T.C.
DÜZCE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ELEKTRİK EĞİTİM ANABİLİM DALI

YAPAY SİNİR AĞLARININ OTOMATİK OLARAK FPGA ÇİPİNE UYGULANMASI İÇİN DENETLEYİCİ TASARIM ARACI

YÜKSEK LİSANS TEZİ

GÜNAY TEMÜR

ŞUBAT 2013

DÜZCE

KABUL VE ONAY BELGESİ

Günay TEMÜR tarafından hazırlanan "Yapay Sinir Ağlarının Otomatik Olarak FPGA Çipine Uygulanması İçin Denetleyici Tasarım Aracı" isimli lisansüstü tez çalışması, Düzce Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun 30/01/2013 tarih ve 2013'e 40 sayılı kararı ile oluşturulan jüri tarafından Elektrik Eğitimi Anabilim Dalı'nda Yüksek Lisans Tezi olarak kabul edilmiştir.

Üye
(Tez Danışmanı)
Doç. Dr. İbrahim ŞAHİN
Düzce Üniversitesi

Üye
Doç. Dr. Pakize ERDOĞMUŞ
Düzce Üniversitesi

Üye
Yrd. Doç. Dr. Devrim AKGÜN
Düzce Üniversitesi

Tezin Savunulduğu Tarih : 01/02/2013

ONAY

Bu tez ile Düzce Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu Günay TEMÜR'ün Elektrik Eğitimi Anabilim Dalı'nda Yüksek Lisans derecesini almasını onamıştır.

Doç. Dr. Haldun MÜDERRİSOĞLU
Fen Bilimleri Enstitüsü Müdürü

BEYAN

Bu tez çalışmasının kendi çalışmam olduğunu, tezin planlanmasından yazımına kadar bütün aşamalarda etik dışı davranışımın olmadığını, bu tezdeki bütün bilgileri akademik ve etik kurallar içinde elde ettiğimi, bu tez çalışmasıyla elde edilmeyen bütün bilgi ve yorumlara kaynak gösterdiğimi ve bu kaynakları da kaynaklar listesine aldığımı, yine bu tezin çalışılması ve yazımı sırasında patent ve telif haklarını ihlal edici bir davranışımın olmadığını beyan ederim.

1 Şubat 2013

Günay TEMÜR

TEŞEKKÜR

Yüksek lisans öğrenimim ve bu tezin hazırlanması süresince gösterdiği her türlü destek ve yardımdan dolayı çok değerli danışman hocam Doç. Dr. İbrahim ŞAHİN'e, bilgi ve tecrübeleri ile desteklerini esirgemeyen değerli hocalarım Doç. Dr. Pakize ERDOĞMUŞ'a ve Yrd. Doç. Dr. Devrim AKGÜN'e en içten dileklerimle teşekkür ederim.

Tez çalışmam boyunca değerli katkılarını esirgemeyen sevgili eşim Öğr. Grv. Ayşe SOY TEMÜR'e ve mesai arkadaşım Öğr. Grv. Haydar GÖKSU'ya en içten minnet duygularımı sunarım.

ŞUBAT 2013 **Günay TEMÜR**

İÇİNDEKİLER **Sayfa**

ŞEKİL LİSTESİ

Sayfa No

ÇİZELGE LİSTESİ

Sayfa No

SİMGE VE KISALTMALAR LİSTESİ

ANN	Artificial Neural Networks (Yapay Sinir Ağları)
ANNCONT	Artificial Neural Network CONTroller
ANNGEN	Artificial Neural Network GENerator
ANNSYS	Artificial Neural Network SYStem
ASIC	Application Specific Integrated Circuits
ASSP	Application-Specific Standard Parts
CLB	Configurable Logic Blocks (Yapılandırılabilir Mantıksal Bloklar)
Clk	Clock (Saat Darbesi)
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit (Merkezi İşlem Birimi)
ÇKA	Çok Katmanlı Algılayıcı
DTD	Donanım Tanımla Dilleri
EDA	Electronic Design Automation (Otomatik Elektronik Tasarım Aracı)
F(x)	Transfer Fonksiyonu
FPGA	Field Programmable Gate Array (Sahada Programlanabilir Kapı Dizileri)
FSM	Finite State Machine (Sonlu Durum Makinesi)
GAL	Generic Array Logic
GY	Geriye Yayılım
HDL	Hardware Description Language (Donanım Tanımlama Dili)
IOB	Input/Output Blocks (Giriş/Çıkış Blokları)
IP	Intellectual Property
JTAG	Joint Test Action Group

LOGS	Log- Sigmoid (Logaritmik Sigmoid)
LUT	Look-Up Table (Bakılan Tablo)
PAL	Programmable Array Logic
PLA	Programmable Logic Array
PLD	Programmable Logic Device
RC	Reconfigurable Computing (Yeniden Yapılandırılabilir Hesaplama)
SoC	System on a Chip (Çip Üzerinde Sistem)
US	Uzman Sistemler
VHDL	Very High Speed Integrated Circuit HDL (Çok Yüksek Hızlı Entegre Devre Donanım Tanımlama Dili)
VLSI	Very Large Scale Integration (Çok Büyük Ölçekli Entegrasyon)
YSA	Yapay Sinir Ağı, Yapay Sinir Ağları
YZ	Yapay Zekâ
a_i	Giriş katmanı çıkışı
B_s	Eşik değeri
h	Katman türü sayısı
j	Katman türü içindeki element türü sayısı
k	Katman türü içindeki element türü içindeki katman sayısı
m	Katman sayısı
n_i	Giriş değerleri ile ağırlık değerlerinin çarpımlarının toplamı
r	Sinir hücresi sayısı
X_i	Girişler
$W_{i,s}$	Ağırlık değerleri
y_i	Çıkış değeri
z	Çıkış sayısı
α	Sabit katsayı

ÖZET

YAPAY SİNİR AĞLARININ OTOMATİK OLARAK FPGA ÇİPİNE UYGULANMASI İÇİN DENETLEYİCİ TASARIM ARACI

Günay TEMÜR
Düzce Üniversitesi
Fen Bilimleri Enstitüsü, Elektrik Eğitimi Anabilim Dalı
Yüksek Lisans Tezi
Danışman: Doç. Dr. İbrahim ŞAHİN
Şubat 2013, 92 sayfa

Yapay Sinir Ağları (YSA)'lar insan beyninin ufak bir kopyası gibidirler ve çok değişik alanlarda etkili bir şekilde kullanılmaktadırlar. Yazılımsal olarak gerçeklenen YSA'ların istenen performansı vermediği durumlarda donanımsal YSA uygulamaları tercih edilmektedir. YSA'lar gerçek performanslarını ancak paralel çalışan mimariler üzerinde gösterebilen sistemlerdir. FPGA'lar (Field Programable Gate Array-Sahada Programlanabilir Kapı Dizileri) ise özellikle paralel işlem gerektiren uygulamalarda yaygın olarak kullanılan donanım elemanlarıdır. Bu durum YSA'ların FPGA çiplerine uygulanmasını kaçınılmaz kılmaktadır. Fakat YSA'ların FPGA çiplerine uygulanması uzun zaman alan ve uzman gereksinimine ihtiyaç duyan bir süreçtir. Ayrıca uygulama aşamasında oluşabilecek yazılımsal hatalar, hata düzenleme (debug) aşamasını gerektirmektedir. Bu çalışmada YSA'ların otomatik olarak FPGA çipine uygulanması için bir denetleyici tasarım aracı olan ANNCONT (Artificial Neural Network Controller) ve var olan YSA veri yolu ile birleştirerek YSA sistemini oluşturan ANNSYS (Artificial Neural Network System) geliştirilmiştir. Burada amaç; YSA'ların FPGA'lara uygulanmasını otomatikleştirerek tasarım ve uygulama süresini kısaltmak, debug aşamasını ve uzman gereksinimini ortadan kaldırmaktır. Geliştirdiğimiz tasarım aracı için iki adet örnek test durumu oluşturulmuştur. ANNCONT ve ANNSYS örnek test durumları üzerinde çalıştırılarak saniyeler içinde YSA sistemleri için VHDL (Very High Speed Integrated Circuit HDL (Hardware Description Language) (Çok Yüksek Hızlı Entegre Devre Donanım Tanımlama Dili)) kodu üretilmiştir. Otomatik olarak üretilen VHDL kodları Xilinx in ISE aracında denenerek ANNCONT ve ANNSYS'in etkinliği ve doğruluğu ispatlanmıştır.

Anahtar Sözcükler : YSA, Denetleyici, FPGA, Tasarım Otomasyonu.

ABSTRACT

A CONTROLLER DESIGN TOOL DEVELOPMENT FOR AUTOMATICALLY MAPPING NEURAL NETWORKS ONTO FPGAs

Günay TEMÜR
Düzce University
Graduate School of Natural and Applied Sciences, Departmant of Electrical Education
Master of Science Thesis
Supervisor: Assoc. Prof. Dr. İbrahim ŞAHİN
February 2013, 92 pages

Artificial Neural Networks (ANNs) are small copies of human brain and are used in several different areas efficiently. Hardware implementations of the ANNs are preferred when software implementations do not provide desired performance. ANNs can show their real performance on parallel hardware architectures and using FPGAs (Field Programable Gate Array) is a suitable implementation choice for parallel applications. As a result, FPGAs are good candidate for implementing ANNs hardware. On the other hand, implementing ANNs on FPGAs is time consuming process and requires expert personal. Moreover, debugging is required for the error happens during the implementations process. In this study, ANNCONT (Artificial Neural Network Controller), a controller design tool, and ANNSYS (Artificial Neural Network System), a top level ANN system design tool were developed for automatically mapping ANNs to FPGAs. The purpose of the study was to eliminate the debugging state of the design and implementation process and to eliminate the need for the expert personal and to shorten design and implementatıon time of ANNs on FPGAs by automating the whole mapping process. A couple of test cases were developed for testing ANNCONT and ANNSYS design tools. The tools were applied to the test cases and VHDL (Very High Speed Integrated Circuit HDL (Hardware Description Language)) codes were produced in seconds. The effectiveness and correctness of the tools were proved by synthesizing the automatically produced VHDL codes using through Xilinx's ISE tool.

Keywords **:** ANN, Controller, FPGA, Design Automation.

EXTENDED ABSTRACT

A CONTROLLER DESIGN TOOL DEVELOPMENT FOR AUTOMATICALLY MAPPING NEURAL NETWORKS ONTO FPGAs

Günay TEMÜR
Düzce University
Graduate School of Natural and Applied Sciences, Departmant of Electrical Education
Master of Science Thesis
Supervisor: Assoc. Prof. Dr. İbrahim ŞAHİN
February 2013, 92 pages

1. INTRODUCTION:

Artificial Neural Networks (ANNs) require huge amount of CPU power when implemented as software. Software implementation do not provide desired performance especially when the ANN outputs have to be calculate instantaneously in real time applications. As a solution to this problem, ANNs are implemented using Field Programmable Gate Array (FPGA) devices. Although, FPGA implementation of ANNs provides desired performance, mapping and ANN to an FPGA device is a time consuming process and requires expert personal. The purpose of this study is to eliminate the expert personal need for mapping ANNs to FPGAs and speed up the mapping process. With this idea in mind, in this study, an ANN CONTroller Design Tool (ANNCONT) for automatically designing ANN controllers was developed. Moreover, an ANN SYStem Design Tool (ANNSYS) was developed for automatically designing ANN systems by combining the ANN datapaths and controllers, formed with ANNGEN and ANNCONT, respectively, with a suitable address unit.

2. MATERIAL AND METHODS:

ANNs are parallel distributed data processing structures inspired from human brain cells. They are formed using artificial neural cells which were connected to each other through weights. and each having its own memory for storing weight values of the connections. ANNs are utilized for solving complex optimization and decision making problems. FPGAs are widely used configurable circuit components. They have been

utilized in development of hardware implementation of several algorithms and applications. One of the area in which FPGA devices are utilized is implementation ANN applications. In this study, ANNCONT was developed to help automatically design and implement ANNs systems on FPGA devices. ANNCONT was developed as the next step of ANNGEN, another design tool developed by Saritekin for automatically forming ANNs datapaths. The purpose of ANNCONT is to automatically form a controller circuitry for controlling the datapath formed by ANNGEN. ANNCONT uses Moore Machine model while forming ANN controllers. Through this study, another design tool, ANNSYS, was also developed to form ANN system by automatically designed datapath and controller with an suitable address unit. ANNGEN, ANNCONT and ANNSYS accept a NetList file, a template file and a library file as inputs and they produce VHDL code for desired ANN structure. All three tools were implemented in pure C++ language.

3. RESULT AND DISCUSSIONS:

Two separate test cases were constructed to test ANNCONT and ANNSYS. ANNCONT and ANNSYS were run on the test cases successfully and they produced VHDL code for the desired ANNs in seconds. The produced VHDL code was synthesized using Xilinx's ISE tool to demonstrate the effectiveness of the tools developed in this study.

4. CONCLUSION AND OUTLOOK:

In this study, an controller design tool, ANNCONT, and a ANN system design tools, ANNSYS wee developed as the part of the automatic ANN system design on FPGA chips effort. ANNCONT and ANNSYS were successfully tested on two separate test cases and their effectiveness were demonstrated. With the tools desired ANNs structures were formed in seconds without requiring any experts and with any design bugs. The tools developed through the course of this research work were only able make design for feed forward ANNs. In the future, some other networks topologies can be considered.

1. GİRİŞ

Yapay sinir ağları (YSA'lar) (Artificial Neural Network-ANN) insan beyninin sinir sistemini ve çalışma prensibini temel alan elektriksel ve matematiksel modellerdir. Başka bir bakış açısıyla insan beyninin ufak bir kopyası gibidirler. YSA'lar, öğrenme yoluyla yeni bilgiler üretebilme, keşfedebilme, gözlemleyebilme yeteneklerini, yardım almadan yapabilen sistemleri geliştirmede yapı taşı olarak kullanılmak üzere tasarlanmışlardır. Çünkü bu özellikleri, bilinen programlama algoritmaları ile oluşturabilmek imkânsız gibi görünmektedir ([2], 2012).

YSA'lar, birbirleriyle uyum içinde çalışan yoğun şekilde bağlanmış bilgi işleme merkezlerinden, yani sinir hücrelerinden (nöronlardan) oluşmaktadırlar. İşlem birimleri aslında bir transfer fonksiyonunun denklemi gibidirler. Bilgiyi alır, transfer fonksiyonunu uygulayarak işleme sokar ve bir çıktı oluştururlar. İnsan beynindeki sinir hücrelerinde de bu görevi sırasıyla dentrit, hücre gövdesi ve aksonlar üstlenir. Bir YSA'nın bilgiyi nasıl işleyeceği içerdiği transfer fonksiyonuna, diğer ağlarla bağlanış şekline ve kendi sinaptik ağırlıklarına bağlıdır.

Her hangi bir işlevi yerine getirmek üzere tasarlanan YSA'lar, insanlar gibi benzer davranışlar sayesinde öğrenirler. Nasıl ki insanın öğrenmesi sinaptik bağlardaki (2 sinir hücresi arasındaki bağ) bazı elektriksel ayarlamalar sayesinde oluyorsa, aynı şekilde, yapay sinir ağları da tekrarlanan girdiler sayesinde nöronlar arasındaki bağlantıların ağırlık değerlerini değiştirerek öğrenirler. YSA'lar tıpkı canlıların sinir sistemi gibi uyum sağlayabilen bir yapıya sahiptirler. Karar verme aşamasında bağlantı ağırlıkları da devreye girer. Ağı oluşturan hücreler içerisindeki işlem birimleri her ne kadar tek başlarına çalışıyor gibi gözükse de, aslında birçok yapay sinir ağındaki hücreler aynı anda dağınık ve paralel bir şekilde çalışırlar (distributed and parallel computing).

YSA'lar genel amaçlı işlemcilerin işleyişinden farklı şekilde çalışırlar. Bir bilgisayarın işlemcisi (CPU-Central Processing Unit-(Merkezi İşlem Birimi)) belirli bir algoritma çerçevesinde kendisine verilen görevi adım adım-lineer bir şekilde yaparken, bir yapay sinir ağı, büyük bir problemin sadece kendine ait olan küçük bir parçasını doğrusal olmayan bir şekilde işler ve bir sonuç elde eder. Bu sonucu, bir sonraki ağa iletir. YSA

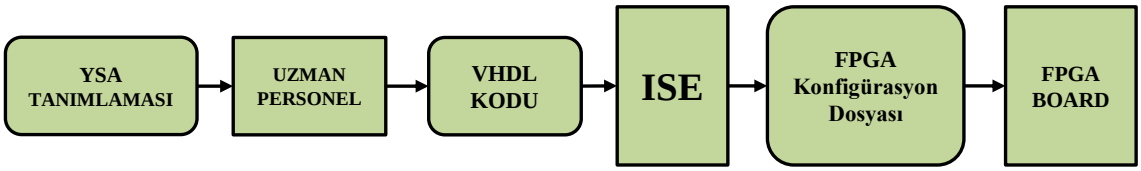
girdiyi işlerken kullandığı denklem, birçok denklemin bir araya gelmesinden oluşabilir. Kullanılan denklemler, genelde optimizasyonda ve grafiksel modellerde gözlenen tarzdadır. Bazı fonksiyonların göreceli olarak aldığı girdiye bağlı olmaması da bir miktar paralel hesaplama sağlamaktadır.

YSA'lar çok değişik alanlarda etkili bir şekilde kullanılmaktadırlar. Örneğin elektrik motorlarının kontrol edilmesinde, sinyal ve görüntü işlemede, değişik tahmin görevlerinde (Hava Tahmini gibi), sınıflandırmada kullanılmaktadırlar (Şahin ve Koyuncu, 2012).

YSA'lar yazılımsal olarak gerçeklenebilmektedirler. Fakat gerçeklenen YSA'nın yapısına, ağda kullanılan hücre adedine ve hücrelerde kullanılan transfer fonksiyonuna göre değişmekle birlikte, yüksek CPU gücü gerektirirler. Özellikle gerçek zamanlı uygulamalarda, anında YSA çıkışının hesaplanabilmesi gerekmektedir. Bu gibi durumlarda, yazılım olarak gerçeklenen YSA'lar yeterli performansı gösterememektedirler. Bu probleme çözüm olarak birçok alternatif yöntem geliştirilmiştir. Geliştirilen alternatiflerden bir tanesi de YSA'ların Sahada Programlanabilir Kapı Dizileri (FPGA) kullanılarak gerçeklenmesidir. Sahada Programlanabilir Kapı Dizileri (FPGA), programlanabilen lojik (mantık) devre blokları ve ayarlanabilir ara bağlantıları içeren sayısal tümleşik devrelerdir.

Burada en önemli adım, YSA'ların bir sayısal devre olarak tasarlanması ve uygun bir donanım tanımlama dili [Hardware Description Language (HDL)]'nde kodlanmalarıdır. YSA'ların gerçeklenmesinde, FPGA'lerin kullanılması ile istenen performansın yakalanmasına karşın, bu işlem uzun zaman alan ve uzman personel tarafından yürütülmesi gereken bir süreçtir. Bir YSA tasarımının FPGA'de gerçeklenmesi için izlenecek adımlar Şekil 1.1'de görülmektedir.

YSA'ların temel işlem elemanı olan hücrenin yapısı ve modeli, girişler ve çıkışlar yönünden düşünüldüğünde standarttır. Yani bir YSA hücresi, *n* adet giriş bilgisini aynı anda alır, bunları ağırlıklandırarak toplar ve bir transfer fonksiyonundan geçirerek çıkış üretir. İçyapıda değişik transfer fonksiyonları kullanılabilmektedir. YSA hücrelerinin bu standart giriş çıkış özelliği sayesinde, önceden tasarlanmış hücreler kullanılarak istenen YSA'nın otomatik olarak bir yazılım aracı sayesinde uzman gereksinimi olmadan ve çok hızlı bir şekilde gerçeklenmesi mümkündür.



Şekil 1.1. Klasik Tasarım Akışı.

1.1. ÇALIŞMANIN AMACI

Kontrol edilecek YSA'nın karmaşıklığına ve kontrol algoritmasına bağlı olarak değişmekle birlikte, bir YSA'yı kontrol edecek denetleyici için yaklaşık o YSA'nın tanımında yazılan kod kadar kod üretilmesi gerekmektedir. Yukarıda belirtildiği gibi bu kodun üretilmesi zaman alan ve uzman gerektiren bir işlemdir. Ayrıca elle yapılan kodlama işlemi hataya çok açıktır ve hata ayıklama işlemi oldukça zordur.

Bu çalışmanın genel amacı, *YSA'ların FPGAçipine uygulanması sürecinde var olan bir YSA veri yolunu denetleyecek, onun uygun bir şekilde hafızadan işlenecek verileri okumasını, veriyi işlemesini ve ürettiği sonucu tekrar hafızaya yazılmasını sağlayacak bir denetleyiciyi otomatik olarak tasarlayacak bir tasarım otomasyon aracı geliştirmektir*. Bu sayede;

Uzman Gereksinimini Azaltmak: YSA'ların FPGA de gerçeklenebilmesi için hem donanım, hem yazılım hem de YSA bilgisine sahip uzman ya da uzmanlara ihtiyaç vardır. Geliştirilen bu araç sayesinde, uzman gereksinimi en aza indirgenerek YSA'lar en verimli bir şekilde FPGA'lere uygulanabilmektedirler.

Uygulama Sürecini Kısaltmak: İnsan eliyle tasarlanan ve gerçeklenen bir YSA denetleyicisinin tasarımı günler, bazen haftalar sürmekte ve bir hata oluştuğunda tasarım sürecinin başına dönülmektedir. Oysa bu işlem geliştirilen tasarım aracı sayesinde saniyeler içinde gerçekleştirebilmektedir.

Debug Aşamasını Ortadan Kaldırmak: Geliştirilen sistem sayesinde kodlamada hata yapma olasılığı çok düşük olduğundan debug aşaması ortadan kaldırılmıştır.

Bu amaçlar doğrultusunda, bu çalışmada Yapay Sinir Ağları'nın otomatik olarak FPGA çipine uygulanmasını gerçekleştiren ve sistem oluşumunu otomatikleştirmeyi sağlayan yardımcı bir tasarım aracı olan ANNCONT (Artificial Neural Network CONTroller) ve

var olan veri yolu ile denetleyiciyi bir araya getirerek YSA sistemini oluşturan ANNSYS (Artificial Neural Network SYStem) tasarım araçları geliştirilmiştir.

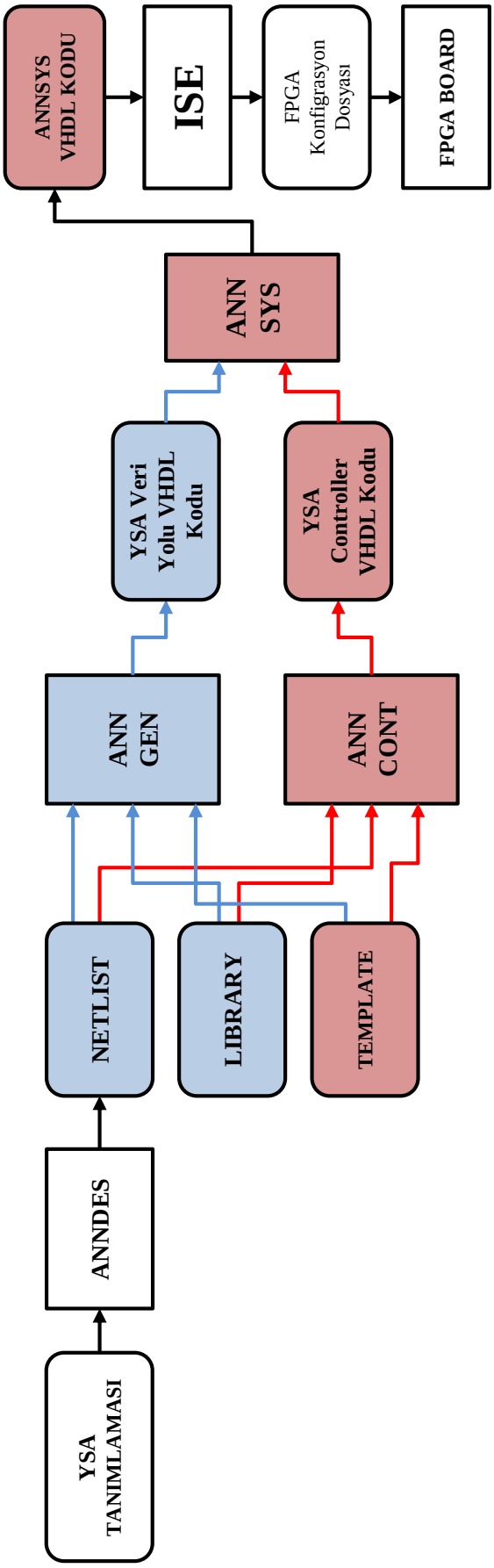
1.2. ANNCONT'A GENEL BAKIŞ

Şekil 1.2'de ANNCONT'un ANNGEN ile entegre bir şekilde çalışma yapısı görülmektedir. ANNCONT girdi olarak; metin tabanlı yalın text formatında yazılmış YSA tanımlamasını (*NetList*), hücre kütüphanesini (*Library*), ve VHDL kod şablon (*Template*) dosyalarını kullanır. *NetList* dosyası; FPGA çipine aktarılacak olan YSA yapısını, kullanılan hücrelerin türlerini ve bağlantılarını, hücre içindeki ağırlık değerlerini tanımlayan metin tabanlı bir dosyadır. Kütüphane (*Library*) dosyası; tanımlamada kullanılacak nöronların listesini, Şablon (*Template*) dosyası ise; VHDL [Very High Speed Integrated Circuit HDL (Çok Hızlı Entegre Devre Donanım Tanımlama Dili)] kodu yazımında gerekli diğer bazı şablonları içermektedir. Sonuç olarak, verilen *NetList*'e uygun YSA'ları FPGA çipine otomatik olarak uygulayacak denetleyici tasarım aracına ait kod üretilir. ANNCONT daha önceden geliştirilen ANNGEN (Sarıtekin, 2011) tasarım aracının bir adım ilerisi olarak tasarlanmıştır. ANNCONT ANNGEN tarafından üretilen YSA yapısını kontrol edecek bir sayısal kontrol devresini otomatik olarak tasarlayan bir otomatik tasarım aracıdır. Otomatik YSA oluşturulmasında ANNCONT'tan bir sonra gelen adım ise ANNSYS'tir.

1.3. ANNSYS

ANNSYS, ANNGEN tarafından oluşturulan veri yolu ile ANNCONT tarafından oluşturulan denetleyici ünitesini uygun bir adres ünitesi ile yeni bir blok içinde birleştirerek en üst seviye YSA sistemini oluşturan otomatik tasarım aracıdır.

ANNCONT ve ANNSYS C++ programlama dili ile yazılmışlardır. Bu araçlar için toplam altıyüz yetmiş altı satırlık yirmi sekiz fonksiyondan oluşan C++ kodu yazılmıştır.



Şekil 1.2. Hedeflenen Tasarım.

2. MATERYAL VE YÖNTEM

2.1. YAPAY SİNİR AĞLARI

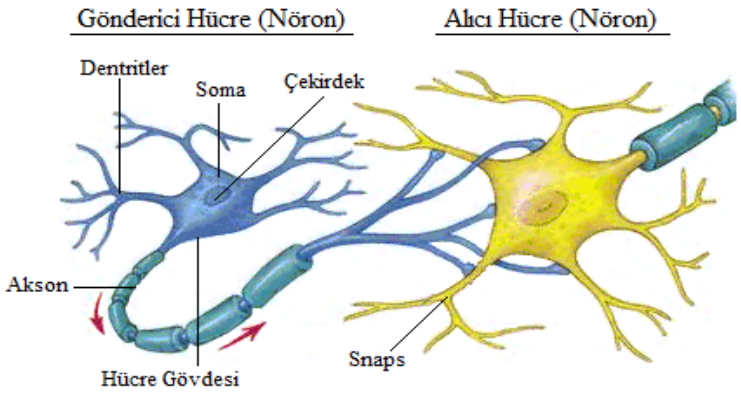
2.1.1. Biyolojik Sinir Sistemi

Biyolojik sinir sistemi; verinin alınması, yorumlanması ve karar üretilmesi gibi işlevlerin yürütüldüğü üç katmanlı bir yapıdan oluşmaktadır. Şekil 2.1'de blok diyagramı verilen sinir sisteminde görüldüğü üzere uyarılar, alıcı sinirler ile birlikte elektriksel sinyallere dönüştürülerek beyne iletilir. Beynin oluşturduğu çıktılar ise, tepki sinirleri tarafından belirli tepkilere dönüştürülür. Şekil 2.2'de verilen insan sinir hücresi ise hücre gövdesi, gövdeye giren alıcı lifler (dentrit) ve gövdeden çıkan sinyal iletici lifler (akson) olmak üzere üç temel bileşenden meydana gelir. Dentritler aracılığı ile bilgiler diğer hücrelerden hücre gövdesine iletilir. Hücrelerde oluşan çıktılar ise akson yardımı ile bir diğer hücreye aktarılır. Aktarımı gerçekleştiren aksonlar ince yollara ayrılabilmekte ve diğer hücrenin dentritlerini oluşturmaktadırlar. Akson-dentrit bağıntısının oluştuğu bu noktalara sinaps adı verilir. Sinapsa ulaşan ve dentritler tarafından alınan bilgiler genellikle elektriksel darbelerdir, fakat bu bilgiler sinapstaki kimyasal ileticilerden etkilenirler. Hücrenin tepki oluşturması için bu tepkilerin belirli bir sürede belirli seviyeye ulaşması gerekmektedir. Bu değer eşik değeri olarak adlandırılır.

Şekil 2.1. Biyolojik Sinir Sistemi Blok Diyagramı.

Tipik bir hücre, hücre gövdesi ve dentritleri aracılığıyla dış kaynaklardan gelen elektrik darbelerinden üç şekilde etkilenir. Gelen darbelerden bazıları hücreyi uyarır, bazıları bastırır, bazıları da davranışında değişikliğe yol açar. Hücre yeterince uyarıldığında çıkış kablosundan (aksonundan) aşağı bir elektriksel işaret göndererek tepkisini gösterir. Genellikle tek bir akson üzerinde çok sayıda dallar bulunur. Aksondan inmekte olan

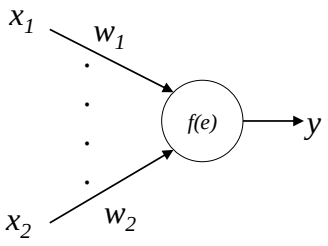
elektrik işareti dallara, alt dallara ve sonunda başka hücrelere ulaşarak onların davranışını etkiler. Hücre, çok sayıda başka hücrelerden elektrik darbesi biçiminde gelen verileri alır. Yaptığı iş bu girdilerin karmaşık ve dinamik bir toplamını yapmak ve bu bilgiyi aksonundan aşağı göndererek bir dizi elektrik darbesi biçiminde çok sayıda başka hücreye iletmektir. Hücre, bu etkinlikleri sürdürmek ve molekül sentezlemek için de enerji kullanır fakat başlıca işlevi işaret alıp, işareti işleyip göndermek, yani bilgi alışverişidir ([3], 2012).



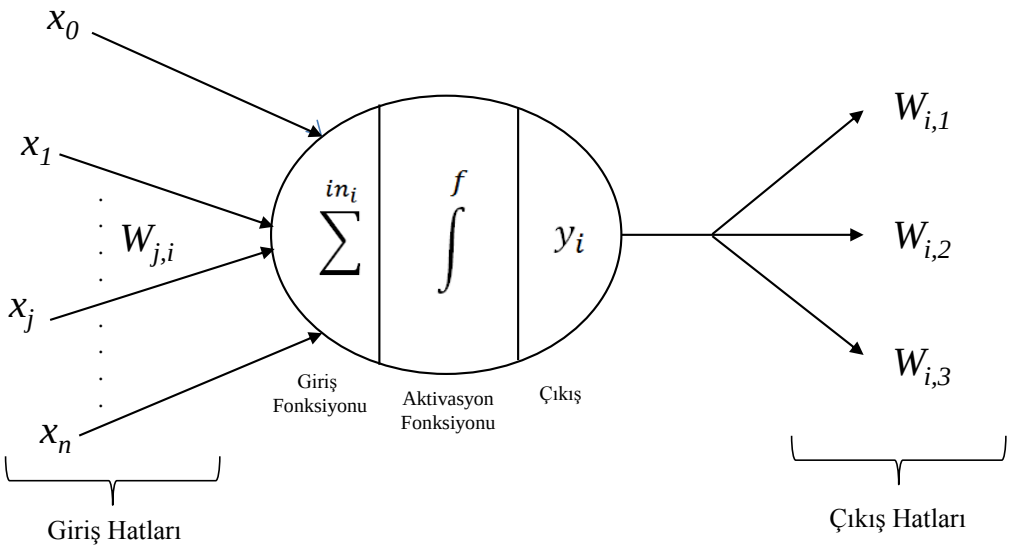
Şekil 2.2. İnsan Sinir Hücresi Yapısı ([4], 2012).

2.1.2. Yapay Sinir Hücresi

İnsan sinir hücresi temel alınarak oluşturulan matematiksel modellere yapay sinir hücresi denir. Hücre yapısında girişler önce belirli bir ağırlık değerleri ile çarpılarak toplanır. Toplanan bu değer belli bir 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 (Yılmaz, 2008). Transfer fonksiyonu olarak sigmoid fonksiyon kullanılmış bir yapay sinir hücresinin yapısı Şekil 2.3'te ve detaylı gösterimi Şekil 2.4'te gösterilmiştir.



Şekil 2.3. Yapay Sinir Hücresi Basit Yapısı ([5], 2012).



Şekil 2.4. Yapay Sinir Hücresi Yapısı ([5], 2012).

Her bir giriş (*x*), ağırlık (*w*) ile çarpılarak toplanır ve sonucu oluşturmak için bir transfer fonksiyonu (aktivasyon fonksiyonu) ile işlem yapılarak eşitlik 2.1'de verildiği gibi hücre çıkışı (*y*) üretilir. Hücre çıkışı $y = f(\sum w * x)$ seklinde hesaplanır ve bu çıkış (*y*), seçilen transfer fonksiyonuna bağlıdır.

$$y_i = f(\sum_{n}^{j=0} w_{i,j} * x_j) \tag{2.1}$$

2.1.3. Yapay Sinir Hücresinin Temel Elemanları

İşlemci olarak adlandırılan yapay hücreler, yapay sinir ağlarının temel birimleridir. Bir yapay hücre biyolojik yapıdaki bir hücre ile kıyaslandığında daha basit bir yapıya sahip olmasına rağmen, genel olarak biyolojik hücrelerin bazı temel işlevlerini taklit etmektedirler. Bu işlevler 5 başlık altında toplanabilir.

2.1.3.1. Girişler

Girişler (x_1,x_2,x_3.....,x_n) dış dünyadan veya başka bir sinir hücresinden gelen bilgilerin ara katmana gönderilmesi işlemini gerçekleştirirler. Bu gönderim sırasında herhangi bir bilgi işleme gerçekleşmez. Gelen bilgiler aynen sonraki katmana iletilirler (Güneren, 2010).

2.1.3.2. Ağırlıklar

Ağırlıklar (*w1,w2,w3,.....,wn*), yapay sinir ağı tarafından alınan girişlerin sinir hücresi üzerindeki etkisini belirleyen katsayılardır. Bir sinir ağında her bir hücrenin her bir girişi için bir ağırlık vardır. Bir ağırlığın değerinin büyük olması, o girişin sinir hücresine güçlü bağlanması ya da önemli olması, küçük olması zayıf bağlanması ya da önemli olmaması anlamına gelmektedir (Elmas, 2007).

2.1.3.3. Toplama İşlevi

Toplama işlevi, sinir ağındaki bir hücrenin her bir giriş (x) ile o girişe ait ağırlık (w) değerlerinin çarpımlarının toplamının eşitlik 2.2'de olduğu gibi ifade edilmesidir.

$$Toplam = \sum_{i}^{n} x_i * w_i \tag{2.2}$$

Bununla beraber çoğu uygulamada eşik değeri olan bir θ değeri toplama dahil edilmektedir (Eşitlik 2.3).

$$Toplam = \sum_{i}^{n} x_i * w_i + \theta \tag{2.3}$$

θ eşik değeri girişlerden tamamen bağımsız bir değerdir ve bütün girişlerin sıfır olması durumunda dahi hücre çıkışının sıfır olma durumunu ortadan kaldırmaktadır.

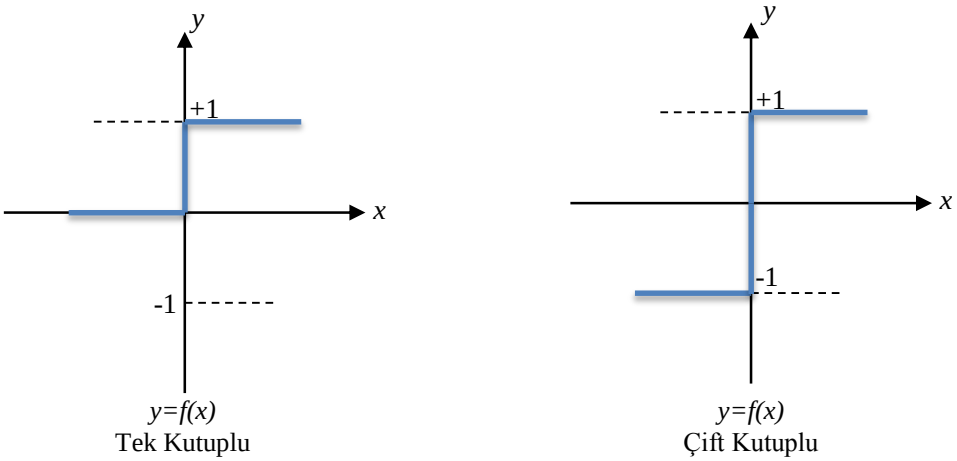
2.1.3.4. Transfer Fonksiyonları

Transfer fonksiyonları, toplama işlevi ile belirlenen toplam hücre giriş değerini işleyerek hücre çıkışını belirleyen fonksiyonlardır. Hücre modellerinde, hücrenin gerçekleştireceği işleve göre çeşitli türlerde transfer fonksiyonları kullanılabilir. Bir ağdaki bütün hücrelerin transfer fonksiyonları aynı olabileceği gibi birbirinden farklı da olabilirler. Yaygın olarak kullanılan bazı transfer fonksiyonları aşağıdaki gibi sıralanabilir.

Simetrik eşik transfer fonksiyonu: McCulloch-Pitts modeli olarak bilinen eşik transfer fonksiyonlu hücreler, mantıksal çıkış verir ve sınıflandırıcı ağlarda tercih edilirler. Perseptron olarak da bilinen eşik fonksiyonlu hücrelerin matematiksel

modelleri eşitlik 2.4'teki denklemlerde olduğu gibi tanımlanırlar. Tek kutuplu ve çift kutuplu olmak üzere iki çeşidi vardır (Şekil 2.5) (Dere, 2009).

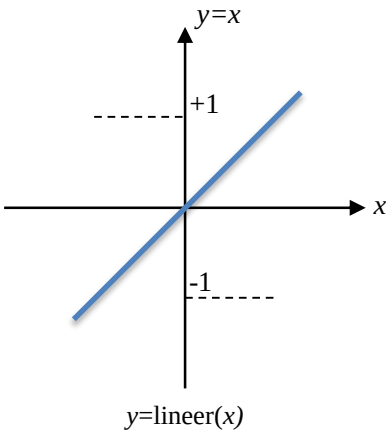
$$y = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \qquad y = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases} \tag{2.4}$$



Şekil 2.5. Simetrik Eşik Transfer Fonksiyonları.

Lineer Transfer Fonksiyonu: Bu transfer fonksiyonunda toplayıcıdan gelen veriler bir α katsayısı ile çarpılır. α=1 ise fonksiyon çıkışı girişine eşittir (Eşitlik 2.5, Şekil 2.6). Adaline olarak ta bilinen bu model, sıklıkla klasik işaret işleme ve regresyon analizlerinde kullanılır.

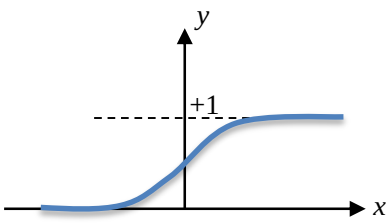
$$y=F(x)=\alpha*x,\ \alpha=1 \tag{2.5}$$



Şekil 2.6. Lineer Transfer Fonksiyonu.

Sigmoid Transfer Fonksiyonu: Şekil 2.7'de grafiksel olarak ve eşitlik 2.6'da formülsel olarak verilen transfer fonksiyonu YSA'larda en çok kullanılan, doğrusal ve doğrusal olmayan davranışlar arasında denge sağlayan sürekli artan bir fonksiyon olarak tanımlanır (Demuth ve diğ.,2011).

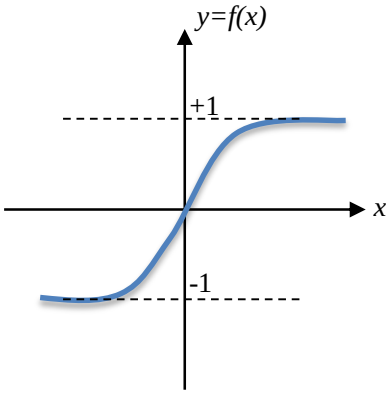
$$y = \frac{1}{1 + e^{-x}} \tag{2.6}$$



Şekil 2.7. Sigmoid Transfer Fonksiyonu.

Hiperbolik Tanjant Transfer Fonksiyonu: Eşitlik 2.7'de formülize edilmiş olarak verilen hiperbolik tanjant transfer fonksiyonu, türevi alınabilir, sürekli ve doğrusal olmayan bir fonksiyon olması nedeniyle doğrusal olmayan problemlerin çözümünde kullanılan bir transfer fonksiyonudur (Şekil 2.8). Bu transfer fonksiyonunda giriş olarak -∞ ile +∞ arasında herhangi bir değer alır ve çıkış değeri *-1* ile *+1* arasındadır. Literatürde çift kutuplu fonksiyon olarak da adlandırılır (Dere, 2009).

$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.7}$$



Şekil 2.8. Hiperbolik Tanjant Transfer Fonksiyonu.

2.1.3.5. Çıkış İşlevi

Çıkış *y=f(x)*, transfer fonksiyonunun diğer sinir hücrelerine gönderilmesi veya elde edilmek istenilen işlem çıkışı ya da sonuç olarak tanımlanır. Ayrıca bir hücrenin çıkışı, kendisine ve kendisinden sonra gelen bir veya daha fazla hücreye giriş olabilir.

2.1.4. Yapay Sinir Ağlarının Yapısı

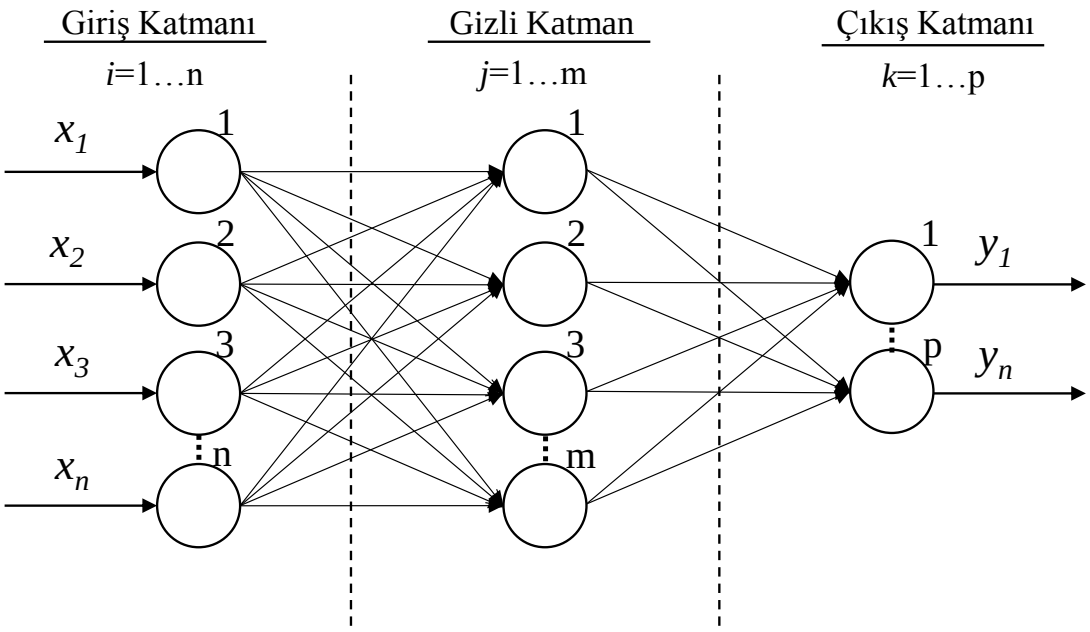
Yapay Sinir Ağları (YSA) insan beyninden esinlenerek geliştirilmiş, ağırlıklı bağlantılar aracılığıyla birbirine bağlanan ve her biri kendi belleğine sahip işlem elemanlarından oluşan paralel ve dağıtılmış bilgi işleme yapılarıdır (Elmas, 2007). YSA'lar, başka bir deyişle biyolojik beyni oluşturan sinir hücrelerini (nöron) matematiksel olarak taklit ederek akıllı bir sistem oluşturmaya çalışan ve bu sinir sisteminin çalışmasını elektronik ortama taşımayı hedefleyen bir bilgisayar programlama yaklaşımıdır.

YSA'lar öğrenme sürecinde bir programcılık yeteneği gerektirmeyen, kendi kendine öğrenebilen düzeneklerdir. Bu ağlar öğrenmenin yanı sıra, ezberleme ve bilgiler arasında ilişkiler oluşturma yeteneğine de sahiptirler (Elmas, 2007).

YSA'lar işlemleri son derece hızlı bir şekilde yapabilmelerine rağmen insan beyni ile yarışabilecek düzeyden oldukça uzaktırlar. Fakat karmaşık eşleştirmelerde ve veri sınıflandırılmasında başarılı sonuçlar vermektedirler (Yılmaz, 2008).

YSA'larda bulunan her düğüm, *n*. dereceden tercihen lineer olmayan bir birimdir. Düğümler arasında bağlantılar bulunmakla birlikte her bağlantı tek yönlü iletim yoludur. Bir düğüm birden fazla düğüme veri aktarabilir. İşlenen bilgiler (sonuç belirleme süreci) bir sonraki katmandaki bir veya birden fazla düğüme iletilir (Yılmaz, 2008).

Sadece giriş ve çıkış katmanlarında oluşan ağlar, karmaşık işlemleri hesaplama yeteneklerine sahip değildirler. Bu sebeple karmaşık hesaplamalar için en az bir ara (gizli) katman olmalıdır. Şekil 2.9'da gizli katmana sahip 3 katmanlı bir yapay sinir ağı görülmektedir (Elmas, 2007).



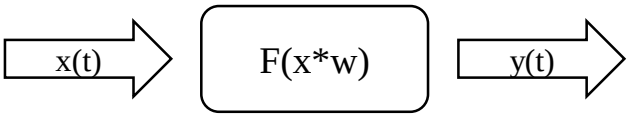
Şekil 2.9. Çok Katmanlı Yapay Sinir Ağı.

Bu tür ağlar giriş katmanı, gizli katman ve çıkış katmanından oluşmaktadırlar. Bir katmandaki her bir sinir hücresinin çıkışı bir sonraki katmanın bütün sinir hücreleri ile bağlantılıdır. Aynı katmandaki sinirler arasında veya geri-besleme şeklinde bağlantıları yoktur.

2.1.5. Yapay Sinir Ağlarının Sınıflandırılması

2.1.5.1. İleri Beslemeli Yapay Sinir Ağları

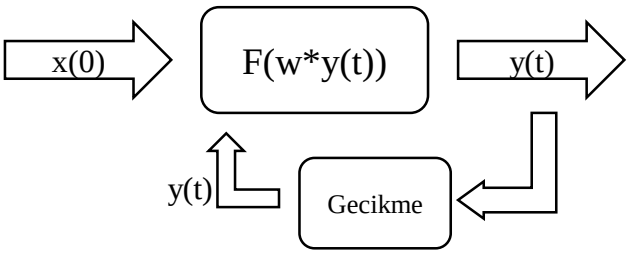
İleri beslemeli YSA'lar da, hücreler katmanlar şeklinde düzenlenir ve bir katmandaki hücrelerin çıkışları bir sonraki katmana ağırlıklar üzerinden giriş olarak verilir. En genel halde giriş katmanı, saklı katman ve çıkış katmanı olmak üzere üç katmanlı bir yapıya sahiptirler. Giriş katmanı, dış ortamlardan aldığı bilgileri hiçbir değişikliğe uğratmadan orta (gizli) katmandaki hücrelere iletir. Bilgi, orta ve çıkış katmanında işlenerek ağ çıkışı belirlenir. En uç katmanın çıktı değerleri de ağın çıktısını belirler. Şekil 2.10'da giriş, orta ve çıkış katmanı olmak üzere üç katmanlı ileri beslemeli YSA'ların basit yapısı verilmiştir (Subaşı, 2010).



Şekil 2.10. İleri Beslemeli Sinir Ağlarının Basit Yapısı.

2.1.5.2. Geri Beslemeli Yapay Sinir Ağları

Geri beslemeli YSA'larda en az bir hücrenin çıkışı kendisine ya da diğer hücrelere giriş olarak verilir ve genellikle geri besleme bir geciktirme elemanı üzerinden yapılır. Geri besleme, bir katmandaki hücreler arasında olduğu gibi katmanlar arasındaki hücreler arasında da olabilir. Bu yapısı ile geri beslemeli YSA'lar, doğrusal olmayan dinamik bir davranış gösterirler. Dolayısıyla, geri beslemenin yapılış şekline göre farklı yapıda ve davranışta geri beslemeli YSA yapıları elde edilebilir. Şekil 2.11'de çıkışlarından giriş katmanına geri beslemeli YSA'ların basit yapısı görülmektedir. (Subaşı, 2010).



Şekil 2.11. Geri Beslemeli Sinir Ağlarının Basit Yapısı.

2.1.6. Yapay Sinir Ağlarının Özellikleri

YSA'ların hesaplama ve bilgi işleme gücünü, paralel dağıtık yapısından, öğrenebilme ve genelleme yeteneğinden aldığı söylenilebilir. YSA'ların genelleme özelliği, öğrenme sürecinde karşılaşılmayan girişler için de uygun tepkileri üretmesi olarak tanımlanır. Bu tür üstün özellikleri, karmaşık problemleri çözebilme yeteneğini gösterir. Ayrıca YSA'lar günümüzde birçok bilim alanında aşağıdaki özellikleri sebebi ile etkin olmuş ve uygulama alanı bulmuştur (Çavuşlu, 2006), (Subaşı, 2010). Bu özellikleri başlıklar halinde sıralayacak olursak.

2.1.6.1. Doğrusal Olmama

YSA'ların yapıları gereği doğrusal ağlar olduğu gibi, daha çok doğrusal olmayan yönleriyle öne çıkmışlardır. Ağın ya da temel işlem elemanı olan hücrenin, doğrusallığı transfer fonksiyonu ile belirlenir. Bu özelliği ile YSA'lar, özelikle doğrusal olmayan karmaşık problemlerin çözümünde en önemli araç durumuna gelmişlerdir (Haykın, 1994).

2.1.6.2. Öğrenme

YSA'ların arzu edilen davranışı gösterebilmesi için amaca uygun olarak tasarlanması gerekir. Bu durum, hücreler arasında doğru bağlantıların yapılması ve bağlantıların uygun ağırlıklara sahip olmasını gerektirir. YSA'ların karmaşık ve lineer olmayan yapısı nedeniyle bağlantılar ve ağırlıklar önceden ayarlı olarak verilemezler ya da tasarlanamazlar (Haykın, 1999). Genellikle ağırlıklar, rasgele ya da sabit bir değerde seçilirler. YSA'lar, istenen davranışı gösterecek şekilde ilgilendiği problemden aldığı eğitim örneklerini kullanarak problemi öğrenmelidirler. Belli bir hata kriterine ve öğrenme algoritmasına göre, ağırlıkların yenilenerek, artık değişmediği durumda öğrenmenin gerçekleştiği söylenebilir (Çavuşlu, 2006).

2.1.6.3. Uyarlanabilirlik

YSA'lar, ilgilendiği problemdeki değişikliklere göre ağırlıklarını ayarlarlar. Yani, belirli bir problemi çözmek amacıyla eğitilen bir YSA, problemdeki değişimlere göre tekrar eğitilebilir, değişimler devamlı ise gerçek zamanda da eğitime devam edilebilir. Bu özelliği ile YSA'lar, sinyal işleme, uyarlamalı sistem tanıma ve kontrol gibi alanlarda etkin olarak kullanılırlar (Haykın, 1994). Uyarlanabilir sistemler çok kısa zaman aralıklarında ağırlıklarını birçok değişikliğe uğratılırlar, bu sebeple bozucu etkilere cevap vermeye yatkındırlar. Bu durum sistem performansını büyük ölçüde etkiler. Uyumluluk özelliklerini tam anlamıyla kullanmak için; sistemin alacağı ilk ağırlık değerleri, çevreden gelen istenilen değişimlere cevap verecek kadar küçük ve sistemin bozucu etkilerden etkilenmemesini sağlayacak kadar büyük olmalıdır (Çavuşlu, 2006).

2.1.6.4. Genelleme

Ağ yapısının, eğitim esnasında kullanılan nümerik bilgilerden eşleştirmeyi betimleyen kaba özellikleri çıkarması ve böylelikle eğitim sırasında kullanılmayan girdiler için de anlamlı yanıtlar üretebilmesidir (Öztemel, 2003). YSA'lar, ilgilendiği problemi öğrendikten sonra eğitim sırasında karşılaşmadığı giriş verileri için de arzu edilen tepkiyi üretebilir. Örneğin, karakter tanıma amacıyla eğitilen YSA'lar, bozuk karakter girişlerinde de doğru karakterleri verebilir ya da bir sistemin eğitilmiş YSA modeli, eğitim sürecinde verilmeyen giriş sinyalleri için de sistemle aynı davranışı gösterebilir (Çavuşlu, 2006).

2.1.6.5. Paralellik

YSA'lar birçok hücreden meydana gelir ve bu hücreler eş zamanlı çalışarak karmaşık işlevleri yerine getirirler. Süreç içerisinde bu hücrelerden her hangi biri işlevini yitirse dahi sistem güven sınırları içerisinde çalışmasına devam edebilir. Alışılagelmiş bilgi işlem yöntemlerinin çoğunda işlemler seri bir düzen içerisindedir. Bu düzen özellikle hız sorununa yol açmaktadır. Bilgisayarlar beyne göre çok hızlı çalışmasına rağmen beynin toplam hızı bilgisayara göre çok yüksektir. YSA'larda işlemler doğrusal değildir ve bu bütün ağa yayılmış durumdadır. Aynı katmanlar arasında zaman bağımlılığı da bulunmamaktadır. Bu durum, tüm sistemin eş zamanlı çalışabilmesine olanak vermekte ve hızı çok arttırmaktadır. Bu sayede doğrusal olmayan karmaşık problemlerin de çözümlenmesi mümkündür ([6], 2012).

2.1.6.6. Hata Toleransı

Klasik hesaplama sistemleri çok az bir hatadan bile etkilenirler. YSA'lar için durum farklıdır. Bu farklılık YSA'ların hata toleranslı olmasıdır. İşlem elemanlarının az da olsa zarar görmesi sistemin bütününü etkiler. YSA'lar paralel dağılmış parametreli bir sistem olduğundan her bir işlem elemanı izole edilmiş bir ada olarak düşünülebilir. Daha çok işlem elemanın zarar görmesi ile sistemin davranışı biraz daha değişir. Performans düşer ama sistem hiç bir zaman durma noktasına gelmez. YSA'ların hata toleranslı olmasının nedeni, bilginin tek bir yerde saklanmayıp, sisteme dağıtılmasıdır. Bu özellik sistemin durmasının önemli bir zarara neden olacağı uygulamalarda önem kazanır.

2.1.6.7. Donanım ve Hız

YSA'lar, paralel yapısından dolayı büyük ölçekli entegre devre (VLSI) teknolojisi ile gerçeklenebilirler. Bu özellik, YSA'ların hızlı bilgi işleme yeteneğini artırır ve gerçek zamanlı uygulamalarda kullanılabilmesini mümkün kılar (Çavuşlu, 2006).

2.1.6.8. Analiz ve Tasarım Kolaylığı

YSA'ların temel işlem elemanı olan hücrenin yapısı ve modeli, bütün YSA'ların yapılarında yaklaşık aynıdır (Haykın, 1999). Dolayısıyla, YSA'ların farklı uygulama alanlarındaki yapıları da standart yapılardaki bu hücrelerden oluşur. Bu nedenle, farklı uygulama alanlarında kullanılan farklı mimarilerdeki YSA'lar, benzer öğrenme

algoritmalarını ve teorilerini paylaşabilirler. Bu özellik, problemlerin YSA'lar ile çözümünde önemli bir kolaylık getirmiştir (Çavuşlu, 2006).

2.1.7. Yapay Sinir Ağlarında Öğrenme

YSA'larda, kullanılan mimarinin yanı sıra, ağırlık değerlerinin belirlenmesi işlemi de ağın performansını etkileyen önemli bir unsurdur. Ağın problem ile ilgili doğru sonuçlar veren ağırlık değerlerine ulaşması işlemine ağın öğrenmesi denir. Öğrenme modelleri üç gruba ayrılmaktadırlar. Bunlar danışmanlı, danışmansız ve karma öğrenme modelleri olarak isimlendirilirler (Vural, 2007).

2.1.7.1. Danışmanlı Öğrenme

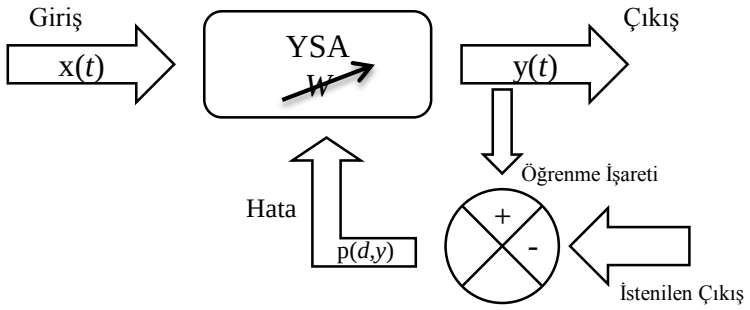
Danışmanlı öğrenme metodu YSA'ların öğrenmesinde kullanılan en yaygın yöntemdir. Bu metot da, YSA'lardan elde edilen gerçek çıktı değeri ile elde edilmesi istenilen çıktı değeri karşılaştırılır. Başlangıçta rastgele atanan ağırlık değerleri, ağ tarafından tekrar düzenlenir, böylelikle bir sonraki döngüde istenilen çıktı ile gerçek çıktı arasında daha yakın değerler üretilir. Bu öğrenme metodu, bütün işlem elementlerinin var olan bütün hatalarını minimize etmeye çalışarak gerçekleşir. Bu şekilde hataların azaltılması işlemi, girdilerin ağırlık değerlerinin sürekli olarak değiştirilerek kabul edilebilir bir ağ performansına ulaşılıncaya kadar devam eder.

Ağın öğrenmesi girdi ve çıktı verilerinin ağa sunulması sonucu gerçekleşir. Bu veriler eğitim seti olarak tanımlanır ve sisteme sunulan her girdiye karşılık istenilen çıktı değerleri de ağa sunulur. Öğrenme çok fazla zaman alabilir ve yetersiz işlem gücüne sahip bir sistemin öğrenmesi de haftalarca sürebilir. YSA'ların öğrenmesi, ağ kullanıcılarının tanımladığı performans seviyesine ulaştığında sonlandırılabilir. Bu seviye, ağın verilen girdi değerlerine karşılık istenilen istatistiksel kesinlikteki çıktı değerlerini üretebildiği nokta olarak tanımlanır. Eğer daha ileri seviyede bir öğrenmeye ihtiyaç yoksa (daha fazla veri girişi) elde edilen ağırlık değerleri uygulamalarda kullanılır. Bazı yapay sinir ağları, çalışma sırasında da düşük hızlarda öğrenmeye devam etmekte, bu da ağın ileri seviyelerdeki değişim koşullarına adapte olmasını sağlamaktadır. Eğer ağdan önemli özellikleri ve ilişkileri öğrenmesi isteniyorsa, girdi ve çıktıların ihtiyaç duyulan bütün bilgileri sağlayacak derecede büyük olması gerekmektedir (Anderson ve McNeill, 1992).

Bir ağın başarılı bir şekilde öğrenmesi için, girdi ve çıktı verilerinin ağa nasıl sunulacağı çok önemlidir. YSA'lar sadece sayısal girdi verileriyle çalışabilmekte, bu nedenle dış dünyadan alınan sembolik verilerin sayısal verilere dönüştürülmesi gerekmektedir. Ayrıca, bu verilerin ölçeklendirilmesi veya ağın algılayabileceği şekilde normalize edilmesi sağlanmalıdır (Anderson ve McNeill, 1992).

Öğrenen bir sinir ağının eğitim verileri üzerinde iyi performans sergilemesinden sonra, daha önce hiç görmediği verilerle ne yapacağını görmek te çok önemlidir. Eğer bir sistem test seti (bir grup girdi ve çıktı) için mantıklı çıktılar veremiyorsa, öğrenme tam olarak gerçekleşmemiştir. Gerçekten de, test aşaması ağın verilen girdileri ezberlemediği, bir uygulama içindeki genel örnekleri öğrendiğini göstermesi açısından önemlidir (Anderson ve McNeill, 1992).

Danışmanlı öğrenme algoritmalarına örnek olarak; Widrow-Hoff tarafından geliştirilen delta kuralı ve Rumelhart-McClelland tarafından geliştirilen genelleştirilmiş delta kuralı algoritmaları verilebilir. Şekil 2.12'de danışmanlı öğrenme yapısına ait bir akış diyagramı verilmiştir.



Şekil 2.12. Danışmanlı Öğrenme Yapısı.

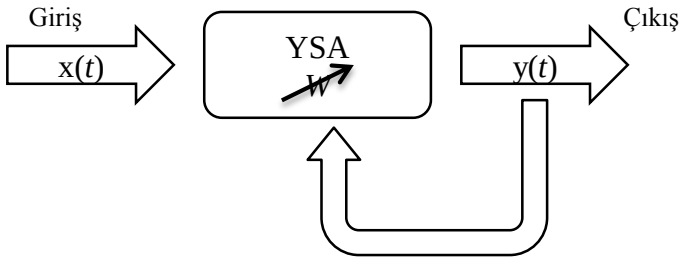
2.1.7.2. Danışmansız Öğrenme

Danışmansız öğrenme gelecekte ümit verici gelişmeler sağlayacak bir metottur. Bu sayede, bilgisayarlar bir gün kendi kendilerine öğrenebileceklerdir. Günümüzde, danışmansız öğrenme metodunun uygulandığı ağların kullanımı çok yaygın olmayıp, sadece temel akademik çalışmalar yürütülmektedir. Bu metot kendi kendine öğrenme olarak da adlandırılabilmektedir.

Ağlar, girdi ağırlıklarını belirlemek için dışarıdan bir etkiye ihtiyaç duymayıp, bunun yerine performanslarını içeriden gözlemektedirler. Bu ağlar girdi sinyallerinde düzen

aramaktadırlar ve ağın fonksiyonuna göre adaptasyon yapmaktadırlar. Ağa bir verinin doğru veya yanlış olup olmadığı belirtilmeden, ağ onu nasıl organize edebileceği hakkında bazı bilgilere sahip olmaktadır. Bu bilgi, ağ topolojisinin ve öğrenme kurallarının içine yerleştirilmiştir.

Günümüzde, Şekil 2.13'de genel akış diyagramı verilen danışmansız öğrenme ile ilgili araştırmalar genellikle hükümetlerin ilgi duyduğu alanlardır. Askeri alanlardaki uygulamalar bunlara en iyi örnekler olabilir. Çünkü ağın öğrenebilmesi için belirli bir veri setine ihtiyaç vardır. Fakat bilindiği gibi olumsuz durumlar her zaman görülmemektedirler ve askeri uygulamalar, herhangi bir çatışma çıkıncaya kadar, ağı eğitmek için gerekli veri setine sahip değildirler. Bu sebeple veri setine ihtiyaç duymayan kendi kendine öğrenebilen ağlar hükümetlerce daha çok tercih edilmektedirler (Anderson ve McNeill,1992).



Şekil 2.13. Danışmansız Öğrenme Yapısı.

2.1.7.3. Karma Öğrenme

Ağın hem danışmanlı ve hem danışmansız öğrenme işlemlerinin birlikte kullanılarak eğitilmesine denir. Radyal tabanlı YSA'lar ve olasılık tabanlı ağlar bunlara örnek olarak verilebilir (Sarıtekin, 2011).

2.2. FPGA

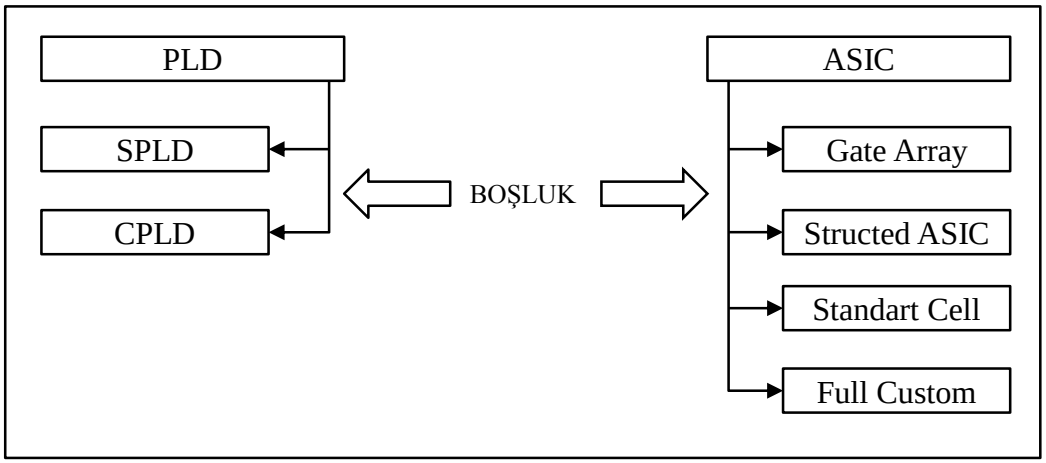
FPGA, Field Programmable Gate Array teriminin kısaltmasıdır. Türkçe karşılığı ise Sahada Programlanabilir Kapı Dizileri'dir. FPGA'lar yaygın olarak kullanılan programlanabilir devre elemanlarıdır. Programlanabilir devre elemanları, geniş uygulama alanları sağlayabilmek için genel amaçlı tümleşik devreler olarak tasarlanmışlardır. Tasarlanan bu sayısal tümleşik devreler yapılandırılabilir mantık blokları ile bu bloklar arasındaki yeniden programlanabilir ara bağlantılardan oluşmaktadırlar. Bu yapılar programlanarak çeşitli görevleri gerçekleştirebilmektedirler.

Sahada programlanabilirlik terimini açıklarken kullandığımız kod yazım dili olan VHDL (VHSIC Hardware Description Language) ile tanımladığımız yapıların bir donanıma denk düştüğünü unutmamamız gerekir. Öyle ki; şematik tasarım vb. tasarım yöntemlerinde çoğunlukla elde edilen donanım üretim bandından çıktıktan sonra test edilebilmektedir. Bu aşamada bulunan bir hata düzeltilmek istendiğinde, donanımın şematik tasarımında ilgili değişiklikler yapıldıktan sonra tekrar üretim aşamasına geçilir. Bu da tasarımcıya önemli bir maliyet getirir. FPGA gibi tümleşik devreler üzerinde tanımladığımız tasarımlar hali hazırda bir donanıma denk düştüğünden, her hangi bir değişiklik gerektiğinde sadece kodumuzu değiştirerek FPGA elemanını tekrar programlamak yeterli olacaktır. İşte bu bize sahada programlanabilirlik terimini açıklamaktadır (Öztürk, 2010).

2.2.1. FPGA'in Doğuşu

Programlanabilir lojik tümleşik devreler ilk olarak PLA (Programmable Logic Array), PAL (Programmable Array Logic), GAL (Generic Array Logic), CPLD (Complex Programmable Logic Device) gibi yapılarla ortaya çıkmışlardır. Bunlardan PLA ve PAL'ların içerdiği kapı dizileri sabit, sadece bir kez programlanabilir ve karmaşık yapılar için son derece yetersiz yapılardır. GAL ise PLA gibi yapılara göre biraz daha kapasiteli ve tekrar programlanabilir olmasına rağmen karmaşık ve geniş yapıların tasarımı için uygun değildir. CPLD ise saydıklarımız içinde en kapasiteli ve tekrar programlanabilme özelliğine sahip bir türdür. Fakat içindeki dizi yapıları sabit olduğundan daha ziyade kombinasyonel lojik tasarımlar için uygundur. Bütün bu eksikliklerle ASIC (Application Specific Integrated Circuit) tasarımında oldukça vakit kaybettiğini düşünen elektronik dünyası FPGA tümleşik devresini tasarlamıştır. Şekil 2.14'te FPGA'ın doğuşu özetlemiştir.

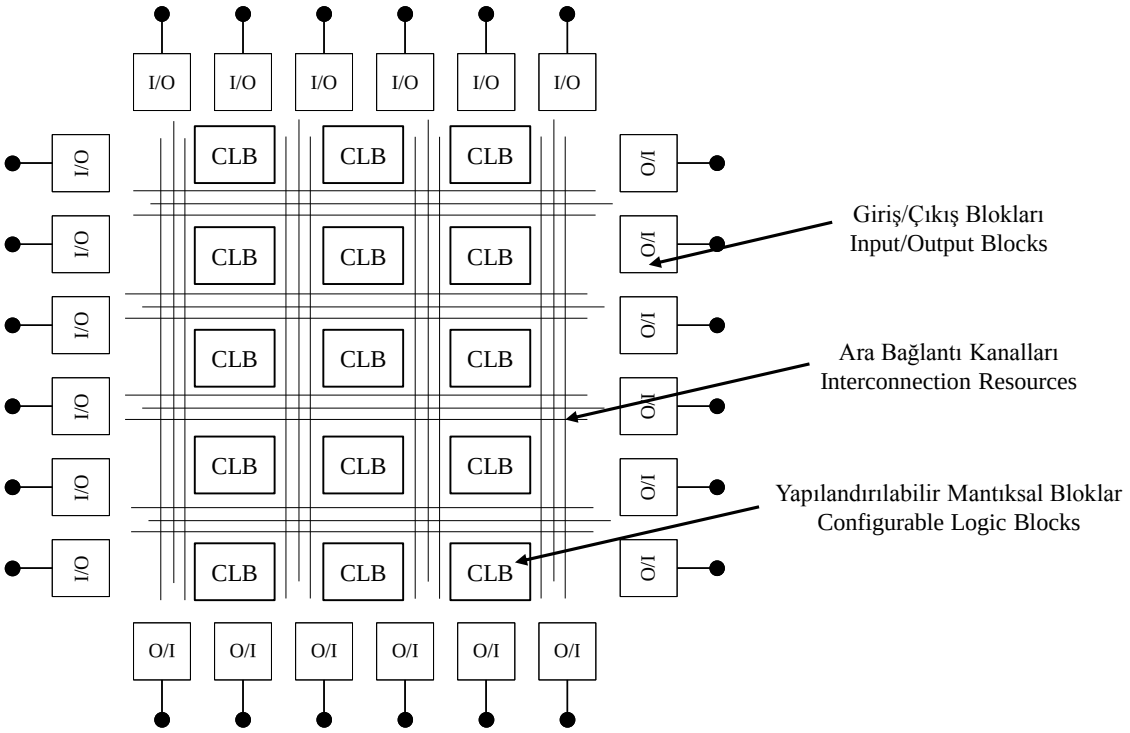
Öyle ki; FPGA'lar içinde her kapı bir diğerinden bağımsız, oldukça büyük kapasiteye sahip, bağımsız kapıların ilişkilendirilmesi tamamen kullanıcıya bırakılmış ve LUT (Look-Up Table) mantığına göre çalışan yapılardır. Yani CPLD'ler gibi sadece kombinasyonel yapılara değil içerisinde olası durum makinaları barındıran ardışık yapıların tasarımına da imkân sağlamaktadır. Ticari anlamda ilk FPGA (XC2064) Xilinx firması tarafından 1985 yılında piyasaya sürülmüştür (Öztürk, 2010).



Şekil 2.14. FPGA'ların Doğuşu (Öztürk, 2010).

2.2.2. FPGA Mimarisi

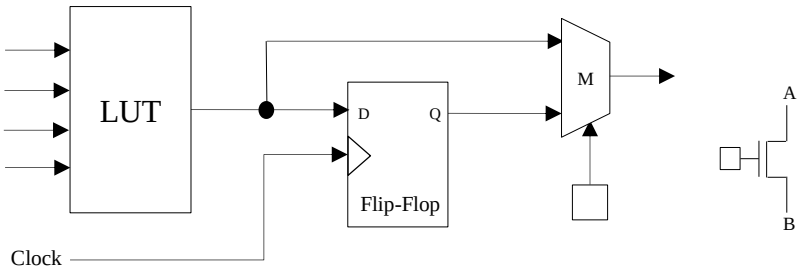
Genel yapısı Şekil 2.15'te gösterilmiş olan FPGA çiplerinin yapısındaki mantıksal bloklar, aralarındaki bağlantılar ve giriş/çıkış blokları sırayla aşağıda açıklanmıştır.



Şekil 2.15. FPGA Mimarisi (Joseph, 2005).

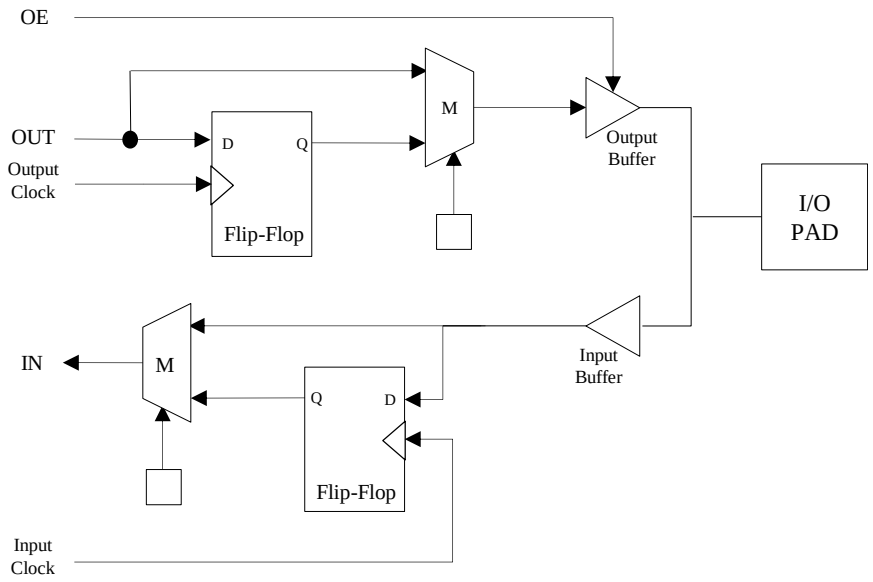
a. Programlanabilir Mantıksal Bloklar (Configurable Logic Blocks (CLB)): CLB'ler, mantıksal (boolean) fonksiyonların oluşturulabildiği LUT (Look-Up Table – Bakılan Tablo-(Şekil 2.16)), Carry Logic ve Flip-Flop'lardan oluşmaktadırlar. Tipik bir FPGA onbinlerce CLB ve flip-flop içerebilir. Büyüklük ölçüsü olarak, CLB'lerin giriş-

çıkış sayısı, CLB'lerin oluşumunda kullanılan transistor sayısı veya CLB'lerin gerçekleyebileceği mantıksal fonksiyon sayısı kullanılmaktadır (Bürhan ve Gülenç, 2011). Hafıza kapasitesi (LUT) giriş sayısı ile sınırlıdır. CLB'ler, kullanıcının oluşturmak istediği mantıksal devre için fonksiyonel elemanlar sağlar. CLB mimarisinin esnekliği ve simetriliği, uygulamaların kolaylıkla yerleştirilmesine olanak tanır.



Şekil 2.16. FPGA Yapılandırılabilir Lojik Blok (Configurable Logic Block (CLB)) (Baumann, 2010).

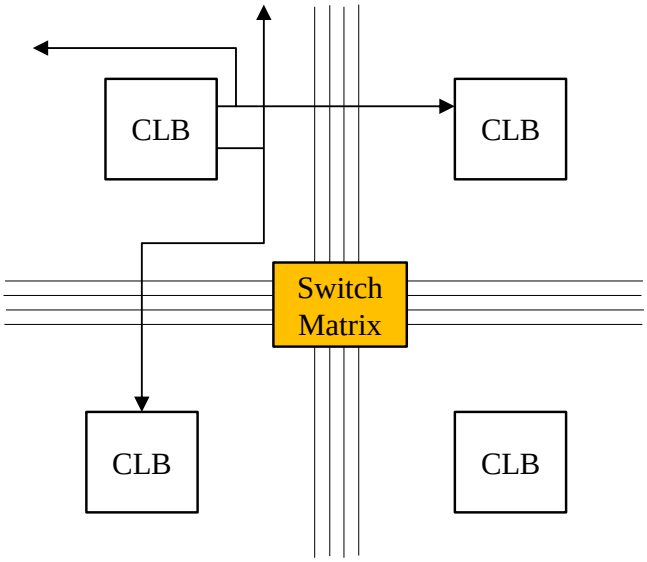
b. Giriş/Çıkış Blokları (I/O Blokları (IOB)): IOB'ler (Şekil 2.17) çipin iç sinyal hatları ile pinleri arasında programlanabilir arabirim görevini yaparlar. IOB'ler sayesinde FPGA'ların pinleri giriş, çıkış ya da çift yönlü olarak programlanabilirler. FPGA çipinin türüne göre bir çipteki IOB sayısı (dolayısıyla pin sayısı) 1000'li sayılara kadar çıkabilmektedir.



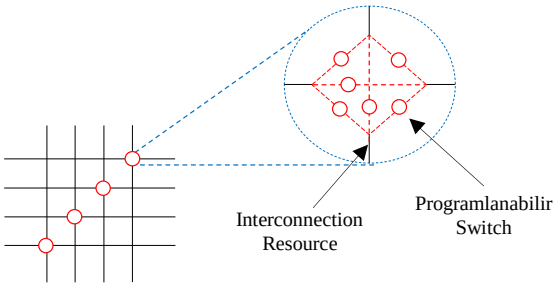
Şekil 2.17. FPGA Programlanabilir I/O Blok (Baumann, 2010).

c. Ara Bağlantı Kanalları (Interconnection Resources): Şekil 2.18'de açık şema ile gösterilen bu birimler hem CLB'ler arasında hem de CLB'ler ile IOB'ler arasında bağlantıları yapılandırmada kullanılırlar. Programlanabilir olduklarından çok esnek bir yapıya sahiptirler (Koyuncu, 2008).

Şekil 2.19'da Yatay ve dikey kanalların birleştiği yerlerde matris anahtarlar (switch matrix) vardır. Bu anahtarların içinde altı transistorlu (pass transistor) yönlendirme mekanizması bulunur. Programlanabilir bu anahtarlar sayesinde giriş yapılan taraftan kendisine komşu diğer üç tarafa yönlendirilme sağlanabilir (Sarıtekin, 2011).



Şekil 2.18. FPGA Programlanabilir Bağlantı [9].



Şekil 2.19. Switch Matrix [11].

2.2.3. FPGA Uygulamaları

FPGA'lar ilk başta ASIC tasarımların prototiplerini oluşturmak ve bunlar üzerinde simülasyon yoluyla fonksiyonel doğrulama yapmak amacıyla geliştirilmişlerdir. Bununla birlikte, geliştirme maliyetlerinin düşük olması ve kısa sürede pazara sunulabilme özelliklerinden dolayı son ürün yelpazesindeki yerleri zamanla artmıştır.

2000'li yılların başında milyonlarca kapı içeren yüksek performanslı FPGA'lar piyasada yerlerini aldılar (Sarıtekin, 2011).

FPGA uygulamaları; medikal resimleme, robotik, ses tanıma, şifreleme, biyoenformatik, havacılık, uzay teknolojileri ve taşıt teknolojileri gibi alanlarda kullanılmaktadır. Ayrıca hızlı üretim avantajlarından dolayı her hangi bir uygulamanın ASIC tabanlı versiyonunun fabrikasyon süreci tamamlanana kadar bu uygulamanın ilk sürümleri genellikle FPGA tabanlı olarak piyasaya sunulur. Böylelikle piyasada zaman açısından avantaj sağlanmış olur. FPGA'lar başlangıçta piyasaya CPLD'lerin rakibi olarak sürüldüler. Büyüklükleri, kapasiteleri ve hızları attıkça piyasada SoC (full Systems on a Chips) yaklaşımı ile daha da fazla yer almaya başladılar (Sarıtekin, 2011).

2.3. FPGA'DA YSA UYGULAMALARI

FPGA tabanlı YSA'lar oldukça kullanışlı ve birçok karmaşık işlemi doğru olarak yapabilecek kapasiteye sahiptirler (Yılmaz, 2008). YSA'ların FPGA ile gerçeklenmesi programlanabilir sistemlerde esneklik sağlar. ASIC olarak üretilen YSA uygulamalarında sonradan değişiklik yapmak mümkün değildir. Değişiklik yapılmak istendiğinde tasarım ve fabrikasyon işlemleri baştan sona yenilenmelidir. Oysa FPGA'ler tekrar yapılandırılabilme özelliklerinden dolayı farklı YSA yapılarının tekrar tekrar gerçeklenebilmesine olanak sağlayabilmektedirler (Çavuşlu, 2007). Ayrıca FPGA tabanlı YSA'lar da düşük sayı duyarlılığı kullanılarak yapılan tasarımların hem zaman hem maliyet açısından çok büyük avantajları vardır. Bunun yanında yapılan gerçek zamanlı ve yoğun matematiksel işlemler gerektirmeyen tasarımlar oldukça başarılı sonuçlar vermektedirler. Tekrar düzenlenebilir FPGA yapısı ile özel amaçlı hızlı donanımlar çok geniş uygulamalar için tasarlanabilmektedirler.

YSA'ların FPGA çipleri kullanılarak gerçeklenmesi konusunda literatürde birçok çalışma bulmak mümkündür. Aşağıda seçilmiş bazı çalışmalar özet olarak sunulmaktadır.

Yapay Sinir Ağlarının Otomatik Olarak FPGA'ya Uygulanması İçin Veri Yolu Tasarım Aracı. Sarıtekin bu çalışmasında, yapay sinir ağlarının FPGA'ya uygulanmasının otomatikleştirilmesi, bu işlem için uzman gereksiniminin azaltılması ve uygulama sürecinin kısaltılması amacıyla, YSA'lar için otomatik veri yolu tasarımı yapabilen bir araç (Yapay Sinir Ağlarının VHDL Kodunu Oluşturucu - Artificial Neural

Network Generator (ANNGEN)) geliştirmiştir. Bu kapsamda öncelikle ANNGEN tarafından kullanılan ve yapay sinir hücrelerinden oluşan örnek bir yapay sinir hücresi kütüphanesi oluşturulmuştur. Kütüphanede hâlihazırda altı değişik sinir hücresi bulunmaktadır. İstenildiğinde kütüphaneye yeni hücreler eklenebilmekte ve bunlar ANNGEN tarafından otomatik olarak tanınmaktadır. ANNGEN'in girdileri, oluşturulmak istenen yapay sinir ağının metin tabanlı tanımlaması (*NetList*), sinir hücresi kütüphanesi (Library), şablon dosyasıdır (Template). ANNGEN, FPGA çiplerine uygulanabilecek formatta tasarlanmak istenen yapay sinir ağı için gerekli veri yolu tasarımını yapan ve VHDL (Very High Speed Integrated Circuit HDL (Çok Hızlı Entegre Devre Donanım Tanımlama Dili)) kodunu üreten bir yazılım aracı olarak geliştirilmiştir (Sarıtekin, 2011).

ANNGEN ile oluşturulan YSA veri yolunun çalışabilmesi için bir denetleyici birime ihtiyaç duyulmaktadır. Bu çalışmada gerekli denetleyiciyi otomatik olarak tasarlayan bir Elektronik tasarım aracı geliştirilmiştir.

FPGA ile Yapay Sinir Ağı Eğitiminin Donanımsal Gerçeklenmesi: Bu çalışmada Çavuşlu, FPGA kullanarak YSA'ların eğitiminin donanım ile gerçeklemiştir. Dijital sistem mimarisi, geriye yayılım algoritması ile çok katmanlı YSA'ların eğitiminin gerçeklenmesi için tasarlanmıştır. Tasarım VHDL (Very High Speed Integrated Circuits Hardware Description Language) dilinde tanımlanmış ve FPGA entegre devresi üzerinde gerçeklenerek FPGA demo kart üzerinde test edilmiştir (Çavuşlu, 2006).

Yapay Sinir Ağı Eğitiminin IEEE 754 Kayan Noktalı Sayı Formatı İle FPGA Tabanlı Gerçeklenmesi: Bu bildiride Çavuşlu ve diğerleri ileri beslemeli iki katmanlı bir YSA'nın ÖZEL-VEYA problemi temel alınarak geriye yayılım algoritması ile eğitiminin FPGA üzerinde gerçeklenmesini sunmuştur. YSA eğitim işlemlerinin [Çok Katmanlı Algılayıcı (ÇKA) ve Geriye Yayılım (GY)] FPGA üzerinde paralel çalışacak şekilde gerçeklenmiştir. Eğitimin gerçeklenmesinde Xilinx FPGA ailesine ait 2vp30fg676-7 çipi kullanılmıştır. Sonuçta, az yer kaplayan ve çıkışta elde edilen hata değerinin önemsenmeyecek kadar küçük olduğu bir YSA eğitimi elde edilmiştir (Çavuşlu ve diğ., 2008).

Sahada Programlanabilir Kapı Dizileri (FPGA) Üzerinde Bir YSA'nın Tasarlanması Ve Donanım Olarak Gerçekleştirilmesi: Yılmaz bu çalışmasında, çip

üzerinde eğitilebilir bir YSA yapısını Altera FPGA çipleri ile gerçekleştirmiştir. Çalışamda XOR problemi ve bir sensör doğrusallaştırma problemi üzerinde çalışılmış, sabit noktalı sayı sistemi tabanlı çalışan 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 (Yılmaz, 2008).

FPGA' deki Esnek Yapay Sinir Ağı Eğitimi için Karma Donanım-Yazılım Yaklaşımı: Bu çalışmada Ramón J. Aliaga ve diğerleri, YSA'ların yalnızca donanımsal olarak gerçeklendiklerinde ya istenen büyüklükte ağın oluşturulamadığını ya da istenen hassasiyette veri temsili yapılamadığını belirtmektedirler. Çözüm olarak donanım ve yazılım tabanlı karma bir yapı önermektedirler. Çalışmada YSA eğitimini hızlı bir şekilde yapabilmek için iki işlemcili bir yaklaşım kullanılmıştır. İşlemcilerden Master işlemci, yazılım tabanlı çalışmakta ve rastgele büyüklükte bir YSA'nın eğitim işlemini koordine etmektedir. Master işlemci eğitim sırasında gerek duyulan vektör işlemleri için FPGA üzerinde çalışan bir yardımcı işlemciyi kullanmaktadır. Yardımcı işlemci içerisinde paralel çalışan işlem üniteleri (Processing Units – PU) barındırdığından hızlı bir şekilde verilen vektörler üzerinde istenen işlemleri yapmakta ve dolayısıyla eğitim işlemi hızlandırılmaktadır. (Ramón J. Aliaga ve diğ., 2009).

Adım (Step) Motor Düşük-Hız-Sönümlenme İşlemi İçin Tasarlanan YSA Tabanlı Denetleyicinin FPGA İle Gerçeklenmesi: Quy Ngoc Le and Jae-Wook Jeon bu çalışmalarında, YSA tabanlı bir denetleyicinin FPGA'de gerçeklenmesini anlatmışlardır. Bu denetleyici, doğrusal olamayan fonksiyon yaklaşımları için YSA tabanlı olarak geliştirilmiştir. Çıktı olarak kompozit denetleyici YSA çıkışı ve hata izleme kısmından oluşmaktadır. Denetleyici, geleneksel faz ilerlemesi (PA) denetleyicisi ile karşılaştırıldığında değişen referans hızında daha az etkili olduğu görülmüştür. Ancak sabit referans hızı altında önerilen PA denetleyicisine kıyasla geliştirilmiş özellikler sergilemektedir.

Geliştirilen denetleyici, herhangi bir tanımlama işlemi olmadan çevrimiçi geri yayılımlı öğrenme uygulamaları sayesinde sistem parametrelerini öğrenebilmekte ve verimli sönümlenme gerçekleştirebilmektedir. Bu süreç sonucunda, bir YSA tabanlı denetleyici

uygulaması, geri yayılım algoritmaları ile tek bir sahada programlanabilir kapı dizisi (FPGA) üzerinde gerçeklenmiştir (Quy Ngoc Le and Jae-Wook Jeon, 2010).

Yapay Sinir Ağları ile RadBas, LogSig, TanSig Aktivasyon Fonksiyonlarının FPGA Üzerinde Tasarımı ve Uygulaması: Bu çalışmada Şahin ve diğerleri, 2, 4 ve 6 girişli biaslı ve biassız her birinde 3 farklı transfer fonksiyonu bulunan toplam 18 farklı yapay sinir hücresi tasarlamışlardır. Hücreler için seçilen transfer fonksiyonları Radial-Basis (RadBas), Logaritmik-sigmoid (*LogSig*) ve Tanjant-sigmoid (*TanSig*) fonksiyonlarıdır. Bu seçilen fonksiyonların ortak özelliği e^x fonksiyonunun hesaplanmasına ihtiyaç duymalarıdır. Tasarlanan yapay sinir hücreleri Virtex-6 FPGA çipi için sentezlenerek hız ve kapasite ölçümü yapılmıştır. Yapılan ölçümlerde hücrelerin saniyede 2.89 ile 6.53 milyon arasında giriş veri setini işleyerek sonuç üretebildikleri ve ayrıca tek bir FPGA (virtex-6) çipine en fazla 10 adet hücrenin yerleştirilebileceği hatta virtex-6'nın daha büyük versiyonlarına daha fazla sayıda hücrenin aynı anda sığdırılabileceği belirtilmiştir (Şahin ve diğ., 2011)

2.4. SONLU DURUM MAKİNELERİ (FINITE STATE MACHINE)

Sonlu Durum Makineleri (Finite State Machine (FSM)), sınırlı sayıdaki durumların arasındaki iletişimini modellemek için kullanılırlar. Bu modelleme, sistemin bir başlangıç durumundan belirli bir olayı temel alarak başka bir duruma geçişi şeklinde açıklanabilir. Bu geçiş, belirli bir durum içerisinde daha önceden tanımlanmış bir olayın gerçekleşmesi ile sağlanmaktadır. Yani o anki durum ile dışardan gelen girdilere bağlı olarak çalışırlar. Sonlu durum makineleri, FPGA'ların programlanmasında kullanılan en önemli metotlardandır.

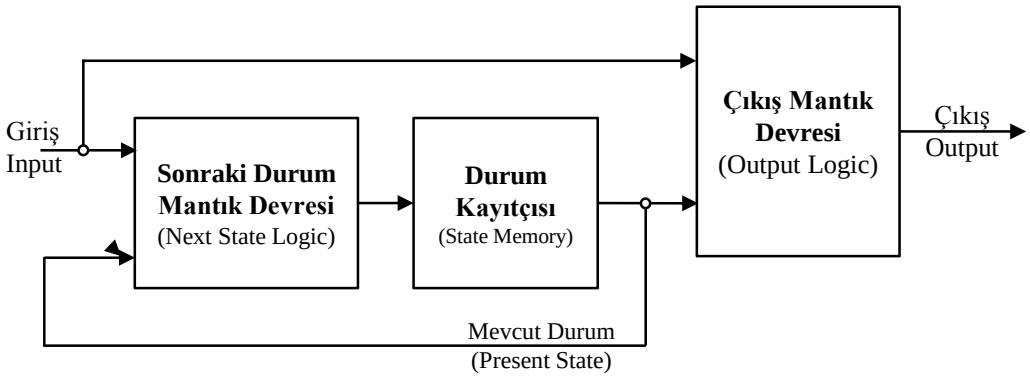
Sonlu durum makineleri için kabul edilen iki genel tasarım yöntemi vardır. Bunlar Moore Machine ve Mealy Machine tasarımlarıdır.

Moore Machine modelinde, sistem akışı sadece makinanın bulunduğu duruma göre belirlenir. Girişler doğrudan çıkışı belirlemede kullanılmazlar. Bunun anlamı çıkış sinyalleri sadece sistem durum değiştirdiğinde değişebilir, giriş değiştiğinde değişmez.

Mealy Machine modelinde ise çıkışlar hem o anda bulunan duruma hem de giriş değerlerine göre belirlenir. Yani bunun anlamı çıkışların makinanın durum değiştirmesinden bağımsız olarak girişler değiştiğinde anlık olarak değişebilmesidir.

2.4.1. Mealy Makinesi (Mealy Machine)

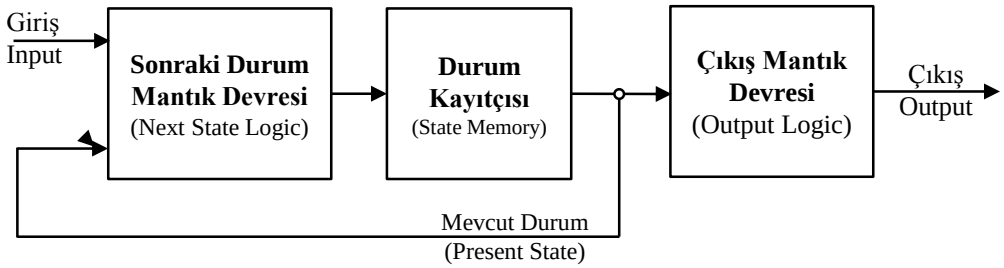
Şekil 2.20'de tasarımı verilmiş mealy makinesinde çıkış değerleri, yalnızca durum değişkenlerine değil giriş değişkenlerine de bağlıdır. Bir kombinezonsal devrenin çıkışı olan bu değerler, saat darbeleriyle değişen durum değişkenlerinden etkilendiği gibi, saat darbelerinin dışında değişen giriş değişkenlerinden de etkilenirler. Bu nedenle çıkışta hatalı çıkış diye adlandırılan istenmeyen çıkışlar görülebilir (Nuss R., 1999).



Şekil 2.20. Mealy Durum Makinesi.

2.4.2. Moore Makinesi (Moore Machine)

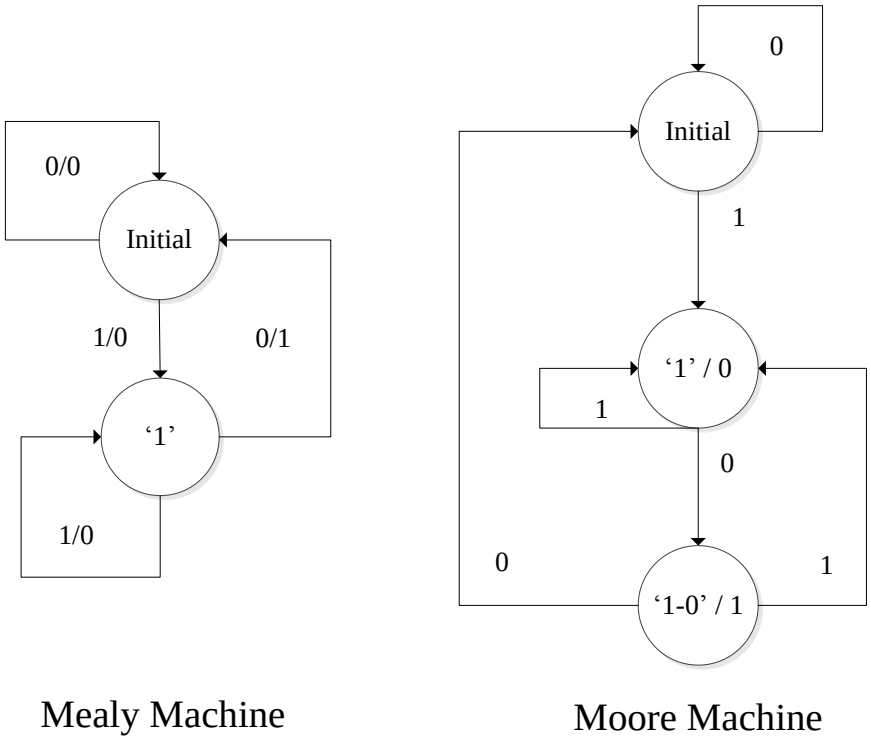
Şekil 2.21'de basit diyagramı verilen Moore Durum Makinesi, iki makine tipinden en basit olanıdır. Moore makinesinde çıkışlar yalnız durum değişkenlerinin değerlerine göre belirlenir. Girişlerdeki değerler sadece bir sonraki durumu belirler. Moore modelinin avantajı davranışın basit olmasıdır.



Şekil 2.21. Moore Durum Makinesi.

2.4.3. Mealy ve Moore Durum Makinelerinin Örnek İle Karşılaştırılması

‘10’ durumunu denetleyen bir sonlu durum makinesi düşünelim. Mealy ve Moore makineleri için temsili durum diyagramları Şekil 2.22’de gösterilmiştir. Buna göre durum diyagramları için durum geçişlerine ait giriş çıkışlar yay üzerindeki ilgili etiketler ile listelenmiştir. Moore makinesi, girdilere bakılmaksızın her durum için benzersiz bir çıktı üretir. Moore makinesi durum diyagramına göre sistemin durumu ile çıkış ilişkilendirilir. Yani çıkış sinyali değişimi sadece sistemin durum değiştirmesine bağlıdır. Bir Mealy makinesinde ise çıkış, hem durum geçişlerindeki değişim hem de giriş değerleri ile ilişkilendirilir. Bir çıkış dizisinde, Moore makinesi ile karşılaştırıldığında Mealy makinesi kullanılarak daha az durum oluşturabilir.



Şekil 2.22. '10 'Sıra Dedektörü için Mealy ve Moore Makineleri Durum Diyagramları.

2.4.4. Sonlu Durum Makineleri (FSM) İçin Durum Kodlama (Encoding) Teknikleri

Bir sayısal mantık tasarımcısı için FSM, tasarlaması en yaygın olan fakat zorlu bir iştir. Bir FSM tasarımını optimize etmek için en önemli faktörlerden biri durum kodlama tekniğinin seçimidir. Bu seçim, mantık fonksiyonlarının karmaşıklığını, devrelerin donanım maiyetlerini, zamanlama sorunlarını, güç kullanımını vb. etkilemektedir. Tasarımcının seçim işlemi, teknoloji, tasarım vb. gibi faktörlere bağlıdır. En sık

kullanılan durum kodlama teknikleri; Binary Encoding, Gray Encoding, One-Hot Encoding olarak bilinir. Bu kodlama teknikleri *n* sayıdaki FSM durumları için şu şekilde oluşturulmaktadırlar ([10], 2012).

2.4.4.1. İkili Kodlama (Binary Encoding) Tekniği

Durum kodu sadece ikili sayı sisteminden oluşmaktadır. Bitlerin sayısı, $\log_2 n$ fonksiyonunu sonucunun bir sonraki doğal sayıya yuvarlanarak oluşturulması ile bulunur. Diyelim ki n=6 için; bit sayısı 3 ve durum kodları:

S0 → 000
S1 → 001
S2 → 010
S3 → 011
S4 → 100
S5 → 101 şeklinde ifade edilirler ([10], 2012).

2.4.4.2. Gri Kodlama (Gray Encoding) Tekniği

Gray kodlama yöntemi, ikili (binary) kodlar olarak da bilinen yüksek hassasiyet gerektiren durumları temsil eden kodlama tekniğidir. Gray kodlama yönteminde bir durumdan diğerine geçerken sadece bir bit değişir. Bunun nedeni oluşabilecek istenmeyen yanlış durumların ortadan kaldırılmasıdır. Bitlerin sayısı binary kodlamada olduğu gibi n durum sayısın $\log_2 n$ olarak çıkan sonucun yuvarlanarak bir sonraki doğal sayıya eşitlenmesi ile elde edilir. Yani eğer n=4 ise, o zaman bit sayımız 2 yani gerekli olan flip-flop sayısı 2 ve durum kodlarımızda;

S0 → 00
S1 → 01
S2 → 11
S3 → 10 şeklinde oluşacaktır ([10], 2012).

2.4.4.3. Bir Aktif Kodlama (One-Hot Encoding) Tekniği

One-Hot kodlama yöntemi hızlı tasarım için kullanılan bir yöntemdir. Bu teknikte her bir durum için bir bit kullanılır ve durum makinası herhangi bir durumda iken sadece bir bit bit aktiftir, diğer tüm durum bitleri sıfırdır. Bu yöntemde *n* tane durumu olan bir

makinayı oluşturmak için *n* tane flip-flop gereklidir. Tasarım zamanında kazanç sağlanırken fazladan donanıma ihtiyaç duyulur;

S0 → 00001

S1 → 00010

S2 → 00100

S3 → 01000

S4 → 10000

One-Hot kodlama tekniğinin başlıca avantajları aşağıda sıralanmıştır. Buna göre;

- Bir tasarımı değiştirmek kolaydır. Durum ekleme, durum silme veya değişen durum geçişleri (FSM içerisindeki kombinezonsal devre) tasarımın geri kalanını etkilemeksizin gerçekleştirilebilmektedir.
- Diğer kodlama tekniklerinden daha hızlıdır. Hız, durumların sayısından bağımsız olup sadece belirli bir surum içerisindeki geçişlerin sayısına bağlıdır.
- Tasarımın kritik yolunun (critical path) bulunması daha kolaydır. (Statik zamanlama analizi).
- One-Hot kodlama tekniği özellikle FPGA uygulamaları için daha avantajlıdır. Büyük bir FSM tasarımı FPGA'e uygulandığında, binary ve gray kodlama tekniklerinde one-hot kodlamaya göre daha az flip-flop kullanılacak ve daha düzenli olacaktır. Fakat durumun çözülmesi ve düzenli olabilmesi için ilave mantık blokları kullanılacaktır. Dolayısıyla kodlama işlemi ve kod çözümü nedeniyle binary ve gray tekniklerinde daha fazla mantık blokları kullanılacaktır.
- Düşük geçiş aktivitesi nedeniyle düşük güç tüketimi ve hatalara daha az eğilimli oluşacaktır ([10], 2012).

2.5. FPGA'LERDE SONLU DURUM MAKİNELERİ (FINITE STATE MACHINE) YÖNTEMİ İLE DENETLEYİCİ (CONTROLLER) TASARIMI UYGULAMALARI

Sonlu durum makineleri, FPGA'ların programlanmasında kullanılan en önemli metotlardandır. FPGA tabanlı gerçeklenen sistemlerin birçoğunun alt yapısında bu metotlar ile karşılaşmak mümkündür. Bu sebeple çalışmamıza kaynak olabilmesi açısından sonlu durum makineleri yöntemi ile denetleyici tasarımları adı altında yapılan çalışmalar incelenmiş ve bu bölümde özet olarak sunulmuştur.

Sonlu Durum Makinesi (Finite State Machine) Otomatik Kod Üretimi: Lazareviæ ve Miliæv bu çalışmada, sonlu durum makinesi için kod üretimi gerçekleştiren bir yazılım aracını anlatmışlardır. Yazılım aracının ana özelliği esnek bir yapıya sahip olmasıdır. Bu esneklik kullanıcı ihtiyaçlarına göre kod üretim yolunda istenilen şekilde kod üretimi ile ilgili tanımlamaların yeniden yapılabilmesini sağlamasıdır. Bu tanımlamadan sonra yazılım aracı asgari düzeyde değişiklikler yaparak çok kısa sürede kullanıcının istediği kod üretimini gerçekleştirmektedir.

Oluşturulan aracın kullanımı oldukça basittir. Yazılım aracı Signaling System No7 TACP's (Transaction Capabilities Application Part) için sonlu durum makinesi kod üretimi gerçekleştirilerek denenmiştir (Lazareviæ ve Miliæv).

Sinir Ağı Denetleyicisi ve Sonlu Durum Makinesi Denetleyicisi Karşılaştırılması: Pintér ve diğerleri bu çalışmalarında, basit bir robot problemi için iki farklı denetleyici modelinin performanslarını karşılaştırmışlardır. Burada problem bir robotun iki aydınlatma sensörü yardımı ile sensör aralık dışındaki ışık kaynağına ulaşmasıdır.

Birinci denetleyici tam bağlantılı sinir ağı (fully meshed artificial neural network) modelidir. Bu model kullanılan denetleyici modellerinin en başında gelir. Oluşabilecek sinir ağı problemlerini anlamada ve değerlendirmede kullanılan en etkili modeldir.

Bu sorunu çözebilmek için ayrıca sonlu durum makine tiplerinde mealy makine tipi kullanılmıştır.

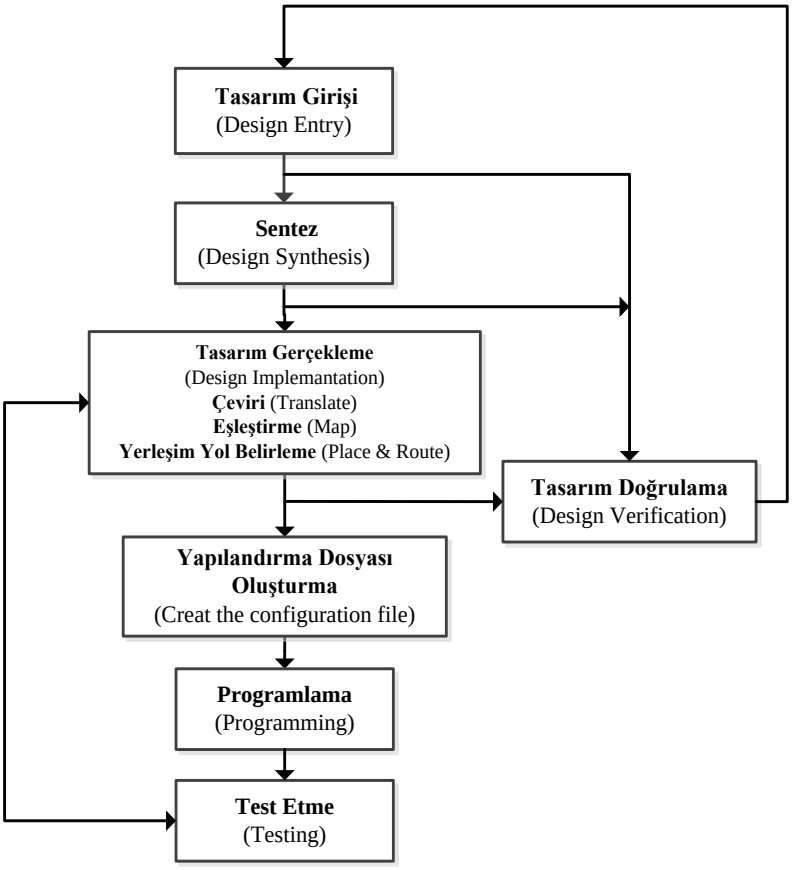
Burada sonuç olarak, her iki denetleyici de sensör aralığının dışına yerleştirilmiş bir ışık kaynağını geliştirilmiş evrimsel algoritmalar yardımı ile bulabilmektedirler. Fakat sinir ağı denetleyici modeli hız ve başarı olasılığı açısından daha iyi performans gösterirken, geliştirilmiş bir mealy sonlu durum makinesi daha anlaşılır bir iç yapıya sahiptir (Pintér ve diğ., 2012).

2.6. ELEKTRONİK TASARIM OTOMASYONU

2.6.1. Elektronik Tasarım Otomasyonuna Giriş

2.6.1.1. FPGA Tasarım Akışı

FPGA tasarım akışı, tasarım girişinden FPGA'e yüklenecek kodun üretilmesine kadar bir dizi adımları içerir. Şekil 2.23'te ana hatları ile gösterilmiş olan bu tasarım akışı sırasında birçok tasarım aracının kullanılması gereklidir.



Şekil 2.23. Tasarım Süreci Akış Diyagramı [12].

2.6.1.2. Tasarım Girişi (Design Entry)

Bu süreçte ilk olarak tasarlanacak uygulamanın sözel tanımlaması yapılır. Daha sonra uygulamaya ait bir sistem tasarımı, yüksek seviyeli HDL donanım tanımlama dillerinden biri (VHDL/Verilog) kullanılarak, şematik girişi, FSM diyagramları oluşturulması, mantık tabloları kullanılması yöntemlerinden biri ile yapılabilir. Bu aşamada FPGA üreticisinin geliştirme ortamları kullanılabilineceği gibi sadece bir text programı kullanarak da HDL tasarım girişi gerçekleştirilebilir. Çoğu text programları VHDL ve Verilog dillerini tanımakta ve kod yazarken bize kolaylıklar sunmaktadır. Şekil 2.24'te VHDL dili ile yazılmış örnek bir kod parçası gösterilmektedir ([7], 2012).

```
11  library IEEE;
12  use IEEE.std_logic_1164.all;
13  USE IEEE.std_logic_arith.all;
14  USE IEEE.std_logic_unsigned.all;
15  entity UPCOUNTER is
16    generic (WIDTH : integer := 32);
17    port (
18      RST   : in std_logic;
19      CLK   : in std_logic;
20      LOAD  : in std_logic;
21      INC   : in std_logic;
22      DATA  : in std_logic_vector(WIDTH-1 downto 0);
23      Q     : out std_logic_vector(WIDTH-1 downto 0));
24  end entity UPCOUNTER;
25  architecture RTL of UPCOUNTER is
26    signal CNT : std_logic_vector(WIDTH-1 downto 0);
27  begin
28     process
29     begin
30        WAIT UNTIL ((CLK'EVENT) AND (CLK= '1'));
31        IF (RST = '1') THEN
32           CNT <= (others => '0');
33        ELSE
34           IF (Load = '1') THEN
35              CNT <= DATA;
36           ELSE
37              IF (INC = '1') THEN
38                 cnt <= cnt + 1;
39              ELSE
40                 cnt <= cnt;
41              END IF;
42           END IF;
43        END IF;
44     end process;
45    Q <= CNT; -- type is converted back to std_logic_vector
46  end architecture RTL;
47
48  --------------------------------------------------------------------------------
```

Şekil 2.24. VHDL Kod Parçası.

2.6.1.3. Sentez (Synthesis)

Sentez adımı, tasarım girişini devre bağlantılarına dönüştüren adımdır. Bu aşamada bir sentez aracı ile uygulamaya ait lojik kapı seviyesinde bağlantı listesi (Gate Level NetList) oluşturulur. Sentezleme işlemi sırasında ayrıca varsa, devreye ait lojik kısıtlamalar (Giriş/Çıkış bacakları, zamanlama, yerleştirme, saat frekansı, kritik yollar gibi) kullanıcı kısıtlama dosyası (user constraints file, Xilinx) dikkate alınarak

sentezleme yapılır. Yine bu aşamada, istenilen fonksiyonların en iyi şekilde gerçeklenebilmesi için gerekli lojik indirgemeler (logic optimization) de yapılır.

2.6.1.4. Eşleştirme (Map)

Bu adım da, oluşturulan sentez dosyası varsa kullanılan diğer sentez dosyaları ile birleştirilerek son halini alır. Eşleştirme aşamasında yapılan tasarım ile hedef FPGA çipinde bulunan kaynaklar arasında eşleme yapılır. Yani tasarımdaki hangi bölümlerin FPGA çipinde hangi bölümler kullanılarak gerçeklenebileceği belirlenir. Örnek; bir VE kapısının (OR), LUT (Look-up-table) içerisine gireceğine karar verilmesi gibi.

2.6.1.5. Yerleşim ve Yol Belirleme (Place & Route)

Eşleştirme işleminden tasarımın FPGA çipi içerisinde hangi konumdaki kaynaklara yerleştirileceği ve bunlar arasındaki bağlantıların nasıl yapılacağı belirlenir. Bu son aşama aslında bir optimizasyon aşamasıdır. Burada hedef, uygulamanın FPGA çipi içindeki CLB'lere en sıkışık şekilde nasıl yerleştirilmesi ve bu yerleşimde toplam bağlantı yolunun olabilecek en kısa değerde tutulmasıdır. Bu son üç aşama (implementation) gerçekleme aşaması olarak da anılır. Gerçekleme aşamasında ayrıca uygulama içindeki kritik yol (Critical Path) ve bu yola bağlı olarak da uygulamanın çalıştırılabileceği en yüksek saat frekansı belirlenir.

2.6.1.6. Simülasyon (Simulation)

Hem tasarım girişinden sonra hem de yerleşim aşamasından sonra yapılabilecek simülasyon adımıdır. Tasarım girişi adımından sonra, tasarımın istenilen fonksiyonu yerine getirip getirmediğinin test edildiği adıma "functional simulation/fonksiyonel simülasyon", yerleşim aşamasından sonra zamanlama bilgileri ile birlikte gerçekleştirilen test adımına ise "timing simulation/zamanlama simülasyonu" denilmektedir. Sentez adımında olduğu gibi, simülasyon için çip üreticilerinin programları kullanılabilineceği gibi diğer firmalar tarafından geliştirişmiş yazılım araçları da kullanılabilir. Tasarım sürecinin her aşamasından sonra uygulamayı simülasyon yoluyla test etmek mümkündür.

2.6.1.7. Kaynak Kodu Oluşturma (Bitstream Generation)

Tasarımın tüm fonksiyonları ile test edilip doğrulandıktan ve yerleşim aşamasına geçildikten sonra FPGA çipine yüklenecek formatta bir bitstream (yapılandırma dosyası) oluşturulur. Bu dosyanın hedef çipe yüklenmesi ile tasarım aşaması sonlanmış olur.

FPGA'e yüklenecek bitstream oluşturulduktan sonra, JTAG (Joint Test Action Group (Ortak Test Eylem Grubu)) ara yüzü ile bu konfigürasyon dosyası karta yüklenebilir. SRAM tabanlı FPGA entegreleri bu dosyayı her açılışta uçucu olmayan bir bellekten okuyup konfigüre oldukları gibi, Flash tabanlı FPGA entegreleri bu kodu içlerinde de tutabilirler. FPGA'ler JTAG ara yüzleri sayesinde karta lehimli iken bile tekrar tekrar programlanabilirler.

Son aşamada ise yapılması gereken FPGA'in fonksiyonlarının kart üzerinde test edilmesidir. Yine bu noktada üretici firmaların FPGA içindeki sinyallerin gözlemlenmesini sağlayan çeşitli programları mevcuttur. Bu programlar sayesinde, kart çalışırken FPGA içindeki sinyallere bakılıp, gerçek çalışma durumlarındaki hataları tespit edilebilir.

2.6.2. FPGA Tasarım Araçları

Bu bölümde, FPGA çip üreticileri tarafından geliştirilmiş olan ve FPGA tabanlı uygulama geliştirmede kullanılan yazılım araçları ISE ve Quartus kısaca tanıtılmaktadır.

2.6.2.1. ISE

ISE Xilinx firması tarafından geliştirilmiş bir elektronik tasarım otomasyon (Eleckronic Desıgn Automation-EDA) aracıdır. Bünyesinde Xilinx firması tarafında üretilen FPGA'leri için uygulama gerçekleştirmeye yönelik bir çok alt elemanı entegre bir şekilde içerir. Ana amacı verilen tasarımı (şematik, VHDL, Verilog) FPGA'e yüklenecek kod haline (bitstream) getirmektir. ISE ile birlikte birçok yardımcı araç da kullanıcıya sunulmaktadır. Bu araçlardan birkaçı aşağıda kısaca açıklanmıştır.

- IPCoreGen (IP (Intellectual Property) Core Generation)

Xilinx IPCoreGen sayesinde FPGA cihazları için Plug-and-Play (Tak Çalıştır) tasarımlar kolayca otomatik olarak oluşturulabilmektedir. Xilinx belirli Plug-and-Play

tasarım ihtiyaçlarının karşılanması için geniş bir çekirdek kataloğu sunmuştur. Bu katalog sayesinde diğer programlara oranla daha hızlı ve hatasız tasarımlar gerçeklenebilmektedir.

- ISE Simülator

ISE tasarım araçları ile birlikte satın alınabilen bir benzeşim (simülasyon) programıdır. Geliştirilen uygulamaların fonksiyonel ve zamanlama testleri yapmak için kullanılır. Lojik simülasyon sonuçları grafiksel olarak (dalga şekilleri halinde) kullanıcıya sunulur.

- Chipscope

Xilinx firmasının ISE'den ayrı olarak sattığı bir programdır. Lojik Gözlemleyici (Logic Analyzer) olarak kullanılmak üzere geliştirilmiştir. Oluşan verileri bağlı olduğu bilgisayara gönderir ve böylece hiçbir ekstra donanım olmadan FPGA iç sinyallerini veya kartta FPGA'e bağlı sinyalleri gözlemleme imkânı sağlar.

- Impact

ISE programı ile birlikte gelen bir programdır. Program JTAG zinciri içerisinde olan entegreleri programlamaya yarar. Bunun için uygun bir programlayıcının bilgisayara bağlı olması gerekmektedir.

- EDK (Embedded Development Kit)

Xilinx FPGA'leri içerisinde soft (MicroBlaze, PicoBlaze) ve hard (PowerPC 405 ve 440) mikroişlemci yapılarının gerçeklenmesi ve bu mikroişlemcilerin çevre birimlerinin yönetilmesini sağlayan yazılımdır.

2.6.2.2. Quartus II

Altera FPGA'leri ile tasarım yapmak için gerekli ana yazılım aracı Quartus II'dir . Ana amacı verilen tasarımı (şematik, VHDL, Verilog) FPGA'e yüklenecek kod haline getirmektir. Quartus da birçok alt bölümden oluşur. Bu bölümlerden bazıları,

- Altera IP Library

Xilinx firmasının sunduğu IPCoreGen kataloğunun bir benzeridir. Kütüphanesinde; dijital sinyal işleme (DSP), bellek ara yüzü ve gömülü işlemciler için çeşitli IP çekirdek üniteleri barındırır.

- ModelSim-Altera Starter Edition

Quartus II yazılımı ile birlikte ücretsiz olarak sunulan Mentor Graphics firmasının simülasyon programıdır. Fonksiyonel ve zamanlama testleri yapmak için kullanılır.

- SignalTap II Embedded Logic Analyzer

FPGA iç sinyallerini, bağlı olduğu bilgisayarda herhangi bir ekstra donanım olmadan gözlemlemeye yarayan ara yüzdür. SignalTap II, Quartus II yazılımı içerisine gömülü olup konfigürasyon ve gözlem işlemleri yine Quartus II ortamı içerisinden gerçekleştirilir.

- Quartus II Programmer

Quartus II programı ile birlikte gelen bir programlayıcı ara yüzdür. Program, JTAG zinciri içerisinde olan Altera entegrelerini programlamaya yarar. Bunun için uygun bir programlayıcının bilgisayara bağlı olması gerekmektedir.

Diğer firma (third party) yazılımları

- Simülatörler

Diğer firmalar tarafından geliştirilmiş ve jenerik olarak kullanılabilen simülatörlere örnek olarak Aldec Active-HDL/Riviera-Pro, Mentor Graphics ModelSim, Cadence NCSim, Synopsys VCS simülatörü verilebilir.

- Sentezleyiciler

Yine diğer firmalar tarafından geliştirilmiş ve eş değer olarak kullanılabilen sentez araçlarında bazıları Synopsys Synplify Pro ve Mentor LeonardoSpectrum dur.

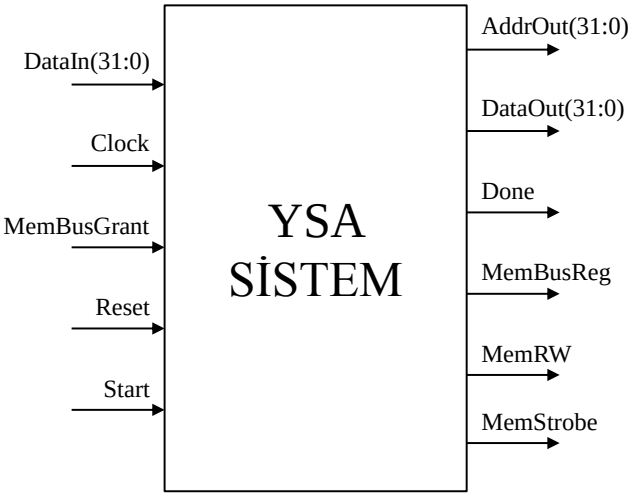
2.7. YAPAY SİNİR AĞLARININ OTOMATİK OLARAK FPGA ÇİPİNE UYGULANMASI İÇİN DENETLEYİCİ TASARIM ARACI

Bu çalışmada, YSA sistemlerinin otomatik olarak tasarlanıp FPGA çipine uygulanmasına yardımcı bir EDA (Elektronic Design Automation (Elektronik Tasarım Otomasyonu)) aracı olan ANNCONT geliştirilmiştir. ANNCONT daha öncesinde Sarıtekin tarafından geliştirilmiş olan ANNGEN'in devamı niteliğindedir. ANNCONT, ANNGEN kullanılarak otomatik bir şekilde oluşturulmuş YSA veri yolları için yine otomatik olarak denetleyici tasarımını gerçekleştirmektedir. ANNCONT tasarımı ile birlikte ayrıca veri yolu (ANNGEN) ve denetleyici (ANNCONT) birleştirilerek çalışmaya hazır bir YSA sistemini otomatik olarak gerçekleyen bir tasarım aracı olarak ANNSYS geliştirilmiştir.

Bu bölümde öncelikle ANNGEN, ANNCONT ve ANNSYS kullanılarak otomatik bir biçimde tasarımı yapılan YSA'ların yapısı sunulacaktır. Ardından ANNCONT tüm detaylarıyla anlatılacaktır.

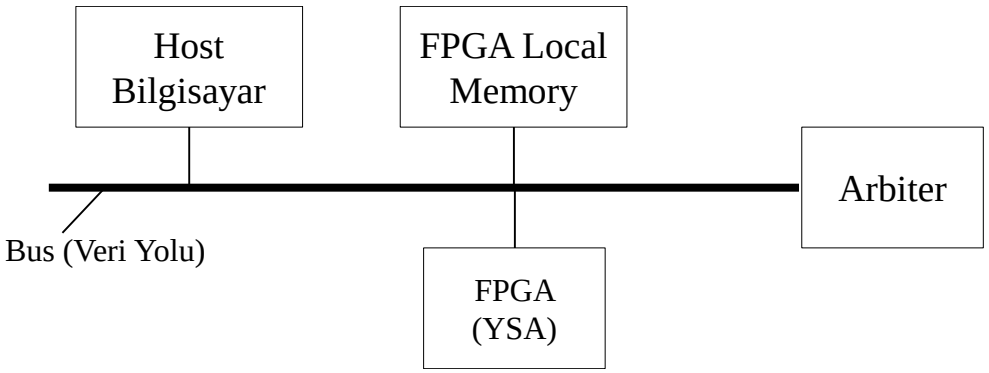
2.7.1. Otomatik Olarak Tasarlanan YSA Sistemin Yapısı

Şekil 2.25'te ANNSYS tarafından otomatik olarak tasarlanan bir YSA sisteminin en üst seviye blok diyagramı görülmektedir. 32-bitlik *DataIn* sinyali iki amaç için kullanılmaktadır. Birincisinde YSA çalışmaya başlamadan önce bir birine seri bağlı ağırlık (*weight*) ve eşik (*bias*) değerlerinin tutulduğu registerlerin ilk değerlerinin yüklenmesinde. İkinci olarak ise YSA'nın normal çalışma durumunda YSA girişlerinin seri bir şekilde girilmesinde kullanılmaktadır. 32-bitlik *DataOut* sinyali ile YSA sisteminin hesapladığı sonuç değeri çıkışa iletilmektedir.



Şekil 2.25. YSA Sistemi En Üst Seviye Blok Diyagramı.

MemBusGrant, MemBusReg, MemRW, MemStrobe, sinyalleri YSA sisteminin hafıza erişimini senkronize etmede kullanılan giriş/çıkış sinyalleridir. *AddrOut* sinyali YSA sistemi içinde denetleyicinin bir parçası olan adres ünitesi tarafından oluşturulan adres bilgisinin hafızaya iletilmesinde kullanılmaktadır. Sistemin girdilerini hafızadan okuyarak sonuçlarını yine hafızaya yazacak şekilde tasarlanmaktadır. *Clock, Start, Reset* ve *Done* sinyalleri ise YSA sisteminin bir host bilgisayar tarafından kontrol edilmesini sağlayan sinyallerdir. YSA Host ilişkisi Şekil 2.26'da gösterildiği gibidir.

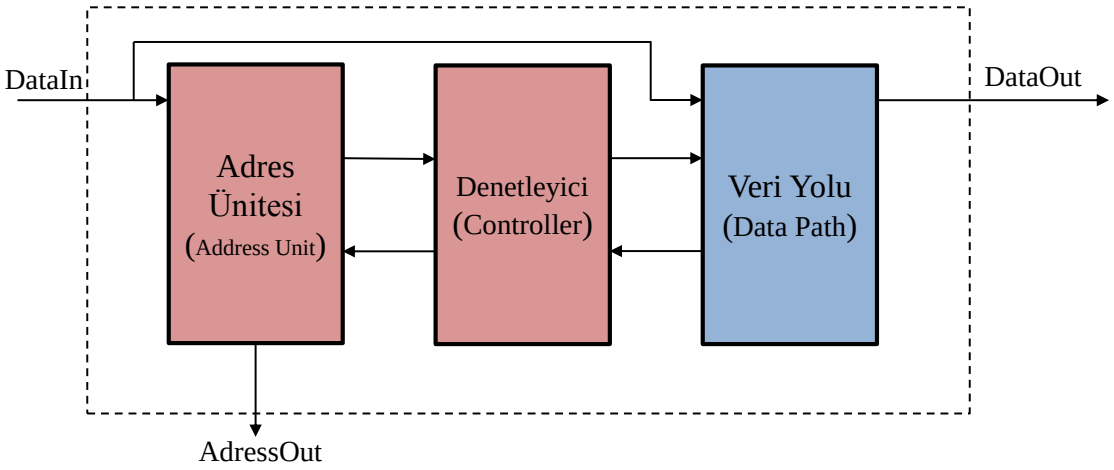


Şekil 2.26. YSA Host Hafıza Erişim İlişkisi.

Host bilgisayar FPGA kartı üzerinde yüklenmiş YSA sistemine bir *Bus* aracılığıyla erişir. YSA sistemi kullanılmadan önce host bilgisayar gerekli parametreleri ve işlenecek veriyi kendi hafızasından FGPA kartının yerel hafızasına aktarır. Bu aşamadan sonra FPGA'de yüklü YSA sistemine de bir başla sinyali gönderilerek veri işlemesi sağlanır.

YSA hafızaya erişmek için *MemBusRequest* ile *Bus*'ı *Arbiter*'den ister. *Arbiter MemBusGrant* ile *Bus'*ı YSA'ya verir. YSA hafızadan önce konfigürasyon bilgilerini (*Weight, Bias*), giriş veri alanı başlangıç adresini, çıkış veri alanı başlangıç adresini ve giriş veri boyutunu okur. Daha sonra YSA giriş verilerini sıra ile hafızadan okur işler ve sonuçlarını hafızaya yazar. Bütün veriler işlenince YSA *Done* (*Bitti*) sinyalini Host bilgisayara gönderir. Host bilgisayar YSA'yı resetler ve sonuçları hafızadan alır.

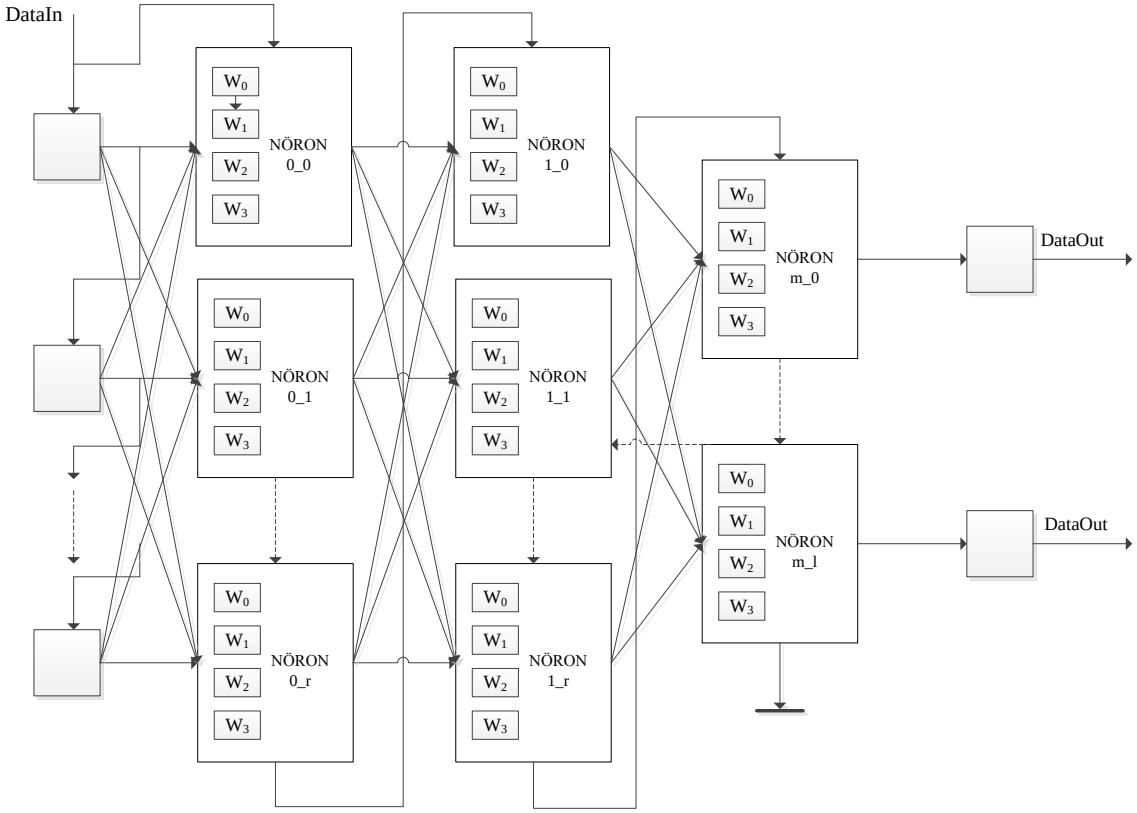
Şekil 2.27'de otomatik olarak tasarlanan YSA'ların genel ikinci seviye blok diyagramı görülmektedir. Sistem, *AdressUnit, DataPath* ve *Controller* olmak üzere üç bölümden oluşmaktadır. *DataPath*, Sarıtekin tarafından geliştirilen ANNGEN aracılığıyla oluşturulmuş YSA veri yoludur (Sarıtekin, 2011).



Şekil 2.27. İkinci Seviye Blok Diyagramı.

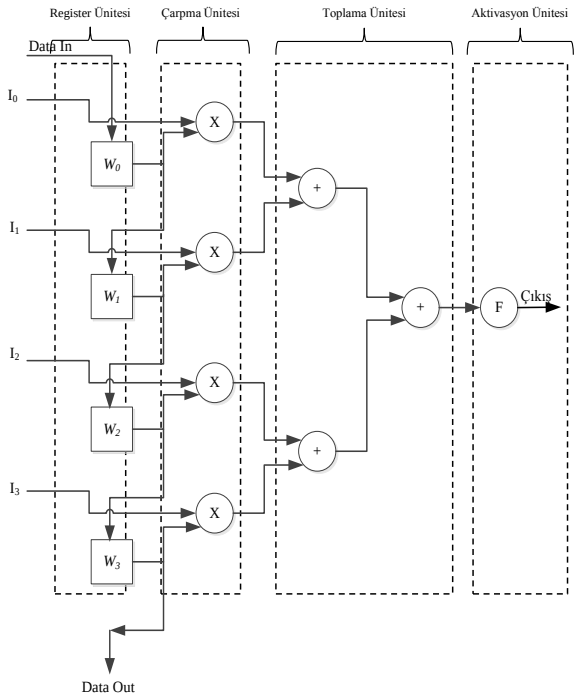
2.7.2. Otomatik Olarak Oluşturulan YSA Veri Yolunun Yapısı

Şekil 2.28'de ANNGEN tarafından yapay sinir hücrelerinin otomatik olarak birbirine bağlanması ile oluşturulmuş YSA veri yolunun genel yapısı görülmektedir. Giriş katmanından gelen verilerin geçici olarak saklanması için girişlere *generic registerler* yerleştirilmiştir. Hücreler içindeki *weight* (ağırlıklar) ve eğer varsa *bias* (eşik) değerlerini tutan registerlere ilgili değerleri yükleyebilmek için tek bir kanaldan gelen *DataIn* sinyali ilk katmanın ilk hücresinin *data* girişine bağlanmış, bu hücreden çıkan *DataOut* sinyali bir sonraki hücrenin *DataIn* girişine bağlanarak bir zincir oluşturulmuştur. Bu zincirler sayesinde veri yolunun giriş sayısı azaltılmıştır (Sarıtekin, 2011).



Şekil 2.28. FPGA'da Dört Girişli Normal Yapay Sinir Modeli (Sarıtekin, 2011).

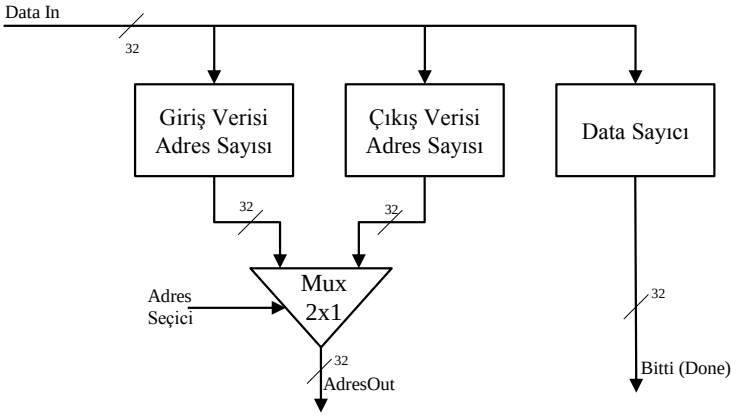
Şekil 2.29'da ise dört girişli normal bir yapay sinir hücresinin içyapısı görülmektedir. Eşik'li hücrelerde eşik değerinin tutulduğu registerde ağırlık registerleri ile aynı zincire eklenir. Eşik değerini işleme katmak için toplama ünitesinde ayrıca toplayıcılar yerleştirilmiştir. Hücre içindeki bütün çarpma, toplama ve transfer fonksiyon üniteleri kanallı (pipelined) olarak tasarlanmıştır. Hücre veri işlemek üzere etkin edildiğinde bir önceki katmandan gelen giriş değerlerinin alır, bunları weight değerleri ile çarparak toplamlarını bulur ve son olarak bulunan değeri transfer fonksiyonundan geçirerek çıkışı üretir (Sarıtekin, 2011).



Şekil 2.29. FPGA'da Dört Girişli Normal YSA Modeli Hücre İç Yapısı (Sarıtekin, 2011).

2.7.3. Adres Ünitesi

Adres (*AdressUnit*) ünitesi, okuma yazma verilerinin hafızanın hangi adreslerinde olduğunun bilgisini tutan adres sayaçlarından oluşur. Bu ünite hafıza erişiminde denetleyici tarafından kullanılır. Şekil 2.30'da görüldüğü gibi Giriş Verisi Adres Sayıcıdan ve Çıkış Verisi Adres Sayıcıdan gelen adres bilgileri, adres seçici multiplexer tarafından seçilerek *AdresOut* olarak çıkışa aktarılır. Data Sayıcı *Bitti(Done)* sinyali ile *Controller*'a işlemin bittiği bilgisini iletir.

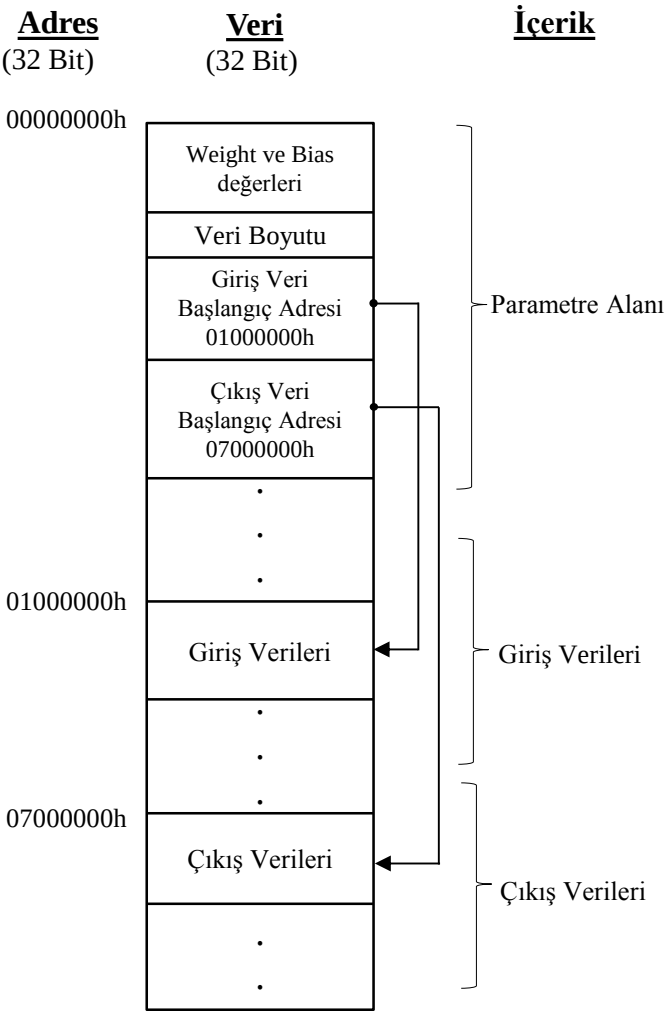


Şekil 2.30. Adres Ünitesi Blok Diyagramı.

2.7.4. Hafıza Haritası

Şekil 2.31'de tasarlanan ünitenin kullandığı hafıza haritası görülmektedir. Hafıza haritası üç bölümden oluşmaktadır. Bu bölümler, Parametre alanı, Giriş Verileri alanı, Çıkış Verileri alanıdır.

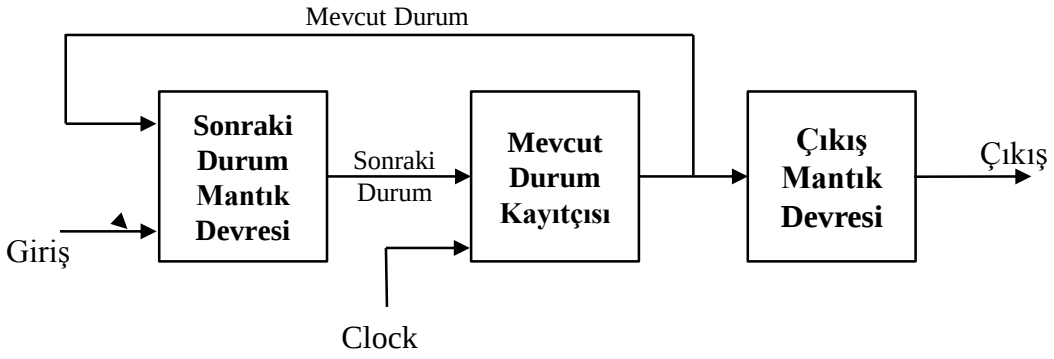
İlk bölümde ağın yazılım versiyonu ile eğitilmesinden elde edilen weight ve bias değerleri tutulmaktadır. YSA sistemi ilk olarak bu değerleri okur ve hücreler içindeki weight ve bias yazaçlarını kurar. İkinci bölümde ise giriş, çıkış verilerinin başlangıç adres değerleri ve işlenecek veri boyutu tutulur. YSA sistemi ikinci aşamada bu değerleri okuyarak adres ünitesinde bulunan adres yazaçlarını kurar. Son bölümde ise giriş ve çıkış veri alanları bulunur. Bu alanlar çakışmamak kaydıyla hafızada herhangi bir yerden başlayabilir ve istenen biri diğerinde önce gelebilir.



Şekil 2.31. Hafıza Haritası.

2.7.5. ANNCONT İle Otomatik Olarak Oluşturulan YSA Denetleyicisi

Denetleyici (*Controller*) ünitesi; YSA sisteminin tüm işleyişini denetleyen ve hafıza erişimini sağlayan ünitedir. Şekil 2.32'de blok diyagramı görülen ve otomatik olarak oluşturulan denetleyici bir Sonlu Durum Makinesi (Finite State Machine) modeli olan moore makinesi modelinde çalışmaktadır.



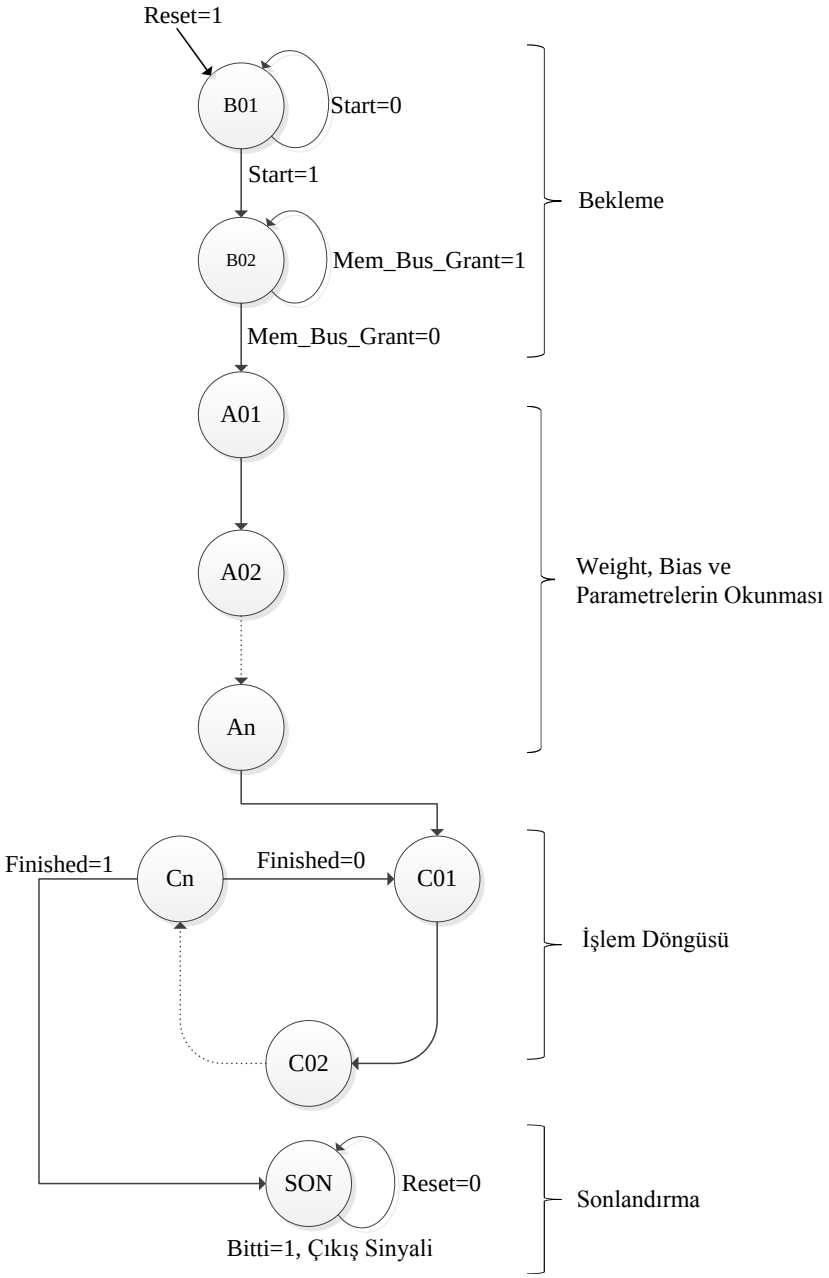
Şekil 2.32. Üçüncü Seviye Denetleyici Ünitesi Blok Diyagramı.

Şekil 2.33'de denetleyicinin genel durum diyagramı görülmektedir. Durum diyagramı toplam dört bölümden oluşmaktadır. Birinci bölüm olan bekleme bölümünde denetleyici host bilgisayardan gelecek başla sinyalini ve bu sinyal geldikten sonra *Bus Arbiter*'dan gelecek *MemBusGrant* sinyallerini bekler. Bu bölümün yapısı sabittir ve iki durumdan oluşur. Birinci durumda olan *B01* durumu, host bilgisayardan gelecek *start* sinyalini bekleyecek şekilde tasarlanmıştır. Beklenen bu sinyal *1* olduğunda makina *B02* duruma geçer. *B02* durumu *Bus Arbiter*'a *MemBusRequest* sinyali gönderecek ve *Arbiter*'den gelecek *MemBusGrant* sinyalini bekleyecek şekilde oluşturulmuştur. *MembusGrant* sinyali geldiğinde ise makina ikinci bölüme geçer.

İkinci bölümün görevi, hafızadan ağın çalışmaya başlamadan önce okunması gereken weight, bias ve veri parametrelerinin hücre içinde bulunan ilgili yazaçlara okunmasını sağlamak, ayrıca adres ve veri boyutu sayaçlarına ilk değerleri okumaktır. İkinci bölümde esnek bir yapıya sahip olup buradaki durumların adedi kontrol edilen YSA veri yolunun giriş çıkış sayısına ve ağda kullanılan sinir hücrelerinin yapısına bağlı olarak değişmektedir. Değişikliğin nedeni ağın yapısına bağlı olarak okunacak veri miktarının (weight ve bias adedinin) farklı farklı olmasıdır. Bu bölüme yerleştirilen durumlar şartsız bir şekilde bir durumdan diğerine geçecek şekilde tasarlanırlar.

Üçüncü bölüme gelindiğinde bütün parametreler okunmuş ve YSA sistemi giriş verilerini işlemek üzere hazır hale gelmiş durumdadır. Bu bölümde verilen *n* büyüklüğünde veriyi işlemek üzere denetleyicide durumlardan (state) oluşan bir döngü meydana getirilir. Döngü, her tekrarlanışında şu işlemler yapılır. Öncelikle bir giriş veri grubu hafızadan okunur. Bu okunan veriler YSA veri yoluna işlenmek üzere verilir, ardından eğer YSA çıkışında bir çıkış varsa (sonuç varsa) bu çıkış değeri hafızaya yazılır. YSA veri yolu kanallı (pipelined) olarak tasarlandığından kanal derinliğine bağlı olarak değişmekle beraber giriş verilerinin sonuca ulaşması belli bir süreç almaktadır. Kanal yapısı içinde registerler kullanıldığı için YSA veri yolundan aynı anda birden fazla giriş verisi değişik aşamalarda işlenerek (tıpkı yürüyen merdivenlerde olduğu gibi) YSA çıkışına doğru hareket eder. Dolayısıyla denetleyicinin bu bölümünde oluşturulan döngü, sırasıyla üç değişik modda çalışır. Birinci modda hafızadan sürekli olarak giriş verilerini okur ve YSA veri yoluna girdi olarak iletir. Bu modda veriler YSA veri yolunda işlenerek adım adım çıkışa doğru ilerler. İlk çıkış üretildiği andan itibaren döngü ikinci modda çalışamaya başlar. Bu modda bir yandan giriş verileri hafızadan okunurken diğer yandan da oluşan sonuçlar hafızada uygun adrese yazılır. Giriş verilerinin tamamının okunması bittiğinde döngü üçüncü moda girer. Bu modda artık okuma işlemi yapılmaz. YSA veri yolundaki bütün veriler işlenip sonuca ulaşana kadar (kanal boşaltılana kadar) döngü tekrar eder ve oluşan çıkış değerlerini hafızaya yazar. Son çıkış ta hafızaya yazıldıktan sonra döngüden çıkılarak son bölüme geçilir.

Döngüden çıkıldıktan sonra işlemin bittiğini hosta bildirmek ve host bilgisayara bir kesme sinyali gönderebilmek üzere denetleyicinin dördüncü bölümünü oluşturan *son state* oluşturulur. Denetleyici bu *son state'te reset* sinyali gelene kadar bekleyecek şekilde tasarlanır. *Reset* sinyali geldiğinde ise bütün işlemi baştan başlatmak üzere birinci bölümün ilk *state*'i olan *B01* durumuna geçilecek şekilde tasarım yapılır.



Şekil 2.33. YSA Sistem Kontrolü Durum Diyagramı.

2.7.6. Otomatik Oluşturulan Denetleyicideki Durum Sayısının Belirlenmesi

Yukarıda belirtildiği gibi ANNCONT tarafından otomatik oluşturulan denetleyici toplam dört bölümden meydana gelmektedir. Birinci bölüme toplam iki durum yerleştirilmektedir ve bu bölümün yapısı sabittir. İkinci bölüme yerleştirilen durum sayısı denetlenecek YSA'nın yapısına göre değişmektedir. Bu bölümün durum sayısını, YSA yapısında kullanılan toplam ağırlık adedi artı her bir YSA için sabit olan giriş verisi başlangıç adresi, çıkış verisi başlangıç adresi ve veri boyutu parametrelerini okumak için gerekli durumlar belirler.

Üçüncü bölüm olan döngüye yerleştirilen durum sayısı yine YSA'nın yapısına göre (giriş ve çıkış adedine göre) değişmektedir. *i* adet girişli *k* adet çıkışlı bir veri yolu için döngüye *i+k+2* adet durum yerleştirilir. Burada *i+2* adet durum her seferinde bir giriş veri setini hafızadan okumak için, *k* adet durum ise sonuçta üretilen *k* adet çıkışı hafızaya yazmak için kullanılır. Döngünün çalışması sırasında birinci modda (sadece veri okumasının yapıldığı mod) *k* yazma durumlarında sistem herhangi bir işlem yapmadan boş geçer. İkinci modda döngünün bütün durumlarında işlem yapılır. Üçüncü modda ise sadece *k* durumlarında hafızaya yazma işlemi yapılır, *i+2* durumlarında işlem yapılmadan boş geçilir. Son bölümde ise kesme sinyalini gönderebilmek üzere tek bir durum yerleştirilir.

i adet girişli, *k* adet çıkışlı ve içinde *wb* adet ağırlık ve bias içeren bir YSA veri yolunu kontrol etmek için ANNCONT tarafından oluşturulan denetleyicinin içereceği toplam durum sayısı eşitlik 2.8'de verilmiştir. Toplam durum sayısı (DS);

$$DS = \sum_{i=1}^{4} B\ddot{o}l_i DS \tag{2.8}$$

Birinci bölüm ve son bölümün yapısı sabittir ve içerdikleri durum sayıları sırasıyla *2* ve *1*'dir. İkinci bölümün durum sayısı;

$$B\ddot{o}l_2 DS = wb + parametre\ adedi + 2 \tag{2.9}$$

Burada sabit olarak *3* parametre okunmaktadır. Dolayısıyla;

$$B\ddot{o}l_2 DS = wb + 3 + 2 \tag{2.10}$$

Üçüncü bölümün durum sayısı *i+k+2* dir. Burada *i* giriş *k* çıkış sayısıdır. Sonuç olarak bölümlerin durum sayılarına göre eşitlik 2.8'i yeniden düzenlersek;

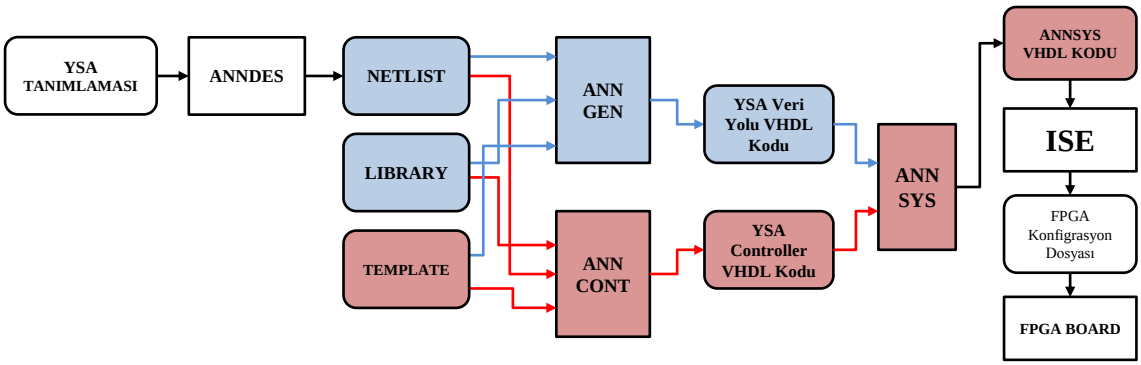
$$DS = \sum_{i=1}^{4} B\ddot{o}l_i DS = wb + i + k + 10 \tag{2.11}$$

Olarak belirlenir. İkinci ve üçüncü bölümün durum sayılarındaki *+2* değerleri hafızadan okuma işlemindeki iki saat darbelik gecikmeye karşılık olarak kullanılmıştır.

2.8. ANNCONT (OTOMATİK YSA DENETLEYİCİSİ TASARIM ARACI)

ANNCONT, verilen metin tabanlı Neural Network tanımlamasına göre oluşturulmuş bir YSA veri yolunu denetleyecek denetleyici tasarımını otomatik bir şekilde yaparak bunu VHDL kodu şeklinde sunan bir yazılım aracıdır. Şekil 2.34'te otomatik YSA sistem tasarım akışı içinde ANNCONT, ANNGEN ve ANNSYS in yerleri görülmektedir.

ANNCONT girdi olarak üç farklı dosya kullanmaktadır. Bunlar metin tabanlı YSA tanımlaması (*NetList*), modül kütüphanesi (*Library*) ve şablon (*Template*) dosyalarıdır.



Şekil 2.34. FPGA Çipine Yerleştirilecek Tasarım.

2.8.1. ANNCONT'un Girdileri

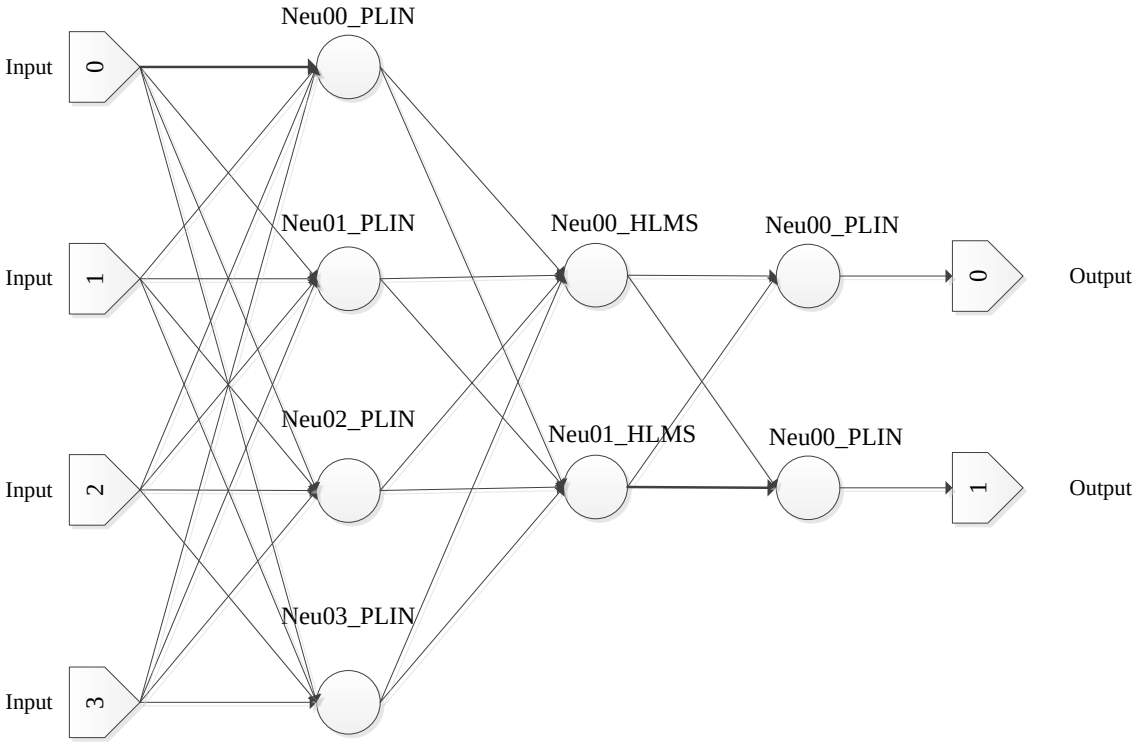
2.8.1.1. NetList

NetList'ler genellikle bir elektronik devrenin yapısını tanımlamada kullanılan veri formatıdır. Çok çeşitli devre türlerini ifade edebilmek için endüstride kullanılan bazı NetList formatları *EDIF* (Electronic Data Interchange Format), *XNF* (Xilinx Netlist Format)'tir. Daha önceden yapılan ANGENN çalışmasında, yapay sinir ağlarını daha kolay bir şekilde tanımlayabilmek için yeni bir *NetList* formatı geliştirilmiştir. Bu çalışmada da aynı *NetList* formatı kullanılmıştır. Şekil 2-35'te, *NetList* dosyasının genel formatı görülmektedir. *NetList* iç içe bloklardan oluşur. Blok başlangıç ve bitişleri özel karakterler ('[' ve ']') ile sembolize edilirler. Her bir blok içerisinde ya bir katman ya da gerekli parametreler tanımlanır. *LAYER* ve *PAREMETERS* kelimeleri blok türünü belirler. *LAYER* kelimesinden sonra gelen *INPUT*/*NEURON*/*OUTPUT* kelimeleri o katmanın giriş, gizli ya da çıkış katmanı olduğunu belirler. Her bir katmanın yapısı diğerinden farklıdır. Giriş ve çıkış katmanları içinde sadece girişler ve çıkışlar ismen tanımlanır. Nöron katmanında ise o katmanda bulunan nöronların türü, girişleri,

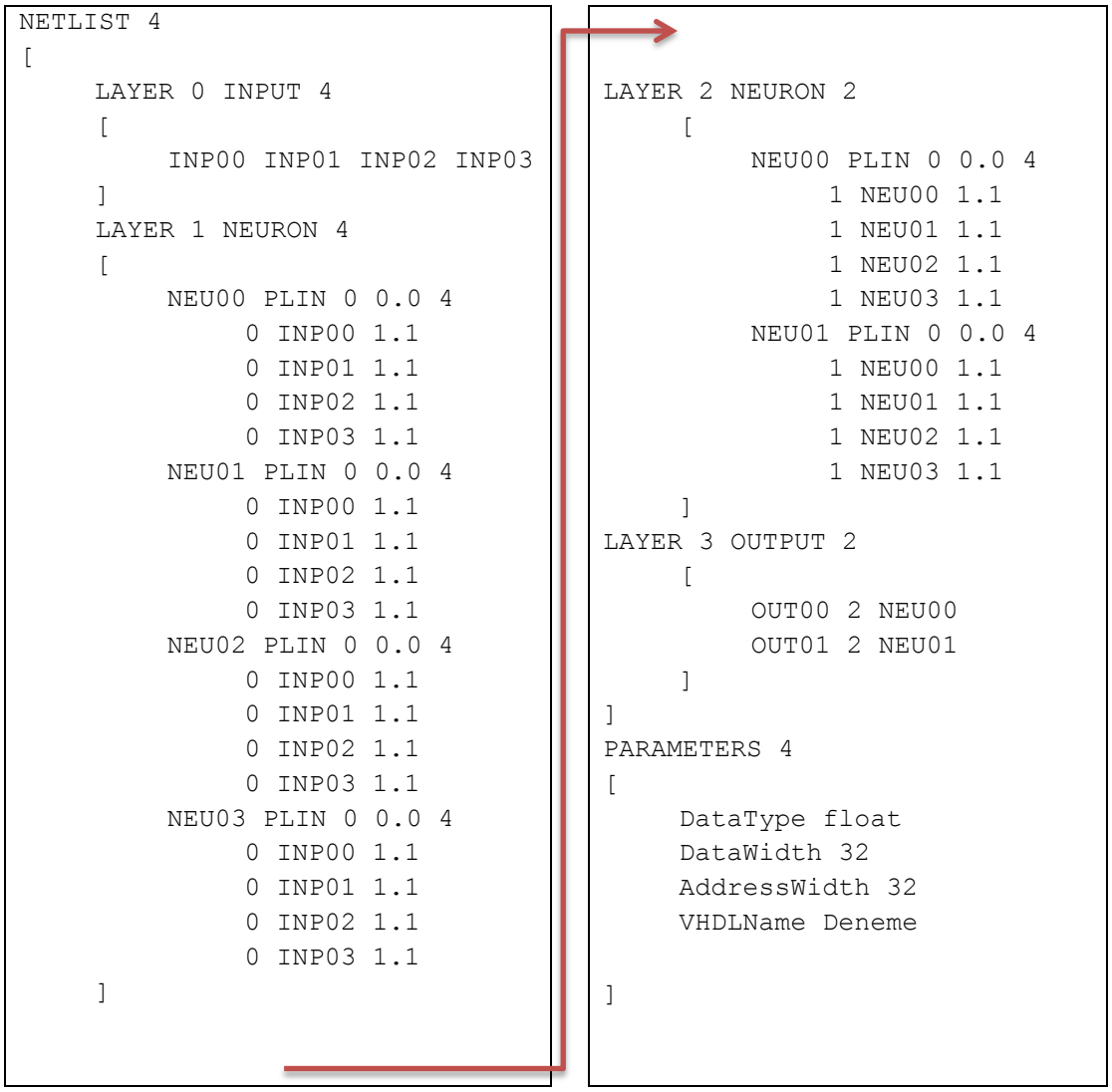
girişlerinin nereden yapıldığı ve ağırlık değerleri ile birlikte tanımlanabilir. Şekil.2.36'da görünen ağı tanımlamak üzere oluşturulmuş *NetList* dosyası Şekil.2.37'de verilmiştir.

```
NETLIST <Katman Sayısı>
[
        LAYER <KatmanNo> <KatmanTürü> <Katmandaki Element Sayısı>
        [
                <Giris0> <Giris1> … <Girisn>  (Giriş Katmanı)
        ]
        LAYER 1 NEURON 4                      (Gizli Katmanı)
        [
                <NeuronİSim> <TransferTürü> <Esik> <EsikDegeri> <GirişSayısı>
                        <GirişAlınan KatmanNo> <Neuron Ismi> <Ağırlık>
                        <GirişAlınan KatmanNo> <Neuron Ismi> <Ağırlık>
                        <GirişAlınan KatmanNo> <Neuron Ismi> <Ağırlık>
                        :
        ]
        :
        PARAMETERS <Parametre Sayisi>
        [
                :
        ]
]
```

Şekil 2.35. *NetList* Formatı (Sarıtekin, 2011).



Şekil 2.36. Örnek Yapay Sinir Ağı



Şekil 2.37. Örnek *NetList* Tanımlaması

2.8.1.2. Kütüphane (Library)

Bu dosyada, hali hazırda tanımlaması yapılmış ANNGEN ve ANNCONT tarafından kullanılabilecek yapay sinir hücrelerinin listesi ve bu hücrelerle ilgili bazı ek bilgiler tutulur. Bu dosya verilen bir *NetList*'in içinde bulunan bütün hücrelerin sistemde tanımlanması amacıyla kullanılır. Yapay sinir ağı oluşturulmadan önce ağdaki bütün hücre türlerinin bu dosyada listelenip listelenmediği kontrol edilir. Eğer herhangi bir hücre bu dosyada yer almıyorsa ağ üretilmez. Dosya çok basit bir formata sahiptir. Çizelge 2.1'de görüldüğü gibi bu dosyada hücrelerin isimleri, eşik durumları ve giriş adetleri gösterilmiştir.

Sinir Hücresinin (Nöron) İsmi	Eşik (Bias) Durumu	Giriş Adedi
LOGS	0	4
LOGS	1	4
TANS	0	2
TANS	1	2

Çizelge 2.1. Kütüphane (*Library*) Dosyası Formatı.

2.8.1.3. Şablon (Template)

Bu dosyada, ANNGEN ve ANNCONT'un yeni VHDL kodu yazımında kullandığı çeşitli şablonları (VHDL kod parçaları) ve ayrıca kütüphanede tanımlanmış hücrelerin VHDL kodları yer almaktadır. Bu bilgiler şablon dosyasına sistematik bir şekilde yerleştirilmiştir. Şablon dosyasındaki ilk bilgi dosyada kaç tane şablonun olduğudur. Ardından sırası ile şablonlar yerleştirilmiştir. Şablon okuyucu bu bilgiye göre şablonları okur. Her bir şablonun formatı ise şablonun ismi, satır sayısı ve şablonun içeriği şeklindedir. Şekil 2.38'de şablon dosyası formatı görülmektedir.

```
<Şablon Adedi>
<Şablon İsmi> <Satır Sayısı>
library IEEE;
use IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
entity COUNTER is
:
end entity COUNTER;
architecture RTL of COUNTER is
:
end architecture RTL;
<Şablon İsmi> <Satır Sayısı>
:
```

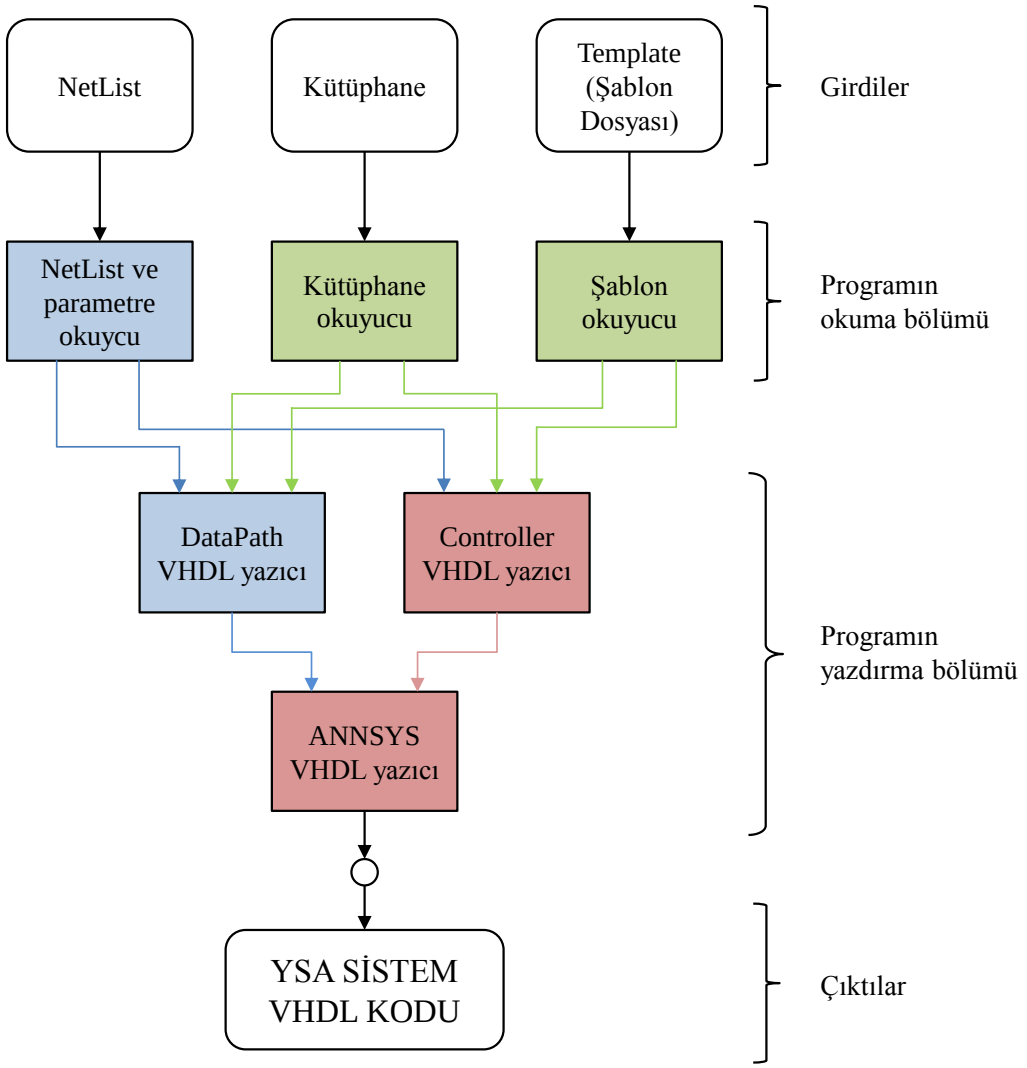
Şekil 2.38. Şablon (*Template*) Formatı.

2.8.2. ANNCONT'un Bileşenleri

Şekil 2.39'da otomatik YSA sistemi tasarım aracı içinde ANNCONT'un yeri görülmektedir. *NetList Okuyucu, Kütüphane Okuyucu ve Şablon Okuyucu* bölümleri hem ANNGEN ve hem de ANNCONT tarafından ortak olarak kullanılan bölümlerdir. Bu bölümler ortak kullanıldığı için ANNCONTun bileşeni olarak kabul edilmişlerdir ve burada açıklanmışlardır.

Şekildeki renklendirmeler şu şekildedir. Daha önceden de belirttiğimiz üzere ANNCONT tasarımımız ANNGEN (Sarıtekin, 2011) tasarımının devamı olarak

tasarlanmıştır. Şekilde verilen mavi dolgulu alanlar ANNGEN tasarımından alınarak ortak kullanılanlar, yeşil dolgulu alanlar tasarımın içerisine eklemeler yapılarak geliştirilmiş alanlar ve kırmızı renkli alanlar ise bu çalışma kapsamında yeni geliştirilen alanlardır.



Şekil 2.39. ANNCONT'un Genel Yapısı (Sarıtekinin ANNGEN'i İle Birlikte).

2.8.3. ANNCONT'un Çalışması

ANNCONT ANNGEN ve ANNSYS ile entegre bir şekilde çalışmaktadır. Dolayısıyla bu bölümde ANNCONT'un çalışmasını açıklamadan önce otomatik YSA tasarım aracının temel algoritması, ardından daha öncesinde Sarıtekin'in çalışmasında geliştirilen ve ortak olarak kullanılan giriş dosyaları okuyucu algoritmaları ve son olarak bu çalışmada geliştirilen ANNCONT ve bileşenlerinin algoritmaları detaylı bir şekilde sunulmuştur.

Otomatik YSA tasarım aracının temel algoritması Şekil 2.40'a verilmiştir. Öncelikle üç okuyucu modül aracılığıyla verilen *NetList*, *Library* ve *Template* dosyaları okunarak ilgili veri yapısına kaydedilir. Ardından *CheckModulResult()* fonksiyonu ile *NetList'*te belirtilen bütün hücre türlerinin hali hazırda tanımlamasının olup olmadığı kontrol edilir. Eğer sonuç pozitifse istenen YSA için önce Sarıtekin tarafından geliştirilen *WriteVHDL()* (ANNGEN) fonksiyonu ile veri yolu oluşturulur. İkinci adımda bu çalışmada geliştirilen *WriteController()* (ANNCONT) aracılığıyla denetleyici otomatik olarak oluşturulur. Son adımda *WriteANNSYS()* (ANNSYS) fonksiyonu ile oluşturulan veri yolu ve denetleyici üniteleri uygun bir adres ünitesi ile birleştirilerek istenen YSA sistemi oluşturulur.

```
Void Main()
    Begin
        Net=ReadNetList("NetList.Txt")
        Lib=ReadLibrary("Library.Txt")
        T=ReadTemplate("Template.Txt")
        if (CheckModulResult(Net,Lib,T))
            WriteVHDL(VHDLName,Net,Lib,T)
            WriteContreller(ContName,Net,Lib,T)
            WriteANNSYS(ANNSYSName,Net,Lib,T)
        else
            msg("Nöron Library'de bulunamadı")
    End
```

Şekil 2.40. Main Fonksiyonu Algoritması.

2.8.3.1. ReadNetlist(): NetList Dosyası Okuma Fonksiyonu

Bu fonksiyonun görevi değişken bir yapıya sahip olan *NetList* dosyasını uygun bir şekilde okuyup analiz ederek daha sonra kullanılmak üzere ilgili veri yapısında saklamaktır. Şekil 2.41'de, *ReadNetList()* fonksiyonunun algoritması görülmektedir.

Algoritma, verilen *NetList* dosyasını kelime kelime okur, okuma işlemi sırasında gerekli veri yapısını oluşturur ve okunan bilgileri oluşturulan veri yapısında saklar. Sonuçta oluşturulan veri yapısı tek bir isim altında ana fonksiyona geri döndürülür. *NetList* dosyası değişken bir yapıya sahip olduğundan fonksiyon veri yapısını oluşturmada pointerler ve dizileri kullanır.

```
NetListType * ReadNetList(char * FileName)
Begin
    NetListType *NL;
    NL ← New NetListType;
    Fpt ← OpenFile (FileName);
    NL.LayerCount ←Fpt;
    NL.Layers ← New LayerType (NL.LayerCount)
    For  (each Layers (i) in the NetList) Do
        NL.Layers(i) = ReadLayer(Fpt)
    End For
    ParamCount ← Fpt
    For (each Parameter (i) in the NetList) Do
        Word←Fpt;
        If (Word = "DataType") Then
            Word ←Fpt
            If (Word= "integer") Then
                NL.DataType = 0   #   0 ==> integer
            Else
                NL.DataType = 1   #   1 ==> float
            End If
        End If
        If (Word = "DataWidth") Then
            NL.DataWidth←Fpt
        End If
        If (Word = "AddressWidth") Then
            NL.AddressWidth←Fpt
        End If
        If (Word = "VHDLName") Then
            NL.VHDLName←Fpt
        End If
    End For
    Fpt←CloseFile()
    Return NL
End
```

Şekil 2.41. *NetList*'i Okuma Fonksiyonu Algoritması (Sarıtekin, 2011).

2.8.3.2. ReadLibrary(): Kütüphane Dosyası Okuma Fonksiyonu

Şekil 2.42'de, kütüphane fonksiyonunun algoritması görülmektedir. Bu fonksiyon *ReadNetList()* fonksiyonuna benzer şekilde *Library* fonksiyonunu analiz ederek okur ve ilgili veri yapısını oluşturarak okunan bilgileri saklar. Fonksiyon sonlanırken okunan kütüphane bilgisi tek bir isim altında geri döndürülür. Yine burada da *Library* dosyasının yapısı da esnek bir yapıya sahip olduğundan bu fonksiyonda *pointer* ve dizileri kullanır.

```
LibraryType *ReadLibrary(char * FileName, int &EC)
Begin
    LibraryType *LB
    LB ← New LibraryType (100)
    Fpt← OpenFile(FileName)
    While (!Fpt.eof()) Do
        Word← Fpt
        LB[EC].Type = StrToNeuron(Word)
        Tmp← Fpt
        LB(EC).Bias  = Tmp
        LB(EC).InputCount← Fpt
        If (EC=100)Then
            Break
        End If
    End While
    Fpt← CloseFile()
    Return LB
End
```

Şekil 2.42. Kütüphane Okuma Fonksiyonu Algoritması (Sarıtekin, 2011).

2.8.3.3. ReadTemplate(): Şablon Dosyası Okuma Fonksiyonu

Şekil 2.43'te *ReadTemplate()* fonksiyonunun algoritması görülmektedir. Bu fonksiyonda diğer iki okuyucu fonksiyona benzer şekilde çalışır. Verilen şablon dosyasını okur, analiz eder ve ilgili veri yapısında depolayarak geri döndürür.

```
TempType *ReadTemplate(char * FileName)
Begin
    TempType *T
    T ← New TempType
    Fpt← OpenFile(FileName)
    T.TempCount← Fpt
    T.Temps ← New TempOneType(T.TempCount)
    For (each element of Temp)
        T.Temps(i).TempName←Fpt
        T.Temps(i).LineCount←Fpt
        T.Temps(i).Written←false
        T.Temps(i).Lines←New LineType (T.Temps(i).LineCount)
        ch ← Fpt
        For (each element of T.Temps(i).Line)
            ch ← Fpt
            While (ch != endl)Do
                T.Temps(i).Lines(k).Line(j)=ch
                ch ← Fpt
            End While
            T.Temps(i).Lines(k).Line(j) = 0
        End For
    End For
    Fpt ←CloseFile()
    Return T
End
```

Şekil 2.43. Şablon Okuma Fonksiyonu Algoritması (Sarıtekin, 2011).

2.8.3.4. CheckModül(): Hücre (Nöron) Kontrol Fonksiyonu

İstenilen YSA için veri yolu ve denetleyiciyi oluşturmadan önce verilen *NetList*'teki hücrelerin kütüphane dosyasında var olup olmadığı kontrol edilmesi gerekmektedir. Çünkü eğer *NetList*'te belirtilen hücrelerden herhangi biri kütüphanede mevcut değilse ağ oluşturulamayacaktır. Bu durumu kontrol etmek için *CheckModül()* fonksiyonu yazılmıştır. Fonksiyon ağda belirtilmiş bütün hücrelerin tanımlamasının kütüphanede olup olmadığını kontrol eder. Eğer bütün hücrelerin tanımlaması kütüphanede varsa DOĞRU (True), diğer durumlarda YANLIŞ (False) değerini gönderir. Fonksiyonun algoritması Şekil 2.44'te verilmiştir.

```
Bool CheckModules (NetListType *N, LibraryType *L, int EC)
Begin
    int i,j
    Bool Result = True
    For (i=0; i<N->LayerCount; i++)
        If (N->Layers[i].Kind == Neuron)
            For (j=0; j<N->Layers[i].ElementCount; j++)
                Result = Result & CheckOneNeuron
                (L, &N->Layers[i].Elements[j],EC)
            End For
        End If
    End For
    Return Result
End
```

Şekil 2.44. Hücre (Nöron) Kontrol Fonksiyonu Algoritması (Sarıtekin, 2011).

2.8.3.5. WriteController(): ANNCONT'un Temel Algoritması

Şekil 2.45'te ANNCONT'un temel çalışma algoritması görülmektedir. Algoritma öncelikle oluşturulacak denetleyici ünitesi ile ilgili açıklama satırlarını çıkış dosyasına yazarak başlar. Ardından *WriteControllerENTITY()* ve *WriteControllerARCHITECTURE()* fonksiyonlarını çağırarak hedef denetleyici ünitesi VHDL kodunun *entity* ve *architecture* bölümlerini iki ayrı aşamada oluşturur. Bu kontrol işlemi hem ANNGENN hem ANNCONT için ortak işlemdir. Eğer bütün hücrelerin tanımlaması varsa ANNCONT, denetleyici ünite (Contreller) için VHDL kodunu üretir.

```
Void WriteController (char * FileName, NetListType *N, LibraryType *L, TempType *T)
  Begin
    Ftpn <- OpenFile(FileName)
    WriteOneTemp(ofn,T,7)
    WriteOneTemp(ofn,T,0)
    time (&rawtime)
    timeinfo = localtime (&rawtime )
    WriteOneTemp(ofn,T,1)
    WriteOneTemp(ofn,T,2)
    WriteOneTemp(ofn,T,3)
    WriteControllerENTITY (ofn,FileName,N)
    WriteControllerARCHITECTURE(FileName,ofn, N, L, T)
  End
```

Şekil 2.45. ANNCONT Fonksiyonu Algoritması.

Denetleyicinin *entity* bölümü sabit bir yapıya sahiptir. *WriteControllerENTITY()* denetleyicinin *entity* kısmını ve *entity* içindeki gerekli *portmap*i uygun bir şekilde tanımlar. Denetleyicinin en karmaşık kısmı *architecture* kısmıdır. Şekil 2.46'da denetleyicinin *architecture* kısmını oluşturan *WriteControllerARCHITECTURE()* fonksiyonunun basitleştirilmiş algoritması görülmektedir. *Architecture* kısmı yapısal olarak sabit bölümlerden oluşur. Denetlenecek YSA veri yoluna göre her bölümün içyapısı değişiklik arz edebilir. *WriteControllerARCHITECTURE()* ilk önce denetleyici oluşturmada kullanılacak gerekli dâhil sinyalleri tanımlar. Ardından denetlenecek veri yolunu analiz ederek veri yolu içinde kullanılmış *weight/bias* register adedini ve veri yolunun giriş adedini tespit eder. Bu sayıların tespit edilmesi oluşturulacak denetleyici için önemlidir. Çünkü bu sayılara göre denetleyicinin sonlu durum makinesi şekil alacaktır. Bu sayılar belirlendikten sonra öncelikle sonlu durum makinesinde kullanılacak durum değişkeni ve bu durum değişkeninin alacağı değerler tanımlanır. Bu tanımlamalarda yukarıda belirlenen register adetleri kullanılır. Denetleyicinin sonlu durum makinesi iki *process* halinde oluşturulur. Birinci *process* sonlu durum makinesinin kendisi, ikinci *process* ise hem sonraki durum belirleyeni hem de denetleyici çıkışını belirleyen kombinasyonel lojik *process*idir. Bu iki *processin* oluşturulmasında da yine algoritmanın başında belirlenen register adetleri kullanılır. *WriteControllerARCHITECTURE()* fonksiyonu son aşamada oluşturduğu dâhili kontrol sinyallerinin sinyallerine bağlantılar yaparak sonlanır.

```
Void WriteControllerArchitecture(char *FileName, NetListType *N, LibraryType *L, TempType *T)
Begin
        Controllerdaki dahili sinyallerin olusturulmasi,
        YSA sistmemindeki toplam weight ve bias register sayisinin bulunmasi,
        YSA sistmemindeki toplam giriş ve çıkış register sayisinin bulunmasi,
        State turunu ve değerlerini tanimlanmasi,
        FSM Process'inin olusturulmasi,
        CLogic Process'in olusturulmasi,
        Dahili kontrol sinyallerinin çıkışa aktarılması,
End
```

Şekil 2.46. *WriteControllerArhitecture()* Fonksiyonu Algoritması.

2.8.3.6. WriteANNSYS(): En Üst Seviye Blok Diyagram Fonksiyonu

Sarıtekin'in çalışmasında oluşturulan veri yolu ile bu çalışmada oluşturulan denetleyicinin bir adres ünitesi ile birlikte bir üst seviye blok içinde birleştirilerek YSA sisteminin oluşturulması gerekmektedir. Bu işlem şekil 2.47'de algoritması verilen *WriteANNSYS()* fonksiyonu sayesinde yapılır. Fonksiyon öncelikle en üst seviye *entity* tanımlamasını yapar. Ardından bu üst seviye *entity* için *architecture* kısmı tanımlanır. Architecture kısmı dört adımda oluşturulur. İlk aşamada üniteleri birbirine bağlamak için gerekli ara sinyaller tanımlanır. Ardından sırasıyla YSA veri yolu, Adres ünitesi, YSA denetleyicisi *instantiate* edilerek otomatik YSA sistem tanımlaması bitirilmiş olur.

```
Void WriteANNSYS(char *FileName, NetListType *N, LibraryType *L, TempType *T)
Begin
   En Üst Seviye ANNSYS Entity oluştur,
   En Üst Seviye ANNSYS Architectıre oluştur,
      Bağlantı sinyallerini tanımla,
      YSA veri yolunu instantiate et,
      Adres Ünitesini instantiate et,
      YSA Denetleyicisini instantiate et,
End
```

Şekil 2.47. *WriteANNSYS()* En Üst Seviye Blok Diyagram (Top Modül) VHDL Yazdırma Fonksiyonu Algoritması.

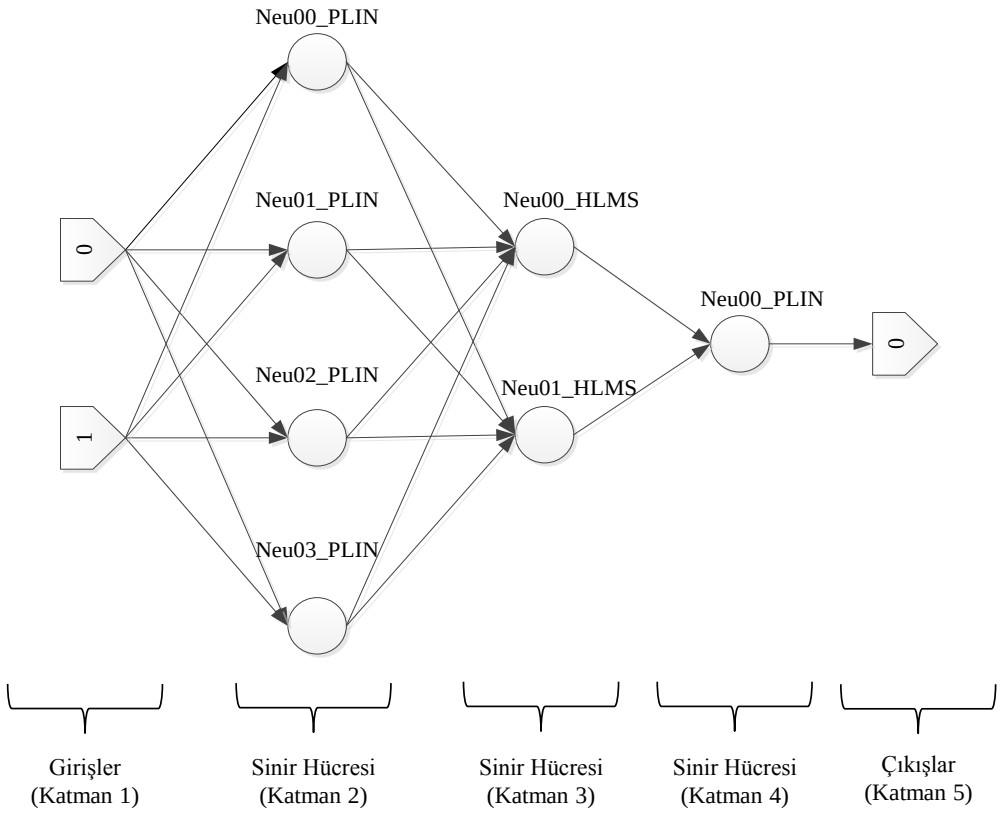
3. BULGULAR VE TARTIŞMA

3.1. ANNCONT'UN TEST EDİLMESİ

Bu çalışmada, geliştirilen ANNCONT'un ve ANNSYS' in test edilmesi için iki farklı test durumu oluşturulmuştur. Oluşturulan test durumları ANNCONT ve ANNSYS aracılığıyla başarıyla çok hızlı bir şekilde YSA sistemine dönüştürülerek sistemin işlerliliği gösterilmiştir.

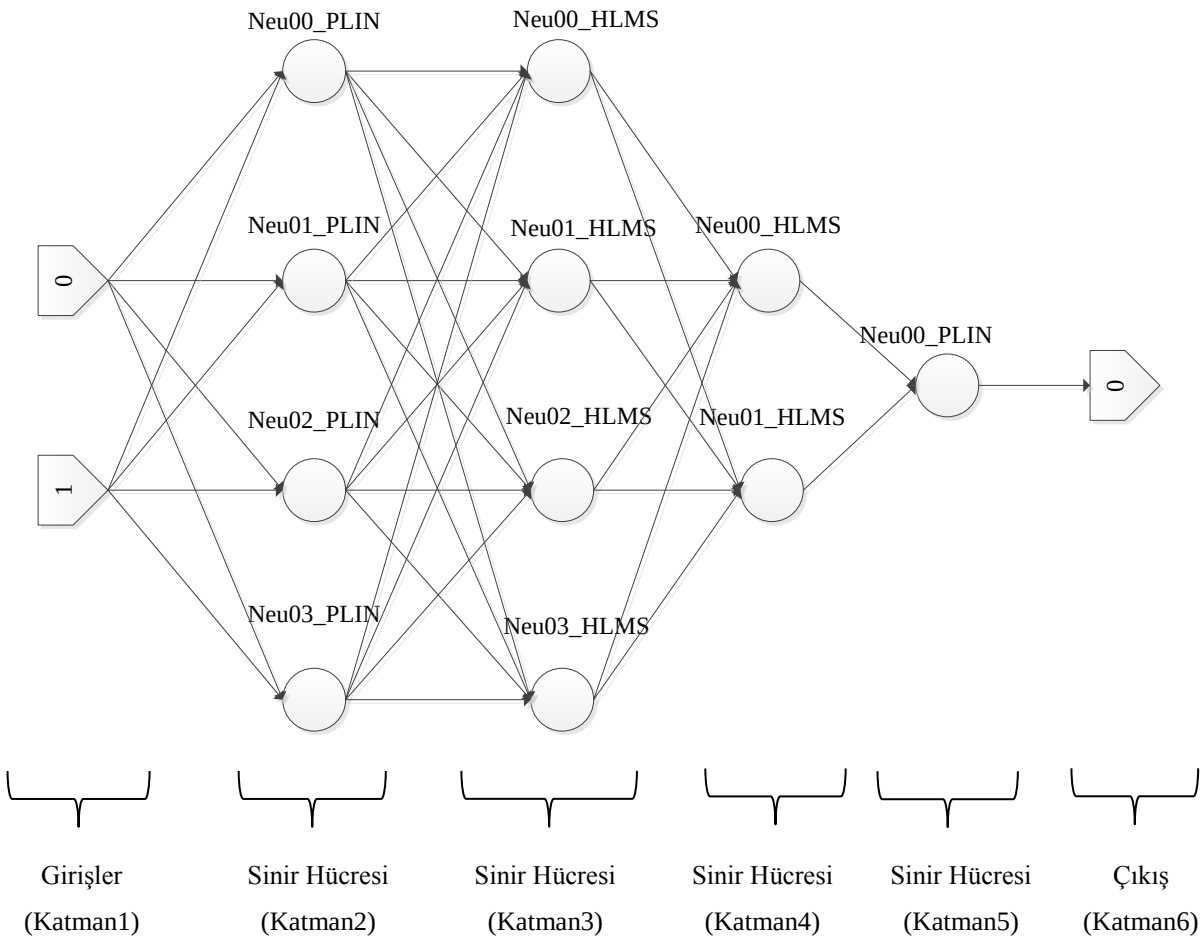
3.1.1. Test Durumları

Şekil 3.1'de birinci test durumu için belirlenen hedef YSA yapısı görülmektedir. Bu YSA yapısı giriş, çıkış ve üç gizli katman olmak üzere toplam 5 katmandan oluşmaktadır. Birinci ve üçüncü gizli katmanda toplam beş adet iki girişli *purelineer* aktivasyon fonksiyonlu hücrelerden oluşmaktadır. İkinci gizli katmanda ise iki adet dört girişli simetrik hard limit aktivasyon fonksiyonuna sahip hücreler kullanılmıştır.



Şekil 3.1. Birinci Test Durumu Sinir Ağı Şeması (Sarıtekin, 2011).

Şekil 3.2'de ise ikinci test durumu için belirlenen hedef YSA yapısı görülmektedir. Bu YSA yapısı giriş, çıkış ve dört gizli katman olmak üzere toplam 6 katmandan oluşmaktadır. Birinci gizli katman, toplam dört adet, dördüncü gizli katmanda toplam bir adet iki girişli purelineer aktivasyon fonksiyonuna sahip hücrelerden, ikinci gizli katman toplam dört adet ve üçüncü gizli katmanda toplam iki adet dört girişli simetrik sabit limit aktivasyon fonksiyonuna sahip hücrelerden oluşmaktadırlar.



Şekil 3.2. İkinci Test Durumu Sinir Ağı Şeması.

Birinci ve ikinci test durumlarındaki hedef YSA'lar için oluşturulan *NetList* tanımlamaları Şekil 3.3'te (a) ve (b) de verilmiştir.

```
NETLIST 5
[
      LAYER 0 INPUT 2
      [
            INP00 INP01
      ]
      LAYER 1 NEURON 4
      [
            NEU00 PLIN 0 0.0 2
                  0 INP00 1.1
                  0 INP01 1.1
            NEU01 PLIN 0 0.0 2
                  0 INP00 1.1
                  0 INP01 1.1
            NEU02 PLIN 0 0.0 2
                  0 INP00 1.1
                  0 INP01 1.1
            NEU03 PLIN 0 0.0 2
                  0 INP00 1.1
                  0 INP01 1.1
      ]
      LAYER 2 NEURON 2
      [
            NEU00 HLMS 0 0.0 4
                  1 NEU00 1.1
                  1 NEU01 1.1
                  1 NEU02 1.1
                  1 NEU03 1.1
            NEU01 HLMS 0 0.0 4
                  1 NEU00 1.1
                  1 NEU01 1.1
                  1 NEU02 1.1
                  1 NEU03 1.1
      ]
      LAYER 3 NEURON 1
      [
            NEU00 PLIN 0 0.0 2
                  2 NEU00 1.1
                  2 NEU01 1.1
      ]
      LAYER 4 OUTPUT 1
      [
            OUT00 3 NEU00
      ]
]
PARAMETERS 4
[
      DataType float
      DataWidth 32
      AddressWidth 32
      VHDLName Deneme
]
```

(a)

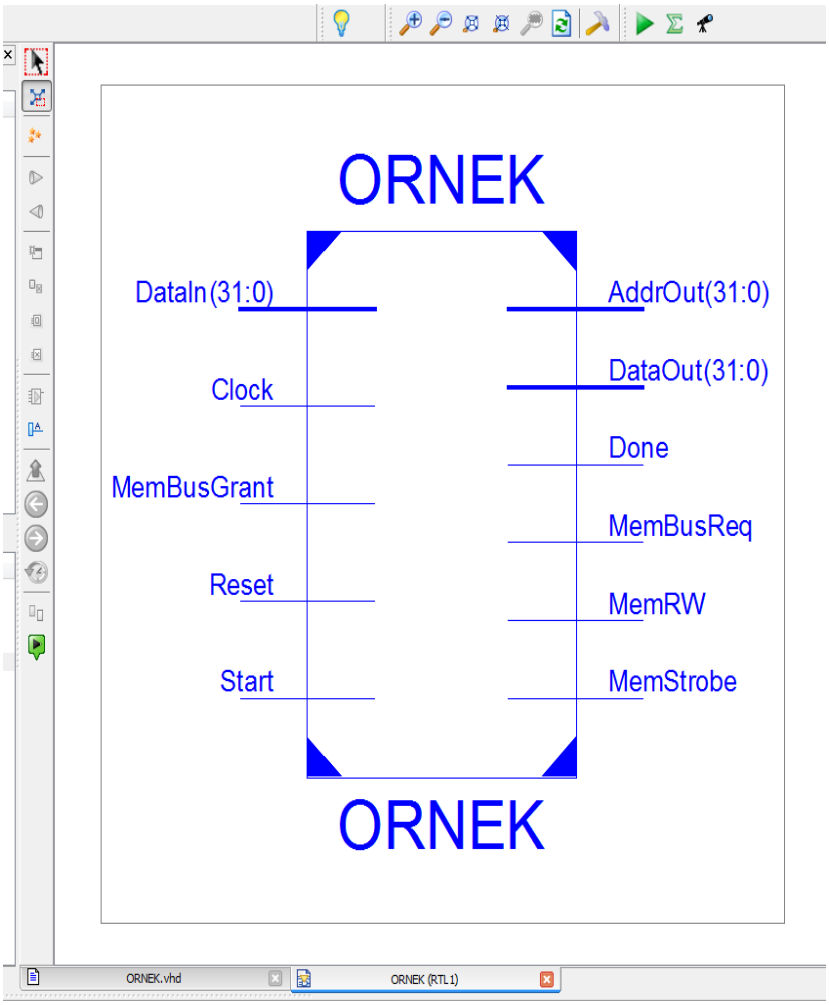
```
NETLIST 6
[
      LAYER 0 INPUT 2
      [
            INP00 INP01
      ]
      LAYER 1 NEURON 4
      [
            NEU00 PLIN 0 0.0 2
                  0 INP00 1.1
                  0 INP01 1.1
            NEU01 PLIN 0 0.0 2
                  0 INP00 1.1
                  0 INP01 1.1
            NEU02 PLIN 0 0.0 2
                  0 INP00 1.1
                  0 INP01 1.1
            NEU03 PLIN 0 0.0 2
                  0 INP00 1.1
                  0 INP01 1.1
      ]
      LAYER 2 NEURON 4
      [
            NEU00 HLMS 0 0.0 2
                  1 NEU00 1.1
                  1 NEU01 1.1
            NEU01 HLMS 0 0.0 2
                  1 NEU00 1.1
                  1 NEU01 1.1
            NEU02 HLMS 0 0.0 2
                  1 NEU00 1.1
                  1 NEU01 1.1
            NEU03 HLMS 0 0.0 2
                  1 NEU00 1.1
                  1 NEU01 1.1
      ]
      LAYER 3 NEURON 2
      [
            NEU00 HLMS 0 0.0 2
                  1 NEU00 1.1
                  1 NEU01 1.1
            NEU01 HLMS 0 0.0 2
                  1 NEU00 1.1
                  1 NEU01 1.1
      ]
      LAYER 4 NEURON 1
      [
            NEU00 PLIN 0 0.0 2
                  1 NEU00 1.1
                  1 NEU01 1.1
      ]
      LAYER 5 OUTPUT 1
      [
            OUT00 2 NEU00
      ]
]
PARAMETERS 4
[
      DataType float
      DataWidth 32
      AddressWidth 32
      VHDLName DataPath
]
```

(b)

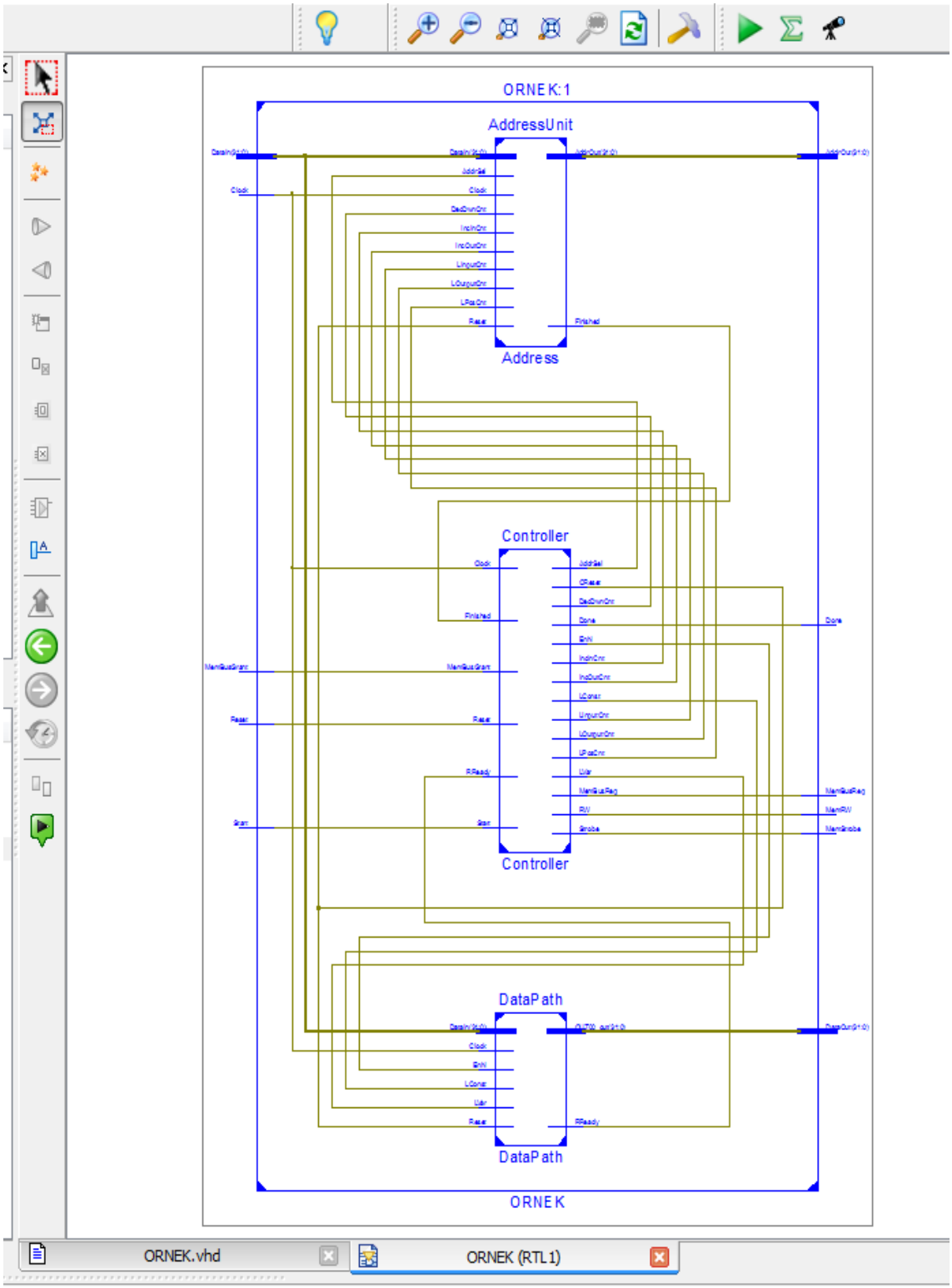
Şekil 3.3. Test Durumları İçin Yapılan *NetList* Tanımlamaları : (a) İki Girişli Üç Gizli Katmanlı Bir Çıkışlı *NetList1* Tanımlaması (Sarıtekin, 2011). (b) İki Girişli Dört Gizli Katmanlı Bir Çıkışlı *NetList2* Tanımlaması

ANNGEN, ANNCONT ve ANNSYS Şekil 3.3'ki *NetList1* ve *NetList2* ile birlikte çalıştırılarak istenen YSA'lar için VHDL kodlarının saniyeler içinde başarılı bir şekilde üretilmesi sağlanmıştır. Ek-1 ve Ek-2 de denetleyici ve YSA sistemi için otomatik olarak oluşturulmuş YSA sistemi VHDL kodları kısaltılarak sunulmuştur.

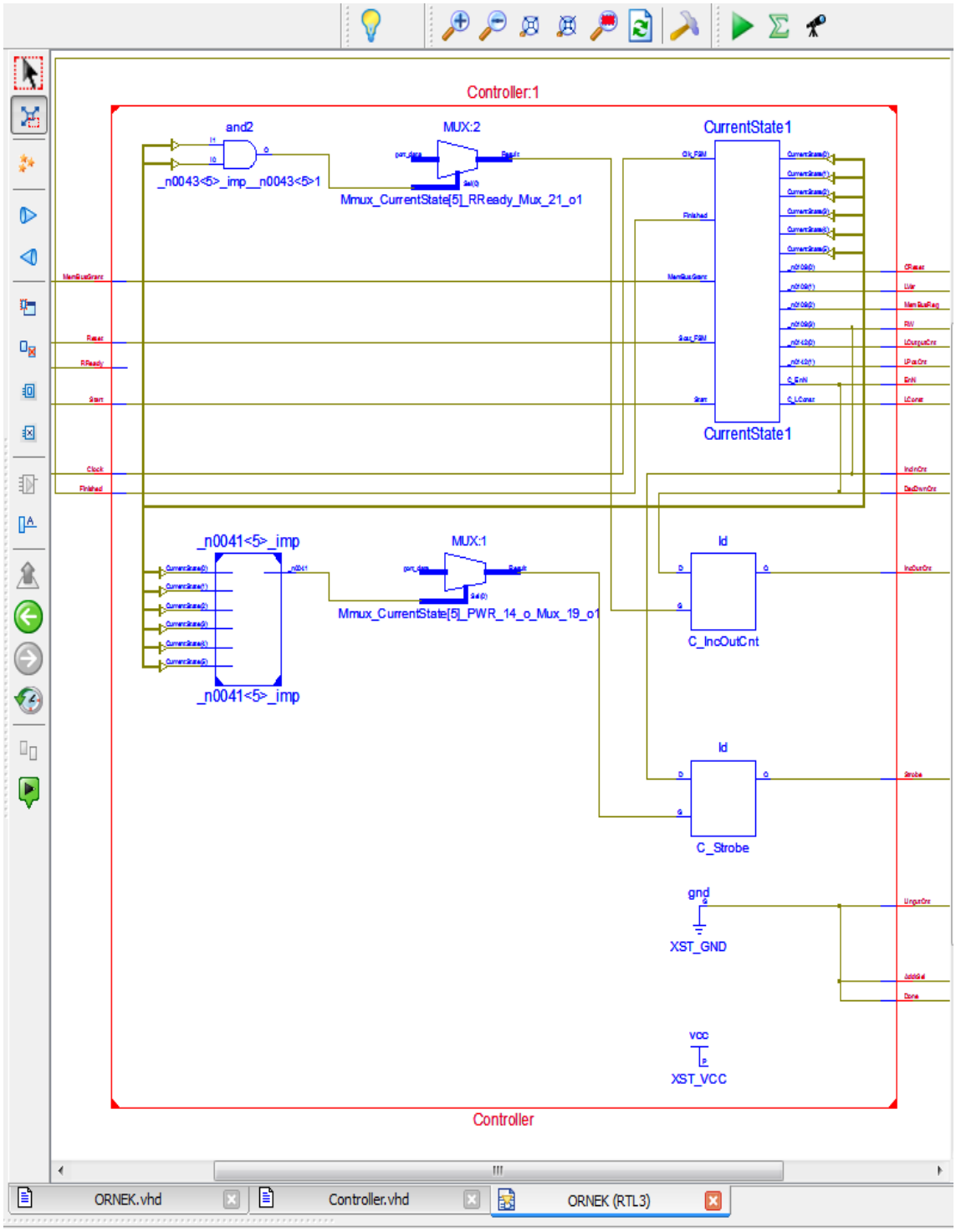
Oluşturulan VHDL kodlarının doğruluğunu test etmek için XILINX ISE tasarım aracı ile kodlar sentezlenerek RTL diyagramları oluşturulmuştur. Şekil 3.4, şekil 3.5, şekil 3.6 ve şekil 3.7'de sırasıyla birinci test durumu için otomatik olarak oluşturulan YSA sistem VHDL kodlarının sentezlenmesi sonucunda ortay çıkan en üst seviye, ikinci seviye, denetleyici iç yapısı ve veri yolu iç yapısı görülmektedir.



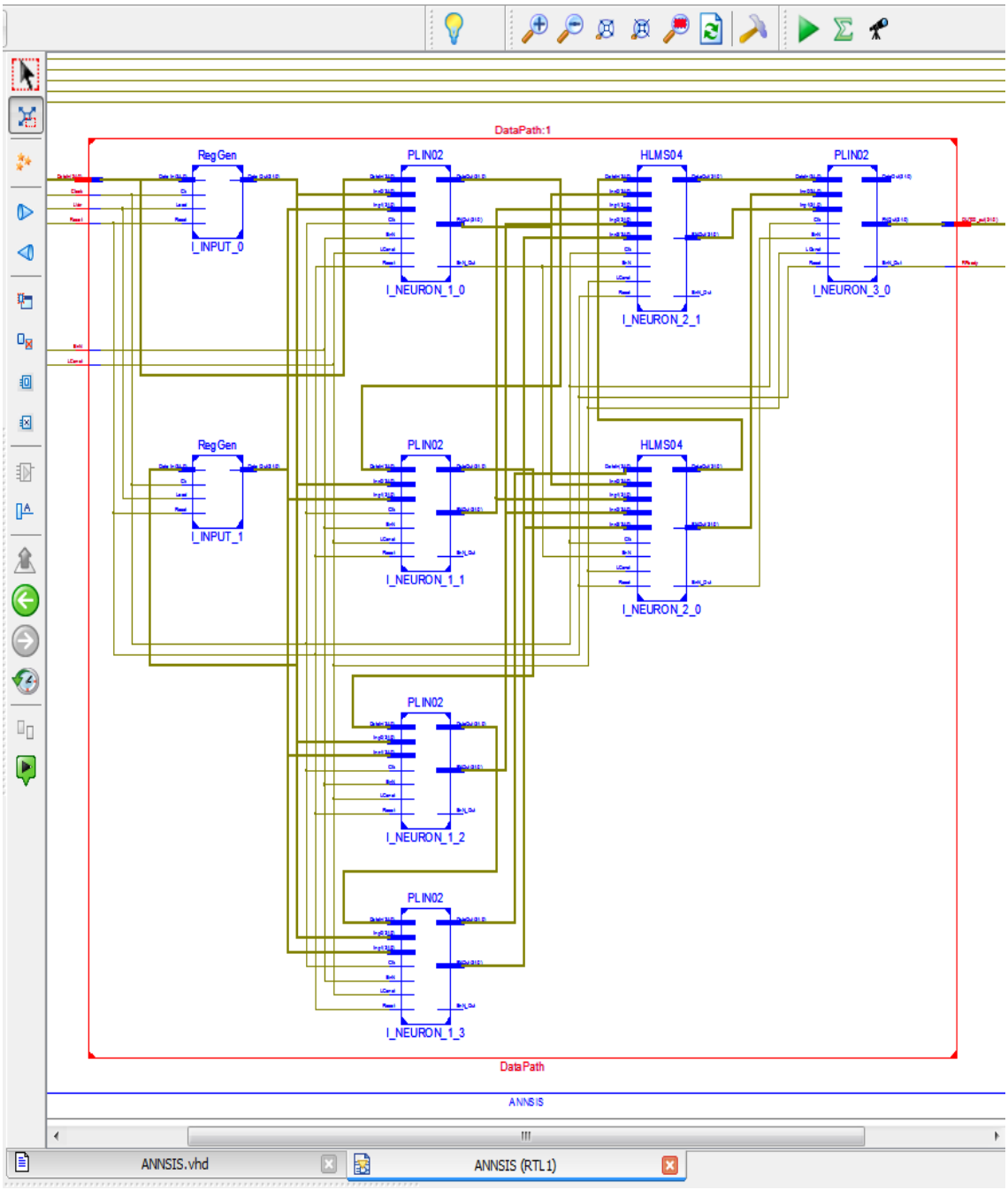
Şekil 3.4. *ÖRNEK RTL* Yapısı (ANNSYS ile Otomatik Olarak Oluşturulmuş Örnek YSA Üst Seviye Blok Diyagramı).



Şekil 3.5. Örnek YSA Sistemi İkinci Seviye RTL Şeması



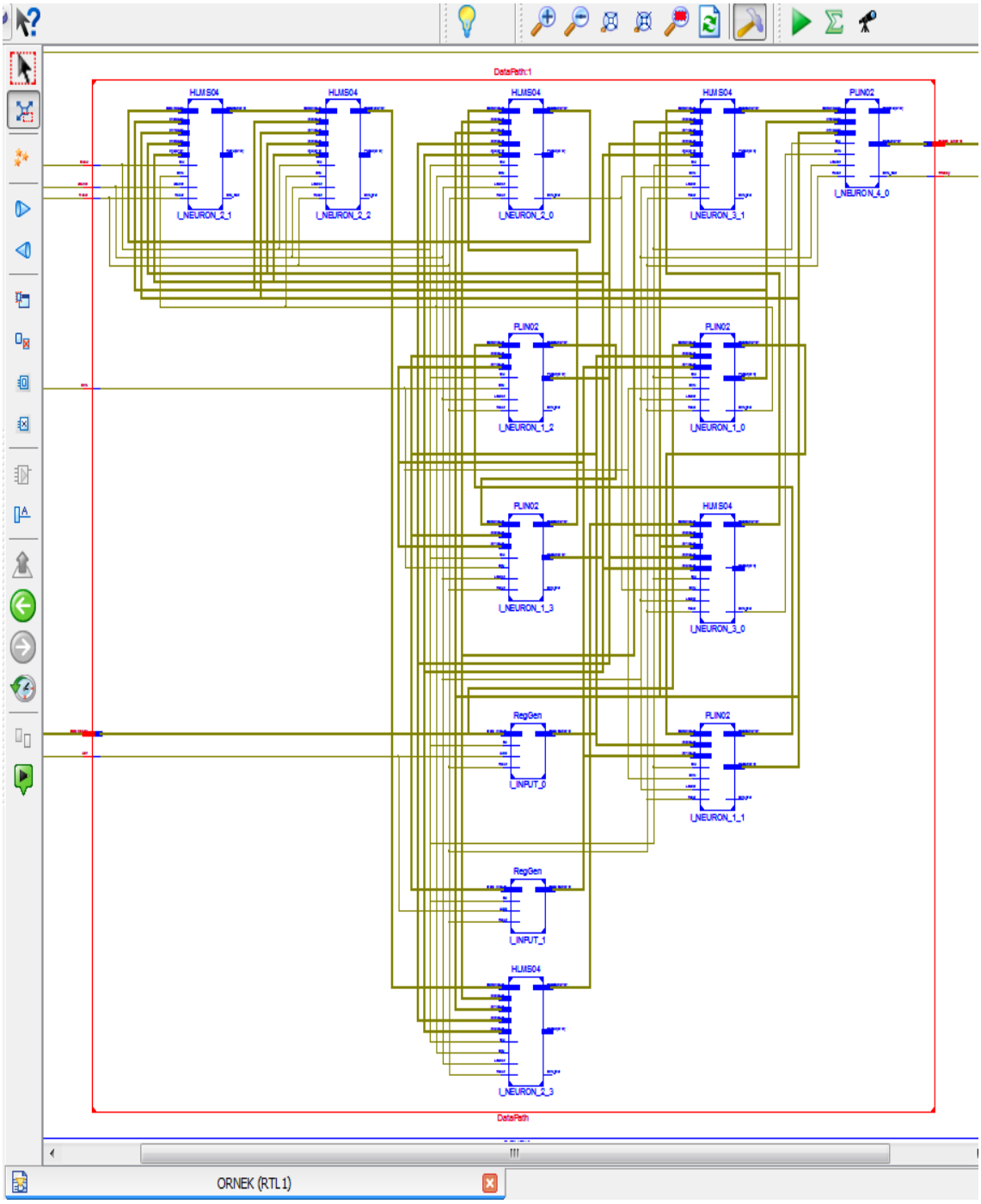
Şekil 3.6. Denetleyici (Controller) RTL Yapısı.



Şekil 3.7. *NetList1* için oluşturulan *YSA* Sistem *RTL* Yapısı (Sarıtekin, 2011).

İkinci test durumu için oluşturulan VHDL kodlarının sentezlenmesi sonucunda ortaya çıkan en üst seviye ikinci seviye, denetleyici içyapısı blok diyagramları aynı olduğu için şekilleri burada verilmemiştir. İkinci test durumunun denetleyicisine baktığımızda denetleyicinin içyapısı aynı olmasına karşın denetleyici içerisinde yer alan *currentstate* ve *clogic* kombinasyonel lojik bloklarının yapısı denetlenen veri yoluna göre farklılık göstermektedir. Bu ünitelerin içyapısı diyagram olarak çok karmaşık olduğu için burada sunulamamıştır.

İkinci test durumu için oluşturulan veri yolu RTL blok diyagramı şekil 3.8'de verilmiştir



Şekil 3.8. *NetList2* için oluşturulan *YSA* Sistem *RTL* Yapısı.

4. SONUÇLAR VE ÖNERİLER

Yapay sinir ağları (YSA'lar) (Artificial Neural Network-ANN) insan beyninin sinir sistemini ve çalışma prensibini temel alan elektriksel ve matematiksel modellerdir. Başka bir bakış açısıyla insan beyninin ufak bir kopyası gibidirler. YSA'lar, öğrenme yoluyla yeni bilgiler üretebilme, keşfedebilme, gözlemleyebilme yeteneklerini, yardım almadan yapabilen sistemleri geliştirmede yapı taşı olarak kullanılmak üzere tasarlanmışlardır. YSA uygulamaları bilindiği gibi iki ayrı aşamada gerçekleşmektedir. Öğrenme ve test aşaması olan bu aşamaların ardından YSA çıkışının anında hesaplanabilmesi gerekmektedir. Bu durum oldukça karmaşık ve uzun bir süreçtir. Ayrıca çok yüksek CPU gücü gerektirir. Bu uzun sürecin kısaltılması için birçok teknoloji kullanılmakta ve gün geçtikçe geliştirilmektedir. Özellikle gerçek zamanlı uygulamalarda yazılım olarak tasarlanan YSA'lar istenen performansı gösterememektedir. Bu sebepten dolayı alternatif yöntemler geliştirilmeye çalışılmıştır.

Bu aşamada FPGA devresi kullanımı ortaya çıkmıştır. FPGA devrelerinin sonsuz kez yeniden yapılandırılabilir olması YSA uygulamaları için büyük bir avantajdır. Bu avantaj sayesinde bir tasarım kısa zamanda değiştirilerek birçok farklı uygulamada kullanılabilmektedir. Bu sebeple FPGA'ların sahip olduğu hız, güvenlik, paralel işlem yapabilme yeteneği ve yeniden tasarlanabilme özelliği sayesinde YSA'lar ile çok uyumlu çalışmalar yapılabilmektedir. Ancak YSA'ların FPGA'lara uygulanması zaman alan ve uzman gerektiren bir iştir. Aynı zamanda uzman tarafından gerçekleştirilen tüm bu işlemlertasarımı kodlama sırasında oluşabilecek kod hataları düzenleme (debug) sorununu da beraberinde getirmektedir.

Bu problemlerden yola çıkarak bu tez çalışmasında Yapay Sinir Ağlarının otomatik olarak FPGA'lara uygulanmasını gerçekleştiren denetleyici tasarım aracı olan ANNCONT ve var olan veri yolu ile birleştirerek YSA sistemini ortaya çıkartan ANNSYS geliştirilmiştir.

Bu çalışmada geliştirilen ANNCONT ve ANNSYS'in etkinliğini test edebilmek ve hedeflenen amaçlara ulaşılabildiğini göstermek amacıyla iki test durumu oluşturulmuş ve ANNCONT ve ANNSYS bu test durumları ile test edilmiştir. ANNCONT ve

ANNSYS saniyeler içerisinde hatasız bir şekilde verilen YSA veri yolunu kontrol etmek için gerekli denetleyici VHDL kodunu ve YSA sistem VHDL kodunu üretmiştir. Üretilen kodun doğruluğunu tespit etmek için XILINX ISE tasarım aracı kullanılmıştır. Üretilen kodlar ISE ile sorunsuz bir şekilde sentezlenerek RTL görüntüleri oluşturulmuştur.

Test durumlarının ANNCONT ve ANNSYS kullanılarak sorunsuz ve çok hızlı bir şekilde YSA sistemine dönüştürülmesi ile çalışmamızda hedeflenen amaçlara ulaşılmıştır.

Bu çalışmada sadece feed-forward (ileri beslemeli) YSA dikkate alınmıştır. Çalışmanın devamında diğer YSA topolojileri içinde sistem geliştirilebilir. Ayrıca burada tasarlanan YSA sistemi sadece hafıza üzerinden tasarlanacak şekilde tasarlanmıştır. Hafıza dışında gerçek zamanlı olarak girdileri alıp sonuç üreten YSA sistemlerini otomatik olarak oluşturmak üzere çalışma genişletilebilir.

5. KAYNAKLAR

Makaleler

ÇAVUŞLU M.A., Karakuzu C., Şahin S., Karakaya F., **2008**, Yapay Sinir Ağı Eğitiminin IEEE 754 Kayan Noktalı Sayı Formatı İle FPGA Tabanlı Gerçeklenmesi, Gomsis 2008,3-4-5 Kasım 2008, İstanbul Teknik Üniversitesi Süleyman Demirel Kültür Merkezi, Maslak Yerleşkesi, İstanbul.

GLACKİN B., Harkin J., McGinnity T., Maguire L., Wu Q. (**2009b**)., "Emulating spiking neural networks for edge detection on FPGA hardware," in Proceedings of International Conference on Field Programmable Logic and Applications FPL 2009, Karlsruhe

MONMASSON E., Cirstea M. N., **2007**, FPGA Design Methodology for Industrial Control Systems—A Review, Ieee Transactıons On Industrıal Electronıcs, VOL. 54, NO. 4

QUY Ngoc Le and Jae-Wook Jeon, Member, **2010**, Neural-Network-Based Low-Speed-Damping Controller for Stepper Motor With an Fpga Ieee Transactıons On Industrıal Electronıcs, VOL. 57, NO. 9.

RAMÓN J. Aliaga, Rafael Gadea, Ricardo J. Colom, Joaquín Cerdá, Néstor Ferrando, and Vicente Herrero, **2009**, A Mixed Hardware-Software Approach to Flexible Artificial Neural Network Training on FPGA, Institute for the Implementation of Advanced Information and Communication Technology (ITACA) Universidad Politécnica de Valencia 46022 Valencia, Spain.

SIRMAÇEK B., **2007**, FPGA İle Mobil Robot İçin Öğrenme Algoritması Modellenmesi,

ŞAHIN I., **2010**, A 32-Bit Floating-Point Module Design for 3D Graphic Transformations, Scientific Research and Essays Vol. 5 (20), pp. 3070-3081, 18

October, 2010, Available Online at http://www.academicjournals.org/SRE, ISSN 1992-2248 ©2010 Academic Journals, Full Length Research Paper.

ŞAHIN I., Çakıcı S., Erdoğmuş P., **2011**, Performans And Cost Evaluatıons Of Adders Used In FPGA-BASED Systems.

ŞAHIN I., Koyuncu I., **2012**, "Design and Implementation of Neural Networks Neurons with RadBas, LogSig, and TanSig Activation Functions on FPGA", Elektronika Ir Elektrotechnika,. No. 4 (120), pp. 51-54, 2012. (Science Citation Index Expanded)

PINTER B-A., Sobe A., Elmenreich W., **2012**, Towards the Light - Comparing Evolved Neural Network Controllers and Finite State Machine Controllers, Institute of Networked and Embedded Systems University of Klagenfurt/Lakeside Labs Klagenfurt, Austria.

LAZAREVIÆ L., **2009**, Miliæev D., Finite State Machine Automatic Code Generation, University of Belgrade, Faculty of Electrical Engineering.

Kitaplar

ELMAS, Ç., **2003**, Yapay Sinir Ağları (Kuram, Mimari, Eğitim, Uygulama), Seçkin Yayıncılık, Ankara, 9.789.753.476.126.

ELMAS, Ç., **2007**, Yapay Zekâ Uygulamaları, Seçkin Yayıncılık, Ankara, 9.789.750.206.146.

ÖZTEMEL E., **2003**. Yapay Sinir Ağları, Papatya Yayıncılık, İstanbul.

JOSEPH P. ELM, **2005**. Designing For Reuse Of Configurable Logic, Software Engineering Instute Carnegie Mellon University, Pitsburg, F19628-00-C-0003

Tezler

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

YILMAZ N., **2008**, Sahada Programlamalı Kapı Dizileri (Fpga) Üzerinde Bir Ysa'nın Tasarlanması Ve Donanım Olarak Gerçekleştirilmesi, Yüksek Lisans Tezi, Selçuk Üniversitesi, Konya.

KOYUNCU İ., **2008**, A Matrix Multiplication Engine for Graphic Systems Designed to run on FPGA Devices, Düzce Üniversitesi, Yüksek Lisans Tezi, Düzce.

DERE A., **2009**, Yapay Sinir Ağları Yöntemi İle Sıvılaşma Analizi Ve Adapazarı İçin Örnek Bir Uygulama, Yüksek Lisans Tezi, Sakarya Üniversitesi, Sakarya

KAPLAN T., **2009**, Des Blok Şifreleme Algoritmasının FPGA Üzerinde Düşük Enerjili Tasarımı, İstanbul Teknik Üniversitesi, İstanbul

GÜNEREN H., **2010**, FPGA Üzerinde Kayan Nokta Sayı Formatı Kullanılarak Yapay Sinir Ağı Tabanlı Sınıflandırma İşlemi, Lisans Tezi, Yıldız Teknik Üniversitesi, İstanbul

ÖZTÜRK E., **2010**, Fpga Kullanarak 16 Bitlik Mikroişlemci Tasarımı, Lisans Tezi, Yıldız Teknik Üniversitesi, İstanbul.

SUBAŞI H., **2010**, Yapay Sinir Ağları İle Atık Su Arıtma Performansının Modellenmesi, Yüksek Lisans Tezi, Çukurova Üniversitesi, Adana

SARITEKİN Kemal N., **2011**, Yapay Sinir Ağlarının Otomatik Olarak Fpga'ya Uygulanması İçin Veri Yolu Tasarım Aracı, Yüksek Lisans Tezi, Düzce Üniversitesi, Düzce

Bildiriler

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

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

DEMUTH, H., Hagan, M. T., Beale M., **2011**[Online], Neural Network Toolbox™ 7 For User's Guide with MATLAB, The MathWorks, USA, Revised for Version 7.0.1 (Release 2011a), April 2011.

ANDERSON, D. & McNeil G., Artificial Neural Networks Technology.

WILLERT, C. **2000**. The Evolution of Programable Logic Design Technology, Xilinx Inc.

BAUMANN C., **2010**, Field Programmable Gate Array (FPGA), Summer Paper For The Seminar "Embedded System Architecture", University of Innsbruck.

Elektronik Yayınlar

ANDERSON, D. & McNeil G., Artificial Neural Networks Technology, http://www.dacs.dtic.mil/techs/neural/neural_ToC.html [Ziyaret Tarihi:13 Şubat **2012**].

ANONİM[1], http://tr.wikipedia.org/wiki/FPGA [Ziyaret Tarihi:2 Şubat **2012**].

ANONİM[2], http://e-bergi.com/2008/Subat/Yapay-Sinir-Aglari [Ziyaret Tarihi:2 Şubat **2012**].

ANONİM[3], http://www.suleymantosun.com/wp-content/uploads/2010/05/Yapay_Sinir_Aglari.pdf, [Ziyaret Tarihi:2 Şubat **2012**].

ANONİM[4], http://lowercolumbia.edu/students/academics/facultyPages/rhode-cary/intro-neural-net.htm [Ziyaret Tarihi:2 Şubat **2012**].

ANONİM[5], http://lowercolumbia.edu/students/academics/facultyPages/rhode-cary/backpropagation.htm [Ziyaret Tarihi:2 Şubat **2012**].

ANONİM[6], http://www.yapay-zeka.org/modules/wiwimod/index.php?page=ANN [Ziyaret Tarihi:8 Mayıs **2012**]

ANONİM[7], http://www.politekno.com/fpga-tasarim-akisi-araclari [Ziyaret Tarihi:10 Nisan **2012**].

ANONİM[8], http://www.tola.com.tr/docs/article/FPGAgiris.pdf [Ziyaret Tarihi:21 Ekim 2012]

ANONİM[9], http://www.eetimes.com/design/programmable-logic/4014815/All-about-FPGAs?pageNumber=0 [Ziyaret Tarihi:31 Ekim 2012]

ANONİM[10], http://only-vlsi.blogspot.com/search/label/FSM [Ziyaret Tarihi:15 Aralık 2012]

ANONİM[11], http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html [Ziyaret Tarihi: 05 Ocak 2013]

ANONİM[12], http://www.so-logic.net/documents/knowledge/tutorial/Basic_FPGA_Tutorial/Basic_FPGA_Tutorial2.html [Ziyaret Tarihi: 10 Ocak 2013]

BÜRHAN, Y., Gülenç , H., *2011,* FPGA, [Online], *Türkiye,* web.firat.edu.tr/bilmuh/gaydin/dersler/0809/bmu401/ppt/fpga.doc, [Ziyaret Tarihi:10 Ocak 2012].

6.EKLER

EK-1: *NETLİST1* İÇİN OLUŞTURULAN YSA SİSTEM VHDL KODU

Aşağıda, *NetList1* için üretilmiş YSA sistem VHDL kodu görülmektedir. VHDL kodu dört ana bölümden oluşmaktadır. Birinci bölüm tüm parçaları içerisinde bulunduran en üst seviye blok diyagram kodu olan ANNSYS, ikinci bölüm alt katmanlardan biri olan sinir ağı yapısının oluşturulduğu DataPath, üçüncü bölüm sistemin otomatik olarak gerçeklenmesini sağlayan CONTROLLER ve dördüncü bölüm de sistem çalışmasında gerekli adres bilgilerini tutan Adres Ünitesi bölümüdür. Bu ana bölümlerin dışında ana yapıyı oluşturan alt bölümlerde burada sunulmuşlardır. Üretilen kodlar çok uzun olduğundan VHDL kod parçaları bazı bölümlerde ":" ile kısaltılarak verilmiştir.

```
-------------------------------------------------------------------------------------------------------------------
```

```
-----------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :      Ibrahim SAHİN-Günay TEMÜR
-- Date      :      Fabruary 2013
-- Design    :      ANNSYS
-----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ANNSYS is
    Port ( DataIn         : in  STD_LOGIC_VECTOR (31 downto 0);
           Start          : in  STD_LOGIC;
           Reset          : in  STD_LOGIC;
           MemBusGrant    : in  STD_LOGIC;
           Clock          : in  STD_LOGIC;
           DataOut        : out  STD_LOGIC_VECTOR (31 downto 0);
           AddrOut        : out  STD_LOGIC_VECTOR (31 downto 0);
           MemBusReq      : out  STD_LOGIC;
           MemStrobe      : out  STD_LOGIC;
           MemRW          : out  STD_LOGIC;
           Done           : out  STD_LOGIC);
end ANNSYS;
architecture Behavioral of ANNSYS is
      signal CReset       : std_logic;
      signal LConst       : std_logic;
      signal Lvar         : std_logic;
      signal EnN          : std_logic;
      signal RReady       : std_logic;
      signal LInputCnt    : STD_LOGIC;
      signal LOutputCnt   : STD_LOGIC;
      signal LPcsCnt      : STD_LOGIC;
      signal AddrSel      : STD_LOGIC;
      signal IncInCnt     : std_logic;
      signal IncOutCnt    : std_logic;
      signal DecDwnCnt    : std_logic;
      signal Finished     : std_logic;
begin
```

```
	DataPath  : entity work.Deneme
	port map(Clock,CReset,LConst,LVar,EnN,DataIn,RReady,DataOut);
	Controller: entity work.Controller
	portmap(Clock,Reset,Start,MemBusGrant,Finished,RReady,CReset,LConst,LVar
	,EnN,MemBusReq,MemStrobe,MemRW,LInputCnt,LOutputCnt,LPcsCnt,AddrSel,IncI
	nCnt,IncOutCnt,DecDwnCnt,Done);
	Address : entity work.AddressUnit
	Generic Map(32)
	Portmap(clock,CReset,LInputCnt,LOutputCnt,LPcsCnt,AddrSel,IncInCnt,IncOu
	tCnt,DecDwnCnt, DataIn,AddrOut,Finished);
end Behavioral;
----------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :      Ibrahim SAHİN-Namık Kemal SARITEKİN
-- Date      :      Fabruary 2013
-- Design    :      FpMult
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY FpMult8 IS
	PORT( InA : IN std_logic_vector(31 DOWNTO 0);
		InB : IN std_logic_vector(31 DOWNTO 0);
		EnN : IN std_logic;
		Clk : IN std_logic;
		Data_Out     : OUT std_logic_vector(31 DOWNTO 0);
		EnN_Out : OUT std_logic
		);
END FpMult8;
ARCHITECTURE Behavioral OF FpMult8 IS
BEGIN
	Data_Out <= InA and InB;
	EnN_Out <= EnN;
END Behavioral;
----------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :      Ibrahim SAHİN- Namık Kemal SARITEKİN
-- Date      :      Fabruary 2013
-- Design    :      FpAdd
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY FpAdd8 IS
	PORT( InA : IN std_logic_vector(31 DOWNTO 0);
		InB : IN std_logic_vector(31 DOWNTO 0);
		EnN : IN std_logic;
		Clk : IN std_logic;
		Data_Out     : OUT std_logic_vector(31 DOWNTO 0);
		EnN_Out : OUT std_logic
		);
END FpAdd8;
ARCHITECTURE Behavioral OF FpAdd8 IS
BEGIN
	Data_Out <= InA or InB;
	EnN_Out <= EnN;
END Behavioral;
----------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :      Ibrahim SAHİN- Namık Kemal SARITEKİN
-- Date      :      Fabruary 2013
-- Design    :      Generic Register
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY RegGen IS
	generic(width: integer := 4);
	PORT(
			Ck		: IN std_logic;
			Reset		: IN std_logic;
			Load		: IN std_logic;
			Data_In		: IN std_logic_vector(width-1 DOWNTO 0);
			Data_Out	: OUT std_logic_vector(width-1 DOWNTO 0));
END RegGen;
ARCHITECTURE Behavioral OF RegGen IS
	SIGNAL D : std_logic_vector(width-1 DOWNTO 0);
BEGIN
	PROCESS
	BEGIN
		WAIT UNTIL ((Ck'EVENT) AND (Ck= '1'));
			IF (Reset = '1') THEN
			D <= (OTHERS => '0');
		ELSIF (Load = '1') THEN
			D <= Data_In;
		ELSE
			D <= D;
		END IF;
	END process;
	Data_Out <= D;
END Behavioral;
----------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name		:	Ibrahim SAHİN- Namık Kemal SARITEKİN
-- Date		:	Fabruary 2013
-- Design	:	YSA 2 PURELINE
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity PLIN02 is
    Port (   Clk    : in  STD_LOGIC;
	     Reset	: in  STD_LOGIC;
	     LConst	: in  STD_LOGIC;
	     Inp0	: in  STD_LOGIC_VECTOR (31 downto 0);
	     Inp1	: in  STD_LOGIC_VECTOR (31 downto 0);
	     DataIn	: in  STD_LOGIC_VECTOR (31 downto 0);
	     DataOut	: out  STD_LOGIC_VECTOR (31 downto 0); --biasli cikis
	     FNOut	: out  STD_LOGIC_VECTOR (31 downto 0); --biasli cikis
	     EnN	: in  STD_LOGIC;
	     EnN_Out	: out std_logic);
end PLIN02;
architecture RTL of PLIN02 is
	SIGNAL Out_W0		: std_logic_vector(31 DOWNTO 0);-- W0 Çıkış
	SIGNAL Out_W1		: std_logic_vector(31 DOWNTO 0);-- W1 Çıkış
	SIGNAL EnN_1		: std_logic;
	SIGNAL EnN_2		: std_logic;
	SIGNAL EnN_3		: std_logic;
	SIGNAL Mul_1_Out	: std_logic_vector(31 DOWNTO 0);-- Çarpma 1 Çıkış
	SIGNAL Mul_2_Out	: std_logic_vector(31 DOWNTO 0);-- Çarpma 2 Çıkış
	SIGNAL Add_1_Out	: std_logic_vector(31 DOWNTO 0);-- Toplama 1 Çıkış
component FpMult8
	port (
	a		: IN std_logic_VECTOR(31 downto 0);
	b		: IN std_logic_VECTOR(31 downto 0);
	operation_nd	: IN std_logic;
	clk		: IN std_logic;
	result		: OUT std_logic_VECTOR(31 downto 0);
	rdy		: OUT std_logic);
end component;
component FpAdd8
	port (
```

```
      a            : IN std_logic_VECTOR(31 downto 0);
      b            : IN std_logic_VECTOR(31 downto 0);
      operation_nd : IN std_logic;
      clk          : IN std_logic;
      result       : OUT std_logic_VECTOR(31 downto 0);
      rdy          : OUT std_logic);
end component;
BEGIN
      W0     :entity work.RegGen       Generic Map(32)           PORT
MAP(clk,Reset,LConst,DataIn,Out_W0);
      W1     :entity work.RegGen       Generic Map(32)           PORT
MAP(clk,Reset,LConst,Out_W0,Out_W1);
      Mult0  :entity work.FpMult8                                PORT
MAP(Inp0,Out_W0,EnN,clk,Mul_1_Out,EnN_1);
      Mult1  :entity work.FpMult8                                PORT
MAP(Inp1,Out_W1,EnN,clk,Mul_2_Out,EnN_2);
      EnN_3 <= EnN_1 and EnN_2;
      Add1   :entity work.FpAdd8
      PORT MAP(Mul_1_Out,Mul_2_Out,EnN_3,clk,Add_1_Out,EnN_Out);
      FNOut <= Add_1_Out;
      DataOut <= Out_W1;
END RTL;
-----------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :     Ibrahim SAHİN- Namık Kemal SARITEKİN
-- Date      :     Fabruary 2013
-- Design    :     YSA 4 HARDLIMS
-----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity HLMS04 is
      Port ( Clk   : in  STD_LOGIC;
      Reset        : in  STD_LOGIC;
      LConst       : in  STD_LOGIC;
      Inp0         : in  STD_LOGIC_VECTOR (31 downto 0);
      Inp1         : in  STD_LOGIC_VECTOR (31 downto 0);
      Inp2         : in  STD_LOGIC_VECTOR (31 downto 0);
      Inp3         : in  STD_LOGIC_VECTOR (31 downto 0);
      DataIn       : in  STD_LOGIC_VECTOR (31 downto 0);
      DataOut      : out  STD_LOGIC_VECTOR (31 downto 0);
      FNOut        : out  STD_LOGIC_VECTOR (31 downto 0);
      EnN          : in  STD_LOGIC;
      EnN_Out      : out std_logic);
end HLMS04;
architecture RTL of HLMS04 is
      SIGNAL Out_W0       : std_logic_vector(31 DOWNTO 0);-- W0 Çıkış
      SIGNAL Out_W1       : std_logic_vector(31 DOWNTO 0);-- W1 Çıkış
      SIGNAL Out_W2       : std_logic_vector(31 DOWNTO 0);-- W2 Çıkış
      SIGNAL Out_W3       : std_logic_vector(31 DOWNTO 0);-- W3 Çıkış
      SIGNAL EnN_1        : std_logic;
      SIGNAL EnN_2        : std_logic;
      SIGNAL EnN_12       : std_logic;
      SIGNAL EnN_3        : std_logic;
      SIGNAL EnN_4        : std_logic;
      SIGNAL EnN_34       : std_logic;
      SIGNAL EnN_5        : std_logic;
      SIGNAL EnN_6        : std_logic;
      SIGNAL Mul_1_Out    : std_logic_vector(31 DOWNTO 0);-- Çarpma 1 Çıkış
      SIGNAL Mul_2_Out    : std_logic_vector(31 DOWNTO 0);-- Çarpma 2 Çıkış
      SIGNAL Mul_3_Out    : std_logic_vector(31 DOWNTO 0);-- Çarpma 3 Çıkış
      SIGNAL Mul_4_Out    : std_logic_vector(31 DOWNTO 0);-- Çarpma 4 Çıkış
      SIGNAL Add_1_Out    : std_logic_vector(31 DOWNTO 0);-- Toplama 1 Çıkış
      SIGNAL Add_2_Out    : std_logic_vector(31 DOWNTO 0);-- Toplama 2 Çıkış
      SIGNAL Add_3_Out    : std_logic_vector(31 DOWNTO 0);-- Toplama 3 Çıkış
component FpMult8
      port (
```

```
      a            : IN std_logic_VECTOR(31 downto 0);
      b            : IN std_logic_VECTOR(31 downto 0);
      operation_nd : IN std_logic;
      clk          : IN std_logic;
      result       : OUT std_logic_VECTOR(31 downto 0);
      rdy          : OUT std_logic);
end component;
component FpAdd8
      port (
      a            : IN std_logic_VECTOR(31 downto 0);
      b            : IN std_logic_VECTOR(31 downto 0);
      operation_nd : IN std_logic;
      clk          : IN std_logic;
      result       : OUT std_logic_VECTOR(31 downto 0);
      rdy          : OUT std_logic);
end component;
BEGIN
      W0    :entity work.RegGen       Generic Map(32)           PORT
MAP(clk,Reset,LConst,DataIn,Out_W0);
      W1    :entity work.RegGen       Generic Map(32)           PORT
MAP(clk,Reset,LConst,Out_W0,Out_W1);
      W2    :entity work.RegGen       Generic Map(32)           PORT
MAP(clk,Reset,LConst,Out_W1,Out_W2);
      W3    :entity work.RegGen       Generic Map(32)           PORT
MAP(clk,Reset,LConst,Out_W2,Out_W3);
      --Bias       :entity work.RegGen Generic Map(32)           PORT
MAP(clk,Reset,LConst,Out_W3,Out_Bias);
      Mult0  :entity work.FpMult8                                PORT
MAP(Inp0,Out_W0,EnN,clk,Mul_1_Out,EnN_1);
      Mult1  :entity work.FpMult8                                PORT
MAP(Inp1,Out_W1,EnN,clk,Mul_2_Out,EnN_2);
      Mult2  :entity work.FpMult8                                PORT
MAP(Inp2,Out_W2,EnN,clk,Mul_3_Out,EnN_3);
      Mult3  :entity work.FpMult8                                PORT
MAP(Inp3,Out_W3,EnN,clk,Mul_4_Out,EnN_4);
      EnN_12 <= EnN_1 and EnN_2;
      EnN_34 <= EnN_3 and EnN_4;
      Add1   :entity work.FpAdd8                                 PORT
MAP(Mul_1_Out,Mul_2_Out,EnN_12,clk,Add_1_Out,EnN_5);
      Add2   :entity work.FpAdd8                                 PORT
MAP(Mul_3_Out,Mul_4_Out,EnN_34,clk,Add_2_Out,EnN_6);
      Add3   :entity work.FpAdd8                                 PORT
MAP(Add_1_Out,Add_2_Out,EnN_6,clk,Add_3_Out,EnN_Out);
      --Add4 :entity work.FpAdd8                                 PORT
MAP(Add_3_Out,Out_Bias,EnN_7,clk,Add_4_Out,EnN_Out);
                  Process (Clk,Add_3_Out)
            Begin
                  If (Add_3_Out(31) = '0') Then
                        FNOut <= "00111111100000000000000000000000";

                  Else
                        FNOut <= "10111111100000000000000000000000";

                  End if;
            End process;
            DataOut <= Out_W3;
END RTL;
-------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :     Ibrahim SAHİN- Namık Kemal SARITEKİN
-- Date      :     Fabruary 2013
-- Design    :     DATAPATH
-------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY DataPath IS
```

```
            PORT (
            Clock : in STD_LOGIC;
            Reset : in STD_LOGIC;
            LConst : in STD_LOGIC;
            LVar : in STD_LOGIC;
            EnN : in STD_LOGIC;
            DataIn : in STD_LOGIC_VECTOR (31 downto 0);
            RReady : Out STD_LOGIC;
            OUT00_out : out STD_LOGIC_VECTOR (31 downto 0)
            );
END ENTITY DataPath;
ARCHITECTURE RTL OF DataPath IS
            --Internal data flow signal
            SIGNAL S_INP00_0 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_INP01_0 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU00_1 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU01_1 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU02_1 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU03_1 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU00_2 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU01_2 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU00_3 : STD_LOGIC_VECTOR (31 downto 0);
            --Init Chain "IC" signals
            SIGNAL S_NEU00_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU01_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU02_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU03_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU00_IC_2 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU01_IC_2 : STD_LOGIC_VECTOR (31 downto 0);
            SIGNAL S_NEU00_IC_3 : STD_LOGIC_VECTOR (31 downto 0);
            --EnN Singal Definitions
            SIGNAL EnN_Out_1_0 : STD_LOGIC;
            SIGNAL EnN_Out_1_1 : STD_LOGIC;
            SIGNAL EnN_Out_1_2 : STD_LOGIC;
            SIGNAL EnN_Out_1_3 : STD_LOGIC;
            SIGNAL EnN_Out_2_0 : STD_LOGIC;
            SIGNAL EnN_Out_2_1 : STD_LOGIC;
            SIGNAL EnN_Out_3_0 : STD_LOGIC;
BEGIN
            --Register Instantiations
            I_INPUT_0: entity work.RegGen Generic Map (32) Port Map (Clock,
Reset, LVar, DataIn, S_INP00_0);
            I_INPUT_1: entity work.RegGen Generic Map (32) Port Map (Clock,
Reset, LVar, S_INP00_0, S_INP01_0);
            --Neurons
            --Layer No : 1
            I_NEURON_1_0: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_INP00_0, S_INP01_0, DataIn, S_NEU00_IC_1, S_NEU00_1, EnN, EnN_Out_1_0);
            I_NEURON_1_1: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_INP00_0, S_INP01_0, S_NEU00_IC_1, S_NEU01_IC_1, S_NEU01_1, EnN,
EnN_Out_1_1);
            I_NEURON_1_2: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_INP00_0, S_INP01_0, S_NEU01_IC_1, S_NEU02_IC_1, S_NEU02_1, EnN,
EnN_Out_1_2);
            I_NEURON_1_3: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_INP00_0, S_INP01_0, S_NEU02_IC_1, S_NEU03_IC_1, S_NEU03_1, EnN,
EnN_Out_1_3);
            --Layer No : 2
            I_NEURON_2_0: entity work.HLMS04 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU03_IC_1, S_NEU00_IC_2,
S_NEU00_2, EnN_Out_1_0, EnN_Out_2_0);
            I_NEURON_2_1: entity work.HLMS04 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU00_IC_2, S_NEU01_IC_2,
S_NEU01_2, EnN_Out_1_0, EnN_Out_2_1);
            --Layer No : 3
            I_NEURON_3_0: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_NEU00_2, S_NEU01_2, S_NEU01_IC_2, S_NEU00_IC_3, OUT00_out, EnN_Out_2_0,
EnN_Out_3_0);
```

```
      RReady <= EnN_Out_3_0;
END ARCHITECTURE RTL;
-------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :     Ibrahim SAHİN-Günay TEMÜR
-- Date      :     Fabruary 2013
-- Design    :     UpCounter
-------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
entity UPCOUNTER is
      generic (WIDTH : integer := 32);
      port (
      RST   : in std_logic;
      CLK   : in std_logic;
      LOAD  : in std_logic;
      INC   : in std_logic;
      DATA  : in std_logic_vector(WIDTH-1 downto 0);
      Q     : out std_logic_vector(WIDTH-1 downto 0));
end entity UPCOUNTER;
architecture RTL of UPCOUNTER is
      signal CNT    : std_logic_vector(WIDTH-1 downto 0);
begin
      process
      begin
            WAIT UNTIL ((CLK'EVENT) AND (CLK= '1'));
            IF (RST = '1') THEN
                  CNT <= (others => '0');
            ELSE
                  IF (Load = '1') THEN
                        CNT <= DATA;
                  ELSE
                        IF (INC = '1') THEN
                              cnt <= cnt + 1;
                        ELSE
                              cnt <= cnt;
                        END IF;
                  END IF;
            END IF;
      end process;
            Q <= CNT; -- type is converted back to std_logic_vector
end architecture RTL;
-------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :     Ibrahim SAHİN-Günay TEMÜR
-- Date      :     Fabruary 2013
-- Design    :     DownCounter
-------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
entity DownCOUNTER is
      generic (WIDTH : integer := 32);
      port (
       RST   : in std_logic;
       CLK         : in std_logic;
       LOAD        : in std_logic;
       Dec   : in std_logic;
       DATA        : in std_logic_vector(WIDTH-1 downto 0);
       Finished    : out std_logic);
end entity DownCOUNTER;
architecture RTL of DownCOUNTER is
      signal CNT   : std_logic_vector(WIDTH-1 downto 0);
      signal Dummy : std_logic_vector(WIDTH-1 downto 0);
begin
```

```
	Dummy <= (others => '0');
	process
	begin
		WAIT UNTIL ((CLK'EVENT) AND (CLK= '1'));
		IF (RST = '1') THEN
			CNT <= (others => '0');
		ELSE
			IF (Load = '1') THEN
				CNT <= DATA;
			ELSE
				IF (Dec = '1') THEN
					cnt <= cnt - 1;
				ELSE
					cnt <= cnt;
				END IF;
			END IF;
		END IF;
	end process;
	process (Cnt,Dummy)
	begin
		if Cnt = Dummy then
			Finished <= '1';
		else
			Finished <= '0';
		end if;
	end process;
end architecture RTL;
-------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name		:	Ibrahim SAHİN-Günay TEMÜR
-- Date		:	Fabruary 2013
-- Design	:	Mux2x1
-------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
entity Mux2x1 is
  generic (WIDTH : integer := 32);
  port (	Ain   : in std_logic_vector (WIDTH-1 downto 0);
	Bin   : in std_logic_vector (WIDTH-1 downto 0);
	Sel   : in std_logic;
	Q     : out std_logic_vector(WIDTH-1 downto 0));
end entity Mux2x1;
architecture RTL of Mux2x1 is
begin
	process (Ain,Bin,Sel)
	begin
		if (Sel = '0') THEN
			Q<= Ain;
		ELSE
			Q<= Bin;
		END IF;
	end process;
end architecture RTL;
-------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name		:	Ibrahim SAHİN-Günay TEMÜR
-- Date		:	Fabruary 2013
-- Design	:	Address Unit
-------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity AddressUnit is
	Generic (WIDTH : integer := 32);
	Port ( Clock : in  STD_LOGIC;
		Reset		: in  STD_LOGIC;
		LInputCnt	: in  STD_LOGIC;
```

```
            LOutputCnt   : in  STD_LOGIC;
            LPcsCnt      : in  STD_LOGIC;
            AddrSel      : in  STD_LOGIC;
            IncInCnt           : in  STD_LOGIC;
            IncOutCnt          : in  STD_LOGIC;
            DecDwnCnt          : in  STD_LOGIC;
            DataIn             : in  STD_LOGIC_VECTOR (Width-1 downto 0);
            AddrOut            : out  STD_LOGIC_VECTOR (Width-1 downto 0);
            Finished           : out  STD_LOGIC);
end AddressUnit;
architecture Behavioral of AddressUnit is
      signal InDataCntOut  : std_logic_vector (width-1 downto 0);
      signal OutDataCntOut : std_logic_vector (width-1 downto 0);
begin
      InputDataCNT : entity work.UPCOUNTER   generic map (Width) port map
(Reset,Clock,LInputCnt,IncInCnt,DataIn,InDataCntOut);
      OuTputDataCNT: entity work.UPCOUNTER   generic map (Width) port map
(Reset,Clock,LOutputCnt,IncOutCnt,DataIn,OutDataCntOut);
      PcsCounter   : entity Work.DownCounter generic map (width) port map
(Reset, clock, LPcsCnt,DecDwnCnt,DataIn,Finished);
      AddrMux            : entity Work.Mux2x1      generic map (width) port
map (InDataCntOut,OutDataCntOut,AddrSel,AddrOut);
end Behavioral;
----------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :     Ibrahim SAHİN-Günay TEMÜR
-- Date      :     Fabruary 2013
-- Design    :     Controller
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY Controller IS
      PORT (
            Clock       : in STD_LOGIC;
            Reset       : in STD_LOGIC;
            Start       : in STD_LOGIC;
            MemBusGrant : in STD_LOGIC;
            Finished    : in STD_LOGIC;
            RReady      : in STD_LOGIC;
            CReset      : out STD_LOGIC;
            LConst      : out STD_LOGIC;
            LVar        : out STD_LOGIC;
            EnN         : out STD_LOGIC;
            MemBusReq   : out STD_LOGIC;
            Strobe      : out STD_LOGIC;
            RW          : out STD_LOGIC;
            LInputCnt   : out STD_LOGIC;
            LOutputCnt  : out STD_LOGIC;
            LPcsCnt     : out STD_LOGIC;
            AddrSel     : out STD_LOGIC;
            IncInCnt    : out STD_LOGIC;
            IncOutCnt   : out STD_LOGIC;
            DecDwnCnt   : out STD_LOGIC;
            Done        : out STD_LOGIC);
END ENTITY Controller;
ARCHITECTURE RTL OF Controller IS
            SIGNAL C_CReset      : STD_LOGIC;
            SIGNAL C_LConst      : STD_LOGIC;
            SIGNAL C_LVar        : STD_LOGIC;
            SIGNAL C_EnN         : STD_LOGIC;
            SIGNAL C_MemBusReq   : STD_LOGIC;
            SIGNAL C_Strobe      : STD_LOGIC;
            SIGNAL C_RW          : STD_LOGIC;
            SIGNAL C_LInputCnt   : STD_LOGIC;
            SIGNAL C_LOutputCnt  : STD_LOGIC;
            SIGNAL C_LPcsCnt     : STD_LOGIC;
```

```
		SIGNAL C_AddrSel     : STD_LOGIC;
		SIGNAL C_IncInCnt    : STD_LOGIC;
		SIGNAL C_IncOutCnt   : STD_LOGIC;
		SIGNAL C_DecDwnCnt   : STD_LOGIC;
		SIGNAL C_Done        : STD_LOGIC;
	type StateType is (Bekle,BusBekle,
	A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,A19,A20,
	B1,B2,B3,
	C1,C2,C3,C4,C5,SON);
	signal CurrentState : StateType;
BEGIN
		FSM : process
		BEGIN
			wait until Clock'EVENT and Clock='1';
				if(Reset='1') then
					CurrentState <= Bekle;
				else
				CASE (CurrentState) IS
					WHEN Bekle =>
						if (Start='0') then
							CurrentState <= Bekle;
						else
							CurrentState <= BusBekle;
						end if;
					WHEN BusBekle =>
						if (MemBusGrant='1') then
							CurrentState <= BusBekle;
						else
							CurrentState <= A1;
						end if;
					WHEN A1 => CurrentState <= A2;
					WHEN A2 => CurrentState <= A3;
					WHEN A3 => CurrentState <= A4;
					WHEN A4 => CurrentState <= A5;
					WHEN A5 => CurrentState <= A6;
					WHEN A6 => CurrentState <= A7;
					WHEN A7 => CurrentState <= A8;
					WHEN A8 => CurrentState <= A9;
					WHEN A9 => CurrentState <= A10;
					WHEN A10 => CurrentState <= A11;
					WHEN A11 => CurrentState <= A12;
					WHEN A12 => CurrentState <= A13;
					WHEN A13 => CurrentState <= A14;
					WHEN A14 => CurrentState <= A15;
					WHEN A15 => CurrentState <= A16;
					WHEN A16 => CurrentState <= A17;
					WHEN A17 => CurrentState <= A18;
					WHEN A18 => CurrentState <= A19;
					WHEN A19 => CurrentState <= A20;
					WHEN A20 => CurrentState <= B1;

					WHEN B1 => CurrentState <= B2;
					WHEN B2 => CurrentState <= B3;
					WHEN B3 => CurrentState <= C1;

					WHEN C1 => CurrentState <= C2;
					WHEN C2 => CurrentState <= C3;
					WHEN C3 => CurrentState <= C4;
					WHEN C4 => CurrentState <= C5;
					WHEN C5 =>
						if (Finished='1') then
							CurrentState <= SON;
						else
							CurrentState <= C1;
						end if;
					WHEN OTHERS  =>
						CurrentState <= Bekle;
				end case;
```

```
                end if;
        end process FSM;

    CLojic : process (CurrentState)
        begin
                case (CurrentState) is
                    WHEN Bekle =>
                            C_CReset      <= '0';
                            C_LConst      <= '0';
                            C_LVar        <= '0';
                            C_EnN         <= '0';
                            C_MemBusReq   <= '1';
                            C_Strobe      <= '0';
                            C_RW          <= '0';
                            C_LInputCnt   <= '0';
                            C_LOutputCnt  <= '0';
                            C_LPcsCnt     <= '0';
                            C_AddrSel     <= '0';
                            C_IncInCnt    <= '0';
                            C_IncOutCnt   <= '0';
                            C_DecDwnCnt   <= '0';
                            C_Done        <= '0';
                            :
                end case;
    end process CLojic;
        CReset       <= C_CReset     ;
        LConst       <= C_LConst     ;
        LVar         <= C_LVar       ;
        EnN          <= C_EnN        ;
        MemBusReq    <= C_MemBusReq  ;
        Strobe       <= C_Strobe     ;
        RW           <= C_RW         ;
        LInputCnt    <= C_LInputCnt  ;
        LOutputCnt   <= C_LOutputCnt ;
        LPcsCnt      <= C_LPcsCnt    ;
        AddrSel      <= C_AddrSel    ;
        IncInCnt     <= C_IncInCnt   ;
        IncOutCnt    <= C_IncOutCnt  ;
        DecDwnCnt    <= C_DecDwnCnt  ;
        Done         <= C_Done       ;
end architecture;
```

EK-2: *NETLİST2* İÇİN OLUŞTURULAN YSA SİSTEM VHDL KODU

NetList2 için üretilmiş YSA sistem VHDL kodları aşağıda verilmiştir. NetList2 için oluşturulan kodların bazıları NetList1 için oluşturulan kodlarla aynı olduğundan bu bölümde tekrar sunulmamışlardır. Değişiklik gösteren ve aşağıda sunulan kodlar sadece YSA veri yolu (DataPath) ve denetleyici (Controller) VHDL kodlarıdır.

```
--------------------------------------------------------------------------------------------------------------------

------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :      Ibrahim SAHİN-Günay TEMÜR
-- Date      :      Fabruary 2013
-- Design    :      DATAPATH
------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY DataPath IS
       PORT (
              Clock : in STD_LOGIC;
              Reset : in STD_LOGIC;
              LConst : in STD_LOGIC;
              LVar : in STD_LOGIC;
              EnN : in STD_LOGIC;
              DataIn : in STD_LOGIC_VECTOR (31 downto 0);
              RReady : out STD_LOGIC;
              OUT00_out : out STD_LOGIC_VECTOR (31 downto 0)
              );
END ENTITY DataPath;
ARCHITECTURE RTL OF DataPath IS
              --Internal data flow signal
              SIGNAL S_INP00_0 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_INP01_0 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU00_1 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU01_1 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU02_1 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU03_1 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU00_2 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU01_2 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU02_2 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU03_2 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU00_3 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU01_3 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU00_4 : STD_LOGIC_VECTOR (31 downto 0);
              --Init Chain "IC" signals
              SIGNAL S_NEU00_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU01_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU02_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU03_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU00_IC_2 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU01_IC_2 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU02_IC_2 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU03_IC_2 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU00_IC_3 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU01_IC_3 : STD_LOGIC_VECTOR (31 downto 0);
              SIGNAL S_NEU00_IC_4 : STD_LOGIC_VECTOR (31 downto 0);
              --EnN Singal Definitions
              SIGNAL EnN_Out_1_0 : STD_LOGIC;
              SIGNAL EnN_Out_1_1 : STD_LOGIC;
              SIGNAL EnN_Out_1_2 : STD_LOGIC;
              SIGNAL EnN_Out_1_3 : STD_LOGIC;
```

```
            SIGNAL EnN_Out_2_0 : STD_LOGIC;
            SIGNAL EnN_Out_2_1 : STD_LOGIC;
            SIGNAL EnN_Out_2_2 : STD_LOGIC;
            SIGNAL EnN_Out_2_3 : STD_LOGIC;
            SIGNAL EnN_Out_3_0 : STD_LOGIC;
            SIGNAL EnN_Out_3_1 : STD_LOGIC;
            SIGNAL EnN_Out_4_0 : STD_LOGIC;
BEGIN
            --Register Instantiations
            I_INPUT_0: entity work.RegGen Generic Map (32) Port Map (Clock,
Reset, LVar, DataIn, S_INP00_0);
            I_INPUT_1: entity work.RegGen Generic Map (32) Port Map (Clock,
Reset, LVar, S_INP00_0, S_INP01_0);
            --Neurons
            --Layer No : 1
            I_NEURON_1_0: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_INP00_0, S_INP01_0, DataIn, S_NEU00_IC_1, S_NEU00_1, EnN, EnN_Out_1_0);
            I_NEURON_1_1: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_INP00_0, S_INP01_0, S_NEU00_IC_1, S_NEU01_IC_1, S_NEU01_1, EnN,
EnN_Out_1_1);
            I_NEURON_1_2: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_INP00_0, S_INP01_0, S_NEU01_IC_1, S_NEU02_IC_1, S_NEU02_1, EnN,
EnN_Out_1_2);
            I_NEURON_1_3: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_INP00_0, S_INP01_0, S_NEU02_IC_1, S_NEU03_IC_1, S_NEU03_1, EnN,
EnN_Out_1_3);
            --Layer No : 2
            I_NEURON_2_0: entity work.HLMS04 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU03_IC_1, S_NEU00_IC_2,
S_NEU00_2, EnN_Out_1_0, EnN_Out_2_0);
            I_NEURON_2_1: entity work.HLMS04 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU00_IC_2, S_NEU01_IC_2,
S_NEU01_2, EnN_Out_1_0, EnN_Out_2_1);
            I_NEURON_2_2: entity work.HLMS04 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU01_IC_2, S_NEU02_IC_2,
S_NEU02_2, EnN_Out_1_0, EnN_Out_2_2);
            I_NEURON_2_3: entity work.HLMS04 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU02_IC_2, S_NEU03_IC_2,
S_NEU03_2, EnN_Out_1_0, EnN_Out_2_3);
            --Layer No : 3
            I_NEURON_3_0: entity work.HLMS04 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU03_IC_2, S_NEU00_IC_3,
S_NEU00_3, EnN_Out_2_0, EnN_Out_3_0);
            I_NEURON_3_1: entity work.HLMS04 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU00_IC_3, S_NEU01_IC_3,
S_NEU01_3, EnN_Out_2_0, EnN_Out_3_1);
            --Layer No : 4
            I_NEURON_4_0: entity work.PLIN02 Port Map (Clock, Reset, LConst,
S_NEU00_1, S_NEU01_1, S_NEU01_IC_3, S_NEU00_IC_4, OUT00_out, EnN_Out_3_0,
EnN_Out_4_0);
      RReady <= EnN_Out_4_0;
END ARCHITECTURE RTL;
-------------------------------------------------------------
-- Copyright 2013 by Duzce University All rights reserved.
-- Name      :     Ibrahim SAHİN-Günay TEMÜR
-- Date      :     Fabruary 2013
-- Design    :     Controller
-------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Controller IS
      PORT (
            Clock       : in STD_LOGIC;
            Reset       : in STD_LOGIC;
            Start       : in STD_LOGIC;
```

```
            MemBusGrant : in STD_LOGIC;
            Finished    : in STD_LOGIC;
            RReady      : in STD_LOGIC;
            CReset      : out STD_LOGIC;
            LConst      : out STD_LOGIC;
            LVar        : out STD_LOGIC;
            EnN         : out STD_LOGIC;
            MemBusReq   : out STD_LOGIC;
            Strobe      : out STD_LOGIC;
            RW          : out STD_LOGIC;
            LInputCnt   : out STD_LOGIC;
            LOutputCnt  : out STD_LOGIC;
            LPcsCnt     : out STD_LOGIC;
            AddrSel     : out STD_LOGIC;
            IncInCnt    : out STD_LOGIC;
            IncOutCnt   : out STD_LOGIC;
            DecDwnCnt   : out STD_LOGIC;
            Done        : out STD_LOGIC
            );
END ENTITY Controller;
ARCHITECTURE RTL OF Controller IS
            SIGNAL C_CReset      : STD_LOGIC;
            SIGNAL C_LConst      : STD_LOGIC;
            SIGNAL C_LVar        : STD_LOGIC;
            SIGNAL C_EnN         : STD_LOGIC;
            SIGNAL C_MemBusReq   : STD_LOGIC;
            SIGNAL C_Strobe      : STD_LOGIC;
            SIGNAL C_RW          : STD_LOGIC;
            SIGNAL C_LInputCnt   : STD_LOGIC;
            SIGNAL C_LOutputCnt  : STD_LOGIC;
            SIGNAL C_LPcsCnt     : STD_LOGIC;
            SIGNAL C_AddrSel     : STD_LOGIC;
            SIGNAL C_IncInCnt    : STD_LOGIC;
            SIGNAL C_IncOutCnt   : STD_LOGIC;
            SIGNAL C_DecDwnCnt   : STD_LOGIC;
            SIGNAL C_Done        : STD_LOGIC;
            type StateType is (Bekle,BusBekle,
      A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,A19,A20,A
21,A22,A23,A24,A25,A26,A27,A28,A29,A30,A31,A32,A33,A34,A35,A36,
                                                B1,B2,B3,
                                                C1,C2,C3,C4,C5,SON);
            signal CurrentState : StateType;
BEGIN
            FSM : process
            BEGIN
                  wait until Clock'EVENT and Clock='1';
                        if(Reset='1') then
                              CurrentState <= Bekle;
                        else
                        CASE (CurrentState) IS
                              WHEN Bekle =>
                                    if (Start='0') then
                                          CurrentState <= Bekle;
                                    else
                                          CurrentState <= BusBekle;
                                    end if;
                              WHEN BusBekle =>
                                    if (MemBusGrant='1') then
                                          CurrentState <= BusBekle;
                                    else
                                          CurrentState <= A1;
                                    end if;
                              WHEN A1 => CurrentState <= A2;
                              WHEN A2 => CurrentState <= A3;
                              WHEN A3 => CurrentState <= A4;
                              WHEN A4 => CurrentState <= A5;
                              WHEN A5 => CurrentState <= A6;
                              WHEN A6 => CurrentState <= A7;
```

```
                        WHEN A7 => CurrentState <= A8;
                        WHEN A8 => CurrentState <= A9;
                        WHEN A9 => CurrentState <= A10;
                        WHEN A10 => CurrentState <= A11;
                        WHEN A11 => CurrentState <= A12;
                        WHEN A12 => CurrentState <= A13;
                        WHEN A13 => CurrentState <= A14;
                        WHEN A14 => CurrentState <= A15;
                        WHEN A15 => CurrentState <= A16;
                        WHEN A16 => CurrentState <= A17;
                        WHEN A17 => CurrentState <= A18;
                        WHEN A18 => CurrentState <= A19;
                        WHEN A19 => CurrentState <= A20;
                        WHEN A20 => CurrentState <= A21;
                        WHEN A21 => CurrentState <= A22;
                        WHEN A22 => CurrentState <= A23;
                        WHEN A23 => CurrentState <= A24;
                        WHEN A24 => CurrentState <= A25;
                        WHEN A25 => CurrentState <= A26;
                        WHEN A26 => CurrentState <= A27;
                        WHEN A27 => CurrentState <= A28;
                        WHEN A28 => CurrentState <= A29;
                        WHEN A29 => CurrentState <= A30;
                        WHEN A30 => CurrentState <= A31;
                        WHEN A31 => CurrentState <= A32;
                        WHEN A32 => CurrentState <= A33;
                        WHEN A33 => CurrentState <= A34;
                        WHEN A34 => CurrentState <= A35;
                        WHEN A35 => CurrentState <= A36;
                        WHEN A36 => CurrentState <= B1;

                        WHEN B1 => CurrentState <= B2;
                        WHEN B2 => CurrentState <= B3;
                        WHEN B3 => CurrentState <= C1;

                        WHEN C1 => CurrentState <= C2;
                        WHEN C2 => CurrentState <= C3;
                        WHEN C3 => CurrentState <= C4;
                        WHEN C4 => CurrentState <= C5;
                        WHEN C5 =>
                              if (Finished='1') then
                                    CurrentState <= SON;
                              else
                                    CurrentState <= C1;
                              end if;
                        WHEN OTHERS  =>
                              CurrentState <= Bekle;
                  end case;
            end if;
      end process FSM;
CLojic : process (CurrentState)
      begin
            case (CurrentState) is
                  WHEN Bekle =>
                        C_CReset      <= '0';
                        C_LConst      <= '0';
                        C_LVar        <= '0';
                        C_EnN         <= '0';
                        C_MemBusReq   <= '1';
                        C_Strobe      <= '0';
                        C_RW          <= '0';
                        C_LInputCnt   <= '0';
                        C_LOutputCnt  <= '0';
                        C_LPcsCnt     <= '0';
                        C_AddrSel     <= '0';
                        C_IncInCnt    <= '0';
                        C_IncOutCnt   <= '0';
                        C_DecDwnCnt   <= '0';
```

```
                    C_Done        <= '0';

                                  :
                    end case;
end process CLojic;
CReset       <= C_CReset     ;
LConst       <= C_LConst     ;
LVar         <= C_LVar       ;
EnN          <= C_EnN        ;
MemBusReq    <= C_MemBusReq  ;
Strobe       <= C_Strobe     ;
RW           <= C_RW         ;
LInputCnt    <= C_LInputCnt  ;
LOutputCnt   <= C_LOutputCnt ;
LPcsCnt      <= C_LPcsCnt    ;
AddrSel      <= C_AddrSel    ;
IncInCnt     <= C_IncInCnt   ;
IncOutCnt    <= C_IncOutCnt  ;
DecDwnCnt    <= C_DecDwnCnt  ;
Done         <= C_Done       ;
                 end architecture;
```

ÖZGEÇMİŞ

Kişisel Bilgiler

Soyadı, Adı : TEMÜR, Günay

Uyruğu : T.C

Doğum Tarihi Ve Yeri : 08.02.1983 Cide/KASTAMONU

E-mail : gunaytemur@duzce.edu.tr

Eğitim

Derecesi	**Eğitim Birimi**	**Mezuniyet Tarihi**
Lisans	Süleyman Demirel Üniversitesi/ Teknik Eğitim Fak. Bil. Eğt. Böl./ Bilgisayar Sistemleri Öğretmenliği	2006
Lise	Sakarya Teknik Lisesi	2001

İş Deneyimi

Yıl	**Yer**	**Görev**
2009-……	Düzce Üniversitesi-Kaynaşlı MYO	Öğretim Görevlisi
2007-2009	Cumhuriyet İlköğretim Okulu/Hatay	Bilgisayar Öğretmeni
2007-2007	Şeker İlköğretim Okulu/Sakarya	Bilgisayar Öğretmeni

Yayınlar

1. Şafak İ.T., Bilgiç İ., Soy A., **Temür G.** (2012). Uluslararası Karayolu Taşımacılık Faaliyeti Gösteren Firmaların Yenileşim ve AR-GE Yaklaşımları: Düzce İli Örneği, 1. Ulusal Ünye İşletmecilik Sempozyumu, 24-25 Nisan 2012

2. Soy A., **Temür G.**, Bilgiç İ, Şafak İ.T. (2012). Kalite Güvencesi Yönetimi Çerçevesinde Lojistik Ve Dış Ticaret Eğitiminin Kalitesini Artırmaya Yönelik Çalışma Düzce Üniversitesi Kaynaşlı Meslek Yüksekokulu Örneği, 1. Ulusal Lojistik ve Tedarik Zinciri Kongresi, 10-12 Mayıs 2012

3. Şafak İ.T., Bilgiç İ., Soy A., **Temür G.** (2012). Düzce İlinde Faaliyet Gösteren Uluslararası Karayolu Taşımacılık Firmalarının Kurumsallaşma Sürecinde Yenileşim Yaklaşımları, Sorunlar ve Çözüm Önerileri, 1. Ulusal Lojistik ve Tedarik Zinciri Kongresi, 10-12 Mayıs 2012