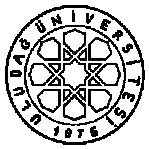
T.C.
ULUDAĞ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

FPGA İLE NLMS VE GAUSS-SEİDEL ALGORİTMALARININ HİBRİT FORMDA GERÇEKLENMESİ

Sedat TİRYAKİ

YÜKSEK LİSANS TEZİ
ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

BURSA-2008



T.C.
ULUDAĞ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

FPGA İLE NLMS VE GAUSS-SEİDEL ALGORİTMALARININ HİBRİT FORMDA GERÇEKLENMESİ

Sedat TİRYAKİ

Yrd.Doç.Dr. Osman Hilmi KOÇAL
(Danışman)

YÜKSEK LİSANS TEZİ
ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

BURSA-2008

T.C.
ULUDAĞ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

FPGA İLE NLMS VE GAUSS-SEİDEL ALGORİTMALARININ HİBRİT FORMDA GERÇEKLENMESİ

Sedat TİRYAKİ

YÜKSEK LİSANS TEZİ
ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

Bu Tez ..../...../200... tarihinde aşağıdaki jüri tarafından oybirliği/oy çokluğu ile kabul edilmiştir.

Yrd.Doç.Dr.Osman Hilmi Koçal
Danışman

ÖZET

Bu çalışmada, uyarlamalı süzgeç katsayılarını ayarlamak için GS (Gauss-Seidel) ve NLMS (Normalized Least Mean Squares) algoritmalarının birlikte kullanıldığı hibrit bir algoritma önerilmiş ve önerilen yeni algoritmanın yakınsama hızı ve işlem karmaşıklığı incelenmiştir. Önerilen algoritma yapılan bir benzetim çalışmasıyla yakınsama hızı ve işlem yükü açısından benzer algoritmalarla karşılaştırmalı olarak incelenmiştir.

Hibrit formda gerçeklenen algoritma, GS algoritması ve NLMS algoritması VHDL(VHSIC Hardware Description Language) kullanılarak donanımsal olarak tanımlanmıştır. Her üç algoritmanın VHDL tanımları ile sentezleme ve benzetimleri yapılarak algoritmaların FPGA üzerinde kapladıkları alan ve maksimum saat frekansı gibi parametreleri belirlenmiştir.

Elde edilen sonuçlara göre, önerilen hibrit algoritmanın bir ara yöntem olarak işlem karmaşıklığı veya yakınsama hızı açısından diğer algoritmalara iyi bir alternatif olduğu görülmüştür.

Anahtar Kelimeler: Uyarlamalı süzgeç, Gauss-Seidel algoritması, Normalleştirilmiş En Küçük Ortalamalar Karesi algoritması, Kanal Dengeleyici, Fpga, Vhdl.

ABSTRACT

In this thesis, a hybrid algorithm based on the use of GS(Gauss-Seidel) and NLMS(Normalized Least Mean Squares) algorithms together is introduced for adjusting adaptive filter coefficients and also convergence rate and computational complexity of the proposed algorithm is studied. The proposed algorithm is compared with similar algorithms by viewpoints of computational complexity and convergence rate via an adaptive channel equalizer example.

The proposed hybrid algorithm has been designed as hardware via VHDL as well as NLMS and Gauss-Seidel algorithms. All three designs has been synthesized and simulated to determine FPGA area and maximum clock frequency parameters.

According to the results obtained, it is shown that the proposed hybrid algorithm is a good alternative to the others as an intermediate method by viewpoints of computational complexity or convergence rate.

Keywords: Adaptive filters, Gauss-Seidel algorithm, Normalized Least Mean Squares Algorithm, Channel Equalization, Fpga, Vhdl.

İÇİNDEKİLER

KISALTMALAR DİZİNİ

AP: Affine Projection

CMOS: Complementary metal oxide semiconductor

EDS:Euclidean Direction Search

EPROM: Erasable Programmable Read Only Memory

FIR: Finite Impulse Response

FPGA: Field Programmable Gate Array

FPU: Floating point unit

GS: Gauss-Seidel

LMS: Least mean squares

LUT: Look Up Table

NLMS:Normalized Least Mean Squares

RLS:Recursive Leasst Squares

SRAM:Static Random Access Memory

VHDL: VHSIC Hardware Description Language

ŞEKİLLER DİZİNİ

ÇİZELGELER DİZİNİ

1. GİRİŞ

Uyarlamalı süzgeç katsayılarının ayarlamasında kullanılan algoritmalar eğim tabanlı algoritmalar ve en küçük kareler tabanlı algoritmalar olmak üzere iki temel grupta ele alınabilirler (Diniz, 1987, Farhang-Boroujeny, 1998, Haykin, 2002). Eğim tabanlı algoritmalar düşük işlem yükü avantajına sahip olup yüksek örnekleme hızlarında ve çoğunlukla uyarlamalı sinyal işleme ve haberleşme uygulamalarında tercih edilmektedir. En sık kullanılan eğim tabanlı algoritmalar LMS (Least Mean Squares), NLMS (Normalized LMS) ve AP (Affine Projection) algoritmalarıdır (Haykin, 2002, Haykin ve Widrow, 2003). Fakat bu algoritmalar giriş sinyaline ait korelasyon matrisinin özdeğer yayılımına da bağlı olarak yavaş yakınsama hızı dezavantajına sahiptir. En küçük kareler tabanlı algoritmalar ise yüksek işlem yüküne rağmen yakınsama özelliklerinin eğim tabanlı algoritmalara göre çok iyi olmasından dolayı tercih edilmektedir. RLS (Recursive Least Squares) algoritması bu gruba girmektedir. Fakat yüksek örnekleme hızı gerektiren durumlar yoğun işlem yükünden dolayı bu algoritmaların kullanım alanlarını sınırlamaktadır.

Diğer taraftan, bu iki algoritma grubuna alternatif olarak yüksek yakınsama hızı ve en küçük kareler tabanlı algoritmalara oranla daha az işlem yükü olan bazı algoritmalar da önerilmiş ve kullanılmıştır (Koçal, 1998, Bose, 2004). Bu algoritmalar Wiener-Hopf denkleminin GS (Gauss-Seidel) iterasyonlarıyla çözümü üzerine kurulu olup, yakınsama hızı açısından giriş sinyaline ait korelasyon matrisinin özdeğer yayılımına bağımlı olmasına rağmen eğim tabanlı algoritmalara oranla daha yüksek bir yakınsama hızına sahiptir ve korelasyon matrisinin özdeğer yayılımının küçük olması durumunda RLS algoritmasına çok yakın sonuçlar vermektedir (Koçal, 1998). Ayrıca işlem yükü RLS algoritmasına göre daha azdır, fakat LMS grubu algoritmalardan fazladır. Bu çalışmanın

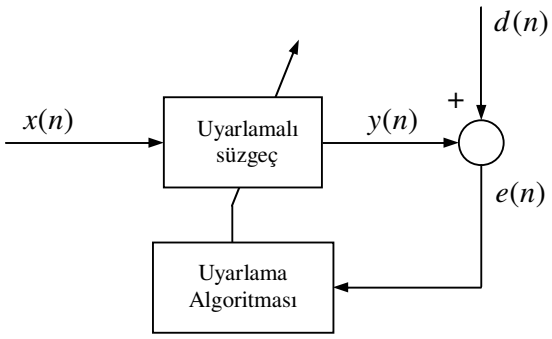
amacı, düşük işlem yüküne sahip eğim tabanlı algoritmalar ile bu algoritmalara oranla daha hızlı yakınsama özelliğine sahip olan GS algoritmasının birlikte kullanarak uyarlamalı süzgeçlerde istenen özellikleri ön plana çıkaran yeni bir algoritma elde etmektir.

Bu çalışmada, düşük işlem yüküne sahip olan eğim tabanlı NLMS algoritması ile bu algoritmaya göre yakınsama hızı daha iyi olan GS algoritması birlikte kullanılarak hibrit bir algoritma elde edilmiştir. Yapılan bir uyarlamalı kanal dengeleyici uygulamasıyla önerilen algoritmanın yakınsama özellikleri ve işlem yükü benzer algoritmalarla karşılaştırmalı olarak incelenmiştir.Ayrıca NLMS, Gauss-Seidel ve hibrit algoritmanın donanım mimarisi VHDL kullanılarak oluşturulmuştur. Uyarlamalı kanal dengeleyici uygulaması yardımıyla tasarlanan donanımların FPGA üzerinde kapladığı alan ve hız parametreleri her bir algoritma için ayrı ayrı belirlenmiştir. Son olarak elde edilen sonuçların doğrulanması için tasarımların benzetimleri yapılmıştır.

Bu tez aşağıdaki bölümlerden oluşmaktadır. Bölüm 2' de uyarlamalı süzgeçler ve katsayılarının ayarlanmasında kullanılan mevcut bazı algoritmalar tanıtılacaktır. Bölüm 3' te önerilen hibrit algoritma tanıtıldıktan sonra Matlab ortamında elde edilen benzetim sonuçları sunulacaktır. 4. bölümde FPGA yapıları hakkında bilgi verilecektir. Son bölümde ise Gauss-Seidel, NLMS ve hibrit algoritmaların VHDL tasarımları tanıtıldıktan sonra bu tasarımların sentezlenmesi sonucu elde edilen alan ve hız parametreleri ile birlikte benzetim sonuçları verilecektir.

2. UYARLAMALI SÜZGEÇLER

Uyarlamalı süzgeçler transfer fonksiyonunu bir uyarlama algoritması yardımıyla kendisi ayarlayan elektronik süzgeç türüdür.Uyarlamalı süzgeç blok şeması Şekil 2.1' de verilmiştir.



Şekil 2.1 :
Uyarlamalı süzgeç blok şeması

Burada $x(n)$ uyarlamalı süzgecin ayrık-zamanlı giriş sinyali, $y(n)$ uyarlamalı süzgecin tahmin edilen parametreler kullanılarak hesaplanan çıkış sinyali, $d(n)$ ise uyarlamalı süzgecin takip etmesi istenen sinyaldir. Parametre ayarlama algoritmalarıyla, tahmin edilen parametreleri kullanarak hesaplanan $y(n)$ sinyali ile uyarlamalı süzgecin izlemesi istenen $d(n)$ sinyali arasında $e(n) = d(n) - y(n)$ olarak tanımlanan hata sinyalinin karesinin beklenen değerini minimum yapan parametre tahminleri iteratif olarak hesaplanır (Widrow ve Stearns, 1985, Treichler ve diğ., 1987, Diniz, 1997, Farhang-Boroujeny, 1998, Haykin, 2002).

Uyarlama algoritmaları uyarlamalı süzgeçlerin yakınsama hızı ve işlem yükü gibi karakteristik özelliklerini belirler. Geliştirilecek olan hibrit algoritma LMS,

NLMS ve Gauss-Seidel algoritmalarına dayandığı için bu bölümün alt bölümlerinde sadece bu algoritmalar tanıtılmıştır.

2.1 LMS Algoritması

LMS algoritması, azalan adım optimizasyon yönteminin tahmini bir versiyonudur. Azalan adım yönteminde optimum parametreleri bulmak için, öncelikle başlangıç değerinden başlanarak bir sonraki parametre değerinin hesaplanmasında bir sonraki adımın yönü ve büyüklüğü belirlenir. Bir sonraki adımın yönü ortalama karesel hata fonksiyonunun türeviyle belirlenir. Ortalama karesel hata fonksiyonunu minimum yapan azalan adım optimizasyon yöntemi

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu[\mathbf{p} - \mathbf{R}\mathbf{w}(n)] \tag{2.1}$$

olarak verilmektedir (Diniz, 1997, Haykin, 2002, Bose, 2004). Burada μ seçilen yönde gidilecek adım boyutunu, $\mathbf{R}$ uyarlamalı süzgeçin giriş sinyalinin otokorelasyon matrisini ve $\mathbf{p}$ ise uyarlamalı süzgecin takip etmesi gereken sinyal ile giriş sinyali arasındaki çapraz-korelasyon vektörünü göstermektedir ve sırasıyla

$$\mathbf{R} = E[\mathbf{x}(n)\mathbf{x}^T(n)] \quad , \quad \mathbf{p} = E[\mathbf{x}(n)d(n)] \tag{2.2}$$

olarak tanımlanmaktadır, Burada E beklenen değer operatörüdür. Uyarlamalı süzgecin uzunluğu M olmak üzere uyarlamalı FIR (Finite Impulse Response) süzgecin giriş işareti vektörü $\mathbf{x}(n)$ ve katsayı vektörü $\mathbf{w}(n)$

$$\mathbf{x}^T(n) = [x(n) \quad x(n-1) \quad \cdots \quad x(n-M+1)] \tag{2.3}$$

$$\mathbf{w}^T(n) = [w_1(n) \quad w_2(n) \quad \cdots \quad w_M(n)] \tag{2.4}$$

olarak tanımlanmaktadır. Pratik uygulamalarda, $\mathbf{R}$ korelasyon matrisi ve $\mathbf{p}$ korelasyon vektörünün tam değeri bilinemediği için tahmin edilmiş değeri kullanılır. Azalan adım optimizasyon algoritmasında $\mathbf{R}$ ve $\mathbf{p}$ matrislerinin anlık tahminleri

$$\mathbf{R}(n) \cong \mathbf{x}(n)\mathbf{x}^T(n) \quad , \quad \mathbf{p}(n) \cong \mathbf{x}(n)d(n) \tag{2.5}$$

şeklinde kullanılarak elde edilen aşağıdaki algoritma LMS algoritması olarak bilinmektedir (Widrow ve Stearns, 1985, Diniz, 1997, Farhang-Boroujeny, 1998, Haykin, 2002, Bose, 2004).

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\,\mathbf{x}(n)e(n) \tag{2.6}$$

LMS algoritmasının en önemli avantajı algoritmanın gerçeklenmesinde $(2M+1)$ çarpma işlemi yapılmasıdır. LMS algoritması tahminlerinin kararlı bir şekilde optimum parametre değerlerine yakınsayabilmesi için μ adım parametresi

$$0 < \mu < 2/\lambda_{\max} \tag{2.7}$$

aralığında seçilmelidir. Burada $\lambda_{\max}$ korelasyon matrisinin en büyük özdeğeridir.

LMS algoritmasındaki en önemli eksikliklerinden biri algoritmanın kararlı kalmasını sağlayacak kadar küçük ve aynı zamanda parametre tahminlerinin kısa zamanda optimum değerine yakınsamasını sağlayacak kadar da büyük bir μ adım parametresi seçiminin zorluğudur.Bu problemin üzerinden gelinebilmek için NLMS algoritması geliştirilmiştir.

2.2 NLMS Algoritması

LMS algoritmasının, giriş işaretinin süzgeç uzunluğu kadar kısmının gücü ile normalize edilmesiyle NLMS algoritması elde edilir. NLMS algoritması

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \frac{\mu\,\mathbf{x}(n)e(n)}{\alpha + \mathbf{x}^T(n)\mathbf{x}(n)} \tag{2.8}$$

olarak yazılabilir, burada paydada olası bir sıfıra bölme işlemine engel olması için küçük bir $\alpha > 0$ katsayısı kullanılır. NLMS algoritması tahminlerinin kararlı bir şekilde optimum parametre değerlerine yakınsayabilmesi için μ adım parametresi

$$0 < \mu < 2 \tag{2.9}$$

aralığında seçilmelidir (Treichler ve diğ., 1987, Diniz, 1997, Farhang-Boroujeny, 1998, Haykin, 2002, Haykin ve Widrow, 2003). Her adımda yapılan normalizasyon işlemi μ adım parametresinin $\lambda_{\max}$ özdeğerine olan bağımlılığını ortadan kaldırır. Böylece μ adım parametresinin sayısal değeri algoritmanın daha hızlı yakınsayabilmesi için rahatlıkla büyük seçilebilir. NLMS algoritmasının gerçeklenmesinde yapılan normalizasyon işleminden dolayı çarpma sayısı ilk bakışta $(3M + 2)$ gibi görünse de, $\mathbf{x}^T(n)\mathbf{x}(n)$ ifadesinin değerinin hesaplanması için gereken işlem sayısı M 'den 1' e düşürülebilir. Burada NLMS algoritmasındaki $\mathbf{x}^T(n)\mathbf{x}(n)$ çarpımı

$$\mathbf{x}^T(n)\mathbf{x}(n) = x^2(n) + x^2(n-1) + \cdots + x^2(n-M+1) \tag{2.10}$$

şeklinde yazılabilir. Buradan $\mathbf{x}^T(n+1)\mathbf{x}(n+1)$ çarpımı

$$\mathbf{x}^T(n+1)\mathbf{x}(n+1) = \mathbf{x}^T(n)\mathbf{x}(n) + x^2(n+1) - x^2(n-M+1) \tag{2.11}$$

şeklinde, daha önce hafızada saklanan toplama bir sonraki adımda sadece $x^2(n+1)$ çarpımı ilave edilip önceden hesaplanmış olan son terim $x^2(n-M+1)$ çıkarılarak elde edilebilir. Yani sadece $x^2(n+1)$ çarpımından dolayı çarpma sayısı 1 olur (Treichler ve diğ., 1987). Bu durumda NLMS algoritmasının gerçeklenmesi için yapılan çarpma sayısı LMS algoritmasına göre 2 artarak $(2M + 3)$ olmaktadır.

2.3 Gauss-Seidel Algoritması

GS(Gauss-Seidel) algoritmasındaki orjinal fikir, en küçük kareler tahminlerinin elde edilmesinde, lineer denklem takımlarının çözümünde kullanılan iteratif yöntemlerden yararlanmaktır. GS algoritması, uyarlamalı süzgeç katsayılarının ayarlamak için $\mathbf{R}\mathbf{w}_{opt} = \mathbf{p}$ ile verilen normal denklemin çözümünde kullanılmaktadır. Burada $\mathbf{w}_{opt}$ M adet süzgeç katsayısını içeren $M \times 1$ boyutlu optimum katsayı vektörüdür. $\mathbf{R}$ matrisi ve $\mathbf{p}$ vektörü (2) ile verilmiştir. Bilindiği gibi $\mathbf{R}$ korelasyon matrisi simetrik ve pozitif tanımlı olan bir matristir (Haykin, 2002). Nümerik olarak, doğrusal denklem takımını oluşturan kare matrisin simetrik ve pozitif tanımlı olması durumunda, GS algoritmasının herhangi bir başlangıç değeri için denklem takımını sağlayan çözüm değerine iteratif olarak yakınsadığı matematiksel olarak Golub ve Van Loan (1996) tarafından gösterilmiştir.

GS algoritmasıyla tekrarlamalı parametre tahmin işleminin başlangıç noktası, zaman ortalamalı normal denklemin GS algoritmasıyla çözümü üzerine kuruludur. Çünkü pratik uygulamalarda $\mathbf{R}$ matrisinin ve $\mathbf{p}$ vektörünün değerinin bilinmemesi durumunda tahmin edilmiş değerleri kullanılarak bir yaklaşıklık yapılır. GS algoritması $\mathbf{R}$ korelasyon matrisinin ve $\mathbf{p}$ korelasyon vektörünün birikimli tahmini değerini kullanır. Bu yaklaşık tahmini değerler n adet veri grubu kullanıldığında aşağıdaki gibi yazılabilir,

$$\mathbf{R}(n) \cong \frac{1}{n}\sum_{k=1}^{n}\mathbf{x}(k)\mathbf{x}^T(k) \quad , \quad \mathbf{p}(n) \cong \frac{1}{n}\sum_{k=1}^{n}\mathbf{x}(k)d(k) \tag{2.12}$$

veya $1/n$ çarpanı göz önüne alınmadan veri örnekleri alındıkça iteratif olarak

$$\mathbf{R}(n) = \mathbf{R}(n-1) + \mathbf{x}(n)\mathbf{x}^T(n) \quad , \quad \mathbf{p}(n) = \mathbf{p}(n-1) + \mathbf{x}(n)d(n) \tag{2.13}$$

şeklinde güncellenebilir. Daha sonra bir adımlık GS iterasyonu $\mathbf{R}(n)\mathbf{w}(n) = \mathbf{p}(n)$ olarak yazılan zaman ortalamalı normal denklemin çözümünde

$$\mathbf{w}(n+1) = -\left(\mathbf{R}_L(n) + \mathbf{R}_D(n)\right)^{-1}\mathbf{R}_U(n)\mathbf{w}(n) + \left(\mathbf{R}_L(n) + \mathbf{R}_D(n)\right)^{-1}\mathbf{p}(n) \tag{2.14}$$

şeklinde kullanılır, yani iterasyon indisi ayrık zaman indisi olarak alınmıştır. Burada $\mathbf{R}_L(n)$, $\mathbf{R}_D(n)$ ve $\mathbf{R}_U(n)$ sırasıyla korelasyon matrisinin (13) ile verilen tahmini değerinin alt üçgen, köşegen ve üst üçgen kısımlarını içeren kare matrisleri gösterir. Burada (14) ile verilen eşitlikte iki işlem birleştirilmiştir. Bunlar, korelasyon matrisi ile korelasyon vektörünün tahmin edilmesi ve GS algoritması ile zaman ortalamalı normal denklemin çözümüdür (Koçal,1998). Pratik uygulamada süzgeç katsayıları

$$w_i(n+1) = \left[p_i(n) - \sum_{j=1}^{i-1} R_{ij}(n) w_j(n+1) - \sum_{j=i+1}^{M} R_{ij}(n) w_j(n) \right] \Big/ R_{ii}(n) \quad , \quad (i = 1,2,\ldots,M) \tag{2.15}$$

şeklinde sırayla güncellenir. Burada $R_{ij}(n)$ korelasyon matrisinin *i.* satırına ve *j.* sütununa denk düşen elemanını gösterir, $p_i(n)$ korelasyon vektörünün *i.* elemanını gösterir ve $w_i(n)$ katsayı vektörünün *i.* elemanını gösterir. Ayrıca, uyarlamalı FIR süzgeç kullanıldığında ve korelasyon matrisinin simetrik olduğu göz önüne alındığında aynı iterasyonu giriş işaretinin korelasyon katsayılarına bağlı olarak

$$w_i(n+1) = \left[p_i(n) - \sum_{j=1}^{i-1} r_{i-j}(n) w_j(n+1) - \sum_{j=i+1}^{M} r_{j-i}(n) w_j(n) \right] \Big/ r_0(n) \quad , \quad (i = 1,2,\ldots,M) \tag{2.16}$$

şeklinde de yazılabilir. Burada $\mathbf{R}(k)$ korelasyon matrisinin elemanlarını oluşturan oto-korelasyon katsayılarını ve $\mathbf{p}(k)$ korelasyon vektörünün elemanlarını oluşturan çapraz-korelasyon katsayıları sırasıyla aşağıdaki gibi tanımlanabilir.

$$r_i = E[x(n)x(n-i)] \quad , \quad (i = 0,1,\ldots,M-1) \tag{2.17}$$

$$p_i = E[d(n)x(n-i)] \quad , \quad (i = 1,2,\ldots,M) \tag{2.18}$$

Pratik uygulamalarda bu katsayıların tahmini değerleri veri örnekleri alındıkça iteratif olarak $1/n$ çarpanı göz önüne alınmadan aşağıdaki gibi güncellenebilir.

$$r_i(n) = r_i(n) + x(n)x(n-i) \quad , \quad (i = 0,1,\ldots,M-1) \tag{2.19}$$

$$p_i(n) = p_i(n) + d(n)x(n-i) \quad , \quad (i = 1,2,\ldots,M) \tag{2.20}$$

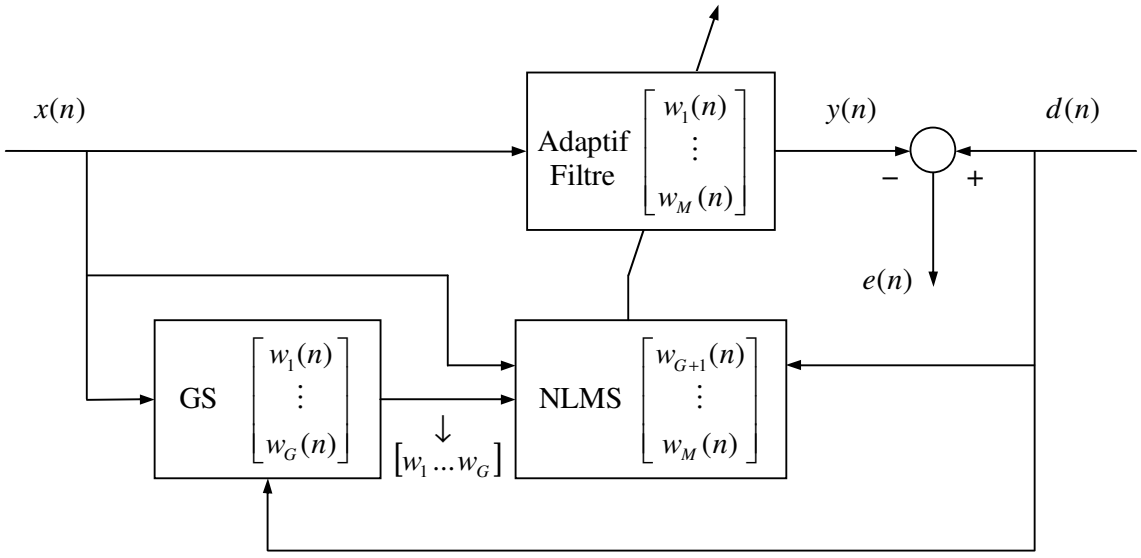
Burada sırasıyla (19) ve (20) ile verilen korelasyon katsayıları tahminlerinin (16) ile birlikte kullanıldığında elde edilen algoritma stokastik GS algoritması olarak adlandırılmıştır (Koçal, 2008). Bu algoritmanın gerçeklenmesinde $(M^2 + 2M)$ adet çarpma işlemi yapılmaktadır. Korelasyon matrisinin (13) ile verilen tahmin edilmiş değerinin pozitif tanımlı olması durumunda stokastik GS algoritması tahminleri optimum değerine yakınsamaktadır (Hatun ve Koçal, 2005).

GS algoritması farklı bir bakış açısıyla bir optimizasyon algoritması şeklinde EDS (Euclidean Direction Search) adı ile de önerilmiş ve bazı uyarlamalı sinyal işleme uygulamalarında kullanılmıştır (Xu ve diğ. 1998, 1999a, 1999b, Bose ve Xu, 2002, Mabey ve diğ. 2004, Bose, 2004). Bu çalışmalarda Gauss-Seidel algoritması uyarlamalı FIR süzgeç katsayılarının ayarlanmasında kullanılmıştır. Önerilen bu GS tabanlı algoritmaların işlem yükü RLS algoritmasından daha az olup, yakınsama hızı RLS algoritmasına yakındır ve korelasyon matrisinin özdeğer yayılımının küçük olması durumunda RLS algoritmasına çok yakın sonuçlar vermektedir (Koçal, 1998, Bose, 2004).

3. HİBRİT GS-NLMS ALGORİTMASI

GS algoritmasının yakınsama hızı RLS algoritmasına yakın olmakla birlikte, gerçekleme esnasında $(M^2 + 2M)$ adet çarpma işlemi yapılması algoritmanın yüksek örnekleme hızlarında kullanılması durumunda dezavantaj oluşturmaktadır. Bu çalışmada, yakınsama hızı iyi olan GS algoritmasıyla işlem yükü az olan NLMS algoritmaları birlikte kullanılarak işlem yükü GS algoritmasından daha az olan ve NLMS algoritmasından daha hızlı yakınsayan hibrit bir algoritma elde edilmiştir. Önerilen hibrit GS-NLMS algoritmasının adaptif filtreleme işleminde kullanımı Şekil 3.1'deki blok diyagramda görülmektedir.

Önerilen hibrit GS-NLMS algoritmasında, Şekil 3.1'de de gördüğü gibi, M uzunluklu bir adaptif filtrenin katsayılarının $1 \le G < M - 1$ olmak üzere G tanesi GS algoritması ile güncellenmekte ve geriye kalan $(M - G)$ tanesi de NLMS algoritması ile güncellenmektedir. Hibrit GS-NLMS algoritmasında bir önceki adımda güncellenmiş katsayı vektörünün G elemanı GS algoritması kullanılarak güncellenir. NLMS algoritmasında ise öncelikle G elemanı güncellenmiş katsayı vektörü kullanılarak anlık hata bilgisi hesaplanır, sonra hesaplanan anlık hata bilgisi kullanılarak katsayı vektörünün geriye kalan $(M - G)$ tane elemanı güncellenir. Elde edilen yeni hibrit algoritmanın işlem yükü Eşitlik 3.11' de verilmiştir. Görülebileceği gibi önerilen hibrit algoritma işlem yükü açısından GS ile NLMS arasında kalmaktadır. Ayrıca, önerilen hibrit GS-NLMS algoritmasının işlem yükü, GS ve NLMS algoritmaları arasında paylaşılan parametre sayısına bağlı olarak değişmektedir. Önerilen hibrit GS-NLMS algoritmasının gerçeklenmesi ve işlem yükü aşağıda detaylı olarak açıklanmaktadır.



Şekil 3.1:
Hibrit GS-NLMS algoritması

Önerilen hibrit algoritma iki kısımdan oluşmaktadır. Bu durumda ilk kısımda kullanılan GS algoritmasının kararlı olması için (2.13) eşitliğindeki gibi hesaplanan korelasyon matrisi tahminlerinin pozitif tanımlı olması durumunda ve aynı zamanda ikinci kısımda kullanılan NLMS algoritmasının kararlı olması için kullanılan adım parametresinin $0 < \mu < 2$ aralığında algoritma kararlı olacak şekilde seçilmesi durumunda hibrit GS-NLMS algoritması da kararlı olacaktır.

3.1 Hibrit GS-NLMS Algoritmasının Gerçeklenmesi ve İşlem Yükü

Bu kısımda, GS ve NLMS algoritmalarında hesaplanan benzer ifadeler arasındaki bağlantılar belirlenerek hibrit GS-NLMS algoritmasının işlem yükünün azaltılması için gerekli çalışmalar yapılmıştır.

Önce GS algoritmasında kullanılan korelasyon katsayılarıyla NLMS algoritmasında hesaplanan $\mathbf{x}^T(n)\mathbf{x}(n)$ vektör çarpımı arasında bir ilişki bulunarak hibrit algoritmanın işlem yükünde bir azalma sağlanabilir. NLMS algoritmasındaki $\mathbf{x}^T(n)\mathbf{x}(n)$ çarpımı

$$\mathbf{x}^T(n)\mathbf{x}(n) = \sum_{i=0}^{M-1} x^2(n-i) \tag{3.1}$$

şeklinde yazılabilir. Toplam sembolündeki indisler değiştirilerek aynı çarpım

$$\mathbf{x}^T(n)\mathbf{x}(n) = \sum_{i=n-(M-1)}^{n} x^2(i) \tag{3.2}$$

olarak da yazılabilir. Diğer taraftan, korelasyon katsayılarının birikimli tahmini değerleri ise $1/n$ çarpanı göz önüne alınmadan aşağıdaki gibi yazılabilir.

$$r_k(n) = \sum_{i=1}^{n} x(i)x(i-k) \quad , \quad k = 0,1,\ldots,(M-1) \tag{3.3}$$

Burada $k = 0$ için sıfırıncı korelasyon katsayısı

$$r_0(n) = \sum_{i=1}^{n} x^2(i) \tag{3.4}$$

olarak yazıldığında iki parçanın toplamı halinde

$$r_0(n) = \sum_{i=1}^{n-M} x^2(i) + \sum_{i=n-(M-1)}^{n} x^2(i) \tag{3.5}$$

olarak yazılabilir. Bu eşitlikte (3.2) ve (3.4) eşitlikleri kullanıldığında

$$r_0(n) = r_0(n-M) + \mathbf{x}^T(n)\mathbf{x}(n) \tag{3.6}$$

sonucuna ulaşılır. Bu sonuca göre $\mathbf{x}^T(n)\mathbf{x}(n)$ çarpımı sıfırıncı korelasyon katsayısına bağlı olarak

$$\mathbf{x}^T(n)\mathbf{x}(n) = r_0(n) - r_0(n-M) \tag{3.7}$$

şeklinde hesaplanabilir. Görüldüğü gibi NLMS algoritmasında bulunan ve (2.10) ile verilen $\mathbf{x}^T(n)\mathbf{x}(n)$ çarpımı hibrit algoritmanın gerçeklenmesinde çarpma işlemi yapmadan tek bir çıkarma işlemi kullanılarak hesaplanabilir. GS algoritmasında $r_0(n)$ korelasyon katsayısı zaten hesaplanmaktadır. Bu durumda hibrit GS-NLMS algoritmasının gerçeklenmesinde ikinci aşamada, $(M-G)$ tane filtre katsayısını güncellemek için kullanılan NLMS algoritmasında (3.4) eşitliğinden yararlanıldığında, işlem yükü μ adım boyu parametresinin de 1 olarak seçilmesi durumunda $2M-G+1$ olmaktadır.

Hibrit GS-NLMS algoritmasının gerçeklenmesinde ilk aşamada, G adet filtre katsayısının güncellenmesinde GS algoritmasının kullanılması durumunda, her bir parametrenin güncellenmesi için M adet olmak üzere G adet katsayının güncellenmesi için GM adet çarpma işlemi yapılmaktadır. Ayrıca, M adet oto-korelasyon katsayısının güncellenmesinde M adet çarpma işlemi yapılmaktadır ve G adet çapraz-korelasyon katsayısının güncellenmesinde G adet çarpma işlemi yapılmaktadır. Sonuç olarak hibrit algoritmada ilk aşamada filtre katsayılarının ilk G tanesinin GS algoritmasıyla güncellenmesi durumunda $(GM+G+M)$ adet çarpma işlemi yapılmaktadır. Hibrit algoritmanın ikinci aşamasında geriye kalan $(M-G)$ tane filtre katsayısının NLMS algoritmasıyla güncellenmesi durumunda ise $2M-G+1$ adet çarpma işlemi yapılmaktadır. Sonuç olarak hibrit algoritmanın gerçeklenmesinde toplam $MG+3M+1$ adet çarpma ve bölme işlemi yapılmaktadır.

Hibrit algoritmanın gerçeklenmesinde, ilk aşamada G tane filtre katsayısını güncellemek için kullanılan GS algoritmasındaki parametrelerin yeri de işlem yükünü bir miktar etkilemektedir. Burada M uzunluklu bir adaptif filtrenin katsayılarını güncellemek için kullanılacak olan GS algoritması daha detaylı olarak aşağıdaki gibi verilebilir.

$$w_1(n+1)=\left\{p_1(n)-\begin{bmatrix}r_1(n) & r_2(n) & \cdots & r_{M-1}(n)\end{bmatrix}\cdot\begin{bmatrix}w_2(n)\\ w_3(n)\\ \vdots\\ w_M(n)\end{bmatrix}\right\}/r_0(n)$$

$$\vdots \qquad (3.8)$$

$$w_M(n+1)=\left\{p_M(n)-\begin{bmatrix}r_{M-1}(n) & r_{M-2}(n) & \cdots & r_1(n)\end{bmatrix}\cdot\begin{bmatrix}w_1(n+1)\\ w_2(n+1)\\ \vdots\\ w_{M-1}(n+1)\end{bmatrix}\right\}/r_0(n)$$

$$w_{\frac{M+1}{2}}(n+1)=\left\{p_{\frac{M+1}{2}}(n)-\begin{bmatrix}r_{\frac{M-1}{2}}(n) & \ldots & r_1(n) & r_1(n) & \ldots & r_{\frac{M-1}{2}}\end{bmatrix}\cdot\begin{bmatrix}w_1(n+1)\\ w_2(n+1)\\ \vdots\\ w_M(n)\end{bmatrix}\right\}/r_0(n) \qquad (3.9)$$

Burada süzgecin orta noktasında yer alan katsayı M tek olduğunda Eşitlik 3.9' da verildiği gibi hesaplanabilir. Buradan görülebileceği gibi süzgecin orta noktasındaki katsayının güncellenmesi için $r_0(n)$ katsayını da hesaba katarsak toplam $\frac{M+1}{2}$ adet r(n) katsayısı yeterlidir. *N(i)* i'nci w katsayısını güncellemek için gerekli r(n) katsayı adetini göstermek üzere aşağıdaki fonksiyonu tanımlayabiliriz.

$$N(i)=\frac{M+1}{2}+\left|i-\frac{M+1}{2}\right| \qquad i=1,2,...,M \qquad (3.10)$$

Eşitlik (3.10)' da verilen fonksiyona göre süzgecin orta noktasından uzaklaşıldıkça kullanılan r(n) katsayı adeti de artmaktadır. Hibrit GS-NLMS algoritmasında sınırlı sayıda katsayı Gauss-Seidel metodu ile güncelleneceğinden bu katsayıların süzgecin merkezine yerleştirilmesi işlem yükünde azalmalara sebep olacaktır.

Hibrit GS-NLMS algoritmasında (M – G) değerinin çift olması durumunda G adet katsayı süzgeç merkezine göre simetrik olarak yerleştirilebilir. Bu durumlarda süzgeç merkezinden en uzak w katsayıları $w_{\frac{M-G}{2}+1}$ ve $w_{\frac{M+G}{2}}$ katsayılarıdır. Bu katsayıların indislerini Eşitlik (3.10)' da yerine koyduğumuzda bu katsayıları güncellemek için gereken toplam otokorelasyon katsayısı adeti $\frac{M+G}{2}$ olarak bulunur.

(M - G) değerinin tek olması durumunda ise G adet katsayı süzgeç merkezine simetrik olarak yerleştirilemez. Böyle durumlarda G adet katsayı süzgeçin başlangıç katsayısı $w_1(n)$ yada son katsayısı $w_M(n)$ taraflarından birine daha fazla yaklaşır. Bu ise süzgeç merkezinden en uzak katsayının $w_{\frac{M-G+1}{2}}$ veya $w_{\frac{M-G+1}{2}}$ olmasına neden olur. Bu tarz bir durum işlem yükünde yine de azalmalara sebep olsada (M - G) değerinin çift olması durumuna göre biraz daha fazla işlem yüküne neden olur. Bu durumda w katsayılarını güncellemek için $\frac{M+G+1}{2}$ adet katsayıya ihtiyaç vardır. Bahsedilen özellikler kullanılarak hibrit GS-NLMS algoritmasının işlem yükü Eşitlik 3.11' de verilmiştir.

$$\left\{\begin{array}{ll} MG+\frac{5M+G}{2}+1 & ;\quad M-G \quad çift \\ MG+\frac{5M+G+3}{2} & ;\quad M-G \quad tek \end{array}\right\} \tag{3.11}$$

3.2. Benzetim ve Yakınsama Özellikleri

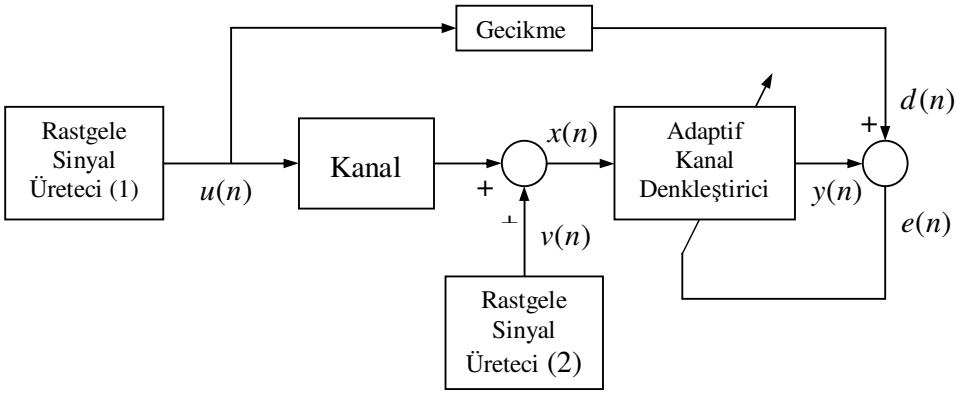
Bu altbölümde sunulan hibrit GS-NLMS algoritmasının yakınsama hızına etki eden faktörlerin incelenmesi ve bu algoritmanın yakınsama hızının NLMS ve GS algoritmaları ile kıyaslanması için yapılan benzetim çalışmaları ve elde edilen sonuçlar verilmiştir.

3.2.1. Benzetim koşulları

Önerilen hibrit GS-NLMS algoritmasının yakınsama özellikleri yapılan bir benzetim çalışmasıyla örnek bir uyarlamalı kanal denkleştirme problemi üzerinde test edilip benzer algoritmalarla birlikte karşılaştırmalı olarak incelenmiştir. Örnek olarak Haykin (2002) ve Bose (2004) tarafından da kullanılan ve darbe cevabı

$$h(n) = \begin{cases} \frac{1}{2}\left[1+\cos\left(\frac{2\pi}{W}(n-2)\right)\right] & , \quad n = 1,2,3 \\ 0 & , \quad diğer \end{cases} \tag{3.12}$$

olan doğrusal kanal modeli kullanılmıştır. Benzetim çalışmasında kullanılan sistemin blok diyagramı Şekil 3.2'te görülmektedir.



Şekil 3.2:
Adaptif kanal denkleştiricinin blok diyagramı

Blok diyagramda görülen rastgele sayı üreteci (1) kanalı uyarmak için kullanılan $u(n)$ test sinyalini üretmektedir. Burada $u(n)$ sinyali ∓ 1 seviyeli ve sıfır ortalamalı rastgele sinyaldir. Rastgele sinyal üreteci (2) ise kanal çıkışını bozan rastgele gürültü kaynağıdır. Benzetim çalışmasında sıfır ortalamalı ve varyansı $\sigma_v^2 = 0.001$ olan normal dağılıma sahip rastgele gürültü dizisi kullanılmıştır. Bu durumda kanal denkleştiricinin giriş işareti

$$x(n) = \sum_{k=1}^{3} h(k)u(n-k) + v(n) \tag{3.13}$$

eşitliği kullanılarak hesaplanmıştır. Kanal denkleştirici olarak uzunluğu $M = 11$ olan adaptif FIR filtre kullanılmıştır. Ayrıca, 7 adım geciktirilmiş giriş sinyali de kanal denkleştiricinin takip etmesi gereken $d(n)$ sinyali olarak kullanılmıştır. Kanalın darbe cevabındaki W parametresi giriş işaretinin korelasyon matrisinin özdeğer yayılımını kontrol etmek için kullanılmaktadır. Örneğin yapılan benzetim çalışmasında kullanılan $W = 3.5$ değeri için korelasyon matrisinin özdeğer yayılımı 46.8216 olmaktadır ve W değeri arttığında korelasyon matrisinin özdeğer yayılımı da artmaktadır.

Kanal denkleştiricinin girişine gelen 11 uzunluklu u(n), u(n-1) ... u(n-10) işaretlerinin tümü gerçel değerlerden oluştuğundan dolayı **R** korelasyon matrisi simetrik 11x11 lik bir matristir. Ayrıca h(n) darbe cevabı n=1,2,3 dışındaki değerler için 0 olduğundan ve v(n) gürültüsü sıfır ortalamalı beyaz gürültü olduğundan dolayı korelasyon matrisi **R** quintdiagonal dir. Yani **R** korelasyon matrisinin ana diagonali ve anadiagonalin altında ve üstündeki ikişer diagonallerin dışında kalan tüm matris elemanlarının değeri 0 dır. **R** korelasyon matrisi Eşitlik 3.14' de verilmiştir.

$$\mathbf{R} = \begin{bmatrix} r(0) & r(1) & r(2) & 0 & . & . & . & 0 \\ r(1) & r(0) & r(1) & r(2) & . & . & . & 0 \\ r(2) & r(1) & r(0) & r(1) & . & . & . & 0 \\ 0 & r(2) & r(1) & r(0) & . & . & . & 0 \\ . & . & . & . & \ddots & & & . \\ . & . & . & . & & \ddots & & . \\ . & . & . & . & & & \ddots & . \\ 0 & 0 & 0 & 0 & . & . & . & r(0) \end{bmatrix} \tag{3.14}$$

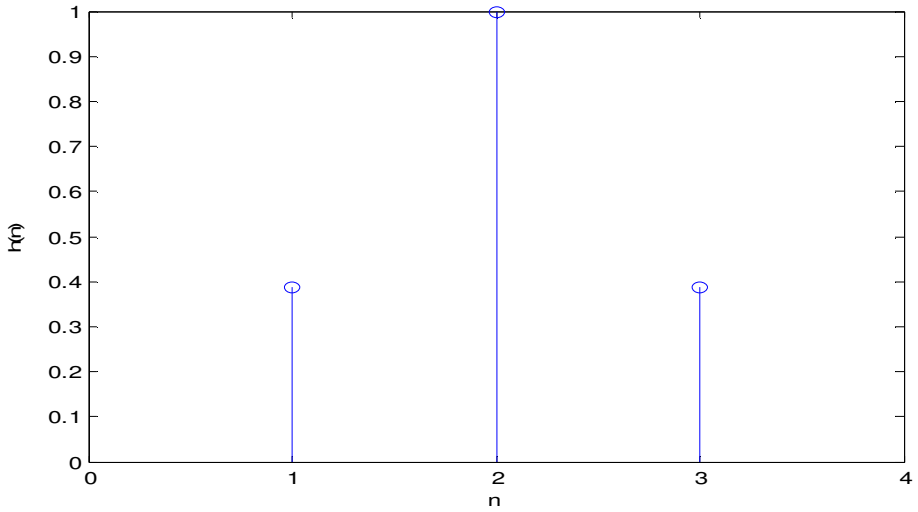
Ve

$$r(0) = h_1^2 + h_2^2 + h_3^2 + \sigma_v^2$$

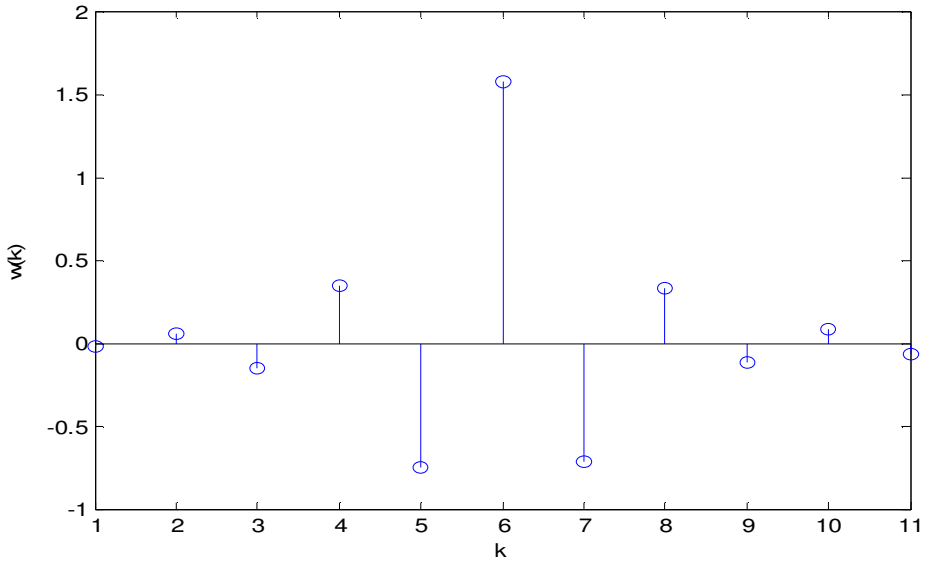
$$r(1) = h_1h_2+h_2h_3$$

$$r(2) = h_1h3$$

dür. W=3.5 değeri için h(n) darbe cevabı ve denkleştiricide oluşan filtrenin darbe cevabı Şekil 3.3 ve Şekil 3.4'te verilmiştir.



Şekil 3.3:
h(n) darbe cevabı



Şekil 3.4:
Kanal denkleştiricinin darbe cevabı

3.2.2 Hibrit GS-NLMS algoritmasının GS ve NLMS algoritmalarıyla karşılaştırılması

İlk olarak hibrit GS-NLMS algoritması GS ve NLMS algoritmalarıyla birlikte karşılaştırmalı olarak incelenmiştir. Yapılan 1000 adet benzetimin ortalaması alınarak elde edilen ortalama karesel hata grafikleri Şekil 3.5'te görülmektedir. Burada hibrit GS-NLMS algoritmasındaki GS algoritmasıyla uyarlamalı FIR süzgecinin katsayı vektörünün {w5,w6,w7} elemanları güncellenmiştir, yani $G = 3$ alınmıştır. Geriye kalan 8 eleman ise NLMS algoritması ile güncellenmiştir.

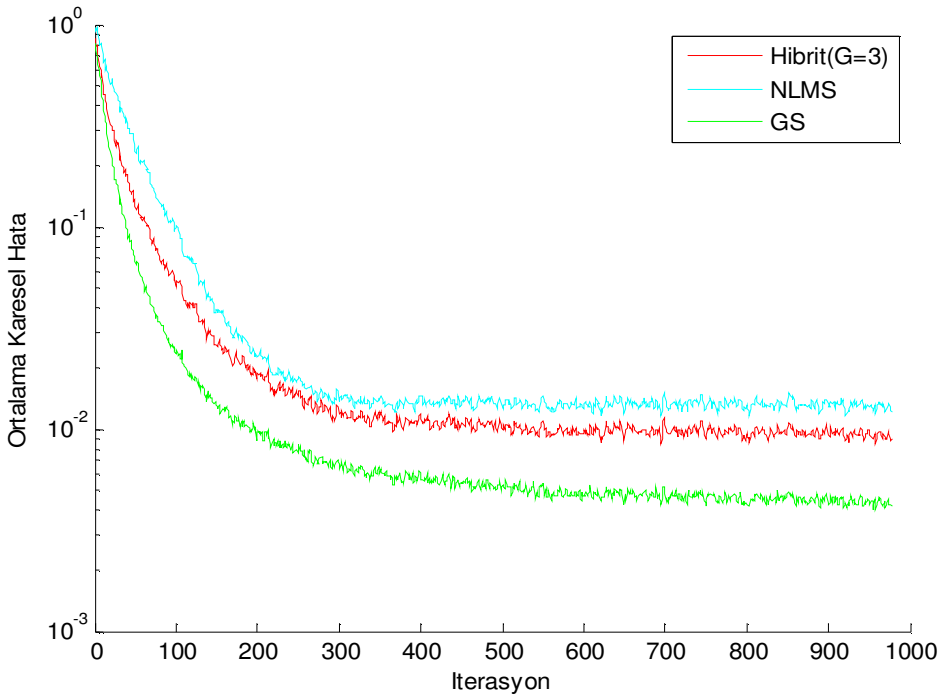
Gauss-Seidel algoritmasının yakınsama hızının katsayı güncelleme sıralaması ile değiştiği göz önünde bulundurularak bu algoritmanın katsayı güncelleme sıralamasıda hibrit GS-NLMS algoritmasına benzetilmiştir. Buna göre GS algoritmasında w_5, w_6, w_7, w_1, w_2, w_3, w_4, w_8, w_9, w_{10}, w_{11} sıralaması ile katsayılar güncellenmiştir. Dolayısı ile örneğin w1 katsayısı güncellenirken w_5, w_6, w_7 katsayılarının güncellenmiş değerleri kullanılmıştır.

Elde edilen sonuçlara göre, hibrit GS-NLMS algoritmasının yakınsama hızı GS ve NLMS algoritmalarının arasında kalmaktadır.

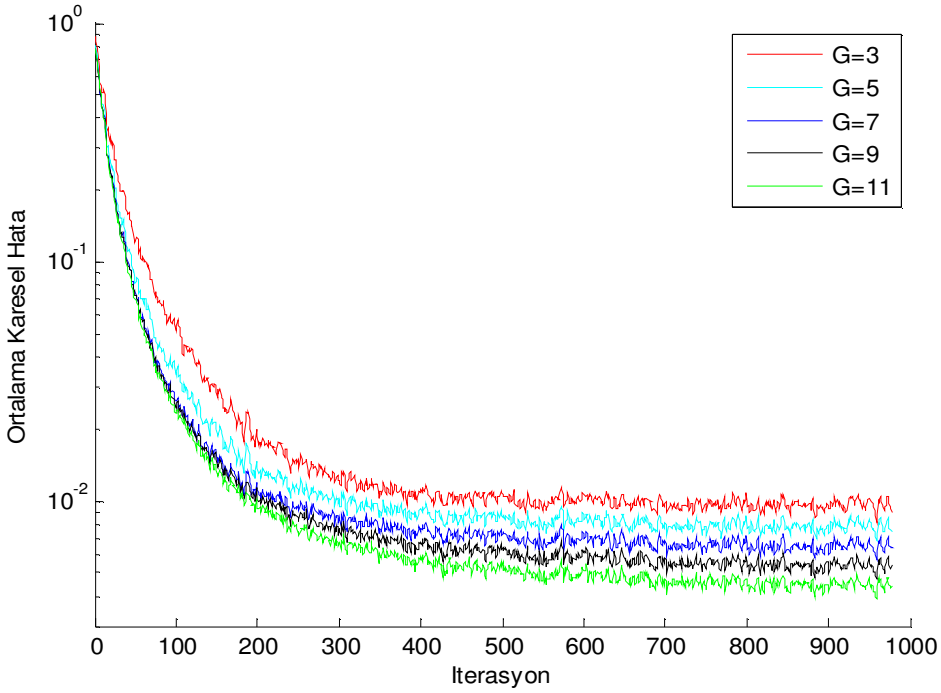
3.2.3 G Parametresinin yakınsama hızına etkisi

Çalışmanın bu kısmında GS algoritmasıyla güncellenen parametre sayısını belirleyen farklı G değerlerine göre hibrit GS-NLMS algoritmasının yakınsama hızının değişimi incelenmiştir. Seçilen farklı G değerleri için 1000 adet benzetimin ortalaması alınarak hesaplanan ortalama karesel hata grafikleri Şekil 3.6'da görülmektedir. Burada tüm G parametreleri denkleştirici katsayılarının ortasına yerleştirilmiştir.

Elde edilen sonuçlara bakıldığında, G değerinin artması durumunda, hibrit GS-NLMS algoritmasıyla elde edilen sonuçların, GS algoritmasıyla elde edilen sonuçlara yaklaştığı görülmektedir.



Şekil 3.5:
Hibrit GS-NLMS algoritmasıyla elde edilen karşılaştırmalı benzetim sonuçları

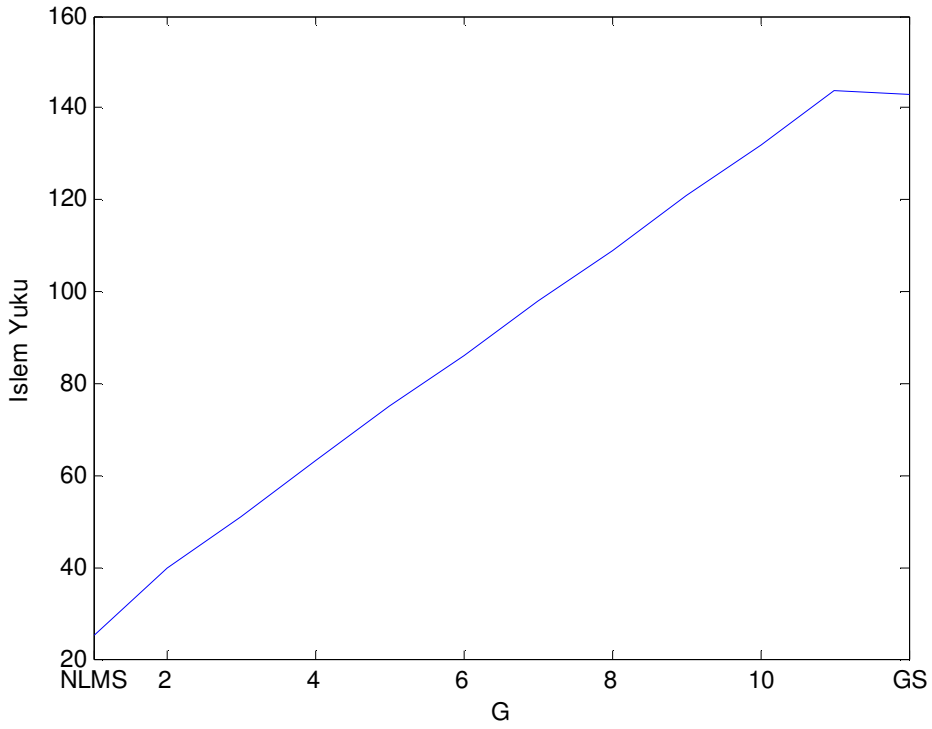


Şekil 3.6:
Hibrit GS-NLMS algoritmasının yakınsama hızının farklı G değerlerine bağlı değişimi

Hibrit GS-NLMS algoritmasının işlem yükü açısından diğer algoritmalarla karşılaştırılması durumunda, $M = 11$ için Çizelge 3.1'deki sonuçlar elde edilir. Bu sonuçlara göre, GS-NLMS algoritmasının işlem yükü, kullanılan G değerine de bağlı olarak $G < M - 1$ olduğu sürece GS ve NLMS algoritmaları arasında kalmaktadır. G değeri küçük olduğunda işlem yükü NLMS algoritmasına yakın olmakta, $G < M - 1$ olmak üzere büyük değerler aldığında ise işlem yükü GS algoritmasına yakın olmaktadır. Şekil 3.7' de G parametresinin değişiminin işlem yüküne etkisi verilmiştir. Görüldüğü gibi G parametresinin artışı işlem yükünde lineer bir artışa neden olmaktadır. Ancak $G = M - 1$ olduğunda hibrit GS-NLMS algoritmasında $MG + \frac{5M + G + 3}{2}$ olarak belirlenen işlem yükü miktarı $M^2 + 2M + 1$ olarak GS algoritmasının işlem yükünü ($M^2 + 2M$) geçecektir.

Çizelge 3.1
M = 11 için hibrit GS-NLMS algoritmasının işlem yükü

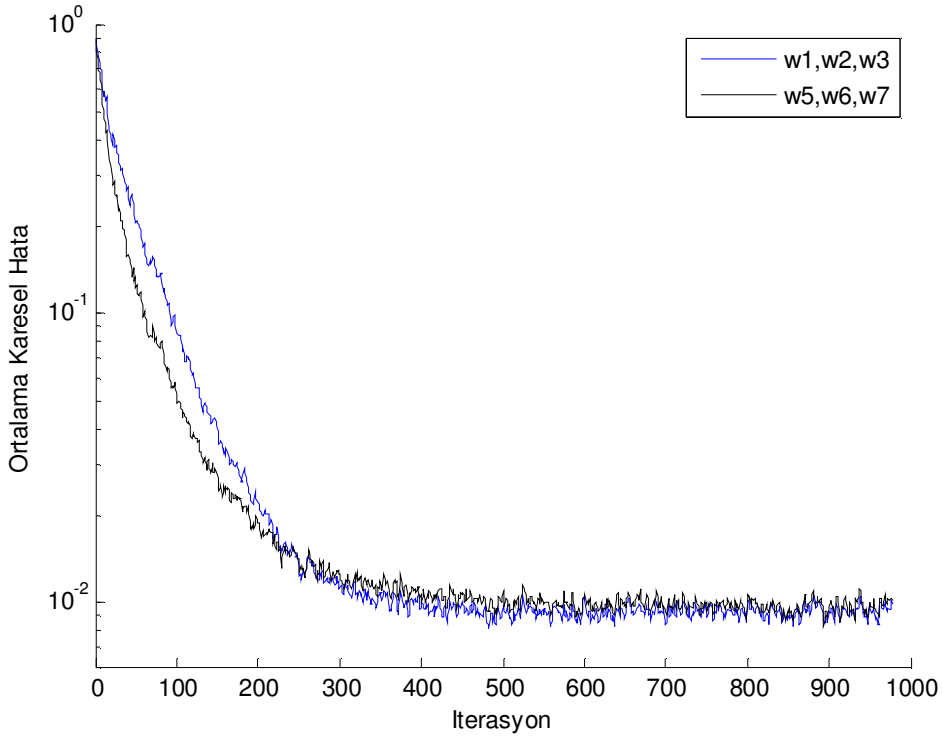
Algoritma:	$M = 11$ **için işlem yükü:**
NLMS Algoritması	25
GS Algoritması	143
Hibrit GS-NLMS ($G = 1$)	40
Hibrit GS-NLMS ($G = 2$)	51
Hibrit GS-NLMS ($G = 3$)	63
Hibrit GS-NLMS ($G = 4$)	75
Hibrit GS-NLMS ($G = 5$)	86
Hibrit GS-NLMS ($G = 6$)	98
Hibrit GS-NLMS ($G = 7$)	109
Hibrit GS-NLMS ($G = 8$)	121
Hibrit GS-NLMS ($G = 9$)	132
Hibrit GS-NLMS ($G = 10$)	144



Şekil 3.7:
G parametresinin değişiminin işlem yüküne etkisi

3.2.4 Hibrit GS-NLMS algoritmasında GS ile güncellenen katsayıların konumlarının yakınsamaya etkisi

Bu kısımda Hibrit GS-NLMS algoritmasında GS ile güncellenen katsayıların konumlarının yakınsamaya ve yakınsama hızına etkisi araştırılmıştır. Buna göre G=3 için iki farklı algoritma oluşturulmuştur. Bu algoritmalardan ilkinde { w_5, w_6, w_7 } katsayıları GS algoritması ile , { w_1, w_2, w_3, w_4, w_8, w_9, w_{10}, w_{11} } katsayıları ise NLMS algoritması ile güncellenmiştir. İkinci algoritmada ise { w_1, w_2, w_3 } katsayıları GS ile güncellenirken { w_4, w_5, w_6, w_7, w_8, w_9, w_{10}, w_{11}} katsayıları NLMS algoritması ile güncellenmiştir.Yapılan 1000 adet benzetimin ortalaması alınarak hesaplanan ortalama karesel hata grafikleri Şekil 3.8' te verilmiştir.



Şekil 3.8:
Hibrit GS-NLMS algoritmasında GS ile güncellenen katsayıların konumlarının yakınsamaya etkisi

Kolayca görülebileceği gibi { w_5, w_6, w_7 } katsayılarının GS ile güncellenmesi durumunda başlangıçta yakınsama biraz daha hızlı gerçekleşmektedir. Buna karşılık her iki algoritmada yeterli iterasyon sayısından sonra yaklaşık 0.01 lik ortalama karesel hata değerine oturmaktadır.

Hibrit algoritmanın gerçeklenmesinde, ilk aşamada GS algoritmasıyla güncellenen sınırlı sayıdaki G adet filtre katsayısının $\mathbf{w}(n)$ filtre katsayı vektörünün ortasından seçilmesi durumunda, daha az oto-korelasyon katsayısının kullanılmasından dolayı işlem yükünün bir miktar daha azaldığı Çizelge 3.2'de görülmektedir.

Çizelge 3.2

M = 11 için hibrit GS-NLMS algoritmasının işlem yükünün azaltılması

Algoritma:	**İlk G adet katsayı için:**	**Ortadaki G adet katsayı için:**
Hibrit GS-NLMS ($G=1$)	$[w_1] \rightarrow 45$	$[w_6] \rightarrow 40$
Hibrit GS-NLMS ($G=2$)	$[w_1 \quad w_2] \rightarrow 56$	$[w_5 \quad w_6] \rightarrow 51$
Hibrit GS-NLMS ($G=3$)	$[w_1 \quad w_2 \quad w_3] \rightarrow 67$	$[w_5 \quad w_6 \quad w_7] \rightarrow 63$
Hibrit GS-NLMS ($G=5$)	$[w_1 \quad \cdots \quad w_5] \rightarrow 89$	$[w_4 \quad \cdots \quad w_8] \rightarrow 86$
Hibrit GS-NLMS ($G=7$)	$[w_1 \quad \cdots \quad w_7] \rightarrow 111$	$[w_3 \quad \cdots \quad w_9] \rightarrow 109$
Hibrit GS-NLMS ($G=9$)	$[w_1 \quad \cdots \quad w_9] \rightarrow 133$	$[w_2 \quad \cdots \quad w_{10}] \rightarrow 132$

Buradan $M=11$ filtre uzunluğu ve farklı G değerleri için GS algoritmasıyla $\mathbf{w}(n)$ katsayı vektörünün ortasındaki G adet filtre katsayının güncellenmesi durumunda, hibrit algoritmanın toplam işlem yükünde bir miktar daha azalma olduğu görülmektedir. Bu azalma miktarı işlem yükü açısından küçük G değerleri için daha fazla olmaktadır. Özellikle hibrit GS-NLMS algoritmasında $\mathbf{w}(n)$ katsayı vektörünün tam ortasındaki 2 katsayının GS algoritması tarafından güncellenmesi durumunda $G=2$ için 51 adet, tam ortadaki 1 katsayının GS algoritması tarafından güncellenmesi durumunda $G=1$ için 40 adet çarpma işlemi yapılmaktadır.

3.2.5 Özdeğer yayılımının yakınsamaya etkisi

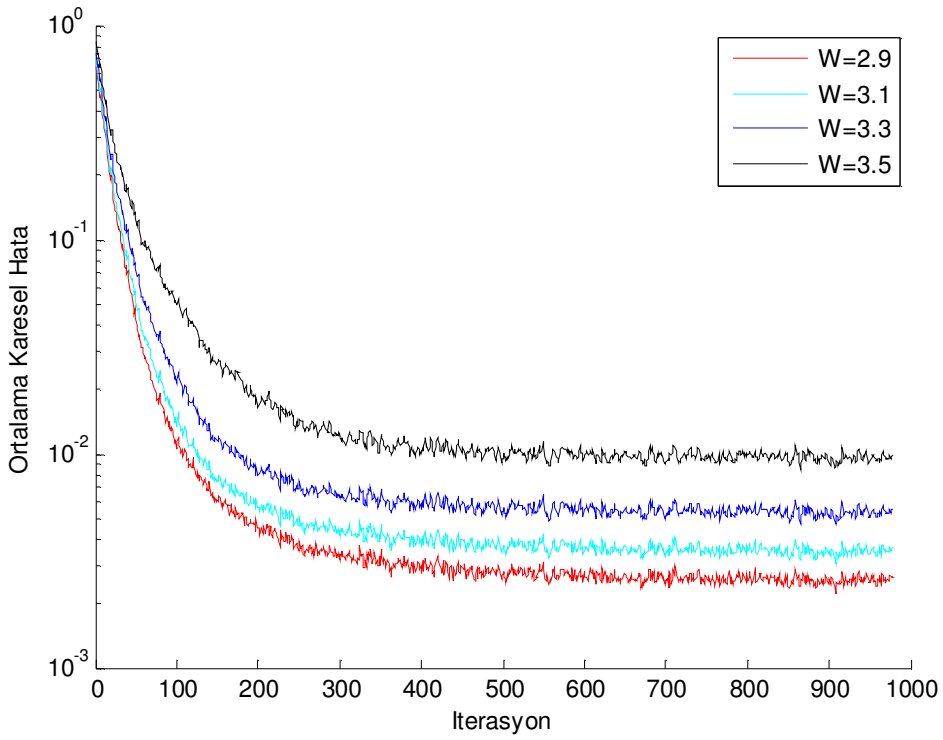
Üzerinde çalışılan kanal denkleştirici sisteminde W parametresi özdeğer yayılımını doğrudan etkilemektedir. Çizelge 3.3 de W parametresinin değişimi ile kanal denkleştirici girişindeki işaretin korelasyon katsayılarının, korelasyon matrisinin özdeğerlerinin ve korelasyon matrisinin özdeğer yayılımlarının değişimi verilmiştir.

Çizelge 3.3

W parametresinin değişimi ile kanal denkleştirici girişindeki işaretin korelasyon katsayılarının,korelasyon matrisinin özdeğerlerinin ve korelasyon matrisinin özdeğer yayılımlarının değişimi

W	2.9	3.1	3.3	3.5
r(0)	1.0963	1.1568	1.2264	1.3022
r(1)	0.4388	0.5596	0.6729	0.7774
r(2)	0.0481	0.0783	0.1132	0.1511
Λmin	0.3339	0.2136	0.1256	0.0656
Λmax	2.0295	2.3761	2.7263	3.0707
χ(**R**)	6.0782	11.1238	21.7132	46.8216

Benzetimlerde Rastgele sinyal üreteci (1) den alınan işaretler farklı W değerleri için elde edilen farklı h(n) işaretleri ile konvolüsyona sokulmuştur. Daha sonra Rastgele sinyal üreteci (2) den gelen gürültü sinyali tüm konvolüsyon sonuçlarına ayrı ayrı eklenmiştir.

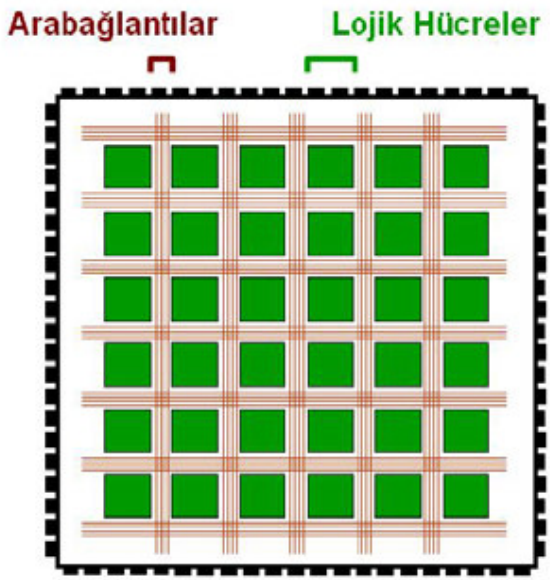


Şekil 3.9:
Hibrit GS-NLMS algoritmasının yakınsamasının özdeğer yayılımına bağlı değişimi

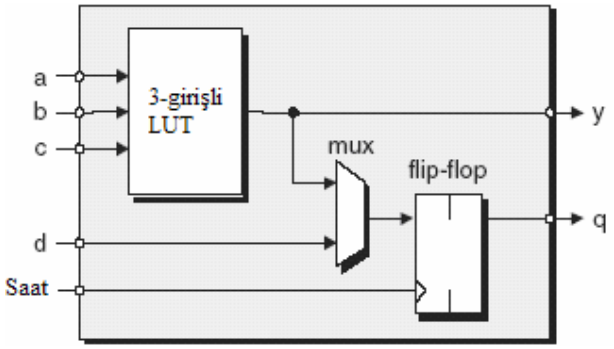
Farklı W değerleri için yapılan 1000 adet benzetimin ortalaması alınarak hesaplanan ortalama karesel hata grafikleri Şekil 3.9'da verilmiştir. Beklenildiği gibi Hibrit GS-NLMS algoritmasının yakınsama oranı özdeğer yayılımı arttıkça düşmüştür.

4. FPGA MİMARİSİ

FPGA, programlanabilir mantık blokları ve bu bloklar arasındaki ara bağlantılardan oluşan ve geniş uygulama alanlarına sahip olan sayısal tümleşik devrelerdir. Tasarımcının ihtiyaç duyduğu mantık fonksiyonlarını gerçekleştirme amacına yönelik olarak üretilmiştir. Dolayısıyla her bir mantık bloğunun fonksiyonu kullanıcı tarafından düzenlenebilmektedir. FPGA ile temel mantık kapılarının ve yapısı daha karmaşık olan devre elemanlarının işlevselliği artırılmaktadır. Alanda programlanabilir ismi verilmesinin nedeni, mantık bloklarının ve ara bağlantıların imalat sürecinden sonra programlanabilmesidir.Şekil 4.1‘ de temel FPGA yapısı, Şekil 4.2’ de ise bu yapının temel birimi olan lojik hücreler basitleştirilmiş şekilde verilmiştir.



Şekil 4.1:
Fpga yapısı



Şekil 4.2:
Lojik hücre yapısı

4.1 Üretim Teknolojileri

4.1.1 SRAM Tabanlı Mimari

FPGA'nın üstünlüklerinden en önemlisi SRAM yapılandırma hücreleri kullanmasıdır. Bu hücreler aygıtın tekrar kullanılmasını sağlar. Bu sayede yeni tasarımlar çok kolay bir şekilde hazırlanıp test edilebilir. Ayrıca, sistem içerisinde ana görevinden önce farklı görevleri yerine getirmesi için programlanabilir. Örneğin sistemin ilk açılış anında öz sınama yapması için programlanabilir. Açılış işlemleri bittikten sonra ise asıl görevini gerçekleştirecek şekilde programlanabilir.

SRAM tabanlı FPGA'ların diğer önemli bir avantajı ise önü açık bir teknoloji olmasıdır. Birçok bellek üreticisi müşterilerinin istekleri doğrultusunda bu konu üzerinde önemli araştırma ve geliştirme faaliyetlerinde bulunurlar. Yongada kullanılan diğer birimlerle aynı CMOS teknolojisine sahip olduklarından geliştirilme süreçlerinde ek bir işlem gerektirmezler. Eskiden üretim aşamasında yeni teknoloji geliştirmek için çoğunlukla bellek birimleri kullanılırdı. Şimdilerde ise boyut, karmaşıklık ve düzenlilik ölçütleri ele alınarak bu alanda da FPGA kullanımı artmıştır. FPGA'nın bellek birimlerine karşı diğer bir avantajı ise, bir hata oluşması durumunda FPGA yapısının hata bulma ve düzeltme işlemlerini kısaltmasıdır. SRAM tabanlı FPGA'nın anlatılan avantajlarına rağmen bazı dezavantajları da vardır. FPGA her sistem açılışında tekrar yapılandırılmak zorundadır. Bu nedenle sistemde harici bir bellek bulunması gerekir.

4.1.2 Karşıt Sigorta Tabanlı Mimari

SRAM tabanlı FPGA'ların aksine devre dışından özel araçlarla programlanırlar. Yapılandırılan tasarımlar SRAM tabanlı FPGA'larda olduğu gibi geçici değildir. Sistem açılışlarında tekrar yapılandırılma gereksinimleri yoktur. Böylece sistemde harici bellek bulundurma zorunluluğu ortadan kalkmış olur.

Karşıt sigorta tabanlı mimarinin özelliklerinden bir diğeri ise radyasyona karşı olan dayanıklılığıdır. Bu özellik askeri ve uzay uygulamalarında önemli bir avantaj sağlamaktadır. Çünkü SRAM tabanlı mimariler belirli bir düzeyde radyasyona maruz kaldıklarında yapılandırma hücrelerinde bozulmalar olmaktadır. Karşıt sigorta tabanlı mimarilerde ise yapı bir kez kurulduktan sonra bu şekilde değiştirilmesi mümkün değildir. Ancak flipflop'ların radyasyona duyarlılıklarını göz ardı etmemek gerekir. Bu nedenle önemli uygulamalarda hata durumları düşünülerek bazı önlemlerin alınması gereklidir. Her şeye rağmen bu mimarinin en önemli özelliği yapılandırma verilerinin FPGA'nın derinliklerine gömülmesidir. Böylece programcı bu verileri rahatlıkla okuyabilir ve aygıttın tamamen programlandığından emin olana kadar sınamasını sürdürebilir.

Karşıt sigorta tabanlı FPGA'nın boyut ve enerji tüketimi yönünden SRAM tabanlı FPGA'ya karşı avantajı olmasına rağmen, fazladan yapılandırma devresi gerektirdiği için bu avantajı çok fazla kendi lehine çevirememiştir. Yönlendirme gecikmesinin az olması ise SRAM tabanlı FPGA'ya göre daha hızlı olmasını sağlar. Bir kez programlanabilir olması ise en büyük kusurudur ve bu nedenle uygulama geliştirme için uygun değildir.

4.1.3. E2PROM/Flash Tabanlı Mimari

Flash tabanlı FPGA hücreleri SRAM tabanlı FPGA mimarisindekine benzer şekilde uzun ötelemeli yazmaç şeklindeki zincirlerle bağlıdırlar. Aygıt içerisinde ve dışarısında programlamaya izin veren çeşitleri bulunmaktadır. Ancak SRAM tabanlı FPGA'lara göre 3 kata kadar daha yavaş yapılandırılabilirler. Flash tabanlı mimaride veriler kalıcı

olduğundan her sistem başlangıcında yeniden yapılandırılması gerekmez. Ancak koruma amaçlı olarak çoklu anahtar denilen ve boyu elli ile birkaç yüz bit arasında değişen bir bit dizisi kullanılabilir.

İki transistorlu Flash tabanlı aygıtlar EPROM tabanlı aygıtlara göre yaklaşık olarak 2,5 kat büyük olmalarına karşın SRAM tabanlı aygıtlara göre daha küçüktürler. Bu özellik, kalan lojik elemanların birbirlerine daha yakın olmasını ve bu sayede bağlantı gecikmelerinin azalmasını sağlar. Öte yandan standart CMOS teknolojisine göre fazladan 5 adım gerektirirler. Bu nedenle SRAM tabanlı aygıtların birkaç nesil gerisinde kalmışlardır. İçerdikleri pull-up resistörlerinden dolayı güç tüketimleri de fazladır.

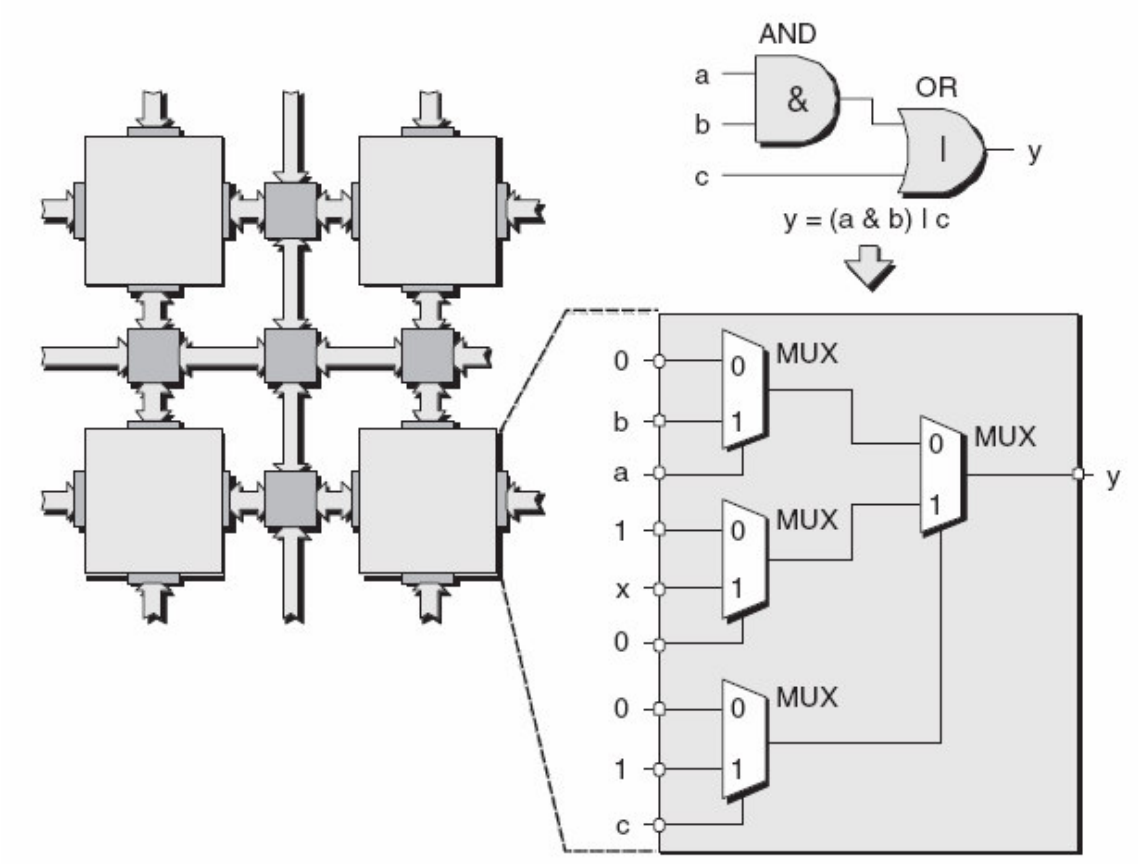
4.1.4. Melez SRAM-Flash Tabanlı Mimari

Yapılandırma hücreleri SRAM tabanlı ve Flash tabanlı aygıt hücrelerinin birleşimi şeklindedir. Bu mimaride Flash hücreleri önceden yapılandırılır. Sistem başlangıcından sonra ise Flash hücrelerindeki veriler paralel olarak SRAM hücrelerine kopyalanır. Böylece karşıt sigorta mimarideki kalıcılık sağlanmış olur. Sistem yeniden başlatıldığında aygıt vakit kaybetmeden hazır hale gelir. Ayrıca karşıt sigorta mimarinin tersine sistem başladıktan sonra SRAM hücrelerindeki veriler değiştirilebilir. Bu veriler bir sonraki açılışta geçerli olacaktır. Bunun yanında Flash hücreleri kullanılarak sistem içinden veya dışından yapılandırma mümkün olabilmektedir.

4.2. Programlanabilir Hücre Mimarileri

4.2.1. MUX Tabanlı Hücre

Üç girişli y = (a & b) | c fonksiyonunun sadece çoğullayıcılardan (multiplexer) oluşan bir blokla nasıl gerçeklenebileceği Şekil 4.3' de verilmiştir.



Şekil 4.3:
3 girişli lojik fonksiyonun mux tabanlı hücre ile gerçeklenmesi

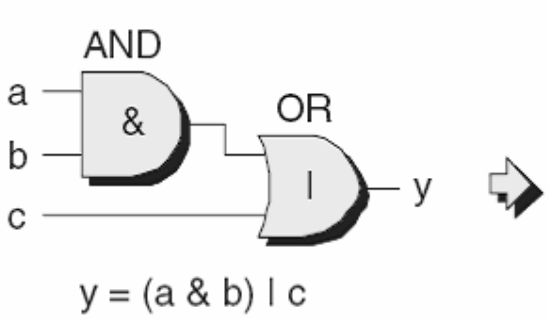
Bu blok girişlere verilen lojik 0, lojik 1 ve asıl girişler olan a, b, c ve onların tümleyenlerinin girişe direk verilmesi ile veya başka bir bloğun çıkışının bağlanması ile yapılandırılabilir. X ile gösterilen girişlerin çıkışa bir etkisi olmadığını gösterir. Bu yöntem her bloğun bir fonksiyonu oluşturması için sayısız yol sağlar.

4.2.2. LUT Tabanlı Hücre

Bu yapıda giriş işaretleri başvuru tablosundan (lookup table) doğru çıkışı bulmak için işaretçi olarak kullanılır. Girişlerin alabileceği her değer için tabloda bir çıkış değeri bulunur.

y = (a & b) | c fonksiyonunu bu kez LUT tabanlı mimaride gerçeklemek istersek oluşturmamız gereken başvuru tablosu Şekil 4.4 'deki gibi olmalıdır.

$$y = (a \,\&\, b) \mid c$$

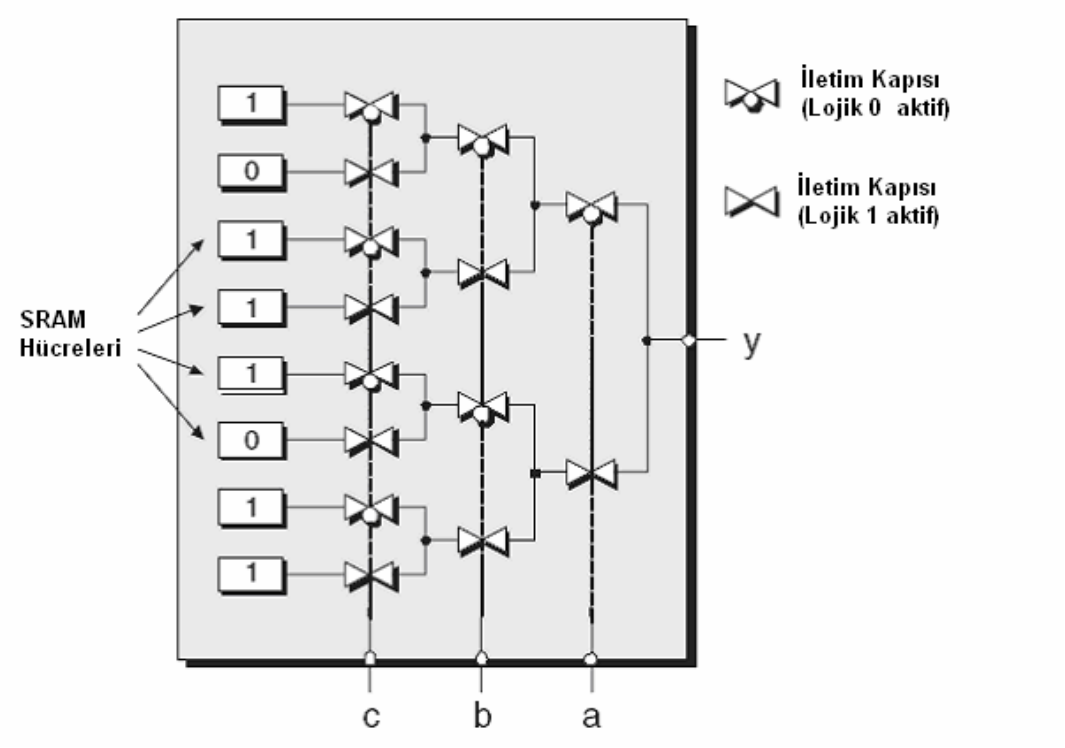


a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Şekil 4.4:
3 girişli lojik fonksiyonun doğruluk tablosu

Bu fonksiyon 3 girişli bir LUT ile gerçeklenebilmektedir. LUT yapısının SRAM bellek birimleri ile gerçekleştirildiği varsayılırsa Şekil 4.5 'deki gibi bir yapı ortaya çıkar. Burada a, b ve c girişleri ilgili SRAM hücresini basamaklı iletim kapılarını kullanarak seçer.

Basamaklı iletim kapıları aktif durumdayken sinyali çıkışına iletir. Aksi durumda ise giriş ile çıkış arasındaki bağlantı elektriksel olarak kopmuş olur. Kapılardaki yuvarlaklar o kapının alçak (lojik 0) aktif olduğunu gösterir. Yuvarlak olmayan kapılar ise yüksek (lojik 1) aktif kapılardır. Çıkışın basamaklı iletim kapıları ile seçilen SRAM hücresinin değerini alacağı açıkça görülmektedir.

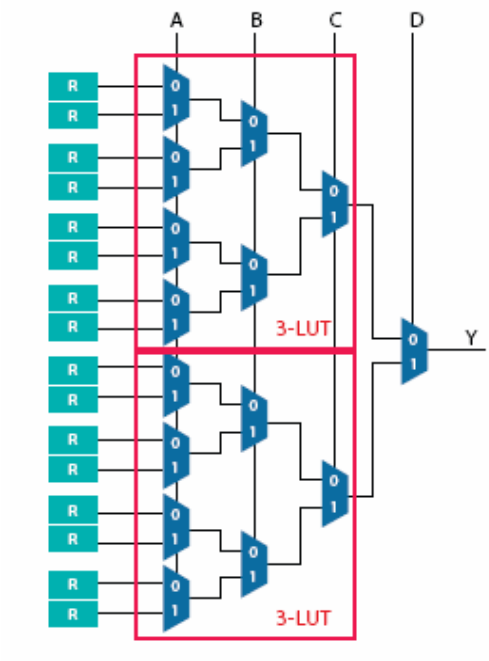


Şekil 4.5:
3 girişli lojik fonksiyonun LUT tabanlı hücre ile gerçeklenmesi

LUT tabanlı mimari MUX tabanlı mimarilere göre daha hızlı bir sonuç vermektedir. Haberleşme ve ağ sistemlerinde fazlasıyla FPGA'lar kullanılmaktadır. Bu alanlarda kullanılan yüksek miktarda verilerin yazılması için de LUT mimarisi daha uygundur.

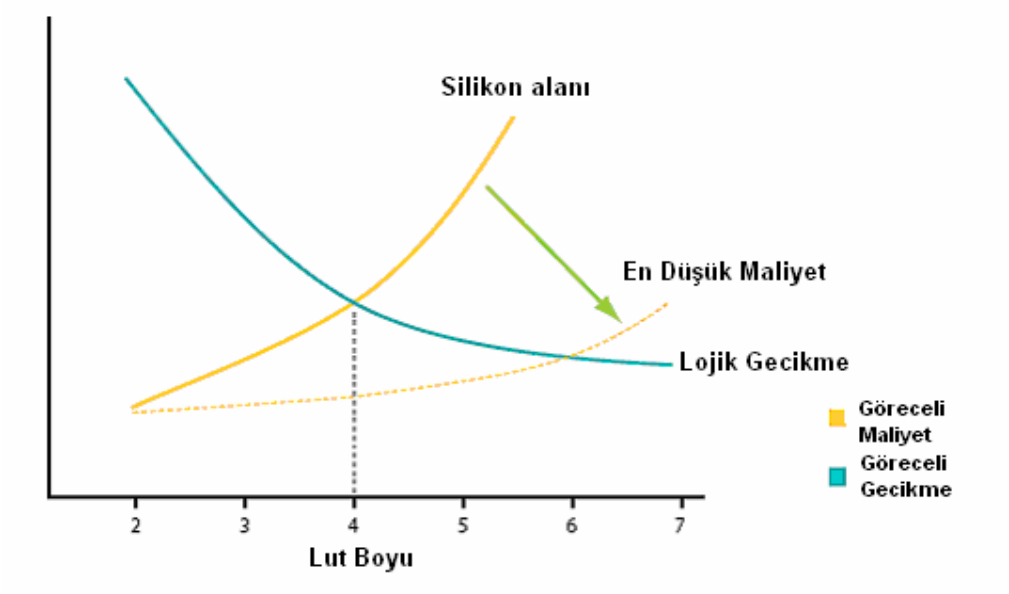
4.2.2.1. LUT Mimarisinde Giriş Sayısı

Bu mimarinin en büyük avantajlarından birisi n girişli bir LUT ile n girişli bir birleşimsel lojik fonksiyonun gerçekleştirilebilmesidir. Giriş sayısının artması daha karmaşık fonksiyonların gerçekleştirilmesini sağlar ancak eklenen her yeni giriş aynı zamanda SRAM hücrelerinin sayısının ikiye katlanmasına sebep olur.



Şekil 4.6:
4 girişli LUT yapısı

İlk FPGA'lar 3 girişli LUT'lar kullanılarak tasarlanmıştır. Daha sonra 3, 4, 5 ve 6 girişli LUT yapılarının birbirlerine olan avantajları incelenmiştir. 4 giriş ise en iyi çözüm olarak kabul edilmiştir.Şekil 4.6' da 4 girişli LUT yapısı gösterilmektedir.



Şekil 4.7:
LUT giriş sayısının değişiminin maliyete ve lojik gecikmelere etkisi

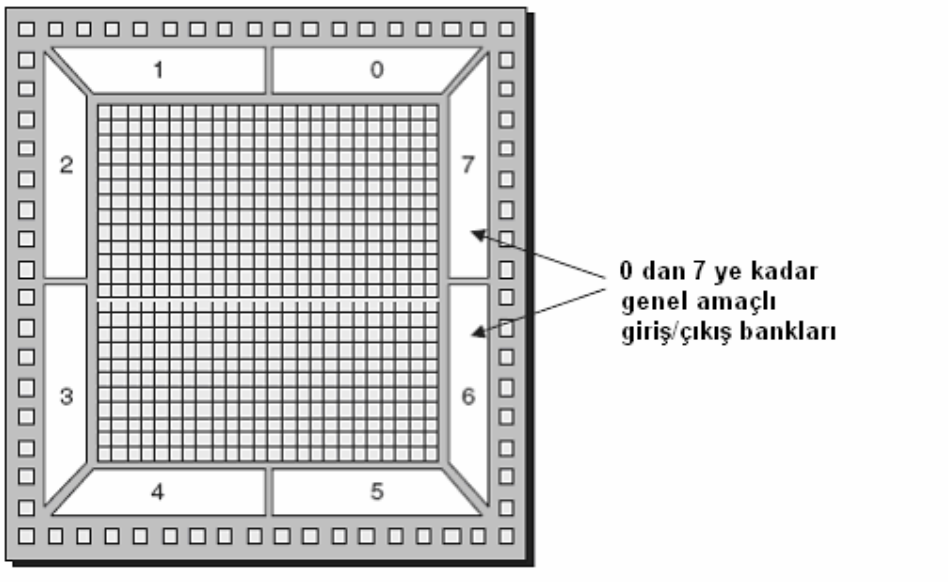
Geçmişte 3 girişli ve 4 girişli LUT'ların beraber kullanıldığı mimariler ortaya çıkmıştır. Bu mimariler aygıttan en iyi şekilde yararlanılmasını sağlamıştır. Ancak lojik sentezinin önemli kıstaslarından benzerlik ve düzene pek uygun olmadığı için vazgeçilmek zorunda kalınmıştır. Günümüzde en başarılı mimarilerden büyük bir çoğunluğu sadece 4 girişli LUT kullanılarak yapılmıştır. Şekil 4.7' de LUT giriş sayısının değişiminin maliyete ve lojik gecikmelere etkisinin grafiği verilmiştir.

4.3. Giriş/Çıkış Birimleri

Günümüzde bir FPGA yongasının altına sıralı bir şekilde yerleştirilmiş 1000 veya daha fazla bacak bulunabilmektedir.

Giriş/çıkış birimlerinde veri iletim standardı, tasarıma, kullanılan aygıtlara ve çevresel birimlere göre değişmektedir. Buradaki problem tüm standartları destekleyen bir mimari tasarlama gereksinimidir. Bunun için FPGA daki giriş-çıkış birimi herhangi bir standarttaki veriyi kabul edebilecek ve gönderebilecek şekilde yapılandırılabilir olmalıdır. Bu gereksimi karşılamak amacıyla FPGA'daki giriş-çıkış birimleri belirli sayıda kümeye bölünebilir. Böylece her küme belirli bir standarda uygun biçimde yapılandırılarak tüm standartlar desteklenmiş olur.

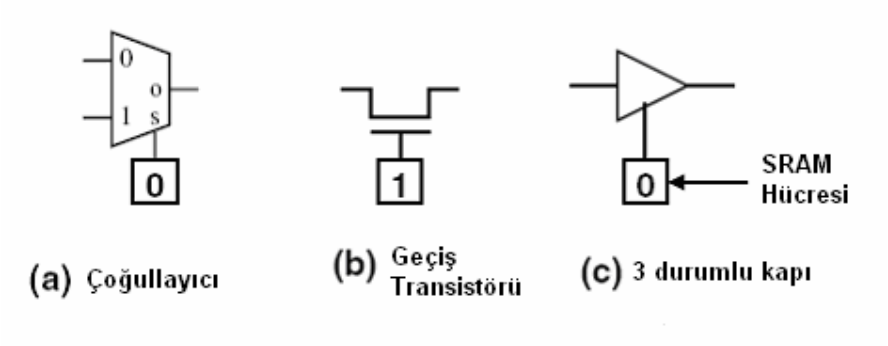
Şekil 4.8'de 0–7 arasında kümelenmiş giriş-çıkış birimleri içeren FPGA'nın yapısı gösterilmektedir.



Şekil 4.8:
FPGA giriş/çıkış birimleri

4.4. Ara Bağlantılar

FPGA tasarımcıları çok çeşitli ara bağlantı yapıları kullanmışlardır. Bu bağlantılar sayesinde birden fazla lojik hücre birleşerek daha büyük fonksiyonları sağlayacak yapıyı kurabilirler. Programlanabilir ara bağlantıları sağlamak için üç ana anahtarlama yöntemi kullanılır: çoğullayıcı, geçiş transistoru ve üç durumlu kapı. Şekil 4.9'da SRAM hücreleri ile kontrol edilen bahsi geçen anahtarlama devreleri gösterilmiştir.



Şekil 4.9:
Sram hücreleri ile kullanılan anahtarlama devreleri

Çoğullayıcılar FPGA'larda yoğun olarak kullanılırlar. Ara bağlantının karmaşıklığına göre iki girişten sekiz girişe kadar desteklediği için geniş bir kullanım alanı edinmişlerdir.

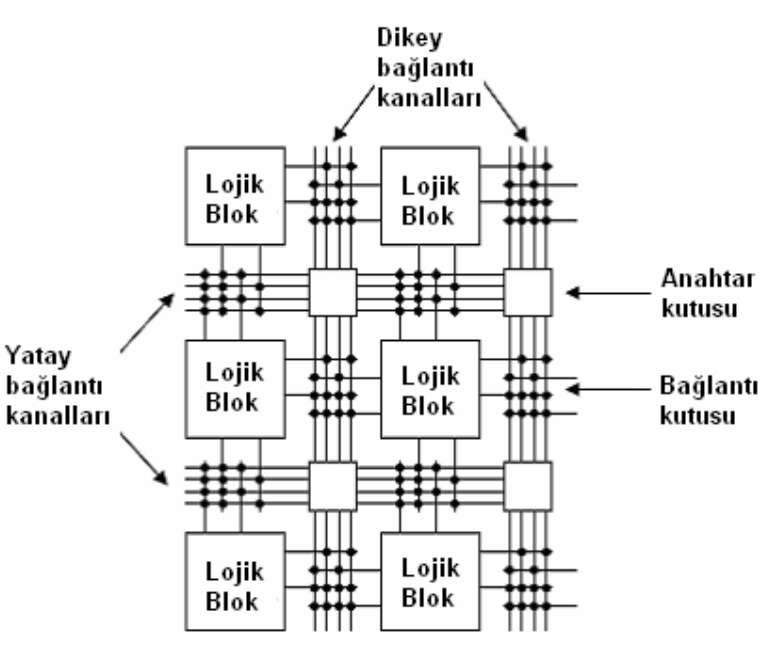
Lojik kümelerde ara bağlantılar birkaç farklı amaçla kullanılabilirler. Bunlardan birincisi lojik elemanlara gelen giriş sinyallerinin ve lojik elementten çıkan çıkış sinyallerini bağlantılarının belirlenmesidir. Diğer bir kullanımı ise bu sinyallerin lojik elemanlar arasındaki yayılımının nasıl olacağının belirlenmesidir. Değiştirilemez ara bağlantıların kullanım amacı ise değiştirilebilir ara bağlantılarda oluşan gecikmeyi en aza indirmektir.

FPGA'larda çeşitli ara bağlantı yapıları kullanılmaktadır. Bunlardan başlıca dört tanesi; ada bağlantısı, hücresel bağlantı, uzun hat bağlantısı ve sıralı bağlantıdır. Günümüzdeki ara bağlantı yapıları çok daha karmaşık olmasına rağmen, söz edilen bağlantılar genel yapıyı açıklamada yardımcı olurlar.

4.4.1. Ada Bağlantı Modeli

Bu bağlantı mimarisinde lojik bloklar, yatay ve dikey parçalı bağlantı kanalları ile çevrilidirler. Lojik bloklar bu kanallara bağlantı kutusu (connection box) yardımı ile, kanallar ise birbirine anahtar kutusu yardımı ile bağlanırlar. Bu mimarinin baskın özelliği lojik blokların birbirlerine parçalı bağlantılar yardımıyla bağlanmasıdır. Çoğu Xilinx FPGA mimarisinde bu bağlantı yöntemi kullanılmaktadır. Xilinx mimariyi daha etkin hale getirmek için farklı boyutlarda parçalı bağlantılar kullanır ve lojik bloklar arasında yerel bağlantılar oluşturur.

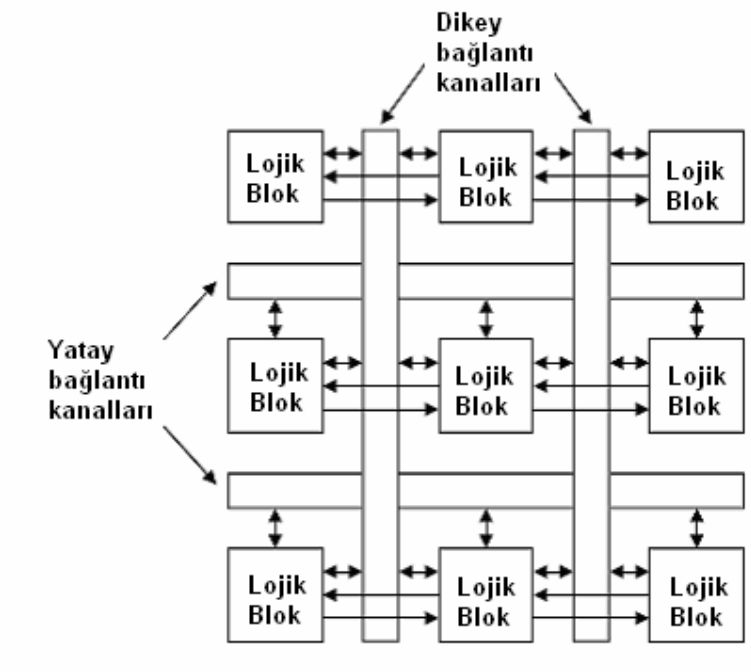
Ada bağlantı mimarisinin genel görünüşü Şekil 4.10'da gösterilmiştir.



Şekil 4.10:
Ada bağlantı modeli

4.4.2. Uzun Hat Bağlantı Modeli

Bu mimaride lojik bloklar birden fazla hattan oluşan yatay ve dikey bağlantı kanalları ile çevrilidirler. Bu kanallardaki hat sayısı aygıtın genişliğini arttırır. Bu mimaride iki lojik bloğu birbirine bağlamak için bir yatay ve bir dikey uzun hat yeterlidir. İki hattın kesiştirilmesi ile bağlantı kurulmuş olur. Altera FPGA'larında kullanılan başlıca bağlantı yapısı bu şekildedir. Ayrıca Actel ProASIC FPGA'lar da bu yapıdadır. Uzun hat bağlantı yapısı Şekil 4.11'de gösterilmektedir.



Şekil 4.11:
Uzun hat bağlantı modeli

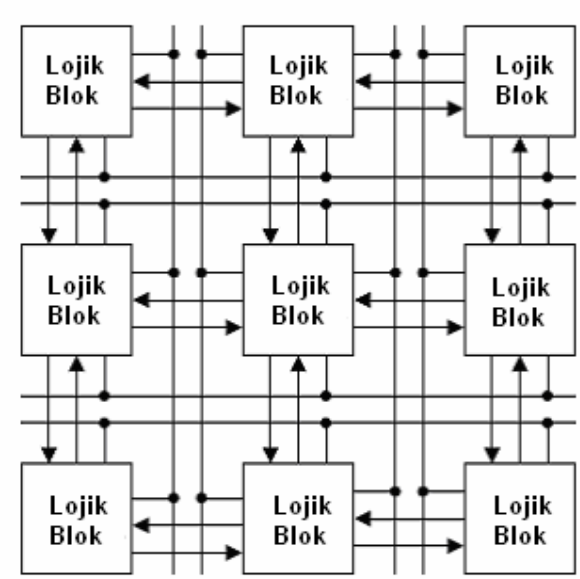
4.4.3. Hücresel Bağlantı Modeli

Bu mimaride bağlantılar lojik bloklar arasında ve olabildiğince az miktarda uzun hatlarla yapılır. Xilinx XC6200, CLi/Atmel 6000 ve Plessey/Pilkington ERA FPGA'ları bu mimariyi kullanan aygıtlara birer örnektir. Şekil 4.12 de hücresel bağlantı modeli gösterilmiştir.

Bu mimaride lojik bloklar sınırlı olan bağlantılara yardımcı olacak şekilde düzenlenirler. Birbirine uzak lojik blokları bağlamak için başka lojik bloklar kullanılabilir. Hücresel bağlantı mimarisi aşağıdaki nedenlerden dolayı pek tercih edilen bir mimari olamamıştır.

• Birbirine yakın olmayan komşuların bağlanması için oluşturulmuş birleşik yolların meydana getirdiği gecikme.

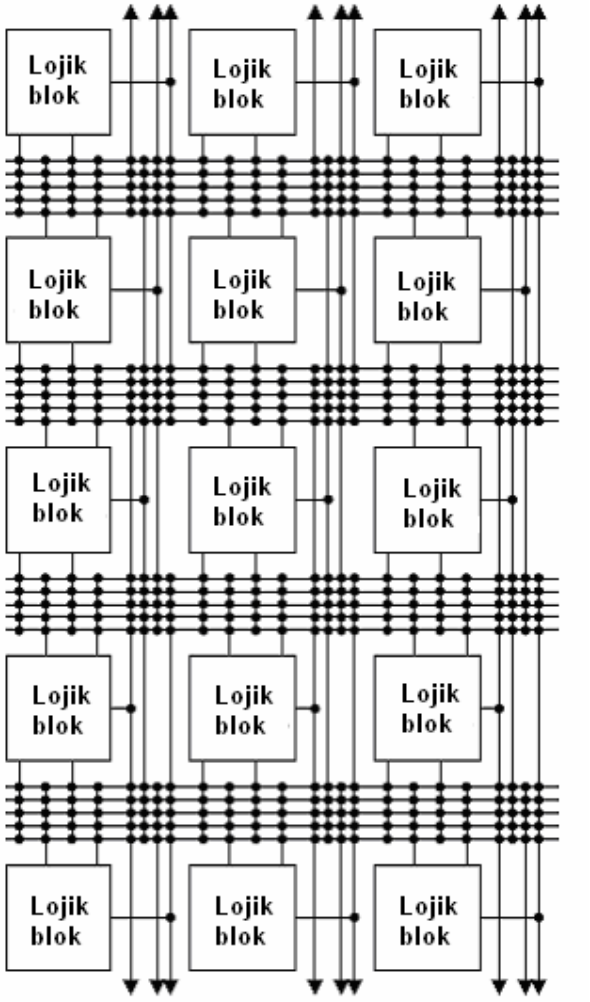
• Kullanılan programlama araçlarının bu bağlantıları yapılandırmakta çok zorlanmaları.



Şekil 4.12:
Hücresel bağlantı modeli

4.4.4. Sıralı Bağlantı Modeli

Sıralı bağlantı mimarisi Actel'in karşıt sigorta tabanlı FPGA'larının çoğunda olduğu gibi daha çok tekrar programlanamayan FPGA'larda bulunur. Tekrar programlanabilir mimarilerde genel olarak kullanılmaz. Sıralı bağlantı mimarisinde yatay bağlantı kanalları ustaca kullanılır. Bu mimarideki çoğu FPGA'da yatay kanallar arasındaki bağlantıları sağlamak için bazı dikey bağlantılar kullanılır.Şekil 4.13'de sıralı bağlantı modeli gösterilmiştir.



Şekil 4.13:
Sıralı bağlantı modeli

5. HİBRİT GS-NLMS ALGORİTMASININ DONANIMSAL TASARIMI

Bu bölümde önerilen hibrit GS-NLMS algoritması donanım olarak tasarlanmıştır. Tasarımın ilk aşamasında algoritma VHDL kullanılarak donanımsal olarak tanımlanmış, daha sonra Altera firması tarafından üretilmiş Quartus II yazılımı kullanılarak sentezlenmiş ve yine Altera tarafından üretilen Stratix II FPGA ailesi için yerleştirme & yönlendirme (place & route) işlemleri yapılmıştır. Son aşamada ise uyarlamalı kanal denkleştirme problemi benzetim ortamında tekrar edilerek tasarlanan donanım doğrulanmıştır. Ayrıca donanımsal parametreleri (silikon alanı, hız) kıyaslama amacı ile GS ve NLMS algoritmaları da donanımsal olarak tasarlanmıştır.

5.1 Hibrit GS-NLMS Algoritmasının VHDL İle Donanımsal Olarak Tanımlanması

5.1.1 VHDL nedir?

VHDL (Very High Speed Integrated Circuit Hardware Description Language) adından da anlaşılabileceği gibi bir donanım tanımlama dilidir. VHDL geliştirme ilk olarak Amerikan Savunma Departmanı tarafından başlatılmıştır. Donanımı tanımlamak için, bilgisayar ve insanlar tarafından aynı anda okunabilir olacak ve geliştiricileri yapısal ve anlaşılır kod yazmaya zorlayacak, yani kaynak kodun kendisi bir tür belirtim dokümanı (specification document) olarak sunulabilinecek bir dil istenmekteydi. Ayrıca kompakt yapıdaki kompleks fonksiyonları modelleyecek ardışık deyimleri desteklemeliydi.

1987 ‘de, VHDL Amerikan Elektrik ve Elektronik Mühendisleri Enstitüsü(IEEE) tarafından ilk kez standartlaştırıldı.1993 yılında güncelleştirilmesi yapıldı. Dosya işleme prosedürü dışında bu iki standart birbiriyle uyumludur. Dilin standardı (Language Reference Manual (LRM)) Dil Referans Elkitabı ile tanımlanmıştır. VHDL‘i analog ve karma sinyal dili elemanları için yükseltme çabaları ile yeni ve zor bir döneme girilmiştir. Bu yükseltmeye VHDL-AMS (analogue mixed signal) denir ve

VHDL onun bir üst kümesidir. Sayısal mekanizma ve metotlar bu eklenti ile değişikliğe uğratılmamıştır. Şu ana kadar, sadece analog kısmı için benzetim yapılabilmiştir çünkü analog sentez birçok sınırlı şartlardan etkilenen çok kompleks bir problemdir. Karma sinyal benzetimi, şu ana kadar henüz tam olarak çözülememiş dijital ve analog benzetim yazılımlarının senkronizasyon sorununun üstesinden gelebilmelidir.

5.1.2 VHDL 'in temel dil bileşenleri

VHDL dijital bir sistemi yapısal ve davranışsal açıdan farklı soyutluk seviyelerinde tanımlayabilecek biçimde tasarlamıştır. VHDL'in dil bileşenlerinden çoğu modelleme amaçlı tasarlanmıştır ve çok az bir bölümü fiziksel donanım olarak sentezlenebilir.

5.1.2.1 VHDL program örneği

Bir VHDL programı çeşitli tasarım birimlerinin bir araya gelmesi ile oluşur. Sentezlenebilir bir VHDL programı en az iki tasarım birimine ihtiyaç duyar. Bunlar entity (varlık) tanımı ve bu entity'ye bağlı mimari kısımdır (Architecture body). Şekil 5.1' de basit bir VHDL programı verilmiştir. Bu program a(0), a(1) ve a(2) girişlerine parite biti üreten bir devreyi tanımlar.

5.1.2.2 Entity tanımı

Entity tanımı devrenin dış dünya ile arayüzünü belirler ve devrenin ismi, giriş ve çıkış kapılarının temel karakteristikleri ve isimleri gibi devreye ait bazı özellikleri tanımlar. Entity tanımının basitleştirilmiş yazım kuralı aşağıdaki gibidir.

```
entity  entity_adı  is
    port (
        port_adları: mod veri_tipi;
        port_adları: mod veri_tipi;
        ...
        port_adları: mod veri_tipi
```

) ;

end entity_adı;

Port (kapı) tanımı port adı ,mod ve veri tipi terimlerinden oluşur. Buradaki mod terimi sinyalin yönünü belirtir ve in (giriş) , out (çıkış) ve inout (hem giriş hem çıkış) seçeneklerini kapsar.

Örneğimizde port tanımı kısmı iki portu kapsamaktadır. a kapısı giriş modunda ve std_logic_vector veri tipinde 3 bitlik veriyolundan oluşmuştur.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   entity even_detector is
4       port (
5       a:  in  std_logic_vector(2 downto 0);
6       even:   out std_logic
7       );
8   end even_detector;
9
10
11  architecture sop_arch of even_detector is
12      signal p1,p2,p3,p4 : std_logic;
13  begin
14      even <= (p1 or p2) or (p3 or p4);
15      p1 <= (not a(0)) and (not a(1)) and (not a(2));
16      p2 <= (not a(0)) and  a(1) and a(2);
17      p3 <=  a(0) and (not a(1)) and  a(2);
18      p4 <=  a(0) and  a(1) and (not a(2));
19  end sop_arch;
```

Şekil 5.1:
VHDL program örneği

5.1.2.3 Mimari kısım (Architecture body)

Mimari kısım devrenin dahili işletimini ve organizasyonunu belirler. VHDL'de bir entity tanımı için birden fazla mimari kısım tanımlanabilir. Daha sonra benzetim yada sentezleme için bu mimari kısımlardan biri seçilebilir. Mimari kısmın yazım biçimi aşağıdaki gibidir.

architecture mimari_adı **of** entity_adı **is**

deklarasyonlar;

begin

eşzamanlı_komut;

eşzamanlı_komut;

eşzamanlı_komut;

...

end mimari_adı;

Buradaki ilk satır mimarinin adını ve ait olduğu entity adını belirler. Seçime bağlı olarak mimari kısım deklarasyonlara sahip olabilir. Örneğimizde dahili sinyaller p1, p2, p3 ve p4 deklarasyonlar kısmında tanımlanmıştır. Mimari kısmın ana parçası begin ve end satırları arasında yer alır ve devrenin işletim ya da organizasyonunu eşzamanlı komutlarla belirler.

5.1.2.4 Tasarım birimi ve kütüphane (library)

Tasarım birimleri bir VHDL programının temel yapı bloklarıdır. Beş çeşit tasarım birimi vardır.

- Entity tanımı
- Mimari kısım
- Paket deklarasyonu
- Paket ana kısmı
- Konfigürasyon

Bunlardan entity tanımı be mimari kısım önceki alt bölümlerde tanıtılmıştır. Bir VHDL paketi sıkça kullanılan veri tipleri, altprogramlar ve bileşenlerden oluşur. İsminden de anlaşılabileceği gibi paket deklarasyonu VHDL paketlerinin deklarasyonlarını içerir. Paket ana kısmı altprogramların kodlarını ve yürütmesini (implementation) kapsar. Konfigürasyon birimi bir entity tanımı için birden fazla

mimari kısmın varlığı durumlarında hangi mimari kısmın benzetim ya da sentezleme için kullanılacağını belirler.

VHDL kütüphanesi tasarım birimlerinin barındırıldığı yerdir. Genel olarak sabit diskte bir klasöre haritalanmıştır. Varsayılan olarak tasarım birimleri 'work ' isimli bir kütüphanede saklanır.

5.1.3 Hibrit GS-NLMS algoritmasının VHDL tanımlamasında kullanılan bazı tasarım yöntemleri

Bu alt bölümde önerilen hibrit GS-NLMS algoritmasının VHDL tanımlaması yapılırken devre performansını olumlu yönde etkileyecek tasarım yöntemleri üzerinde durulmuştur.

5.1.3.1 Kaynak paylaşımı

Bir VHDL programı sentezlendiğinde tüm dil bileşenleri donanım üzerine haritalanır. Toplam devre boyutunu düşürmek için farklı operasyonlarda kullanılan ortak kaynakların belirlenmesi sıkça kullanılan bir yöntemdir. Bu yöntem kaynak paylaşımı olarak adlandırılır. Performansın düşmesine neden olabileceğinden bu yöntem ancak büyük yapılarda kullanıldığında etkili olur.

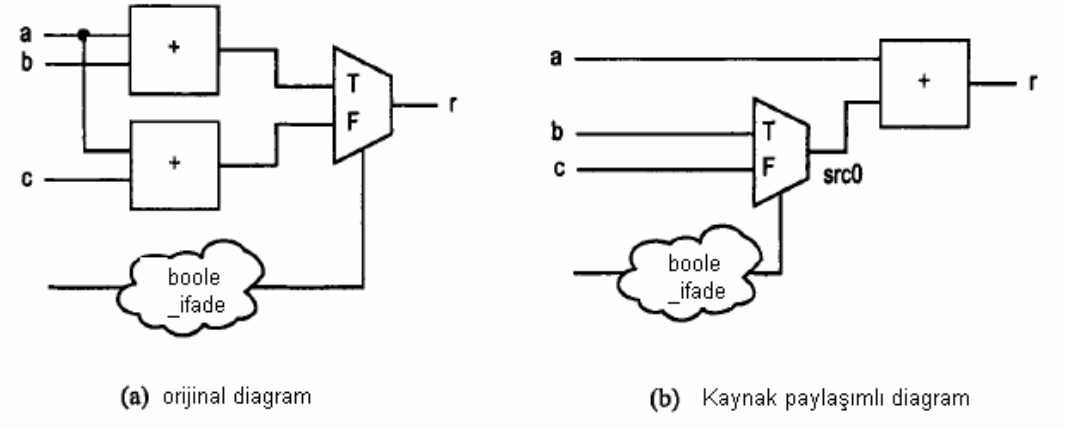
Belirli VHDL yapılarında bir zaman diliminde sadece bir operasyonun aktif olabileceği durumlar vardır. Bu tarz durumlara örnek yapılar olarak bir process de bulunabilen case ve if yapıları ya da koşullu sinyal atama yapıları gösterilebilir.

5.1.3.2 Kaynak paylaşım örneği

Kaynak paylaşımının daha kolay anlaşılabilmesi için bu alt bölümde basit bir kaynak paylaşımı örneği verilmiştir. Örnek aşağıdaki kod satırı üzerinedir.

r<= a + b **when** boole_ifade **else** a+c;

Bu kodun blok diyagramı Şekil 5.2a da gösterilmiştir.



Şekil 5.2:
Kaynak paylaşımı örneği

Şekilden de görülebileceği gibi devre 2 toplayıcı ve 1 multiplexer' dan oluşmuştur. Herhangi bir anda sadece bir toplama işlemi yapılacağından buradaki toplayıcı devresi kaynak paylaşımına müsaittir.

Örnek kodumuzu aşağıdaki gibi yazdığımızda 1 toplama devresi ile aynı işlemi gerçekleyebiliriz.

```
src0 <= b when boole_ifade else c;
r <= a + src0;
```

Bu şekilde düzenlenmiş yeni kodun blok diyagramı Şekil 5.2(b) de gösterilmiştir. Görülebileceği gibi toplama sonuçlarının multiplexer'a sokulması yerine istenen operandın toplama devresine sokulması sağlanmıştır.

Toplayıcının, multiplexer'ın ve boole ifadenin gecikmelerini sırasıyla $T_{toplayıcı}$, T_{mux} ve T_{boole} olarak tanımlayalım. İlk devrede toplayıcılar ve boole ifade paralel olarak çalışmaktadır. Bu yüzden toplam propogasyon gecikmesi $\max(T_{toplayıcı}, T_{boole}) + T_{mux}$ dur. İkinci devrede ise propogasyon gecikmesi $T_{boole} + T_{mux} + T_{toplayıcı}$ dır. Bu, boole

ifadenin toplayıcı ile kaskat olarak bağlanmasından kaynaklanmaktadır. İfadelerden görülebileceği gibi T_{boole} değerinin yeteri kadar küçük olması durumunda performansta büyük değişimler olmayacaktır.

5.1.3.3 Yerleşim ile ilişkili devreler

Sentezleme sonrası yerleştirme ve yönlendirme (Placement & Routing) aşamasında dijital devrenin silikon çip üzerinde fiziksel yerleşimi belirlenir. Her ne kadar VHDL' i kullanarak bu fiziksel yerleşim belirlenemese de en azından devrenin şekli hakkında belirtimler yapılabilir.

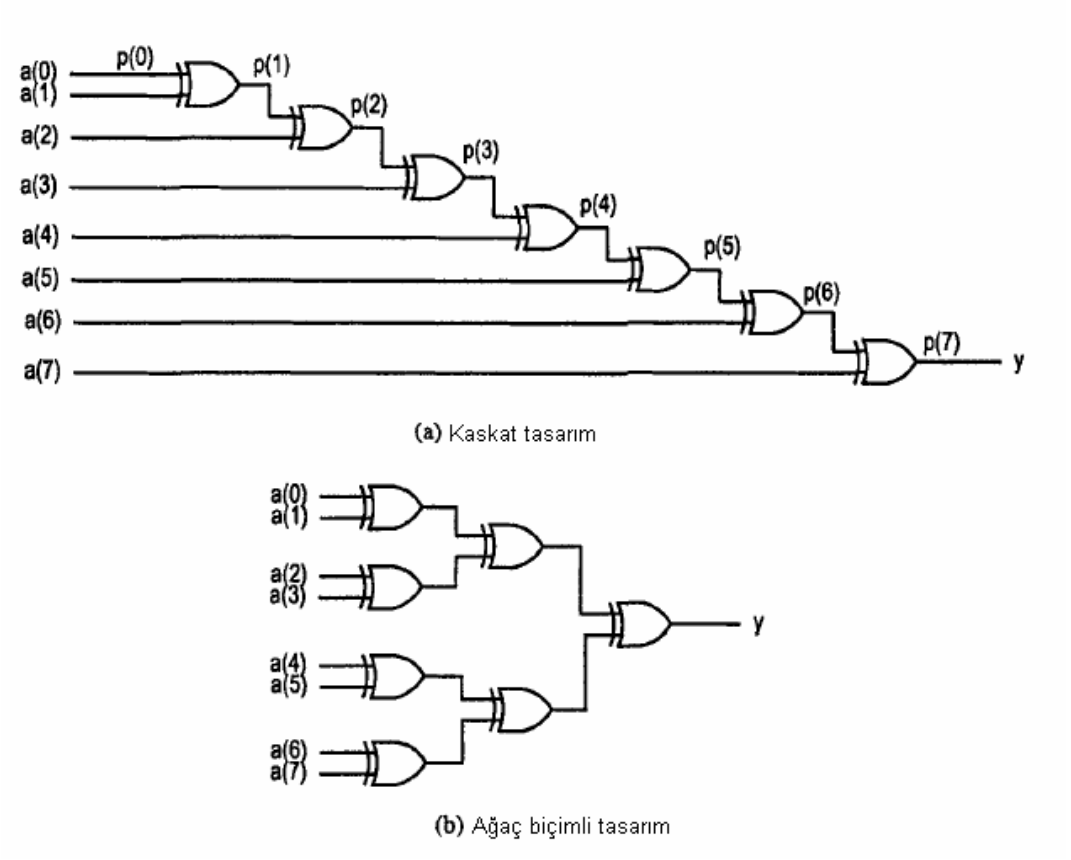
5.1.3.4 Yerleşim ile ilişkili devre örneği

Bu alt bölümde örnek olarak giriş sinyalinin tüm bitleri arasında xor işlemi yapan bir devre incelenecektir. a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 olarak adlandırılan 8 bitlik giriş sinyaline xor operasyonunun uygulanması girişte tek sayıda '1' olması durumunda '1' diğer durumlarda ise '0' üreten bir devreyi tanımlar. Bu tarz bir devre parite biti üretmek için kullanılabilir. İlk tasarım Şekil 5.3' te görüldüğü gibidir.

```
1   library ieee;
2   use ieee.std_logic_1164.all ;
3
4   entity reduced_xor is
5   port (
6   a: in std_logic_vector (7 downto 0) ;
7   y: out std_logic
8   ) ;
9   end reduced_xor ;
10
11  architecture cascade_arch of reduced_xor is
12  begin
13
14  y <= a(0) xor a(1) xor a(2) xor a(3) xor a(4) xor a(5) xor a(6) xor a(7)
15
16  end cascade_arch;
```

Şekil 5.3:
Yerleşimi organize edilmemiş tasarım

Her ne kadar bu tasarım minimum sayıda xor kapısı kullansa da yüksek propogasyon gecikmesine sahiptir. Şekil 5.4(a) da görülebileceği gibi propogasyon gecikmesi birbirine kaskat olarak bağlanmış 7 xor kapısından oluşmaktadır. Dolayısı ile gecikme O(n) mertebesindedir. Girişin sayısı arttıkça gecikmede artacaktır. Xor operasyonunun birleşim özelliğinden faydalanarak operasyon sıralamasını dilediğimiz biçimde değiştirebiliriz. Bu göz önünde bulundurularak başlangıçtaki tasarım ağaç biçimine getirilerek kritik yolun uzunluğu ve dolayısı ile propogasyon gecikmesi azaltılabilir. Bunun için VHDL kodunda parantezler kullanılarak operasyonlar istenen sıralama ile gerçeklenmek için zorlanabilir. Düzenlenmiş kodun mimari kısımı Şekil 5.5' te verilmiştir.



Şekil 5.4:
İki tasarımın şematik gösterimi

```
architecture tree_arch of reduced_xor is
begin

y <= ((a(0) xor a(1)) xor (a(2) xor a(3))) xor ((a(4) xor a(5)) xor (a(6) xor a(7)))

end tree_arch;
```

Şekil 5.5:
Ağaç biçimli tasarımın mimari kısmı

Bu yeni tasarımda kritik yol 3 xor kapısına düşmüştür. Fazladan donanım kaynağı kullanmadan performansdaki artış bu tasarımın ilk tasarımdan daha iyi olduğunu açıkça göstermektedir. Yeni tasarımda propogasyon gecikmesi $O(\log_2 n)$ mertebesine inmiştir. Bu örnekteki devre basit olduğundan sentezleyici yazılımı ilk tasarımıda optimize ederek ikinci tasarım biçiminde sentezleyecektir. Ancak daha karmaşık devrelerde sentezleyici yazılımı optimizasyon konusunda yetersiz kalabilir.

5.1.4 Hibrit GS-NLMS algoritmasının VHDL tanımı

Bu alt bölümde hibrit GS-NLMS algoritmasının VHDL tanımının oluşturulması izah edilecektir.

5.1.4.1 Saat organizasyonu

Hibrit GS-NLMS algoritmasının VHDL tanımı yapılmadan önce veri bağımlılıkları incelenerek saat organizasyonu belirlenmelidir.

M=11 uzunluklu süzgeç için ve G=3 olarak seçilmesi durumunda algoritma en az 6 saat çevriminde tanımlanabilir. Bunlar

- **r** ve **p** vektörlerinin güncellenmesi
- w_5 katsayısının güncellenmesi (GS ile)
- w_6 katsayısının güncellenmesi (GS ile)
- w_7 katsayısının güncellenmesi (GS ile)
- Güncellenen katsayılarla hata ifadesinin hesaplanması
- Kalan katsayıların güncellenmesi (NLMS ile)

dir.

Her ne kadar bir hibrit GS-NLMS iterasyonu 6 saat çevriminde tamamlanabilse de kritik yolun belirlenmesinden sonra bu saat çevrimleri performansı arttırmak için çoğaltılabilir. Yapılan çalışmalarda üzerinde bulunan iterasyonda girişe gelen x değerinin süzgeç çıkışındaki değeri olan y işareti ayrı bir çıkış olarak belirlenmiştir. Bu ise ayrı bir saat çevriminde yapıldığından toplam olarak minimum 7 saat çevriminde bir iterasyon adımı tamamlanabilmektedir.

Oluşturulacak olan VHDL tanımında kritik yol w_5,w_6,w_7 ve hata ifadelerinin hesaplanması adımlarındaki ardışık çarpma ve bölme devreleridir. En iyi performansın bulunabilmesi için iki farklı tanım yapılmıştır. Bu tanımlardan ilki 7 saat çevriminde çalışmaktadır. Bu devrede çarpma ve bölme devreleri kaskat olarak kritik yolu belirlemiştir. İkinci tanımda ise bölme işlemi ayrı bir saat çevriminde yapılarak kritik yolun kısalması ve dolayısı ile çalışma frekansının artması sağlanmıştır. Çizelge 5.1' de iki tasarımın sentezlenmesi sonucu elde edilen parametreler verilmiştir.

Çizelge 5.1

Bölme devresinin ayrı saat çevriminde çalıştırılması ile oluşan sonuçlar

Toplam saat çevrimi sayısı	**Toplam kombinasyonal ALUT sayısı**	**Toplam yazmaç**	**Maksimum saat frekansı**	**Bir iterasyon için gerekli süre**
7	1747	702	19.56 MHz	0.358 µs
11	2166	680	27.96 MHz	0.393 µs

Sonuçlardan görülebileceği gibi bölme devresini ayrı bir saat çevriminde çalıştırmak her ne kadar frekansı arttırsa da bir iterasyonun gerçekleşmesi için gerekli süre daha da artmıştır. Ayrıca bu yaklaşım sonucu oluşan devrenin alanı oldukça artmıştır.

Bölme devresinin ayrı bir saat çevriminde gerçeklenmesinin performansta artış göstermemesinin sebebi çarpma devrelerinin bölme devrelerine göre çok daha az gecikmelere sahip olmasıdır. Sentezleme aşamasında çarpma devreleri FPGA içine

gömülmüş yapılar üzerinde gerçeklenirken bölme devresi LUT yapılarıyla gerçeklenmiştir. Her iki devre için kritik yol gecikmeleri hesaplandığında birbirine kaskat bağlanmış çarpma devresi ve 4 toplama devresinin gecikmesi ($T_{çarpma}$ +4$T_{toplama}$) 0.016 µs iken sadece bölme devresinin gecikmesi 0.035 µs dir. Bu durum propogasyon gecikmelerini herbir saat çevriminde yaklaşık olarak eşit seviyede tutmayı engellemektedir.

5.1.4.2 Kullanılan veri tipinin belirlenmesi

Hibrit GS-NLMS algoritmasının gerçeklenebilmesi için kullanılacak sayıları temsil edebilecek bir sayı sistemi seçilmek zorundadır. Kayar noktalı sayı sisteminin uygulanmasındaki zorluklar (yüksek kaynak ihtiyacı, yavaşlık) nedeni ile daha basit olan sabit noktalı sayı sistemi tercih edilmiştir.

Bu sayı sisteminde toplam bit genişliği, çoğu modern FPGA üzerinde bulunan 18x18 lik çarpma devrelerinden mümkün olabildiğince faydalanabilmek için , 18 olarak seçilmiştir. Sabit noktanın yeri ise uyarlamalı kanal denkleştirici uygulaması göz önünde bulundurularak en değersiz bitten sonraki 5. bitin arkası olarak belirlenmiştir. Ayrıca en değerli bit işaret biti olarak belirlenmiştir. Sayılar ikinin tümleyeni biçiminde ifade edilecektir. Bu durumda iki sayı arasındaki minimum fark 0.031 olacaktır. Sayıların tam kısımı ise -4096 ile 4095 arasında yer alacaktır. Kullanılacak sayı sisteminin bit yerleşimi Şekil 5.6' da verilmiştir.

İ T T T T T T T T T T T T F F F F F

İ = İşaret biti
T=Tamsayı bitleri
F= Fraksiyon bitleri

Şekil 5.6:
Kullanılacak sayı sisteminin bit yerleşimi

5.1.4.3 Temel mimarinin belirlenmesi

Mimari hakkında genel bir kanıya varmak için her bir saat çevriminde yapılan işlemlere göz atmak gereklidir.

M=11 uzunluklu süzgeçte G=3 olarak seçilmesi durumunda ve 7 saat çevrimi ile tanımlanan tasarımda herbir saat çevriminde aşağıdaki gibi işlemler yapılacaktır.

- y çıkış değerinin belirlenmesi: Bu çevrim boyunca **w** katsayı vektörü ile **x** vektörü çarpıldıktan sonra elde edilen çarpım sonuçları toplanarak çıkış sinyali belirlenir. Süzgeç uzunluğu 11 olduğundan dolayı 11 adet çarpma ve 10 adet toplama işlemi yapılmalıdır.
- **r** ve **p** vektörlerinin güncellenmesi: Bu çevrim boyunca r ve p vektörleri yeni giriş sinyalleri ile güncellenecektir. Bu adımda p_5, p_6, p_7 için 3 adet çarpma ve yine 3 adet toplama işlemi, 7 adet r değeri için 7 çarpma ve toplama işlemi ve son olarak NLMS algoritmasında kullanılmak üzere r_0 değerinin 11 adım önceki değerinin hesaplanması için 1 adet çarpma ve 1 adet toplama işlemi gerekmektedir. Toplam olarak bu adımda 11 çarpma ve 11 toplama işlemine ihtiyaç vardır.
- w_5 katsayısının güncellenmesi (GS ile): Bu saat çevriminde güncellenen r ve p katsayıları kullanılarak w_5 katsayı güncellenmektedir. Bunun için 10 adet çarpma , 9 adet toplama, 1 adet çıkarma ve 1 adet bölme işlemine ihtiyaç vardır.
- w_6 katsayısının güncellenmesi (GS ile) : w_5 ile benzer kaynak ihtiyacı vardır.
- w_7 katsayısının güncellenmesi (GS ile): w_5 ile benzer kaynak ihtiyacı vardır.
- Güncellenen katsayılarla hata ifadesinin hesaplanması: Bu adımda hata ifadesinin bulunması için 11 adet çarpma, 10 adet toplama, 1 adet çıkarma ve 1 adet bölme işlemine ihtiyaç vardır. Her ne kadar ilk saat çevriminde hesaplanan y değeri kullanılarak bu adım daha az (3 çarpma,10 toplama ve 1 çıkarma ve 1 bölme) işlemle gerçeklenebilse de bu yaklaşım uygulamada daha fazla kaynak ihtiyacına neden olmuştur. Çizelge 5.2' de 1. saat çevriminde elde edilen

çarpım işlemi sonuçlarının 6.saat çevriminde kullanılması ve kullanılmaması durumlarında elde edilen sentezleme sonuçları verilmiştir .

- Kalan katsayıların güncellenmesi (NLMS ile): Bu adımda 8 adet w katsayısı güncellenmektedir. Hata ifadesinin giriş vektörü **x** ile çarpılması için 8 adet çarpma işlemine ve 8 adet toplama işlemine ihtiyaç vardır.

Çizelge 5.2

Tasarım no	**Toplam kombinasyonal ALUT sayısı**	**Toplam yazmaç**	**Maksimum saat frekansı**	**Bir iterasyon için gerekli süre**
1	1747	702	19.56 MHz	0.358 µs
2	1805	816	18.61 MHz	0.376 µs

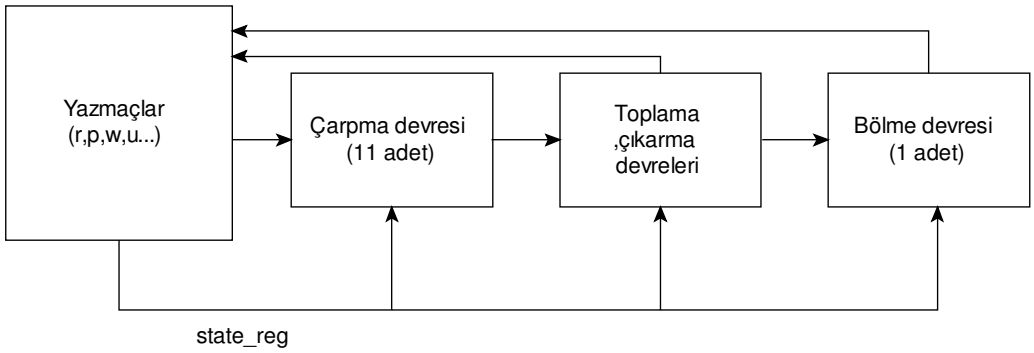
Toplama ve çıkarma devreleri çarpma ve bölme devrelerine göre çok daha az alan kapladıklarından bu devreler için kaynak paylaşımı yapılmayacaktır. Ancak çarpma ve bölme devrelerinde kaynak paylaşımı uygulanarak toplam devre alanı küçültülecektir. Çizelge 5.3' te her bir çevrim için gereken çarpma ve bölme işlemi sayısı gösterilmiştir.

Çizelge 5.3
Çevrimler için gereken çarpma ve bölme sayısı

Çevrim no	**Çarpma sayısı**	**Bölme sayısı**
1	11	0
2	11	0
3	10	1
4	10	1
5	10	1
6	11	1
7	8	0

Çizelgeden de görülebileceği gibi bu saat organizasyonunda devrenin çalışabilmesi için en az 11 çarpma ve 1 bölme devresine ihtiyaç vardır.

Uygulamada kaynak paylaşımı yöntemi ile 11 adet çarpma ve 1 adet bölme devresi kullanılarak VHDL tanımı oluşturulmuştur. Hibrit GS-NLMS algoritmasının temel mimarisi Şekil 5.7' de verilmiştir.



Şekil 5.7:
Temel mimari

5.1.5 Gauss-Seidel ve NLMS algoritmalarının VHDL tanımları

Bu tez çalışmasında önerilen hibrit GS-NLMS algoritmasının donanımsal parametrelerini kıyaslama amacıyla Gauss-Seidel ve NLMS algoritmaları da donanımsal olarak tanımlanmış ve sentezlenmiştir. Her iki tanımda da hibrit GS-NLMS algoritmasının temel mimarisi ve veri tipi kullanılmıştır. Bununla birlikte bu iki algoritmanın gerçeklenmesi için saat organizasyonları belirlenmelidir.

5.1.5.1 NLMS algoritmasının saat organizasyonu

NLMS algoritması aşağıdaki gibi bir saat organizasyonu ile tanımlanmıştır.

- y çıkış ifadesinin ve e hata ifadesinin belirlenmesi: Bu çevrim boyunca **w** katsayı vektörü ile **x** vektörü çarpıldıktan sonra elde edilen çarpım sonuçları toplanarak çıkış sinyali belirlenir. Ardından bu çıkış sinyali d sinyalinden çıkarılarak hata ifadesi e bulunur. Bu çevrimde 11 adet çarpma 1 adet çıkarma ve 10 adet toplama işlemine ihtiyaç vardır.

- e ifadesinin $\mathbf{x}^T(n)\mathbf{x}(n)$ ifadesi ile normalize edilmesi: Bu çevrimde önce $\mathbf{x}^T(n)\mathbf{x}(n)$ ifadesi güncellenmelidir. Bu işlemde her ne kadar 11 adet çarpma işlemine ihtiyaç varmış gibi görünse de aslında bu işlem bir yazmaç vasıtasıyla 2 çarpma 1 toplama ve 1 çıkarma işlemi ile gerçeklenebilir. Eşitlik 5.1 de bu güncelleme prosedürü verilmiştir. $\mathbf{x}^T(n)\mathbf{x}(n)$ ifadesi güncellendikten sonra e hata ifadesi $\mathbf{x}^T(n)\mathbf{x}(n)$ ifadesi ile normalize edilmelidir. Bu işlem ise 1 bölme işlemine ihtiyaç duyar.

$$\mathbf{x}^T(n+1)\mathbf{x}(n+1) = \mathbf{x}^T(n)\mathbf{x}(n) + x(n+1)^2 - x(n-11)^2 \qquad 5.1$$

- w katsayılarının güncellenmesi: Bu çevrimde ise elde edilen normalize hata terimi ile **x**(n) vektörü çarpılır. Çarpma işleminin sonucunda oluşan vektör **w** vektörü ile toplanarak bu vektörün güncellenmesi sağlanır. Bu saat çevriminde 11 adet çarpma ve 11 adet toplama işlemine ihtiyaç vardır.

Görüldüğü gibi NLMS algoritması 3 saat çevriminde tanımlanmıştır.

5.1.5.2 Gauss-Seidel algoritmasının saat organizasyonu

Gauss-Seidel algoritmasının tanımı yapılırken 14 saat çevrimi ile saat organizasyonu belirlenmiştir. Buna göre ilk saat çevriminde giriş vektörü **x** ile süzgeç katsayı vektörü **w** çarpılarak çarpım sonuçları toplanır. Bu şekilde süzgeç çıkışındaki y sinyali belirlenir. Bu işlem için 11 adet çarpma ve 10 adet toplama işlemine ihtiyaç vardır. İkinci saat çevriminde 11 adet **r** elemanı güncellenir . Bunun için 11 adet çarpma ve 11 adet bölme işlemine ihtiyaç duyulur. Üçüncü saat çevriminde benzer şekilde 11 adet **p** elemanının güncellenmesi için 11 çarpma ve 11 toplama işlemine ihtiyaç vardır. Dördüncü ve daha sonraki saat çevrimlerinin her birinde bir w katsayısı güncellenecektir. Bu çevrimlerin herbirinde 10 adet çarpma , 9 adet toplama, 1 adet çıkarma ve 1 adet bölme işlemi gerekir.

5.2 VHDL Tanımlarının Derlenmesi

Derleme işlemi için Altera firmasının ürettiği Quartus II (web edition v7.2 sp3) yazılımı kullanılmıştır. Tasarımlar orta sınıf bir FPGA ailesi olan Altera Stratix II ailesi için derlenmiştir. Her üç tasarımın derleme sonuçları Çizelge 5.4 de verilmiştir.

Çizelge 5.4
Üç algoritmanın derleme sonuçları

	Saat çevrimi	**Fmax (MHz)**	**Kombinasyonal ALUT**	**Yazmaç**	**18x18 Çarpma devresi**	**İterasyon süresi(ns)**
NLMS	3	21.54	1184	471	11	139
Hibrit (G=3)	7	19.06	1798	665	11	367
Gauss-Seidel	14	17.74	2404	888	11	789

Sonuçlardan görülebileceği gibi hibrit GS-NLMS algoritmasının kaynak kullanımı ve bir iterasyonu tamamlama süresi NLMS ve GS algoritmaları arasında kalmaktadır. Yine hibrit GS-NLMS algoritmasının yakınsama oranının GS ve NLMS algoritmaları arasında yer aldığı Şekil 3.5‘ de gösterilmiştir. Maksimum çalışma frekansının belirlenmesinde en baskın faktör bölme devresi olmuştur. Her üç tasarım için de bölme devresi ihtiyacının olması tasarımlar arasında çalışma frekansında büyük farkların oluşmasını engellemiştir. Buna karşın giriş sinyalinin maksimum örneklenme hızı Çizelge 5.5’ de verildiği gibi daha yüksek farklar içermektedir.

Çizelge 5.5
Üç algoritma için giriş sinyalinin maksimum örneklenme hızları

	Giriş sinyalinin maksimum örneklenme hızı (MHz)
NLMS	7.18
Hibrit (G=3)	2.72
Gauss-Seidel	1.26

5.3 VHDL Tanımlarının Benzetim Ortamında Doğrulanması

Bu bölümde sentezlenen hibrit GS-NLMS, NLMS ve Gauss-Seidel algoritmalarının doğrulanması amacı ile benzetimler yapılmıştır. Benzetim için Bölüm 3' te Matlab ortamında uygulanan uyarlamalı kanal denkleştirici problemi kullanılmıştır. Problemin detayları Bölüm 3.2.1' de verilmiştir.

Benzetim için Mentor Graphics firması tarafından üretilen Modelsim (Altera web edition v 6.1g) yazılımı kullanılmıştır.

5.3.1 Benzetim verilerinin oluşturulması

VHDL ile tanımlanan hibrit GS-NLMS, NLMS ve Gauss-Seidel algoritmalarının üçünün de entity tanımları 5 adet port' tan oluşur. Bunlar aşağıda verilmiştir.

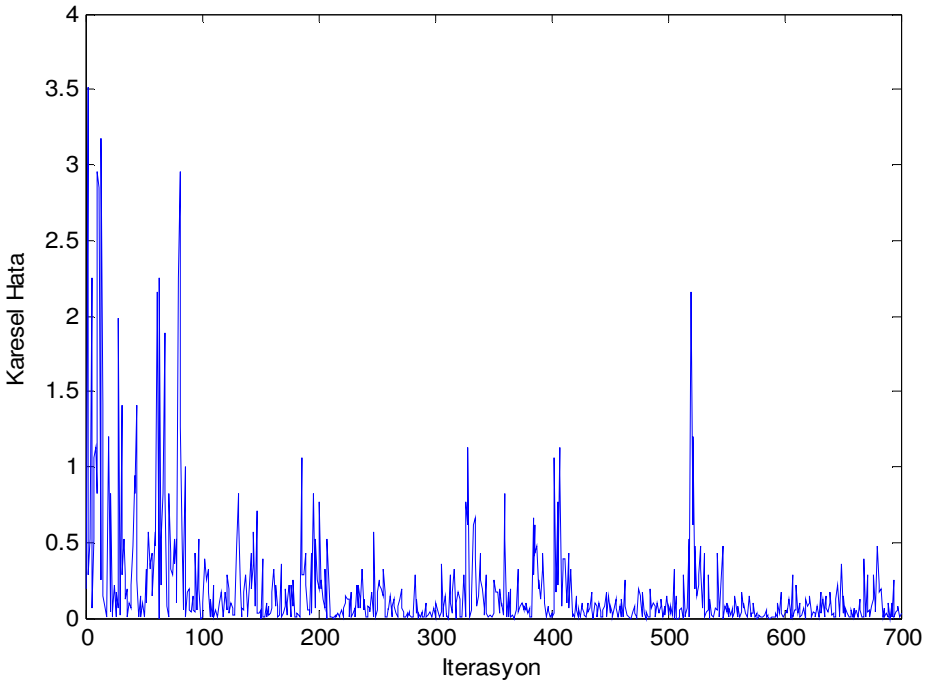
- "x" 18 bitlik giriş sinyali. Bu sinyal kanalın bozucu etkilerine maruz kalmış iletim sinyalidir.
- "d" takip edilmesi istenen 18 bitlik sinyal
- "clk" saat sinyali
- "reset" tüm yazmaçları sıfırlayan sinyal

Bu sinyallerden "x" ve "d" Matlab ortamında oluşturulmuştur. Daha sonra hazırlanan küçük bir program ile bu veriler VHDL tanımlarında kullandığımız sayı sistemine dönüştürülüp bir dosyaya yazılmıştır. Son olarak testbench dosyasında bu veri dosyası okunup test altındaki VHDL tanımlarına geçilmiştir. "clk" ve "reset" sinyalleri ise testbench dosyasında oluşturulmuş.

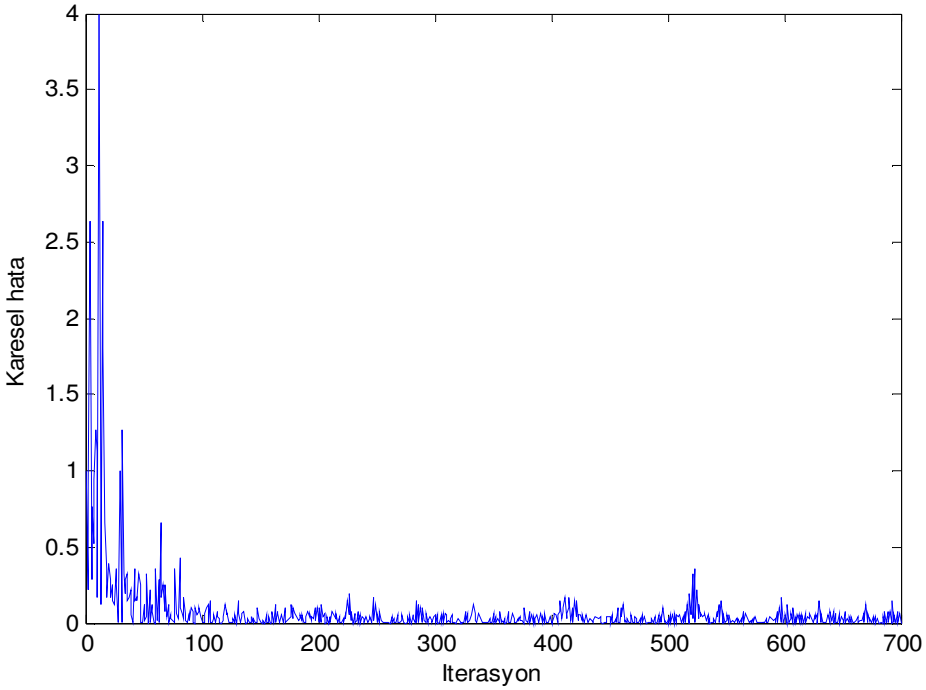
5.3.2 Benzetim sonuçları

Oluşturulan testbench dosyası sadece giriş sinyallerini üretmeyip aynı zamanda çıkış sinyali "y"nin her iterasyonun sonucunda belirlenen değerini bir dosyaya yazmaktadır. Bu çıkış dosyası Matlab ortamında oluşturulan bir program sayesinde Matlab ortamına alınmakta ve karesel hata belirlenmektedir. Bölüm 3' te 1000 deneyin ortalaması ile

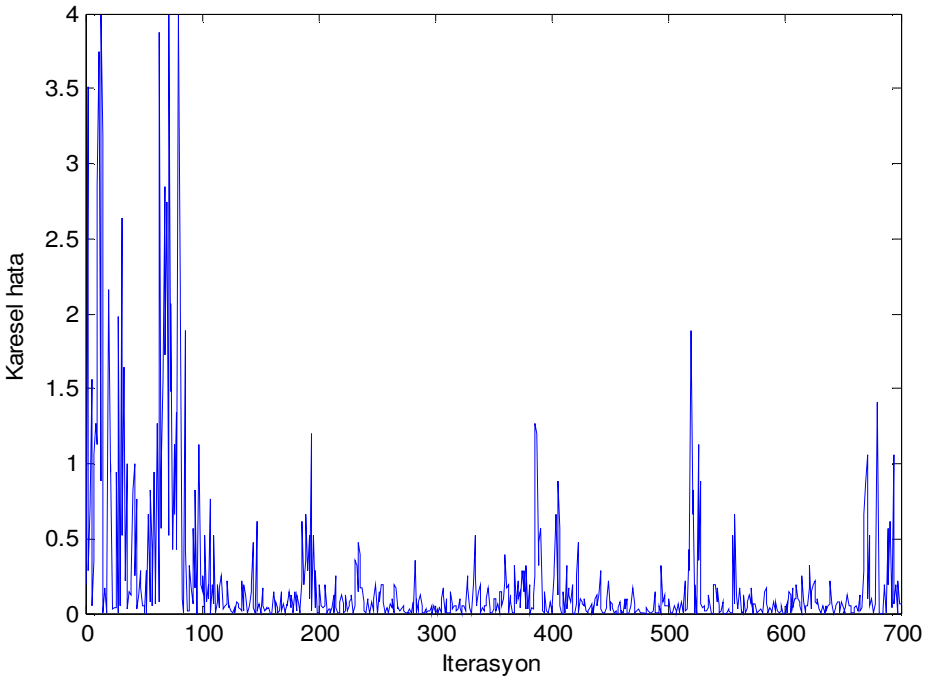
belirlenen bu hata sinyali VHDL benzetiminde sadece 1 deney ile oluşturulmuştur. Hibrit GS-NLMS algoritması, Gauss-Seidel algoritması ve NLMS algoritmalarının benzetim sonuçları sırasıyla Şekil 5.8, Şekil 5.9 ve Şekil 5.10' da verilmiştir.



Şekil 5.8:
Hibrit GS-NLMS algoritmasının benzetim sonuçları



Şekil 5.9:
Gauss-Seidel algoritmasının benzetim sonuçları



Şekil 5.10:
NLMS algoritmasının benzetim sonuçları

6. SONUÇLAR

Bu tez çalışmasında yazılım ve/veya donanım tasarımcılarına adaptif süzgeç algoritmaları konusunda daha iyi çözünürlük sağlayabilecek yeni bir algoritma oluşturulmuştur. Yeni algoritma Gauss-Seidel ve NLMS algoritmalarının ortak kullanımı üzerine kurulmuştur. Bu algoritmanın en önemli yanı işlem yükü-yakınsama oranı karşılaştırmasında Gauss-Seidel ve NLMS algoritmaları arasındaki boşluğu doldurmasıdır.

Önerilen yeni algoritma Bölüm 3' te tanımlanmıştır. Yine aynı bölümde yazılımsal olarak benzetimler yapılmıştır. Algoritmanın en önemli parametrelerinden biri G parametresidir. Bu parametre tasarımcıya işlem yükü-yakınsama oranı takasında daha özgür davranabilme şansı tanımaktadır. G parametresi arttıkça hibrit GS-NLMS algoritmasının işlem yükü de artacak, buna karşın yakınsama oranı da iyileşecektir.

Bölüm 5' de Hibrit GS-NLMS algoritması bu kez donanım olarak tasarlanmıştır. FPGA üzerinde elde edilen sonuçlar yine algoritmanın Gauss-Seidel ve NLMS algoritmaları arasında işlem yükü ve yakınsama oranına sahip olduğunu göstermektedir. Donanımsal tasarımda en büyük sorun bölme devrelerinin yavaşlığı olmuştur. Tasarımların tümünde herbir bölme işlemi bir önceki bölme işleminin sonucuna ihtiyaç duyduğundan bu işlemleri paralel yapmak ya da en azından pipeline yapısına sokmak mümkün olmamıştır. Gauss-Seidel, NLMS ve Hibrit GS-NLMS algoritmalarının her üçü de bölme devresine ihtiyaç duymaktadır. Gelecekte FPGA' ların içine gömülü bölme devreleri ya da FPU (Floating point unit) eklenmesi durumunda her üç algoritmanın performansıda oldukça artacaktır.

KAYNAKLAR

Anonim , (2007) *Stratix II Device Handbook Volume 1-2*, Altera Corp.

Anonim, (2008) *Quartus II Development Software Handbook* , Altera Corp.

Bose, T. (2004) *Digital Signal and Image Processing*, John Wiley, New Jersey.

Chu P. P. , (2006) *RTL Hardware Design Using VHDL* , Wiley-Interscience, New Jersey.

Diniz, P. S. R. (1997) *Adaptive Filtering: Algorithms and Practical Implementation*, Kluwer Academic Publishers, Boston.

Farhang-Boroujeny, B. (1998) *Adaptive Filters: Theory and Applications*, John Wiley & Sons, Chicester.

Golub, G. H. and Van Loan, C. F. (1996) *Matrix Computations*, 3rd Ed., John Hopkins University Press, Baltimore and London.

Goodwin, G. C. and Sin, K. S. (1984) *Adaptive Filtering, Prediction and Control*, Prentice-Hall, Englewood Cliffs, New Jersey.

Hatun, M. ve Koçal, O. H. (2005) Adaptif filtrelerde Gauss-Seidel algoritmasının stokastik yakınsama analizi, *Uludağ Üniversitesi Mühendislik-Mimarlık Fakültesi Dergisi*, 10(2), 87-92.

Haykin, S. (2002) *Adaptive Filter Theory*, 4th Ed., Prentice-Hall, Upper Saddle River, New Jersey.

Haykin, S., Widrow, B. (editors). (2003) *Least-Mean-Square Adaptive Filters*, Wiley-Interscience, New Jersey.

Koçal, O. H. (1998) A new approach to least squares adaptive filtering, *IEEE International Symposium on Circuits and Systems*, Monterey, California, 261-264.

Maxfield C. , (2004) *Design Warrior's Guide to FPGA*, Mentor Graphics Corporation and Xilinx Inc.

Perry D. L. , (2002) *VHDL Programming by Example,*McGraw Hill ,New York.

Treichler, J. R., Johnson, C. R., Larimore, M. G. (1987) *Theory and Design of Adaptive Filters*, Wiley-Interscience, New York.

Widrow, B., Stearns, S. D. (1985) *Adaptive Signal Processing*, Prentice-Hall, Upper Saddle River, New Jersey.

Xu, G. F., Bose, T., Schroeder, J. (1998) Channel equalization using an Euclidean direction search based adaptive algorithm, *IEEE Global Telecommunication Conference,* 6, 3063-3068.

EK-1 VHDL Tanımlarında Ortak Kullanılan Çarpma Devrelerinin Tanımları

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multa11 is
generic(inputsize:natural;inputbigwidth:natural;inputsmallwidth:natural );
port (a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11:in
std_logic_vector(inputsize-1 downto 0);

r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11:out std_logic_vector(inputsize-1 downto 0));

end multa11;
architecture arch_multa11 of multa11 is
signal
mult1o,mult2o,mult3o,mult4o,mult5o,mult6o,mult7o,mult8o,mult9o,mult10o,mult11o:signed(2*inputsize
-1 downto 0);
signal
a1s,a2s,a3s,a4s,a5s,a6s,a7s,a8s,a9s,a10s,a11s,b1s,b2s,b3s,b4s,b5s,b6s,b7s,b8s,b9s,b10s,b11s:signed(input
size-1 downto 0);
begin
a1s<=signed(a1);
a2s<=signed(a2);
a3s<=signed(a3);
a4s<=signed(a4);
a5s<=signed(a5);
a6s<=signed(a6);
a7s<=signed(a7);
a8s<=signed(a8);
a9s<=signed(a9);
a10s<=signed(a10);
a11s<=signed(a11);

b1s<=signed(b1);
b2s<=signed(b2);
b3s<=signed(b3);
b4s<=signed(b4);
b5s<=signed(b5);
b6s<=signed(b6);
b7s<=signed(b7);
b8s<=signed(b8);
b9s<=signed(b9);
b10s<=signed(b10);
b11s<=signed(b11);

mult1o<=a1s*b1s;
mult2o<=a2s*b2s;
mult3o<=a3s*b3s;
mult4o<=a4s*b4s;
mult5o<=a5s*b5s;
mult6o<=a6s*b6s;
mult7o<=a7s*b7s;
mult8o<=a8s*b8s;
mult9o<=a9s*b9s;
mult10o<=a10s*b10s;
mult11o<=a11s*b11s;
```

```
r1(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult1o(2*inputsize-1)&
mult1o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r1(inputsmallwidth-1 downto 0)<=std_logic_vector (mult1o(2*inputsmallwidth-1 downto
inputsmallwidth));
r2(inputsize-1 downto inputsmallwidth)<=  std_logic_vector (mult2o(2*inputsize-1)&
mult2o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r2(inputsmallwidth-1 downto 0)<=std_logic_vector (mult2o(2*inputsmallwidth-1 downto
inputsmallwidth));
r3(inputsize-1 downto inputsmallwidth)<=  std_logic_vector( mult3o(2*inputsize-1)&
mult3o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r3(inputsmallwidth-1 downto 0)<=std_logic_vector (mult3o(2*inputsmallwidth-1 downto
inputsmallwidth));
r4(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult4o(2*inputsize-1)&
mult4o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r4(inputsmallwidth-1 downto 0)<=std_logic_vector (mult4o(2*inputsmallwidth-1 downto
inputsmallwidth));
r5(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult5o(2*inputsize-1)&
mult5o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r5(inputsmallwidth-1 downto 0)<=std_logic_vector (mult5o(2*inputsmallwidth-1 downto
inputsmallwidth));
r6(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult6o(2*inputsize-1)&
mult6o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r6(inputsmallwidth-1 downto 0)<=std_logic_vector (mult6o(2*inputsmallwidth-1 downto
inputsmallwidth));
r7(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult7o(2*inputsize-1)&
mult7o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r7(inputsmallwidth-1 downto 0)<=std_logic_vector (mult7o(2*inputsmallwidth-1 downto
inputsmallwidth));
r8(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult8o(2*inputsize-1)&
mult8o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r8(inputsmallwidth-1 downto 0)<=std_logic_vector (mult8o(2*inputsmallwidth-1 downto
inputsmallwidth));
r9(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult9o(2*inputsize-1)&
mult9o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r9(inputsmallwidth-1 downto 0)<=std_logic_vector (mult9o(2*inputsmallwidth-1 downto
inputsmallwidth));
r10(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult10o(2*inputsize-1)&
mult10o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r10(inputsmallwidth-1 downto 0)<=std_logic_vector (mult10o(2*inputsmallwidth-1 downto
inputsmallwidth));
r11(inputsize-1 downto inputsmallwidth)<= std_logic_vector (mult11o(2*inputsize-1)&
mult11o(2*inputsize-inputbigwidth-2 downto 2*inputsmallwidth));
r11(inputsmallwidth-1 downto 0)<=std_logic_vector (mult11o(2*inputsmallwidth-1 downto
inputsmallwidth));
end arch_multa11;
```

EK-2 VHDL Tanımlarında Ortak Kullanılan Bölme Devresinin Tanımı

Tasarımlarda lpm_divide megafunction ı kullanılmıştır. Bu megafunction' ı kullanan entity tanımı aşağıda verilmiştir.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.all;

ENTITY divider IS
	PORT
	(
		denom		: IN STD_LOGIC_VECTOR (12 DOWNTO 0);
		numer		: IN STD_LOGIC_VECTOR (17 DOWNTO 0);
		quotient	: OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
		remain		: OUT STD_LOGIC_VECTOR (12 DOWNTO 0)
	);
END divider;


ARCHITECTURE SYN OF divider IS

	SIGNAL sub_wire0	: STD_LOGIC_VECTOR (17 DOWNTO 0);
	SIGNAL sub_wire1	: STD_LOGIC_VECTOR (12 DOWNTO 0);



	COMPONENT lpm_divide
	GENERIC (
		lpm_drepresentation		: STRING;
		lpm_hint		: STRING;
		lpm_nrepresentation		: STRING;
		lpm_type		: STRING;
		lpm_widthd		: NATURAL;
		lpm_widthn		: NATURAL
	);
	PORT (
			denom	: IN STD_LOGIC_VECTOR (12 DOWNTO 0);
			quotient : OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
			remain	: OUT STD_LOGIC_VECTOR (12 DOWNTO 0);
			numer	: IN STD_LOGIC_VECTOR (17 DOWNTO 0)
	);
	END COMPONENT;

BEGIN
	quotient    <= sub_wire0(17 DOWNTO 0);
	remain    <= sub_wire1(12 DOWNTO 0);

	lpm_divide_component : lpm_divide
	GENERIC MAP (
		lpm_drepresentation => "SIGNED",
		lpm_hint => "MAXIMIZE_SPEED=6,LPM_REMAINDERPOSITIVE=TRUE",
		lpm_nrepresentation => "SIGNED",
```

```
		lpm_type => "LPM_DIVIDE",
		lpm_widthd => 13,
		lpm_widthn => 18
	)
	PORT MAP (
		denom => denom,
		numer => numer,
		quotient => sub_wire0,
		remain => sub_wire1
	);


END SYN;
```

EK-3 Hibrit GS-NLMS Algoritmasının VHDL Tanımı

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity hibritbolmebutun is
generic(inputsize:natural:=18;inputbigwidth:natural:=13;inputsmallwidth:natural:=5);
port(
x:in std_logic_vector(inputsize-1 downto 0);
d:in std_logic_vector(inputsize-1 downto 0);
y:out std_logic_vector(inputsize-1 downto 0);
clk,reset:in std_logic
);
end hibritbolmebutun;
architecture arch_hibrit of hibritbolmebutun is

signal
u0,u1,u2,u3,u4,u5,temp2,tempnext,u6,u7,u8,u9,u10,u11,u11next,u0next,operand1next,u1next,u2next,u3n
ext,u4next,u5next,u6next,u7next,u8next,u9next,u10next,temp,r0,r1,r2,r3,r4,r5,r6,r011,r0next,r1next,r2ne
xt,r3next,r4next,r5next,r6next,r011next,p5,p6,p7,p5next,p6next,p7next,e,enext,w1,w2,w3,w4,w5,w6,w7,
w8,w9,w10,w11,w1next,w2next,w3next,w4next,w5next,w6next,w7next,w8next,w9next,w10next,w11nex
t,mula1,mula2,mula3,mula4,mula5,mula6,mula7,mula8,mula9,mula10,mula11,mulb1,mulb2,mulb3,mulb
4,mulb5,mulb6,mulb7,mulb8,mulb9,mulb10,mulb11,mulo1,mulo2,mulo3,mulo4,mulo5,mulo6,mulo7,mu
lo8,mulo9,mulo10,mulo11,result:std_logic_vector(17 downto 0);

type statetype is (ser,srp,sw5,sw6,sw7,se,slms);
signal statereg,statenext:statetype;
signal operand2next:std_logic_vector(12 downto 0);
component multa11 is
generic(inputsize:natural;inputbigwidth:natural;inputsmallwidth:natural );
port (a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11:in
std_logic_vector(inputsize-1 downto 0);
r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11:out std_logic_vector(inputsize-1 downto 0));
end component;

component divider1
        PORT
        (
                denom           : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                numer           : IN STD_LOGIC_VECTOR (17 DOWNTO 0);
                quotient        : OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
                remain          : OUT STD_LOGIC_VECTOR (12 DOWNTO 0)
        );
end component;



begin
process(clk,reset)
begin
if reset='1' then
r0<= (5=>'1',others=>'0');
```

```
r1<=(others=>'0');
r2<=(others=>'0');
r3<=(others=>'0');
r4<=(others=>'0');
r5<=(others=>'0');
r6<= (others=>'0');
r011<=(others=>'0');
p5<=(others=>'0');
p6<=(others=>'0');
p7<=(others=>'0');
e<=(others=>'0');
w1<= (others=>'0');
w2<=(others=>'0');
w3<=(others=>'0');
w4<=(others=>'0');
w5<=(others=>'0');
w6<=(others=>'0');
w7<= (others=>'0');
w8<=(others=>'0');
w9<=(others=>'0');
w10<=(others=>'0');
w11<=(others=>'0');
statereg<=ser;
u0<=(others=>'0');
u1<=(others=>'0');
u2<=(others=>'0');
u3<=(others=>'0');
u4<=(others=>'0');
u5<=(others=>'0');
u6<=(others=>'0');
u7<=(others=>'0');
u8<=(others=>'0');
u9<=(others=>'0');
u10<=(others=>'0');
u11<=(others=>'0');
temp<=(others=>'0');
elsif (clk'event and clk='1') then
statereg<=statenext;
r0<= r0next;
r1<=r1next;
r2<=r2next;
temp<=tempnext;
r3<=r3next;
r4<=r4next;
r5<=r5next;
r6<= r6next;
r011<=r011next;
p5<=p5next;
p6<=p6next;
p7<=p7next;
e<=enext;
w1<= w1next;
w2<=w2next;
w3<=w3next;
w4<=w4next;
w5<=w5next;
w6<=w6next;
w7<= w7next;
```

```
w8<=w8next;
w9<=w9next;
w10<=w10next;
w11<=w11next;
u0<=u0next;
u1<=u1next;
u2<=u2next;
u3<=u3next;
u4<=u4next;
u5<=u5next;
u6<=u6next;
u7<=u7next;
u8<=u8next;
u9<=u9next;
u10<=u10next;
u11<=u11next;
end if;
end process;
process(statereg,x,result,temp2,u11,u11next,enext,operand1next,operand2next,mulo1,mulo2,mulo3,mulo
4,mulo5,mulo6,mulo7,mulo8,mulo9,mulo10,mulo11,mula1,mula2,mula3,mula4,mula5,mula6,mula7,mul
a8,mula9,mula10,mula11,mulb1,mulb2,mulb3,mulb4,mulb5,mulb6,mulb7,mulb8,mulb9,mulb10,mulb11,
u0,temp,r1,r2,r3,r4,r5,r6,p5,p6,p7,r0,r011,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,u1,u2,u3,u4,u5,u6,u
7,u8,u9,u10,d,e)
begin
operand1next<=(others=>'0');
operand2next<=(others=>'0');
temp2<=(others=>'0');
tempnext<=temp;
r0next<= r0;
u11next<=u11;
r1next<=r1;
r2next<=r2;
r3next<=r3;
r4next<=r4;
r5next<=r5;
r6next<= r6;
r011next<=r011;
p5next<=p5;
p6next<=p6;
p7next<=p7;
w1next<= w1;
w2next<=w2;
w3next<=w3;
w4next<=w4;
w5next<=w5;
w6next<=w6;
w7next<= w7;
w8next<=w8;
w9next<=w9;
w10next<=w10;
w11next<=w11;
enext<=e;
statenext<=ser;

u0next<=u0;
u1next<=u1;
u2next<=u2;
u3next<=u3;
```

```
u4next<=u4;
u5next<=u5;
u6next<=u6;
u7next<=u7;
u8next<=u8;
u9next<=u9;
u10next<=u10;

case statereg is

when ser=>
mula1<=x;
mulb1<=w1;
mula2<=u0;
mulb2<=w2;
mula3<=u1;
mulb3<=w3;
mula4<=u2;
mulb4<=w4;
mula5<=u3;
mulb5<=w5;
mula6<=u4;
mulb6<=w6;
mula7<=u5;
mulb7<=w7;
mula8<=u6;
mulb8<=w8;
mula9<=u7;
mulb9<=w9;
mula10<=u8;
mulb10<=w10;
mula11<=u9;
mulb11<=w11;
y<=std_logic_vector((((signed(mulo1)+signed(mulo2))+signed(mulo11))+((signed(mulo3)+signed(mulo
4))+(signed(mulo5)+signed(mulo6)))+((signed(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10)
)));
statenext<=srp;
u0next<=x;
u1next<=u0;
u2next<=u1;
u3next<=u2;
u4next<=u3;
u5next<=u4;
u6next<=u5;
u7next<=u6;
u8next<=u7;
u9next<=u8;
u10next<=u9;
u11next<=u10;



when srp=>
mula1<=u0;
mulb1<=u0;
mula2<=u0;
mulb2<=u1;
mula3<=u0;
```

```
mulb3<=u2;
mula4<=u0;
mulb4<=u3;
mula5<=u0;
mulb5<=u4;
mula6<=u0;
mulb6<=u5;
mula7<=u0;
mulb7<=u6;
mula8<=d;
mulb8<=u4;
mula9<=d;
mulb9<=u5;
mula10<=d;
mulb10<=u6;
mula11<=u11;
mulb11<=u11;

r0next<=std_logic_vector(signed(r0)+signed(mulo1));
r1next<=std_logic_vector(signed(r1)+signed(mulo2));
r2next<=std_logic_vector(signed(r2)+signed(mulo3));
r3next<=std_logic_vector(signed(r3)+signed(mulo4));
r4next<=std_logic_vector(signed(r4)+signed(mulo5));
r5next<=std_logic_vector(signed(r5)+signed(mulo6));
r6next<=std_logic_vector(signed(r6)+signed(mulo7));
r011next<=std_logic_vector(signed(r011)+signed(mulo11));
p5next<=std_logic_vector(signed(p5)+signed(mulo8));
p6next<=std_logic_vector(signed(p6)+signed(mulo9));
p7next<=std_logic_vector(signed(p7)+signed(mulo10));
statenext<=sw5;

when sw5=>
mula1<=r4;
mulb1<=w1;
mula2<=r3;
mulb2<=w2;
mula3<=r2;
mulb3<=w3;
mula4<=r1;
mulb4<=w4;
mula5<=r1;
mulb5<=w6;
mula6<=r2;
mulb6<=w7;
mula7<=r3;
mulb7<=w8;
mula8<=r4;
mulb8<=w9;
mula9<=r5;
mulb9<=w10;
mula10<=r6;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');
```

```
operand1next<=std_logic_vector((signed(p5)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));
operand2next<=r0(17 downto 5);
w5next<=result;
statenext<=sw6;


when sw6=>
mula1<=r5;
mulb1<=w1;
mula2<=r4;
mulb2<=w2;
mula3<=r3;
mulb3<=w3;
mula4<=r2;
mulb4<=w4;
mula5<=r1;
mulb5<=w5;
mula6<=r1;
mulb6<=w7;
mula7<=r2;
mulb7<=w8;
mula8<=r3;
mulb8<=w9;
mula9<=r4;
mulb9<=w10;
mula10<=r5;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

operand1next<=std_logic_vector((signed(p6)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));
operand2next<=r0(17 downto 5);

w6next<=result;
statenext<=sw7;


when sw7=>
mula1<=r6;
mulb1<=w1;
mula2<=r5;
mulb2<=w2;
mula3<=r4;
mulb3<=w3;
mula4<=r3;
mulb4<=w4;
mula5<=r2;
mulb5<=w5;
mula6<=r1;
mulb6<=w6;
mula7<=r1;
mulb7<=w8;
mula8<=r2;
mulb8<=w9;
```

```
mula9<=r3;
mulb9<=w10;
mula10<=r4;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

operand1next<=std_logic_vector((signed(p7)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));
operand2next<=r0(17 downto 5);

w7next<=result;

statenext<=se;


when se =>
mula1<=u0;
mulb1<=w1;
mula2<=u1;
mulb2<=w2;
mula3<=u2;
mulb3<=w3;
mula4<=u3;
mulb4<=w4;
mula5<=u4;
mulb5<=w5;
mula6<=u5;
mulb6<=w6;
mula7<=u6;
mulb7<=w7;
mula8<=u7;
mulb8<=w8;
mula9<=u8;
mulb9<=w9;
mula10<=u9;
mulb10<=w10;
mula11<=u10;
mulb11<=w11;

enext<=std_logic_vector(signed(d)-
(((signed(mulo1)+signed(mulo2))+signed(mulo11))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+s
igned(mulo6)))+((signed(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10)))));

operand1next<=enext;
temp2<=std_logic_vector(signed(r0)-signed(r011));
operand2next<=temp2(17 downto 5);

tempnext<=result;
statenext<=slms;


when slms =>


mula1<=u0;
mulb1<=temp;
```

```
mula2<=u1;
mulb2<=temp;
mula3<=u2;
mulb3<=temp;
mula4<=u3;
mulb4<=temp;
mula5<=(others=>'0');
mulb5<=(others=>'0');
mula6<=(others=>'0');
mulb6<=(others=>'0');
mula7<=(others=>'0');
mulb7<=(others=>'0');
mula8<=u7;
mulb8<=temp;
mula9<=u8;
mulb9<=temp;
mula10<=u9;
mulb10<=temp;
mula11<=u10;
mulb11<=temp;


w1next<=std_logic_vector(signed(w1)+signed(mulo1));
w2next<=std_logic_vector(signed(w2)+signed(mulo2));
w3next<=std_logic_vector(signed(w3)+signed(mulo3));
w4next<=std_logic_vector(signed(w4)+signed(mulo4));
w8next<=std_logic_vector(signed(w8)+signed(mulo8));
w9next<=std_logic_vector(signed(w9)+signed(mulo9));
w10next<=std_logic_vector(signed(w10)+signed(mulo10));
w11next<=std_logic_vector(signed(w11)+signed(mulo11));
statenext<=ser;

end case;
end process;

multa11ins:multa11
generic map(inputsize=>18,inputbigwidth=>13,inputsmallwidth=>5)
port
map(a1=>mula1,a2=>mula2,a3=>mula3,a4=>mula4,a5=>mula5,a6=>mula6,a7=>mula7,a8=>mula8,a9=
>mula9,a10=>mula10,a11=>mula11,b1=>mulb1,b2=>mulb2,b3=>mulb3,b4=>mulb4,b5=>mulb5,b6=>
mulb6,b7=>mulb7,b8=>mulb8,b9=>mulb9,b10=>mulb10,b11=>mulb11,r1=>mulo1,r2=>mulo2,r3=>mu
lo3,r4=>mulo4,r5=>mulo5,r6=>mulo6,r7=>mulo7,r8=>mulo8,r9=>mulo9,r10=>mulo10,r11=>mulo11);
divider1_inst : divider1 PORT MAP (
                        denom    => operand2next,
                        numer    => operand1next,
                        quotient  => result,
                        remain    => open
            );
end arch_hibrit;
```

EK-4 Gauss-Seidel Algoritmasının VHDL Tanımı

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;



entity gs is
generic(inputsize:natural:=18;inputbigwidth:natural:=13;inputsmallwidth:natural:=5);
port(
x:in std_logic_vector(inputsize-1 downto 0);
d:in std_logic_vector(inputsize-1 downto 0);
y:out std_logic_vector(inputsize-1 downto 0);
clk,reset:in std_logic
);

end gs;

architecture Behavioral of gs is
type statetype is (sy,sr,sp,sw1,sw2,sw3,sw4,sw5,sw6,sw7,sw8,sw9,sw10,sw11);
signal statereg,statenext:statetype;
signal
u0,u1,u2,u3,u4,u5,u6,u7,divtemp,divtempnext,u8,u9,u10,ytemp,ynext,u0next,u1next,u2next,u3next,u4ne
xt,u5next,u6next,u7next,u8next,u9next,u10next,temp,r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r0next,r1next,r2next,
r3next,r4next,r5next,r6next,r7next,r8next,r9next,r10next,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p1next,p2n
ext,p3next,p4next,p5next,p6next,p7next,p8next,p9next,p10next,p11next,w1,w2,w3,w4,w5,w6,w7,w8,w9,
w10,w11,w1next,w2next,w3next,w4next,w5next,w6next,w7next,w8next,w9next,w10next,w11next,mula
1,mula2,mula3,mula4,mula5,mula6,mula7,mula8,mula9,mula10,mula11,mulb1,mulb2,mulb3,mulb4,mul
b5,mulb6,mulb7,mulb8,mulb9,mulb10,mulb11,mulo1,mulo2,mulo3,mulo4,mulo5,mulo6,mulo7,mulo8,m
ulo9,mulo10,mulo11,result:std_logic_vector(17 downto 0);
signal operand1,divout,operand1next:std_logic_vector(inputsize-1 downto 0);
signal operand2,operand2next:std_logic_vector(12 downto 0);
component multa11 is
generic(inputsize:natural;inputbigwidth:natural;inputsmallwidth:natural );
port (a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11:in
std_logic_vector(inputsize-1 downto 0);
r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11:out std_logic_vector(inputsize-1 downto 0));
end component;
component divider
        PORT
        (
                denom           : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                numer           : IN STD_LOGIC_VECTOR (17 DOWNTO 0);
                quotient        : OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
                remain          : OUT STD_LOGIC_VECTOR (12 DOWNTO 0)
        );
end component;


begin
process(clk,reset)
begin
if reset='1' then
r0<= (5=>'1',others=>'0');
r1<=(others=>'0');
r2<=(others=>'0');
```

```
r3<=(others=>'0');
r4<=(others=>'0');
r5<=(others=>'0');
r6<= (others=>'0');
r7<= (others=>'0');
r8<= (others=>'0');
r9<= (others=>'0');
r10<= (others=>'0');
ytemp<=(others=>'0');
p1<=(others=>'0');
p2<=(others=>'0');
p3<=(others=>'0');
p4<=(others=>'0');
p5<=(others=>'0');
p6<=(others=>'0');
p7<=(others=>'0');
p8<=(others=>'0');
p9<=(others=>'0');
p10<=(others=>'0');
p11<=(others=>'0');

w1<= (others=>'0');
w2<=(others=>'0');
w3<=(others=>'0');
w4<=(others=>'0');
w5<=(others=>'0');
w6<=(others=>'0');
w7<= (others=>'0');
w8<=(others=>'0');
w9<=(others=>'0');
w10<=(others=>'0');
w11<=(others=>'0');

statereg<=sy;
divtemp<=(others=>'0');
u0<=(others=>'0');
u1<=(others=>'0');
u2<=(others=>'0');
u3<=(others=>'0');
u4<=(others=>'0');
u5<=(others=>'0');
u6<=(others=>'0');
u7<=(others=>'0');
u8<=(others=>'0');
u9<=(others=>'0');
u10<=(others=>'0');

operand1<=(others=>'0');
operand2<=(others=>'0');
elsif (clk'event and clk='1') then
statereg<=statenext;
operand1<=operand1next;
operand2<=operand2next;
r0<= r0next;
r1<=r1next;
r2<=r2next;
r3<=r3next;
r4<=r4next;
```

```
r5<=r5next;
r6<= r6next;
r7<= r7next;
r8<= r8next;
r9<= r9next;
r10<= r10next;
p1<=p1next;
p2<=p2next;
p3<=p3next;
p4<=p4next;
p5<=p5next;
p6<=p6next;
p7<=p7next;
p8<=p8next;
p9<=p9next;
p10<=p10next;
p11<=p11next;
y<=ytemp;
ytemp<=ynext;
w1<= w1next;
w2<=w2next;
w3<=w3next;
w4<=w4next;
w5<=w5next;
w6<=w6next;
w7<= w7next;
w8<=w8next;
w9<=w9next;
w10<=w10next;
w11<=w11next;
u0<=u0next;
u1<=u1next;
u2<=u2next;
u3<=u3next;
u4<=u4next;
u5<=u5next;
u6<=u6next;
u7<=u7next;
u8<=u8next;
u9<=u9next;
u10<=u10next;
divtemp<=divtempnext;
end if;
end process;
process(x,divout,operand1next,divtempnext,operand2next,statereg,ytemp,ynext,mulo1,mulo2,mulo3,mul
o4,mulo5,mulo6,mulo7,mulo8,mulo9,mulo10,mulo11,mula1,mula2,mula3,mula4,mula5,mula6,mula7,mu
la8,mula9,mula10,mula11,mulb1,mulb2,mulb3,mulb4,mulb5,mulb6,mulb7,mulb8,mulb9,mulb10,mulb11
,u0,temp,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,r0,w1,w2,w3,w4,w5,w6,w7,w8
,w9,w10,w11,u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,divtemp,d)
begin
operand1next<=(others=>'0');
operand2next<=(others=>'0');
divtempnext<=divtemp;
ynext <=ytemp;
r0next<= r0;
r1next<=r1;
r2next<=r2;
r3next<=r3;
```

```
r4next<=r4;
r5next<=r5;
r6next<= r6;
r7next<= r7;
r8next<= r8;
r9next<= r9;
r10next<= r10;
p1next<=p1;
p2next<=p2;
p3next<=p3;
p4next<=p4;
p5next<=p5;
p6next<=p6;
p7next<=p7;
p8next<=p8;
p9next<=p9;
p10next<=p10;
p11next<=p11;
w1next<= w1;
w2next<=w2;
w3next<=w3;
w4next<=w4;
w5next<=w5;
w6next<=w6;
w7next<= w7;
w8next<=w8;
w9next<=w9;
w10next<=w10;
w11next<=w11;
statenext<=sy;
u0next<=u0;
u1next<=u1;
u2next<=u2;
u3next<=u3;
u4next<=u4;
u5next<=u5;
u6next<=u6;
u7next<=u7;
u8next<=u8;
u9next<=u9;
u10next<=u10;
temp<=(others=>'0');
case statereg is
when sy=>
mula1<=x;
mulb1<=w1;
mula2<=u0;
mulb2<=w2;
mula3<=u1;
mulb3<=w3;
mula4<=u2;
mulb4<=w4;
mula5<=u3;
mulb5<=w5;
mula6<=u4;
mulb6<=w6;
mula7<=u5;
mulb7<=w7;
```

```
mula8<=u6;
mulb8<=w8;
mula9<=u7;
mulb9<=w9;
mula10<=u8;
mulb10<=w10;
mula11<=u9;
mulb11<=w11;
ynext<=std_logic_vector((((signed(mulo1)+signed(mulo2))+signed(mulo11))+((signed(mulo3)+signed(m
ulo4))+(signed(mulo5)+signed(mulo6)))+((signed(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo
10)))));
statenext<=sr;
u0next<=x;
u1next<=u0;
u2next<=u1;
u3next<=u2;
u4next<=u3;
u5next<=u4;
u6next<=u5;
u7next<=u6;
u8next<=u7;
u9next<=u8;
u10next<=u9;
when sr=>
mula1<=u0;
mulb1<=u0;
mula2<=u0;
mulb2<=u1;
mula3<=u0;
mulb3<=u2;
mula4<=u0;
mulb4<=u3;
mula5<=u0;
mulb5<=u4;
mula6<=u0;
mulb6<=u5;
mula7<=u0;
mulb7<=u6;
mula8<=u0;
mulb8<=u7;
mula9<=u0;
mulb9<=u8;
mula10<=u0;
mulb10<=u9;
mula11<=u0;
mulb11<=u10;
r0next<=std_logic_vector(signed(r0)+signed(mulo1));
r1next<=std_logic_vector(signed(r1)+signed(mulo2));
r2next<=std_logic_vector(signed(r2)+signed(mulo3));
r3next<=std_logic_vector(signed(r3)+signed(mulo4));
r4next<=std_logic_vector(signed(r4)+signed(mulo5));
r5next<=std_logic_vector(signed(r5)+signed(mulo6));
r6next<=std_logic_vector(signed(r6)+signed(mulo7));
r7next<=std_logic_vector(signed(r7)+signed(mulo8));
r8next<=std_logic_vector(signed(r8)+signed(mulo9));
r9next<=std_logic_vector(signed(r9)+signed(mulo10));
r10next<=std_logic_vector(signed(r10)+signed(mulo11));
statenext<=sp;
```

```
when sp=>
mula1<=d;
mulb1<=u0;
mula2<=d;
mulb2<=u1;
mula3<=d;
mulb3<=u2;
mula4<=d;
mulb4<=u3;
mula5<=d;
mulb5<=u4;
mula6<=d;
mulb6<=u5;
mula7<=d;
mulb7<=u6;
mula8<=d;
mulb8<=u7;
mula9<=d;
mulb9<=u8;
mula10<=d;
mulb10<=u9;
mula11<=d;
mulb11<=u10;
p1next<=std_logic_vector(signed(p1)+signed(mulo1));
p2next<=std_logic_vector(signed(p2)+signed(mulo2));
p3next<=std_logic_vector(signed(p3)+signed(mulo3));
p4next<=std_logic_vector(signed(p4)+signed(mulo4));
p5next<=std_logic_vector(signed(p5)+signed(mulo5));
p6next<=std_logic_vector(signed(p6)+signed(mulo6));
p7next<=std_logic_vector(signed(p7)+signed(mulo7));
p8next<=std_logic_vector(signed(p8)+signed(mulo8));
p9next<=std_logic_vector(signed(p9)+signed(mulo9));
p10next<=std_logic_vector(signed(p10)+signed(mulo10));
p11next<=std_logic_vector(signed(p11)+signed(mulo11));
statenext<=sw1;
when sw1=>
mula1<=r1;
mulb1<=w2;
mula2<=r2;
mulb2<=w3;
mula3<=r3;
mulb3<=w4;
mula4<=r4;
mulb4<=w5;
mula5<=r5;
mulb5<=w6;
mula6<=r6;
mulb6<=w7;
mula7<=r7;
mulb7<=w8;
mula8<=r8;
mulb8<=w9;
mula9<=r9;
mulb9<=w10;
mula10<=r10;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');
```

```
divtempnext<=std_logic_vector((signed(p1)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);
w1next<=divout;
statenext<=sw2;


when sw2=>
mula1<=r1;
mulb1<=w1;
mula2<=r1;
mulb2<=w3;
mula3<=r2;
mulb3<=w4;
mula4<=r3;
mulb4<=w5;
mula5<=r4;
mulb5<=w6;
mula6<=r5;
mulb6<=w7;
mula7<=r6;
mulb7<=w8;
mula8<=r7;
mulb8<=w9;
mula9<=r8;
mulb9<=w10;
mula10<=r9;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p2)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);
w2next<=divout;
statenext<=sw3;
when sw3=>
mula1<=r2;
mulb1<=w1;
mula2<=r1;
mulb2<=w2;
mula3<=r1;
mulb3<=w4;
mula4<=r2;
mulb4<=w5;
mula5<=r3;
mulb5<=w6;
mula6<=r4;
mulb6<=w7;
mula7<=r5;
mulb7<=w8;
```

```
mula8<=r6;
mulb8<=w9;
mula9<=r7;
mulb9<=w10;
mula10<=r8;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p3)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);


w3next<=divout;
statenext<=sw4;
when sw4=>
mula1<=r3;
mulb1<=w1;
mula2<=r2;
mulb2<=w2;
mula3<=r1;
mulb3<=w3;
mula4<=r1;
mulb4<=w5;
mula5<=r2;
mulb5<=w6;
mula6<=r3;
mulb6<=w7;
mula7<=r4;
mulb7<=w8;
mula8<=r5;
mulb8<=w9;
mula9<=r6;
mulb9<=w10;
mula10<=r7;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p4)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);


w4next<=divout;
statenext<=sw5;
when sw5=>
mula1<=r4;
mulb1<=w1;
mula2<=r3;
mulb2<=w2;
```

```
mula3<=r2;
mulb3<=w3;
mula4<=r1;
mulb4<=w4;
mula5<=r1;
mulb5<=w6;
mula6<=r2;
mulb6<=w7;
mula7<=r3;
mulb7<=w8;
mula8<=r4;
mulb8<=w9;
mula9<=r5;
mulb9<=w10;
mula10<=r6;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p5)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);

w5next<=divout;
statenext<=sw6;
when sw6=>
mula1<=r5;
mulb1<=w1;
mula2<=r4;
mulb2<=w2;
mula3<=r3;
mulb3<=w3;
mula4<=r2;
mulb4<=w4;
mula5<=r1;
mulb5<=w5;
mula6<=r1;
mulb6<=w7;
mula7<=r2;
mulb7<=w8;
mula8<=r3;
mulb8<=w9;
mula9<=r4;
mulb9<=w10;
mula10<=r5;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p6)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);
```

```
w6next<=divout;
statenext<=sw7;
when sw7=>
mula1<=r6;
mulb1<=w1;
mula2<=r5;
mulb2<=w2;
mula3<=r4;
mulb3<=w3;
mula4<=r3;
mulb4<=w4;
mula5<=r2;
mulb5<=w5;
mula6<=r1;
mulb6<=w6;
mula7<=r1;
mulb7<=w8;
mula8<=r2;
mulb8<=w9;
mula9<=r3;
mulb9<=w10;
mula10<=r4;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p7)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);

w7next<=divout;
statenext<=sw8;
when sw8=>
mula1<=r7;
mulb1<=w1;
mula2<=r6;
mulb2<=w2;
mula3<=r5;
mulb3<=w3;
mula4<=r4;
mulb4<=w4;
mula5<=r3;
mulb5<=w5;
mula6<=r2;
mulb6<=w6;
mula7<=r1;
mulb7<=w7;
mula8<=r1;
mulb8<=w9;
mula9<=r2;
mulb9<=w10;
mula10<=r3;
mulb10<=w11;
```

```
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p8)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);

w8next<=divout;
statenext<=sw9;
when sw9=>
mula1<=r8;
mulb1<=w1;
mula2<=r7;
mulb2<=w2;
mula3<=r6;
mulb3<=w3;
mula4<=r5;
mulb4<=w4;
mula5<=r4;
mulb5<=w5;
mula6<=r3;
mulb6<=w6;
mula7<=r2;
mulb7<=w7;
mula8<=r1;
mulb8<=w8;
mula9<=r1;
mulb9<=w10;
mula10<=r2;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p9)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);

w9next<=divout;
statenext<=sw10;
when sw10=>
mula1<=r9;
mulb1<=w1;
mula2<=r8;
mulb2<=w2;
mula3<=r7;
mulb3<=w3;
mula4<=r6;
mulb4<=w4;
mula5<=r5;
mulb5<=w5;
mula6<=r4;
mulb6<=w6;
```

```
mula7<=r3;
mulb7<=w7;
mula8<=r2;
mulb8<=w8;
mula9<=r1;
mulb9<=w9;
mula10<=r1;
mulb10<=w11;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p10)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);

w10next<=divout;
statenext<=sw11;
when sw11=>
mula1<=r10;
mulb1<=w1;
mula2<=r9;
mulb2<=w2;
mula3<=r8;
mulb3<=w3;
mula4<=r7;
mulb4<=w4;
mula5<=r6;
mulb5<=w5;
mula6<=r5;
mulb6<=w6;
mula7<=r4;
mulb7<=w7;
mula8<=r3;
mulb8<=w8;
mula9<=r2;
mulb9<=w9;
mula10<=r1;
mulb10<=w10;
mula11<=(others=>'0');
mulb11<=(others=>'0');

divtempnext<=std_logic_vector((signed(p11)-
((signed(mulo1)+signed(mulo2))+((signed(mulo3)+signed(mulo4))+(signed(mulo5)+signed(mulo6)))+((s
igned(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mulo10))))));

operand1next<=divtempnext;
operand2next<=r0(inputsize-1 downto 5);

w11next<=divout ;
statenext<=sy;



end case;
```

```
end process;

multa11ins:multa11
generic map(inputsize=>18,inputbigwidth=>13,inputsmallwidth=>5)
port
map(a1=>mula1,a2=>mula2,a3=>mula3,a4=>mula4,a5=>mula5,a6=>mula6,a7=>mula7,a8=>mula8,a9=
>mula9,a10=>mula10,a11=>mula11,b1=>mulb1,b2=>mulb2,b3=>mulb3,b4=>mulb4,b5=>mulb5,b6=>
mulb6,b7=>mulb7,b8=>mulb8,b9=>mulb9,b10=>mulb10,b11=>mulb11,r1=>mulo1,r2=>mulo2,r3=>mu
lo3,r4=>mulo4,r5=>mulo5,r6=>mulo6,r7=>mulo7,r8=>mulo8,r9=>mulo9,r10=>mulo10,r11=>mulo11);
divider_inst : divider PORT MAP (
                        denom    => operand2next,
                        numer    => operand1next,
                        quotient  => divout,
                        remain    => open
            );



end Behavioral;
```

EK-5 NLMS Algoritmasının VHDL Tanımı

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity lms is
generic(inputsize:natural:=18;inputbigwidth:natural:=13;inputsmallwidth:natural:=5);
port(
x:in std_logic_vector(inputsize-1 downto 0);
d:in std_logic_vector(inputsize-1 downto 0);
y:out std_logic_vector(inputsize-1 downto 0);
clk,reset:in std_logic
);
end lms;

architecture Behavioral of lms is
type statetype is (se,snorm,slms);
signal statereg,statenext:statetype;
signal
u0,u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,u11,u11next,u0next,tempnext,multnorm,multnormnext,ytemp,u1next,
u2next,u3next,u4next,u5next,u6next,u7next,u8next,u9next,u10next,temp,e,enext,w1,w2,w3,w4,w5,w6,w
7,w8,w9,w10,w11,w1next,w2next,w3next,w4next,w5next,w6next,w7next,w8next,w9next,w10next,w11n
ext,mula1,mula2,mula3,mula4,mula5,mula6,mula7,mula8,mula9,mula10,mula11,mulb1,mulb2,mulb3,mu
lb4,mulb5,mulb6,mulb7,mulb8,mulb9,mulb10,mulb11,mulo1,mulo2,mulo3,mulo4,mulo5,mulo6,mulo7,
mulo8,mulo9,mulo10,mulo11,result:std_logic_vector(17 downto 0);
signal operand1,divout:std_logic_vector(inputsize-1 downto 0);
signal operand2:std_logic_vector(12 downto 0);
signal normmultout1,normmultout2:std_logic_vector(2*inputsize-1 downto 0);

component multa11 is
generic(inputsize:natural;inputbigwidth:natural;inputsmallwidth:natural );
port (a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11:in
std_logic_vector(inputsize-1 downto 0);
r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11:out std_logic_vector(inputsize-1 downto 0));
end component;

component divider
        PORT
        (
                denom           : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                numer           : IN STD_LOGIC_VECTOR (17 DOWNTO 0);
                quotient        : OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
                remain          : OUT STD_LOGIC_VECTOR (12 DOWNTO 0)
        );
end component;




begin
process(clk,reset)
begin
if reset='1' then

e<=(others=>'0');
```

```
w1<= (others=>'0');
w2<=(others=>'0');
w3<=(others=>'0');
w4<=(others=>'0');
w5<=(others=>'0');
w6<=(others=>'0');
w7<= (others=>'0');
w8<=(others=>'0');
w9<=(others=>'0');
w10<=(others=>'0');
w11<=(others=>'0');
statereg<=se;
temp<=(others=>'0');
u0<=(others=>'0');
u1<=(others=>'0');
u2<=(others=>'0');
u3<=(others=>'0');
u4<=(others=>'0');
u5<=(others=>'0');
u6<=(others=>'0');
u7<=(others=>'0');
u8<=(others=>'0');
u9<=(others=>'0');
u10<=(others=>'0');
u11<=(others=>'0');
multnorm<=(5=>'1',others=>'0');
elsif (clk'event and clk='1') then
statereg<=statenext;
e<=enext;
temp<=tempnext;
w1<= w1next;
w2<=w2next;
w3<=w3next;
w4<=w4next;
w5<=w5next;
w6<=w6next;
w7<= w7next;
w8<=w8next;
w9<=w9next;
w10<=w10next;
w11<=w11next;
u0<=u0next;
u1<=u1next;
u2<=u2next;
u3<=u3next;
u4<=u4next;
u5<=u5next;
u6<=u6next;
u7<=u7next;
u8<=u8next;
u9<=u9next;
u10<=u10next;
u11<=u11next;

multnorm<=multnormnext;
end if;
end process;
```

```
process(statereg,x,multnorm,multnormnext,divout,normmultout2,normmultout1,u11,ytemp,mulo1,mulo2,
mulo3,mulo4,mulo5,mulo6,mulo7,mulo8,mulo9,mulo10,mulo11,mula1,mula2,mula3,mula4,mula5,mula6
,mula7,mula8,mula9,mula10,mula11,mulb1,mulb2,mulb3,mulb4,mulb5,mulb6,mulb7,mulb8,mulb9,mulb
10,mulb11,u0,temp,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,d,e)
begin
mula1<=(others=>'0');
mula2<=(others=>'0');
mula3<=(others=>'0');
mula4<=(others=>'0');
mula5<=(others=>'0');
mula6<=(others=>'0');
mula7<=(others=>'0');
mula8<=(others=>'0');
mula9<=(others=>'0');
mula10<=(others=>'0');
mula11<=(others=>'0');
mulb1<=(others=>'0');
mulb2<=(others=>'0');
mulb3<=(others=>'0');
mulb4<=(others=>'0');
mulb5<=(others=>'0');
mulb6<=(others=>'0');
mulb7<=(others=>'0');
mulb8<=(others=>'0');
mulb9<=(others=>'0');
mulb10<=(others=>'0');
mulb11<=(others=>'0');
ytemp<=(others=>'0');
operand1<=(others=>'0');
operand2<=(others=>'0');
tempnext<=temp;
multnormnext<=multnorm;
w1next<= w1;
w2next<=w2;
w3next<=w3;
w4next<=w4;
w5next<=w5;
w6next<=w6;
w7next<= w7;
w8next<=w8;
w9next<=w9;
w10next<=w10;
w11next<=w11;
enext<=e;
statenext<=se;

u0next<=u0;
u1next<=u1;
u2next<=u2;
u3next<=u3;
u4next<=u4;
u5next<=u5;
u6next<=u6;
u7next<=u7;
u8next<=u8;
u9next<=u9;
u10next<=u10;
u11next<=u11;
```

```
case statereg is
when se=>
mula1<=x;
mulb1<=w1;
mula2<=u0;
mulb2<=w2;
mula3<=u1;
mulb3<=w3;
mula4<=u2;
mulb4<=w4;
mula5<=u3;
mulb5<=w5;
mula6<=u4;
mulb6<=w6;
mula7<=u5;
mulb7<=w7;
mula8<=u6;
mulb8<=w8;
mula9<=u7;
mulb9<=w9;
mula10<=u8;
mulb10<=w10;
mula11<=u9;
mulb11<=w11;


u0next<=x;
u1next<=u0;
u2next<=u1;
u3next<=u2;
u4next<=u3;
u5next<=u4;
u6next<=u5;
u7next<=u6;
u8next<=u7;
u9next<=u8;
u10next<=u9;
u11next<=u10;


ytemp<=std_logic_vector((((signed(mulo1)+signed(mulo2))+signed(mulo11))+((signed(mulo3)+signed(
mulo4))+(signed(mulo5)+signed(mulo6)))+((signed(mulo7)+signed(mulo8))+(signed(mulo9)+signed(mu
lo10)))));
enext<=std_logic_vector(signed(d)-signed(ytemp));
y<=ytemp;
statenext<=snorm;
when snorm=>
mula1<=u0;
mulb1<=u0;
mula2<=u11;
mulb2<=u11;
multnormnext<=std_logic_vector((signed(multnorm)+signed(mulo1))-signed(mulo2));

statenext<=slms;

operand1<=e;
```

```
operand2<=multnormnext(17 downto 5);
tempnext<=divout;
when slms =>



mula1<=u0;
mulb1<=temp;
mula2<=u1;
mulb2<=temp;
mula3<=u2;
mulb3<=temp;
mula4<=u3;
mulb4<=temp;
mula5<=u4;
mulb5<=temp;
mula6<=u5;
mulb6<=temp;
mula7<=u6;
mulb7<=temp;
mula8<=u7;
mulb8<=temp;
mula9<=u8;
mulb9<=temp;
mula10<=u9;
mulb10<=temp;
mula11<=u10;
mulb11<=temp;


w1next<=std_logic_vector(signed(w1)+signed(mulo1));
w2next<=std_logic_vector(signed(w2)+signed(mulo2));
w3next<=std_logic_vector(signed(w3)+signed(mulo3));
w4next<=std_logic_vector(signed(w4)+signed(mulo4));
w5next<=std_logic_vector(signed(w5)+signed(mulo5));
w6next<=std_logic_vector(signed(w6)+signed(mulo6));
w7next<=std_logic_vector(signed(w7)+signed(mulo7));
w8next<=std_logic_vector(signed(w8)+signed(mulo8));
w9next<=std_logic_vector(signed(w9)+signed(mulo9));
w10next<=std_logic_vector(signed(w10)+signed(mulo10));
w11next<=std_logic_vector(signed(w11)+signed(mulo11));



end case;
end process;
multa11ins:multa11
generic map(inputsize=>18,inputbigwidth=>13,inputsmallwidth=>5)
port
map(a1=>mula1,a2=>mula2,a3=>mula3,a4=>mula4,a5=>mula5,a6=>mula6,a7=>mula7,a8=>mula8,a9=
>mula9,a10=>mula10,a11=>mula11,b1=>mulb1,b2=>mulb2,b3=>mulb3,b4=>mulb4,b5=>mulb5,b6=>
mulb6,b7=>mulb7,b8=>mulb8,b9=>mulb9,b10=>mulb10,b11=>mulb11,r1=>mulo1,r2=>mulo2,r3=>mu
lo3,r4=>mulo4,r5=>mulo5,r6=>mulo6,r7=>mulo7,r8=>mulo8,r9=>mulo9,r10=>mulo10,r11=>mulo11);

        divider_inst : divider PORT MAP (
```

```
                denom   => operand2,
                numer   => operand1,
                quotient => divout,
                remain  => open
        );


end Behavioral;
```

EK-6 Benzetim için Oluşturulan dosyalar

EK-6.1 Hibrit GS-NLMS algoritması için oluşturulan testbench dosyası

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
use std.textio.all;
ENTITY bench_vhd IS
END bench_vhd;

ARCHITECTURE behavior OF bench_vhd IS

	-- Component Declaration for the Unit Under Test (UUT)
	COMPONENT hibritbolmebutun
	PORT(
		x : IN std_logic_vector(17 downto 0);
		d : IN std_logic_vector(17 downto 0);
		clk : IN std_logic;
		reset : IN std_logic;
		y : OUT std_logic_vector(17 downto 0)
		);
	END COMPONENT;

	--Inputs
	SIGNAL clk :  std_logic := '0';
	SIGNAL reset :  std_logic := '0';
	SIGNAL x :  std_logic_vector(17 downto 0) := (others=>'0');
	SIGNAL d :  std_logic_vector(17 downto 0) := (others=>'0');
 FILE  dataFile : text is in "data.txt";
 file outputfile: TEXT open WRITE_MODE is "output.txt";

	--Outputs
	SIGNAL y :  std_logic_vector(17 downto 0);

BEGIN


	uut: hibritbolmebutun PORT MAP(
		x => x,
		d => d,
		y => y,
		clk => clk,
		reset => reset
	);

	tb : PROCESS
		variable dataFileLine : line;
	variable data:bit_vector(17 downto 0);
	variable i,j:integer;
	variable outputline: LINE;
	BEGIN
```

```
                    -- Wait 100 ns for global reset to finish
                    wait for 100 ns;
                    wait for 50 ns;
                    reset<='1';
                    wait for 50 ns;
                    reset<='0';
for I in  980 downto 0 loop
readline (dataFile, dataFileLine);
read (dataFileLine, data);
x<=to_stdlogicvector(data);
readline (dataFile, dataFileLine);
read (dataFileLine, data);
d<=to_stdlogicvector(data);

for j in 1 to 7 loop
clk<='0';
wait for 50 ns;
clk<='1';
wait for 50 ns;
end loop;


write(outputline,to_bitvector(y));
writeline(outputfile,outputline);
end loop;


        END PROCESS;

END;
```

EK-6.2 Benzetim giriş dosyasını oluşturan Matlab kodu

```
clear all
a=zeros(5010,1);
fid=fopen('data.txt','w');

   for g=1:5010
   if randn(1,1)>0
      a(g)=1;
   else
      a(g)=-1;
   end
end
h1=[0.5*(1+cos(2*pi/3.5*(-1:1)))];
ah1=conv(a,h1);

v=sqrt(0.001)*randn(5012,1);
```

```
ah1=ah1+v;

for i=7:1000
temp1=dec2bin(typecast(int32(ah1(i)*32),'uint32'),32);
temp2=dec2bin(typecast(int32(a(i-6)*32),'uint32'),32);
fprintf(fid, '%s\n',temp1(15:32) );
fprintf(fid, '%s\n',temp2(15:32) );

end
fclose(fid);
```

EK-6.3 Benzetim çıkış dosyasını okuyan Matlab kodu

```
fid=fopen('output.txt','r');
for i=1:700
aa=fgetl(fid);
for j=1:14
ff(j)=num2str(bin2dec(aa(1)));
end
for k=1:18
ff(k+14)=num2str(bin2dec (aa(k)));
end
simoutput(i)=typecast(uint32(bin2dec(ff)),'int32');
end;
fclose(fid);
hata=(double(simoutput(1:700))/32)-a(2:701)';
kareselhata=zeros(1,700);

for i=1:700
kareselhata(i)=hata(i)^2;
end
plot(kareselhata)
```

ÖZGEÇMİŞ

Kişisel Bilgiler

Soyadı, adı : TİRYAKİ, Sedat

Doğum yılı: 1981

Medeni hali : Bekar

e-mail : chaosgate1889@hotmail.com.

Eğitim ve Akademik Durumu :

Lise :1996-1999 Gazi Osman Paşa Lisesi, Tokat

Lisans :2000-2004 Uludağ Üniversitesi Elektronik Mühendisliği Bölümü, Bursa

TEŞEKKÜR

Tez çalışmamda bana yardımcı olan danışman hocam Yrd.Doç.Dr. Osman Hilmi Koçal'a , Arş. Gör. Metin Hatun' a ve metnin imla denetiminde bana yardımcı olan dostum Çiğdem Tunç'a teşekkürü bir borç bilirim.