TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

LOGARİTMİK VE ÜSTEL FONKSİYON HESAPLAMALARI İÇİN BÜTÜNLEŞİK YENİLİKÇİ DONANIM MİMARİSİ GELİŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Gülfem HELVACIOĞLU

Elektrik ve Elektronik Mühendisliği Anabilim Dalı

Tez Danışmanı: Doç. Dr. Coşku KASNAKOĞLU

NİSAN 2017

Fen Bilimleri Enstitüsü Onayı

……..
Prof. Dr. Osman EROĞUL
Müdür

Bu tezin Yüksek Lisans/Doktora derecesinin tüm gereksinimlerini sağladığını onaylarım.

……
Doç. Dr. Tolga GİRİCİ
Anabilimdalı Başkanı

TOBB ETÜ, Fen Bilimleri Enstitüsü'nün 151211027 numaralı Yüksek Lisans Öğrencisi **Gülfem HELVACIOĞLU**'nun ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı **"LOGARİTMİK VE ÜSTEL FONKSİYON HESAPLAMALARI İÇİN BÜTÜNLEŞİK YENİLİKÇİ DONANIM MİMARİSİ GELİŞTİRİLMESİ"** başlıklı tezi **10 Nisan 2017** tarihinde aşağıda imzaları olan jüri tarafından kabul edilmiştir.

Tez Danışmanı : **Doç. Dr. Coşku KASNAKOĞLU** .....
TOBB Ekonomi ve Teknoloji Üniversitesi

Jüri Üyeleri : **Doç.Dr.A. Sanlı ERGÜN(Başkan)** .....
TOBB Ekonomi ve Teknoloji Üniversitesi

Yrd.Doç. Dr. D.Sinan KÖRPE .....
THK Üniversitesi

TEZ BİLDİRİMİ

Tez içindeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edilerek sunulduğunu, alıntı yapılan kaynaklara eksiksiz atıf yapıldığını, referansların tam olarak belirtildiğini ve ayrıca bu tezin TOBB ETÜ Fen Bilimleri Enstitüsü tez yazım kurallarına uygun olarak hazırlandığını bildiririm.

. Gülfem HELVACIOĞLU

ÖZET

Yüksek Lisans Tezi

LOGARİTMİK VE ÜSTEL FONKSİYON HESAPLAMALARI İÇİN
BÜTÜNLEŞİK YENİLİKÇİ DONANIM MİMARİSİ GELİŞTİRİLMESİ

Gülfem HELVACIOĞLU

TOBB Ekonomi ve Teknoloji Üniveritesi
Fen Bilimleri Enstitüsü
Elektrik ve Elektronik Mühendisliği Anabilim Dalı

Danışman: Doç. Dr. Coşku KASNAKOĞLU

Tarih: Nisan 2017

Bu çalışmada, belirli bir tabanda logaritma hesaplaması yapan İndirgemeli CORDIC Tabanlı Logaritma Çeviricisi (İCTLÇ) metodu tanıtılmış ve İCTLÇ'nin FPGA(Field-Programmable-Gate-Array) tabanlı donanım mimarisi ayrıntıları ile anlatılmıştır. İCTLÇ çok düşük kaynak kullanımı ile yüksek bit hassasiyetinde logaritma hesaplarını karmaşık olmayan bir donanım mimarisi ile yapabilmektedir. Toplama ve bit kaydırma işlemlerini barındıran CORDIC metodunun yanında hızlı bir sayı indirgeme yöntemi kullanarak çok geniş aralıktaki sayıların logaritmik çevrimini diğer metotlara kıyasla daha yüksek hassasiyetle ve az kaynak kullanımı ile mümkün kılmaktadır. İçerisinde bulundurduğu indirgeme metodu ile de aynı mimari ile geniş bir operasyon değer aralığı sunmaktadır.

Anahtar Kelimeler: Logaritma, FPGA, CORDIC, Sinyal işleme

ABSTRACT

Master of Science

DEVOLOPING A NOVEL INTEGRATED HARDWARE ARCHITECTURE FOR CALCULATION OF LOGARITHMIC AND EXPONANTIONAL FUNCTIONS

Gülfem HELVACIOĞLU

TOBB University of Economics and Technology
Institute of Natural and Applied Sciences
Electrics and Electronics Engineering Science Programme

Supervisor: Asoc. Prof. Coşku KASNAKOĞLU

Date: April 2017

In this work, a novel method Reduced CORDIC Based Logarithm Converter (RCBLC) is introduced for computing the specific-based logarithm of the binary values, and then the hardware architecture of RCBLC based on FPGA is analyzed in detail. Hardware architecture of RCBLC is implemented such that it enables logarithm conversion with both high output bit-sensitivity and low resource utilization. In addition to CORDIC method which includes only add-and-shift operations, using the rapid value reduction method provides an opportunity for calculating logarithm of a value more precise and lower resource utilization comparing to other methods. Thanks to the reduction method in it, RCBLC can provide wide operating input interval for logarithm conversion.

Keywords: Logarithm, FPGA, CORDIC, Signal processing

TEŞEKKÜR

Çalışmalarım boyunca değerli yardım ve katkılarıyla beni yönlendiren hocam Coşku KASNAKOĞLU'na, kıymetli tecrübelerinden faydalandığım TOBB Ekonomi ve Teknoloji Üniversitesi Elektrik Elektronik Mühendisliği Bölümü öğretim üyelerine,burs sağladığı için TOBB Ekonomi ve Teknoloji Üniversitesi ve TÜBİTAK'a ve manevi desteklerinden ötürü ailem ve Ali Buğra KORUCU'ya sonsuz teşekkürlerimi sunarım.

İÇİNDEKİLER

Sayfa

ÖZET ..... **iv**
ABSTRACT ..... **v**
TEŞEKKÜR ..... **vi**
İÇİNDEKİLER ..... **vii**
ŞEKİL LİSTESİ ..... **viii**
ÇİZELGE LİSTESİ ..... **ix**
KISALTMALAR ..... **x**
1. GİRİŞ **1**
1.1 Logaritma Hesabının Kullanıldığı Alanlar 1
1.2 Logaritma Hesabının Kullanıldığı Geçmiş Yöntemler 1
2. YÖNTEMLER **2**
2.1 Mitchell Yakınsaması 2
2.1.1 Mitchell yakınsaması tabanlı kısmi doğrusallaştırma yöntemi 5
2.1.2 Mitchell yakınsaması tabanlı ağırlıklı enterpolasyon 6
2.2 Mercator Serileri 8
2.3 CORDIC Algoritması 9
3. GELİŞTİRİLEN LOGARİTMA ÇEVİRİCİSİ İCTLÇ METODU **13**
3.1 İCTLC'ye Genel Bakış 14
3.2 FPGA Mimarisi 17
3.3 Donanım ve Benzetim Çalışmaları 21
4. ANALİZ VE KIYASLAMA **29**
5. SONUÇ **34**
KAYNAKLAR **35**
EKLER **36**
ÖZGEÇMİŞ **45**

ŞEKİL LİSTESİ

Sayfa

Şekil 2.1 : Mitchell yakınsaması ve logaritma gerçek değer ilişkisi 3
Şekil 2.2 : Mitchell yakınsama hata miktarı 4
Şekil 2.3 : Mitchell yakınsama hata oranı değerleri 4
Şekil 2.4 : Kısmi bölge sayısını belirten parametrenin maksimum hataya etkisi 5
Şekil 2.5 : Metotlar arasındaki fark 6
Şekil 2.6 : Ağırlıklı enterpolasyonun hafıza elemanları üzerinden yapılışı 7
Şekil 2.7 : Farklı derecelere göre mercator serisinin sonuçları 8
Şekil 2.8 : Mercator serilerinin dereceye göre hata oranları 9
Şekil 2.9: CORDIC algoritması rotasyon modu çalışma şekli 10
Şekil 2.10 : CORDIC algoritması vektör modu çalışma şekli 11
Şekil 3.1 : CORDIC algoritması çalışma aralığı 15
Şekil 3.2 : İCTLÇ blok diyagramı 17
Şekil 3.3 : "n" sayısını bulmanın RTL şeması 18
Şekil 3.4 : ÇAB bloğunun RTL donanım şeması 19
Şekil 3.5 : CORDIC algoritması mimarisi 20
Şekil 3.6 : İCTLÇ RTL şeması 21
Şekil 3.7 : 8-bit aritmetikle yapılan benzetim sonucu. 24
Şekil 3.8 : 8-bit aritmetikte yapılan 120 değeri için benzetim sonucu 26
Şekil 3.9: 3 değeri için 16 bit aritmetikte yapılan benzetim sonucu 27
Şekil 3.10: 11018 değeri 16-bit aritmetikte yapılan için benzetim sonucu 29
Şekil 4.1: 8-bitlik aritmetikte 5 yinelemeli sonuç değerleri 30
Şekil 4.2: İCTLÇ için 16-bit aritmetikte 10 yineleme için hata miktarları 31

ÇİZELGE LİSTESİ

Sayfa

Çizelge 4.1 : İCTLÇ için kaynak ve hız değerleri 32
Çizelge 4.2 : İCTLÇ için doğruluk ve hassasiyet 33

KISALTMALAR

AİB : Aralık İndirgeme Bloğu
CORDIC : Coordinate Rotation Digital Converter
ÇAB : Çözünürlük Artırma Bloğu
DSP : Digital Signal Processor
FF : Flip Flop
FPGA : Field Programmable Gate Array
GÇAB : Giriş Çözünürlük Artırma Bloğu
İCTLÇ : İndirgenmiş CORDIC Tabanlı Logaritma Çeviricisi
İCTLK : İndirgenmiş CORDIC Tabanlı Logaritma Kestirimi
LUT : Taramalı Tablo (Look Up Table)
RAM : Rastgele Erişim Belleği (Random Access Memory)
RMS : Root Mean Square

1.GİRİŞ

1.1 Logaritma Hesabının Kullanıldığı Alanlar

Belirli bir tabanda logaritma hesaplanmasına günümüzdeki teknolojik uygulamalarda ciddi bir talep vardır. Bu teknolojik uygulamalarından bazılarını sayısal işaret işleme, sayısal görüntü işleme, radar işaret işleme, dijital kontrol uygulamaları ve sayısal haberleşme vb. şeklinde sayabiliriz. Gerçek zamanlı radar uygulamalarında genlik farkları çok yüksek iki farklı işaretin aynı göstergede gösterilmesi için logaritmik ölçek kullanılmaktadır [1]. Böylelikle genlikleri birbirinden farklı iki sinyal logaritmik y ekseninde aynı ekranda gösterilebilmekte ve ekran üzerinde kolaylıkla ayırt edilebilmektedir. Logaritmanın kullanıldığı diğer bir alan sayısal haberleşmedir. Sayısal haberleşmede çözücü(decoder) öncesindeki bit karar verici blok gelen sinyallerin logaritmik değerleri üzerinden hesap yapar ve kestirim sonucunu bu hesaba göre neticelendirir. Bir başka elektroniğin alanı otopilot kontrol uygulamalarında gerçek zamanlı çalışabilen hızlı gerçeklemeler yapmak için FPGA veya ASIC mimarileri üzerinde logaritma çevrimleri yapılabilmektedir [2].

1.2 Logaritma Hesaplamada Kullanılan Geçmiş Yöntemler

Belirli tabanda logaritma hesabı yapabilmek için geliştirilen algoritmalar sıklıkla üç yöntem üzerinde yoğunlaşmıştır. Birinci yöntemde kullanıcı tarafından önceden oluşturulmuş taramalı tablolarda (LUT:Look-Up-Table) belirli kesinlikte logaritma değerleri tutulur. Bu değerler kullanılan metodun aşamalarında gerekli olan sayısal değerlerdir. Metot, girdiye göre taramalı LUT'tan uygun değerleri çeker ve çıktıyı bu tablodan çektiği değerlere göre belirler [4],[5]. LUT üzerine kurulmuş bu yöntemler sıklıkla Mitchell yakınsamasına başvurur[3]. [4]'de LUT, parçalı doğrusal kestirim yapabilmek için gerekli olan katsayıları barındırır ve Mitchell yakınsaması ile birleştirir. [5]'de ise Mitchell kestirimi ile girdinin iki tabanında logaritma gerçek değeri arasındaki farkı bir LUT'ta tutar ve aradaki farkı Mitchell kestirimine ekler.

Başvurulan ikinci yöntem polinom tabanlı yöntemdir. Bu yöntemde logaritma fonksiyonun Taylor serisi açılımı kullanılır. Belirli alanda Taylor serisine yakınsayan

logaritma fonksiyonu yine bu alan içerisinde bir polinom fonksiyonuna çevrilerek polinom hesaplaması ile değeri hesaplanır. Taylor serisi açılımı yapar iken sonuç keskinliğini arttırmak için genellikle 3 veya daha yüksek derecede Taylor serisi açılımları kullanılır. Bu metot çarpma işlemleri üzerine temellendiği için yüksek çarpıcı elemanına ihtiyaç duyar.

Logaritma kestirimi için başvurulan üçüncü yöntem ise CORDIC(Coordinate Rotation Digital Computer) tabanlı yinelemeli hesaba dayanmaktadır. Temelinde iki farklı modda kullanılan CORDIC algoritması ters trigonometrik ve ters hiperbolik fonksiyonların hesaplanması için kullanılır. Düzlemsel iki boyutlu koordinat ekseninde herhangi bir bölgedeki bir vektörün x ekseni $(y = 0)$ ile yaptığı açının değerini bulmaya yarayan bu metot aynı zamanda hiperbolik fonksiyonlara da kolaylıkla genişletilebilmektedir. Logaritma çevrimi için ise, bu yöntem ters hiperbolik fonksiyonlardan logaritmik fonksiyonlara geçiş özelliğini kullanır[6-8]. Yinelemeli hesap üzerine kurulu olduğu için adım sayısı önceden belirlenir ve metodun çıktısının hassaslığı adım sayısı ve operasyon bit genişliği ile doğru orantılıdır. CORDIC algoritması yinelemeler sonucunda hesaplanması istenen fonksiyonun (ters trigonometrik veya hiperbolik) sonucuna yakınsar veya tam karşılığını verir. Yineleme adımı sonsuza götürüldüğünde ise hatasız bir şekilde istenilen fonksiyon çıktısını bulmuş olur.

Önerilen İndirgenmiş CORDIC Tabanlı Logaritma Çeviricisi (İCTLÇ), CORDIC tabanlı yüksek hassasiyetli bir logaritma kestiricisidir. Girdi olarak verilen herhangi bir sayısal değerin belirlenen bir tabanda logaritma değerini hesaplar ve çıktı olarak sunar. Makale planına göre ikinci bölümde logaritma kestirimi üzerine yapılmış önceki çalışmalar değerlendirilecektir. Üçüncü bölümde İCTLK yaklaşımı anlatılacaktır. Dördüncü bölümde İCTLÇ'nın hata oranları ve performans ölçümleri gösterilecek, diğer metotlarla performans ve kaynak kullanım kıyaslaması yapılacaktır. Son bölümde İCTLÇ'nın bütün bir değerlendirmesi yapılacaktır.

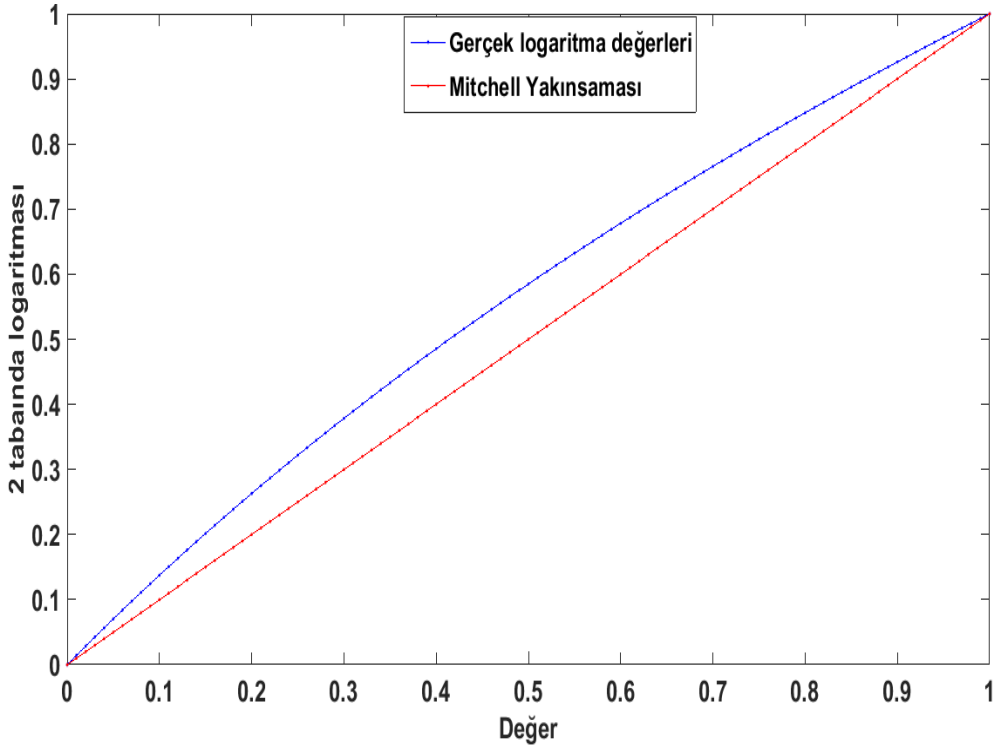
2. YÖNTEMLER

2.1 Mitchell Yakınsaması

1967 yılında Mitchell ve Jean-Charles tarafından geliştirilen logaritma çevrim yöntemi

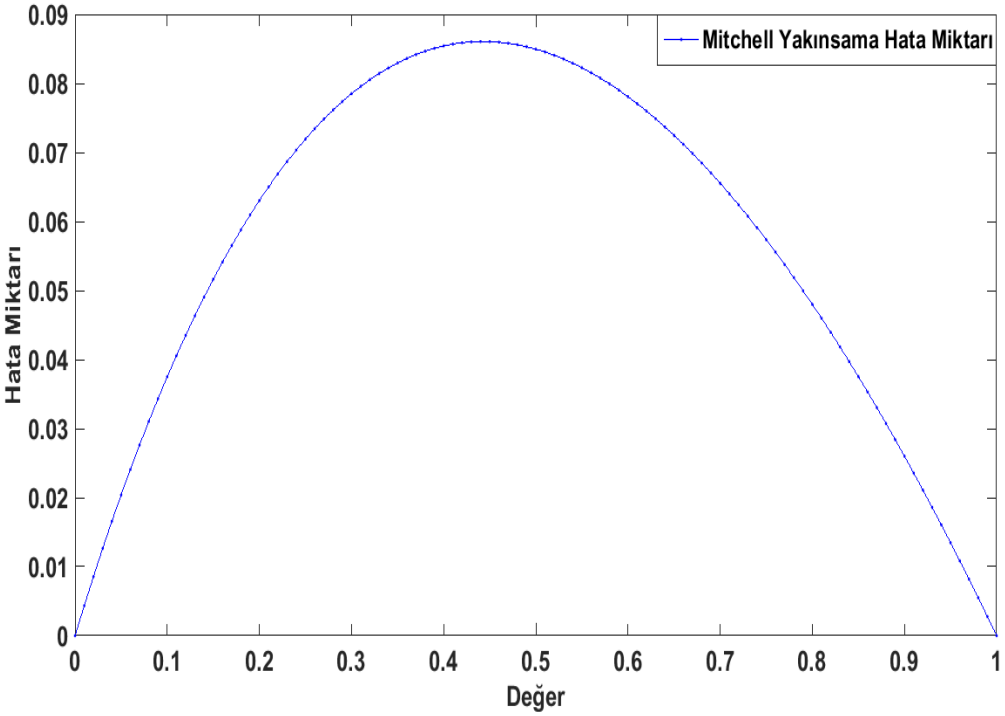
$$\log_2(1+x) \approx x \quad 0 \leq x \leq 1 \tag{2.1}$$

yakınsamasını kullanır [3]. Bu yakınsama çok dar bir aralığı kapsar ve 2'den büyük sayılara uygulanamamaktadır. Aslında Taylor açılımı da ifade eden bu yakınsama $x = 0$ ve $x = 1$ noktası için doğru sonuç verse de diğer değerler için doğru sonuç vermemektedir. Doğrusal bir yakınsama olan Mitchell yakınsaması [0,1] aralığındaki logaritma fonksiyonunu doğrusallaştırmış olur. (1)'deki yakınsama üzerine kurulu bu yöntem, hesaplama açısından basitlik sağlasa da özellikle $x = 0.5$ değerlerine yakın değerlerde yüksek hata oranı vermektedir. Bunun nedeni ise $\log_2(1+x)$ fonksiyonu [0,1] aralığında dış bükey bir eğri çizer iken y = x doğrusu ile en büyük hatayı orta kısmında vermektedir. $x = 0.5$ için ise hata miktarı 0.086 olmaktadır. Şekil 2.1'de Mitchell yakınsaması ve $\log_2(1+x)$'in gerçek değerleri gösterilmektedir.

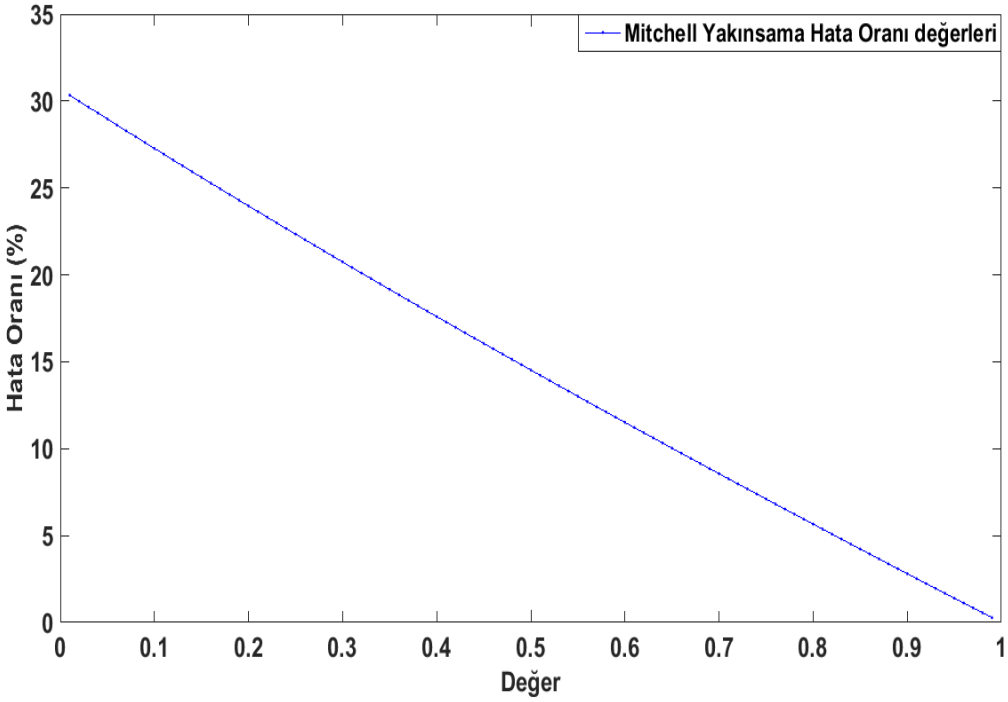


Şekil 2.1 : Mitchell yakınsaması ve logaritma gerçek değer ilişkisi

Şekil 2.2'de ise Mitchell yakınsaması ile gerçek değerler arasındaki gösterilmektedir. Şekil 2.3'de ise Mitchell yakınsamasında oluşan hataların gerçek değerlere oranı yüzde olarak gösterilmektedir.



Şekil 2.2 : Mitchell yakınsama hata miktarı



Şekil 2.3 : Mitchell yakınsama hata oranı değerleri

Şekil 2.2 ve Şekil 2.3'ten anlaşılacağı üzere 0'a yakın değerlerde hata miktarı küçük olsa da hatanın gerçek değere oranı %30'a ulaşmaktadır. Ortalama hata oranı ise %15 olmaktadır.

2.1.1 Mitchell yakınsaması tabanlı kısmi doğrusallaştırma metodu

Yüksek hata oranı, Mitchell yakınsamasının yüksek hassasiyetli çalışmalarda tercih edilmemesine neden olmaktadır. [4]'de bu yakınsamadaki hata oranını azaltmak için yakınsamaya ilave olarak:

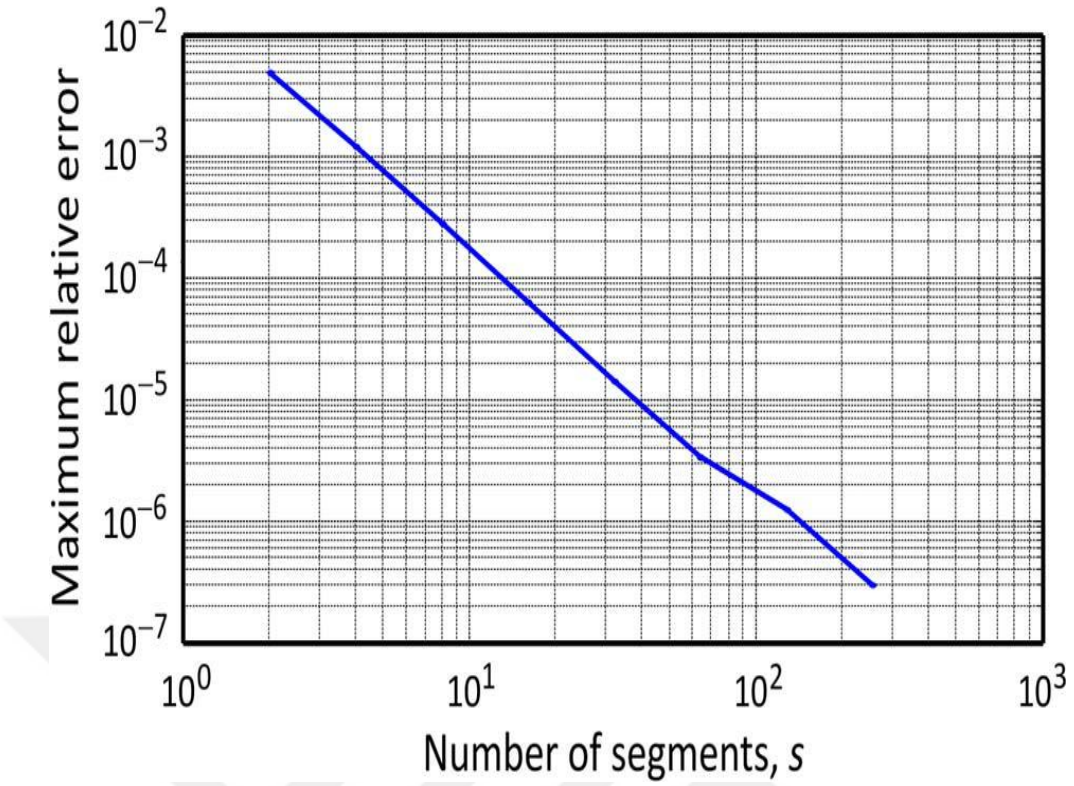
$$\log_2(1+\mathrm{x}) \approx \mathrm{x} + \lambda(\mathrm{x}) \tag{2.2}$$

$$\lambda(\mathrm{x}) = \mathrm{a_j x} + \mathrm{b_j} \quad \frac{\mathrm{j}-1}{\mathrm{s}} < \mathrm{x} < \frac{\mathrm{j}}{\mathrm{s}} \quad \mathrm{j} = 1,2,\ldots.\mathrm{s} \tag{2.3}$$

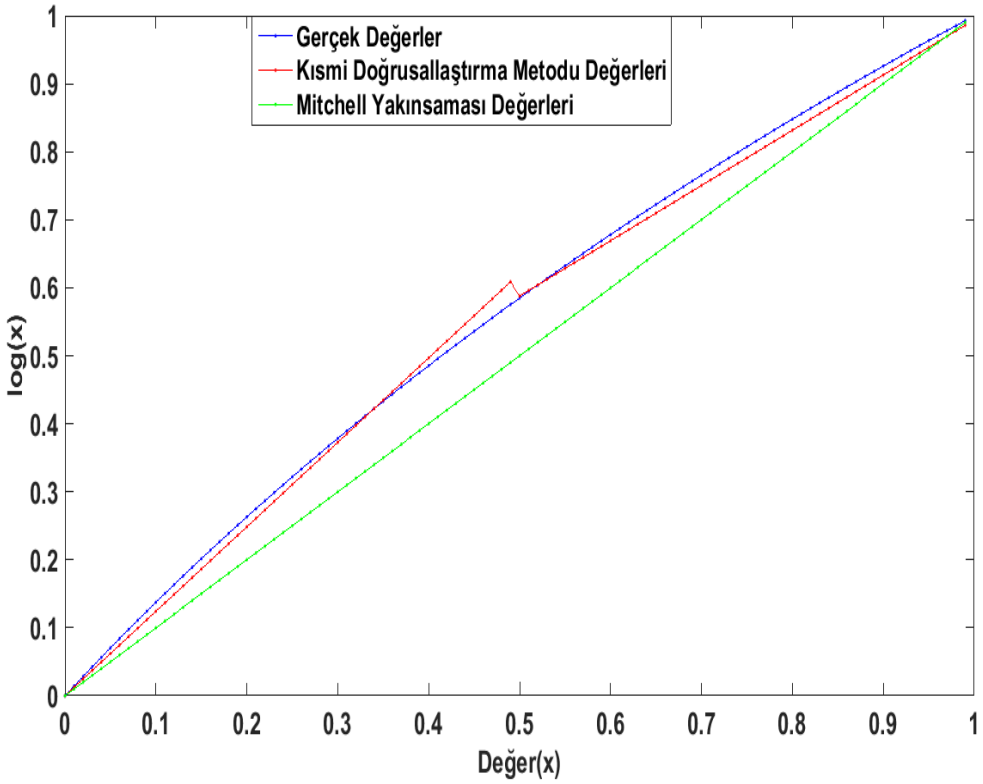
şeklinde kısmi doğrusal fonksiyonlar eklenmektedir. Böylelikle [0,1] sayı aralığı eşit şekilde s parçaya bölünmüş ve [0,1] değer aralığının farklı bölgelerinde farklı doğrusal fonksiyonlar kullanarak hata oranını azaltılması amaçlanmıştır. Her bir doğru parçasının katsayıları önceden hata oranını en aza indirecek şekilde belirlenmektedir. Göreceli yaklaşık hatayı ölçmek için:

$$\mathrm{e(N)} = \frac{\log_2(1+\mathrm{x}) - x - \lambda(\mathrm{x})}{k + \log_2(1+\mathrm{x})} \tag{2.4}$$

metriği kullanılmıştır[4]. Kısmi bölge sayısını belirten s parametresinin maksimum göreceli hataya olan etkisi ise Şekil 2.4'te verilmiştir[4]. Şekil 2.4'e göre kısmi bölge sayısının üstel artışına maksimum göreceli hata doğrusal şekilde tepki vermektedir. Bu da kısmi bölge sayısının arttırılmasının performansta doğrusal bir artışa neden olmayacağını göstermektedir. Kısmi bölge sayısını iki olacak şekilde $\mathrm{a_j}, \mathrm{b_j}$ değerleri [4]'e göre optimize edildiğinde Şekil 2.5'deki grafik çıkmaktadır.



Şekil 2.4: Kısmi bölge sayısını belirten parametrenin maksimum hataya etkisi[4]



Şekil 2.5: Metotlar arasındaki fark

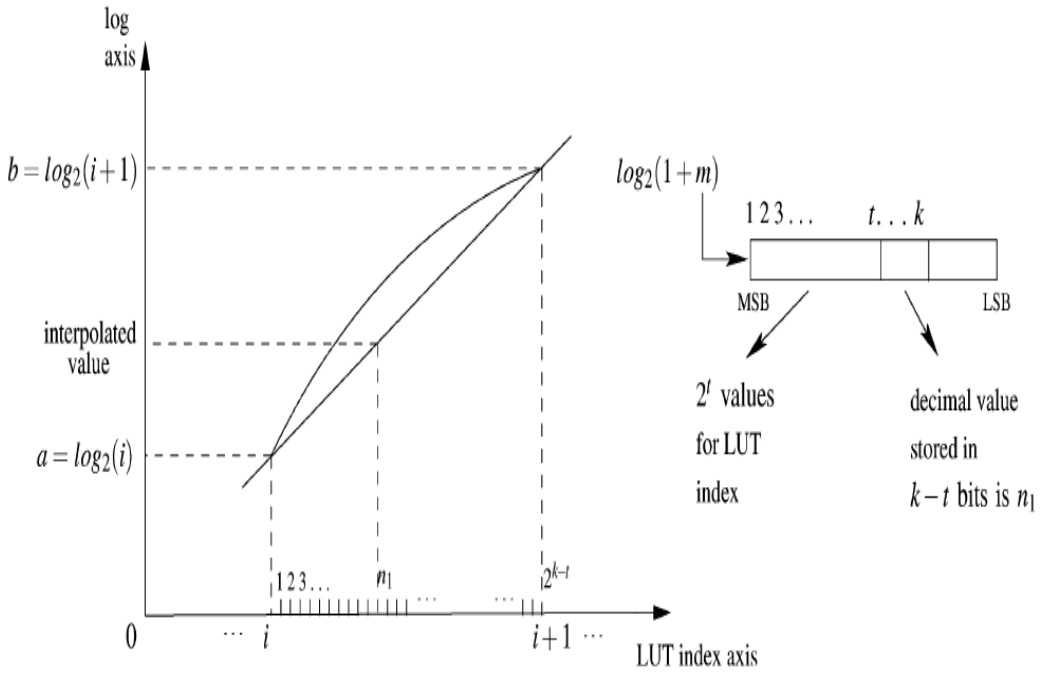
Şekil 2.5'ten de anlaşılacağı gibi [4]'te belirtilen metot Mitchell yakınsamasındaki tek doğru parçasını ikiye kırıp hatayı azaltacak şekilde gerçek değerlere yaklaşmaktadır. Eğer x'in alacağı [0,1] değer aralığı 8 kısmi parçaya ayrılırsa hata oranı %0.0635 değerine kadar düşmüştür [4]. LUT ve çarpıcı kullanımına dayanan bu yöntem ayrıca gerekli a_j ve b_j değerlerinin öncül olarak hesaplanmasını da gerektirmektedir. Yüksek hassasiyetli bu değerler yüksek basamaklı çarpma işlemini de beraberinde getirdiği için çok fazla çarpıcı ihtiyacı gerektirmektedir.

2.1.2 Mitchell yakınsaması tabanlı ağırlıklı enterpolasyon yöntemi

Bir başka Mitchell yakınsamasına dayalı yöntem olan [5]'deki yöntemde öncelikli olarak, x'in alacağı [0,1] değer aralığı eşit sayıda düzgün şekilde örneklenmiştir. Her bir örnek için bu yakınsama ile önceden hesaplanmış örneklerin logaritma değerleri arasındaki farklar bir hata değeri olarak bir LUT'a kaydedilmiştir. Ardından metoda giren girdi değerine göre ilk olarak LUT'tan bu değere ait hata değeri çekilir ve Mitchell yakınsamasına eklenir. Bu sayede her bir girdi değeri için çıktı olarak hatasız logaritma değeri bulunmuş olur.

$$\log_2(1+x) \approx x + m, \quad m = \mathrm{LUT}(x) \qquad m= 1,2\ldots,s. \tag{2.5}$$

Bu metot çıktıda yüksek hassasiyetli doğruluk vermesine rağmen, düşük örnekleme çözünürlüğü istediği için yüksek LUT kaynağına ihtiyaç duymaktadır[5]. LUT ihtiyacını azaltmak için ise [5]'de ağırlıklı enterpolasyon (weighted interpolation) kullanılmaktadır. Hafızaya kaydedilen hata oranları seyrekleştirilerek toplam örnek sayısı azaltılmaktadır. Yöntemin girdisindeki sayının hangi iki örnek arasında olduğu tespit edilir. Ardından tespit edilen iki örneğin girdi olan sayıya göre ağırlıkları tespit edilir. Tespit edilen ağırlıklara göre de enterpolasyon yapılır ve böylece örnekleme dışında kalan bütün ara değerler enterpolasyon ile bulunmuş olur[5]. Şekil 2.6'da bu ağırlıklı enterpolasyonun hafıza elemanları üzerinden nasıl yapıldığı grafik üzerinde gösterilmiştir[5].



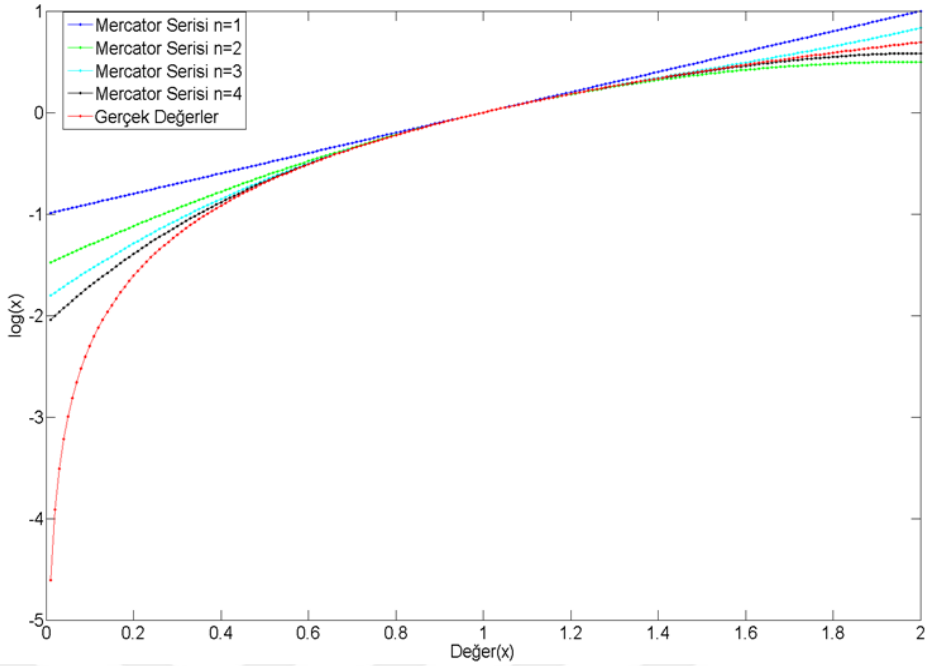
Şekil 2.6: Ağırlıklı enterpolasyonun hafıza elemanları üzerinden yapılışı[5]

2.2 Mercator Serileri

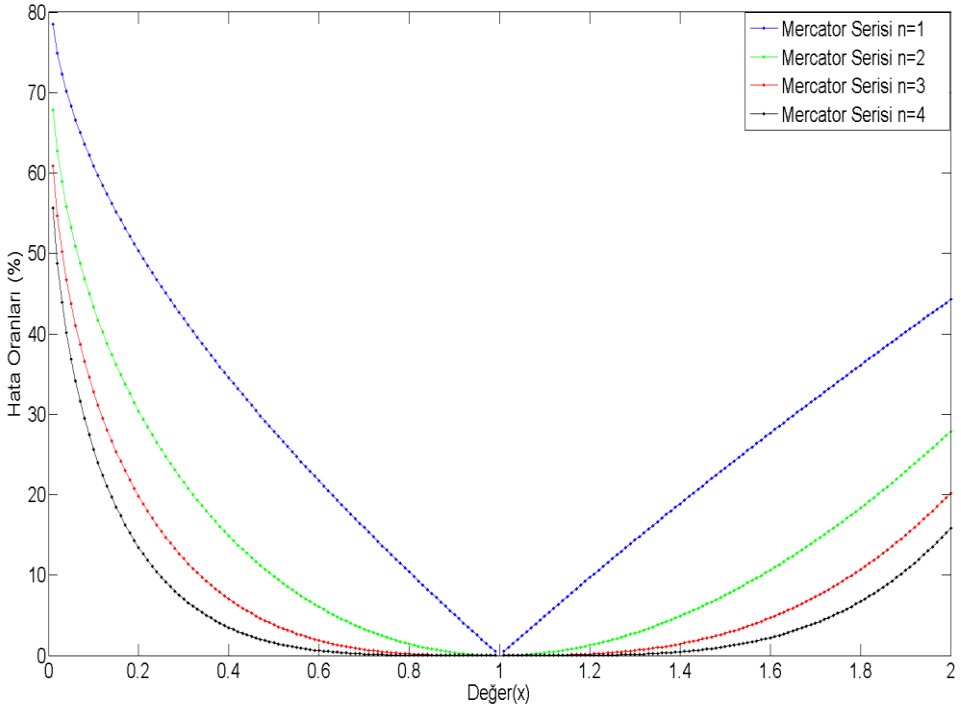
Logaritma hesaplanmasında sıklıkla tercih edilen bir diğer yöntem kümesi ise polinom tabanlı logaritma hesaplamadır. Bu yöntemlerde logaritma fonksiyonun Taylor serisi açılımı üzerinden hesaplama tercih edilir. Logaritma fonksiyonu için Taylor serisi açılımı:

$$\ln(1+\mathrm{x}) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} x^n = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots \quad |x| \leq 1,\ x \neq -1 \qquad (2.6)$$

şeklinde olmaktadır. Bu seri açılımına Mercator serisi de denmektedir. Mercator açılımı [0,2] aralığındaki sayıların doğal tabanda logaritmasının bulunması için kullanılır. Bunun dışındaki aralıklarda ise seri açılımı doğru sonuca yakınsamamaktadır. Bu yüzden polinom tabanlı yöntemler operasyon alanı olarak [0,2] aralığını seçerler. Mercator serisine göre seri açılımının derecesi ne kadar yükselir ise çıktının doğruluk hassasiyeti de yükselmektedir. Farklı derecelere göre Mercator serisinin sonuçları Şekil 2.7'de gösterilmektedir. Şekil 2.7'den de anlaşılacağı gibi Mercator serisinde x'in -1 ve 1'e yakın değerleri için hata oranları yükselmektedir. Bütün Mercator serisi açılımları x=0 için doğru sonuç verse de kenarda kalan değerler için seri açılımı gerçek değerlerden yüksek sapma göstermektedir. Şekil 2.8'de ise Mercator serilerinin farklı dereceleri için gerçek değerlere olan hata oranları verilmektedir.



Şekil 2.7: Farklı derecelere göre mercator serisinin sonuçları



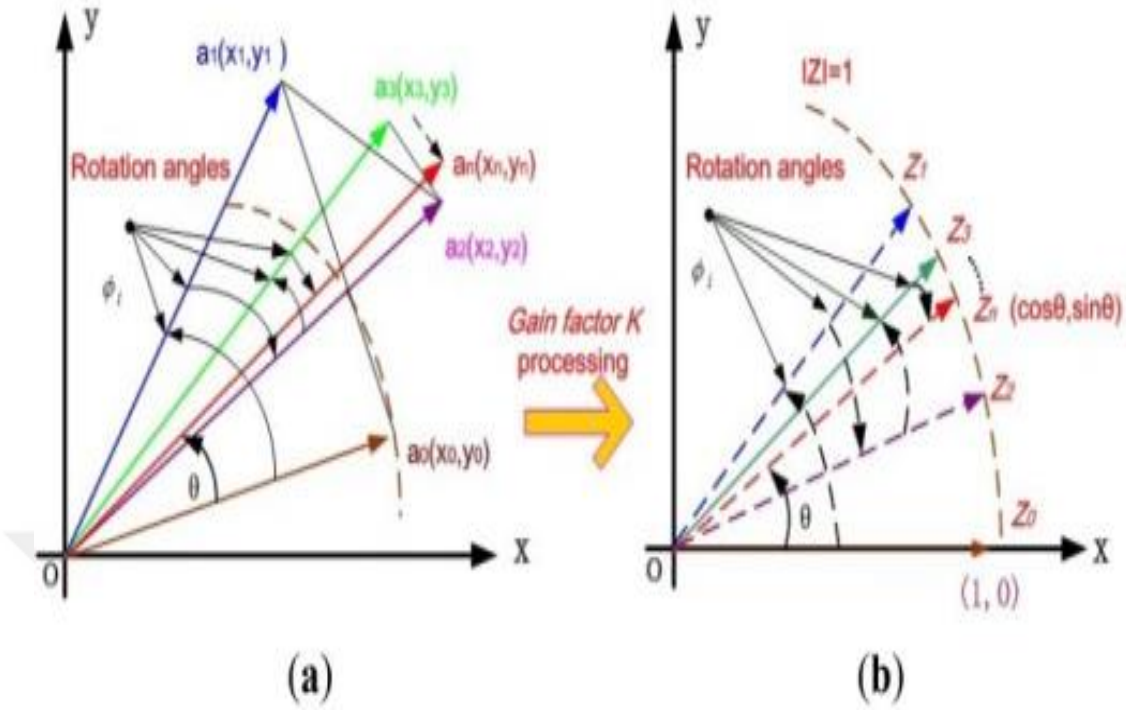
Şekil 2.8: Mercator serilerinin dereceye göre hata oranları

Şekil 2.8'e bakıldığında yakınsamanın kıyı değerlerinde hata oranının %50'nin üzerine çıktığı, x=1 civarında ise küçüldüğü gözlemlenmektedir. Bu nedenle uygulanan Mercator serisi düşük hata oranı ile tüm alanda çalışmamaktadır.

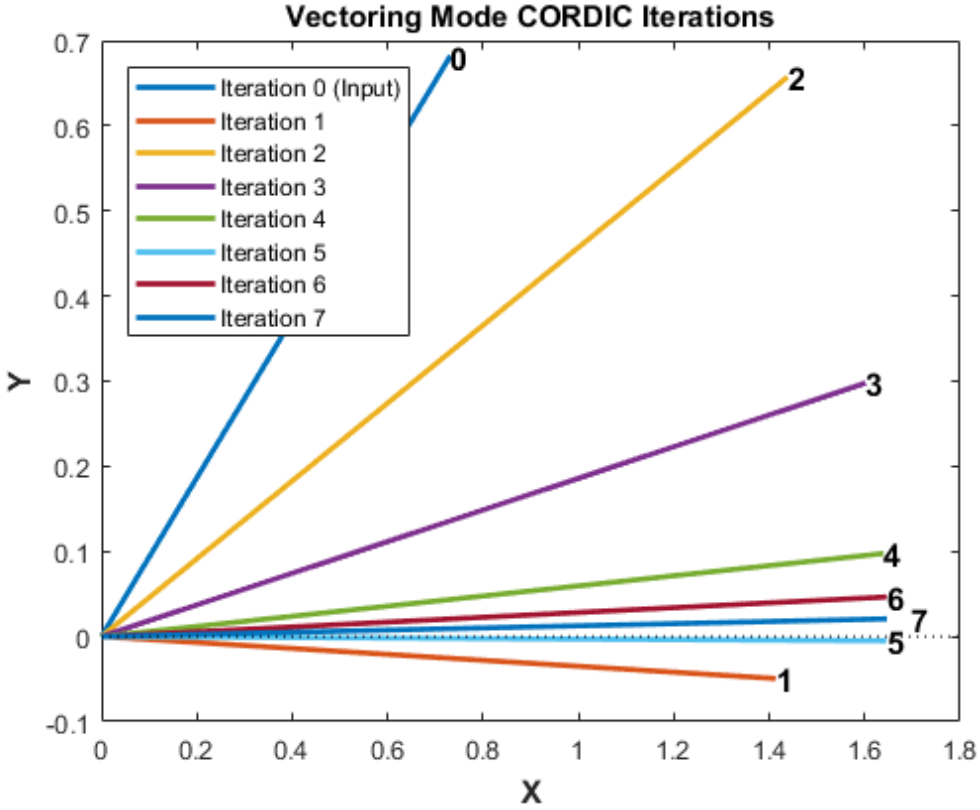
2.3 CORDIC Algoritması

1971 yılında Walther tarafından geliştirilen CORDIC algoritması, trigonometrik ve hiperbolik fonksiyonların hesaplanmasını yinelemeli bir metot ile verimli bir şekilde yapabilmektedir [6]. Trigonometrik değeri verilen açıların hesaplanması için sıklıkla CORDIC algoritmasına başvurulmaktadır. Hiperbolik fonksiyonlara da genişleyebilen bu algoritma, ters hiperbolik fonksiyonların hesabında da kullanılmaktadır. Yinelemeli şekilde ilerleyen bu algoritma girdi olarak 2 sayı ister iken, sonuçta ise tek çıktı verir. Ters trigonometrik fonksiyonlar için, hesaplanmak istenen açıya ait sin() ve cos() değerleri CORDIC algoritmasına girdi olarak verilir. Bunun sonucunda ise hesaplanmak istenen açı bulunmuş olur. Hiperbolik fonksiyonlar için de yine hesaplanmak istenen açının sinh() ve cosh() değerleri CORDIC algoritmasına verildiğinde, sonuçta istenilen açının değeri bulunmuş olur. CORDIC algoritmasının çalışma aralığı, koordinat düzleminin (veya hiperbolik koordinat düzleminin) tamamıdır. Bu algoritma, girdi olarak verilen herhangi bir (x,y) ikilisi ile sonuç verebilmektedir. CORDIC algoritması yineleme sayısını sonsuza götürüldüğünde çıktıdaki değer gerçek değere eşittir. Bir başka deyişle CORDIC algoritması yineleme adım sayısı sonsuza gittiğinde CORDIC algoritması hatasız sonuç bulur. CORDIC algoritmasının trigonometrik ve hiperbolik fonksiyonların hesabını yapar iken temel çalışma prensibi vektörel rotasyondur. Koordinat düzlemi üzerindeki herhangi bir (x, y) vektörünün açısı, açısal rotasyonlar yaparak bulunmaya çalışılır.

CORDIC algoritması bu açısal rotasyonu iki farklı yöntem ile gerçekleştirir. Bu yöntemler rotasyon ve vektör modlarıdır. Rotasyon modunda CORDIC algoritması, y=0 değeri ile yinelemeleri başlatır. Her bir yinelemede girdi olarak verilen (x, y) ikilisine yakınlaşmayı amaçlar. Her bir adımda yönelim değerine göre açı değerini yönlü olarak (+,-) biriktirir. Yinelemeler bittiğinde veya (x, y) değerine ulaştığında CORDIC algoritması biriktirdiği açı değerini vektörün açı değeri olarak çıktıya verir. Vektör modunda ise, CORDIC algoritması (x, y) vektörünü başlangıç noktası olarak alır. Her bir adımda belirli açısal rotasyonlar yapar. Her adımda yapılan açısal rotasyonun açısını yine yönlü olarak biriktirir. Vektör modunda CORDIC algoritması y değerini 0'a eşitlemeyi amaçlar. Yinelemeler bittiğinde veya y değeri 0'a ulaştığında CORDIC algoritması biriktirdiği açı değerini vektörün açı değeri olarak çıktıya verir. Şekil2.9'da rotasyon modunun Şekil 2.10'da ise vektör modunun çalışma şekli gösterilmiştir [9],[10].



Şekil 2.9: CORDIC algoritması rotasyon modu çalışma şekli



Şekil 2.10: CORDIC algoritması vektör modu çalışma şekli

CORDIC algoritmasının vektör modunda herhangi bir adımında yapılan θ derecelik saat (-) yönünde bir rotasyon ile

$$\begin{aligned} x_{i+1} &= x_i \cos(\theta) + y_i \sin(\theta) \\ y_{i+1} &= y_i \cos(\theta) - x_i \sin(\theta) \\ z_{i+1} &= z_i + \theta \end{aligned} \tag{2.7}$$

yeni (x_{i+1}, y_{i+1}) vektörü elde edilir. Açı değerlerini biriktirdiğimiz z_i değeri ise θ kadar artar. Eğer θ derecelik rotasyon saatin tersi(+) yönünde yapılır ise,

$$\begin{aligned} x_{i+1} &= x_i \cos(\theta) - y_i \sin(\theta) \\ y_{i+1} &= y_i \cos(\theta) + x_i \sin(\theta) \\ z_{i+1} &= z_i - \theta \end{aligned} \tag{2.8}$$

şeklinde olur. Bu durumda her bir adımda yapılan rotasyon yönü yinelemedeki işareti değiştirmektedir. (7)'deki yeni (x_{i+1}, y_{i+1}) hesabını matris-vektör çarpımı haline getirecek olursak,

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \cos(\theta) \begin{bmatrix} 1 & \tan(\theta) \\ -\tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2.9}$$

şeklinde olur. Burada $\begin{bmatrix} x_i \\ y_i \end{bmatrix}$ CORDIC algoritmasının bir önceki adımdaki vektör değerleri, $\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix}$ bir sonraki adım vektör değerleri, $\cos(\theta)$ kazanç faktörü ve $\begin{bmatrix} 1 & \tan(\theta) \\ -\tan(\theta) & 1 \end{bmatrix}$ ise rotasyon matrisidir. Her bir yinelemede yapılacak rotasyonun yönüne göre rotasyon matrisindeki $\tan(\theta)$ ve $-\tan(\theta)$ ifadeleri ve açısal birikim hesabı işaret değiştirirken diğer bütün değerler rotasyon yönünden bağımsızdır. Ters hiperbolik fonksiyonlar için (7)'deki ifade,

$$\begin{aligned} x_{i+1} &= x_i \cosh(\theta) + y_i \sinh(\theta) \\ y_{i+1} &= y_i \cos h(\theta) + x_i \sinh(\theta) \\ z_{i+1} &= z_i - \theta \end{aligned} \tag{2.10}$$

şeklinde olur. Aynı şekilde (9)'daki ifade ise,

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \cos h(\theta) \begin{bmatrix} 1 & \tanh(\theta) \\ \tanh(\theta) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

(2.11)

şeklinde dönüşür. Bu durumda d_i'yi rotasyon yönüne göre 1 veya -1 değeri alan bir yönelim değişkeni olarak (9)'a ilave eder isek, hiperbolik fonksiyonlar için CORDIC algoritması yinelemeleri

$$\begin{cases} x_{i+1} = \cosh(t_i)\,[x_i + d_i \tanh(t_i) y_i] \\ y_{i+1} = \cosh(t_i)\,[y_i + d_i \tanh(t_i)\, x_i] \\ z_{i+1} = z_i - d_i t_i \end{cases} \tag{2.12}$$

şeklinde olur. Logaritma hesabını CORDIC algoritması ile yapabilmek için, hiperbolik fonksiyonlardan logaritmik fonksiyonlara geçiş,

$$\tanh^{-1}(t) = \frac{1}{2}\ln\left(\frac{1+t}{1-t}\right) \tag{2.13}$$

ile sağlanır. Vektör yöntemindeki her bir yinelemedeki $\tanh(t_i) = 2^{-i}$ olarak alınırsa (2.12) aşağıdaki duruma dönüşür:

$$\begin{cases} x_{i+1} = \cosh(t_i)\,[x_i + d_i 2^{-i} y_i] \\ y_{i+1} = \cosh(t_i)\,[y_i + d_i 2^{-i} x_i] \\ z_{i+1} = z_i - d_i \tanh^{-1}(2^{-i}) \end{cases} \tag{2.14}$$

Böylelikle CORDIC algoritmasındaki her bir adım bir önceki adımdan çıkan x_i ve y_i değerlerinin 2^{-i} ile çarpılması ile bir sonraki adımdaki x_{i+1} ve y_{i+1} değerleri bulunabilir. Burada $\tanh(t_i) = 2^{-i}$ çarpanının tercih edilmesinin nedeni ise ikili aritmetikte 2^{-i}'nin basamak kaydırmaya denk gelmesidir. Böylece donanım mimarisi olarak kaydırılarak toplama(shift-and-add) ile CORDIC algoritmasının yinelemelerindeki x_{i+1} ve y_{i+1} değerleri kolayca bulunabilir. z_i değeri ise küçük bir tabloda tutulacak $\tanh^{-1}(2^{-i})$ değerlerinin toplamı-çıkarımı ile hesaplanabilir. Bu formüllerdeki d_i yönelim değerleri ise şu şekilde belirlenir:

$$\begin{cases} d_i = 1 & y_i < 0 \\ d_i = -1 & y_i \geq 0 \end{cases} \tag{2.15}$$

Eğer $z_1 = 0$ olur veya yeterli N adım yinelenir ise,

$$z_{N+1} \approx \tanh^{-1}\left(\frac{y_1}{x_1}\right) = \frac{1}{2}\ln\left(\frac{x_1+y_1}{x_1-y_1}\right) \tag{2.16}$$

şeklinde olur. Doğal logaritması hesaplanmak istenen bir k değeri için,

$$(x_1 = k+1, y_1 = k-1) \rightarrow \ln(k) = 2z_{N+1} \tag{2.17}$$

şeklinde bulunur. Böylelikle, doğal logaritması hesaplanmak istenen her k değeri için, CORDIC algoritmasına girdi olarak verilebilecek (x,y) ikilisi k değerinden yukarıda belirtildiği şekilde çıkarılır ve CORDIC algoritmasından da çıktı olarak z_{N+1} değeri alınır.

3. GELİŞTİRİLEN LOGARİTMA ÇEVİRİCİSİ İCTLÇ METODU

3.1 İCTLÇ'ye Genel Bakış

Geliştirilen logaritma çeviricisi CORDIC algoritması tabanlı çalışmaktadır. İCTLÇ metodu donanım mimarisi için optimize edilmiş olup, yüksek performansı az kaynak kullanımı ile yakalamaya çalışmaktadır.1'den büyük eşit herhangi bir reel sayı olan K sayısının 2'li sistemde d-bit işaretsiz(unsigned) olarak k sayısı ile ifade edildiğini ve bu k sayısının İCTLÇ'ye girdi olarak verildiğini düşünelim. Bu durumda,

$$0 \leq k \leq 2^d - 1 \quad k \in Z^+ \tag{3.1}$$

lur. Bir başka ifade ile d-bit işaretsiz k sayısı 0 ile $2^d - 1$ arasında pozitif tamsayılar alabilir. Sayısal donanım ortamında ondalıklı bir sayı virgülü kaçıncı bit'den itibaren seçtiğine bağlı olduğu için d-bitlik k sayısını bir pozitif tamsayı olarak düşünebiliriz. CORDIC algoritmasının çıktısı olan z_{N+1} değeri için,

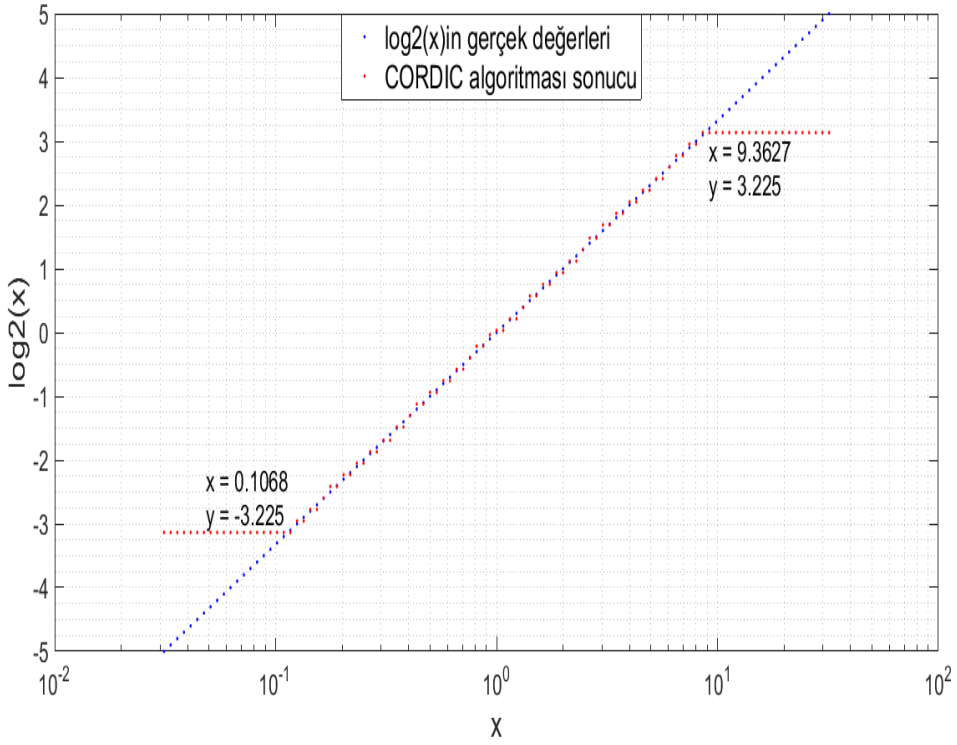
$$|z_{N+1}| \approx \left|\tanh^{-1}\left(\frac{y_1}{x_1}\right)\right| = \sum_{i=1}^{N} \tanh^{-1}\left(2^{-i}\right) \tag{3.2}$$

$$\sum_{i=1}^{N} \tanh^{-1}\left(2^{-i}\right) < 1.118 \rightarrow \left|\frac{y_1}{x_1}\right|_{max} \approx 0.807 \tag{3.3}$$

olur. Bunun sebebi, z_{N+1} üzerinde toplanan açısal değerlerin maksimum değeri yönelimin her zaman negatif yönde olduğu durumda gerçekleşir ve tabloda tutulan $\tanh^{-1}\left(2^{-i}\right)$ değerlerinin toplamına eşittir; toplanan açısal değerlerin minimum değeri ise yönelimin her zaman pozitif yönde olduğu durumda gerçekleşir ve tabloda tutulan $\tanh^{-1}\left(2^{-i}\right)$ değerlerinin toplamının negatif değerine eşittir. Bu durumda logaritma çevirimi için CORDIC algoritmasının çalışma aralığı (3.3)'deki (x,y) ikilisini sağlayacak k aralığına inmek zorundadır. Bu durum ve (2.17)'e göre k'nın alması gereken değer aralığı,

$$\frac{-k+1}{k+1} < 0.807 \,, \frac{k-1}{k+1} < 0.807 \rightarrow 0.1068 \leq k \leq 9.3627 \tag{3.4}$$

şeklindedir. (3.3)'e göre CORDIC algoritmasının çalışma alt sınırı 0.1068'dir. Üst sınırı ise 9,3627'dir. Şekil 3.1'de CORDIC metodunun çıktı değerleri ve gerçek logaritma çevrim değerleri verilmiştir. Grafiğe göre (3.4)'te belirtilen sınırların dışında girdi aldığında CORDIC metodu doğru sonuç vermemeye başlamaktadır. Belirtilen sınırlar CORDIC algoritmasının doyum değerleridir. Çünkü $\tanh^{-1}(2^{-i})$ değerlerini tutan hafızanın bütün elemanlarının toplamı belirli bir sayıyı geçemeyeceği için maksimum değerden büyük bütün değerler için sonuç doyum değeri olacaktır.



Şekil 3.1: CORDIC algoritması çalışma aralığı

Bu durumu aşmak için İCTLÇ, CORDIC metodunu kullanmadan önce girdi olarak verilen k değerini (3.4)'te belirtilen bölgeye indirir. Bu indirgemeyi ise,

$$\alpha(k) = 2^n = \{k'\text{dan büyük ilk } 2'\text{nin kuvveti olan sayı}\} \qquad (3.5)$$

$$\beta(k) = m = \frac{k}{\alpha(k)} \qquad (3.6)$$

$$k = m * 2^n, \quad 2^{n-1} < k \leq 2^n \ , \quad 0.5 < m \leq 1 \qquad (3.7)$$

şeklinde yapar. İlk aşamada k'dan büyük en küçük 2'nin kuvveti sayı bulunur. Daha sonra ise k değeri bulunan bu sayıya bölünerek logaritma çevriminin ondalıklı kısmını belirleyecek m sayısı bulunur. Böylelikle logaritma çevrimi,

$$\log_2 k = \log_2 m + n \tag{3.8}$$

şeklinde olur. Bu ayrışımı donanımsal olarak yapabilmek için İCTLÇ'nin içinde Aralık İndirgeme Blok'u(AİB) bulunmaktadır. İCTLÇ'ye girdi olan k değeri doğrudan bu bloğa girdi olarak girer ve bu bloktan (3.7)'deki m ve n değerleri çıktı olarak sunulur. AİB'den çıkan n değeri (3.8)'deki İCTLÇ'nin çıktısını hesaplamak için yapılan toplama işlemine sunulurken, m değerinden CORDIC Algoritması Blok(CAB)'una girdi olabilmesi için (2.17)'ye göre (x_1, y_1) değerleri bulunur. Bulunan (x_1, y_1) değerleri sayısal olarak ifade edildiği için sonsuz çözünürlük barındırmaz. Eğer bu değerler küçük ise, CORDIC algoritmasının her bir yinelemesinde yapılan (2.14)'deki ikili aritmetik (binary arithmetic) işlemler, y_i sayısını bütün yinelemeleri tamamlamadan hızlı bir şekilde sıfıra çeker. Bu durumda CORDIC algoritması ardışık yinelemelerini yeterince sağlayamadığı için doğru sonuç vermez. Bu yüzden, bulunan (x_1, y_1) değerleri CORDIC algoritmasına girmeden önce ifade edildikleri bit sayıları değişmeden çözünürlükleri yükseltilir. Böylelikle daha fazla kaynak harcamadan girdi olarak verilen her k değeri için yüksek çözünürlüklü CORDIC algoritması gerçeklenebilir. Bunun için İCTLÇ'nin içerisinde Giriş Çözünürlüğü Arttırma Blok(GÇAB)'u tanımlanmıştır. Bu blok d-bit (x_1, y_1) sayısını alarak,

$$(x_b, y_b) = \gamma(x, y) = (2^l * x, 2^l * y) \tag{3.8}$$

CORDIC algoritmasına girdisi olan (x_b, y_b) ikili başlangıç değerini üretir. (3.2)'deki l değeri ise,

$$l = \min\{d - (\lfloor \log_2(x) \rfloor + 1), d - (\lfloor \log_2(y) \rfloor + 1)\} \tag{3.9}$$

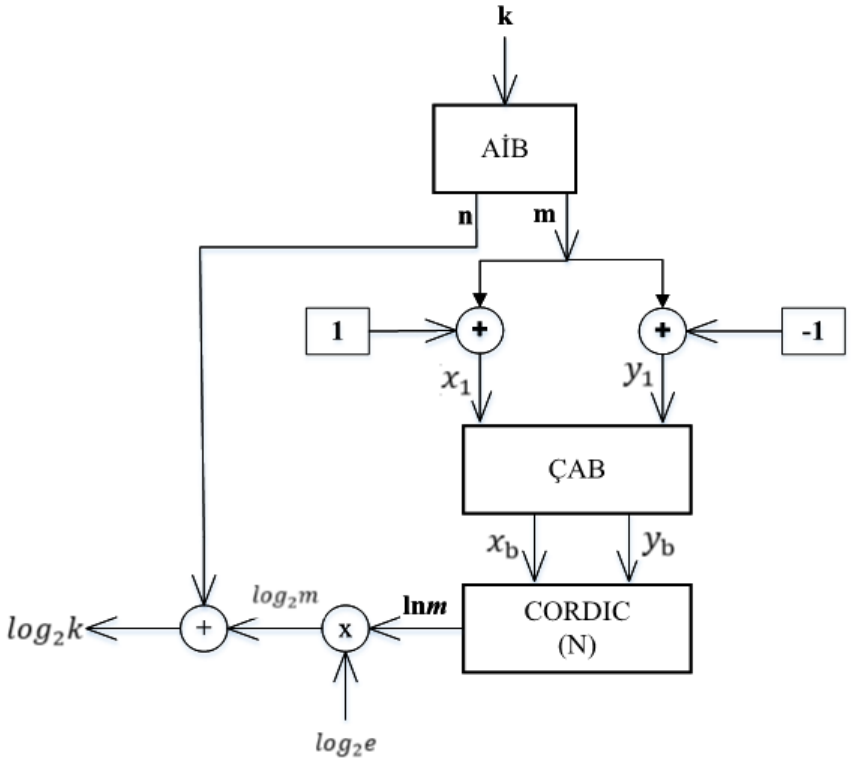
şeklinde bulunur ve (3.9)'daki $\lfloor\ \rfloor$ operatörü, aşağı yuvarlama işlemini belirtmektedir. Bir başka ifade ile, d-bit işaretli(signed) (x_1, y_1) ikilisi yine d-bit içinde ifade edilecek şekilde eşit miktarda yükseltilir. Böylelikle d-bit aritmetik işlemleri yine sabit nokta aritmetiği olarak devam eder iken, aritmetik işlem çözünürlüğü de arttırılmış olur. Ardından, bulunan (x_b, y_b) ikilisi CORDIC algoritmasına girer ve N sayıdaki yinelemeler sonucunda $z_{N+1} \approx n(m)$ olarak m değerinin doğal logaritmasını çıktı olarak verir. Çıktıda alınan $\ln(m)$ değeri e tabanında olduğu için,

$$\log_2 m = \log_2 e * \ln(m) \tag{3.10}$$

şeklinde geri iki tabanına çevrilir. Buradaki $\log_2 e$ sabit bir değer olduğu için kayıt(register)'da tutulur ve çarpma işlemi bu sabit sayı ile gerçekleştirilir. Sonuçta bulunan $\log_2 m$ sayısı, k'nin bütün değerleri için m'in 1'den küçük olmasından dolayı negatif bir sayıdır. Ardından bulunan $\log_2 m$ değeri n ile toplanır ve İCTLÇ'ye girdi olarak verilen k değerinin 2 tabanında logaritması böylelikle bulunmuş olur. Eğer k'nin başka bir tabanda logaritması bulunmak istenirse, herhangi bir f tabanı için,

$$\log_f k = \log_f 2 * \log_2 k \tag{3.11}$$

şeklinde bulunur. Böylelikle herhangi bir f tabanında herhangi bir k sayısının logaritması hesaplanmış olur. İCTLÇ'nin blok diyagramı Şekil 3.2'de belirtilmiştir.



Şekil 3.2: İCTLÇ Blok Diyagramı

3.2 FPGA Mimarisi

İCTLÇ'nin FPGA mimarisinde d-bitlik işaretsiz(unsigned) k değeri hariç bütün sayılar işaretli gösterimde(signed representation) belirtilmiştir. d-bitlik işaretsiz(unsigned) k girdisinden İCTLÇ'nin içerisindeki AİB bloğu m ve n değerlerini üretir iken,

$$n = \{1 \text{ değerinin olduğu en yüksek basamak numarası}\} \tag{3.12}$$

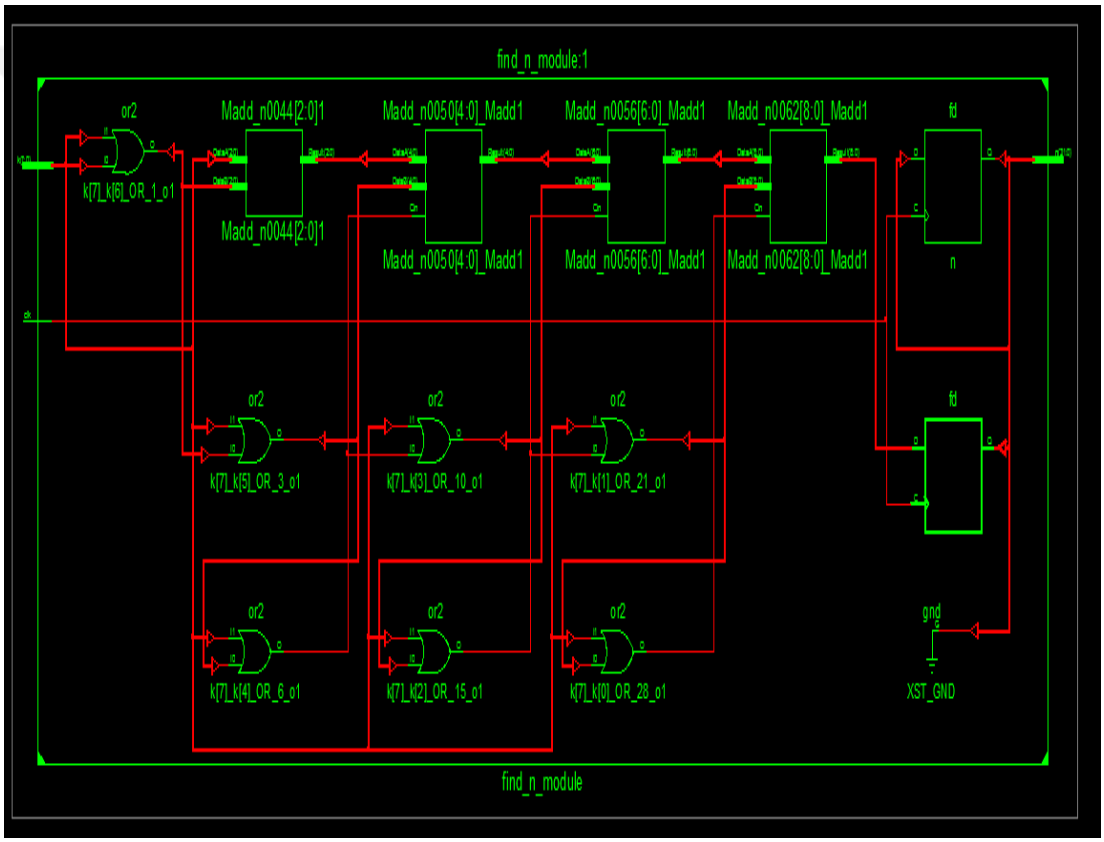
mimari metodunu kullanmaktadır. Örneğin, 8-bit ile ifade edilen 75 sayısı için,

$$(75)_{10} = (01001011)_2 \rightarrow \mathrm{n} = 7$$

olur. Bu sayede tek bir işlem saatinde n değeri tespit edilmiş olur. n sayısını bulmaya yarayan bu mimarını RTL şeması Şekil 3-3'te verilmiştir. Sadece d-1 adet OR kapısı ve $\frac{d}{2}$ adet 1-bit toplayıcı(adder) içeren bu mimarı ile n sayısı bulunabilmektedir. Ardından $(\mathrm{x}_1, \mathrm{y}_1)$ ikilisini bulmak için,

$$\mathrm{x}_1 = \mathrm{k} + 2^{\mathrm{n}}, \ \mathrm{y}_1 = \mathrm{k} - 2^{\mathrm{n}} \qquad (3.13)$$

şeklinde bulunur.



Şekil 3.3 : "n" sayısını bulmanın RTL şeması

k=75 için x_1 değeri,

$$\mathrm{x}_1 = \frac{75}{128} + 1 = \frac{203}{128}$$

şeklinde olur. x_1'i bulabilmek için yukarıdaki gibi toplama işlemi yapmak yerine k'nın $(\mathrm{n}+1)$. basamaktaki değerini 0'dan 1'e çekilir. Örneğin, yine k değerinin 75 olduğunu varsayalım. Bu durumda,

$$(75)_{10} = (01001011)_2 \rightarrow x_1 = (011001011)_2 = (203)_{10}$$

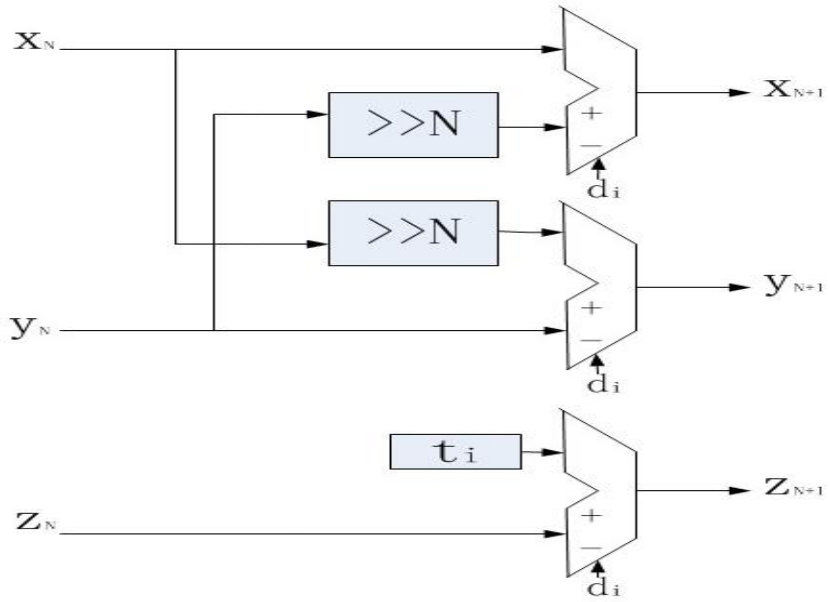
şeklinde sadece 8.bit 0'dan 1'e çekilebilir. y_1'i bulabilmek için çıkarma işlemi yerine k değerinin $(n+1)$. basamağından büyük eşit bütün basamak değerlerini 1'e çekilir yapılır. Yine k = 75 için,

$$(75)_{10} = (01001011)_2 \rightarrow y_1 = (111001011)_2 = (-53)_{10}$$

şeklinde olur. Böylelikle sadece bit değiştirme ile 1 ile toplama ve 1 çıkarma işlemleri donanım mimarisi olarak yapılmış olur.(x_b, y_b) ikilisini bulabilmek için ÇAB bloğundaki l sayısı,

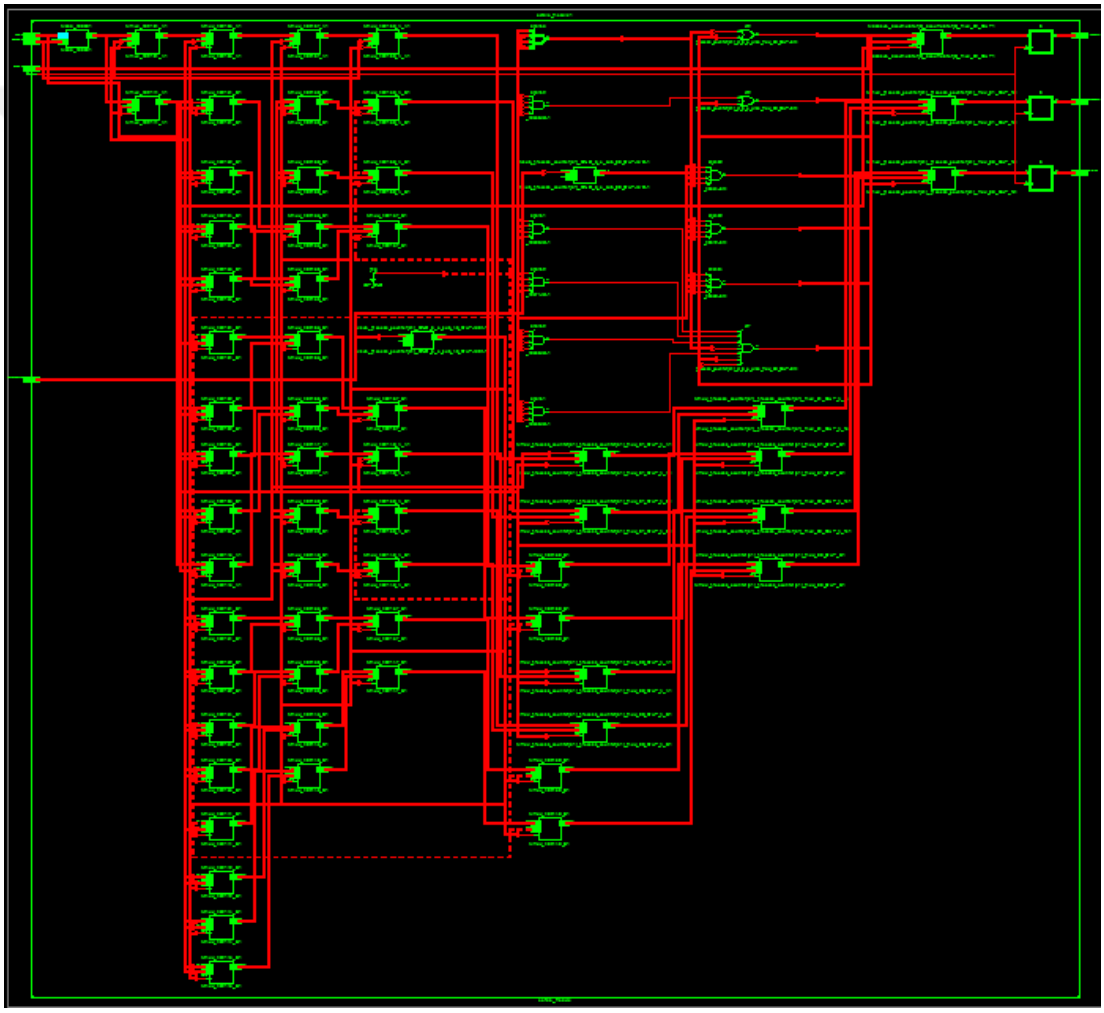
$$l = d - n \tag{3.14}$$

şeklinde bulunur ve (x_b, y_b) ikilisini bulmak için yapılacak çarpma işlemleri sadece l -bit-kaydırma(l-bit-shifting) işlemi ile yapılır. Fakat kaydırma miktarı girdiye göre değişken olduğu için statik bir kaydırma değildir, aksine kaydırma miktarı girdiye göre değişkendir. Bu durumda kaydırma miktarını belirlemek için seçici(multiplexer) elemanlarına dayalı mimarı gerekir. Şekil 3.4'te ÇAB bloğunun RTL donanım şeması verilmektedir. Şekil 3.4'te de görüldüğü üzere x_b ve y_b için iki ayrı seçici elemanı kullanılmıştır.



Şekil 3.4: ÇAB Bloğunun RTL Donanım Şeması

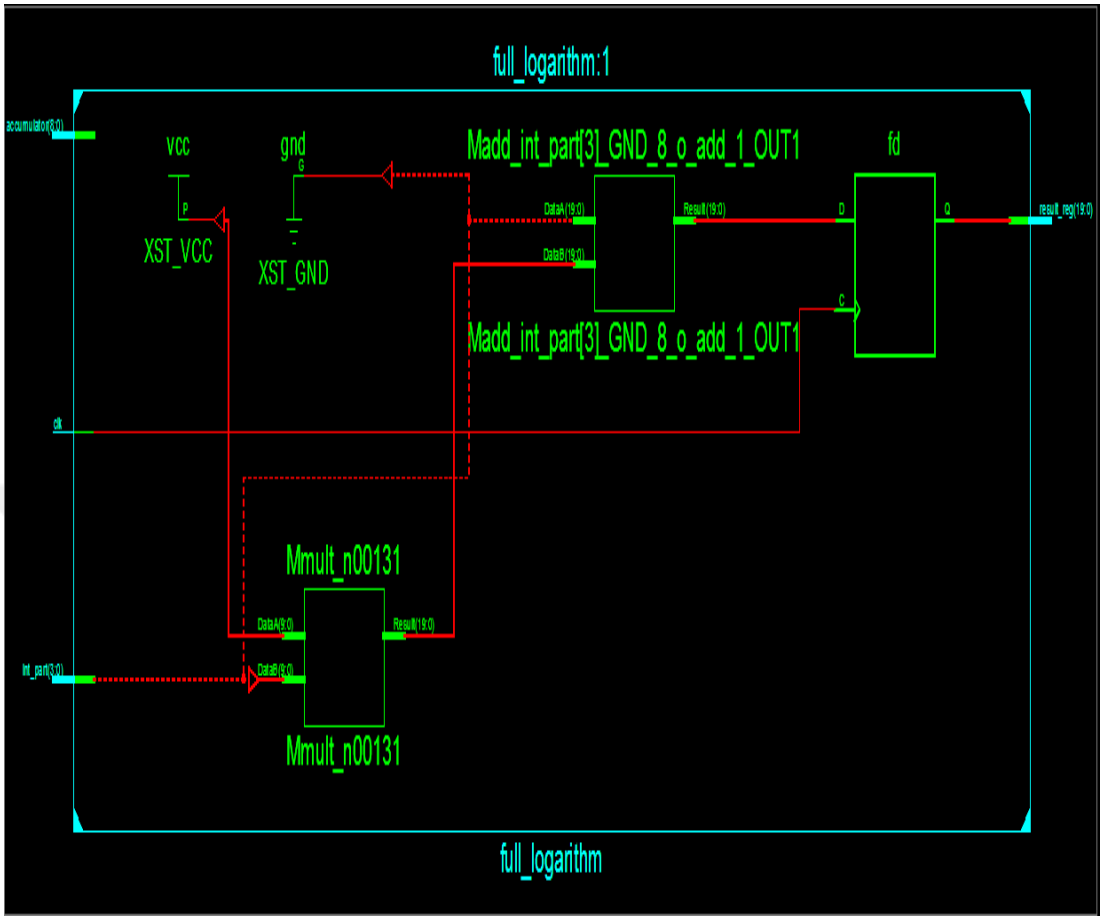
CORDIC algoritmasının mimarisi Şekil 3.5'te belirtilmiştir[7]. Bu mimaride sadece 3 tane toplama/çıkarma kullanılarak 2^{-i} ile çarpma işlemleri ise N-bit-kaydırma ile yapılmıştır. Ayrıca mimarideki $t_i = \tanh^{-1}(2^{-i})$ değerlerini tutmak için N boyutta LUT tutulmuştur. Şekil 3.6'da CORDIC bloğunun RTL şeması görülmektedir. Şemaya yakından bakıldığında yoğun bir seçici elemanı kullanıldığı görülür. Bu elemanlar yönelim elemanı olan (15)'deki d_i değerine göre (14)'ün fonksiyon işaretlerini belirlemeye yaramaktadır. Ayrıca CORDIC bloğu üzerinde iki tane toplayıcı/çıkarıcı (adder/subtractor) elemanı bulunmaktadır. Bunlarda yine Şekil 3.5'e uygun olarak CORDIC algoritmasının toplama çıkarma kısmını gerçeklemektedir.



Şekil 3.5: CORDIC Algoritması Mimarisi[7]

CORDIC bloğundan çıkan değer $\log_2 e$ ile çarpılmak için çarpıcıya girdi olarak girer. Daha sonra da k'nin logaritmasının tamsayı kısmını bulduğumuz n değeri ile toplanmak için toplayıcıya gider. Bu ikisinin birlikte yapıldığı bloğun RTL şeması Şekil 3.6'dadır. Fakat buradaki çarpıcı sabit sayı ile $\ln(m)$ değerini çarptığı için FPGA

içindeki çarpıcı(DSP) elemanı yerine mantık elemanları ile yapılır ve Şekil 3.6'da da görüldüğü gibi çarpıcının bir girişi sabit sayı olarak 1(Vcc) ve 0(GND)'a bağlanmıştır.



Şekil 3.6: İCTLÇ RTL şeması

3.3 Donanım ve Benzetim Çalışmaları

Donanım mimarisi geliştirilen İCTLÇ metodu, sayısal ortamdaki benzetim çalışmaları ile test edilmiştir. Benzetim çalışmaları için Xilinx ISE 14.7 kullanılmıştır. İlk benzetim 8-bit aritmetik üzerinde, CORDIC bloğunun 10 yineleme yapması üzerine kurulmuştur. İlk olarak $\tanh^{-1}(2^{-i})$ hafızaya kaydedetmek için MATLAB ortamında önceden hesaplanmıştır. 10 yineleme için $\tanh^{-1}(2^{-i})$ değerleri şu şekilde bulunmuştur:

$\tanh^{-1}(2^{-1}) = 0.5493$
$\tanh^{-1}(2^{-2}) = 0.2554$
$\tanh^{-1}(2^{-3}) = 0.1257$
$\tanh^{-1}(2^{-4}) = 0.0626$
$\tanh^{-1}(2^{-5}) = 0.0313$
$\tanh^{-1}(2^{-6}) = 0.0156$
$\tanh^{-1}(2^{-7}) = 0.0078$
$\tanh^{-1}(2^{-8}) = 0.0039$
$\tanh^{-1}(2^{-9}) = 0.0020$
$\tanh^{-1}(2^{-10}) = 0.0010$

Daha sonra hesaplanan bu değerler FPGA donanım mimarisi için sayısallaştırılmıştır (quantize). Yine 8-bit aritmetik işlemler yapabilmek için her bir hafıza elemanı da 8-bit ile ifade edilmesi istenmiştir. Ayrıca $\tanh^{-1}$ fonksiyonu için,

$$\tanh^{-1}(x) < 1 \leftrightarrow x < 0.7616 \tag{3.15}$$

olduğu bilinmektedir. Hafıza elemanlarının her biri de (3.15)'i sağladığı için $\tanh^{-1}(2^{-i})$ hafıza elemanlarını 0 ile 1 arasında sayısallaştırılabilir. Bu durumda sayısallaştırılmış şekliyle hafıza elemanları şu şekilde olmaktadır:

$\tanh^{-1}(2^{-1})$ = 10001101 = 141
$\tanh^{-1}(2^{-2})$ = 01000001 = 65
$\tanh^{-1}(2^{-3})$ = 00100000 =32
$\tanh^{-1}(2^{-4})$ = 00010000 =16
$\tanh^{-1}(2^{-5})$ = 00001000 = 8
$\tanh^{-1}(2^{-6})$ = 00000100 = 4
$\tanh^{-1}(2^{-7})$ = 00000010 = 2
$\tanh^{-1}(2^{-8})$ = 00000001 =1
$\tanh^{-1}(2^{-9})$ = 00000001 = 1
$\tanh^{-1}(2^{-10})$ = 00000000 = 0

Yukarıdan da anlaşılacağı üzere $\tanh^{-1}(2^{-i})$ hızlı bir sönümlemeye uğradığı için 8-bit çözünürlükte 8. ve 9. elemanlar aynı şekilde ifade edilmiş ve 10.eleman ise 0 olmuştur. Bunun nedeni [a,b] arasını d-bit ile sayısallaştırıldığında,

$$\mathrm{R} = \frac{|b-a|}{2^d} \tag{3.16}$$

şeklinde çözünürlük (R) sağlamasıdır. Bu durumda ise 0 ile 1 arası 8-bit ile ifade edildiği için,

$$\mathrm{R} = \frac{|1-0|}{2^8} = 0.0039$$

olmaktadır. Bu durumda,

$$|]x[| = |]y[| \leftrightarrow \mathrm{Rn} < \mathrm{x,y} < \mathrm{R(n+1)} \tag{3.17}$$

olur. (3.17)'deki |] [| operatörü sayısallaştırma işlemidir. Hafızanın 8. Ve 9. elemanlarına bakıldığında 0.0039 ve 0.0020 değerleri vardır. Bu değerler sayısallaştırma da aynı sayısallaştırma adımı içinde kaldıkları için birbirinden ayırt edilemez ve aynı değeri alırlar. Bu yüzden ikisi de 8-bit sayısallaştırmada *00000001* değerini alır. Ayrıca, sayısallaştırma işleminde duyarlılık(S) ise,

$$\mathrm{S} = \frac{|b-a|}{2^{d+1}} \tag{3.18}$$

şeklinde bulunur. $(a + \mathrm{S})$'den küçük bütün sayılar sayısallaştırma sırasında 0'a çekilir. Bundan dolayı hafızanın 10. elemanı olan $tanh^{-1}(2^{-10})$, 0 ile 1 arası 8-bit ile sayısallaştırıldığında,

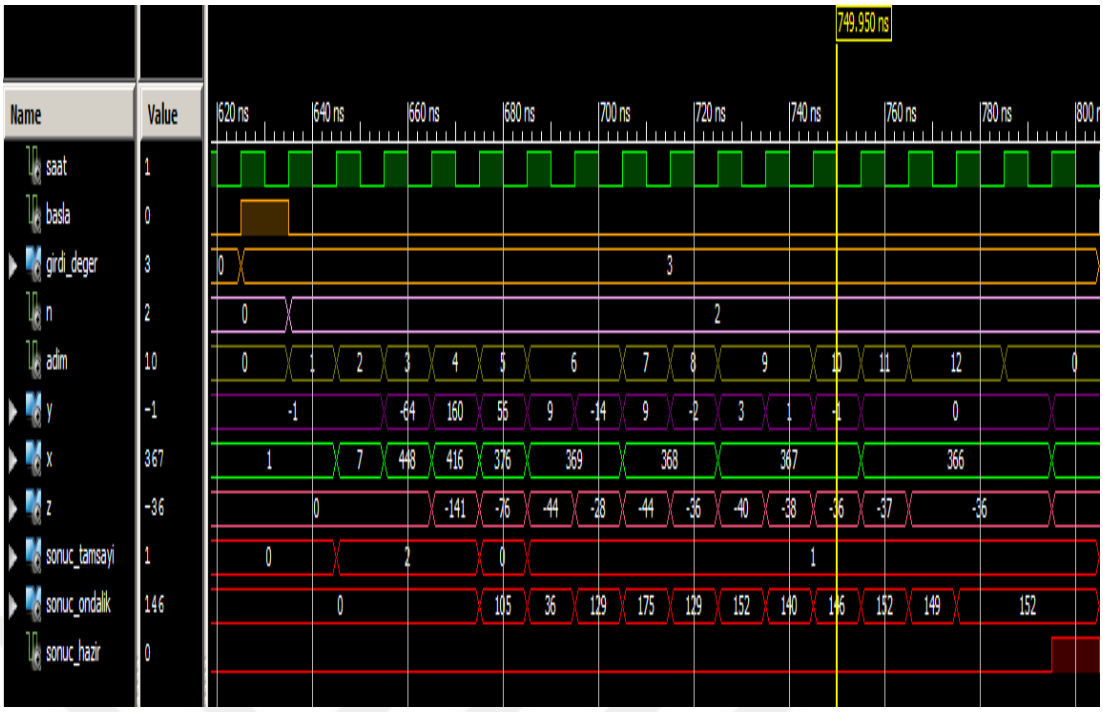
$$\mathrm{S} = \frac{|1-0|}{2^9} = 0.0020$$

duyarlılık değeri olan 0.0020'nin altında kalmaktadır. Bu yüzden de 00000000 ile ifade edilmiştir.

İCTLÇ'nin modül olarak giriş ve çıkışları VHDL dilinde şu şekildedir:

```
entity cordic_top is
port(
        saat            : in std_logic;
        basla           : in std_logic;
        girdi_deger     : in std_logic_vector(data_width-1 downto 0);
        sonuc_tamsayi   : out std_logic_vector(log2dw-1 downto 0);
        sonuc_ondalik   : out std_logic_vector(data_width-1 downto 0);
        sonuc_hazir     : out std_logic
);
end cordic_top;
```

Yukarıda verilen modül başlığında(module entity) data_width aritmetik ve girdi bit sayısını ifade etmektedir ve bu benzetimde 8'dir; log2dw ise data_width değerinin 2 tabanında logaritma değeridir ve bu benzetimde 3'tür. Yukarıda belirtilen saat işareti işlem periyodunu belirlemektedir ve saat işaretinin her bir yükseliş anında gerekli işlemler yapılmaktadır. Donanım ve işlem karmaşıklığı saat hızını düşürmektedir. Bütün donanım saat işaretine senkronize şekildedir. basla işareti ise İCTLÇ'nin başlama anını belirtmektedir ve saat işaretine senkrondur. girdi_deger işareti logaritması çevrilmek istenen girdi sayı değeridir ve 8-bit ile ifade edilmektedir. Girdi değeri 8 bit ifade edilip maksimum $2^8 - 1 = 255$ değerini alabileceği için çevrim sonucunun tamsayı kısmının maksimum değeri 7 olabilmektedir. Bundan dolayı sonuc_tamsayi işareti 3 bit ile gösterilmiştir. sonuc_ondalik ise sonucun ondalıklı kısmını belirtir. sonuc_hazir işareti ise İCTLÇ logaritmik çevrimi bitirdiğini belirtir ve saat işaretine senkrondur. İCTLÇ için 8-bit aritmetikte yapılan simulasyon sonucunda Şekil 3.7 elde edilmiştir.



3.7: 3 değeri için 8-bit Aritmetikle Yapılan Benzetim Sonucu

Şekil 3.7'den de anlaşılacağı gibi girdi olarak 8-bitlik 3 (00000011) değeri İCTLÇ'ye sunulmuştur. Ardından ilk saat döngüsünde (3.12)'deki n değeri hesaplanmıştır ve 2 olarak bulunmuştur. Ardından bir saat döngüsü sonra (3.13)'teki x_1 ve y_1 değerleri hesaplanmıştır. k=3 ve n=2 için (3.13)'e göre,

$$x_1 = k + 2^n = 3 + 2^2 = 7$$
$$y_1 = k - 2^n = 3 - 2^2 = -1$$

şeklinde bulunmuştur. Ardından bulunan (x_1,y_1) değerleri ÇAB bloğuna girmiş ve bir saat döngüsü sonra (3.8),(3.9) ve (3.14)'e göre,

$$x_b = x_1 * 2^l = 7 * 2^6 = 448$$
$$y_b = y_1 * 2^l = -1 * 2^6 = -64$$

şeklinde olmaktadır. Ardından elde edilen (x_b, y_b) değerleri CORDIC bloğuna girmiş ve ilk yinelemede (14) ve (15)'e göre,

$$\begin{cases} x_2 = [x_1 + d_1 2^{-1} y_1] = [448 + 2^{-1}(-64)] = 416 \\ y_2 = [y_1 + d_1 2^{-1} x_1] = [(-64) + 2^{-1} 448] = 160 \\ z_2 = z_1 - d_1 \tanh^{-1}(2^{-1}) = 0 - 141 = -141 \end{cases}$$

şeklinde olmuştur. d_1 değeri (2.15)'e göre y_1'in negatif olmasından dolayı 1'dir. Yine aynı şekilde ikinci yinelemede,

$$\begin{cases} x_3 = [x_2 + d_2 2^{-2} y_2] = [416 - 2^{-2}(160)] = 376 \\ y_3 = [y_2 + d_2 2^{-2} x_2] = [160 - 2^{-2} 416] = 56 \\ z_3 = z_2 - d_1 \tanh^{-1}(2^{-2}) = -141 + 65 = -76 \end{cases}$$

olmuştur.

Fakat [6]'da belirtildiği üzere CORDIC algoritmasının hiperbolik koordinat sisteminde sonuca yakınsaması için bazı yinelemelerin tekrarlanması gerekmektedir. Bunlar (3k+1) şeklinde olan yinelemelerdir. Bu durumda, İÇTLÇ'nin içindeki CORDIC bloğu normal yinelemelerine ilaveten 4,7,… yinelemeleri bir kez daha yapmaktadır. Şekil-18'de, *adim* adlı işaret İCTLÇ'nin kaçıncı adımda olduğunu göstermektedir. CORDIC bloğu öncesi 3 adım olduğu için CORDIC bloğu *adim* değerinin 2 olduğu yerden itibaren başlamaktadır. CORDIC bloğunda toplam 10 yineleme olduğu için 4,7 ve 10. adımlar ikişer kez yapılmıştır. Bu yüzden de Şekil 3.7'de *adim* işareti iki kez 6, 9 ve 12 olmuştur. İCTLÇ'nin son adımında ise CORDIC bloğundan çıkan çıktı değerinin doğal logaritma tabanında olmasından dolayı e tabanından 2 tabanına geçiş yapılmaktadır. CORDIC bloğunun çıkışında Şekil3.7'e göre z değeri -36 olmaktadır. (3.10)'e göre

$$\log_2 z = \log_2 e * \ln(z)$$

çarpımı yapılmalıdır. Yine önceden hesaplanmış $\log_2 e$ değeri hafıza sabit bir işaret olarak tutulur. $\log_2 e$ değerine (0,2) aralığını kapsayacak 9-bitlik sayısallaştırma uygulanacak olursa,

$$\log_2 e = 1.4427 = 101110001$$

şeklinde olur. Burada 9-bitlik sayısallaştırma uygulanmasının sebebi bu sayısallaştırmanın (0,2) aralığını kapsadığı için (0,1) aralığını kapsayan 8-bitlik sayısallaştırma ile eşit çözünürlük ve sayısallaştırma adım boyu (quantization step size) sahip olmasını sağlamaktır. Buna göre bu benzetimde, e tabanından 2 tabanına geçiş,

$$\log_2 z = \log_2 e * \ln(z) = -52$$

olmaktadır. Bulunan bu değer İCTLÇ'nin son adımındaki, tamsayı kısmı ile toplama girer ve İCTLÇ'nin çıktısı hesaplanmış olur. Şekil-18'de görüleceği üzere sonuc_hazir işaretinin 1olduğu yerde İCTLÇ doğru çıktıyı sunmaktadır. Sonuçta 3'ün 2 tabanında logaritması için İCTLÇ tamsayı olarak 1 ondalık sayı olarak 152 vermiştir. Bu durumda, İCTLÇ tarafından hesaplanmış değer,

$$\log_2(\text{girdi}) = \text{sonuc\_tamsayi} + \frac{sonuc\_ondalik}{2^d} \tag{3.19}$$

şeklinde hesaplanır ve,

$$\log_2(3) = 1 + \frac{152}{2^8} = 1.5938$$

olarak bulunur. Bu sayıyı 3 sayısının 2 tabanındaki logaritmasının gerçek değeri olan ile kıyaslanacak olursa 0.0088 hata miktari olduğu gözlemlenir. Efektif bit sayısı (ENOB),

$$\text{ENOB} = \text{d} + \log_2 d - \log_2\left(\frac{hata\ miktari}{R}\right) \tag{3.20}$$

şeklinde hesaplanır. Bu benzetimde 3 girdisi için,

$$\text{ENOB} = 8 + 3 - \left\lceil \log_2\left(\frac{0.0088}{0.0039}\right)\right\rceil = 9$$

olur. Böylelikle İCTLÇ'nin çıktısı olan 11 bitin en üst basamak 9 biti anlamlı iken en alt basamak iki biti ise anlamsızdır.

Bir başka değer olarak girdiye 120 verildiğinde yapılan benzetim Şekil 3.8'dedir.

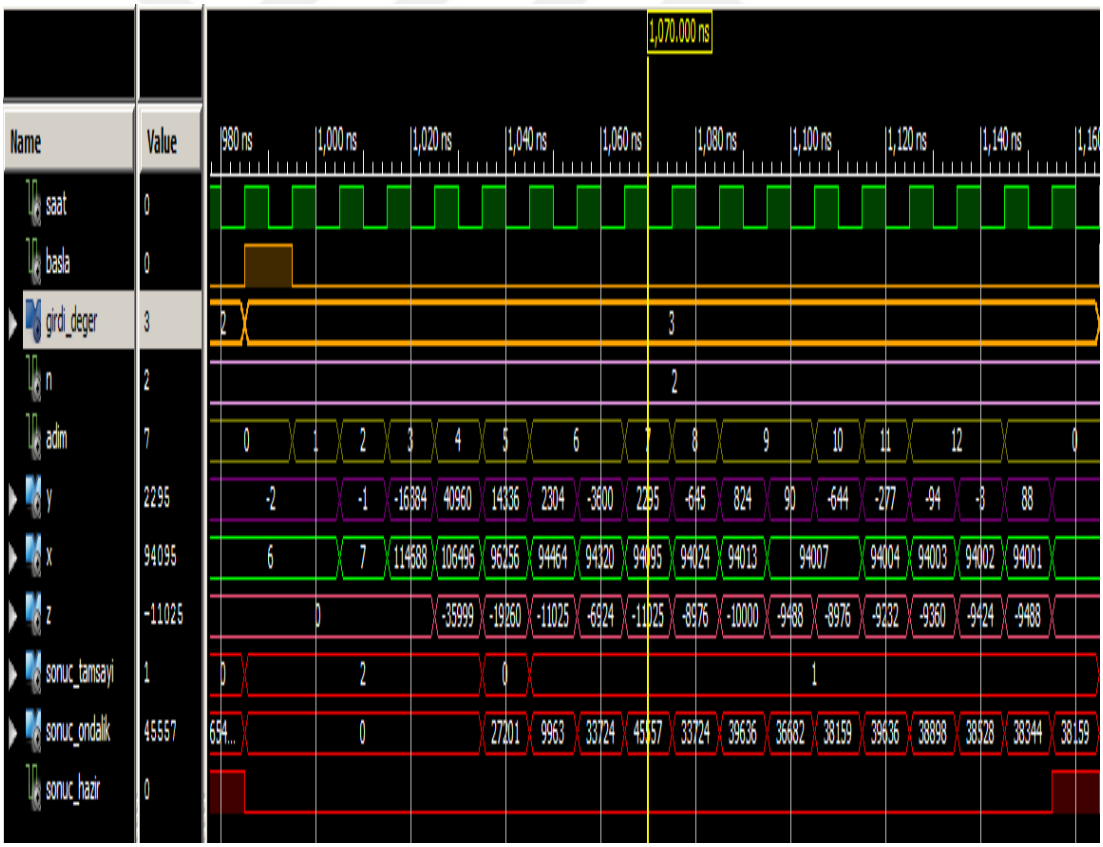
Şekil 3.8: 8-bit aritmetikte yapılan 120 değeri için benzetim sonucu

Şekil 3.8' göre İCTLÇ'nin çıktısında tamsayı olarak 6, ondalıklı kısım olarak 232 sayısı verilmiştir. 120 sayısı için İCTLÇ'nin çıktısı

$$\log_2(120) = 6 + \frac{232}{2^8} = 6.9063$$

olarak bulunmuştur. Bu değeri 120'nin 2 tabanındaki logaritmasının gerçek değeri olan 6.9069 ile karşılaştırıldığında hata miktarı olarak 0.0006 olduğu görülür. Bu değer duyarlılıktan daha düşük olduğu için hatalı bit oluşmaz ve bütün bitler anlamlı hale gelir. ENOB ise 11 olarak gerçekleşir. Bu benzetimden de anlaşılacağı üzere 8-bitlik aritmetikte CORDIC bloğunun 9.yinelemeyi yapmasından sonra çözünürlükten kaynaklı CORDIC algoritmasında durma meydana gelmektedir. Çünkü hafızada tutulan $\tanh^{-1}(2^{-i})$ değerleri 0'a düşmüş ve z değerinin her yinelemede aynı kalmasına neden olmuştur.

Bir başka benzetim çalışması ise 16-bitlik aritmetik için yapılmıştır.



Şekil 3.9: 3 değeri için 16 bit aritmetikte yapılan benzetim çalışması

16-bitlik aritmetikte çözünürlük artarken, duyarlılık düşmektedir. Aynı zamanda işlem ve donanım karmaşıklığı da arttığı için saat hızı da düşmektedir. İCTLÇ için önceden hazırlanıp hafızada tutulan $\tanh^{-1}(2^{-i})$ değerleri 16-bitlik aritmetikte,

$\tanh^{-1}(2^{-1})$ = 1000110010011111= 35999
$\tanh^{-1}(2^{-2})$ = 0100000101100011= 16739
$\tanh^{-1}(2^{-3})$ = 0010000000101011=8235
$\tanh^{-1}(2^{-4})$ = 0001000000000101=4101
$\tanh^{-1}(2^{-5})$ = 0000100000000001= 2049
$\tanh^{-1}(2^{-6})$ = 0000010000000000= 1024
$\tanh^{-1}(2^{-7})$ = 0000001000000000= 512
$\tanh^{-1}(2^{-8})$ = 0000000100000000=256
$\tanh^{-1}(2^{-9})$ = 0000000010000000= 128
$\tanh^{-1}(2^{-10})$ = 0000000001000000= 64

olmuştur. Bu değerlerden de anlaşılacağı üzere, aritmetik bit sayısındaki artış çözünürlükte ve duyarlılıkta da artma sağladığı için $\tanh^{-1}(2^{-i})$ hafızasındaki her bir eleman ayırt edici ve anlamlı olmuştur. Şekil 3.9'da 16-bitlik aritmetikte 10 yinelemeli İCTLÇ donanımı için yine girdi olarak 3 değerinin benzetim grafiği verilmiştir. Şekil 3.9'a bakıldığında, İCTLÇ'nin çıktısında tam kısımda 1, ondalıklı kısımda ise 38159 değerleri gözükmektedir. Bu durumda (3.19)'a göre İCTLÇ'nin sonucu hesaplandığında,

$$\log_2(3) = 1 + \frac{38159}{2^{16}} = 1.5823$$

olarak bulunur. Daha sonra 16-bitlik aritmetik için (3.16)'ya göre çözünürlük değeri hesaplandığında,

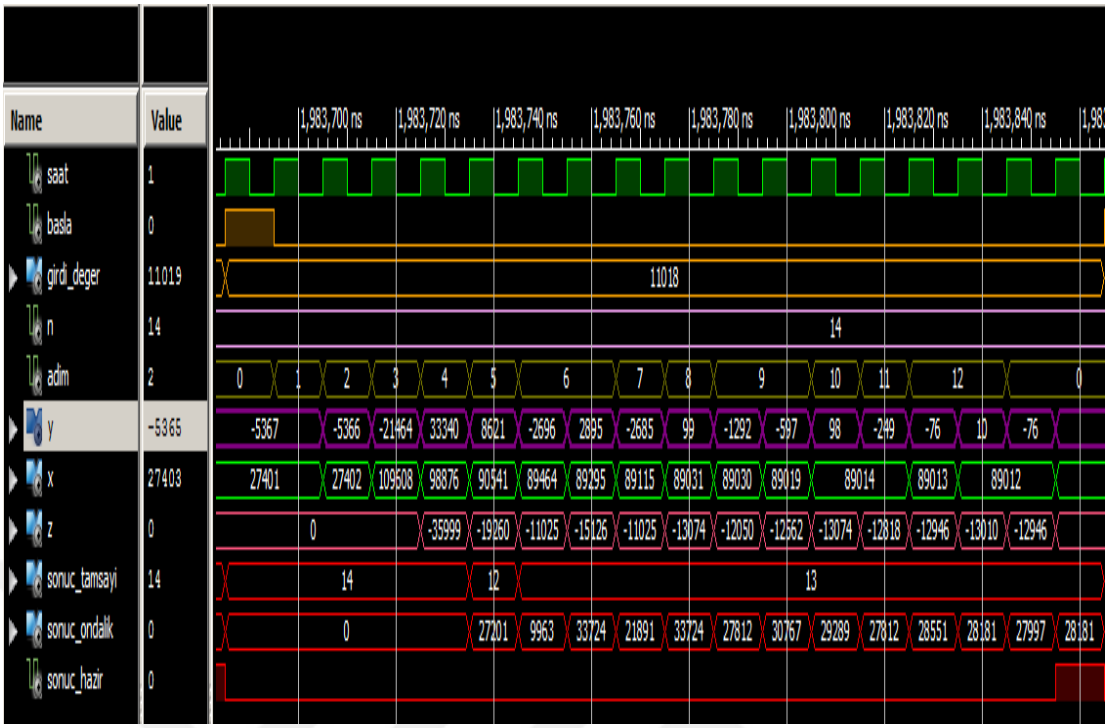
$$R = \frac{|1-0|}{2^{16}} = 1.5259e\text{-}05$$

değeri bulunur. 3 değerinin 2 tabanındaki logaritmasının gerçek değeri olan 1.5850 ile İCTLÇ'nin sonucu arasındaki hata miktarı ise 0.0027 olmaktadır. Bu duruma göre ENOB hesaplandığında ise,

$$\text{ENOB} = 16 + 4 - \left\lceil \log_2\left(\frac{0.0027}{1.5259e-05}\right)\right\rceil = 12$$

olmaktadır. ENOB'a göre girdi olarak 3 verildiğinde İCTLÇ'nin sonucunda verilen 20 bitin 12'si anlamlı iken 8'i anlamsızdır.

Şekil 3.10'da ise İCTLÇ'ye girdi olarak sunulan 11018 değerinin benzetim sonuçları verilmiştir. Şekil 3.10'a bakıldığında, İCTLÇ'nin çıktısında tam kısımda 13, ondalıklı kısımda ise 28181 değeri gözükmektedir.



Şekil 3.10 11018 değeri için benzetim

Bu durumda (3.19)'a göre İCTLÇ'nin sonucu hesaplandığında,

$$\log_2(11018) = 13 + \frac{28181}{2^{16}} = 13.43$$

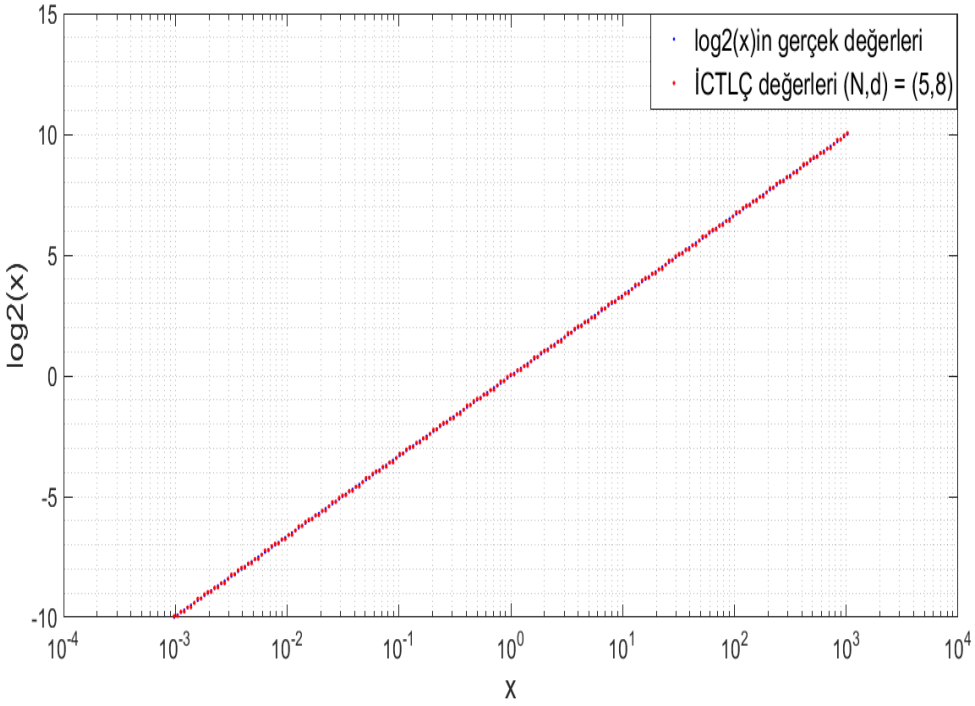
olarak bulunur. 11018 değerinin 2 tabanındaki logaritmasının gerçek değeri olan 13.4276 ile İCTLÇ'nin sonucu arasındaki hata miktarı ise 0.0024 olmaktadır. Bu duruma göre ENOB hesaplandığında ise,

$$\text{ENOB} = 16 + 4 - \left\lceil \log_2\left(\frac{0.0024}{1.5259e-05}\right)\right\rceil = 12$$

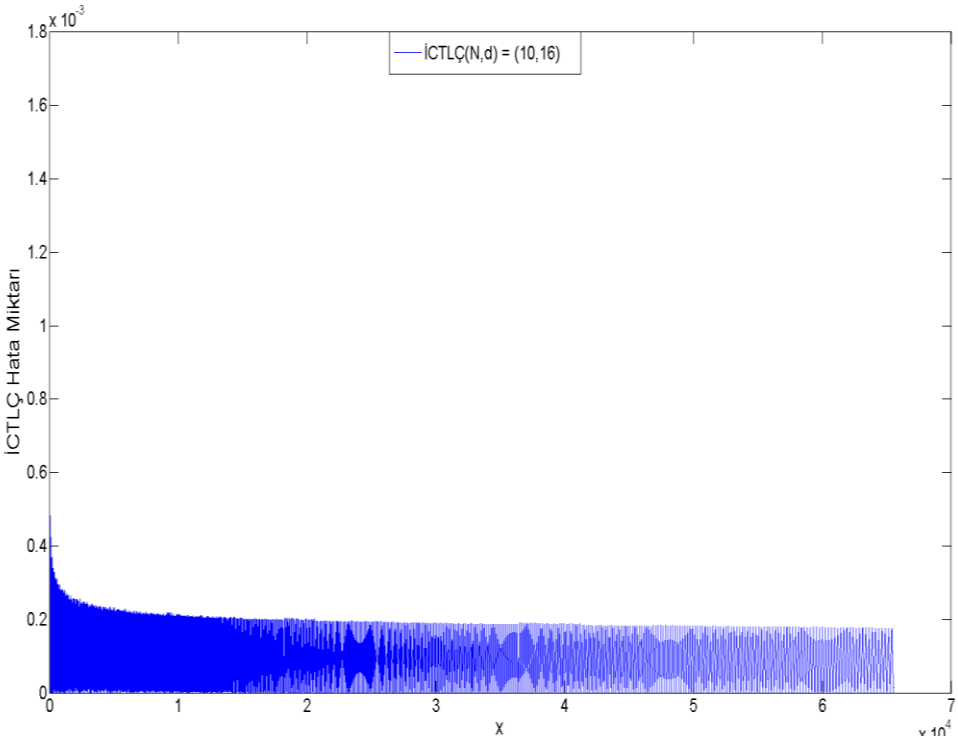
olduğu bulunur. ENOB'a göre girdi olarak 11018 verildiğinde İCTLÇ'nin sonucunda verilen 20 bitin 12'si anlamlı iken 8'i anlamsızdır.

4. ANALİZ VE KIYASLAMA

İCTLÇ ile yüksek miktarda LUT'lara ve öncül hesaplara gerek kalmadan çok az bir kaynak ve öncül bilgi kullanarak, yüksek hassasiyetli ve hızlı logaritma çevirici FPGA mimarisi oluşturulmaya çalışılmıştır. Bu mimariyi aynı zamanda yüksek bir operasyon alanında (operating range) çalışabilmesi istenmiştir. İCTLÇ sayesinde $[1, \infty)$ aralığındaki tüm sayılar d-bit ile ifade edilerek 2 tabanında logaritması bulunabilir. Eğer bir başka tabanda logaritması alınmak istenirse İCTLÇ'nin çıktısı olan değer, istenilen tabana sadece bir çarpma işlemi ile çevrilebilir. İCTLÇ'nin arkasına eklenecek bir çarpma bloğu ile bütün tabanlarda logaritma almak mümkün olmaktadır. Böylelikle, İCTLÇ logaritma fonksiyonunun tanımlı olduğu her aralıkta çalışma kapasitesindedir. $(0,1)$ aralığındaki sayıların logaritma çevrimi yapılmak istenirse, istenilen sayının çarpmaya göre tersi alınarak $[1, \infty)$ aralığına aktarılır ve aktarılan değer İCTLÇ'ye girdi olarak verilir ve İCTLÇ'nin çıktısındaki değerin toplamaya göre tersi alınarak istenilen logaritma çevirme işlemi tamamlanır. Böylelikle, logaritma fonksiyonun tanım kümesindeki bütün pozitif reel sayıların logaritma çevirimi İCTLÇ ile sağlanmış olur. Şekil 4.1'de İCTLÇ'nin 8-bitlik aritmetikte 5 yinelemeli sonuç değerleri ile 2 tabanında logaritma çevriminin gerçek değerleri verilmiştir. Şekil 4.1 ve Şekil 3.1'den de anlaşılacağı üzere CORDIC algoritmasının limit sorunu İCTLÇ sayesinde aşılmıştır. Doyuma uğrayan CORDIC algoritması İCTLÇ'nin içindeki AİB bloğu sayesinde bütün sayılara uygulanabilmektedir. Şekil 4.2'de İCTLÇ için 16-bit aritmetikte 10 yineleme için hata miktarları verilmiştir. Şekil 4.2'den anlaşılacağı üzere İCTLÇ sayesinde hata miktarı sabit bir aralık içinde kalmakta farklı girdi değerleri için fazla değişmemektedir



Şekil 4.1: 8-bitlik aritmetikte 5 yinelemeli sonuç değerleri



Şekil 4.2: İCTLÇ için 16-bit aritmetikte 10 yineleme için hata miktarları

Farklı d-bit değerleri için İCTLÇ'nin kaynak kullanım ve hız çizelgesi Çizelge 4.1'deki gibidir. Çizelge 4.1'deki N değeri CORDIC algoritmasındaki yineleme adım sayısıdır. N değerinin değişmesi kaynak kullanımını ve saat hızını değiştirmez iken,

sadece gecikmede doğrusal bir etki yapmaktadır. İCTLÇ'nin doğruluk hassasiyetine bakıldığında ise yineleme sayısı(N) ve sayısal sayı temsil bit-genişliği olan d değeri ile orantılı olduğu Çizelge 4.2'de görülmektedir. Yineleme sayısı arttıkça CORDIC bloğundan çıkan çıktının doğruluk hassasiyeti arttığı için İCTLÇ'nin de doğruluk hassasiyeti artar. d değerinin büyümesi ise daha yüksek çözünürlüklü aritmetik işlemler yapmaya olanak sağladığı ve kaydedilen LUT değerlerinin doğruluk hassasiyetini arttırdığı için İCTLÇ'nin de doğruluk hassasiyet performansı artmıştır.

Çizelge 4.1: İCTLÇ için kaynak ve hız değerleri

operasyon alanı 2^d	Kaynak ve Hız Değerleri				
	Flip-Flop	LUT	DSP	Saat Hızı(MHz)	Gecikme
2^8	62	387	-	203	$\frac{4N}{3}+3$
2^{16}	116	683	-	166	$\frac{4N}{3}+3$
2^{24}	142	1346	-	157	$\frac{4N}{3}+3$
2^{32}	213	4426	-	128	$\frac{4N}{3}+3$

Farklı d-bit değerleri için İCTLÇ'nin kaynak kullanım ve hız tablosu Çizelge 4.1'deki gibidir. Çizelge 4.1'deki N değeri CORDIC algoritmasındaki yineleme adım sayısıdır. N değerinin değişmesi kaynak kullanımını ve saat hızını değiştirmez iken, sadece gecikmede doğrusal bir etki yapmaktadır. İCTLÇ'nin doğruluk hassasiyetine bakıldığında ise yineleme sayısı(N) ve sayısal sayı temsil bit-genişliği olan d değeri ile orantılı olduğu Çizelge 4.2'de görülmektedir. Yineleme sayısı arttıkça CORDIC bloğundan çıkan çıktının doğruluk hassasiyeti arttığı için İCTLÇ'nin de doğruluk hassasiyeti artar. d değerinin büyümesi ise daha yüksek çözünürlüklü aritmetik işlemler yapmaya olanak sağladığı ve kaydedilen LUT değerlerinin doğruluk hassasiyetini arttırdığı için İCTLÇ'nin de doğruluk hassasiyet performansı artmıştır.

Çizelge 4.1 ve Çizelge 4.2'den anlaşılacağı üzere RMS hata oranı, N ve d değerlerinin artması ile azalmaktadır. Fakat N'in artması gecikmede artmaya neden olur iken, d'nin artması ise kaynak kullanımını arttırmaktadır. Bu yüzden uygulanmak istenen operasyon noktasına göre N ve d değerlerinin seçilmesine İCTLÇ, önemli bir esneklik

sağlamaktadır. Ayrıca Çizelge 4.1'de görülmektedir ki İCTLÇ'nin çıktısında alınacak sayının anlamlı bit sayısı (N,d) ikilisinin değişmesi ile 23.74'e kadar ulaşmaktadır.

Çizelge 4.2 : İCTLÇ için doğruluk ve hassasiyet

N, d	RMS Hata Oranı(%)	Kesirli Kısım Efektif Bit Sayısı
5,8	6.5916	3.92
5,24	0.2273	8.78
10,8	0.2443	8.67
10,16	0.0115	13.08
10,24	0.0073	13.74
20,16	0.0003	18.34
20,24	7.14e-6	23.74

İCTLÇ ile büyük hafıza bloğuna ihtiyaç duymadan, FPGA içindeki Flip-Flop(FF) ve LUT kaynaklarını kullanarak yüksek hassasiyetli logaritma çevrimi yapılabilmektedir. LUT tabanlı logaritma çeviriciler ile kıyaslandığında hafızaya(RAM) ihtiyaç duymadan daha az FPGA kaynak kullanımıyla bunu başarmaktadır. Örneğin [5]'deki Tablo-4'e göre 16.68 bit doğruluğunu sağlamak için 415 FF,210 LUT ve 6656 RAM elemanı ihtiyacı görülür iken İCTLÇ ile 18.34'lük bit doğruluğu 142 FF ve 1346 LUT elemanı ile sağlanmaktadır. [4]'daki polinom tabanlı logaritma çeviricisiyle kıyaslandığında, parabolik sentez metodu 15 bit doğruluğunu 481 LUT ve 31 çarpıcı ile sağlarken, İCTLÇ 13.08 bit doğruluğunu 103 FF ve 737 LUT elemanı ile çarpıcı kullanmadan başarabilmektedir. CORDIC tabanlı diğer metotlar ile kıyaslandığında, operasyon alanını genişletmek için diğer CORDIC tabanlı metotlar negatif indisli yinelemeleri kullanırken[7], İCTLÇ CORDIC algoritmasının çalışma alanına indirgeme işlemini tek işlem saatinde, yineleme yapmadan az kaynak kullanımı ile yapmaktadır. Böylelikle, İCTLÇ diğer CORDIC tabanlı metotlara göre daha az gecikme ile yüksek hassasiyetli sonuç vermektedir. Ayrıca ÇAB ile girdi olan bazı sayılarda CORDIC metodundaki aritmetik işlemlerin hızlı bir şekilde sönümlenmesini engellemekte, yinelemelerin düzgün bir şekilde tekrarlanmasını sağlamaktadır. Yüksek bit hassasiyetini daha az kaynak kullanarak sunma, operasyon noktasına göre kaynak kullanımı-gecikme değiş-tokuşu(trade-off) imkânı verme, büyük hafıza

alanlarına ve çarpıcılara ihtiyaç duymama gibi özellikleri ile İCTLÇ, diğer metotlara üstünlük sağlamaktadır.

5. SONUÇ

Herhangi bir tabanda logaritma çevrimi günümüzde kontrol ve sinyal işleme alanlarında önemli bir kullanım alanına sahiptir. Özellikle hızlı, bit doğruluğu yüksek ve az kaynak kullanımlı logaritma çeviricileri gerçek zamanlı uygulamalarda sıkça talep edilmektedir. Bu tezde önerilen İCTLÇ, CORDIC ve indirgeme metotlarını kullanan yüksek hassasiyetli bir logaritma çeviricisi yaklaşımıdır. İCTLÇ'nin yeniliği ise içerisinde CORDIC algoritmasını ve yüksek hızlı indirgeme metodunu birlikte barındırarak hem yüksek operasyon alanı hem de yüksek bit doğruluğu sunmasıdır. Bunları yaparken de hiç çarpıcı ve hafıza elemanı kullanmadan, diğer metotlara göre daha az kaynak kullanımı ile başarabilmektedir.

KAYNAKLAR

[1] **Lee J. Pottier E.,** "Polarimetric Radar Imaging: From Basics to Applications", CRC Press, February 2009.

[2] **Alvis W.** ," Development of an FPGA Based Autopilot Hardware Platform for Research and Development of Autonomous Systems", Phd thesis, March 2008.

[3] **J. N. Mitchell,** "Computer multiplication and division using binary logarithms," IRE A**.** G. M. Strollo, D. De Caro,and N. Petra., "Efficient Logarithmic Converters for Digital Signal Processing Applications"IEEE Trans. Electron. Comput., vol.58, No.10,Oct.2011.

[4] **S.Paul, N. Jayakumar, and S. P. Khatri**", A Fast Hardware Approach for approximate, Efficient Logarithm and Antilogarithm Computations" IEEE Trans.on very large scale int. Systems, Vol.17,no.2, February 2009.

[5] **J. S. Walther** "A unified algorithm for elementary functions" Spring Joint Computer Conference, 1971.

[6] **S.Wang, Y.Shang, H.Ding, C.Wang and J. Hu**" An FPGA Implementation of the Natural Based on CORDIC Algorithm" Research Journal of Applied Sciences and Technology 6(1): 119-122, 2013.

[7] **P.Pouyan, E.Hertz, P.Nilsson**," A VLSI Implementation of Logarithmic and Exponential Functions Using a Novel Parabolic Synthesis Methodology Compared to the CORDIC Algorithm 20th European Conference on Circuit Theory and Design (ECCTD)"2011.

[8] https://openi.nlm.nih.gov/detailedresult.php?img=PMC3545561_sensors-12-13150f4&req=4

[9] https://www.mathworks.com/help/fixedpoint/ug/calculate-fixed-point-arctangent.html?requestedDomain=www.mathworks.com

EKLER

A. MATLAB KODLARI

a. İCTLÇ Script

```
clc;clear all;close all;
total_iter = 10;
data_width = 16;
byte_amount = 1+data_width/8;
baud_rate = 230400;

%% FPGA ARCTANH ROM GENERATION
atanh(1./(2.^(1:total_iter)'))
rom_data = round((2^data_width)*atanh(1./(2.^(1:total_iter)')))
rom_data_bin = dec2bin(rom_data,data_width)

fid = fopen('D:\VHDL_ARGE\CORDIC\arctanh_rom_data.txt', 'w');

for i = 1 : total_iter
   for j = 1 : data_width
      fprintf(fid,'%c',rom_data_bin(i,j));
   end
   fprintf(fid,'\n');
end

fclose(fid);
clear fid;

w=(3:2^data_width-1)';

x_array= zeros(length(w),1);
y_array= zeros(length(w),1);

for i=1:length(w)
   x_array(i) = 2^(floor(log2(w(i)))+1)+w(i);
   y_array(i) = w(i)-2^(floor(log2(w(i)))+1);
```

```
end
z_array=zeros(length(x_array),1);

mult=1;
for k=1: length(x_array)
   z = 0;
   x = x_array(k);
   y = y_array(k);
   for i=1: total_iter
      if rem(i,3) == 1 && i > 3
         for t= 1:2
            xtemp=x;
            ytemp=y;
            ztemp=z;
            u= 2*(y >= 0)-1;
            x= x - ytemp*u/(2^i);
            y= y - u*xtemp/(2^i);
            z= ztemp +   u*round((2^data_width)*atanh(1/(2^i)));
         end
      else
         xtemp=x;
         ytemp=y;
         ztemp=z;
         u= 2*(y >= 0)-1;
         x= x - ytemp*u/(2^i);
         y= y - u*xtemp/(2^i);
         z= ztemp +   u*round((2^data_width)*atanh(1/(2^i)));
      end
   end
   z_array(k,1) = z;
end

computing_value= zeros(length(w),1);
for i=1:length(w)
   computing_value(i) =
floor(log2(w(i)))+1+log2(exp(1))*2*z_array(i)/(2^data_width);
end

real_value = log2(w);

error_array = abs(computing_value-real_value);
rms_error = sqrt(mean(error_array.^2));
semilogx(w,real_value,'.b');
hold on;
% figure;
semilogx(w,computing_value,'.r','LineWidth',10);

grid minor
ylabel('log2(x)','fontsize',20);
xlabel('x','fontsize',20);
```

```
% legend('log2(x)in gerçek değerleri','CORDIC algoritması sonucu','fontsize',20);
legend('log2(x)in gerçek değerleri','İCTLÇ değerleri (N,d) = (5,8)','fontsize',20);

figure;plot(w,error_array);
ylabel('İCTLÇ Hata Miktarı','fontsize',20);
xlabel('x','fontsize',20);
legend('İCTLÇ(N,d) = (10,16)','fontsize',20);
```

a. Mercator Serileri Script

```
clc;clear all;close all;

n=4;
step=0.01;

x=(-1+step:step:1)';
len_x = length(x);
y=zeros(len_x,n);

for i=1:n
    y(:,i) = (-1)^(i+1).*(x.^i)./i;
end

res = zeros(len_x,n);

res(:,1) = y(:,1);
for i=2:n
    res(:,i) = res(:,i-1) + y(:,i);
end
figure;
plot(x+1,res(:,1),'.-b');hold on;
plot(x+1,res(:,2),'.-g');hold on;
plot(x+1,res(:,3),'.-c');hold on;
plot(x+1,res(:,4),'.-k');hold on;
hold on;plot(x+1,log(1+x),'.-r');
xlabel('Değer(x)','fontname','Arial','fontsize',20);
ylabel('log(x)','fontname','Arial','fontsize',20);
legend('Mercator Serisi n=1','Mercator Serisi n=2','Mercator Serisi n=3','Mercator
Serisi n=4','Gerçek Değerler','fontname','Arial','fontsize',20);

figure;
plot(x+1,100*abs((res(:,1)-log(1+x))./log(1+x)),'.-b');hold on;
plot(x+1,100*abs((res(:,2)-log(1+x))./log(1+x)),'.-g');hold on;
plot(x+1,100*abs((res(:,3)-log(1+x))./log(1+x)),'.-r');hold on;
plot(x+1,100*abs((res(:,4)-log(1+x))./log(1+x)),'.-k');
xlabel('Değer(x)','fontname','Arial','fontsize',20);
ylabel('Hata Oranları (%)','fontname','Arial','fontsize',20);
legend('Mercator Serisi n=1','Mercator Serisi n=2','Mercator Serisi n=3','Mercator
Serisi n=4','fontname','Arial','fontsize',20);
```

B. VHDL KODLARI

a. İCTLÇ Top Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.cordic_package.all;

use IEEE.NUMERIC_STD.ALL;

entity cordic_top is
generic(
		UTILIZATION_MODE: integer:=0 -- 0 => ln(x) , 1=> arctan(y/x)
);
port(
		clk: in std_logic;
		start: in std_logic;
		x: in std_logic_vector(data_width-1 downto 0);
		y: in std_logic_vector(data_width-1 downto 0);
		int_result: out std_logic_vector(log2dw downto 0);
		decimal_result: out std_logic_vector(data_width-1 downto 0);
		ready: out std_logic
);
end cordic_top;

architecture Behavioral of cordic_top is

component full_logarithm is
port(
		clk: in std_logic;
		accumulator: in std_logic_vector(data_width downto 0);
		int_part: in integer range 0 to data_width;
		result_reg: out std_logic_vector(log2dw+2*data_width downto 0)
);
end component;

signal x_reg: std_logic_vector(data_width-1 downto 0) := (others => '0');
signal ready_reg: std_logic;

signal int_part: integer range 0 to data_width; -- logarithm decimal part
signal first_one_indicator: std_logic_vector(data_width-1 downto 0);-- first one of the
bit vector indicator

signal repeat_flag: std_logic;

signal numerator: std_logic_vector(data_width+1 downto 0);
signal denumerator: std_logic_vector(data_width+1 downto 0);

signal numerator_extended: std_logic_vector(2*data_width+1 downto 0);
```

```
signal denumerator_extended: std_logic_vector(2*data_width+1 downto 0);

signal accumulator: std_logic_vector(data_width downto 0);
signal process_counter: integer range 0 to iteration_amount+2;

signal result_reg: std_logic_vector(log2dw+2*data_width downto 0);
signal result_reg2: std_logic_vector(log2dw+2*data_width downto 0);

signal base2accumulator: std_logic_vector(2*data_width+1 downto 0);

attribute keep : string;
attribute keep of x_reg: signal is "true";
attribute keep of ready_reg: signal is "true";
attribute keep of int_part: signal is "true";
attribute keep of first_one_indicator: signal is "true";
attribute keep of repeat_flag: signal is "true";
attribute keep of numerator: signal is "true";
attribute keep of denumerator: signal is "true";
attribute keep of numerator_extended: signal is "true";
attribute keep of denumerator_extended: signal is "true";
attribute keep of accumulator: signal is "true";
attribute keep of process_counter: signal is "true";
attribute keep of result_reg: signal is "true";
attribute keep of result_reg2: signal is "true";
attribute keep of base2accumulator: signal is "true";

begin

int_result <= result_reg(log2dw+2*data_width downto 2*data_width);
decimal_result <= result_reg(2*data_width-1 downto data_width);

full_logarithm_inst: full_logarithm
port map(
                clk => clk,
                accumulator => accumulator,
                int_part => int_part,
                result_reg =>result_reg
);

numerator_extended <= numerator & all_zeros;
denumerator_extended <= denumerator & all_zeros;

process(clk)
begin
        if rising_edge(clk) then
                case process_counter is
                when 0 =>
                        if start = '1' then
                                process_counter <= process_counter +1;
                        else
```

```
                        process_counter <= process_counter;
                    end if;
                when iteration_amount+2 =>
                    if repeat_flag = '1' then
                        process_counter <= process_counter;
                    else
                        process_counter <= 0;
                    end if;
                when others =>
                    if repeat_flag = '1' then
                        process_counter <= process_counter;
                    else
                        process_counter <= process_counter +1;
                    end if;
                end case;
        end if;
end process;

process(clk)
begin
        if rising_edge(clk) then
                if start = '1' then
                    repeat_flag <= '0';
                else
                    if (process_counter rem 3 = 2) and process_counter > 4 then
                        repeat_flag <= '1';
                    else
                        repeat_flag <= '0';
                    end if;
                end if;
        end if;
end process;

process(clk)
begin
        if rising_edge(clk) then
                if start = '1' then
                    int_part <= vector_and_sum_operate(x);
                    x_reg <= x;
                else
                    int_part <= int_part;
                    x_reg <= x_reg;
                end if;

                if process_counter < 2 then
                    numerator                                                   <=
std_logic_vector(to_signed(to_integer(unsigned(x_reg))-
to_integer(unsigned(one_selection_array(int_part))),data_width+2));
```

```
                        denumerator <=
std_logic_vector(to_signed(to_integer(unsigned(x_reg))+to_integer(unsigned(one_se
lection_array(int_part))),data_width+2));
                        accumulator <= (others => '0');
                elsif process_counter = 2 then
--                      numerator <= numerator(int_part+1 downto 0) &
all_zeros(data_width-int_part-1 downto 0);
--                      denumerator <= denumerator(int_part+1 downto 0) &
all_zeros(data_width-int_part-1 downto 0);
                        numerator <= numerator_extended(data_width+int_part+1
downto int_part);
                        denumerator <= denumerator_extended(data_width+int_part+1
downto int_part);
                else
                        if numerator(data_width+1) = '1' then
                                numerator <=
std_logic_vector(to_signed(to_integer(signed(numerator))+to_integer(signed(denum
erator(data_width+1) & denumerator(data_width+1 downto process_counter-
2))),data_width+2));
                                denumerator <=
std_logic_vector(to_signed(to_integer(signed(denumerator))+to_integer(signed(num
erator(data_width+1) & numerator(data_width+1 downto process_counter-
2))),data_width+2));
                                accumulator <=
std_logic_vector(to_signed(to_integer(signed(accumulator))-
to_integer(unsigned(atanh_rom_data(process_counter-3))),data_width+1));
                        else
                                numerator <=
std_logic_vector(to_signed(to_integer(signed(numerator))-
to_integer(signed(denumerator(data_width+1) & denumerator(data_width+1 downto
process_counter-2))),data_width+2));
                                denumerator <=
std_logic_vector(to_signed(to_integer(signed(denumerator))-
to_integer(signed(numerator(data_width+1) & numerator(data_width+1 downto
process_counter-2))),data_width+2));
                                accumulator <=
std_logic_vector(to_signed(to_integer(signed(accumulator))+to_integer(unsigned(ata
nh_rom_data(process_counter-3))),data_width+1));
                        end if;
                end if;
        end if;
end process;

process(clk)
begin
        if rising_edge(clk) then
                ready <= ready_reg;
                if start = '1' then
                        ready_reg <= '0';
                else
```

```
                        if process_counter = iteration_amount+2 and repeat_flag = '0'
then
                                ready_reg <= '1';
                        else
                                ready_reg <= '0';
                        end if;
                end if;
        end if;
end process;

end Behavioral;
```

b. full_logarithm module

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use work.cordic_package.all;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_SIGNED.ALL;

entity full_logarithm is

port(

                clk: in std_logic;

                accumulator: in std_logic_vector(data_width downto 0);

                int_part: in integer range 0 to data_width;

                result_reg: out std_logic_vector(log2dw+2*data_width downto 0)

);

end full_logarithm;

architecture Behavioral of full_logarithm is


signal accumulator_reg: std_logic_vector(data_width downto 0);
```

```
signal int_part_reg: integer range 0 to data_width;
signal result_reg_buf: std_logic_vector(log2dw+2*data_width downto 0);
begin
process(clk)
begin
        if rising_edge(clk) then
                result_reg_buf <= (CONV_STD_LOGIC_VECTOR(int_part,log2dw+1) & all_zeros & all_zeros) + ('0' & log2e) * (accumulator & '0');
        end if;
end process;
result_reg<= result_reg_buf;
end Behavioral;
```

ÖZGEÇMİŞ

Ad-Soyad **:**Gülfem HELVACIOĞLU

Uyruğu **:**TC

Doğum Tarihi ve Yeri **:**1990 / K.Maraş

E-posta **:**ghelvacioglu@etu.edu.tr

ÖĞRENİM DURUMU:

- **Lisans** **:**2015, TOBB ETÜ, Mühendislik Fakültesi, Elektrik Elektronik Mühendisliği
- **Yüksek lisans** **:**2017, TOBB ETÜ, Fen Bilimleri Anabilim Dalı, Elektrik Elektronik Mühendisliği Tezli Yüksek Lisans Programı

YABANCI DİL: İngilizce

Almanca

TEZDEN TÜRETİLEN YAYINLAR, SUNUMLAR VE PATENTLER:

- **Helvacıoğlu. G.**,Korucu A., Alp Y. ve Kasnakoğlu C., 2017.İndirgenmiş CORDIC Tabanlı Logaritma Çeviricisi, Sinyal İşleme ve Uygulamaları(SİU) Konferansı Mayıs 15-18 Antalya/Türkiye