T.C
SELÇUK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ALAN PROGRAMLAMALI KAPI DİZİLERİ (FPGA) ÜZERİNDE BİR YSA'NIN TASARLANMASI VE DONANIM OLARAK GERÇEKLEŞTİRİLMESİ

NECLA YILMAZ

YÜKSEK LİSANS TEZİ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

KONYA, 2008

T.C
SELÇUK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

ALAN PROGRAMLAMALI KAPI DİZİLERİ (FPGA) ÜZERİNDE BİR YSA'NIN TASARLANMASI VE DONANIM OLARAK GERÇEKLEŞTİRİLMESİ

NECLA YILMAZ

YÜKSEK LİSANS TEZİ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

Bu tez 21/ 07 /2008 tarihinde aşağıdaki jüri tarafından oybirliği/oyçokluğu ile kabul edilmiştir.

Prof.Dr.Şirzat KAHRAMANLI	Yrd.Doç.Dr. Fatih BAŞÇİFTÇİ	Yrd.Doç.Dr. Ömer Kaan BAYKAN
(Danışman)	(Üye)	(Üye)

ÖZET

Yüksek Lisans Tezi

ALAN PROGRAMLAMALI KAPI DİZİLERİ (FPGA)
ÜZERİNDE BİR YSA'NIN TASARLANMASI VE DONANIM OLARAK
GERÇEKLEŞTİRİLMESİ

Necla YILMAZ

Selçuk Üniversitesi Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman : Prof.Dr.Şirzat KAHRAMANLI

2008, 77 Sayfa

Jüri : Prof.Dr.Şirzat KAHRAMANLI
Yrd.Doç.Dr. Fatih BAŞÇİFTÇİ
Yrd.Doç.Dr. Ömer Kaan BAYKAN

İnsan sinir sisteminin çalışma mekanizmasından esinlenerek gerçekleştirilen Yapay Sinir Ağları (YSA), doğrusal olmayan bir yapıya sahip olması nedeniyle pek çok uygulamada kullanılan popüler bir metottur. YSA uygulamalarında öğrenme ve test aşaması olarak iki aşama bulunur. Bu aşamalardan öğrenme aşaması oldukça karmaşık ve uzun bir süreçtir. Bu uzun süreyi kısaltmak için YSA yapısı VLSI teknolojisi kullanılarak gerçekleştirilebilir. Fakat bu yapının oluşturulması uzun zaman alır ve maliyeti oldukça yüksektir. Bu durumda VLSI ile benzer özelliklere sahip olan FPGA kullanmak, hızlı eğitilebilen YSA'lar için en uygun çözüm olacaktır. FPGA ile gerçekleştirilen tasarımlar VLSI'lara nazaran daha az maliyetli ve tekrar tekrar düzenlenebilir bir yapıya sahiptir.

Bu çalışmada, çip üzerinde eğitilebilir bir YSA yapısı, Altera FPGA devreleri ile gerçekleştirilmiştir. XOR problemi ve bir sensör doğrusallaştırma problemi ile çalışılmış ve sabit noktalı sayı sistemi tabanlı ve hatanın geri yayılımı algoritması ile eğitilen bir YSA yapısı kullanılmıştır. Öğrenme kuralı olarak delta bar delta kuralı seçilmiştir. Bu uygulamalar Altera'nın QUARTUS II FPGA tasarım programı ve MATLAB ile tasarlanmış ve simüle edilmiştir. Bunlara ek olarak, basitleştirilmiş YSA yapısı ile XOR problemi Altera Cyclone EP1C6Q240C8 FPGA tabanlı UP3 geliştirme kartı ile gerçekleştirilmiştir.

Bu çalışma ile bazı YSA tabanlı sistemler için FPGA'nın maliyet, zaman tasarrufu, tekrar düzenlenebilirlik ve paralel tasarım yeteneği açılarından daha uygun bir çözüm olduğu gösterilmiştir.

Anahtar Kelimeler: YSA, FPGA, VHDL, Sabit Noktalı Sayı

ABSTRACT

Master Science Thesis

DESIGN AND IMPLEMENTATION OF THE AN ANN STRUCTURE AS HARDWARE ON FIELD PROGRAMMABLE GATE ARRAYS (FPGA)

Necla YILMAZ

Selçuk University

Graduate School of Natural and Applied Sciences

Department of Computer Engineering

Supervisor: Prof.Dr.Şirzat KAHRAMANLI

2008, 77 Page

Jury: Prof.Dr.Şirzat KAHRAMANLI

Assist.Prof.Dr. Fatih BAŞÇİTFÇİ

Assist.Prof.Doç.Dr. Ömer Kaan BAYKAN

ANN designed with help of working mechanism of human neuron system is a popular method used in among of application because of it has nonlinear structure. There are two phase in ANN applications as test and train. The train phase is very complex and it takes a long time. For reducing of the training time, ANN can be realized with VLSI technology. A VLSI structure has high cost and long design period. In this case, FPGA circuits having similar functions with VLSI are very suitable solution for fast trainable ANN structures. Design realized with FPGA is lower cost than VLSI design and have reconfigurable structure unlike VLSI.

In this study, an trainable ANN structure on chip is realized with Altera FPGA devices. The XOR problem and a sensor linearization problem are studied in this thesis. A Back propagation algorithm based on fixed point number is used. Delta Bar Delta learning rule is selected for all application. The applications are designed and simulated in QUARTUS II FPGA design program and MATLAB. In addition to these, XOR problem with simplified ANN structure is realized on UP3 Development Board based on Altera Cyclone EP1C6Q240C8 FPGA chip.

With this study, FPGA is more suitable for some ANN based system than computer and VLSI based systems in point of cost, time saving, reconfigurable structure and parallel design ability properties is shown.

Key words: ANN, FPGA, VHDL, Fixed point numbers

ÖNSÖZ

FPGA tabanlı Yapay Sinir Ağları yapısı oldukça kullanışlı ve birçok karmaşık işlemi doğru olarak yapabilecek kapasiteye sahiptir. FPGA yazı tahtasına benzer yapısı ile kolay değiştirilebilir ve tekrar kullanılabilirdir. YSA yapısı FPGA yerine ASIC üzerinde yapılırsa dizaynın büyüklüğünün oldukça fazla olacağı bir gerçektir.

FPGA'lerin programlama kabiliyetleri mikroişlemcilere göre daha sınırlıdır. Fakat yeni FPGA versiyonları içinde mikroişlemci ve DSP bulundurabildikleri için bu dezavantajı aşmaya çalışmaktadırlar. FPGA programlamada bir çok özel program olduğu gibi CAD programları da FPGA tasarımını desteklemektedir.

Bu çalışmada bir yapay sinir ağı tasarım adımları ve simülasyon sonuçları verilmiştir. Tasarımlar ve simülasyonlar Altera Quartus II programında yapılmış, uygulama Cylone EP1C6Q240C8 devre kartı üzerinde basit bir gösterim olarak sunulmuştur.

Bu çalışmada benden yardımlarını esirgemeyen değerli hocam Prof. Dr. Şirzat KAHRAMANLI ve beni FPGA-VHDL konusunda çalışmaya iten bu çalışmalar esnasında fikirleri ile bana yardımcı olan eşime teşekkür ederim.

İÇİNDEKİLER

SİMGELER

AHDL : Altera Hardware Description Language .Altera Donanım Tanımlama Dili

ANN Artifical Neural Network

ASIC :Application Specific Integrated Circuit – Uygulamaya Özel Entegre Devreler

CLB : Configuration Logic Block . Düzenleme Lojik Bloğu(Xilink)

CORDIC : Coordinate Rotation Digital Computer .Dijital Koordinat Döndürme Hesabı

CPLD : Complex Programmable Logic Device –Karmaşık Programlanabilir Lojik Devre

E : hata değeri

FPGA : Field Programmable Gate Array .Alan Programlamalı Kapı Dizileri

HDL : Hardware Description Language.Donanım Tanımlama dili

k :Öğrenme katsayısı artma faktörü

LAB : Logic Array Block . Lojik Dizi Bloğu(Altera)

LUT : Look-up-Table . FPGA içinde yer alan danışma tablosu

MPGA : Mask Programmable Gate Array . Maske Programlanabilir Lojik Devre

SPLD : Simple Programmable Logic Device – Basit Programlanabilir Lojik Devre

VHDL : VHSIC Hardware Description Language .VHSIC Donanım Tanımlama Dili

VHSIC : Very High Speed Integrated Circuit . Yüksek Hızlı Entegre Devreler

VLSI : Very Large Scale Integration .Büyük Ölçekli Entegre Devreler

YSA : Yapay Sinir Ağları

w :Ağırlık değeri

$\alpha_{\max}$: Maksimum öğrenme katsayısı

$\mu_{\max}$: Momentum katsayısı üst sınırı

κ_{α} : Sabit öğrenme hızı skala vektörü

γ_{α} : Sabit öğrenme hızı üstel faktörü

φ_{α} :sabit öğrenme hızı azaltma faktörü

a : Lineer Fonksiyon Katsayısı

α : Momentum Katsayısı

δ : Bir nörona ait hata faktörü

ε : öğrenme katsayısı

η : Öğrenme Katsayısı

θ : Konveks ağırlık faktörü

κ : Kırpılma katsayısı

λ : Düzeltme tolerans parametresi

μ :Momentum

τ : Lineer Fonksiyon İçin Üst eşik Değeri

$-\tau$: Linner Fonksiyon için Alt Eşik Değeri

φ :Öğrenme katsayısı azalma faktörü

1 GİRİŞ

İnsanlığın doğayı araştırma ve taklit etme çabalarının en son ürünlerinden bir tanesi yapay sinir ağları(YSA) teknolojisidir. YSA, biyolojik sinir sisteminin çalışma şeklinin simüle edilmesi ile tasarlanan programlama yaklaşımıdır. Simüle edilen sinir hücreleri (nöronlar) içerirler ve bu nöronlar çeşitli şekillerde birbirine bağlanarak bir ağ oluştururlar. Bu ağlar öğrenme, hafızaya alma ve veriler arasındaki ilişkiyi ortaya çıkarma kapasitesine sahiptirler. YSA'lar normalde insanın düşünmeye ve gözlemlemeye yönelik doğal yeteneklerini gerektiren problemlere çözüm üretmektedirler.

YSA'nın gerçekleştirimi, yapay sinir hücrelerinin donanıma haritalanması şeklindedir. YSA'nın gerçekleştirimini iki şekilde yapılabilmektedir.

Birincisi yazılım gerçeklemesidir. Yazılım gerçeklemesi ile yapılan YSA yapısı esneklik ve çeşitlilik özelliklerine sahiptir. Yazılımın dezavantajı eğitim süresi ve maliyet olmaktadır. Bu dezavantajlardan eğitim süresi donanım gerçeklemesi ile ortadan kaldırılmaya çalışılmıştır. VLSI sistemleri yüksek hız ve yoğun hesaplama isteyen uygulamalarda kullanılan bir yöntemdir ve özel bir YSA için tasarlanırlar.

VLSI ile oluşturulmuş YSA uygulamaları hız ve işlem yoğunluğu probleminin üstesinden gelmişlerdir. Fakat farklı YSA'ların aynı devreyi kullanamayışı, üretim aşamalarının hem maliyetli hem de çok zaman almasından dolayı bu tarz tümleşik devreler yaygın bir kullanım alanı bulamamışlardır.

Bu aşamada FPGA'ların kullanımı söz konusu olmuştur. FPGA'lar tasarım esnekliği sayesinde değişiklikleri birkaç saat içinde başarabilmeleri sebebiyle zamandan ve maliyetten kazanımlar sağlamıştır.

YSA yapısı öğrenme ve test aşamalarından oluşur. Her iki aşama farklı gereksinimlere ihtiyaç duyar. Öğrenme aşaması dikkatle takip edilmesi gereken karmaşık bir süreçtir. Donanımdaki ağ parametreleri bu aşamada belirlenir ve test aşamasında kullanılmak üzere donanıma yüklenir. Eğitim aşamasında kullanılan sayıların duyarlılığı ve aritmetik yoğunluğu fazla olmalıdır. Test zamanında sayı duyarlılığı düşük olabilir.

FPGA mimarisi üzerinde en etkili şekilde gerçekleştirilen ağlar sınıflandırma yapan ağlardır. Genellikle sınıflandırma uygulamasında en çok kullanılan YSA modeli ileri beslemeli ağ ile geriye yayılım öğrenme algoritmasıdır. Nöronlar ağda katmanlar halinde konumlandırılmışlardır. Bir katmanda bulunan her bir yapay sinir hücresine ulaşan girişler ya bir önceki katmandan ya da dışarıdan gelir. Test safhasında bir sinir ağı tarafından yapılan temel işlem toplamadır. Toplamadan sonra toplamlar doğrusal olmayan bir fonksiyondan geçirilirler. (Burr 1993)

FPGA'da haritalanma yapılırken paralellikten mümkün olan en büyük ölçüde yararlanılması beklenir. Yer tasarrufu ve hız artırımı için YSA yapısında kullanılacak doğrusal olmayan fonksiyonun çeşitli girişlere vereceği çıkışlar bir tablodan elde edilebilir.(Yu ve ark. 1994)

YSA'nın FPGA kullanılarak gerçeklenmesi alanında birçok çalışma bulunmaktadır. Bunlardan bazıları aşağıda verilmiştir.

Guccione ve Gonzalez (1993) geleneksel bir programlama modeli önermişlerdir. Bu model hesaplamalarda vektör tabanlı veri-paralel yaklaşımıdır. Bu model, yüksek seviyeli dillerde (örneğin C) yazılan algoritmaları, yüksek performanslı dijital devrelere dönüştürür.

Kalp hastalarının sınıflandırılması için taşınabilir dijital bir sistem prototipi geliştirilmiştir. Bu sistem eğitilmiş bir ağın Xilinx XC 4000 serisi FPGA'nın üzerinde gerçeklenmesiyle oluşturulmuştur.(Burr 1993)

YSA yapay bir beyin oluşturmak için kullanılabilecek bir yöntemdir. CAM-Brain Machine çalışması genetik algoritması ve hücresel sinir ağı ile FPGA üzerinde gerçekleştirilen yapay bir beyin modellemesidir. Modelde 72 adet FPGA ile yaklaşık 1000 nörondan oluşan bir robot kedi beyni oluşturulmuştur. Bu çalışma yapay beyin araştırmalarında gerçek zamanlı robot kontrolü için umut verici bir çalışma olmuştur.(Garisa ve ark. 1994)

E. Won oluşturduğu beş adet 8-bit desen girişi, altı gizli nod ve bir 8-bit çıkış ile 11 saat saykılında 200 ns'den az bir zamanda karar oluşturmayı başarmıştır. (Won 2007)

Prevotat ve arkadaşları LHC(Large Hadron Collider) deneyinde tetikleyici zamanını yapay sinir ağları ile uygulamışlar ve başarılı sonuçlar elde etmişlerdir. Bir

tetikleyici hesaplayan, yüzlerce nörondan oluşan YSA sistemi ile sadece 600 ns'de sonuç elde etmişlerdir. (Prlevotet ve ark 2003)

Bu tezde her bir bölümde anlatılan konular şunlardır.

Bölüm 2'de FPGA hakkında bilgiler verilmiştir. Programlanabilir lojik elemanların yapısı ve gelişim süreci, FPGA 'ın genel yapısı, FPGA çeşitleri ve devre yapısı bu bölümde anlatılmıştır.

Bölüm 3'de kullanılan Quartus II programı ile ilgili kısaca bilgi verilmiştir..

Bölüm 4'de YSA'nın gelişimi ve YSA çekici kılan özelliklerden bahsedilmiştir. Ayrıca YSA'nın matematiksel modellemesi ve ileri beslemeli YSA hakkında bilgi verilmiştir.

Bölüm 5'de YSA programlama teknikleri ve bu çerçevede VHDL dili anlatılmıştır. VHDL dilinin oluşma süreci, bu dilde kullanılan ana yapılar örnekler ile beraber anlatılmıştır.

Bölüm 6'da YSA'nın eğitiminin, VHDL kullanılarak FPGA üzerinde eğitiminin gerçeklenmesi anlatılmıştır. Gerçekleştirme esnasında kullanılan sabit noktalı sayı aritmetiği, VHDL içerisinde gösterimi ve isleyişi hakkında bilgi verilmiştir. Oluşturulan ağ yapısının özellikleri, kullanılan aktivasyon fonksiyonu ve deneysel sonuçlar bu bölümde genişçe açıklanmıştır.

2 FPGA (Alan Programlamalı Kapı Dizileri)

Programlanabilir lojik; programlanabilen bağlantılar ile birbirine bağlanan programlanabilen devreleri ifade etmektedir. Birçok farklı programlanabilir lojik devreler bulunmasına rağmen temelde hepsi aynı düşünceye sahiptir. Programlanabilir aygıtlar imaj, ses, sinyal işleme, bilgi sistemleri, kontrol, düzenleme alanlarında kullanılmaktadır. Birçok program FPGA'lara yerleştirilerek çalıştırılabilmektedir.

2.1 FPGA Tarihçesi

FPGA tanımana geçmeden önce programlanabilir lojik devrelere deyinmekte yarar vardır.

2.1.1.1 Programlanabilir Lojik

Geniş kullanım alanı bulan ilk programlanabilir lojik devre ROM (Read Only Memory) devreleridir. ROM devreleri üretici tarafından programlanan ve son kullanıcı tarafından programlanan olarak ikiye ayırmak mümkündür. Üretici tarafından programlanan ROM elamanları PROM'lardır. PROM'lar silinemeyen ve bir defa programlanabilen devrelerdir. Kullanıcı tarafından programlanabilen ROM devreleri ise EPROM ve E^2PROM devre elemanlarıdır.

2.1.2 SPLD

Bu alanda kullanılan ikinci devre programlanabilir lojik devre (PLD) çipleridir. PLD çipler mantıksal devreler oluşturmak için kullanılırlar. PLD içinde AND ve OR kapılarından oluşan diziler bulunmaktadır. Programlanabilir lojik dizi (PAL) olarak isimlendirilen aygıtlar, sabit bir OR kapıları dizisi düzlemi ile

programlanabilir ve AND kapıları dizisi içerir. PAL ufak mantıksal devrelerde kullanılmaktadır.

SPLD (Simple Programmable Logic Device) basit programlanabilen devreler olarak tanımlanırlar. Dört ile yirmi iki arasında makro hücre içerir. Programlanabilir lojik ailesinin en ufak ve en ucuz üyesidir.

2.1.3 CPLD

Complex Programmable Logic Device (CPLD), SPLD'lere göre daha fazla kapasiteye sahip olan aygıtlardır. Bir CPLD, on ile birkaç yüz makro hücre içerir. Sekizden on altıya kadar makro hücreler birleşerek daha büyük bir fonksiyon bloğu oluşturur. Genel olarak bir fonksiyon bloğu içerisinde yer alan makro hücreler birbirine tamamen bağlıdır. Hangi blokların nasıl bağlandığı üreticiye ve ait olduğu aileye göre değişmektedir. Şekil 2.2' de genel CPLD yapısı görülmektedir. CPLD, programlanabilir bir anahtarlama matrisi ile birbirine bağlanmış birden fazla PAL aygıtından oluşmaktadır. Her PAL aygıtı, mimariye bağlı olarak 4 ile 16 arası makro hücre içerir.

CPLD'ler SPLD'lere göre daha yüksek yoğunluğa sahiptirler. Pinden pine yüksek performansı sebebiyle, CPLD çipler kontrol merkezli uygulamalar için çok elverişlidir.

CPLD çipler arasında bir makro hücre içerisinde yer alan product term sayısı, bir makro hücrenin diğer makro hücreden product term alıp almaması ve anahtarlama matrisinin tamamen veya kısmen kullanılması gibi mimari farklılıklar vardır. Mimariler arasındaki diğer önemli bir fark ise anahtarlama matrisindeki bağlantı sayısıdır. Her türlü bağlantıyı sağlayabilen matris tam kullanılabilirdir. Kısmi kullanılabilir matris ise, bazı bağlantıları sağlamasa da çoğu bağlantıyı sağlar. Matrisin sağlayabileceği bağlantı sayısı, dizaynın çip içerisine yerleştirilebilme kolaylığını ve esnekliğini doğrudan etkiler. Tam kullanılabilir matriste bütün aygıt kaynakları, giriş çıkış pinleri arasında bağlantı sağlanabilir. Tam kullanılabilir mimariye sahip çiplerde, bekleme süresi sabittir ve hesaplanabilir. Kısmi kullanılabilir matrislerde kompleks dizaynlar içerisindeki yönlendirmelerde sorunlar

yaşanabilir. Hatta bu tür mimarilerde, pin çıkışlarını değiştirmeden, dizayn değişikliği yapmak çok zordur. Sabit pin çıkışına dizayn içi yönlendirme yapabilmek önemli bir konudur. Tekrar programlanabilir bir cihazın içini değiştirmek, devrenin şemasını değiştirmekten daha kolaydır. Dolayısı ile dizayn değişikliklerinde aynı devre şemasını kullanmak daha verimlidir. Kısmi kullanabilir matrislerde bekleme süresi, çoğu FPGA aygıtında olduğu gibi, sabit değildir ve önceden hesaplamak mümkün değildir. Kısmi kullanılabilir mimarinin sınırlamaları olmasına karşın üretimi daha ucuzdur.

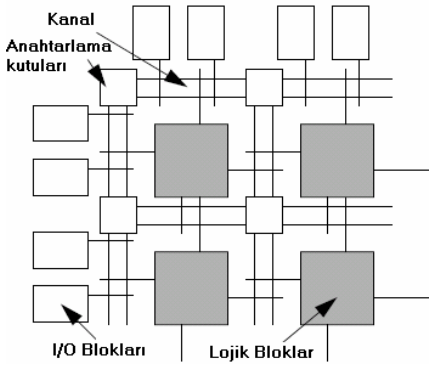
2.1.4 MPGA

Daha büyük mantıksal devreleri oluşturabilmek için Mask Programmable Gate Array (MPGA) çipler üretilmiştir. Genel bir MPGA, istenen mantıksal devreyi oluşturabilmek için birbirleri ile bağlanmış transistor satırları içerir. Kullanıcı tarafından tanımlanmış bağlantılar, hem satır içerisinde, hem satır aralarında bulunur. Bu yöntem, mantıksal kapıları oluşturabilme ve oluşturulmuş mantıksal kapıları birbirine bağlayabilme imkânı sağlar. Metal tabakalar, üretici tarafında tanımlandığı için üretim sayısı arttıkça fiyat ve zamandan kazanç sağlanır.

2.1.5 FPGA

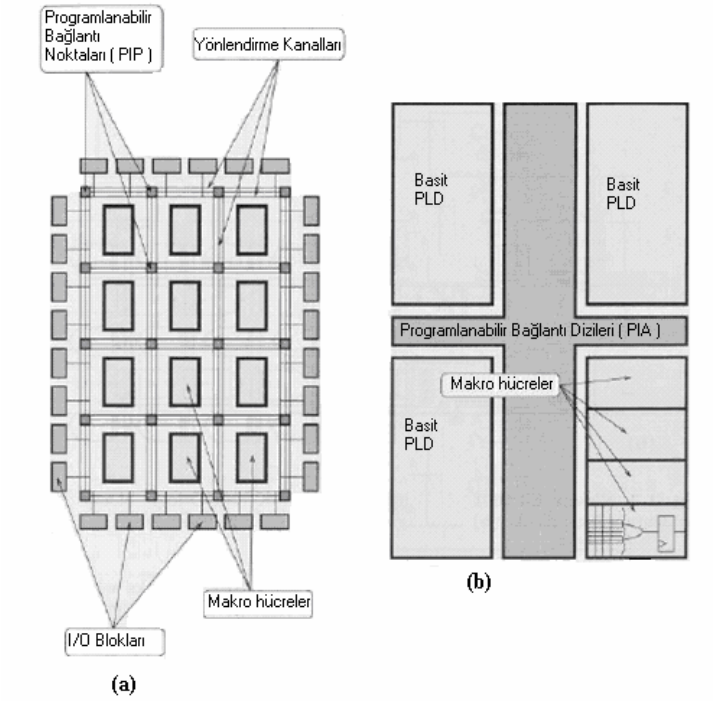
FPGA (Field Programmable Gate Array) , bir lojik blok dizisi, bu dizinin çevresinde bir halka oluşturan giriş çıkış birimleri ve bütün bu birimleri bağlayan programlanabilir ara bağlantılardan oluşan aygıttır.

FPGA kökleri 1980'li yıllarda bulunan CPLD devrelere dayanır. FPGA'lar on binden birkaç milyona kadar varan lojik kapı içerirken CPLD birkaç bin ile on bin arası lojik kapı içermektedir. Genel FPGA yapısı Şekil 2.1'de gösterildiği gibidir.



Şekil 2.1 FPGA mimarisi

CPLD ve FPGA arasında en önemli fark mimarilerindedir. CPLD az sayıda saat ile beslenen bir veya birkaç programlanabilen devre içerir. Sonuçta bu yüksek bağlantı oranı ve önceden tahmin edilebilen gecikme zamanı ile birlikte az bir esneklik verir. FPGA mimarisine bağlantılar egemendir. Bu daha esnek bir yapı ve daha karmaşık bir dizayn demektir. İkinci önemli fark ise FPGA devrelerinin içinde CPLD devrelerine göre daha fazla yüksek seviye fonksiyon ve hafıza bulunmasıdır. Şekil 2.2'de her iki devrenin yapıları görülmektedir.



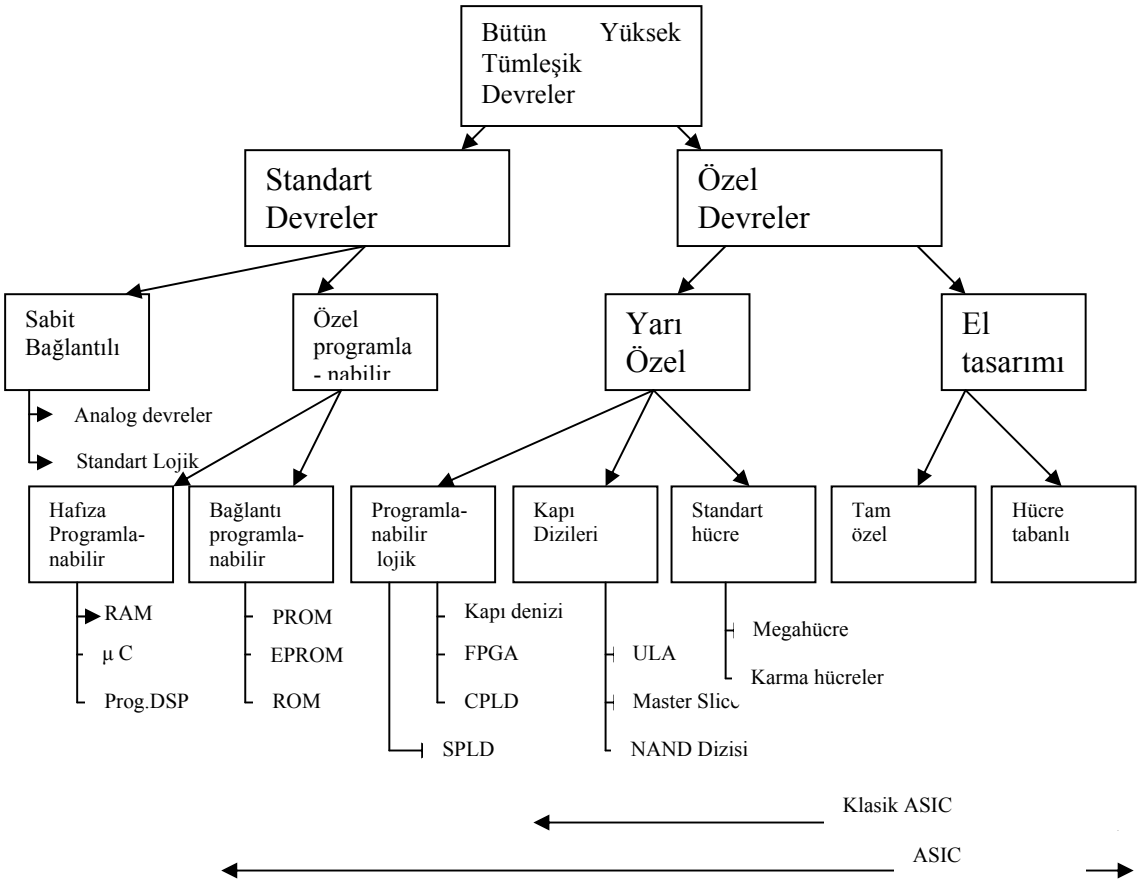
Şekil 2.2 (a) FPGA mimarisi (b) CPLD mimarisi

Başka bir dikkat çekici nokta ise FPGA'lara gömülü olan hafıza ve yüksek düzey fonksiyonlardır(çarpıcı ve toplayıcı gibi).

Şu an piyasada dört çeşit FPGA tipi bulunmaktadır. Bunlar; simetrik dizi, satır bazlı, hiyerarşik PLD ve kapı denizidir. FPGA üretiminin %80'i Altera ve Xilink firmaları tarafından gerçekleştirilmektedir.

2.2 FPGA Sınıflandırmaları

VLSI (Very Large Scale Integration) devreleri sınıflandırılması Şekil 2.3'de gösterilmektedir.(Meyer-Bease 2001) FPGA'lar bu sınıflandırmada alan programlanabilir lojik (FPL) arasında yer almaktadır. FPGA'lar ASIC (Application Specific Integrated Circuit) özel uygulama gerçekleştiren devreler içindedir.



Şekil 2.3 VLSI'a göre programlanabilir lojik sınıflandırması

2.2.1 Taneciklilik (granularity) Sınıflandırması

Bir devrenin inceliği; lojik bloklar arasındaki ilişki uzunluğu veya iki blok arasında iletişimi sağlamak için gerekli olan ilişki uzunluğu denilebilir. Üç farklı incelik sınıfı tanımlanabilir. (Meyer-Bease 2001)

- İyi tanecikli (Pilkington veya kapı denizi mimarisi)
- Orta tanecikli(FPGA)
- Büyük tanecikli(CPLD)

2.2.1.1 İyi-tanecikli Devreler

İyi-tanecikli(fine-grain) devreler ilk Plessey tarafından yapılmış ve daha sonra Motorola tarafından lisanslanmıştır. Pilkington Semiconductor tarafından kullanılmaya başlanmıştır. Bir anahtarlama ve basit NAND kapıları içerir. Çünkü NAND kapıları kullanarak birçok lojik fonksiyonu gerçekleştirmek mümkündür. NAND kapıları evrensel fonksiyonlar olarak çağrılırlar. NAND kapıları arasındaki bağlantılar metal katmanlar kullanılarak sağlanmıştır. Programlanabilir mimarilerde bu bazı darboğazlar getirmektedir. Çünkü gerçekleştirilen lojik fonksiyonlar ile birlikte oldukça fazla gönderici kaynak kullanılmaktadır. Örneğin 4-bit bir toplayıcı için 130 NAND kapısı kullanılmaktadır.

2.2.1.2 Orta-tanecikli Devreler

Birçok FPGA mimarisi orta incelikli devreler içermektedir. Lojik blok bileşenler küçük tablolar veya çoklayıcılardır. Gönderilen kanal kısadan uzuna derecelendirilip seçilirler. Flip-flop'lar ile birlikte programlanabilir giriş/çıkış blokları devrenin fiziksel sınırına bağlanır.

2.2.1.3 Büyük-tanecikli Devreler

Büyük-tanecikli devreler CPLD devreleridir. SPLD olarak da tanımlanabilirler. Evrensel giriş/çıkış blokları ve AND/OR kapılarından gerçeklenmiş programlanabilir lojik devreler (PLA) içerirler. CPLD içinde kullanılan SPLD'ler tipik olarak sekiz-on girişli, üç-dört çıkışlı ve yirmi destek üretim birimine sahiptirler. İki SPLD blok arasındaki geniş yollar kısa gecikmeleri mümkün kılar. CPLD'lerin pin-to-pin gecikmeleri kısadır ve tahmin edilebilir şartları oluşturur.

2.2.2 Teknoloji Sınıflandırmaları

Programlanabilir lojik elemanları tüm hafıza birimlerini (SRAM, EPROM, E^2PROM) kullanır. Programlanabilir ve bir kere programlanabilir olarak tanımlanabilirler. FPGA teknolojilerinde SRAM devreleri kapladıkları alanın büyük olmasına rağmen sistem üzerindeyken tekrar programlanabilmesi sebebiyle baskındırlar.

İkinci tip devre seçimi statik CMOS hafıza teknolojisi tabanlıdır ve sistem yeniden programlanabilirdir. EPROM devreleri bir kez CMOS programlamak için kullanılırlar çünkü silinmeleri için ultraviole ışınlarına ihtiyaç duyarlar. E^2PROM yeniden programlanabilen elektrikle silinebilen bir devredir. EPROM ve E^2PROM için kısa yüklenme zamanları ve elektriksel yük harcamamaları avantajdır. E^2PROM teknolojisi flash hafıza olarak çağırılmaktadır.

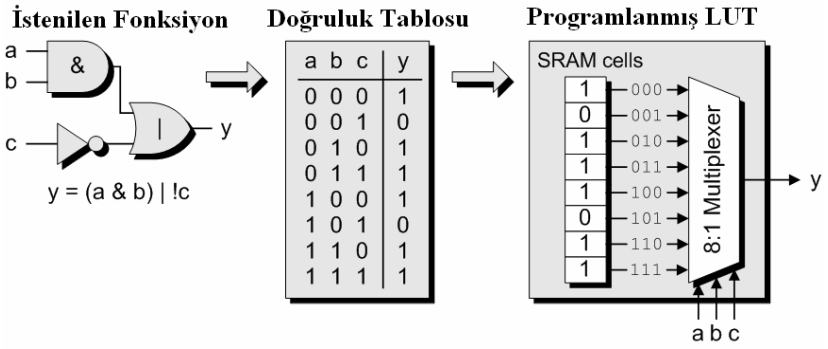
2.2.3 FPGA'ların Lojik Hücre Yapısı

FPGA'ların hücre yapısı doğruluk tablosu tabanlı veya çoklayıcı tabanlı olmak üzere iki sınıfta incelenebilir. **(**Çavuşoğlu 2006)

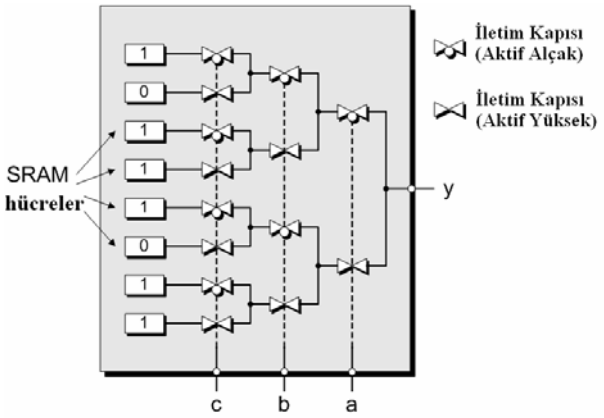
2.2.3.1 Doğruluk Tablosu Tabanlı Yapı

Doğruluk tablosu tabanlı yapının temel bloğu, LUT (Look Up Table) adı verilen ve m (m>1) değişkenli her Boolean fonksiyonunu gerçekleyen bir devredir.

Bu yapı statik RAM ile gerçekleştirilir. Şekil 2.4'de üç değişkenli bir fonksiyon verilmiştir. Bu fonksiyonun doğruluk tablosu çıkarılmıştır. Doğruluk tablosuna göre fonksiyon sonuçları SRAM hücrelerindedir. Çoklayıcı sayesinde değişken girişlere karşı olan fonksiyonun cevabı direkt olarak verilmektedir. Şekil 2.4'de kullanılan çoğullayıcı basitleştirilmiş gösterimdir. Şekil 2.5'de programlanmış LUT'un daha gerçekçi modeli gösterilmiştir.



Şekil 2.4 Bir fonksiyona göre düzenlenmiş LUT'un basit gösterimi

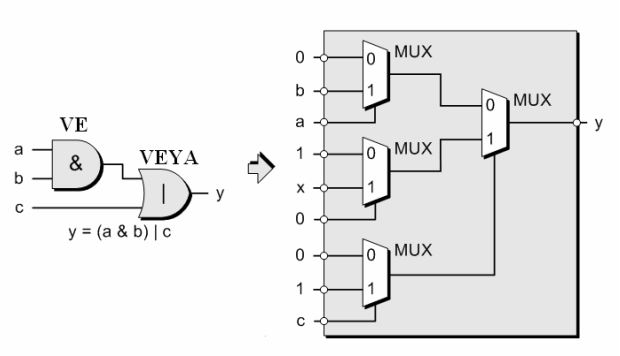


Şekil 2.5 Programlanmış LUT'un daha gerçekçi gösterimi

2.2.3.2 Çoklayıcı Tabanlı Yapı

Çoklayıcı tabanlı yapının temel bloğu çeşitli konfigürasyonlardan ve olabildiğince az VE ve VEYA gibi lojik kapılardan oluşur. Bu yapıdaki FPGA'ların içinde latch ve flip-flop bulunmadığından çoklayıcı kullanılarak bu elemanların

gerçeklenmesi gerekmektedir. Şekil 2.6'da bir lojik fonksiyonun çoklayıcı tabanlı yapı kullanılarak gerçeklenmesi gösterilmiştir.



Şekil 2.6 Bir lojik fonksiyonun çoklayıcı tabanlı yapı kullanılarak gerçeklenmesi

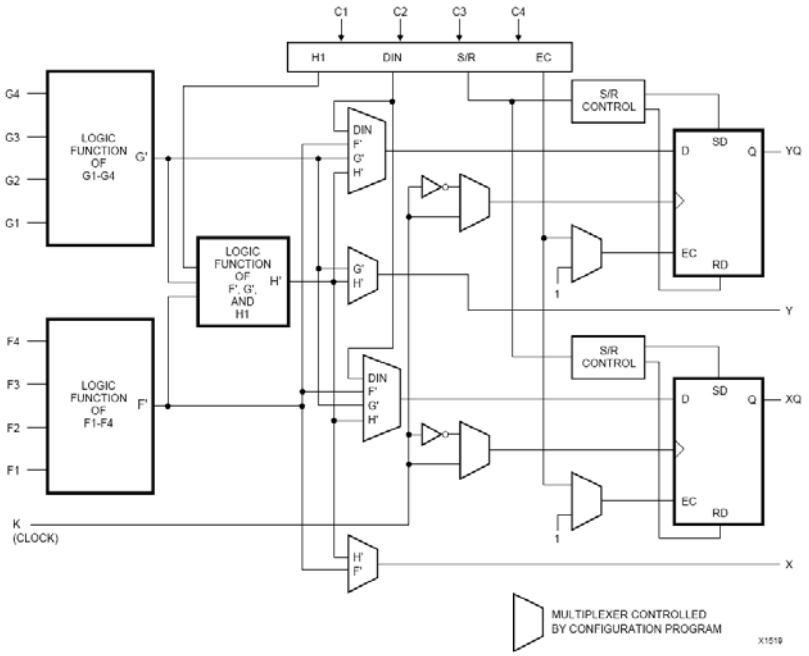
2.3 FPGA Mimarileri

Xilink ve Altera tarafından üretilen FPGA mimarileri bazı yönlerden farklılık içerirler. Üretim temelinde Xilink XC4000 devre ailesi ile Altera FLEX10K devre ailesi bulunmaktadır

Xilink XC4000'de bulunan her bir CLB Şekil 2.7'de gösterildiği gibi ikiye ayrılmış dört giriş –bir çıkış LUT (look-up table) , elde, ikiye ayrılmış üç giriş bir çıkışlı bir toplayıcı ve iki D tipi flip flop içermektedir. Xilink XC4000 toplam 20 adet CLB bloğu içermektedir. Her CLB bloğu single-port 16x1 veya dual-port 32x1 lik RAM alanına sahiptir.

Diğer bloklar pin tamponları, ayarlanabilen sonlandırıcılar ve giriş/çıkış kaydedicileri içeren giriş/çıkış (IOB-Input Output Block) bloklarıdır. 4000 ailesi iki ile dokuz giriş için hızlı kod çözücüler içerir. 4000 ailesi iç üç durum sinyalleri ve yollardan destek almaktadır.

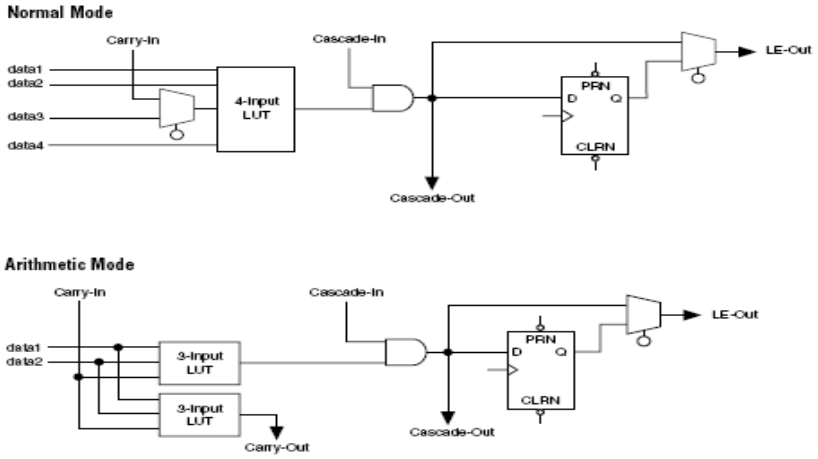
Xilink devreleri SRAM yönlendirmeleri ve metal katmanlarla dizi anahtarlamaları kullanır. Bu daha az silikon ve daha iyi bir zamanlama demektir. Donanım olarak Xilink programcıları daha az yormaktadır.



Şekil 2.7 XC4000 Xilink CLB yapısı

Xilink sonraları Virtex serilerini piyasaya sürmüştür. Virtex serilerinin en sonuncusu şu anda prototipi sunulan Virtex-5 LXT (4 Aralık 2006) devresidir. Virtex-5 LX330T devresi CLB biriminde 240x108 dizi uzunluğuna, 51.840 bölüme, 331.776 lojik hücreye, 207.360 flip fop'a, hafıza kaynakları toplam 11.664 Kbit RAM bloğuna, 960 adet giriş/çıkış pinine sahip 65 nm teknolojisi ile üretilmiştir. Her bir Virtex-5 CLB bloğu dört 6-LUT ve dört flip flop içeren iki bölümden oluşmuştur.

Altera devrelerinin başlangıcında FLEX (Flexible Logic Element matriX) serileri gelmektedir. FLEX serileri orta tanecikli küçük LUT'lar kullanılarak yapılmıştır. SRAM tabanlıdır. Altera lojik bileşenlerine Logic Element adı verilir. Her bir LE bir flip flop, dört giriş bir çıkışlı veya üç giriş bir çıkışlı LUT ve elde bulundurmaktadır. Sekiz logic element birleşerek bir Logic Array Block (LAB) oluştururlar. Xilink ile karşılaştırıldığında LAB'lar arası daha az bağlantı söz konusudur.



Şekil 2.8 FLEX lojik bileşen devresi

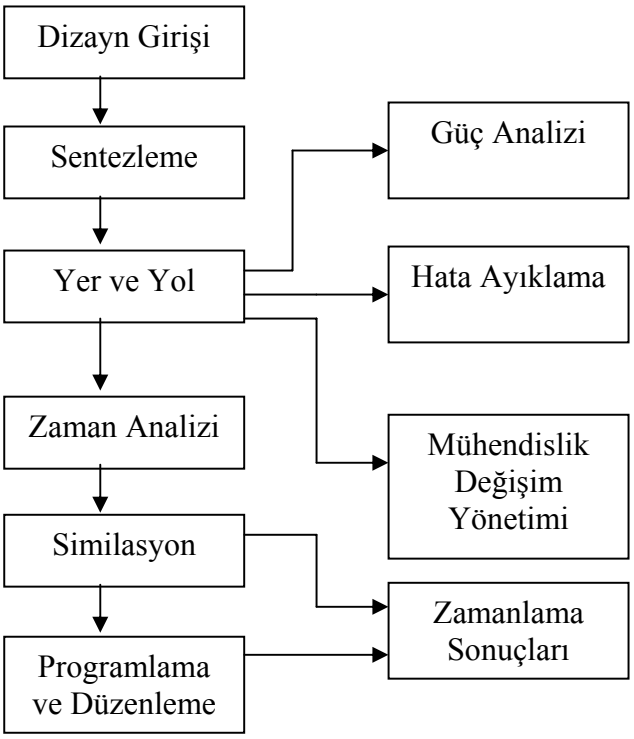
Altera APEX, FLEX, MAX, CYCLONE, STRATIX gibi devre aileleri piyasaya sürmüştür En son Stratix III (Kasım 2006) serisini üretmiştir. Cyclone daha ucuz devreleri tercih edenler için üretilmiştir. Stratix III 47.500 ile 338.000 lojik bileşen(LE) , 1.836 ile 17.208 Kbit hafıza, 288 ile 1.104 giriş/çıkış pini, 216 ile 576 18x18 çoklayıcı içeren 65 nm teknolojisi ile üretilmiştir.

FPGA yapılarını karşılaştırıldığında Xilink daha çok lokal bağlantı daha az global bağlantı içerir. Altera geniş bağlantı yollarına sahiptir. Başka bir açıdan karşılaştırmada ise Xilink ‘in donanımı, Altera’nın yazılımı daha iyidir diye bir ifade kullanılmaktadır(Alan 1993). Altera yazılımları daha anlaşılır ve daha kolay kullanılabilirdir.

Nios Altera FPGA içinde bulunan işlemcinin adıdır. Hazırlanan dizaynlar lojik bloklar tarafından derlenirken diğer programlar (C ++ , Assembler ,) Cyclone EP1C6Q240C8 içinde bulunan iki adet (master ve slave) nios işlemci tarafından derlenir. İşlemci üzerinde ayarlar Nios programı ile olmaktadır.

3 QUARTUS II YAZILIMI

Altera Quartus II yazılımı system-on-programable-chip (chip üzerinde programlanabilir sistem) dizaynını mümkün kılan bir yazılımdır. Quartus II tam olarak yüklendiğinde birçok farklı platformda özel dizaynları gerçekleştirme imkânı sağlar. Quartus II FPGA ve CPLD dizaynları için çözüm içeren bir yazılımdır. Aşağıda Quartus II dizayn oluşturma akış şeması Şekil 3.1'da görülmektedir.



Şekil 3.1 Quartus II Dizayn Akış Şeması

3.1 Quartus II Derleme Adımları

Quartus II derleme işlemi aşağıdaki adımları içerir. Her bir adım Quartus II arayüzünde görülebilmektedir. Adımlardan * işareti ile belirtilmiş olanlar seçmelidir.

- Analysis & Synthesis
- Partition Merge*
- Fitter
- Assembler *
- Classic Timing Analyzer and TimeQuest Timing Analyzer *
- EDA Netlist Writer*
- HardCopy Netlist Writer*

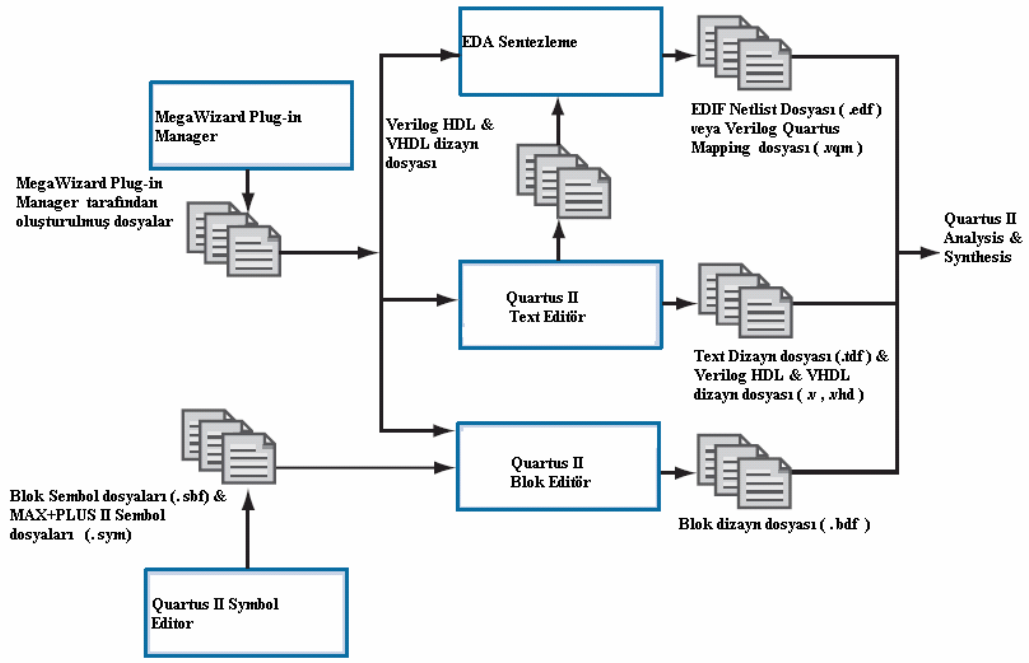
Derleme işlemi Compile menüsünden Start Compile ile başlar ve adımlar Compile Tool arayüzü ile gösterilir.

Quartus II iki farklı arayüz imkânı tanır. Quartus II yazılımı içinde MAX+PLUS II yazılımı da bulunmaktadır. Hangisinin kullanılacağı kullanıcıya kalmıştır ve Tools→Customize ekranından belirlenebilir. MAX+PLUS II yazılımı Altera şirketinin ilk çip üzerinde programlanabilir sistem dizaynı yazılımıdır. MAX+PLUS II yazılımı kullanmak yerine sadece hızlı menüleri de kullanılabilir. Hızlı menüler ana menünü sol tarafında yer alırlar ve tüm araçların kısa yollarını içerirler.

Quartus II yazılımı genel olarak üç tür yazılımı giriş olarak almaktadır. Quartus II aşağıda bulunan üç tür dosya ile çalışma imkânı tanır.

- **Devre dizayn dosyaları (Device Design Files);** AHDL (Altera HDL), blok diyagramı şematik, VHDL ve Verilog HDL dosyalarıdır. Devre dizayn dosyaları FPGA içine bir devre yerleştirmek için kullanılır.
- **Yazılım dosyaları (Software Files);** Assembly ,C ,C/C++ ve C++ dosyalarıdır.Yazılım dosyaları FPGA içindeki Nios işlemci tarafından derlenir ve çalıştırılırlar.
- **Diğer dosyalar (Other Files) ;** AHDL içeren , blok sembol , hexadecimal, text ,hafıza başlangıç,.. gibi FPGA çalışmasında ek dosyaları hazırlamak için kullanılır.

Quartus II dosya giriş ve sentezleme adımları Şekil 3.2'de gösterilmektedir.



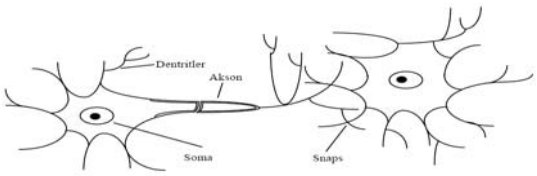
Şekil 3.2 Quartus II Giriş ve Sentezleme Adımları

Quartus II dizayn girişleri herhangi bir HDL dili veya blok sembol ile olmaktadır. Quartus II Block Editor kullanışlı bir devre çizme arayüzüdür. Ayrıca HDL kodlarının RTL Editor ile devre yapısını görebilir, FloorPlan Editor ile FPGA içindeki devre yerlerini değiştirebilir, Assigment Editör ile pin yönlendirmelerini yapabiliriz.

4 YAPAY SİNİR AĞLARI

4.1 Biyolojik Sinir Sistemi

İnsan sinir sistemi sürekli bilgi alan, bu bilgileri yorumlayan ve uygun kararlar veren karmaşık bir yapı halindedir. İnsan sini sisteminde iki tür sinir bulunur. Alıcı sinirler algıladıkları bilgileri beyine elektrik sinyallerine dönüştürerek iletirler. Tepki sinirleri ise beynin ürettiği kararları vücuda iletmektedir. Merkezi sinir ağında , bilgiler alıcı ve tepki sinirleri arasında ileri ve geri besleme yönünde değerlendirilerek uygun tepkiler üretilir. Merkezi sinir sisteminin temel elamanı sinir hücresidir ve nöron olarak isimlendirilirler. Sinir hücresi; hücre gövdesi, dendritler ve aksonlar olmak üzere 3 bileşenden meydana gelirler.



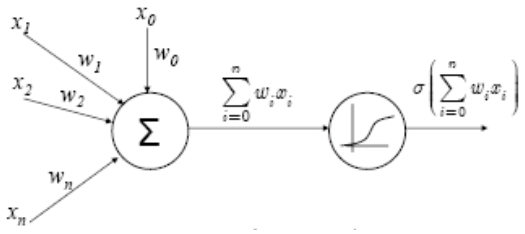
Şekil 4.1 İnsan sinir hücresi yapısı

Akson ve dendritler arası bağlantı yerine snaps adı verilir. Snaps'a gelen ve dendritler tarafından alınan bilgiler genellikle elektriksel darbelerdir ve hücre snapsdaki kimyasal ileticilerden etkilenir. Belirli bir sürede bir hücreye gelen girişlerin değeri, belirli bir eşik değerine ulaştığında hücre tepki üretir. Hücrenin tepkisini artırıcı yöndeki girişler uyarıcı, azaltıcı yöndeki girişler ise önleyici girişler olarak söylenir ve snaps tarafından belirlenir

4.1.1 Yapay Sinir Hücresi

İnsan sinir hücresi temel alınarak oluşturulan yapıya yapay sinir hücresi (nöron) denir. Nöron yapısında girişler toplanır ve girişler eşiği aştığı zaman hücre bir etki oluşturur. Bu yapı yapay sinir hücresinin temel mantığıdır. Yapay sinir hücresi giriş verilerini toplayan ve eşik fonksiyonuna göre bir değer üreten yapıdır.

Eşik fonksiyonu olarak sigmoid fonksiyon kullanılmış bir yapay sinir hücresi Şekil 4.2 de gösterilmiştir.



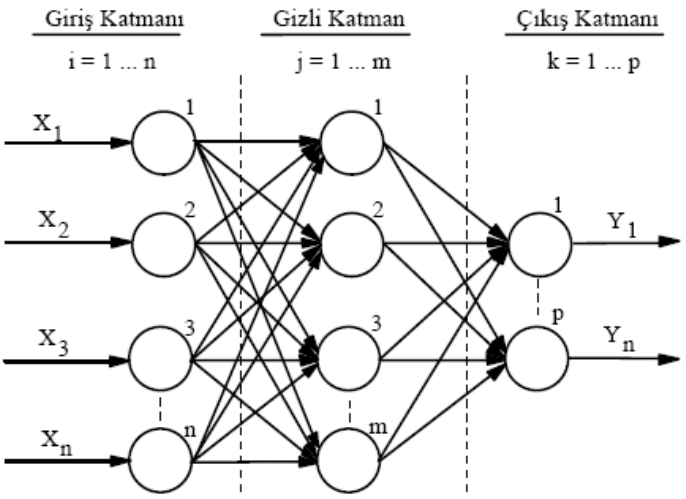
Şekil 4.2 Yapay sinir hücresi

4.2 Yapay Sinir Ağı

Beynin özellikleri dikkate alınarak yapay bir ağ oluşturulmaya çalışılmıştır. Bu matematik modellemede temel bileşeni yapay sinir hücresi oluşturmaktadır. Yapay sinir ağı birbirine bağlı yapay sinir hücreleridir ve genelde katmanlar halinde düzenlenir. Donanım olarak elektronik devrelerle veya yazılımlarla gerçeklenebilir. YSA, öğrenme sonrası bilgiyi toplama, hücreler arası ağırlıkları ve bu bilgiyi depolama ve genelleme yeteneğine paralel dağılmış bir işlemcidir. Öğrenme süreci istenilen sonuca ulaşmak için ağırlıkların yenilenmesidir.

YSA işlemleri son derece hızlı bir şekilde yapabilmesine rağmen insan beyni ile yarışabilecek düzeyden oldukça uzaktır. Fakat karmaşık eşleştirmelerde ve veri sınıflandırılmasında başarılı sonuçlar vermektedirler.

YSA'da bulunan her düğüm, n. Dereceden tercihen lineer olmayan bir birimdir. Düğümler arasında bağlantılar bulunmaktadır. Her bağlantı tek yönlü iletim yoludur. Bir düğüm birden fazla düğüme veri aktarabilir. İşlenilen bilgiler (sonuç belirleme süreci) bir sonraki katmandaki bir veya birden fazla düğüme iletilir. Çok katmanı bir YSA Şekil 4.3'te gösterildiği gibidir.



Şekil 4.3 Çok katmanlı Yapay Sinir Ağı

4.3 Yapay Sinir Ağlarının Özellikleri

YSA'nın en önemli özelliği dağıtılmış paralel yapısı, öğrenebilme ve genelleme yapabilmedir. Genelleme öğrenme süresinde karşılaşılmayan girişler için YSA'nın uygun tepkiler üretmesidir. YSA bu sebeple birçok karmaşık uygulamada kullanılmaktadır.

Doğrusal Olmama

YSA yapı olarak doğrusal bir yapıya sahiptir. Fakat temel birim olan hücre doğrusal bir yapı olmadığından YSA doğrusal olmayan sonuçlar üretir. Bu sebeple karmaşık problemlerin çözümünde tercih edilir.(Haykin 1994)

Öğrenme

YSA çıkış olarak arzu edilen çıkışlara göre eğitilebilir. Eğitme işleminde bağlantılar arası ağırlıklar gerekmektedir. YSA yapısı gereği öğrenme sürecinden sonra belli bir ağırlığa sahip bağlantılarla çalışmaz(Haykin 1999). Bu sebeple YSA, önceden eğitildiği verilere göre öğrenmektedir.

Genelleme

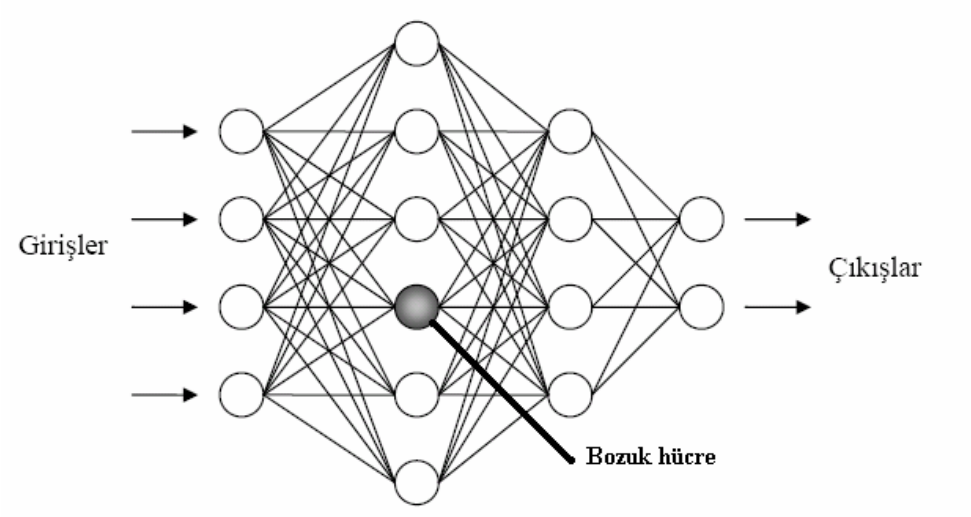
YSA belli bir problemi öğrendikten sonra eğitim esnasında karşılaşmadığı test örnekleri içinde istenilen çıkışı üretebilir.(Öztemel 2003) Örneğin karakter tanımada bozuk veya eksik bir karakteri de tanıyabilir.

Uyarlanabilirlik

Belli bir problem göre eğitilmiş YSA problemde değişiklikler yapılarak yeni bir probleme göre de eğitilebilir. Desen tanıma için kurulmuş bir YSA daha sonra sinyal işleme için kullanılmaya ayarlanabilir.

Hata Toleransı

YSA'da birbirine bağlı birçok hücre söz konusudur. Ağın sahip olduğu bilgi tüm ağdaki hücrelere dağıtılmıştır.(Haykin 1994) Bir hücrenin etkisiz hale gelmesi ağın ürettiği bilgiyi büyük ölçüde etkilemez. Geleneksel yöntemlere göre hata toleransı oldukça yüksektir.



Şekil 4.4 YSA Hata Toleransı

Donanım ve Hız

YSA, paralel bir yapıya sahip olduğu için bir devre olarak dizayn edilebilir. Bu özellik, YSA'nın bilgi işleme yeteneğini hızlandırır ve gerçek zamanlı uygulamalarda kullanılabilirliğini sağlar.

Analiz ve Tasarım Kolaylığı

YSA hücre yapısı genel olarak tüm YSA'larda aynıdır. Bu nedenle farklı uygulama alanlarında kullanılan YSA'lar benzer öğrenme algoritmalarını ve teorilerini paylaşabilirler.

Bağlantı Geometrileri

YSA yapısında her bir bağlantının bir ağırlık değeri vardır. Bağlantıların başlangıç ve bitiş değerlerinin bilinmesi işlemlerin doğru yapılabilmesi için oldukça önemlidir. Bağlantı ağırlıkları genelde (nxn) boyutlu bir matris olarak tanımlanırlar.

$$\left[w_{ij}\right] = \begin{bmatrix} w_{11} & w_{12} & . & . & w_{1n} \\ w_{21} & w_{22} & . & . & w_{2n} \\ . & . & . & . & . \\ . & . & . & . & . \\ w_{n1} & w_{n2} & . & . & w_{nn} \end{bmatrix}$$

n elamanlı tanımlanmış bir ağda en fazla n^2 ağırlık tanımlanabilir. Genelde YSA içinde bulunan her hücre aynı tip olduğu için bağlantıların sahip olduğu ağırlıklar aynı matematiksel tipten olması istenir. YSA içinde bulunan her bir katmandaki elemanlar diğer katmandaki her elemana bağlı ise ağ "tam bağlı" olarak tanımlanır.

4.4 Ağ Tipleri

Hücrelerin bağlantı şekillerine, öğrenme kurallarına ve aktivasyon fonksiyonlarına göre çeşitli ağ yapıları bulunmaktadır.(Özbay 1999)

1.**İleri beslemeli Yapay Sinir Ağları:** İleri beslemeli ağlarda hücreler katmanlar halindedir ve bir katmandaki hücrelerin çıkışları diğer katmandaki hücrelerin girişlerine bağlıdır. Giriş katmanı dışarıdan aldığı verileri değişikliğe uğratmadan orta(gizli) katmandaki hücrelere iletir. Bilgi orta ve çıkış katmanında işlenir. Bu yapı sayesinde ileri beslemeli ağlar doğrusal olmayan bir işlevi gerçekleştirebilirler. İleri beslemeli üç katmanlı bir ağ orta katmanında yeterince hücre olmak kaydıyla, herhangi bir sürekli fonksiyonu istenilen doğrulukta

yaklaştırabileceği gösterilmiştir. İleri beslemeli ağlarda bilgi akışı sadece ileri yöndedir.

2.**Kaskat Bağlı Ağ :** Hücrelerin sadece önceki katmanlarda bulunan hücreler tarafından beslendiği ağdır.

3.**Geri Beslemeli Ağ:** En az bir hücrenin çıkışı kendisine ya da diğer hücrelere giriş olarak verilir ve bir geciktirme sonrası ağ tekrar çalıştırılır. Geri besleme bir katmandaki hücreler arasında olduğu gibi farklı katmanlar arasındaki hücreler arasında da olabilir. Geri beslemeli YSA doğrusal olmayan dinamik bir yapı gösterir. Bu ağ tipinde hem ileri hem de geri bir veri akışı söz konusudur.

4.5 Eşik Fonksiyonları

Giriş değerleri kümesine göre belirlenmiş sınırlar içinde çıkış üreten fonksiyonlardır. Transfer, aktivasyon fonksiyonları olarak da bilinirler. Aşağıda yaygın olarak kullanılan fonksiyonlar kısaca açıklanmıştır.

4.5.1 Doğrusal Fonksiyonlar

Lineer fonksiyonlar giriş değerinin belli bir katını veren fonksiyonlardır. Şekil 4.3a ‘da görülen doğrunun fonksiyonu aşağıdaki gibidir.

$$F(x) = a \cdot x \qquad (4.1)$$

a işlem elemanın aktivitesidir ve sabit bir değeri vardır. Lineer fonksiyon $(-\tau, +\tau)$ arasında sınırlandığında rampa eşik fonksiyonu olarak adlandırılır. Denklemi

$$f(x) = \begin{cases} +\tau : & x \geq \tau \quad ise \\ x \quad : & |x| \quad ise\ yani (-\tau < x < \tau) \\ -\tau : & x \geq -\tau \ ise \end{cases} \qquad (4.2)$$

şeklini alır. $+\tau/-\tau$ fonksiyonun maksimum veya minimum alabileceği değerdir ve doyma seviyesi olarak adlandırılır. Doğru belirli aralıklarla kısıtlandığında Şekil 4.4b ‘deki şekli alır.

4.5.2 Doğrusal Olmayan Fonksiyonlar

Doğrusal olmayan fonksiyonların ilki sigmoid fonksiyonudur. Sigmoid fonksiyonu türevi alınabilir, sürekli ve doğrusal olmayan bir fonksiyon olması sebebiyle tercih edilendir. Çift yönlü sigmoid fonksiyonun denklemi aşağıdaki gibidir.

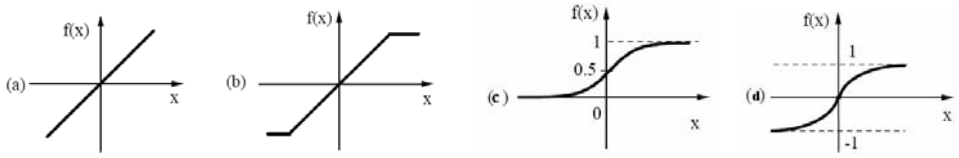
$$f(x) = \frac{1}{1+e^{-x}} \tag{4.3}$$

Eğrisi ise Şekil 4.5c 'de görülmektedir.

Diğer bir doğrusal olamayan fonksiyon ise tanh(x) fonksiyonudur. Denklemi

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1} \tag{4.4}$$

biçimindedir. tanh(x) fonksiyonu Şekil 4.5d 'de görülmektedir.



a-Lineer b-Rampa c-Sigmoid d-tanh(x)

Şekil 4.5 Aktivasyon fonksiyonları

YSA içinde kullanılan aktivasyon fonksiyonlarının yazılım gerçekleştirmelerinde doğruluk ve performans oldukça önemlidir. CORDIC (Coordinate Rotation Digatal Computer) yazılım gerçekleştirmelerinde kullanılan yöntemlerden bir tanesidir. (Meyer-Base. 2001) CORDIC algoritması: kartezyen koordinat sisteminde bir vektörün döndürülerek vektöre ait açı, uzunluk ve yeni kartezyen koordinat bileşenlerinin hesaplanması esasına dayanan, polar ve kartezyen koordinat sistemleri arasında dönüşüm gerçekleştiren bir algoritmadır. Cordic ile sadece öteleme-toplama yapılarak trigonemetrik, exponensiyel, logaritmik fonksiyonlar hesaplanabilir.

4.6 YSA'da Eğitme

4.6.1 Eğitme Algoritmaları

Öğrenme; gözlem, eğitim ve hareketin yapıda meydana getirdiği davranış değişikliği olarak tanımlanır. YSA'da birtakım metod ve kurallar eğitime göre ağdaki ağırlıkların değiştirilmesini sağlamalıdır. Eğitim için genel olarak üç eğitme metodundan ve bunların uygulandığı değişik öğrenme kurallarından söz edilebilir.

1.**Danışmanlı Eğitme (Supervised training):** Bu tip eğitmede, YSA'ya örnek olarak bir doğru çıkış verilir. İstenilen ve gerçek çıktı arasındaki farka göre bağlantıların ağırlıkları düzenlenir. Bu sebeple öğreticili eğitme algoritmasının bir öğretmene veya danışmana ihtiyacı vardır. Delta kuralı, genelleştirilmiş delta kuralı ve geri besleme algoritması danışmanlı öğrenme algoritmalarına örnek olarak verilebilir.

2.Skor ile eğitme (Graded training): Skor ile eğitmede giriş bilgilerine karşılık gelen çıkışların bilinmesine gerek yoktur. Çıkış işareti yerine skor verilerek ağın değerlendirmesi yapılır.

3. Kendini düzenleme ile eğitme(self-organization training) : Giriş verilerine karşılık gelen çıkış verileri mevcut değildir. Girişleri gruplandırarak eğittikten sonra verilen herhangi bir girişin eğitme sınıflarından hangisine ait olduğu gösterilir. Ağ kendini düzenleyerek işlemlerine devam eder.

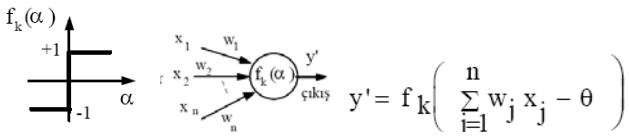
4.6.2 Bellek

YSA'nın yapı olarak bilgi saklayabilen bir dizayn biçimidir. YSA'da bilgiler bir yerel bellek içinde tutulur ve ağa aktarılır. Bellekte tutulan bilgiler ağın o anki ağırlıklarıdır. Ağın öğrenmesi gizli katman veya katmanlardaki özellikler ile olmaktadır.

4.7 Ağ Çeşitleri

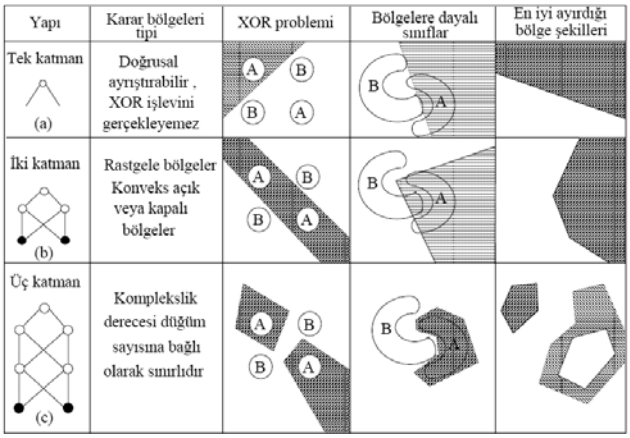
4.7.1 Perceptron

Perceptron ağı ilk defa 1943 yılında Mc Culloch ve Pitts tarafından ortaya atılmıştır. Bu basit ağ basit örüntüleri öğrenebilmektedir. Burada iki çeşit arasından bir sınıflandırma yapılması söz konusudur. Fakat ağ öğrenmesi oldukça basit kalmış ve üç veya daha fazla durumlar için kullanılamayacağı belirlenmiştir. Şekil 4.6'da bir perceptronun yapısı görülmektedir.



Şekil 4.6 Perceptron Yapısı

Problemin karar bölgesinin karmaşıklığına göre gizli katman sayısını belirlemek gerekir. XOR gibi üç veya daha fazla sınıfa ihtiyaç duyulan durumlarda ek katman eklenmelidir. Aşağıda Şekil 4.7'da katmanlara göre YSA'nın ayırabileceği karar alanları görülmektedir.



Şekil 4.7 Katman miktarlarına göre oluşan karar bölgeleri(Lippmann 1987)

Her bir katmanda bulunan düğüm sayıları ise problemin karmaşıklığına göre artırılıp azaltılmalıdır. Gutierrez ve arkadaşları çok fazla düğümünde çok az düğüm gibi zararlı olduğunu bulmuşlarıdır(Guitierrez ve ark 1989).Az sayıda gizli katman olduğunda ağ öğrenmeyi başaramaz. Gereğinden fazla katman olduğunda ise ağ ezberler. Amaç genellemeyi optimum yapan en az sayıda katmanı kullanmaktır.

4.7.2 Çok Katmanlı Perceptron

Çok katmanlı YSA giriş ve çıkış katmanı arasında birden fazla katmanın bulunduğu ileri yayılımlı bir ağdır.

Bu ağlarda verilen girişlere göre bir çıkış üretilir. Çıkışın istenen cevap ile farkı bulunur ve bu farkı azaltacak şekilde ağdaki ağırlıklar yeniden düzenlenir. Gizli katmanlarda istenilen çıkışlar bilinmediği için çıkış katmanı sonuçları dikkate alınarak gizli katman bilgileri değiştirilir.

Ağırlıkların düzenlenmesinde çıkış katmanındaki ağırlıklar ile başlanır ve ters yönde giriş katmanına doğru devam eder. İstenilen sonuç alınana kadar bu işlemler tekrar edilir. Bu yönteme "hatanın geriye yayılımı algoritması" (back-propagation algoritm) denilir. Hatanın geriye yayılım algoritmasında iki katsayı önem taşır. η öğrenme ve α momentum katsayılarıdır.

η öğrenme katsayısıdır ve ağırlıkların bir sonraki düzenlemedeki değişme oranıdır. Küçük öğrenme katsayıları ağın öğrenmesini yavaşlatırken büyük öğrenme katsayıları hesaplamalarda büyük değişimlere sebep olur. Öğrenme katsayısı 0.01-0.9 arasında değişen bir sayıdır. Genelde 0.7 kullanılmaktadır.

α momentum katsayıdır ve ağdaki salınımları engellemek için kullanılır. Böylece hata bölgesel minumum noktalarından kaçmış olur.

4.8 Hatanın Geriye Yayılma Algoritması

Hatanın geriye yayılması eğitme algoritması, çok katmanlı, ileri yayılımlı bir perceptrondan elde edilen çıkışların istenilene yaklaştırmak için kullanılan bir algoritmadır. j'inci elemanın çıkışı

$$\text{net}_j = \sum_{i=1}^{m} x_i w_{ji} \qquad (4.5)$$

olur. y^t_j j. elemanın hedef çıkışıdır.δ_j ara veya çıkış katmanındaki bir nörona ait faktördür ve aşağıdaki formülle hesaplanabilir. Çıkış katmanı için

$$\delta_j = \frac{\partial f}{\partial net_j}(y_j^{\ t} - y_j) \tag{4.6}$$

dir.Ara katmanlar için ise bu faktör;

$$\delta_j = (\frac{\delta_f}{\partial net_j})\sum w_{qi}\delta_q \tag{4.7}$$

olarak verilir. Ara katmanlarda bu eşitlik kullanılmaktadır. Bu eşitlikten faydalanarak bütün düğümler için hesaplanır.

4.8.1 Delta-Bar- Delta kuralı

Delta-bar-delta çok katmanlı YSA'larda ağırlıkların hedefe ulaşma hızını artırmak için kullanılan bir yaklaşımdır. Hata değişimlerini açıklamak için ağın her bağlantısının kendi öğrenme katsayısı olmalıdır. Her bağlantıya bir öğrenme katsayısı atanırsa ve bu katsayılar zamanla değişirse hedefe yaklaşma zamanı azalır ve serbestlik artmış olur. Öğrenme katsayıları kümesini belirlemek uzun zaman alabilir. Bir bağlantı için ağırlık değişimlerinin işareti, birkaç ardışık zaman adımları sırayla değiştiği zaman, bağlantı için öğrenme katsayısı azalmalıdır. Ağırlık boyutu hata yüzeyi ile ilgili küçük bir eğriliğe sahiptir ve yüzey önemli bir mesafe için aynı doğruluktaki eğime göre devam eder.

Standart geri yayılım algoritmasında eğim bileşeni aşağıdaki şekilde verilir.

$$\delta(k) = \frac{\partial E(k)}{\partial w(k)} \tag{4.8}$$

Burada E(k) k anındaki hata değerini, w(k) bağlantı ağırlığını ve δ(k) is ağırlık değişiminin eğim bileşenini göstermektedir. Standart geri yayılım algoritmasında bağlantı ağırlığı,

$$w(k+1) = w(k) + \alpha\delta(k) \tag{4.9}$$

olarak güncelleştirilir. Burada α sabit bir öğrenme katsayısıdır. DBD öğrenme kuralında, her bağlantı için değişken öğrenme oranı α(k) atanır ve bağlantı ağırlığının güncelleşmesi

$$w(k+1) = w(k) + \alpha(k)\delta(k) \tag{4.10}$$

şeklinde yapılır. δ(k) eğim bileşeninin ağırlıklı ortalamasıdır. Bu ağırlıklı ortalama $\bar{\delta}(k)$ aşağıdaki şekilde gösterilmiştir.

$$\bar{\delta}(k) = (1-\theta)\delta(k) + \theta\delta(k-1) \tag{4.11}$$

Burada , θ konveks ağırlık faktörüdür. Önceki eğim bileşeninin üstel artması ve şu anki eğim bileşeni aynı işaretli ise, öğrenme oranı k sabiti ile artan ağırlıkla birleştirilir. Mevcut eğim bileşeni üstel ortalamadan farklı işaretli ise, öğrenme oranı değeri ile orantılı olarak azalır. Bu öğrenme oranının güncelleştirilmesi aşağıdaki eşitlikle tanımlanmıştır.

$$\Delta\alpha(k) = \begin{cases} k & \bar{\delta}(k-1)\delta(k) > 0 \\ -\varphi\alpha(k) & \bar{\delta}(k-1)\delta(k) < 0 \\ 0 & diğiğ \end{cases} \tag{4.12}$$

Burada k öğrenme katsayısı artma faktörü ,φ öğrenme katsayısı azaltma faktörünü , α(k) k anındaki öğrenme oranını göstermektedir. Algoritma, öğrenme katsayılarını lineer olarak artırmakta, fakat geometrik olarak azalmaktadır.

4.8.2 Genelleştirilmiş Delta-Bar-Delta

Elde edilen çıkışlar ile hedef çıkışlar arasındaki hataların karesinin ortalamasını minimum yapmak için geliştirilmiş bir algoritmadır. Belirli bir anda ölçülen hata bir önceki anda bulunan hatadan büyük ise ağın ağırlıkları bir önceki ağırlık değerini alır.

Genelleştirilmiş delta-bar-delta kuralında μ(k) momentum hızı ve α(k) öğrenme hızı zamanla değişmektedir. Bağlantı ağırlıklarının değiştirilmesi

$$\Delta w(k+1) = \alpha\delta(k) + \mu\Delta w(k) \tag{4.13}$$

değerinin ağırlıklara katkısıyla

$$w(k+1) = w(k) + \Delta w(k+1) \tag{4.14}$$

elde edilebilir. Ağırlıklar

$$\Delta w(k+1) = \alpha(k)\delta(k) + \mu(k)\Delta w(k) \tag{4.15}$$

değerinin bir önceki ağırlığa eklenmesiyle $w(k+1) = w(k) + \Delta w(k+1)$ eşitliği elde edilir.

Burada μ(k) , k anındaki momentum hızıdır. Momentum hızı μ(k) ve öğrenme hızı α(k) aşağıdaki kurallara göre ayarlanır ve $\bar{g}$ hesaplanır.

$$\bar{g}(k) = (1-\theta)\big(\delta(k) + \theta\delta(k+1)\big) \tag{4.16}$$

Öğrenme hızı

$$\Delta\alpha(k) = \begin{cases} \kappa_\alpha \exp\big(-\gamma_\alpha \left|\bar{\delta}(k)\right|\big) & \bar{\delta}(k-1)\delta(k) > 0 \\ -\varphi_\alpha \quad \alpha(k) & \bar{\delta}(k-1)\delta(k) < 0 \\ 0 & \textit{aksi durumda} \end{cases} \tag{4.17}$$

formülünden elde edilir. Burada exp üstel fonksiyonunu , κ_α sabit öğrenme hızı skala faktörünü , φ_α sabit öğrenme hızı azaltma faktörünü ve γ_α sabit öğrenme hızı üstel faktörünü göstermektedir.

k anındaki momentum hızı değişimi

$$\Delta\mu(k) = \begin{cases} \kappa_\mu \exp\big(-\gamma_\mu \left|\bar{\delta}(k)\right|\big) & \bar{\delta}(k-1)\delta(k) > 0 \\ -\varphi_\mu \mu(k) & \bar{\delta}(k-1)\delta(k) > 0 \\ 0 & \textit{aksi durumda} \end{cases} \tag{4.18}$$

ifadesi kullanılabilir. Burada κ_μ sabit momentum hızı skala faktörünü , φ_μ sabit momentum hızı azaltma faktörü ve γ_μ sabit momentum hızı üstel faktörünü göstermektedir.

Öğrenme ve momentum hızları, onların azalması ve artmasını kontrol eden farklı sabitlere sabittir. $\bar{\delta}(k)\delta(k)$ 'nın işareti artma mı yoksa azalma mı olduğunu tespit etmek için kullanılır. Azalma delta-bar-delta kuralına göredir. Öğrenme ve momentum oranlarının artırılması $\left|\bar{\delta}(k)\right|$ 'nın üstel azalan fonksiyonu olarak değiştirilir. Böylece küçük eğimli bölgelerde, büyük eğimli bölgelere göre daha büyük artırımlar yapılabilir.

Ağırlık uzayında, osilasyonları ve aşırı atlamaları engellemek için üst sınırlar her bir bağlantı öğrenme hızları ve momentum hızları üzerinden belirlenir. Tüm bağlantılar için matematiksel ifade aşağıdaki gibidir.

$$\alpha(k) \leq \alpha_{\max} \tag{4.19}$$

$$\mu(k) \leq \mu_{\max} \tag{4.20}$$

Burada $\alpha_{\max}$ öğrenme katsayısının sınırı , $\mu_{\max}$ momentum katsayısının üst sınırıdır.

Eğitme verilerinin verilişinden sonra her iterasyon sonrasında hata değerlendirilir. Hata, E(k) bir önceki minimum hatadan küçük ise ağırlıkları o anki iyi değerler hafızada saklanır. Düzeltme tolerans parametresi λ , düzeltme sürecini kontrol eder. O anki hata önceki minimum hatayı aşarsa,

$$E(k) > E_{\min}\lambda \tag{4.21}$$

olursa tüm bağlantı ağırlıkları, hafızada saklı olan en iyi değerlerle değiştirilir. Öğrenme ve momentum hızları düzeltmeyi başlatmak için azaltılır.

4.8.3 Hızlı Yayılım Algoritması

Scott Fahlman tarafından geliştirilmiş bir öğrenme algoritmasıdır. En iyiye yakın çözümü bulmayı hedefler. Bu algoritmada iki geleneksel yaklaşım bulunmaktadır. Bunlar;

- Hesaplamanın geçmişteki durumu hakkında dinamik olarak ağırlıkların ayarlanması
- Her bir ağırlığa göre hatanın ikinci türevinin belirgin kullanımı

Bu algoritmada iki kabul vardır.

- Her bir ağırlık için, ağırlık eğrisi kolları yukarı doğru açık olan bir parabol ile yaklaştırılabilir.
- Hata eğrisinin eğilimindeki değişim, diğer tüm ağırlıkların aynı andaki değişimden etkilenmez.

Parabol birbirinden bağımsız her bir ağırlık için şimdiki ve önceki hata eğimleri ve bu eğimlerin ölçüldüğü noktalar arasındaki değişim kullanılarak belirlenir. Daha sonra, algoritma doğrudan bu minimum noktasına atlar.

Bu algoritmada ağırlık hızlandırma ve kırpılması ihmal edilerek t-1 den t'ye kadar ağırlıkların değişimi

$$\Delta w(t) = \varepsilon L(t) + \alpha Q(t) \tag{4.22}$$

formülü ile elde edilir. Burada ε öğrenme katsayısı ve α momentum katsayısıdır.

$$L(t) = \begin{Bmatrix} \text{h(t)} & h(t)h(t-1) \geq 0 \\ 0 & aksi\ \ durumda \end{Bmatrix} \tag{4.23}$$

ve

$$Q(t) = \begin{Bmatrix} \mu \Delta w(t-1) & h(t)\left(h(t) - \left(\frac{\mu}{\mu+1} \right) h(t-1) \right) \geq 0 \\ \Delta q(t) & aksi\ \ durumlarda \end{Bmatrix} \tag{4.24}$$

eşitlikleri ile verilir. Bu ifadelerde ; μ momentum büyüme faktörü $h(t) = \frac{\partial E}{\partial w(t)}$ eğimi, $\Delta q(t) = \frac{\Delta w(t) h(t)}{h(t-1) - h(t)}$ ise minimum adım miktarını göstermektedir. Buradan ağırlık fonksiyonunu güncelleştirmek için delta ağırlık fonksiyonu ve ağırlık hızlandırma katsayısı işleme katılır.

$$w(t) = (1 - \delta) w(t-1) + \Delta w(t) \tag{4.25}$$

Burada δ hızlandırma katsayısıdır. Son olarak ağırlık küçük ise , 0 alınarak kırpılır.

$|w(t)| < \kappa$ ise w(t)=0 alınır. κ kırpılma katsayısıdır.

5 PROGRAMLAMA TEKNİKLERİ

Sayısal işlem sistemleri, sayısal verileri işlemek için tasarlanmış hızlı donanımlara ve bu donanımlara işlevsellik kazandıracak yazılımlarla gerçekleştirilir. Sayısal işlem sistemi oluşturmada donanım ve yazılım tabanlı iki yöntem bulunmaktadır.(Çavuşoğlu 2006)

5.1 Donanım Tabanlı Yöntem

Sayısal verileri işlemek için kullanılan özel tüm devreler kullanılır. Bu tip devreler özel bir fonksiyonu gerçekleştirmek amacıyla üretildiğinden, bu fonksiyonları etkin ve hızlı bir şekilde gerçeklerler. Ancak yapabildikleri sınırlıdır ve ilgili oldukları uygulamaya yönelik üretildikleri için esneklikleri oldukça azdır. Yeni problemler için yeni ASIC yapıları tasarlanmalıdır. Bu maliyet ve zaman kaybına neden olmaktadır.

Yazılım tabanlı yöntemin hızlı, farklı yapılar, algoritmalar, aktivasyon fonksiyonları için esnek yapısı sebebiyle tercih edilir. FPGA-VHDL yardımı ile bu esneklik, hızlı dizayn ve yeniden kullanılabilirlik sağlanmıştır. Ayrıca donanım tabanlı yöntemdeki paralellik yazılımda sağlanamamaktadır.(Giron'es ve ark. 2004)

5.2 Yazılım Tabanlı Yöntem

Bu yöntemde tasarım oldukça esnek bir yapıya sahiptir. Mikroişlemcinin çalıştırdığı komutlar değiştirilerek bir donanım değişikliğine gereksinim duymadan yeni fonksiyonlar eklenebilir. Mikroişlemciler üzerinde çalışan uygulamalar tek bir işlemcinin kaynaklarını kullanırken yavaş fakat esnek yazılımlar ile çalışırlar. Bu tip kullanımlar ASIC devre elemanlarına göre daha işlevseldir. Fakat basit bir uygulama için komutların bellekten okunup, yorumlanıp gerçekleştirilmesi, sistemin performansını ve hızını düşürür.

Tekrar düzenlenebilen işlem sistemleri olarak da adlandırılan bu sistemler, esnek ve genel amaçlı yapıları sayesinde yeni bir üretim aşamasına ihtiyaç duymadan, değişik protokollere, sistem özelliklerine ve kullanıcı ihtiyaçlarına kısa sürede cevap verebilirler. Yine bu özellikleri sayesinde tekrar düzenlenebilen işlem sistemleri, donanım tabanlı sayısal işlem sistemlerine göre daha hızlı sayısal tasarım ortamı oluşturarak, bu iki sistem arasındaki boşluğu doldururlar.(Haykin 1999)

Tekrar düzenlenebilir sayısal işlem sistemlerinin ihtiyaç duyduğu esnek donanımlar Alan Programlamalı Kapı Dizileri (Field Programable Gate Array) ile mümkündür. Özellikle SRAM tabanlı FPGA'lar tekrar düzenlenebilirlik yetenekleri ve yüksek performansları sayesinde genel amaçlı sayısal donanımların tasarlanmasında önemli bir yer tutmaktadır.

Yakın bir tarihten beri FPGA'lar kadar etkili olan başka bir kavram ise HDL'dir (Hardware Description Language-Donanım tanımlama dili). Donanım tanımlama dillerinin kullanılması ile Systen-on-Chip (SoC) teknolojisini oluşturmuştur. SoC teknolojisinde çoğunlukla donanım tanımlama dilini kullanılmaktadır. Sistemin tanımlama aşamasından sonra derleme ve davranışsal benzetim adımları gerçekleştirilir. Sistemden beklenen cevap elde edildiğinde zamansal benzetim aşamasına gelinir. Bütün birimler sentezleme ve yerleştirme işlemi sonunda tekrar programlanabilir olan devreye aktarılır.

FPGA ve VHDL beraber kullanılarak donanımın hızı ve paralelliği ile yazılımın esnekliği birleştirilmek istenmiştir.

5.3 VHDL Tasarım Dili

VHDL, VHSIC Hardware Description Language sözcüklerinden oluşan bir kısaltmadır. VHSIC ise Very High Speed Integrated Circuit sözcüklerini içeren bir kısaltmadır.

Bu lojik çevrim,

- Davranış(behavior)
- Yapı(structure)

- Zamanlama(timing)

şeklinde tanımlanabilir.

VHDL, C yada C+ gibi bir programlama dili değildir. VHDL, dijital sistemlerdeki donanımların oluşturulmasında kullanılır.

5.3.1 VHDL'in Gelişim Süreci

VHDL, başlangıçta Department of Defense (DoD) tarafından geliştirilmeye başlanmıştır. DoD, makine ve insan için aynı zamanda okunaklı ve yapısal olarak güçlü, anlaşılabilir kod yazmaya elverişli, böylece kaynak kodunun kendisi bir çeşit kullanım kılavuzu görünümünde olan bir donanım tanımlama dilini talep etmekteydi. DoD geliştirici grubu dili geliştirerek 1985'te ilk versiyonu yayınlandı ve 1987'de standardı alındı. 1993 ve 2006 yıllarında dilin gelişmiş yeni versiyonları sunuldu.

5.3.2 VHDL Uygulama Alanı

VHDL, ASIC'in geliştirilmesi için kullanılır. VHDL kodu kapılara dönüşerek devreye serilir.

Karmaşık dizayn ile en iyi biçimde sentez kurulduğunda çoğunlukla en uygun sonuç elde edilir. Bu yüzden VHDL, karmaşık olmayan programlanabilir lojik aygıtların (PLD) dizaynı için hemen hemen hiç kullanılmaz.

5.3.3 VHDL ve Donanım Tasarımı Karşılaştırması

VHDL dilinin tasarım yapımında standart donanım tasarıma göre bazı üstünlükleri vardır.

Tasarım Süresi : Teknolojinin gelişimi beraberinde devrelerin ömrü azalmaktadır. Bu devrenin tasarım zamanının kısıtlanması demektir. Bu durumda devrenin optimum tasarımlanmış olmasının yanında tasarım süresinin kısıtlığı ön plana çıkar. VHDL dili doğrudan tasarıma göre daha kısa sürede sonuçlanır.

Tasarım Esnekliği : Zaman ilerledikçe devre elemanlarının yapıları da teknolojiye bağlı olarak değişmektedir. Yapı değişikliklerinin daha önceki tasarımlarda da çalışabilmesi için kullanılan tasarım ortamının buna uygun olabilmesi gerekir. VHDL dili fonksiyon bağımlı olarak çalışır. Dönüştürücü programlar yardımıyla yazılım donanım yapısı oluşturulur. Teknoloji değiştiğinde sadece bu dönüştürücü programların yeni teknolojiye uygun hale getirilmiş olması yeterli olacaktır.

Tasarım Kolaylığı : Genel olarak VHDL dili kullanarak yapılan tasarımlarda, klasik donanım tasarımına göre daha az donanım bilgisine ihtiyaç vardır.

Yenileme Kolaylığı : Bir devrede yapılan bir yenileme işlemi kısa sürede gerçekleşebilmektedir. VHDL dili ile yapılan tasarımın esnekliği tasarımcının yazılım gücüne bağlıdır.

5.3.4 VHDL Veri Nesneleri

Her bir veri nesnesi belirli bir tipe ait olan değerleri tutmaktadır. Kullanılan nesneler donanımda değerler alırlar ve donanım şeklini belirlerler.

1.Sinyal (Signal) : Fiziksel olarak devre içinde yer alan donanım içindeki ara değişkenlerdir. Güncel değeri ve belirlenmiş bir sonraki değerleri tutarlar. Sinyal tanımlama *architecture* alanı içinde en başta *begin*'den önce olmalıdır.

Sinyal tanımlama;

```
SIGNAL  sayi:STD_LOGIC_VECTOR(7 downto 0);
SIGNAL bit_1 , bit_2 :STD_LOGIC;
SIGNAL veri_sayi :INTEGER ;
```

şeklindedir.

2.Değişken (Variable) : Yapılan işlemlerin sonuçlarını tutmak için kullanılırlar. Fiziksel bir yapı ile donanım içinde bulunmazlar. *Function, procedure* ve *process* içinde *variable* kullanılmalıdır.

Değişken tanımlama;

```
Variable deg_1 :integer range  0 to 255:=8 ; -- ön değer atama
Variable  hafiza : bit_matrix ( 0 to 5, 0 to 1023);
Variable bit_1 : std_logic_vector(7 downto 0):=(others=>'0');
```

3.Sabit (Constant) : Program içinde kullanılacak olan değişmeyen bir değeri içerir. Sabit tanımlama;

```
Constant  iterasyon_sayi:integer:=100;
Constant deger:std_logic_vector(15 downto 0):="0001110001110001";
```

4.Tip tanımı (type): Her hangi bir veri tipi tanımlamak için kullanılan VHDL yapısıdır. Genelde birden fazla boyutu olan diziler veya katar yapıları oluşturmak için kullanılırlar. *Subtype* ise bir veri tipinin kısıtlanmış halini yeni bir veri tipi olarak tanımlamak için kullanılır.

Tip tanımları ;

```
TYPE  agirliklar_h IS  ARRAY  (1 to 3 ) OF std_logic_vector(16 downto 0);
TYPE  agirliklar_hidden IS  ARRAY (1 to 5) OF agirliklar_h;
TYPE k IS (k_1, k_2,k_3);
SUBTYPE  toplam  IS std_logic_vector(15 downto 0);
```

Ön Tanımlamalı Veri Tipleri

1. Standart Lojik Tip : STD_LOGIC(0 ve 1) , STD_LOGIC_VECTOR(bit dizisi)
2. Integer (tamsayı tip): INTEGER
3. Floating Point (kayan noktalı sayı): REAL
4. Physical (fiziksel) Tip : TIME
5. Enumeration (Liste) Tip : BOOLEAN ,CHARACTER

VHDL' de bulunan her bir değişkeni tanımlayabilmemiz için kütüphanelere ihtiyacımız vardır. VHDL kütüphane dosyalarına dayanan bir yapıya sahiptir. Her bir kütüphane dosyası aslında birer paket dosyadırlar ve işlem, değişken tipi ve

fonksiyon kullanmak için program öncesinde çağırılmalı ve projeye eklenmelidirler. Örneğin STD_LOGIC_VECTOR veri tipi std_logic_1164 paketi içindedir.

Kendi veri tiplerimizi oluşturmak istediğimizde paket program yazıp dosyaya eklememiz veya program başlangıcında tanımlamamız gerekmektedir.

5.3.5 VHDL Yapısal Elemanları

- Varlık (Entity)
- Mimari(Architecture)
- Biçim(Configuration)
- Paket(Package)
- Kütüphane(Library)
- İşlem(Process)
- Alt Programlar(function ,procedure)

Temel bir devre entity , library ve architecture yapısal elemanlarını içermek zorundadır.

5.3.5.1 Varlık (Entity) Tanımlanması

Entity alanı bir devrenin giriş ve çıkışlarının tanımlandığı bir alandır. Tüm işlemler bu giriş ve çıkış bilgilerine uygun olarak gerçekleştirilmelidir. *Entity* alanında ilk yapılan işlem port tanımlanmasıdır. Port içindeki verilerin dört değişik durumu olabilir.

- IN : Devreye giriş sağlayan (okunabilen) port bileşen biçimidir.
- OUT :Devreden çıkış sağlayan (yazılabilen) port bileşen biçimidir.
- INOUT:Hem giriş hem de çıkış olabilen port bileşen biçimidir.
- BUFFER: Hem giriş hem de çıkış olarak kullanılabilen port bileşen biçimidir.

Generic port ise devreye dizayn sırasında kullanılacak değişkenleri girmek için kullanılan özel bilgi tanımlama alanıdır. Bu port yapısı devrenin esnekliğini sağlamaktadır.

Örnek bir entity alanı;

```
entity say is
        generic(n: natural :=2);
          port(  clock:   in std_logic;
                         sil:        in std_logic;
                         deger:    in std_logic;
                         sonuc:    out std_logic_vector(n-1 downto 0)
                  )
end say;
```

Yukarıdaki *entity* alanı 2 bit sayma yapan bir devre dizaynı için yazılmıştır. Generic içinde bulunan n değeri değiştirildiğinde sayma işlemi bit sayısı ve buna bağlı olarak devre biçimi değişmektedir. *Entity* ismi ile vhdl dosya ismi aynı olmak zorundadır.

5.3.5.2 Mimari (Architecture) Tanımlanması

Devrenin nasıl davranması gerektiğinin belirlendiği alan *architecture* alanıdır. Bir devreye istenirse birden fazla *architecture* yazılıp bunlardan bir tanesinin seçilmesi durumu söz konusu olabilmektedir. Fakat genelde her bir devrenin bir adet davranış biçimi olur.

Yukarıdaki entity'e ait mimari tanımlanması;

```
architecture davranis of say is
   signal sonra: std_logic_vector(n-1 downto 0);
begin
   -- sayac davranısı
   process(clock, deger, sil)
   begin
        if sil= '1' then
           sonra <= sonra - sonra;
        elsif (clock='1' and clock'event) then
```

```
            if deger= '1' then
                sonra <= sonra + 1;
            end if;
        end if;
    end process;
    sonuc <= sonra ;
end davranis;
```

şeklinde olmalıdır. Tanımlanan devre içerikleri istenirse başka devrelerde de kullanılabilirler. *Generic* ve *port* alanlarına değer atanarak devre başka bir devrenin içinde yer alabilir. Devreyi başka bir devrede kullanmak için ***component*** veya ***configuration*** olarak program içinde çağırmak gerekir.

Örnek :

```
Entity ornek is
  Port ( clk1  : in std_logic;
         s1    : out std_logic_vector( 15 downto 0);
         clk2  : in std_logic;
         s2    : out std_logic_vector( 9 downto 0);
        sil    :in std_logic
        );
end ornek;
architecture  mimari of ornek is
   component say is
     generic(n: natural :=2);
     port    (     clock   :in std_logic;
                   sil     :in std_logic;
                   deger   :in std_logic;
                   sonuc   :out std_logic_vector(n-1 downto 0)
      );
     end component;
     signal a : std_logic;
     signal  say1,say2: integer;
       begin
          a<='1';b<='1';
         say1<=15;  say2<=9;
          u1:say   generic map (n =>say1)
                   port map(clock=>clk1,    sil=>sil ,    deger=>a , sonuc=>s1) ;
          u2:say   generic map (n =>say2)
                    port map(clock=>clk2,    sil=>sil ,    deger=>b , sonuc=>s2) ;
```

```
end mimari;
```

Yukarıdaki örnekte görüldüğü gibi daha önceden dizayn edilmiş devreler başka bir devrede component olarak çağırılıp kullanılabilirler. Component olarak çağırılan dizaynlar alt dizaynlar olarak bütünün içinde yer alırlar.

5.3.5.3 Biçim (Configuration) Tanımlanması

Biçim tanımı bir entity'e birden fazla architecture yerleştirildiğinde ve bu architecture'ların çağırılması gerektiğinde kullanılır. Mimari çağırılması *use entity work.entity_ismi(mimari_ismi)* şeklinde olmaktadır. Work ifadesi , dışarıdan yazılan bir devre veya kütüphane tanımlanması durumunda kullanılır.

Örnek :

```
entity say2 is
        generic (gecikme : Time := 10 ns);
        port (    clk : in bit;
                  q1, q0 : out bit);
end say2;
--say2 üzerinde tanımlanmış  birinci mimari
architecture davranis1 of say2 is
   begin
     a1: process (clk)
          variable sayac_deger : natural := 0;
           begin
              if clk = '1' then
               sayac_deger := (sayac-deger + 1) mod 4;
               q0 <= bit'val(sayac_deger mod 2) after gecikme;
               q1 <= bit'val(sayac_deger / 2) after gecikme;
             end if;
      end process a1;
end davranis1;
-- say2 üzerinde tanımlanmış ikinci mimari
architecture davranis2 of say2 is
       component t_flipflop
                  port (ck : in bit; q : out bit);
       end component;
       signal ff0, ff1, inv_ff0 : bit;
          begin
```

```
            bit_0 : t_flipflop port map (ck => clk, q => ff0);
            bit_1 : t_flipflop port map (ck => inv_ff0, q => ff1);
            q0 <= ff0;
             q1 <= ff1;
 end davranis2;
-------------------------------------------------
 entity test_say2 is
 end test_say2;
 architecture deneme of test_say2 is
   signal clock, q0, q1 : bit;
   component say2
         port (clk : in bit;
               q1, q0 : out bit);
   end component;
  begin
  sayac : say2    port map (clk => clock, q0 => q0, q1 => q1);
end deneme;
configuration test_say2_d of test_say2 is
       for deneme -- of test_say22
              for sayac : say2
               use entity work.say2(davranis1);
               end for;
        end for;
end test_say2_d;
```

Yukarıdaki örnek configuration alanından da anlaşılacağı gibi test_say2 deneme isimli architecture alanında bulunan sayac label'ında say2 entity'sinin davranis1 mimarisini kullanılacaktır. Bu şekilde farklı mimari ve label alanlarında bir entity'nin farklı mimarileri kullanılabilir.

5.3.5.4 Paket (Package) Yapısı

Vhdl dilinin temel bileşenlerinden bir tanesi de paketlerdir. Paketler farklı değişken isimleri ve tipleri tanımlamak için kullanıldığı gibi, procedure ve fonksiyon tanımlamak içinde kullanılırlar. Paketler iki önemli kısımdan oluşurlar. Birincisi paketin kendisidir ve sabit, değişken, procedure ve fonksiyon tanımlamak için kullanılır. İkincisi ise eğer procedure ve fonksiyon tanımlandı ise programların

kodlarının bulunduğu paket vücududur. Eğer sadece sabit ve değişken tanımlandı ise paket vücuduna gerek yoktur.

Örnek

```
package  fonksiyon is
        constant g_node_sayi : integer :=2;-- 8;--giris node sayisi
         constant h_node_sayi : integer := 4;--gizli katman node sayisi
        constant c_node_sayi:integer:=2; --cikis node sayisi
    constant agirlik_uzunluk:integer:=7;    --agirlik degeri uzunlugu
  function carpma_h
              (
                 a:        STD_LOGIC_VECTOR (giris_uzunluk DOWNTO 0);
                 b:         STD_LOGIC_VECTOR (agirlik_uzunluk DOWNTO 0) )
                  return STD_LOGIC_VECTOR;
end fonksiyon;
package body fonksiyon is
function carpma_h
         (
                 a:        STD_LOGIC_VECTOR (giris_uzunluk DOWNTO 0);
                 b:         STD_LOGIC_VECTOR (agirlik_uzunluk DOWNTO 0))
                  return STD_LOGIC_VECTOR is
        variable a_int:      SIGNED (giris_uzunluk downto 0);
        variable b_int:    signed(agirlik_uzunluk downto 0);
        variable pdt_int:SIGNED (giris_uzunluk+agirlik_uzunluk+1 downto 0);
begin
        a_int := SIGNED (a);
        b_int := SIGNED (b);
        pdt_int := a_int * b_int;
        return  STD_LOGIC_VECTOR(pdt_int);
end carpma_h;
end fonksiyon;
```

şeklinde olmalıdır. Paket adı , paket vücudu adı , vhdl dosya adı aynı olmalıdır.

5.3.5.5 Kütüphane (Library) Yapısı

Her entity başlangıcında bu devre içinde kullanılacak olan programlar ve tip tanımlamaları paket programlar ile entity içinden çağırılmalıdır. Paketler genelde üç kütüphane içinde tanımlanırlar. Bunlar ieee , std ve work kütüphaneleridir. ieee ve

std kütüphaneleri vhdl diline ait olan kütüphanelerdir ve bir kısmı vhdl yüklenmesi ile beraber gelirken bir kısmı ise internetten indirilebilmektedir. ieee kütüphanesinde matematiksel işlemler, nümerik işlemler, text işlemleri paketleri yer alırken std kütüphanesinde dosya okuma fonksiyonları ve standart değişkenlerin bulunduğu paketler yer alır. work kütüphanesi ise kullanıcı tarafından yazılmış olan paketleri içerir. Diğer kütüphaneler ise kullanılan devre tipini veya özel işlemleri içeren paketlerdir. Örneğin altera devrelerini ile quartus yazılımı kullanıldığında lpm devrelerinin içeriklerini taşıyan lpm kütüphanesi de gelmektedir. Lpm devreleri çarpma ,toplama ,… işlemleri yapan özel devrelerin kodlarını içerir.

Örnek:

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.ALL;
library std;
use std.textio.all;
library work;
use work.fonksiyon.all;
```

Yukarıdaki kütüphane çağırımlarında use std_textio.all satırı dizayna textio_vhdl87.vhd ve textio_vhdl93.vhd dosyalarını eklemektedir. work kütüphanesi ile çağırılan fonksiyon paketi ise yukarıdaki örnekte verilen fonksiyon paketini dizayna ekler.

5.3.5.6 İşlem (Process) Tanımlanması

Tanımlanmış bir entity alanı içindeki architecture alanında if, for gibi kontrol ifadelerini kullanabilmek için işlem bloğuna(process) ihtiyaç vardır. İşlem bloğu olmadan architecture alanında sadece label ve case-when komutları çalışabilmektedir. İşlem tanımı iki şekildedir. Birincisi değişken girişleri veya label alanı ile tanımlanmış işlemler ikincisi ise tek başına tanımlanmış işlem bloklarıdır. Değişken ile tanımlanmış işlemler normal bir program bloğudur ve içindeki komutlar çalıştırır. Tek başına tanımlanmış işlemler ise wait until yapısı sayesinde belirli bir sinyal ile tetiklenen program bloklarıdır.

Örnek:

```
library ieee;
use ieee.std_logic_1164.all;
entity DECODER is
port(     clk: in std_logic;
            I:      in std_logic_vector(1 downto 0);
           O1:      out std_logic_vector(3 downto 0);
           O2:      out std_logic_vector(3 downto 0)
);
end DECODER;
architecture behv of DECODER is
begin
   process
   begin
   wait until clk='1' and clk'event ;
      case I is
            when "00" => O1 <= "0001";
            when "01" => O1 <= "0010";
            when "10" => O1 <= "0100";
            when "11" => O1 <= "1000";
            when others => O1 <= "XXXX";
         end case;
   end process;
  u1: process (I)
   begin
      case I is
            when "00" => O2 <= "0001";
            when "01" => O2 <= "0010";
            when "10" => O2 <= "0100";
            when "11" => O2 <= "1000";
            when others => O2 <= "XXXX";
         end case;
   end process u1;
end behv ;
```

Yukarıdaki örnekte iki adet kodçözücü bulunmaktadır. Birinci işlem bloğundaki birinci kodçözücü clk sinyali değiştiğinde veya 1 olduğunda çalışarak O1 çıkışını üretir. Sonuç üretimi clk sinyaline bağlıdır. İkinci kodçözücü ise I sinyaline bağlıdır ve I sinyali değiştiğinde çalışarak O2 sinyalini üretmektedir.

5.3.5.7 Alt Programlar (Fonksiyon ve Prosedür) Tanımlanması

Fonksiyon ve prosedür vhdl diline bir esneklik sağlayan program bloklarıdır. Fonksiyonlar diğer programlama dillerinde olduğu gibi girilen değişkenleri işleyen ve sonuçta bir değer döndüren bloklardır. Prosedür ise girişleri işleyen fakat fonksiyondan farklı olarak birden fazla değer döndürebilen program bloklarıdır. Prosedür ve fonksiyon tanımları paket içinde veya architecture başlangıcında tanımlanmalıdırlar.

Örnek:

```
library ieee ;
use ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.std_logic_unsigned.ALL;
entity func_proc is
  port (sec: std_logic;
        a ,b :in std_logic_vector(3 downto 0);
        topla,cikar : out std_logic_vector(3 downto 0);
        carp : out std_logic_vector(7 downto 0)
       );
end  func_proc;
architecture d of func_proc is
function  carpma(a, b:  std_logic_vector(3 downto 0)  ) return
                                    std_logic_vector is
begin
                    return  signed(a) *signed(b);
end function carpma ;
procedure topla_cikar (sec_p:in std_logic;
                c, d: in  std_logic_vector(3 downto 0 );
                topla :out std_logic_vector(3 downto 0);
                cikar :out  std_logic_vector(3 downto 0) ) is
begin
                  if (sec_p = '1') then
                            topla := signed(c) + signed(d);
                  else
                            cikar:= signed(c) -signed(d);
                  end if;
```

```
end procedure;
begin
   u1:process(sec,a,b)
       variable topla_v,cikar_v : std_logic_vector (3 downto 0);
       variable carpma_v :std_logic_vector(7 downto 0);
   begin
       carpma_v:=carpma(a,b);
       carp<=carpma_v;
       if sec='1' then
          topla_cikar(sec ,a,b,topla_v,cikar_v);
          topla<=topla_v ;
       else
          topla_cikar(sec ,a,b,topla_v,cikar_v);
          cikar<=cikar_v;
       end if;
    end process u1;
end ;
```

6 FPGA TABANLI AĞ MİMARİSİ

YSA teknolojisi her geçen gün gelişmekte ve uygulama alanları genişlemektedir. Desen tanıma, işaret işleme, kontrol sistemleri gibi birçok problem YSA ile çözülebilmektedir. Günümüze kadar yapılan araştırmalar genelde benzetim aşamasındadırlar. YSA'ların kalıtımsal yapısını ve paralelliğinin avantajlarını görebilmek için donanım gerçeklemesi şarttır.(Burr 1993)

Analog ve dijital devrelerle oluşturulmuş sistem mimarileri yapay sinir ağları için önerilebilir. Analog uygulamaların hassasiyet oranı yüksektir fakat gerçeklenmesi zordur ve ağırlık saklamada sorun yaşarlar. Dijital sistemlerde ise hassasiyet oranı düşüktür fakat ağırlık saklama sorunu aşılmıştır. Kullanıcı tarafından tasarlanabilen ve güçlü tasarım, sentezleme ve programlama araçları olan FPGA barındırdığı bir çok avantajla programlanabilir lojik eleman olarak oldukça ilgi gördü. YSA'larının mimarisi biyolojik sinirlerden esinlendiği için doğasında paralellik barındırmaktadır. Mikroişlemci ve DSP paralel tasarımlar için uygun değildir. Tamamen paralel bir yapı ASIC ve VLSI teknolojileriyle yapılabilir fakat bu yöntem maliyet ve zaman açısından oldukça masraflıdır. Buna ek olarak özel bir YSA için üretilmiş ASIC mimari sadece hedef uygulama için kullanılabilecektir. FPGA paralel mimarinin yanında esnek tasarım, maliyet ve zamandan tasarruf sağlar(Yu ve ark. 1994).

6.1 Veri Gösterimi

Donanımsal olarak yapı tanımlanmadan önce giriş, ağırlık, aktivasyon fonksiyonu ve çıkış için sayı duyarlılığı düşünülmelidir. Tasarımda sayı duyarlılığını artırmak FPGA kaynaklarını da artırmak demektir.

Öğrenme aşamasında sayıların duyarlılığının yüksek tutulması önemlidir. Bu öğrenme işlemindeki doğruluğu artıracaktır. Fakat test aşamasında düşük duyarlıklı sayılar kullanılabilir.

YSA sistemlerinde genelde analog veriler işlenir. FPGA ise dijital devre elemanıdır. FPGA giriş olarak sadece dijital girişleri kabul eder. Uygulama safhasına ilk olarak verilerin girişe hazırlanması yani dijital biçime dönüştürülmesi gerekmektedir. FPGA içinde kullanılabilen iki çeşit sayı formatı vardır. Bunlardan biri kayan noktalı sayı formatı ikincisi ise sabit noktalı sayı formatıdır. FPGA içinde yapılan uygulamalar bu iki sayı tipine göre yapılmak zorundadır.

6.2 Kayan Noktalı Sayı Aritmetiği

Kayan noktalı sayı aritmetiği işaret, üs ve çarpan yardımı ile reel sayıların gösterimidir. Altera FPGA içinde kayan noktalı sayı ile işlem yapan devre elemanı megafunction *altfp_add_sub* (toplama –çıkarma) , *altfp_mult* (çarpma) devrelerini sunmaktadır. Bu devreleri kullanmak mecburi değildir istenirse vhdl kodu bulunarak veya yazılarak işlem devresi farklı yapılabilir. Kayan noktalı sayılar hassasiyetleri sebebiyle tercih edilirler. Fakat donanım kaynaklarını sabit noktalı sayılara göre daha fazla tüketirler.

6.2.1 Kayan Noktalı Sayı Gösterimi

Şekil 6.1 İEEE 754 32 bit kayan noktalı sayı gösterimi

32 bit kayan noktalı sayı gösterimi yukarıdaki şekildedir. Kayan noktalı sayı formatında verilen sayının dönüşümü için aşağıdaki denklem kullanılır.

$$Sayı = (-1)^{s}(1+f).2^{e-bias} \qquad (6.1)$$

s işaret , bias ieee 754 standartlarında verilen ve üs sayısından çıkarılan değer , f çarpan ‘dır. 7 sayısının standartlara uygun olarak gösterimi

0 – 10000001-11000000000000000000000 şeklindedir.

6.2.2 Sabit Noktalı Sayı Gösterimi

IEEE standartlarına göre sabit sayı gösterimi işaret, tam ve ondalıklı kısım olarak ayrılmıştır. Sabit noktalı sayıların sunumumu için internette farklı paketler bulunmaktadır. Bunlar fix_std.vhdl (Altera'nın sunduğu) ve fixed_pkg.vhdl dosyalarıdır. Oluşturulan devrede fized_pkg.vhdl kullanılmıştır. Sabit noktalı sayı gösteriminde iki türlü noktalı sayı bulunur. SFixed ve Ufixed dir. Sfixed ve Ufixed sayı tipleri numeric_std içindeki unsigned ve signed tiplerini kullanır. Ufixed ve Sfixed std_logic veri dizisi şeklindedir. Ufixed işaretsiz sayı, Sfixed ise pozitif ve negatif veri tipinin ikili gösterimidir. 7 bit işaretsiz bir sabit noktalı sayı aşağıdaki biçimdedir. Sabit noktalı sayı altera'da lpm_mult , lpm_add_sub megafunction blokları tarafından işlenirler. İstenirse başka kodlar ve devre elamanları kullanılabilir.

6.2.3 İşaretsiz Sayı Gösterimi

İşaretsiz sabit noktalı sayı ikili sayı gösterimi olarak sunulur. N bitlik bir sayı n-1 den 0 'a kadar değer alır. Unsigned olarak gösterimi (n-1 downto 0) biçimindedir.

Bit Numarası	4	3	2	1	0	-1	-2
	1	0	1	1	1	0	1
Bit değeri	16	8	4	2	1	½	¼

Şekil 6.2 7 bit işaretsiz sabit noktalı sayı gösterimi

Yukarıdaki sayının değeri 23◦25 dir. Tam kısımların her birinin değeri 2^b değerindedir. Ondalıklı kısım ise eksi üs değerleri ile gösterilir.

6.2.4 İkiye Tümleyen İşaretli Sabit Noktalı Sayı

Pozitif sayıların gösteriminde kullanılan bir yöntemde ikiye tümleyen yöntemidir Burada işaret biti yerinde olan bit eksi bir değer olarak sayıdan çıkarılmaktadır. DSP uygulamalarındaki çarpmada yaygın olarak kullanılmasına

rağmen toplama ve çıkarmada pek tercih edilmez. Şekil 6.3 ‘da ikiye tümleyen gösterim için birkaç örnek verilmiştir.

Bit Numarası	3	2	1	0
+3	0	0	1	1
-7	1	0	0	1
Bit değeri	-8	+4	+2	+1

Şekil 6.3 +3 ve -7 Sayılarının İkiye Tümleyen Gösterimi

4 bit kullanılarak gösterilecek en büyük pozitif sayı +7 (0111) , en küçük negatif sayı ise -8 (1000) dir. İkiye tümleyen aritmetiğinde toplama ve çıkarma devreleri büyük ölçüde kullanılabilir. Ayrıca normal bit dizisinde yer alan sıfır (+0 ve -0) değerinin iki gösterimi ikiye tümleyende yer almamaktadır. Sayı değerlerinde kullanılan ikiye tümleyen mantığı noktalı kısımda da uygulanabilmektedir. Noktalı sayı kısmında yine aynı şekilde en yüksek değerlikli bit eksi değer ile sayıdan çıkartılmaktadır. Şekil 6.4’de -1/8 değerinin ikiye tümleyeni gösterilmektedir.

Bit Numarası	-2	-3
-⅛	1	1
Bit değeri	-¼	+⅛

Şekil 6.4 -1/8 Sayısının İkiye Tümleyen Gösterimi

6.2.5 Taşma ve Yuvarlama

Aritmetik işlemler sonrasında sonucun eldeki bit uzunluğuna sığmaması veya hedef ile kaynak bit uzunluklarının eşit olmaması durumu ortaya çıkabilir. Bu durumlardan en az kayıp almak için taşma ve yuvarlama teknikleri kullanılır.

6.2.5.1 Taşma

Sonucun hedef uzunluğuna sığmaması durumunda ortaya çıkan sorundur. Taşma işaretli sayılar ve işaretsiz sayılarda farklı sonuçlara sebep olmaktadır.

Örneğin +12 (1100) 3-bit ile gösterilemez çünkü 3-bit ile gösterilebilen en büyük değer +7 dir. Fakat ikiye tümleyen gösterimde -3 (1101) 3-bit'e sığar çünkü ikiye tümleyen üç bit gösterimde -4 (100) dür.

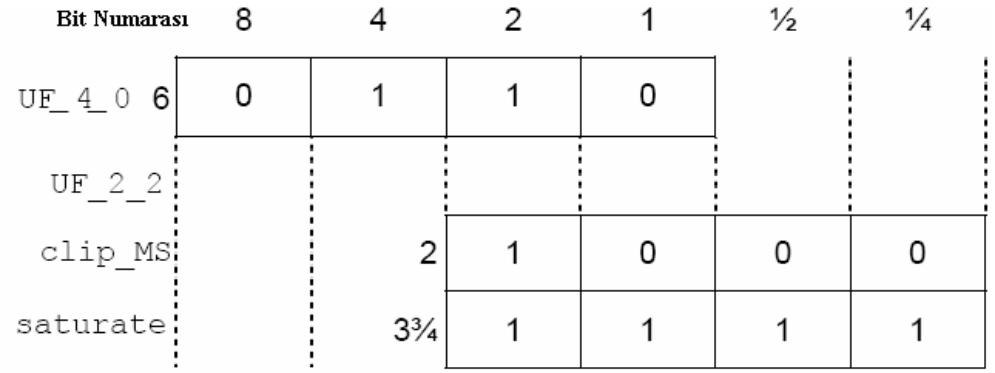
İki tip taşma söz konusudur.

- İşaretsiz sonuçlar için
- İşaretli sonuçlar için

Fixed_pkg paketinde taşma durumunda iki mod tercih edilebilir. Clip_ms ve saturate.

Clip_ms taşma durumunda en önemsiz biti silme metodudur. Bu yöntemin donanıma uygulanması oldukça kolaydır. Fakat işaretli aritmetik işlemlerde bazen kaybedilen bit önemli olabilmektedir.

Saturate modunda ise sonuç değeri sonuca en yakın istenilen değer ile değiştirilir. Analog sinyal işlemede bu yöntem basittir ve bazı DSP kaynaklarında da uygulanması yararlıdır. Şekil 6.5 'de 4 bit integer 6 sayısını 2 tam 2 noktalı sabit noktalı sayıya yerleştirilmesi her iki moda göre gösterilmektedir.



Şekil 6.5 6 Sayısının Clip_ms ve Saturate ile Yerleştirilmesi

6.2.5.2 Yuvarlama

Yuvarlama, noktadan sonraki bitlerin hedef değerine yaklaştırmak için kullanılır. Fix_std paketinde üç yuvarlama mümkün olmaktadır. Clip_ls, towords_zero ve to_nearest

Clip_ls en önemsiz bitlerin sonuca sığmaması sebebiyle silinmesidir. Bu uygulama oldukça kolaydır. Fakat kaybolan bit değerleri bazen önemli olabilir.

Towords_zero, sıfıra doğru sayıları yuvarlar. İşaretsiz sayılar için clip_ls gibi çalışır. Fakat negatif sayılarda sıfıra doğru sayı yukarı artırılmalıdır. Bu durumda sayının ikiye tümleyeni alınıp clip_ls metodu uygulanır.

To_nearest, sayıya en yakın değere yuvarlanması demektir. Bitler kontrol edilir ve bit bir ise yukarı doğru bir artırılır. Eğer sıfır ise aşağı doğru azaltılır. Şekil 6.6 'da 1◦75 iki tam iki noktalı sayısının clip_ls ve to _nearest metodu ile dört tam sabit noktalı sayısına sığdırılması gösterilmiştir.

Bit Değeri	8	4	2	1	½	¼
UF_2_2 1¾			0	1	1	1
UF_4_0						
clip_LS 1	0	0	0	1		
to_nearest 2	0	0	1	0		

Şekil 6.6 1.75 Sayısının clip_ls ve to_nearest ile Yerleştirilmesi

Varsayılan Taşma modu clip_ms, yuvarlama modu ise clip_ls'dir.

6.2.6 Sabit Noktalı Sayı Sunumu

Sabit noktalı sayılar fixed_pkg paketinde std_logic_vector tipi olarak sunulur. Tanımlama her zaman azalan aralıklarla olmalıdır (downto). Sayı sunumunda tam kısım pozitif noktalı kısım ise negatif olarak yazılmalıdır. I integer bit sayısına, F noktalı bit sayısına sahip bir sayı için noktalı sayı tanımı (I-1 downto -F) şeklindedir. Pratikte UFixed ve Sfixed aynıdır.

Örnek : *variable 3_int_2_nokta : Ufixed(2 downto -2) :=("11101") ;-- 7.25*

variable 2_int_2_nokta_s:Sfixed(2 downto -2):=("11101");-- -3.25

Bit numarası	+2	+1	+0	-1	-2
7 ¼	1	1	1	0	1
Bit Değeri	4 (tam)	2	1	½ (noktalı)	¼
- 3¼	1	1	1	0	1
Bit Değeri	işaret	2 (tam)	1	½ (noktalı)	¼

Şekil 6.7 7,25 ve -3,25 Sayısının fixed_point Gösterimi

6.3 Yapay Sinir Hücre Yapısı

Bu çalışmada işlem birimleri için 16 bit işaretli sabit noktalı sayılar kullanılmıştır. Sayı 1 işaret biti, 5 tam, 10 noktalı kısım için ayrılmıştır.

6.3.1 Ağ Mimarisi

Ağ yapısında paralellik oldukça önemli bir kavramdır. Paralellik işlemlerin hızını artırmakta fakat kapıların sayısının artışına sebep olmaktadır. Paralel işlemlerde her bir katman ayrıca tasarlanmalıdır. Bu çalışmada paralel işleyen bir ağ yapısı oluşturulmuştur. Bu yapı esneklik özelliğini azaltmıştır. Fakat yapı için bir kod yazıldığı düşünülürse esneklik sağlanmıştır.

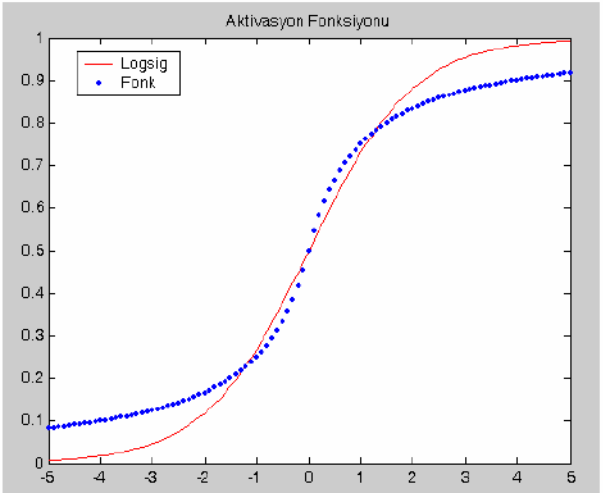
Amaç hayatta uygulanabilecek ve bir YSA yapısı oluşturmak olmuştur. Oluşturulan yapı farklı uygulamaları da başarı ile yapabilecek hassaslığa ve esnekliğe sahiptir.

6.3.2 Aktivasyon Fonksiyonu

YSA içinde genelde tanh veya logsig fonksiyonları tercih edilendir. Fakat logsig fonksiyonunu benzeri fonksiyonları FPGA içinde gerçeklemek oldukça fazla yer kaybına sebep olmaktadır. Bu sebeple uygulamada logsig fonksiyonuna çok benzer olan bir fonksiyon kullanılmıştır.(Çavuşoğlu 2006):

$$\log sig(x) = \frac{1}{1+e^{-x}} \tag{6.1}$$

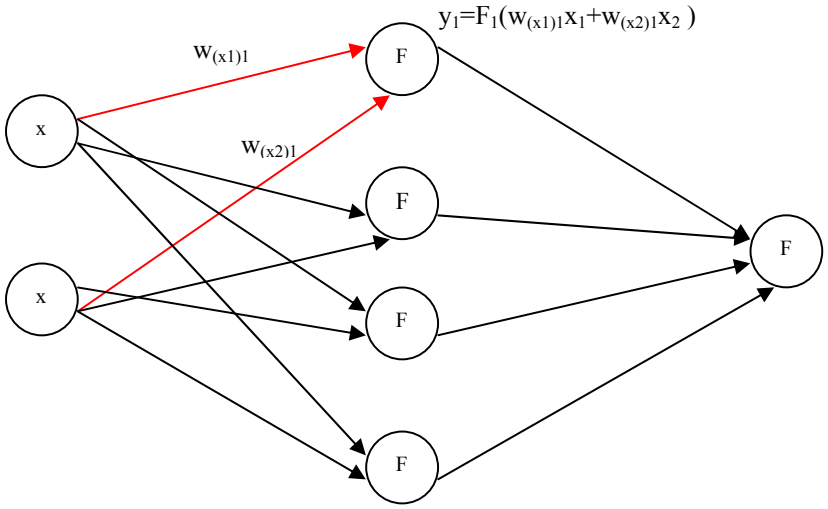
$$fonksiyon(x) = \frac{1}{2}\left[1 + \frac{x}{1+|x|}\right] \tag{6.2}$$



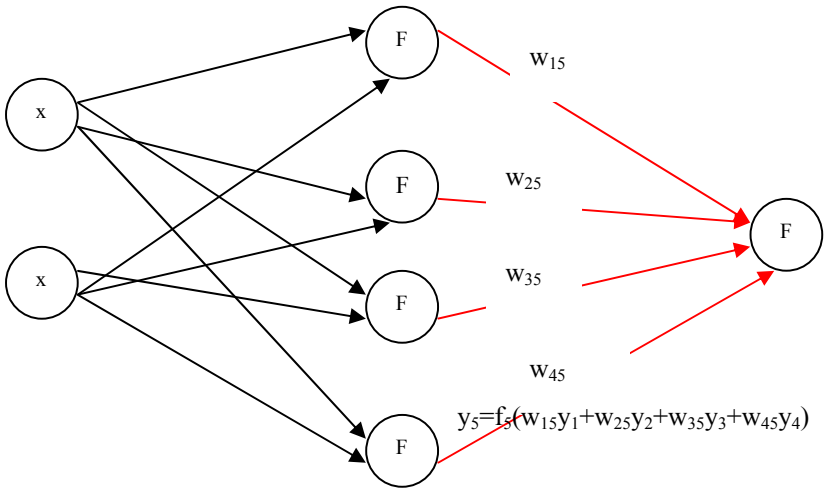
Şekil 6.8 Aktivasyon Fonksiyonu

6.3.3 Network Yapısı

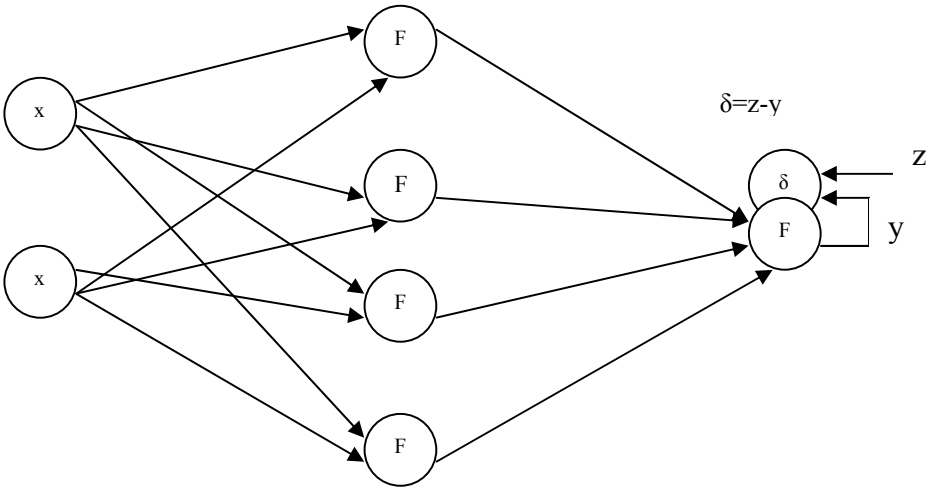
Ağ yapısı hatanın geriye yayılım algoritmasını kullanmaktadır. Ağırlık güncellemeleri ve hesaplamalar aşağıdaki şekillerde sıra ile gösterilmiştir.(Bernacci ve ark. 1999)



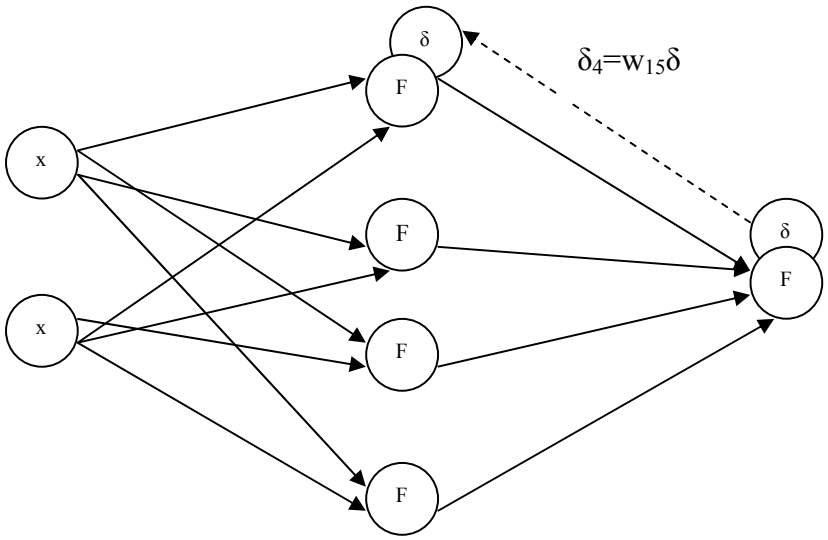
Şekil 6.9 Birinci katmanda ileri yönde ağ girişlerinin hesaplanması



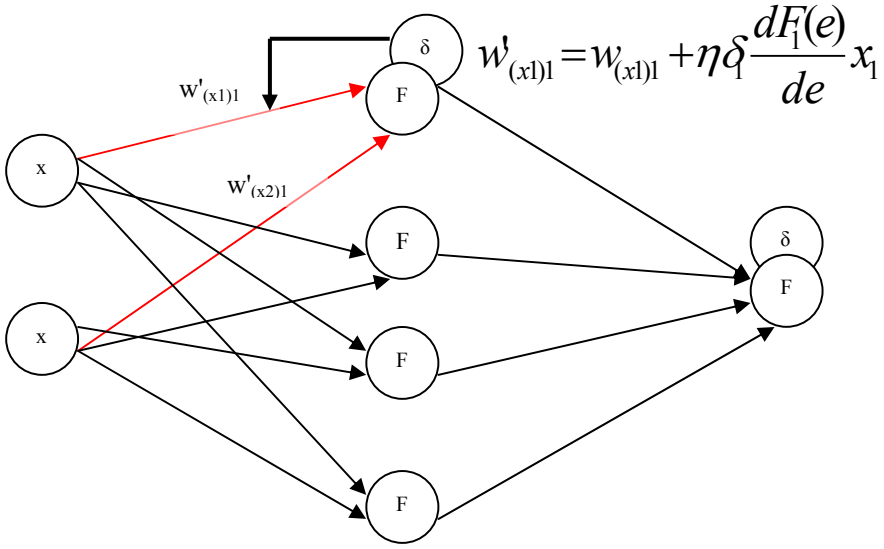
Şekil 6.10 İkinci (gizli) katmanda ileri yönde çıkış girişlerinin hesaplanması



Şekil 6.11 Oluşan çıkış ile beklenen çıkış arasındaki fark hesabı(hata hesabı)



Şekil 6.12 Hatanın geriye yayılım ile geri nöronlara yayılımı



$$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{dF_1(e)}{de} x_1$$

Şekil 6.13 Elde edilen değerleri ile yeni ağırlıkların hesabı

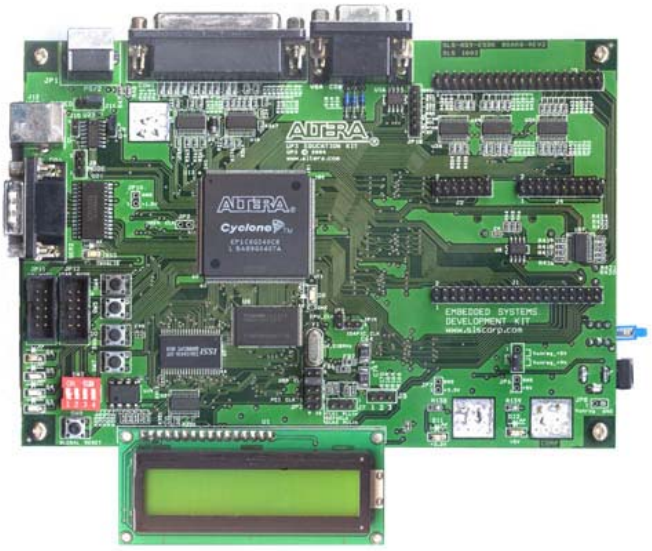
YSA içinde yer alan işlemler sıra ile gösterilmiştir. Bu bir devre gerçeklemesi olduğu için her bir adım içinde yapılacaklar belli olmalıdır. Kontroller ise programda kontrol ifadeleri ile yapılırken burada elle veya yine başka devre elemanları ile yapılmak zorundadır.

7 DENEYSEL SONUÇLAR

YSA'lar desen tanıma, sistem tanıma, durum tahmini gibi birçok uygulama da kullanılmaktadır. YSA'larda en çok kullanılan yapı tanım alanının karmaşıklığı sebebiyle çok katmanlı yapılardır. Bu yapılar genelde hatanın geriye yayılımı algoritmasını kullanırlar. YSA içinde önemli olan eğitim aşamasıdır. Eğitim oldukça uzun süre almaktadır. Gerçek zamanlı uygulamalar ise eğitim sürecinin kısalığını talep etmektedirler. Ayrıca bir YSA yapısı birçok farklı uygulama için de kullanılabilmektedir.

FPGA programlanabilen bir lojik elemandır ve yazılım desteği sunar. Bu da yapının yazılım esnekliğine sahip olması demektir. Yapı bir kez oluşturulduktan sonra sonsuz kez düzenlenebilir. FPGA'ler genel olarak donanım tasarımında ilk örnek olarak kullanılırlar. YSA dizayn ederken ağın eğitim hızı, sayısal duyarlılık, alan maliyeti arasında dengenin kurulması gerekmektedir.

Bu çalışmada yapay sinir ağlarının veri işleme kabiliyeti FPGA üzerinde gerçekleştirilmeye çalışılmıştır. Çalışmada Stratix II EP2S60F484C3 ve Cyclone EP1C6Q240C8 devreleri kullanılmıştır. Bu devrelerden Cyclone EP1C6Q240C8 deneme kartının fotoğrafı Şekil 7.1'de verilmiştir.



Şekil 7.1 Kullanılan FPGA kartı

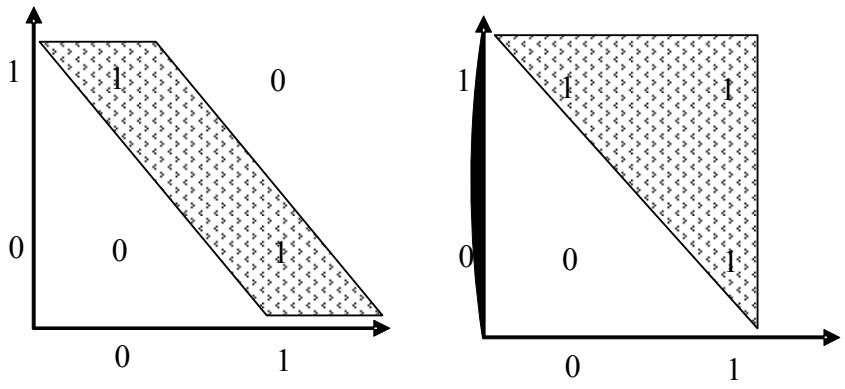
7.1.1 XOR Problemi

Bu çalışmada tasarımı yapılan ve FPGA üzerinden yürütülecek olan yapay sinir ağının istenilen şekilde çalışıp çalışmadığı, klasik bir problem olan XOR ile test edilmiştir.

ÖZEL VEYA (XOR) problemi sahip olduğu doğrusal olmayan yapı sayesinde sınıflama ve modelleme algoritmalarının test edilmesi ve başarılarının ölçülmesinde önemli bir ölçüt haline gelmiştir. Doğruluk tablosu Tablo 7.1'de verilen bu problemin 2 boyutlu uzaydaki gösterimi ise Şekil 7.2' de verilmiştir. Bu şekilden de anlaşılabileceği gibi basit bir doğru ile lojik 1 ve lojik 0 olan değerleri birbirlerinden ayrıştırılamamaktadır. Oysaki diğer lojik operatörler örneğin VE (AND), VEYA (OR) gibi işlemler Şekil 7.2b'den de görüleceği gibi bir doğru ile kolaylıkla sınırlanabilmektedir.

Tablo 7.1 XOR lojik operatörünün doğruluk tablosu

A	B	Y=A XOR B	Y=A OR B
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0



a) XOR operatörü b) OR operatörü

Şekil 7.2 XOR ve OR iki boyutlu gösterimi

7.1.1.1 XOR operatörünün VHDL uygulaması

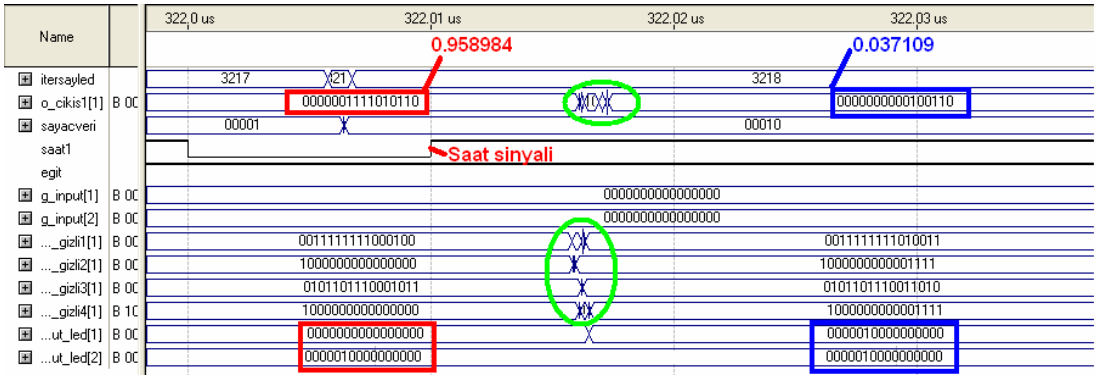
FPGA üzerinde eğitme ve test aşamaları gerçekleştirilen XOR problemi için oluşturulan YSA ; iki giriş, dört adet nöronu bulunan bir gizli katman ve bir çıkış (2:4:1) içerir. Bu seçilen mimaride Delta kuralı ve hatanın geri yayılımı algoritması eğitme için seçilmiştir.

Bu çalışmada yapay sinir ağlarının veri işleme kabiliyeti FPGA üzerinde gerçekleştirmek amaçlanmıştır. Çalışmada Stratix II EP2S60F484C3 ve Cyclone EP1C6Q240C8 devreleri kullanılmıştır. Devre, ayarlar(Settings) bölümünden otomatik olarak belirlenebildiği gibi elle de seçilebilmektedir. Yapılan 16 bit çalışma sonrasında Stratix II EP2S60F484C3 için donanım bilgileri aşağıdaki şekilde olmuştur.

Tablo 7.2 FPGA kaynak kullanımı

Register Sayısı	423 / 48,352 (< 1 %)
Pin Sayısı	183 / 335 (55 %)
LUT sayısı	20,428 / 48,352 (42 %)
9-bit DSP blok Sayısı	128 / 288 (44 %)

Eğitim esnasında yapılan tüm işlemler paraleldir. Eğitim veya test bilgisinin verilip verilmediği bir egit sinyali ile kontrol edilmektedir. Eğer egit sinyali 1 ise eğitim için belirlenen eğitim verileri her bir saat sinyaline bağlı bir sayaç yardımıyla alınmakta ve hata hesaplanıp ağırlıklar yenilenmektedir. Eğer eğit sinyali 0 değerine sahip ise dışarıdan verilen veriler eldeki ağırlık değerleri ile test edilmektedir. Başarının tam sağlanabilmesi için eğitim süresinin ve hesaplamada kullanılan saat sinyalinin yeteri değerlerde olması şarttır. Aksi takdirde devre doğru olmasına rağmen bir eğitim söz konusu olmamaktadır. Saat sinyalinin devrenin derlenmesi esnasında oluşan bilgiler sayesinde hangi değerden daha fazla olması gerektiği anlaşılabilmektedir. (register'lar arası veri gönderim zamanları).Eğitim ve test sonuçları simülasyon ile görülebilmektedir. Simülasyonda Quartus'un kendi simülasyon programı kullanılmıştır. Aşağıda simülasyon sonucu yer almaktadır.

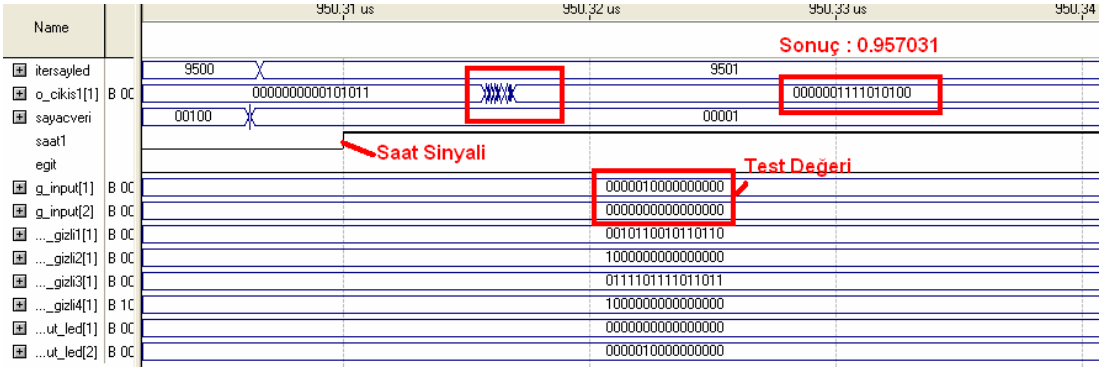


Şekil 7.3 Simülasyon eğitme sonucu

830 (3218/4) numaralı iterasyon bloğunda görüldüğü gibi eğitim başarı ile sağlanmaktadır. 830 iterasyon bloğu 322.03 mikro saniye sürmüştür. Saat1 sinyalinden önce 0-1 değeri eğitilmiş ve oluşan çıkış 0.958984 olmuştur. Saat1 sinyalinden sonra 1-1 değeri eğitimi için ağırlıklar güncellenmiş ve oluşan çıkış değeri 0.037109 olmuştur. Hatanın geriye yayılımı algoritması sonucunda ağırlık değişimi ise aşağıda yer alan daire içinde görülmektedir. Eğitim Şekil 7.3'de görüldüğü üzere başarı ile yapılmıştır. Test sonuçları ise Şekil 7.4'de gösterildiği gibidir.

Şekil 7.4 0-0 Simülasyon test sonucu

Egit sinyali 0 olduğunda artık test yapılabilir. İlk saat sinyalinde 0-0 değeri test edilmiştir. O_cikis değeri görüldüğü gibi 0.04199 şeklinde oluşmuştur.



Şekil 7.5 1-0 Simülasyon test sonucu

7.1.1.2 Matlab ile Karşılaştırma

Matlab matematiksel işlemler, yapay sinir ağları, fuzzy lojik, genetik algoritmaları gibi birçok işlemin gerçekleştirebildiği bir yazılım ortamıdır. Kullanıcılarına birçok kolaylık sunması sebebiyle sıkça tercih edilen bir program olmuştur. FPGA içinde yapılan YSA dizaynının doğruluk ve zaman olarak Matlab kodu ile karşılaştırılması mantıklıdır.

İlk karşılaştırma zaman açısındandır. FPGA içinde yer alan YSA dizaynı kendi toplayıcı ve çarpıcı devrelerini kullanırken, Matlab programı işlemciyi kullanmaktadır. FPGA içindeki işlemler yerden kazanmak amacı ile 16 bit olarak yapılmıştır. Fakat VHDL esnekliği sayesinde istenirse kısa bir zamanda değişiklik yapılıp daha fazla bit ile daha fazla duyarlılıkta işlemler yapmak mümkündür. Aşağıdaki tabloda belirtilen iterasyon zamanları değerleri verilmiştir.

Tablo 7.3 Eğitim süreleri karşılaştırması

İtersyon Sayısı	Matlab	FPGA
2249	2 sn(2 000 000 µs)	900 µs

Görüldüğü gibi FPGA içindeki YSA ile Matlab-YSA arasındaki eğitim zamanı farkı oldukça fazladır. Yaklaşık eğitim zamanı oranı 1/2000 dir. FPGA içindeki dizaynın daha da hızlandırılması mümkündür: Burada saat 200 ns olarak ayarlanmıştır. Saatin daha da hızlandırılması sonuçların doğru olarak elde edilememesine sebep olabilmektedir.

İkinci karşılaştırma doğruluk kontrolüdür. Doğruluk kontrolünde FPGA -YSA ile Matlab-YSA arasında bulunan bazı farklar dikkate alınmalıdır. Matlab'ın bulunduğu bilgisayar Intel(R) Core(TM) Duo Cpu T7300 @ 2.00 Ghz mikroişlemci iken FPGA üzerindeki YSA 16 bit ile işlem yapmaktadır. Bu sebeple noktadan sonraki 3 haneden sonraki sayılar arsında farklılık bulunması normal olarak karşılanmalıdır.

Tablo 7.4 Eğitim hatalarının karşılaştırması

İterasyon sayısı	Matlab	FPGA
2249	Maks. Hata 0.04 Min. Hata 0.02	Maks. Hata 0.06 Min. Hata 0.04

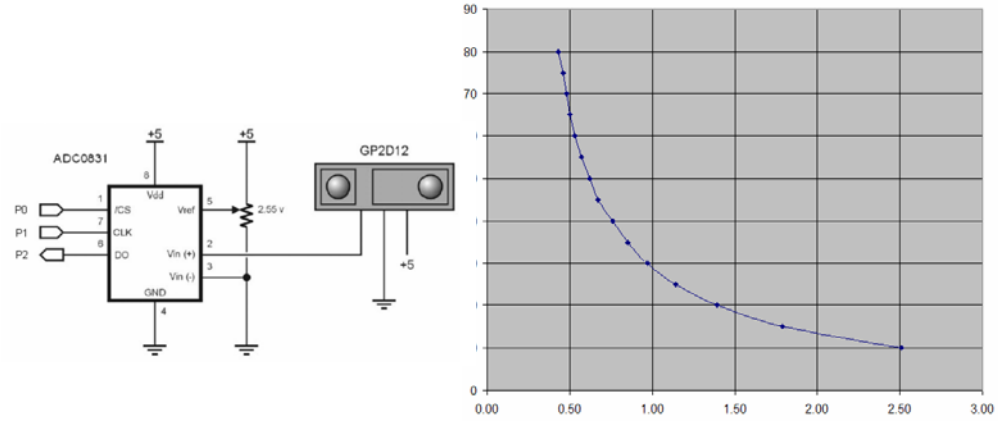
Matlab Kodu

```
clear all;
clc;
format long
giris=[0.0 0.0;0.0 1.0;1.0 0.0; 1.0 1.0]
cikis=[0.0;1.0;1.0;0.0]
[ep2,gs]=size(giris)
[ep1,cs]=size(cikis)
hs=5;
WG=double(rand(gs,hs));
WH=double(rand(hs,cs));
%neth=zeros(gs);
%netc=zeros(hs);
nu=0.4;
for itersay=1:1000000
for ep=1:ep1
g=giris(ep,:);
c=cikis(ep,:);
%yapay sinir ağı test ediliyor;
for j=1:hs
  neth(j)=0;
  for i=1:gs
      neth(j)=neth(j)+WG(i,j)*g(i);
   end
oh(j)=aktif(neth(j));
```

```
end
for k=1:cs
   netc(k)=0;
   for j=1:hs
      netc(k)=netc(k)+WH(j,k)*oh(j);
   end
oc(k)=aktif(netc(k));
end
e=c-oc;
fprintf('hata=%+f   iterasyon=%ld \n',e,itersay)
for k=1:cs
   deltac(k)=oc(k)*(1-oc(k))*(c(k)-oc(k));
end
for j=1:hs
   T=0;
   for k=1:cs
      T=T+deltac(k)*WH(j,k);
   end
   deltah(j)=oh(j)*(1-oh(j))*T;
end
for i=1:gs
   for j=hs
      WG(i,j)=WG(i,j)+nu*deltah(j)*g(i);
   end
end
for j=1:hs
   for k=cs
      WH(j,k)=WH(j,k)+nu*deltac(k)*oh(j);
   end
end
end
end
```

7.1.2 Sensör doğrusallaştırıcı

Robotik sistemlerde kullanılan pek çok sensordan biri olan lazer mesafe ölçüm sensoru, ölçmesi gereken büyüklük olan mesafe ile ters orantılı ve doğrusal olmayan bir çıkış üretir. Bu sensorların kalibrasyonu ve hassas ölçüm ihtiyaçları için kullanılması amacıyla üreticilerinin de önerdiği bazı yöntemleri mevcuttur. Bu uygulamada, uygulama devresi Şekil 7 .6 (a)'daki gibi üreticisi tarafından verilen sensör devresi ile yine üreticisi tarafından sağlanan ve Şekil 7.6 (b)'de verilen ölçüm karakteristiği kullanılarak bir doğrusallaştırma ve dönüştürme modülü oluşturmak amaçlanmıştır.



(a) (b)

Şekil 7.6 Sharp GP2D12 analog mesafe ölçme sensörü

Bu örnekte de bir önceki uygulamada oluşturulan FPGA üzerindeki YSA yapısı kullanılmıştır. Yani Delta-Bar-Delta öğrenme kuralı kullanan hatanın geri yayılımı algoritması ile eğitilen 2:7:1 yapay sinir ağı mimarisi kullanılmıştır. Girişin biri 43 ile 251 arasında bir değer olan sinyal çıkış değeri iken diğeri bias amaçlı sabit 1 değeridir. Çıkış olarak 80 ile 10 cm değerleri verilmiştir. Algoritma çalıştırılmadan önce giriş ve çıkış değerleri 0-1 aralığına normalize edilmiştir. Normalizasyon işlemleri aşağıdaki denklemlerde verilmiştir.

$$cikis = \frac{cikis - 10}{80 - 10} \qquad giris = \frac{giris - 43}{251 - 43} \qquad (7.1)$$

Bu uygulama oluşturulan FPGA modelinin elimizde mevcut bulunan UP3 geliştirme kitine sığmaması nedeniyle sadece simülasyon olarak gerçeklenmiştir.

Aynı zamanda algoritmanın parametrelerinin belirlenmesi aşamasında ve karşılaştırmalarda kullanılmak üzere bir Matlab programı yazılmış sonuçlar bu program ile karşılaştırılmıştır. Eğitme sonuçları Tablo 7.5, 7.6 ve 7.7 ‘de verilmiştir.

Tablo 7 .5 FPGA kaynak kullanımı

Register Sayısı	550 / 106 032 (< 1 %)
Pin Sayısı	231 / 743 (31 %)
LUT sayısı	47 105 / 106,032 (44 %)
9-bit DSP blok Sayısı	376 / 504 (75 %)

Tablo 7.6 Eğitim süreleri karşılaştırması

İtersyon Sayısı	Matlab	FPGA
100 000	4.74 dk	43 ms

Tablo 7.7 Eğitim hata değerleri

İterasyon sayısı	Matlab	FPGA
100 000	Maks. Hata 0.03 Min. Hata 0.01	Maks. Hata 0.05 Min. Hata 0.04

8 SONUÇ VE ÖNERİLER

Yapay sinir ağları insan beyninden esinlenmesi ve doğrusal bir yapıya sahip olmaması sebebiyle birçok uygulamada kullanılan farklı bir yapıdır. YSA uygulamalarında öğrenme ve test aşaması olarak iki aşama bulunur. Bu aşamalardan öğrenme aşaması oldukça karmaşık ve uzun bir süreçtir. Bu uzun süreyi kısaltmak için YSA yapısı VLSI teknolojisi kullanılarak gerçekleştirilebilir. Fakat bu yapının oluşturulması uzun zaman alır ve maliyeti oldukça yüksektir.

Bu aşamada FPGA devresi kullanmak mantıklıdır. FPGA ile yapılan dizaynların VLSI ile yapılan tasarımlardan maliyeti daha düşüktür, oluşturma zamanı daha kısadır ve değiştirme imkânı bulunmaktadır. FPGA tabanlı YSA' da düşük sayı duyarlılığı kullanılarak yapılan yoğun matematiksel işlem gerektirmeyen dizaynlar gerçek zamanlı uygulamalarda başarılı sonuçlar vermektedir.

Ayrıca kullanılan FPGA devresinin sonsuz kez düzenlenebilir olması ve VHDL ile büyük bir esneklik elde edilmesi bir avantajdır. Böylece bir dizayn kısa zamanda değiştirilerek birçok farklı uygulamada kullanılabilmektedir. FPGA'ların geleneksel işlemcilerin sahip olmadığı hız, güvenlik ve paralel işlem yapabilme yeteneğine ve VLSI teknolojinin sahip olmadığı düzenlenebilirlik kabiliyetine sahip olması sebebiyle yapay sinir ağları ile çok uyumlu çalışmalar yapılabilmekte ve yapay sinir ağları konusuna ışık tutmaktadır.

FPGA'lar üzerinde devre tasarlarken, mevcut olan VHDL, Verilog, Sematik dizayn yöntemlerinden devre tasarım ve tanımlama dili olan VHDL tercih edilmiştir. Bu sayede ileri donanım bilgisine ihtiyaç olmadan ve çok karmaşık tasarımların kolaylıkla tasarlanması mümkün olabilmektedir. VHDL dili yapı olarak farklı bir dil olmasına rağmen çok fazla donanım bilgisi olmayan biri tarafından da rahatça öğrenilip kullanılabilir. Bu da donanım bilgisi olmayanlarında donanım oluşturup uygulamasına fırsat tanır.

Bu çalışmada, çip üzerinde eğitilebilir bir YSA yapısı, Altera FPGA devreleri ile gerçekleştirilmiştir. XOR ve bir sensör doğrusallaştırma problemi ile çalışılmış ve Fixed-point sayı sistemi tabanlı ve hatanın geri yayılımı algoritması ile eğitilen bir

YSA yapısı kullanılmıştır. Öğrenme kuralı olarak delta bar delta kuralı seçilmiştir. Bu uygulamalar Altera'nın FPGA tasarım programı QUARTUS II ve MATLAB ile tasarlanmış ve simüle edilmiştir. Bunlara ek olarak, basitleştirilmiş YSA yapısı ile XOR problemi Altera Cyclone EP1C6Q240C8 FGPA tabanlı UP3 geliştirme kartı kullanılarak gerçekleştirilmiştir.

FPGA kullanılarak oluşturulan YSA yapısının oldukça kısa bir zamanda doğru sonuca ulaştığı yapılan deneylerle görülmüştür. Donanım tabanlı YSA tasarımlarında FPGA kullanılması son derece mantıklı ve doğru karardır. Ayrıca FPGA ile VHDL dili kullanılabilmesi sayesinde program yazarak dizayn oluşturmak mümkündür. Böylece dizaynın elle çizilmesine gerek kalmamaktadır.

FPGA içerisinde yer alan donanım tabanlı YSA, eğitim zamanını oldukça kısaltmıştır. Matlab ile günlerce sürecek olan eğitim zamanı FPGA-YSA ile bir-iki dakikada yapılabilmektedir.

Bu çalışma ile bazı YSA tabanlı sistemler için FPGA'nın maliyet, zaman tasarrufu, tekrar düzenlenebilirlik ve paralel tasarım yeteneği açılarından daha uygun bir çözüm olduğu gösterilmiştir.

İleriki çalışmalarda, endüstriyel bir problem üzerinden YSA tabanlı bir çözümü üretirken Altera ve Xilinx firmalarının yeni nesil FPGA'larını kullanmak tasarım konusunda daha da esnek bir çalışma ortamı sağlayacaktır. Bu sayede elde edilen tasarımın düşük kapasiteli FPGA' lere sığmama problemleri yaşanmayacaktır.

Bu çalışmada çalışılan problemler, paralel çalışma yapısından ödün verilerek aynı fonksiyonların daha az devre elemanı ile yerine getirilmesi çalışmaları da ayrı bir araştırma konusu olabilir. Bu çalışmalar neticesinde, nispeten yavaş ama daha düşük kapasiteli FPGA devrelerinin kullanılmasını mümkün kılan tasarımlar elde edilebilecektir.

9 KAYNAKLAR

Bernacki M., Wlodarczyk P. 1999, Neural Network and Artifical İntellegence in electronics, Report.

Burr , J. 1993. Digital Neurochip Design in Parallel Digital Implementations of Neural Networks. Prentice Hall Inc..

Burr , J. 1993. Digital Neurochip Dsign in Parallel Digital Implementations of Neural Networks, Prentice Hall Inc., pp 223-1214.

Çavuşoğlu M.A. 2006. FPGA ile Yapay Sinir Ağı Eğitiminin Donanımsal Olarak Gerçekleştirilmesi .Yüksek Lisans Tezi. Kocaeli Üniversitesi.

de Garisa H., Korkinb M. 2002. .The CAM-Brain Machine (CBM): an FPGA-based hardware tool that evolves a 1000 neuron-net circuit module in seconds and updates a 75 million neuron arti-cial brain for real-time robot control . Neurocomputing, Elsevier Sci., pp.35-68.

Ethem Bilgin, İbrahim Özçelik, Murat İskefiyeli, FPGA ve VHDL Nedir?, http://www.bilesim.com.tr/mistoportal/ showmakale .nsf ?xd =2296.xml

Giron'es R.G., Palero R.C., Cerd J. 2004. FPGA Implementation of a Pipelined On-Line Backpropagation. A Boluda And Angel Sebastia Cort´es. 2004

Gutierrez,W. J., Grandin, R.D. 1989. *Estimating hidden units for two layer perceptrons*. Proc. of the 1st

Haykin, S. 1994. Neural Networks A Comprehensive Foundation, Prentice Hall Publishing, New Jersey , USA, Vol.1, pp 1-14.

Haykin, S. 1999. Neural Networks A Comprehensive Foundation. 2nd edition, Prentice Hall Publishing, New Jersey 07458, USA, Vol.1, pp 6-7.

http://herseyekitap.googlepages.com/YapaySinirAglari.doc

http://users.ece.gatech.edu/~hamblen/UP3/

http://www.altera.com

www.yapayzeka.org/modules/mydownloads/viewcat.php?cid=5&orderby=titleD

Meyer-Bease U. 2001. Digital Signal Processing with Field Programmable Gate Arrays. Springer –Verlag Berlin.

Moving NN Triggers to Level-1 at LHC Rates . 2003 Elsevier Science .

Özbay Y.,1999. EKG Aritmilerini Hızlı Tanıma. Doktora Tezi. Selçuk Üniversitesi, Fenbilimleri Enstitüsü.

Öztemel, E., 2003. Yapay Sinir Ağları, Papatya Yayıncılık, İstanbul.

Prlevotet J.-C., Denby B., Garda P., Granado B., Kiesling C.,2003,

Show A.W. 1993, Logic Circuit Design, Sounder Collega Publishing. UK.

Won E. .2007 . A hardware implementation of artificial neural networks using field programmable gate arrays . Nuclear Instruments and Methots in Physics Research. 581, pp.816-820.

Yu, X., Deni D. 1994. Implementing Neural Networks In FPGAs, The Institution of Electrical Engineers. IEE published, Savoy Place, London WC2R 0BL,

EKLER

EK A -BASİTLEŞTİRİLMİŞ FONK1.VHDL PAKETİ

```
library ieee ;
use ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.std_logic_unsigned.ALL;
use ieee.math_real.all;
use ieee.math_utility_pkg.all;
use ieee.numeric_std.all;
use ieee.fixed_pkg.all;

package  fonk1 is

   constant g_node_sayi : integer :=2;-- 8;--giris node sayisi
   constant h_node_sayi : integer := 4;--gizli katman node sayisi
   constant c_node_sayi:integer:=2; --cikis node sayisi
   constant giris_uzunluk:integer:=1;--1;--giris degeri uzunlugu
   constant agirlik_uzunluk:integer:=16;
   constant tam :integer:=5;   --agirlik degeri uzunlugu
   constant noktali:integer:=-10;
   constant cikis_uzunluk:integer:=1;
   constant iterasyon_sayi :integer:=10; --iterasyon sayisi
   constant g_cikis_uzunluk:integer:=1;

--agirlik , giris ,hedef cikis , olusan cikis iki boyutlu dizisi
subtype fixsayi is sfixed (5 downto -10);
subtype say is integer range 0 to 255;
type agirliklar_gir is array (1 to h_node_sayi ) of fixsayi;
type agirliklar_giris is array (1 to g_node_sayi) of agirliklar_gir;
type giris_agirlikno is array(1 to 4) of agirliklar_giris;

--agirlik  hidden
type agirliklar_h is array (1 to c_node_sayi ) of fixsayi;
type agirliklar_hidden is array (1 to h_node_sayi) of agirliklar_h;
type gizli_agirlikno  is array (1 to 4 ) of agirliklar_hidden;
-------------------------------------------------------
type delta is array (1 to c_node_sayi) of  fixsayi;
type girisler is array (1 to g_node_sayi) of fixsayi;
type girisno is array ( 1 to 4 ) of girisler;
-------------------------------------------------------
type cikis_c is array (1 to c_node_sayi) of fixsayi;
type hedefno is array (1 to 4) of cikis_c;
type cikis_g is array (1 to h_node_sayi) of fixsayi;

constant ogr_oran:fixsayi:=to_sfixed(0.7,5,-10);
-------------------------------------------------------

-------------------------------------------------------
function carpma_h(a:fixsayi; b:fixsayi)return fixsayi   ;
function topla_h(a:fixsayi;            b:fixsayi;ilk:STD_LOGIC)return fixsayi;
function cikar_h(a: fixsayi; b:fixsayi)return fixsayi;
function hardlin(A: fixsayi) return fixsayi;
```

```
function giris(g_input:girisler;g_agirlik:agirliklar_giris;ilk:std_logic) return cikis_g ;
function cikis(cikis_gizli :cikis_g;agirlik_c:agirliklar_hidden;ilk:std_logic) return cikis_c;

------------------------------------------------------

end fonk1;

library IEEE;
use IEEE.std_logic_1164.all;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.std_logic_unsigned.ALL;
use ieee.math_utility_pkg.all;
use ieee.numeric_std.all;
use ieee.fixed_pkg.all;
use ieee.math_real.all;

package body fonk1 is

function carpma_h(a: fixsayi; b:fixsayi)return fixsayi  is
variable pdt,pdt1:            fixsayi;
variable bias:fixsayi;
begin
   pdt:=resize(a*b,5,-10);
   return  pdt ;
end carpma_h;

function topla_h (a:fixsayi;b:fixsayi;ilk: STD_LOGIC) return fixsayi is
          variable pdt : fixsayi;
          variable sifir:fixsayi :=(others=>'0');
begin
           case (ilk)  is
           when '1' =>
                pdt :=(others=>'0');
                return sifir;
           when '0'=>
                pdt:=resize(a+b,5,-10);
                return  pdt ;
           end case;
end topla_h;

function cikar_h(a: fixsayi; b:fixsayi)return fixsayi  is
variable pdt:       fixsayi;
begin
   pdt:=resize(a-b,5,-10);
   return  pdt ;
end cikar_h;

function hardlin(A: fixsayi) return fixsayi is
variable d1,d5,d4:fixsayi;
begin
 --1/2 (1 + x /(1+|x|))
  if A<0 then d1:=A;d1(5):='0'; else d1:=A;end if;
  d4:=resize(1+A/(1+d1),5,-10);
  d5:=resize(d4/2,5,-10);
return d5;
end hardlin;
```

```
function giris(g_input:girisler;g_agirlik:agirliklar_giris;ilk:std_logic)
            return cikis_g is
variable carpma_sonuc: fixsayi;
variable toplama1:fixsayi;
variable sifir:fixsayi:=(others=>'0');
variable cikis_gizli: cikis_g;
variable i,j:integer;
begin
for i in 1 to h_node_sayi loop
 toplama1:=(others=>'0');
 for j in 1 to g_node_sayi loop
    carpma_sonuc:=carpma_h(g_input(j),g_agirlik(j)(i));
    toplama1:=topla_h(carpma_sonuc,toplama1,ilk);
 end loop;
 cikis_gizli(i):=hardlin(toplama1);
end loop;
return cikis_gizli;
end giris;

function cikis(cikis_gizli:cikis_g;agirlik_c:agirliklar_hidden;ilk:std_logic) return cikis_c is
 variable carpma_sonuc:fixsayi;
 variable toplama1:fixsayi;
 variable c_cikis:cikis_c;
 variable i,j:integer;
begin
for i in 1 to c_node_sayi loop
toplama1:=(others=>'0');
 for j in 1 to h_node_sayi loop
    carpma_sonuc:=carpma_h(cikis_gizli(j),agirlik_c(j)(i));
    toplama1:=topla_h(carpma_sonuc,toplama1,ilk);
 end loop;
 c_cikis(i):=hardlin(toplama1);
end loop;
return c_cikis;
end cikis;


end fonk1;
```

EK B. BASİTLEŞTİRİLMİŞ YSA KOD ÖRNEĞİ

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.std_logic_unsigned.ALL;
use ieee.math_real.all;
use ieee.numeric_std.all;

library gizli_neron;
use gizli_neron.fonk1.all;
use gizli_neron.math_utility_pkg.all;
use gizli_neron.fixed_pkg.all;

entity gizli_neron is
port
	(
				g_input:in girisler;
				g_input_led:out girisler;
				egit :in std_logic;
				saat1:in std_logic;
				cikis_gizli1:buffer agirliklar_h;
	--			cikis_gizli2:buffer agirliklar_h;
	--			cikis_gizli3:buffer agirliklar_h;
	--			cikis_gizli4:buffer agirliklar_h;
			itersayled:out integer range 0 to 255;
	--			sayacveri:out integer range 0 to 255;
				o_cikis1:buffer cikis_c;
				deltany: out cikis_c
);
end gizli_neron ;

architecture test  of gizli_neron is

 signal o_cikis:cikis_c;
 signal cikis_s:cikis_c;
--signal giris_agirlik_s:agirliklar_giris;
--signal gizli_agirlik_s:agirliklar_hidden;
 signal sayac:say:=0;
 signal itersay:integer:=0;
 signal hedef:hedefno;
 signal g_input1:girisno;
 signal giris_agirlik_s:agirliklar_giris;
 signal gizli_agirlik_s:agirliklar_hidden;
 signal giris_agirlik_e:agirliklar_giris;
 signal gizli_agirlik_e:agirliklar_hidden;
 --signal deltany:fixsayi;

 begin -----
 g_input1(1)(1)<=to_sfixed(1,tam,noktali);g_input1(1)(2)<=to_sfixed(0,tam,noktali);
 g_input1(2)(1)<=to_sfixed(0,tam,noktali);g_input1(2)(2)<=to_sfixed(1,tam,noktali);
 g_input1(3)(1)<=to_sfixed(1,tam,noktali);g_input1(3)(2)<=to_sfixed(1,tam,noktali);
 g_input1(4)(1)<=to_sfixed(0,tam,noktali);g_input1(4)(2)<=to_sfixed(0,tam,noktali);
```

```
 hedef(1)(1)<=to_sfixed(1,tam,noktali);--hedef(1)(2)<=to_sfixed(0,tam,noktali);
 hedef(2)(1)<=to_sfixed(1,tam,noktali);--hedef(2)(2)<=to_sfixed(0,tam,noktali);
 hedef(3)(1)<=to_sfixed(0,tam,noktali);--hedef(3)(2)<=to_sfixed(0,tam,noktali);
 hedef(4)(1)<=to_sfixed(0,tam,noktali);--hedef(4)(2)<=to_sfixed(0,tam,noktali);

 --gizli_agirlik_s(1)(1)<=to_sfixed(0.5,tam,noktali);--
gizli_agirlik_s(1)(2):=to_sfixed(0.5,tam,noktali);
 --gizli_agirlik_s(2)(1)<=to_sfixed(0.5,tam,noktali);--
gizli_agirlik_s(2)(2):=to_sfixed(0.5,tam,noktali);
 --gizli_agirlik_s(3)(1)<=to_sfixed(0.5,tam,noktali);--
gizli_agirlik_s(3)(2):=to_sfixed(0.5,tam,noktali);
 --gizli_agirlik_s(4)(1)<=to_sfixed(0.5,tam,noktali);--
gizli_agirlik_s(4)(2):=to_sfixed(0.5,tam,noktali);


 --
giris_agirlik_s(1)(1)<=to_sfixed(0.5,tam,noktali);giris_agirlik_s(1)(2)<=to_sfixed(0.5,tam,noktali);
 --
giris_agirlik_s(1)(3)<=to_sfixed(0.5,tam,noktali);giris_agirlik_s(1)(4)<=to_sfixed(0.5,tam,noktali);
 --
giris_agirlik_s(2)(1)<=to_sfixed(0.5,tam,noktali);giris_agirlik_s(2)(2)<=to_sfixed(0.5,tam,noktali);
 --
giris_agirlik_s(2)(3)<=to_sfixed(0.5,tam,noktali);giris_agirlik_s(2)(4)<=to_sfixed(0.5,tam,noktali);

process
--variable hata:hedefno;
variable yeni,gizli_agirlik:agirliklar_hidden;
variable giris_agirlik:agirliklar_giris;
--variable sayi1,sayi2,sayi3:fixsayi;
variable cikis_gizli: cikis_g;
variable o_cikis:cikis_c;
variable i,j,k:integer range 0 to 255;
variable agirlik_farki,agirlik_farki1,d1,d2,d3,d4: fixsayi;
variable normal_agirlik,f1,fark1,n_agirlik1,f2,f3,f4:fixsayi;
variable n_agirlik ,sum ,delta1 ,d:fixsayi;
variable agirlik_yeni : agirliklar_hidden;
variable delta_r:delta;
variable hedefara:cikis_c;
variable girisara:girisler;
--variable giris_agirlik_s:agirliklar_giris;
--variable gizli_agirlik_s:agirliklar_hidden;
variable delta_k:cikis_c;
variable delta_j:cikis_g;
variable toplams : t_topla;
begin

if (itersay=0) then
 gizli_agirlik_s(1)(1)<=to_sfixed(0.5,tam,noktali);
 gizli_agirlik_s(2)(1)<=to_sfixed(0.5,tam,noktali);
 gizli_agirlik_s(3)(1)<=to_sfixed(0.5,tam,noktali);
 gizli_agirlik_s(4)(1)<=to_sfixed(0.5,tam,noktali);

 giris_agirlik_s(1)(1)<=to_sfixed(0.5,tam,noktali);giris_agirlik_s(1)(2)<=to_sfixed(0.5,tam,noktali);
 giris_agirlik_s(1)(3)<=to_sfixed(0.5,tam,noktali);giris_agirlik_s(1)(4)<=to_sfixed(0.5,tam,noktali);
 giris_agirlik_s(2)(1)<=to_sfixed(0.5,tam,noktali);giris_agirlik_s(2)(2)<=to_sfixed(0.5,tam,noktali);
 giris_agirlik_s(2)(3)<=to_sfixed(0.5,tam,noktali);giris_agirlik_s(2)(4)<=to_sfixed(0.5,tam,noktali);
end if;
```

```
--wait until  ((saat1'event) and (saat1='1'));
wait until  (saat1='1');
case (egit) is
when '0'=>

toplams(1):=resize((g_input(1)*giris_agirlik_s(1)(1))+(g_input(2)*giris_agirlik_s(2)(1)),tam,noktali)
;
cikis_gizli(1):=hardlin(toplams(1));
toplams(2):=resize((g_input(1)*giris_agirlik_s(1)(2))+(g_input(2)*giris_agirlik_s(2)(2)),tam,noktali)
;
cikis_gizli(2):=hardlin(toplams(2));
toplams(3):=resize((g_input(1)*giris_agirlik_s(1)(3))+(g_input(2)*giris_agirlik_s(2)(3)),tam,noktali)
;
cikis_gizli(3):=hardlin(toplams(3));
toplams(4):=resize((g_input(1)*giris_agirlik_s(1)(4))+(g_input(2)*giris_agirlik_s(2)(4)),tam,noktali)
;
cikis_gizli(4):=hardlin(toplams(4));
toplams(5):=resize((cikis_gizli(1)*gizli_agirlik_s(1)(1))+(cikis_gizli(2)*gizli_agirlik_s(2)(1)),tam,no
ktali);
toplams(6):=resize((cikis_gizli(3)*gizli_agirlik_s(3)(1))+(cikis_gizli(4)*gizli_agirlik_s(4)(1)),tam,no
ktali);
toplams(7):=resize((toplams(5)+toplams(6)),tam,noktali);
o_cikis(1):=hardlin(toplams(7));
o_cikis1<=o_cikis;

when '1'=>

girisara:=g_input1(sayac);
toplams(1):=resize((girisara(1)*giris_agirlik_s(1)(1))+(girisara(2)*giris_agirlik_s(2)(1)),tam,noktali
);
cikis_gizli(1):=hardlin(toplams(1));
toplams(2):=resize((girisara(1)*giris_agirlik_s(1)(2))+(girisara(2)*giris_agirlik_s(2)(2)),tam,noktali
);
cikis_gizli(2):=hardlin(toplams(2));
toplams(3):=resize((girisara(1)*giris_agirlik_s(1)(3))+(girisara(2)*giris_agirlik_s(2)(3)),tam,noktali
);
cikis_gizli(3):=hardlin(toplams(3));
toplams(4):=resize((girisara(1)*giris_agirlik_s(1)(4))+(girisara(2)*giris_agirlik_s(2)(4)),tam,noktali
);
cikis_gizli(4):=hardlin(toplams(4));
toplams(5):=resize((cikis_gizli(1)*gizli_agirlik_s(1)(1))+(cikis_gizli(2)*gizli_agirlik_s(2)(1)),tam,no
ktali);
toplams(6):=resize((cikis_gizli(3)*gizli_agirlik_s(3)(1))+(cikis_gizli(4)*gizli_agirlik_s(4)(1)),tam,no
ktali);
toplams(7):=resize((toplams(5)+toplams(6)),tam,noktali);
o_cikis(1):=hardlin(toplams(7));
o_cikis1<=o_cikis;

hedefara:=hedef(sayac);
  --    cikis_gizli:=giris(girisara,giris_agirlik_s);
  --    o_cikis:=cikis(cikis_gizli,gizli_agirlik_s);
delta_k(1):=resize(o_cikis(1)*(1-o_cikis(1))*(hedefara(1)-o_cikis(1)),tam,noktali);
gizli_agirlik_s(1)(1)<=resize(gizli_agirlik_e(1)(1)+(ogr_oran*delta_k(1)*cikis_gizli(1)),tam,noktali)
;
gizli_agirlik_s(2)(1)<=resize(gizli_agirlik_e(2)(1)+(ogr_oran*delta_k(1)*cikis_gizli(2)),tam,noktali)
;
```

```
gizli_agirlik_s(3)(1)<=resize(gizli_agirlik_e(3)(1)+(ogr_oran*delta_k(1)*cikis_gizli(3)),tam,noktali)
;
gizli_agirlik_s(4)(1)<=resize(gizli_agirlik_e(4)(1)+(ogr_oran*delta_k(1)*cikis_gizli(4)),tam,noktali)
;

delta_j(1):=resize((1-cikis_gizli(1))*cikis_gizli(1)*delta_k(1)*gizli_agirlik_s(1)(1),tam,noktali);      --
delta
delta_j(2):=resize((1-cikis_gizli(2))*cikis_gizli(2)*delta_k(1)*gizli_agirlik_s(2)(1),tam,noktali);      --
delta
delta_j(3):=resize((1-cikis_gizli(3))*cikis_gizli(3)*delta_k(1)*gizli_agirlik_s(3)(1),tam,noktali);      --
delta
delta_j(4):=resize((1-cikis_gizli(4))*cikis_gizli(4)*delta_k(1)*gizli_agirlik_s(4)(1),tam,noktali);      --
delta

giris_agirlik_s(1)(1)<=resize(giris_agirlik_e(1)(1)+(ogr_oran*delta_j(1)*girisara(1)),tam,noktali);
giris_agirlik_s(1)(2)<=resize(giris_agirlik_e(1)(2)+(ogr_oran*delta_j(2)*girisara(1)),tam,noktali);
giris_agirlik_s(1)(3)<=resize(giris_agirlik_e(1)(3)+(ogr_oran*delta_j(3)*girisara(1)),tam,noktali);
giris_agirlik_s(1)(4)<=resize(giris_agirlik_e(1)(4)+(ogr_oran*delta_j(4)*girisara(1)),tam,noktali);
giris_agirlik_s(2)(1)<=resize(giris_agirlik_e(2)(1)+(ogr_oran*delta_j(1)*girisara(2)),tam,noktali);
giris_agirlik_s(2)(2)<=resize(giris_agirlik_e(2)(2)+(ogr_oran*delta_j(2)*girisara(2)),tam,noktali);
giris_agirlik_s(2)(3)<=resize(giris_agirlik_e(2)(3)+(ogr_oran*delta_j(3)*girisara(2)),tam,noktali);
giris_agirlik_s(2)(4)<=resize(giris_agirlik_e(2)(4)+(ogr_oran*delta_j(4)*girisara(2)),tam,noktali);
 ------------
                    deltany<=delta_k;
                    --o_cikis1<=o_cikis;
                    g_input_led<=girisara;
                    cikis_gizli1<=gizli_agirlik_s(1);
--                  cikis_gizli2<=gizli_agirlik_s(2);
--                  cikis_gizli3<=gizli_agirlik_s(3);
--                  cikis_gizli4<=gizli_agirlik_s(4);
end case;
end process ; --ysatest

process
begin
wait until  ((saat1'event) and (saat1='0'));
itersay<=itersay+1;
sayac<=itersay mod 4+1;
--sayac<=sayac+1;
itersayled<=itersay;
--sayacveri<=sayac;
end process ;


end test;
```