resolution when compared to 32 bit 24 bit and 16 bit fixed numbers. However, when

Table 3.1 Comparison of design summaries.

	Four input LUTs	Registers	Path delay/Min. Period	DSP48A1s
Nine Term Maclaurin $1 - e^{\alpha}$	-	-	67.029ns	24/58(%41)
Nine Term Maclaurin $1 - e^{\alpha}$ (no DSP48A1)	1523/27288(%5)	-	41.400ns	-
Two Pendulum With CORDIC IP CORE 32bit	9135/27288(%33)	654/54576(%1)	118.375ns	-
Two Pendulums With CORDIC IP CORE 24bit	5664/27288(%20)	586/54576(%1)	85.710ns	-
Three Pendulums With CORDIC IP CORE 24bit	8497/27288(%31)	680/54576(%1)	85.710ns	-
Four Pendulums With CORDIC IP CORE 24 bit	11465/27288(%42)	774/54576(%1)	85.716ns	-
Five Pendulums With CORDIC IP CORE 24 bit	14236/27288(%52)	845/54576(%1)	85.716ns	-
Six Pendulums With CORDIC IP CORE 24 bit	17102/27288(%62)	893/54576(%1)	85.716ns	-



(a) Implemented FPGA solver of two node pendulum system.



(b) Obtained norm $e[k]$.

Figure 3.13 Two node synchronization.

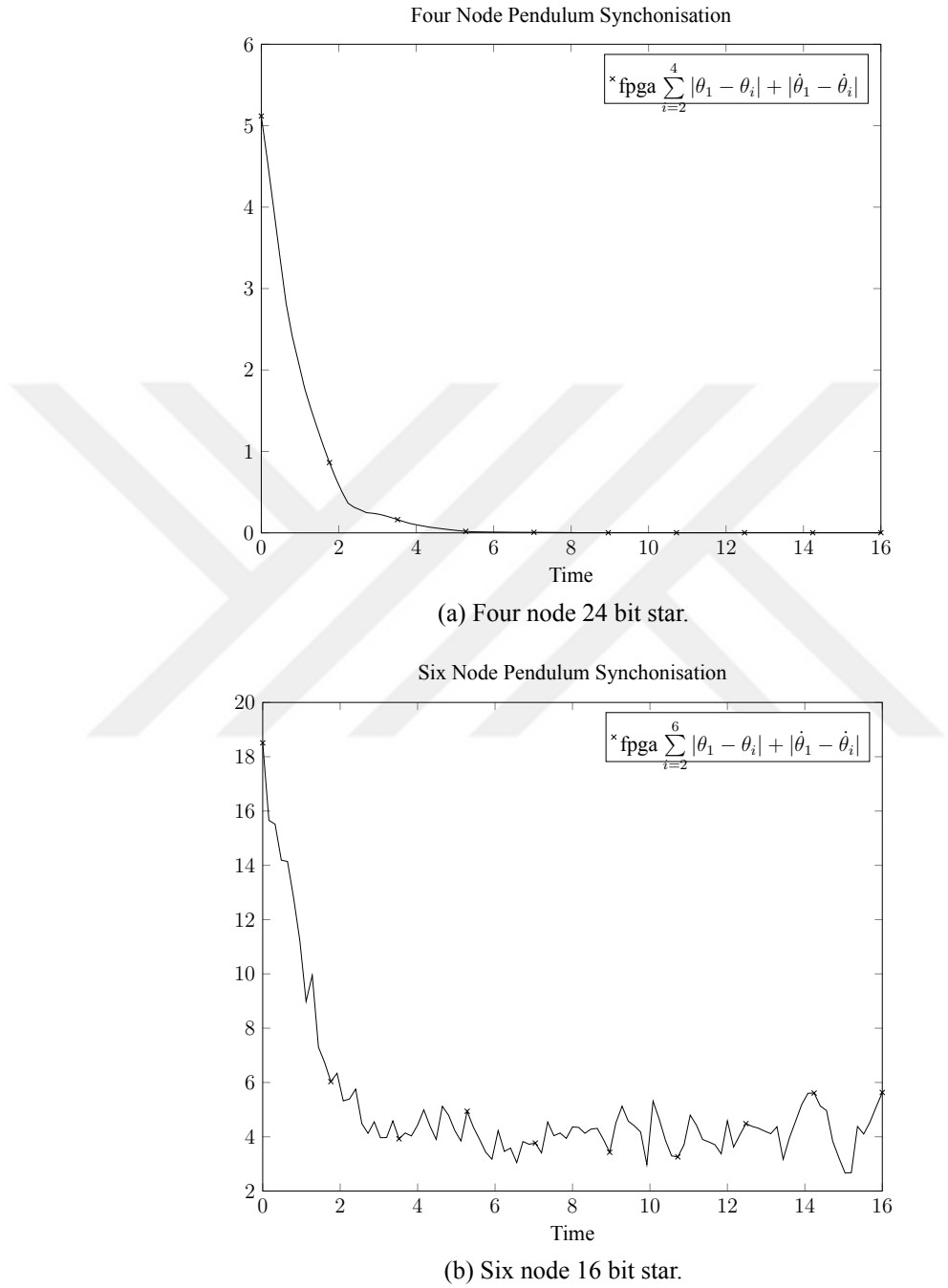


Figure 3.14 Four node star synchronization for 24 bit and six node star synchronization 16 bit implementations.

Table 3.2 Comparison of design summaries 16 bit

# of nodes	Four input LUTs	Registers	Path delay/Min. Period	DSP48A1s
2	3121/27288(%11)	522/54576(%0)	55.426ns	-
3	4441/27288(%16)	584/54576(%1)	55.426ns	-
4	6084/27288(%22)	631/54576(%1)	55.426ns	-
5	7416/27288(%27)	678/54576(%1)	55.426ns	-
6	9025/27288(%33)	710/54576(%1)	55.426ns	-

Table 3.3 Mean time needed for one iteration to be completed on PC at 1.2GHz with IEEE754 32 bit float numbers. The execution of the C code does not include any `printf(...)` command that takes additional time.

2 node PC MATLAB™ode45 IEEE754-32	201μs
3 node PC MATLAB™ode45 IEEE754-32	227μs
4 node PC MATLAB™ode45 IEEE754-32	249μs
5 node PC MATLAB™ode45 IEEE754-32	285μs
6 node PC MATLAB™ode45 IEEE754-32	308μs
2 node PC MATLAB™ode23 IEEE754-32	399μs
3 node PC MATLAB™ode23 IEEE754-32	456μs
4 node PC MATLAB™ode23 IEEE754-32	586μs
5 node PC MATLAB™ode23 IEEE754-32	720μs
6 node PC MATLAB™ode23 IEEE754-32	643μs
2 node PC C Explicit Euler IEEE754-32	78ns
3 node PC C Explicit Euler IEEE754-32	120ns
4 node PC C Explicit Euler IEEE754-32	168ns
5 node PC C Explicit Euler IEEE754-32	199ns
6 node PC C Explicit Euler IEEE754-32	235ns
2 node Explicite Euler PC $I = 2, F = 29$ bit Signed	427ns
3 node Explicite Euler PC $I = 2, F = 29$ bit Signed	650ns
4 node Explicite Euler PC $I = 2, F = 29$ bit Signed	783ns
5 node Explicite Euler PC $I = 2, F = 29$ bit Signed	1056ns
6 node Explicite Euler PC $I = 2, F = 29$ bit Signed	1265ns

it comes to FPGA with fixed point numbers, one iteration is approximately 2000 times faster than MATLAB™implementation at 1.2GHz CPU. The C implementation of the Explicit Euler solution is approximately closer to FPGA solution and fixed point FPGA solution is faster for 1.2GHz CPU clock PC for more than two node implementations.

MATLAB™ codes, C codes and Verilog codes are given in the Appendix.



CHAPTER FOUR

CONCLUSION

The discretization of an ODE system yields a difference equation that gives the discretized state evolution. When there exist an analytic solution the difference equation reaches the solution at once and the FPGA implementation becomes a single monolithic feed forward circuitry. In the cases where there exist no analytic solution, the discretization introduces iterations of the difference equation function in the FPGA implementation, whether this is an Euler solution, RK solution or other explicit or implicit methods. In Chapter Three, it is shown that those iterations can be unrolled to complete a single feed forward circuitry, which is consisted of replicas of the right hand side of the difference equation. Those replicas that eliminate both set up and hold times of clocked registers of the algebraic looped structure and this results in a faster implementation. However, it uses much more resources and this implementation needs a wider FPGA area and can only solve finite number of iterations in time, that is, time domain is finite for this solution.

In the initialization of the system states of the dedicated ODE solver system, the data have to be signaled from outside, so I used Xilinx's VIO core to solve this issue. Finally, synchronizing star topology network of pendulum models are analyzed and the resultant hardware's bottleneck is found to be the calculation of trigonometric function sinus where MAD operations are carried out. The result of FPGA implementation is a dedicated ODE solver system for network structure. Even for a single model dedicated FPGA implementation yields parallel paths in difference equation implementation.

There is no question that for larger networks that consist more number of nodes and complex ODE models that consist more algebra, we need to have a wider FPGA area. We can easily deduce that in order to build a network of nodes of a physical model, this area at least has to be proportional to the number of nodes.

REFERENCES

- Arbaugh, J. T. (2004). *Table look-up CORDIC: effective rotations through angle partitioning*. PhD thesis, University of Texas, USA.
- Barahona, M., & Pecora, L. M. (2002). Synchronization in small-world systems. *Physical Review Letters*, 89(5), 054101.
- Belykh, I., Hasler, M., Lauret, M., & Nijmeijer, H. (2005). Synchronization and graph topology. *International Journal of Bifurcation and Chaos*, 15(11), 3423–3433.
- Brown, R., & Kocarev, L. (2000). A unifying definition of synchronization for dynamical systems. *Chaos*, 10(2), 344–349.
- Caldarelli, G. (2012). *Networks: a very short introduction*. Oxford University Press, 1st ed., USA.
- Chen, C. T. (1998). *Linear system theory and design (Oxford series in electrical and computer engineering)* (3 ed.). Oxford University Press, USA.
- Chua, L., & Green, D. (1976). Graph-theoretic properties of dynamic nonlinear networks. *Circuits and Systems, IEEE Transactions*, 23(5).
- Dorigo, M. (1992). *Learning and natural algorithms*. PhD thesis, Politecnico di Milano, ITALY.
- Ernst, E. (2004). Bioresonance, a study of pseudo-scientific language. *Forsch Komplementärmed Klass Naturheilkd*, 11, 171-173.
- Fan, Z., & Chen, G. (2002). Pinning control of scale-free complex networks. *Proceedings - IEEE International Symposium on Circuits and Systems*, 310, 521–531.
- Fink, K., Johnson, G., Carroll, T., Mar, D., & Pecora, L. (2000). Three coupled oscillators as a universal probe of synchronization stability in coupled oscillator arrays. *Physical review. E, Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, 61(5A), 5080–90.

- Fujisaka, H., & Yamada, T. (1986). Stability theory of synchronized motion in coupled-oscillator systems. *Progress of Theoretical Physics*, 72(5), 885–894.
- Huang C, Vahid F, G. T. (2011). A custom fpga processor for physical model ordinary differential equation solving. *IEEE Embedded Systems Letters*, 3(4), 113-116.
- Luo, A. C. J. (2013). *Dynamical system synchronization*. Springer, New York.
- Mathews, J. H. (1992). *Numerical methods for mathematics, science, and engineering*. Prentice-Hall, 2nd ed., USA.
- Nelson, A. E. (2000). *Implementation of image processing algorithms on FPGA hardware*. MSc. Thesis, Graduate School of Vanderbilt University, USA.
- Nievergelt, J. (1964). Parallel methods for integrating ordinary differential equations. *Communications of The ACM*, 7(12).
- Pikovsky A, Rosenblum M, K. J. (2003). *Synchronization : a universal concept in nonlinear Science*. UK, Cambridge University Press.
- Sanchez, E., Matias, M. A., & Perez-Munuzuri, V. (2000). Chaotic synchronization in small assemblies of driven chua's circuits. *IEEE Transactions On Circuits And Systems I Fundamental Theory And Applications*, 47(5), 644–654.
- Venkataraman, V. (2004). Simulation of a heterogeneous system at multiple levels of abstraction using rendezvous based modeling. *10th International Workshop on Microprocessor Test and Verification*, 113-116.
- Wang, W., & Cao, J. (2006). Synchronization in an array of linearly coupled networks with time-varying delay. *Physica A: Statistical Mechanics and its Applications*, 366, 197–211.
- Wang, X. F. (2002). Complex networks: topology, dynamics and synchronization. *International Journal of Bifurcation and Chaos*, 12(05), 885–916.
- Wu, C. W. (2005). Synchronization in networks of nonlinear dynamical systems coupled via a directed graph. *Nonlinearity*, 18, 1057–1064.

Wu, C. W., & Chua, L. O. (1996). On a conjecture regarding the synchronization in an array of linearly coupled dynamical systems. *IEEE Transactions on Circuits and Systems*, 43(2), 161–165.

Xiang, L. Y., Liu, Z. X., Chen, Z. Q., Chen, F., & Yuan, Z. Z. (2007). Pinning control of complex dynamical networks with general topology. *Physica A*, 379, 298–306.

Xilinx™(2014). *Spartan-6 FPGA DSP48A1 Slice User Guide (UG389)*, technical report, retrieved on 02/01/2017 from the link http://www.xilinx.com/support/documentation/user_guides/ug389.pdf.

Yu, W., Chen, G., & Lü, J. (2009). On pinning synchronization of complex dynamical Networks. *Automatica*, 45(2), 429–435.

APPENDICES

A.1 $F = 31$ Bit $N = 20$ Iteration CORDIC Block With Algebraic Loop

```
'timescale 1ns / 1ps
'include "macros.v"

module cordic(
    theta ,
    cosTheta ,
    sinTheta ,
    startConversion ,
    outputReady ,
    clk
);

    input wire signed [(‘BITWIDTH-1):0] theta;
    output wire signed [(‘BITWIDTH-1):0] cosTheta;
    output wire signed [(‘BITWIDTH-1):0] sinTheta;
    input wire startConversion;
    output reg outputReady;
    input wire clk;

    reg [5:0] i; //for loop variable
    reg signed [(‘BITWIDTH-1):0] xSum;
    reg signed [(‘BITWIDTH-1):0] ySum;
    reg signed [(‘BITWIDTH-1):0] xSumBuffer;
    reg signed [(‘BITWIDTH-1):0] ySumBuffer;
    reg signed [(‘BITWIDTH-1):0] zSum;
    reg signed [(‘BITWIDTH-1):0] thetaReg;

    wire signed [(‘BITWIDTH-1):0] lookUpOneByOnePlusTwoToTheMinusTwoI;
    wire signed [(‘BITWIDTH-1):0] lookUpArtTanTwoToTheMinusOneI [19:0];

    wire signed [(2*‘BITWIDTH-1):0] xSumScaled;
    wire signed [(2*‘BITWIDTH-1):0] ySumScaled;

    assign lookUpOneByOnePlusTwoToTheMinusTwoI = ‘BITWIDTH’
        b000000001000011011100100101010011110;
    assign lookUpArtTanTwoToTheMinusOneI[0] = ‘BITWIDTH’
        b0000101010011011010001100100101001;
    assign lookUpArtTanTwoToTheMinusOneI[1] = ‘BITWIDTH’
        b00001000110110110111000011001001011;
    assign lookUpArtTanTwoToTheMinusOneI[2] = ‘BITWIDTH’
        b000001100100100001111110110101010001;
    assign lookUpArtTanTwoToTheMinusOneI[3] = ‘BITWIDTH’
        b000000111011010110001100111000001011;
```

```

assign lookUpArtTanTwoToTheMinusOneI[4] = 'BITWIDTH'
    b000000011111010110110111010111111001;
assign lookUpArtTanTwoToTheMinusOneI[5] = 'BITWIDTH'
    b00000000111111010101101110101001101;
assign lookUpArtTanTwoToTheMinusOneI[6] = 'BITWIDTH'
    b00000000011111110101011011101110;
assign lookUpArtTanTwoToTheMinusOneI[7] = 'BITWIDTH'
    b00000000001111111110101010110111;
assign lookUpArtTanTwoToTheMinusOneI[8] = 'BITWIDTH'
    b0000000000011111111110101010110;
assign lookUpArtTanTwoToTheMinusOneI[9] = 'BITWIDTH'
    b0000000000001111111111110101011;
assign lookUpArtTanTwoToTheMinusOneI[10] = 'BITWIDTH'
    b00000000000001111111111111010101;
assign lookUpArtTanTwoToTheMinusOneI[11] = 'BITWIDTH'
    b0000000000000011111111111111011;
assign lookUpArtTanTwoToTheMinusOneI[12] = 'BITWIDTH'
    b0000000000000001111111111111111;
assign lookUpArtTanTwoToTheMinusOneI[13] = 'BITWIDTH'
    b00000000000000001000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[14] = 'BITWIDTH'
    b00000000000000001000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[15] = 'BITWIDTH'
    b00000000000000001000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[16] = 'BITWIDTH'
    b00000000000000001000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[17] = 'BITWIDTH'
    b00000000000000001000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[18] = 'BITWIDTH'
    b00000000000000001000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[19] = 'BITWIDTH'
    b00000000000000001000000000000000;

assign xSumScaled = xSum □ lookUpOneByOnePlusTwoToTheMinusTwoI;
assign ySumScaled = ySum □ lookUpOneByOnePlusTwoToTheMinusTwoI;

assign cosTheta = xSumScaled >>> ('BITWIDTH-1');
assign sinTheta = ySumScaled >>> ('BITWIDTH-1');

```

```

‘ifndef TOP_MODULE // if top module does not exist assign an input value of 1rad.
‘ifndef SYNTHESIZE
    real xSumScaled2 ;
    real ySumScaled2 ;

    reg [('BITWIDTH-1):0] thetaDriver;

```

```

    assign theta = thetaDriver;
    initial begin
        #10
        thetaDriver = 'BITWIDTH'b000001111001110000000000000000000000;
        #30
        $finish();
    end
end
`endif
`endif

always @(posedge clk) begin

    if (startConversion == 0) begin
        i = 0;
        xSum = 'BITWIDTH'b00000000000000000000000000000000;
        ySum = (theta > 0)?
        'BITWIDTH'b000010000000000000000000000000000000 :
        'BITWIDTH'b111110000000000000000000000000000000;
        zSum = (theta > 0)?
        'BITWIDTH'b000011001001000011111101101010100010 :
        'BITWIDTH'b111100110110111100000010010101011110;
        thetaReg = theta;
    end

    if (i < 20) begin
        if (zSum < thetaReg) begin
            zSum = zSum + lookUpArtTanTwoToTheMinusOneI[i];
            xSum = xSum - (ySum >>> i);
            ySum = ySum + (xSum >>> i);
        end
        else if (zSum > thetaReg) begin
            zSum = zSum - lookUpArtTanTwoToTheMinusOneI[i];
            xSum = xSum + (ySum >>> i);
            ySum = ySum - (xSum >>> i);
        end
        i = i + 1;
    end

    if (i == 20) begin
        outputReady = 1;
    end

    //xSum = xSumBuffer;
    //ySum = ySumBuffer;

    `ifndef SYNTHESIZE
    #1
    xSumScaled2 = xSumScaled / $itor(1'b1 <<< 31);
    ySumScaled2 = ySumScaled / $itor(1'b1 <<< 31);

```

```

        'endif
        'ifndef SYNTHESIZE
        $display("i=%d\n,T=%1.10f\n,ySumScaled=%1.10f,xSumScaled=%1.10f\n,
                zSum=%1.10f\n",
                i,
                theta / $itor(1'b1 <<< 31),
                ySumScaled2 / $itor(1'b1 <<< 31),
                xSumScaled2 / $itor(1'b1 <<< 31),
                zSum / $itor(1'b1 <<< 31) );
        'endif

        'ifndef SYNTHESIZE
        $display("i=%d\n,T=%1.10f\n,ySumScaled=%1.10f,xSumScaled=%1.10f\n,zSum=%1.10f
                \n",
                i,
                theta / $itor(1'b1 <<< 31),
                ySumScaled2 / $itor(1'b1 <<< 31),
                xSumScaled2 / $itor(1'b1 <<< 31),
                zSum / $itor(1'b1 <<< 31) );
        'endif
    end
endmodule

```

A.2 $F = 31$ Bit $N = 20$ Iteration CORDIC Unrolled Design

```

`include "macros.v"

module cordic_unrolled ( theta , cosTheta , sinTheta );

input  wire signed [(`BITWIDTH-1):0] theta ;
output wire signed [(`BITWIDTH-1):0] cosTheta ;
output wire signed [(`BITWIDTH-1):0] sinTheta ;

wire signed [(`BITWIDTH-1):0] xSum [(`ITERATIONS-1):0];
wire signed [(`BITWIDTH-1):0] ySum [(`ITERATIONS-1):0];
wire signed [(`BITWIDTH-1):0] zSum [(`ITERATIONS-1):0];

wire signed [(`BITWIDTH-1):0] lookUpOneByOnePlusTwoToTheMinusTwoI ;
wire signed [(`BITWIDTH-1):0] lookUpArtTanTwoToTheMinusOneI [(`ITERATIONS-1):0];

wire signed [(`BITWIDTH/2-1):0] xSumScaled ;
wire signed [(`BITWIDTH/2-1):0] ySumScaled ;

assign lookUpOneByOnePlusTwoToTheMinusTwoI = `BITWIDTH'
    b000001001101101110100111011011010100 ;
assign lookUpArtTanTwoToTheMinusOneI[0] = `BITWIDTH'
    b00000110010010000111110110101010001 ;
assign lookUpArtTanTwoToTheMinusOneI[1] = `BITWIDTH'
    b000000111011010110001100111000001011 ;
assign lookUpArtTanTwoToTheMinusOneI[2] = `BITWIDTH'
    b0000000111101011011011101011111001 ;
assign lookUpArtTanTwoToTheMinusOneI[3] = `BITWIDTH'
    b00000000111111010101101110101001101 ;
assign lookUpArtTanTwoToTheMinusOneI[4] = `BITWIDTH'
    b0000000001111111010101011011101110 ;
assign lookUpArtTanTwoToTheMinusOneI[5] = `BITWIDTH'
    b0000000000111111111010101010110111 ;
assign lookUpArtTanTwoToTheMinusOneI[6] = `BITWIDTH'
    b0000000000011111111111010101010110 ;
assign lookUpArtTanTwoToTheMinusOneI[7] = `BITWIDTH'
    b0000000000001111111111111010101011 ;
assign lookUpArtTanTwoToTheMinusOneI[8] = `BITWIDTH'
    b000000000000011111111111111010101 ;
assign lookUpArtTanTwoToTheMinusOneI[9] = `BITWIDTH'
    b00000000000000111111111111111011 ;
assign lookUpArtTanTwoToTheMinusOneI[10] = `BITWIDTH'
    b00000000000000011111111111111111 ;
assign lookUpArtTanTwoToTheMinusOneI[11] = `BITWIDTH'
    b000000000000000010000000000000000000 ;

```

```

assign lookUpArtTanTwoToTheMinusOneI[12] = 'BITWIDTH'
      b0000000000000000100000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[13] = 'BITWIDTH'
      b0000000000000000100000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[14] = 'BITWIDTH'
      b0000000000000000100000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[15] = 'BITWIDTH'
      b0000000000000000100000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[16] = 'BITWIDTH'
      b0000000000000000100000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[17] = 'BITWIDTH'
      b0000000000000000100000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[18] = 'BITWIDTH'
      b0000000000000000100000000000000000;
assign lookUpArtTanTwoToTheMinusOneI[19] = 'BITWIDTH'
      b0000000000000000100000000000000000;

assign xSumScaled = xSum[( 'ITERATIONS' - 1)] □ lookUpOneByOnePlusTwoToTheMinusTwoI;
assign ySumScaled = ySum[( 'ITERATIONS' - 1)] □ lookUpOneByOnePlusTwoToTheMinusTwoI;

assign cosTheta = (xSumScaled) >>> 'BITWIDTH;
assign sinTheta = (ySumScaled) >>> 'BITWIDTH;

'ifndef TOP_MODULE //if top module does not exist assign an input value of 1rad.
'ifndef SYNTHESIZE //if xst runs the ndoes not set theta initial value
reg signed [(( 'BITWIDTH' ) - 1):0] thetaDriver;
assign theta = thetaDriver;
initial begin
      $monitor("cosTheta=%b, sinTheta=%b", cosTheta, sinTheta);
      #10
      //thetaDriver = 'BITWIDTH' b111001101101111000000100101010111100;
      thetaDriver = 'BITWIDTH' b000001111001110000000000000000000000;
      #10
      $display("cosTheta=%b, sinTheta=%b", cosTheta, sinTheta);
      $finish();
end
'endif
'endif

assign xSum[0] = 'BITWIDTH' b00000000000000000000000000000000;

assign ySum[0] = (theta > 0)?
  'BITWIDTH' b000010000000000000000000000000000000:
  'BITWIDTH' b111110000000000000000000000000000000;
assign zSum[0] = (theta > 0)?
  'BITWIDTH' b0000110010010000011111101101010100010:
  'BITWIDTH' b111100110110111100000010010101011110;

```

```

genvar i;

generate
    for (i = 0; i < ('ITERATIONS-1); i = i + 1)
        begin : b1

            myBuffer u1 (.out(zSum[i+1]),
                .in(
                    ( zSum[i] < theta ) ?
                        zSum[i] + lookUpArtTanTwoToTheMinusOneI[i] :
                        zSum[i] - lookUpArtTanTwoToTheMinusOneI[i]
                )
            );
            myBuffer u2 (.out(xSum[i+1]),
                .in(
                    ( zSum[i] < theta ) ?
                        xSum[i] - (ySum[i] >>> i) :
                        xSum[i] + (ySum[i] >>> i)
                )
            );
            myBuffer u3 (.out(ySum[i+1]),
                .in(
                    ( zSum[i] < theta ) ?
                        ySum[i] + (xSum[i] >>> i) :
                        ySum[i] - (xSum[i] >>> i)
                )
            );
        end
    endgenerate

endmodule

module myBuffer ( output wire [('BITWIDTH-1):0] out, input wire [('BITWIDTH-1):0] in
    );
    assign out = in;
endmodule

```


A.3 $F = 31$ Bit $N = 20$ Iteration CORDIC Test Bench

```
'timescale 1ns / 1ps
'include "macros.v"

module cordic_tb(
);

reg signed [(BITWIDTH-1):0] theta_tb;
wire signed [(BITWIDTH-1):0] cosTheta_tb;
wire signed [(BITWIDTH-1):0] sinTheta_tb;
integer input_file;
integer output_file;
integer read_status;

cordic_unrolled u0 (.theta(theta_tb), .cosTheta(cosTheta_tb), .sinTheta(sinTheta_tb))
;

initial begin
input_file = $fopen(
"/home/gbas/cordic-lut-gen/test_pattern_file.txt","r");
output_file = $fopen(
"/home/gbas/cordic-lut-gen/test_pattern_output_file.txt",
"w");
while (!$feof(input_file)) begin
read_status = $fscanf(input_file, "%b\n", theta_tb);
#30;
$fwrite(output_file, "'%b',\n", cosTheta_tb, sinTheta_tb);
#10;
end
$fclose(input_file);
$fclose(output_file);
$finish();
end
endmodule
```

A.4 $I + F = 24$ Bit Random Number Generator Module

```
'timescale 1ns / 1ps
#include "constants.v"

module randomNumberGenerator(
    input CLK,
    output reg ['WORDSIZE-1:0] b
);

    reg CLK2;
    reg [4:0] counter;

    always @(posedge CLK) begin
        counter = counter + 1;
        if (counter == 10) begin
            counter = 0;
            CLK2 = ~CLK2;
        end
    end

    parameter seed = 'WORDSIZE'sb0011_1100_0011_0011;

    wire feedback; // 24 bit design
    assign feedback = b[23] ^^ b[22] ^^ b[21] ^^ b[16];

    initial begin
        counter = 5;
        b = seed;
        CLK2 = 0;
    end

    integer i;

    always @(negedge CLK2) begin

        b[0] <= feedback;

        for (i = 0; i < 'WORDSIZE-1; i = i + 1) begin
            b[i+1] <= b[i];
        end
    end

    // debugging
    /□
    wire [35:0] CONTROL0;
    my_ila my_ila1 (
```

```

        .CONTROL(CONTROL0), // INOUT BUS [35:0]
        .CLK(CLK), // IN
        .TRIG0(b) // IN BUS [31:0]
    );
    my_icon my_icon1 (
        .CONTROL0(CONTROL0) // INOUT BUS [35:0]
    );
    □/
endmodule

```



A.5 $I + F = 24$ Bit Absolute Value Calculator

```
`timescale 1ns / 1ps
`include "constants.v"
module abs_val(
    input signed ['WORDSIZE-1:0] in ,
    output signed ['WORDSIZE-1:0] out
);

assign out = (in['WORDSIZE-1])?(~in+1):in;

endmodule
```



A.6 $I + F = 24$ Bit Single Pendulum Module

```
'timescale 1ns / 1ps
'include "constants.v"

module top_mod2(
    CLK_OUT, BRST, packed_u, packed_x
);

input wire CLK_OUT;
input wire BRST;
input wire [2*'WORDSIZE-1:0] packed_u;
output wire [2*'WORDSIZE-1:0] packed_x;

wire signed ['WORDSIZE-1:0] u_sub_1;
wire signed ['WORDSIZE-1:0] u_sub_2;

assign u_sub_1 = packed_u['WORDSIZE-1:0];
assign u_sub_2 = packed_u[2*'WORDSIZE-1:'WORDSIZE];

parameter thetaSeedInitial = 'WORDSIZE'sb0001_1011_0010_0001;
parameter thetaDotSeedInitial = 'WORDSIZE'sb0000_0000_0000_0000;
parameter stepSize = 11;
parameter initGain = 2;

reg signed ['WORDSIZE-1:0] theta;
reg signed ['WORDSIZE-1:0] thetaDot;

initial begin
    theta = thetaSeedInitial; //initialize theta state with pi/4
    thetaDot = thetaDotSeedInitial; //initialize thetaDot state with 0;
end

wire signed ['WORDSIZE-1:0] sinTheta;
wire signed ['WORDSIZE-1:0] sinThetaS;
wire signed ['WORDSIZE-1:0] cosTheta;

assign sinTheta = sinThetaS >>> 1;
my_cordic my_cordic_1 (
    .phase_in(theta), // input [15 : 0] phase_in
    //.x_out(cosTheta), // output [15 : 0] x_out
    .y_out(sinThetaS), // output [15 : 0] y_out
    //.rdy(rdy), // output rdy
    //.clk(CLK_OUT) // input clk
);
```

```

wire signed ['WORDSIZE-1:0] theta_k_plus_one;
wire signed ['WORDSIZE-1:0] thetaDot_k_plus_one;

assign theta_k_plus_one = thetaDot >>> stepSize;
assign thetaDot_k_plus_one = ((~sinTheta + 1)) >>> stepSize ;

assign packed_x = {thetaDot, theta};

wire signed ['WORDSIZE-1:0] thetaInitial;
wire signed ['WORDSIZE-1:0] thetaDotInitial;

randomNumberGenerator #(thetaSeedInitial) rng1 (
    .CLK(CLK_OUT),
    .b(thetaInitial)
);

randomNumberGenerator #(thetaDotSeedInitial) rng2 (
    .CLK(CLK_OUT),
    .b(thetaDotInitial)
);

always @(posedge CLK_OUT) begin
    theta = theta + theta_k_plus_one + (u_sub_1 >>> stepSize);
    thetaDot = thetaDot + thetaDot_k_plus_one + (u_sub_2 >>> stepSize);

    if (~BRST)
    begin
        theta = thetaInitial >>> initGain; //initialize theta
        thetaDot = thetaDotInitial >>> initGain; //initialize thetaDot
    end

    if (theta >= 'WORDSIZE'sb0110_0100_1000_0111_1110_1101_0101_0001)
    begin
        theta = theta + 'WORDSIZE'sb1001_1011_0111_1000_0001_0010_1010_1111 +
            'WORDSIZE'sb1001_1011_0111_1000_0001_0010_1010_1111;
    end

    if (theta <= 'WORDSIZE'sb1001_1011_0111_1000_0001_0010_1010_1111)
    begin
        theta = theta + 'WORDSIZE'sb0110_0100_1000_0111_1110_1101_0101_0001 +
            'WORDSIZE'sb0110_0100_1000_0111_1110_1101_0101_0001;
    end

```

end

endmodule



A.7 $I + F = 24$ Bit Six Node Coupled Pendulum Solver Module

```

`timescale 1ns / 1ps
`include "constants.v"

module top_mod3(
    input clk
        // , input BRST
);

parameter gain = 0;

wire BRST;

reg [15:0] counter;
reg CLK_OUT;
initial begin
    counter = 0;
    CLK_OUT = 0;
end

always @(posedge clk) begin //frequency divider block
    counter = counter + 1;
    if (counter == 1000) begin
        counter = 0;
        CLK_OUT = ~CLK_OUT;
    end
end

end

wire signed [2*'WORDSIZE-1:0] packed_x1;
wire signed [2*'WORDSIZE-1:0] packed_x2;
wire signed [2*'WORDSIZE-1:0] packed_x3;
wire signed [2*'WORDSIZE-1:0] packed_x4;
wire signed [2*'WORDSIZE-1:0] packed_x5;
wire signed [2*'WORDSIZE-1:0] packed_x6;
wire signed [2*'WORDSIZE-1:0] packed_u1;
wire signed [2*'WORDSIZE-1:0] packed_u2;
wire signed [2*'WORDSIZE-1:0] packed_u3;
wire signed [2*'WORDSIZE-1:0] packed_u4;
wire signed [2*'WORDSIZE-1:0] packed_u5;
wire signed [2*'WORDSIZE-1:0] packed_u6;

//
wire signed ['WORDSIZE-1:0] abs_out12theta;

```



```

wire signed ['WORDSIZE-1:0] abs_out12thetaDot;
wire signed ['WORDSIZE-1:0] abs_out13theta;
wire signed ['WORDSIZE-1:0] abs_out13thetaDot;
wire signed ['WORDSIZE-1:0] abs_out14theta;
wire signed ['WORDSIZE-1:0] abs_out14thetaDot;
wire signed ['WORDSIZE-1:0] abs_out15theta;
wire signed ['WORDSIZE-1:0] abs_out15thetaDot;
wire signed ['WORDSIZE-1:0] abs_out16theta;
wire signed ['WORDSIZE-1:0] abs_out16thetaDot;

abs_val my_abs_val12theta (
    .in (~packed_x1['WORDSIZE-1:0]+1+packed_x2['WORDSIZE-1:0]),
    .out(abs_out12theta)
);
abs_val my_abs_val12thetaDot (
    .in (~packed_x1[2*['WORDSIZE-1:'WORDSIZE-1]+1+packed_x2[2*['WORDSIZE-1:'WORDSIZE-1]
        -1]),
    .out(abs_out12thetaDot)
);
abs_val my_abs_val13theta (
    .in (~packed_x1['WORDSIZE-1:0]+1+packed_x3['WORDSIZE-1:0]),
    .out(abs_out13theta)
);
abs_val my_abs_val13thetaDot (
    .in (~packed_x1[2*['WORDSIZE-1:'WORDSIZE-1]+1+packed_x3[2*['WORDSIZE-1:'WORDSIZE-1]
        -1]),
    .out(abs_out13thetaDot)
);
abs_val my_abs_val14theta (
    .in (~packed_x1['WORDSIZE-1:0]+1+packed_x4['WORDSIZE-1:0]),
    .out(abs_out14theta)
);
abs_val my_abs_val14thetaDot (
    .in (~packed_x1[2*['WORDSIZE-1:'WORDSIZE-1]+1+packed_x4[2*['WORDSIZE-1:'WORDSIZE-1]
        -1]),
    .out(abs_out14thetaDot)
);
abs_val my_abs_val15theta (
    .in (~packed_x1['WORDSIZE-1:0]+1+packed_x5['WORDSIZE-1:0]),
    .out(abs_out15theta)
);
abs_val my_abs_val15thetaDot (
    .in (~packed_x1[2*['WORDSIZE-1:'WORDSIZE-1]+1+packed_x5[2*['WORDSIZE-1:'WORDSIZE-1]
        -1]),
    .out(abs_out15thetaDot)
);
abs_val my_abs_val16theta (

```

```

        .in (~packed_x1['WORDSIZE-1:0']+1+packed_x6['WORDSIZE-1:0']),
        .out(abs_out16theta)
    );
abs_val my_abs_val16thetaDot (
    .in (~packed_x1[2*['WORDSIZE-1:WORDSIZE-1']+1+packed_x6[2*['WORDSIZE-1:WORDSIZE-1:0]'],
        -1]),
    .out(abs_out16thetaDot)
);

wire [35:0] CONTROL0;
wire [35:0] CONTROL1;
my_icon my_icon1 (
    .CONTROL0(CONTROL0), // INOUT BUS [35:0]
    .CONTROL1(CONTROL1)
);

my_ila my_ila1 (
    .CONTROL(CONTROL0), // INOUT BUS [35:0]
    .CLK(CLK_OUT), // IN
    .TRIG0(abs_out12theta+abs_out12thetaDot +
        abs_out13theta+abs_out13thetaDot +
        abs_out14theta+abs_out14thetaDot +
        abs_out15theta+abs_out15thetaDot +
        abs_out16theta+abs_out16thetaDot), // IN BUS [
        arrange_accordingly:0]
    .TRIG1(BRST) // IN BUS [0:0]
);

//

my_vio my_vio1 (
    .CONTROL(CONTROL1), // INOUT BUS [35:0]
    .ASYNC_OUT(BRST) // OUT BUS [0:0]
);

assign packed_u1['WORDSIZE-1:0] = 0;

assign packed_u1[2*['WORDSIZE-1:WORDSIZE] = 0;

assign packed_u2['WORDSIZE-1:0] = ~packed_x2['WORDSIZE-1:0] +1+
    packed_x1['WORDSIZE-1:0];
assign packed_u2[2*['WORDSIZE-1:WORDSIZE] = ~packed_x2[2*['WORDSIZE-1:WORDSIZE] +1+
    packed_x1[2*['WORDSIZE-1:WORDSIZE];
assign packed_u3['WORDSIZE-1:0] = ~packed_x3['WORDSIZE-1:0] +1+

```

```

packed_x1[ 'WORDSIZE-1:0];
assign packed_u3[2*'WORDSIZE-1':'WORDSIZE'] = ~packed_x3[2*'WORDSIZE-1':'WORDSIZE'] +1+
packed_x1[2*'WORDSIZE-1':'WORDSIZE'];
assign packed_u4[ 'WORDSIZE-1:0] = ~packed_x4[ 'WORDSIZE-1:0] +1+
packed_x1[ 'WORDSIZE-1:0];
assign packed_u4[2*'WORDSIZE-1':'WORDSIZE'] = ~packed_x4[2*'WORDSIZE-1':'WORDSIZE'] +1+
packed_x1[2*'WORDSIZE-1':'WORDSIZE'];
assign packed_u5[ 'WORDSIZE-1:0] = ~packed_x5[ 'WORDSIZE-1:0] +1+
packed_x1[ 'WORDSIZE-1:0];
assign packed_u5[2*'WORDSIZE-1':'WORDSIZE'] = ~packed_x5[2*'WORDSIZE-1':'WORDSIZE'] +1+
packed_x1[2*'WORDSIZE-1':'WORDSIZE'];
assign packed_u6[ 'WORDSIZE-1:0] = ~packed_x6[ 'WORDSIZE-1:0] +1+
packed_x1[ 'WORDSIZE-1:0];
assign packed_u6[2*'WORDSIZE-1':'WORDSIZE'] = ~packed_x6[2*'WORDSIZE-1':'WORDSIZE'] +1+
packed_x1[2*'WORDSIZE-1':'WORDSIZE'];

```

```

top_mod2 #('WORDSIZE' sb0011_1001_0010_0001_0010_0001 ,
          'WORDSIZE' sb0100_0100_0000_0010_0000_0000) top_mod2a

(
  .CLK_OUT(CLK_OUT) ,
  .BRST(BRST) ,
  .packed_u(packed_u1) ,
  .packed_x(packed_x1)
);

```

```

top_mod2 #('WORDSIZE' sb1111_0000_0010_0001_0000_0010 ,
          'WORDSIZE' sb0000_0000_0100_0000_0000_0001) top_mod2b

(
  .CLK_OUT(CLK_OUT) ,
  .BRST(BRST) ,
  .packed_u(packed_u2) ,
  .packed_x(packed_x2)
);

```

```

top_mod2 #('WORDSIZE' sb1111_0010_0010_0001_0000_0010 ,
          'WORDSIZE' sb0000_0000_0100_1000_0000_0001) top_mod2c

(
  .CLK_OUT(CLK_OUT) ,
  .BRST(BRST) ,
  .packed_u(packed_u3) ,
  .packed_x(packed_x3)
);

```

```

top_mod2 #('WORDSIZE' sb1111_0011_0010_0001_0000_0010 ,
          'WORDSIZE' sb0000_0000_0100_1010_0000_0001) top_mod2d

(
  .CLK_OUT(CLK_OUT) ,
  .BRST(BRST) ,
  .packed_u(packed_u4) ,
  .packed_x(packed_x4)
);

top_mod2 #('WORDSIZE' sb1111_0011_0010_0001_0100_0010 ,
          'WORDSIZE' sb0000_0000_0100_1010_0000_0001) top_mod2e

(
  .CLK_OUT(CLK_OUT) ,
  .BRST(BRST) ,
  .packed_u(packed_u5) ,
  .packed_x(packed_x5)
);

top_mod2 #('WORDSIZE' sb1111_0011_0010_0001_0100_0010 ,
          'WORDSIZE' sb0000_0000_0100_1010_0000_0001) top_mod2f

(
  .CLK_OUT(CLK_OUT) ,
  .BRST(BRST) ,
  .packed_u(packed_u6) ,
  .packed_x(packed_x6)
);

```

endmodule

A.8 MATLAB™ Code For Comparison

% this is to determine ODE behaviour of the implemented system

% function pendulumsolver_with_MATLAB_ODE_SOLVER

clear all; close all; clc;

final_time = 20;

f2 = @ (t, x) ...

```
[ x(2)          + 0;...  
  -sin(x(1)) + 0;...  
  x(4)          - x(3) + x(1);...  
  -sin(x(3)) - x(4) + x(2);...  
  ];
```

f3 = @ (t, x) ...

```
[ x(2)          + 0;...  
  -sin(x(1)) + 0;...  
  x(4)          - x(3) + x(1);...  
  -sin(x(3)) - x(4) + x(2);...  
  x(6)          - x(5) + x(1);...  
  -sin(x(5)) - x(6) + x(2);...  
  ];
```

f4 = @ (t, x) ...

```
[ x(2)          + 0;...  
  -sin(x(1)) + 0;...  
  x(4)          - x(3) + x(1);...  
  -sin(x(3)) - x(4) + x(2);...  
  x(6)          - x(5) + x(1);...  
  -sin(x(5)) - x(6) + x(2);...  
  x(8)          - x(7) + x(1);...  
  -sin(x(7)) - x(8) + x(2);...  
  ];
```

f5 = @ (t, x) ...

```
[ x(2)          + 0;...  
  -sin(x(1)) + 0;...  
  x(4)          - x(3) + x(1);...  
  -sin(x(3)) - x(4) + x(2);...  
  x(6)          - x(5) + x(1);...  
  -sin(x(5)) - x(6) + x(2);...  
  x(8)          - x(7) + x(1);...  
  -sin(x(7)) - x(8) + x(2);...  
  x(10)         - x(9) + x(1);...  
  -sin(x(9)) - x(10) + x(2);...  
  ];
```

f6 = @ (t, x) ...

```

[ x(2)          + 0;...
 -sin(x(1)) + 0;...
 x(4)          - x(3) + x(1);...
 -sin(x(3)) - x(4) + x(2);...
 x(6)          - x(5) + x(1);...
 -sin(x(5)) - x(6) + x(2);...
 x(8)          - x(7) + x(1);...
 -sin(x(7)) - x(8) + x(2);...
 x(10)         - x(9) + x(1);...
 -sin(x(9)) - x(10) + x(2);...
 x(12)         - x(11) + x(1);...
 -sin(x(11)) - x(12) + x(2);...
];

n = 100

% to deplete transient
[t1,y1] = ode45(f2,[0 final_time],rand(4,1));

fprintf(1,'\\hline\\\\n');
for i = 1:n,tic;[t2,y2] = ode45(f2,[0 single(final_time)],single(rand(4,1)));b2(i) =
toc;end
fprintf(1,'2_node_pc\\texttt{ode45}_FP32_bit_&_$.0f\\micro\\second$\\\\n',1e6*
mean(b2)/numel(t2));
for i = 1:n,tic;[t3,y3] = ode45(f3,[0 single(final_time)],single(rand(6,1)));b3(i) =
toc;end
fprintf(1,'3_node_pc\\texttt{ode45}_FP32_bit_&_$.0f\\micro\\second$\\\\n',1e6*
mean(b3)/numel(t3));
for i = 1:n,tic;[t4,y4] = ode45(f4,[0 single(final_time)],single(rand(8,1)));b4(i) =
toc;end
fprintf(1,'4_node_pc\\texttt{ode45}_FP32_bit_&_$.0f\\micro\\second$\\\\n',1e6*
mean(b4)/numel(t4));
for i = 1:n,tic;[t5,y5] = ode45(f5,[0 single(final_time)],single(rand(10,1)));b5(i) =
toc;end
fprintf(1,'5_node_pc\\texttt{ode45}_FP32_bit_&_$.0f\\micro\\second$\\\\n',1e6*
mean(b5)/numel(t5));
for i = 1:n,tic;[t6,y6] = ode45(f6,[0 single(final_time)],single(rand(12,1)));b6(i) =
toc;end
fprintf(1,'6_node_pc\\texttt{ode45}_FP32_bit_&_$.0f\\micro\\second$\\\\n',1e6*
mean(b6)/numel(t6));

[t1,y1] = ode23(f2,[0 final_time],rand(4,1));

fprintf(1,'\\hline\\\\n');
for i = 1:n,tic;[t2,y2] = ode23(f2,[0 single(final_time)],single(rand(4,1)));b2(i) =
toc;end

```

```

fprintf(1,'2_node_pc\\texttt{ode23}_FP32_bit_&_$.0f\\micro\\second$\\\\\\n',1e6*
    mean(b2)/numel(t2));
for i = 1:n, tic;[t3,y3] = ode23(f3,[0 single(final_time)],single(rand(6,1)));b3(i) =
    toc;end
fprintf(1,'3_node_pc\\texttt{ode23}_FP32_bit_&_$.0f\\micro\\second$\\\\\\n',1e6*
    mean(b3)/numel(t3));
for i = 1:n, tic;[t4,y4] = ode23(f4,[0 single(final_time)],single(rand(8,1)));b4(i) =
    toc;end
fprintf(1,'4_node_pc\\texttt{ode23}_FP32_bit_&_$.0f\\micro\\second$\\\\\\n',1e6*
    mean(b4)/numel(t4));
for i = 1:n, tic;[t5,y5] = ode23(f5,[0 single(final_time)],single(rand(10,1)));b5(i) =
    toc;end
fprintf(1,'5_node_pc\\texttt{ode23}_FP32_bit_&_$.0f\\micro\\second$\\\\\\n',1e6*
    mean(b5)/numel(t5));
for i = 1:n, tic;[t6,y6] = ode23(f6,[0 single(final_time)],single(rand(12,1)));b6(i) =
    toc;end
fprintf(1,'6_node_pc\\texttt{ode23}_FP32_bit_&_$.0f\\micro\\second$\\\\\\n',1e6*
    mean(b6)/numel(t6));
fprintf(1,'\\hline\\\\\\n');

```

A.9 C Code For Comparison IEEE754-32

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <unistd.h>
#define CLOCK_PROCESS_CPU_TIME_ID 2
#define CLOCK_THREAD_CPU_TIME_ID 3
#define N 12
#define ITER 20000
#define ITER2 2000

int main () {

    time_t t;
    struct timespec t1, t2;

    double total_elapsed_time;
    double avr_elapsed_time;
    int i, j;

    float dt;
    float xn[N], x[N];

    srand((unsigned) time(&t));

    dt = pow(2, -11);

    for (i = 0; i < N; i++) x[i] = (rand() % 4) - 2;

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

    for (j = 0; j < ITER2; j++) {

        // clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

        for (i = 0; i < ITER; i++) {
            xn[0] = x[0] + ( x[1] ) * dt;
            xn[1] = x[1] + ( -sin(x[0]) ) * dt;
            xn[2] = x[2] + ( x[3] + x[0] - x[2] ) * dt;
            xn[3] = x[3] + ( -sin(x[2]) + x[1] - x[3] ) * dt;

            x[0] = xn[0];
            x[1] = xn[1];
            x[2] = xn[2];
            x[3] = xn[3];
        }
    }
}
```



```

}
}
clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

//avr_elapsed_time = 0;
//for (j=0;j<ITER2;j++) {
//avr_elapsed_time += elapsed_time[j];
//}
total_elapsed_time = t2.tv_sec*1e9 - t1.tv_sec*1e9 +
                    t2.tv_nsec - t1.tv_nsec;
printf("Total_Elapsed_Time: %.0f\n", total_elapsed_time);
avr_elapsed_time = total_elapsed_time / ITER / ITER2;
printf("2node_Explicite_Euler_FP32_& %.0f\\nano\\second\\\\\\n",
        avr_elapsed_time);

for (i = 0; i < N; i++) x[i]=(rand() % 4)-2;

clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

for (j = 0; j < ITER2; j++) {

//clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

for (i = 0; i < ITER; i++) {
    xn[0] = x[0] + ( x[1] )dt;
    xn[1] = x[1] + ( -sin(x[0]) )dt;
    xn[2] = x[2] + ( x[3] + x[0] - x[2])dt;
    xn[3] = x[3] + ( -sin(x[2]) + x[1] - x[3])dt;
    xn[4] = x[4] + ( x[5] + x[0] - x[4])dt;
    xn[5] = x[5] + ( -sin(x[4]) + x[1] - x[5])dt;

    x[0] = xn[0];
    x[1] = xn[1];
    x[2] = xn[2];
    x[3] = xn[3];
    x[4] = xn[4];
    x[5] = xn[5];
}
}
clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

total_elapsed_time = t2.tv_sec*1e9 - t1.tv_sec*1e9 +
                    t2.tv_nsec - t1.tv_nsec;
printf("Total_Elapsed_Time: %.0f\n", total_elapsed_time);

```

```

avr_elapsed_time = total_elapsed_time / ITER / ITER2;
printf("3_node_Explicite_Euler_FP32_&_%.0f\\nano\\second\\\\\\n",
      avr_elapsed_time);

for (i = 0; i < N; i++) x[i]=(rand() % 4)-2;

clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

for (j = 0; j < ITER2; j++) {

for (i = 0; i < ITER; i++) {
    xn[0] = x[0] + ( x[1] )dt;
    xn[1] = x[1] + ( -sin(x[0]) )dt;
    xn[2] = x[2] + ( x[3] + x[0] - x[2] )dt;
    xn[3] = x[3] + ( -sin(x[2]) + x[1] - x[3] )dt;
    xn[4] = x[4] + ( x[5] + x[0] - x[4] )dt;
    xn[5] = x[5] + ( -sin(x[4]) + x[1] - x[5] )dt;
    xn[6] = x[6] + ( x[7] + x[0] - x[6] )dt;
    xn[7] = x[7] + ( -sin(x[6]) + x[1] - x[7] )dt;

    x[0] = xn[0];
    x[1] = xn[1];
    x[2] = xn[2];
    x[3] = xn[3];
    x[4] = xn[4];
    x[5] = xn[5];
    x[6] = xn[6];
    x[7] = xn[7];
}
}
clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

total_elapsed_time = t2.tv_sec1e9 - t1.tv_sec1e9 +
                    t2.tv_nsec - t1.tv_nsec;
printf("Total_Elapsed_Time: %.0f\\n", total_elapsed_time);
avr_elapsed_time = total_elapsed_time / ITER / ITER2;
printf("4_node_Explicite_Euler_FP32_&_%.0f\\nano\\second\\\\\\n",
      avr_elapsed_time);

for (i = 0; i < N; i++) x[i]=(rand() % 4)-2;

clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

```

```

for (j = 0; j < ITER2; j++) {

for (i = 0; i < ITER; i++) {
    xn[0] = x[0] + ( x[1] )dt;
    xn[1] = x[1] + ( -sin(x[0]) )dt;
    xn[2] = x[2] + ( x[3] + x[0] - x[2])dt;
    xn[3] = x[3] + ( -sin(x[2]) + x[1] - x[3])dt;
    xn[4] = x[4] + ( x[5] + x[0] - x[4])dt;
    xn[5] = x[5] + ( -sin(x[4]) + x[1] - x[5])dt;
    xn[6] = x[6] + ( x[7] + x[0] - x[6])dt;
    xn[7] = x[7] + ( -sin(x[6]) + x[1] - x[7])dt;
    xn[8] = x[8] + ( x[9] + x[0] - x[8])dt;
    xn[9] = x[9] + ( -sin(x[8]) + x[1] - x[9])dt;

    x[0] = xn[0];
    x[1] = xn[1];
    x[2] = xn[2];
    x[3] = xn[3];
    x[4] = xn[4];
    x[5] = xn[5];
    x[6] = xn[6];
    x[7] = xn[7];
    x[8] = xn[8];
    x[9] = xn[9];
}
}

clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

total_elapsed_time = t2.tv_sec*1e9 - t1.tv_sec*1e9 +
                    t2.tv_nsec - t1.tv_nsec;
printf("Total_Elapsed_Time: %.0f\n", total_elapsed_time);
avr_elapsed_time = total_elapsed_time / ITER / ITER2;
printf("5_node_Explicite_Euler_FP32_& %.0f\\nano\\second\\\\\\n",
        avr_elapsed_time);


for (i = 0; i < N; i++) x[i]=(rand() % 4)-2;

clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

for (j = 0; j < ITER2; j++) {

for (i = 0; i < ITER; i++) {
    xn[0] = x[0] + ( x[1] )dt;

```

```

        xn[1] = x[1] + ( -sin(x[0])
                                ) * dt;
        xn[2] = x[2] + ( x[3] +
                                x[0] - x[2]) * dt;
        xn[3] = x[3] + ( -sin(x[2]) +
                                x[1] - x[3]) * dt;
        xn[4] = x[4] + ( x[5] +
                                x[0] - x[4]) * dt;
        xn[5] = x[5] + ( -sin(x[4]) +
                                x[1] - x[5]) * dt;
        xn[6] = x[6] + ( x[7] +
                                x[0] - x[6]) * dt;
        xn[7] = x[7] + ( -sin(x[6]) +
                                x[1] - x[7]) * dt;
        xn[8] = x[8] + ( x[9] +
                                x[0] - x[8]) * dt;
        xn[9] = x[9] + ( -sin(x[8]) +
                                x[1] - x[9]) * dt;
        xn[10]=x[10] + ( x[11] +
                                x[0] - x[10]) * dt;
        xn[11]=x[11] + ( -sin(x[10]) +
                                x[1] - x[11]) * dt;

        x[0] = xn[0];
        x[1] = xn[1];
        x[2] = xn[2];
        x[3] = xn[3];
        x[4] = xn[4];
        x[5] = xn[5];
        x[6] = xn[6];
        x[7] = xn[7];
        x[8] = xn[8];
        x[9] = xn[9];
        x[10] = xn[10];
        x[11] = xn[11];
    }
}

clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

total_elapsed_time = t2.tv_sec * 1e9 - t1.tv_sec * 1e9 +
                    t2.tv_nsec - t1.tv_nsec;
printf("Total Elapsed Time: %.0f\n", total_elapsed_time);
avr_elapsed_time = total_elapsed_time / ITER / ITER2;
printf("6node Explicite Euler FP32 & %.0f \\ nano \\ second \\ \\ \\ \\n",
        avr_elapsed_time);
return 0;
}

```

A.10 C Code For Comparison $I = 2, F = 29$ Signed Fixed Point

```

#include <stdio.h>
#include <time.h>
#include <unistd.h>
#define N 12
#define CLOCK_THREAD_CPU_TIME_ID 3
#define ITER 1000
#define ITER2 10000

int atanTable [30] =
{
    0x1921FB54, 0x0ED63383, 0x07D6DD7E, 0x03FAB753,
    0x01FF55BB, 0x00FFEAAE, 0x007FFD55, 0x003FFAB,
    0x001FFFF5, 0x000FFFFF, 0x00080000, 0x00040000,
    0x00020000, 0x00010000, 0x00008000, 0x00004000,
    0x00002000, 0x00001000, 0x00000800, 0x00000400,
    0x00000200, 0x00000100, 0x00000080, 0x00000040,
    0x00000020, 0x00000010, 0x00000008, 0x00000004,
    0x00000002, 0x00000001
};

int scaleFactor = 0x1B7B2B63;

int main() {
    time_t t;
    struct timespec t1, t2;
    int xn[N], x[N];
    double total_elapsed_time;
    double avr_elapsed_time;

    int i, j;
    int dt = 0x00040000; //  $2^{(-11)}$ 

    srand( time(NULL) );

    for (i = 0; i < 12; i++) x[i] = rand()%4 - 2;

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

    for (i = 0; i < ITER; i++)
        for (j = 0; j < ITER2; j++) {
            xn[0] = x[0] + (x[1]
                                )□dt;
            xn[1] = x[1] + (-cordicSin(x[0])
                                )□dt;
            xn[2] = x[2] + (x[3]
                                + x[2] - x[0]
                                )□dt;
            xn[3] = x[3] + (-cordicSin(x[2])
                                + x[3] - x[1]
                                )□dt;
            x[0] = xn[0];

```

```

        x[1] = xn[1];
        x[2] = xn[2];
        x[3] = xn[3];
    }

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

    total_elapsed_time = t2.tv_sec*1e9 - t1.tv_sec*1e9 +
                        t2.tv_nsec - t1.tv_nsec;
    printf("Total Elapsed Time %.0f\n", total_elapsed_time);
    avr_elapsed_time = total_elapsed_time / ITER / ITER2;
    printf("2node Explicite Euler $I=2$, $F=29$ bit signed Fixed Numbers & %.0f
           \\ nano \\ second \\ \\ \\n", avr_elapsed_time);

    for (i = 0; i < 12; i++) x[i] = rand()%4 - 2;

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

    for (i = 0; i < ITER; i++)
        for (j = 0; j < ITER2; j++) {
            xn[0] = x[0] + (x[1] )dt;
            xn[1] = x[1] + (-cordicSin(x[0]) )dt;
            xn[2] = x[2] + (x[3] + x[2] - x[0] )dt;
            xn[3] = x[3] + (-cordicSin(x[2]) + x[3] - x[1] )dt;
            xn[4] = x[4] + (x[5] + x[4] - x[0] )dt;
            xn[5] = x[5] + (-cordicSin(x[4]) + x[5] - x[1] )dt;
            x[0] = xn[0];
            x[1] = xn[1];
            x[2] = xn[2];
            x[3] = xn[3];
            x[4] = xn[4];
            x[5] = xn[5];
        }

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

    total_elapsed_time = t2.tv_sec*1e9 - t1.tv_sec*1e9 +
                        t2.tv_nsec - t1.tv_nsec;
    printf("Total Elapsed Time %.0f\n", total_elapsed_time);
    avr_elapsed_time = total_elapsed_time / ITER / ITER2;
    printf("3node Explicite Euler $I=2$, $F=29$ bit signed Fixed Numbers & %.0f
           \\ nano \\ second \\ \\ \\n", avr_elapsed_time);

    for (i = 0; i < 12; i++) x[i] = rand()%4 - 2;

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

```

```

for (i = 0; i < ITER; i++)
    for (j = 0; j < ITER2; j++) {
        xn[0] = x[0] + (x[1] )□dt;
        xn[1] = x[1] + (−cordicSin(x[0]) )□dt;
        xn[2] = x[2] + (x[3] + x[2] − x[0] )□dt;
        xn[3] = x[3] + (−cordicSin(x[2]) + x[3] − x[1] )□dt;
        xn[4] = x[4] + (x[5] + x[4] − x[0] )□dt;
        xn[5] = x[5] + (−cordicSin(x[4]) + x[5] − x[1] )□dt;
        xn[6] = x[6] + (x[7] + x[6] − x[0] )□dt;
        xn[7] = x[7] + (−cordicSin(x[6]) + x[7] − x[1] )□dt;
        x[0] = xn[0];
        x[1] = xn[1];
        x[2] = xn[2];
        x[3] = xn[3];
        x[4] = xn[4];
        x[5] = xn[5];
        x[6] = xn[6];
        x[7] = xn[7];
    }

clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

total_elapsed_time = t2.tv_sec□1e9 − t1.tv_sec□1e9 +
                    t2.tv_nsec − t1.tv_nsec;
printf("Total□Elapsed□Time□%.0f\n", total_elapsed_time);
avr_elapsed_time = total_elapsed_time / ITER / ITER2;
printf("4□node□Explicite□Euler□I=2$,□F=29$□bit□signed□Fixed□Numbers□&□%.0f
      \\nano\\second\\\\\\n", avr_elapsed_time);

for (i = 0; i < 12; i++) x[i] = rand()%4 − 2;

clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

for (i = 0; i < ITER; i++)
    for (j = 0; j < ITER2; j++) {
        xn[0] = x[0] + (x[1] )□dt;
        xn[1] = x[1] + (−cordicSin(x[0]) )□dt;
        xn[2] = x[2] + (x[3] + x[2] − x[0] )□dt;
        xn[3] = x[3] + (−cordicSin(x[2]) + x[3] − x[1] )□dt;
        xn[4] = x[4] + (x[5] + x[4] − x[0] )□dt;
        xn[5] = x[5] + (−cordicSin(x[4]) + x[5] − x[1] )□dt;
        xn[6] = x[6] + (x[7] + x[6] − x[0] )□dt;
        xn[7] = x[7] + (−cordicSin(x[6]) + x[7] − x[1] )□dt;
        xn[8] = x[8] + (x[9] + x[8] − x[0] )□dt;
        xn[9] = x[9] + (−cordicSin(x[8]) + x[9] − x[1] )□dt;
        x[0] = xn[0];
        x[1] = xn[1];

```

```

        x[2] = xn[2];
        x[3] = xn[3];
        x[4] = xn[4];
        x[5] = xn[5];
        x[6] = xn[6];
        x[7] = xn[7];
        x[8] = xn[8];
        x[9] = xn[9];
    }

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

    total_elapsed_time = t2.tv_sec*1e9 - t1.tv_sec*1e9 +
                        t2.tv_nsec - t1.tv_nsec;
    printf("Total_Elapsed_Time_%.0f\n", total_elapsed_time);
    avr_elapsed_time = total_elapsed_time / ITER / ITER2;
    printf("5_node_Explicite_Euler_I=2$,F=29$bit_signed_Fixed_Numbers_&_.0f\n\nano\\second\\\\\\n", avr_elapsed_time);

    for (i = 0; i < 12; i++) x[i] = rand()%4 - 2;

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t1);

    for (i = 0; i < ITER; i++)
        for (j = 0; j < ITER2; j++) {
            xn[0] = x[0] + (x[1] )dt;
            xn[1] = x[1] + (-cordicSin(x[0]) )dt;
            xn[2] = x[2] + (x[3] + x[2] - x[0] )dt;
            xn[3] = x[3] + (-cordicSin(x[2]) + x[3] - x[1] )dt;
            xn[4] = x[4] + (x[5] + x[4] - x[0] )dt;
            xn[5] = x[5] + (-cordicSin(x[4]) + x[5] - x[1] )dt;
            xn[6] = x[6] + (x[7] + x[6] - x[0] )dt;
            xn[7] = x[7] + (-cordicSin(x[6]) + x[7] - x[1] )dt;
            xn[8] = x[8] + (x[9] + x[8] - x[0] )dt;
            xn[9] = x[9] + (-cordicSin(x[8]) + x[9] - x[1] )dt;
            xn[10] = x[10] + (x[11] + x[10] - x[0] )dt;
            xn[11] = x[11] + (-cordicSin(x[10]) + x[11] - x[1] )dt;
            x[0] = xn[0];
            x[1] = xn[1];
            x[2] = xn[2];
            x[3] = xn[3];
            x[4] = xn[4];
            x[5] = xn[5];
            x[6] = xn[6];
            x[7] = xn[7];
            x[8] = xn[8];
            x[9] = xn[9];
        }

```



```

        x[10] = xn[10];
        x[11] = xn[11];
    }

    clock_gettime(CLOCK_THREAD_CPU_TIME_ID, &t2);

    total_elapsed_time = t2.tv_sec*1e9 - t1.tv_sec*1e9 +
                        t2.tv_nsec - t1.tv_nsec;
    printf("Total_Elapsed_Time_%.0f\n", total_elapsed_time);
    avr_elapsed_time = total_elapsed_time / ITER / ITER2;
    printf("6_node_Explicite_Euler_I=2$,F=29$bit_signed_Fixed_Numbers_&_%.0f\n\n\nsecond\\second\\", avr_elapsed_time);

    return 0;
}

int cordicSin (int z) {
    int xSum = 0;
    int ySum = (z>0)? 0x20000000:0xE0000000;
    int zSum = (z>0)? 0x3243F6A9:0xCDBC0957;

    int i;

    for (i = 0; i < 22; i++) {

        xSum = (z < zSum) ? xSum + ySum >> i : xSum - ySum >> i;
        ySum = (z < zSum) ? ySum - xSum >> i : ySum + xSum >> i;
        zSum = (z < zSum) ? zSum - atanTable[i]:zSum + atanTable[i];

    }
    ySum /= scaleFactor;
    return ySum; //return sin z;
}

```