T.C.
FIRAT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ALAN PROGRAMLANABİLİR KAPI DİZİSİ İLE SİGMA-DELTA MODÜLATÖRLERİN GERÇEKLENMESİ

Ömer Faruk ALÇİN

Yüksek Lisans Tezi
Elektronik ve Bilgisayar Eğitimi Anabilim Dalı
Danışmanı: Yrd. Doç. Dr. Mehmet GEDİKPINAR

MAYIS–2011

T.C.
FIRAT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ALAN PROGRAMLANABİLİR KAPI DİZİSİ İLE SİGMA-DELTA MODÜLATÖRLERİN GERÇEKLENMESİ

YÜKSEK LİSANS TEZİ
Ömer Faruk ALÇİN
07231104

Tezin Enstitüye Verildiği Tarih: 06/05/2011
Tezin Savunulduğu Tarih:05/05/2011

Tez Danışmanı:	**Yrd. Doç. Dr. Mehmet GEDİKPINAR**
Diğer Jüri Üyeleri:	**Doç. Dr. Abdulkadir ŞENGÜR**
	Yrd. Doç. Dr. Çetin GENCER

MAYIS–2011

ÖNSÖZ

Bu tez çalışması FPGA ile Sigma-Delta modülatörlerin gerçeklenmesi üzerine hazırlanmıştır. Günlük hayatta kullanım yerleri ses, konuşma sinyallerinin dijital ortama aktarılması ve işlenmesi amaçlanmaktadır.

Çalışmam esnasında her türlü hoşgörü, destek ve bilgisiyle bana katkıda bulunan değerli danışmanım Yrd. Doç Dr. Mehmet GEDİKPINAR'a,

Ayrıca benden yardımlarını esirgemeyen Yrd. Doç. Dr. Davut HANBAY, Arş. Gör. Deniz KORKMAZ, Aykut DİKER, Ümit BUDAK ve her daim bana destek veren ve yanımda olan çok kıymetli aileme, arkadaşlarıma ve özellikle anneme teşekkürü bir borç bilirim.

Bu tez, Fırat Üniversitesi Bilimsel Araştırma Projeleri Birimi tarafından FÜBAP–2087 nolu proje olarak desteklenmiştir.

Ömer Faruk ALÇİN

ELAZIĞ- 2011

İÇİNDEKİLER

ÖZET

Fiziksel sistemlere veya olaylara ait bilgi taşıyan sinyaller çoğunlukla sürekli zamanlı (analog) formdadır. Yani zamanın her anında tanımlıdır ve sonsuz uzunlukta herhangi bir değere sahiptir. Bu sistemlerin çözümlenmesinde sürekli zamanlı yöntemler kullanılır. Fakat bazı sistemlerin sürekli zamanlı yöntemlerle çözümü çok zordur, fazla zaman alır ya da imkânsızdır. Günümüzde gelişen sayısal yöntemler ve elektronik devre teknolojileri sayesinde karşılaşılan bu sorunlara çözümler üretilebilmektedir. Yüksek hızlı sistemler tasarlanarak zaman tasarrufu sağlanmakta ve gelişmiş olan sayısal yöntemler zor ya da imkansız olan problemlere uygulanabilmektedir. Sayısal yöntemlerde sinyaller ayrık zamanlıdır. Sürekli zamanlı sinyaller ayrık zamana dönüştürülerek sayısal yöntemlerin uygulanabilmesine imkan tanımaktadır. Bu işlem yapılırken en önemli adım sürekli zamanlı sinyali mümkün olduğunca doğru bir değerde sayısala dönüştürmektir. Bu amaçla kullanılan devrelere Analog/Sayısal çevirici (ASÇ) denir. Başlangıçtan günümüze çok farklı ASÇ yapıları geliştirilmiştir. ASÇ'ler hız ve çözünürlük özellikleriyle birbirinden ayrılırlar. Uygulamaya bağlı olarak uygun yapı seçilmektedir.

Günümüzde en popüler ASÇ yapılarından biri Sigma-Delta modülatörlerdir(SDM). SDM'ler çeşitli sinyal işleme uygulamalarında yaygın olarak kullanılmaktadır. Özellikle yüksek çözünürlükte ve doğrulukta dönüşüm gerektiren, düşük bant genişlikli sinyallerin işlenmesinde yaygın olarak kullanılan ASÇ'lerdir. SDM'lerdeki geri besleme yapısından dolayı kullanılan elemanların toleransları yüksek olabilmektedir. Fazla kullanılmasına rağmen SDM yapılarının davranışları tam olarak tanımlanmamıştır. Bundan dolayı SDM üzerinde yapılan çalışmalar giderek artmaktadır.

SDM gerçeklemelerinde sürekli zamanlı devre elemanları ya da sayısal yongalar kullanılmaktadır. Gelişen teknoloji ile sayısal yongalar hızlı, kapasiteli ve esnek yapı kazanmışlardır. Alan programlanabilir kapı dizileri (Field programmable gate arrays-FPGA), bu sayısal yongalar arasında gelmektedir.

Bu tez çalışmasında, birinci derece SDM FPGA üzerinde gerçeklenmiştir. Öncelikle birinci derece SDM yapısı *MATLAB/Simulink* ortamında farklı giriş sinyalleri uygulanarak benzetim çalışması hazırlanmıştır. Birinci derece ayrık zamanlı SDM "*Xilinx System Generator for DSP*" kullanılarak FPGA üzerinden yapılmıştır. Gerçeklemede Xilinx Virtex5 (XC5SX50T) FPGA'sı kullanılmıştır.

Anahtar Kelimeler: Sigma-Delta Modülatör, FPGA, VHDL, System Generator.

ABSTRACT

IMPLEMENTATION OF SIGMA-DELTA MODULATOR VIA FPGA

Data carrying signals of the physical systems or the events is mainly in the continuous-time form. So, these signals are defined in every point of the time and have infinite length any value. The continuous- time methods are used to analysis of these systems. But, analyzed of the some systems are very difficult with the continuous - time methods, take more time or impossible. Nowadays, solutions can be produced to these problems with developing digital methods and electronic circuit technology. Time saving is provided by designed high-speed systems and advanced digital methods are applied to the difficult or impossible of the problems. Signals are discrete-time in digital domain. Continuous-time signals are transformed into discrete-time and then digital methods are applied. During this process the most important step is to convert continuous-time digital signal value as accurately as possible. These circuits for this purpose are called analog to digital converter (ADC). Recently, different structures of ADC have been developed. ADC's are separated from each other with speed and resolution properties. The appropriate structure is chosen depending on the application.

Nowadays, The Sigma-Delta ADC is the most popular structure of the ADC. The Sigma-Delta Modulator (SDM) is widely used in various signal processing applications. ADC is especially used in signal processing with low band-width which is required the conversion of the high resolution and accuracy. Due to structure of the feedback, the tolerances of components, which use in SDM can be high. Although SDM is commonly used, structure of its behaviors is not fully defined. Thus, studies on the SDM have been gradually increased.

Continuous-time (analog) circuit elements or digital chips are used in implementation of the SDM. Digital chips are gained the fast, capable and flexible structures with emerging technology. FPGA have these features in a chip structure.

In this thesis, First-order SDM is implemented on the FPGA. Firstly, first-order SDM has been simulated using various input signals in *MATLAB/Simulink* Environment. First-order Discrete-time SDM has been implemented on FPGA using the "*Xilinx System Generator for DSP*". In implementation, Xilinx Virtex5 (XC5SX50T) FPGA has been used.

Keywords*:* Sigma-Delta Modulator, FPGA, VHDL, System Generator.

ŞEKİLLER LİSTESİ

Sayfa No

KISALTMALAR LİSTESİ

CLB : Düzenlenebilir lojik blok

CPLD : Karmaşık programlanabilir lojik aygıt

EEPROM : Elekriksel EPROM

EPROM : Silinebilir PROM

FB : Fonksiyon bloğu

FF : Flip-Flop

FPGA : Alan programlanabilir kapı dizisi

GGY : Güç Görünge Yoğunluğu

HDL : Donanım tanımlama dili

IOB : Giriş/Çıkış blokları

LUT : Doğruluk tablosu

MPGA : Maske programlanabilir kapı dizileri

Mux : Çoklayıcı

OSR : Aşırı örnekleme oranı

PAL : Programlanabilir dizi lojik

PLA : Programlanabilir lojik dizi

PLD : Programlanabilir Lojik Aygıtlar

PROM : Programlanabilir sadece okunabilir hafızalar

RAM : Rastgele erişebilir hafıza

SDM : Sigma-Delta Modülatör

SNR : Sinyal-gürültü oranı

SPLD : Basit programlanabilir lojik aygıt

SRAM : Statik RAM

TSA : Tam skala aralığı

VHDL : Yüksek seviyeli donanım tanımlama dili

VLSI : Çok geniş ölçekli entegre devre

WCDMA : Geniş-bant kod-bölmeli çoklu erişim

SEMBOLLER LİSTESİ

$\mathbf{f_{Ns}}$ **:** Nyquist örnekleme frekansı

$\mathbf{f_B}$ **:** Temel bant frekansı

$\mathbf{y_k}$ **:** Nicemleyici çıkışı

i **:** Nicemleyici girişi

$\mathbf{g_q}$ **:** Nicemleyici kazancı

e **:** Nicemleme hatası

Δ **:** Ardışık nicemleme seviyeler arasındaki adım aralığı

P **:** Güç

$\mathbf{e_{rms}}$ **:** Nicemleme hatasının efektif değeri

$\mathbf{s_{rms}}$ **:** Sinyalin efektif değeri

dB **:** Desibel

$\mathbf{S_{TF}}$ **:** Sinyal transfer fonksiyonu

$\mathbf{N_{TF}}$ **:** Gürültü transfer fonksiyonu

H **:** Süzgeç transfer fonksiyonu

Y,y **:** SDM çıkışı

1. GİRİŞ

Gerçek dünyada fiziksel olaylar sürekli zamanlı sinyaller ile ifade edilir. Sürekli zamanlı sinyaller zamanın herhangi bir anında farklı büyüklükte değerler alabilirler. Sayısal işlemciler kullanarak bu sinyalleri işlerken bazı sorunlarla karşılaşılmaktadır. Sürekli zamanlı sinyaller sonsuz kelime uzunluğunda değerler alabilirler bu yüzden hatasız veya kayıpsız iletim yada verilerin depolanması bazen imkansızlaşır. Ayrık zamanlı sistemler, sürekli zamanlı sistemler ile karşılaştırıldığında işlem hızı ve iletim verimliliği gibi konularda büyük üstünlükler sağlarlar. Sürekli zamanlı sinyallerin ayrık zamanlı işlenmesi için sayısal sistemler tarafından algılanabilecek bir forma dönüştürülmesi gerekir [1-5].

Sinyalleri sürekli zamandan ayrık zamana dönüştürmek için kullanılan dönüştürücüler, kullanılan bant genişliğine ve dönüştürme hızına göre iki sınıfa ayrılır. Bu sınıflar, Nyquist oranı (geleneksel) çeviriciler ve aşırı örneklemeli (OSR) çeviricilerdir [2]. Nyquist teoremine göre; örneklenmiş sinyalden orijinal sinyalin bozulmasız bir şekilde elde edilebilmesi için sinyal en yüksek frekanslı bileşeninden en az iki kat fazla frekansta örneklenmelidir. Bu örnekleme frekansına Nyquist frekansı denir [3]. Aşırı örneklemeli çeviriciler ise örnekleme frekansı Nyquist frekansından daha yüksek olan çeviricilerdir. Aşırı örneklemeli çeviriciler sayısal domende (ayrık zaman) işlem yaparlar. Böylece; yüksek doğruluklu, karmaşık sürekli zamanlı devre elemanları yerine sayısal devre elemanları kullanırlar. Aşırı örneklemeli çeviriciler örnekleme oranı sinyalin bant genişliğinden daha büyük olduğundan çevirici girişinde örnekle ve tut devresine ihtiyaç duymazlar. Aşırı örneklemeli çeviricilerde sayısal süzme ve onluk seyreltme (decimation) işlemi tek bir süzgeç ile gerçekleştirilir. Yüksek çözünürlük ve doğruluk için Nyquist çeviricilere göre daha toleranslı elemanlar kullanılabilinir [6].

SDM'ler aşırı örneklemeli Analog/Sayısal çeviricilerden (ASÇ) biridir. SDM'ler nicemleme gürültüsünü temel bandın dışına iterek gürültü şekillendirme sağladığından gürültü şekillendirici modülatörler de denir. Çok geniş ölçekli entegre devre (Very Large Scala Integration, VLSI) gerçeklemesi için uygun olduğundan günümüzde giderek artan ilgiye sahip olmuştur [7-9].

[8]'te geniş-bant kod-bölmeli çoklu erişim (wide-band code-division multiple access - WCDMA) gibi geniş bant radyo frekans sistemlerinde de kullanıldığı bildirilmiştir. Delta

modülatörler ve SDM'ler hakkında temel fikir "Cutler" tarafından ileri sürülmüştür [10]. Bu fikir 1960'larda kabul görmüştür. Bir Cutler sisteminin performansının geliştirmesi, en uygun hale getirilmesi ve ilerletilmesi 1962'de "Spang" ve "Schultheiss" tarafından gerçekleştirilmiştir [10]. Temel SDM'ler, ilk tanımlanmalarından itibaren değişime uğramıştır. İlk önemli değişim 1977 yılında Ritchie tarafından önerilmiştir. 1985'de Candy, Ritchie'nin değiştirdiği modelden kaynaklanan dengesizliği ortadan kaldıran bir çalışma yayımlamıştır [10]. Lee and Sodini 1987 yılında Candy'nin geliştirdiği sistem için bazı teknikler önermiştir ve Carley 1989 yılında bu tekniklerden doğan problemlerin üstesinden gelen bir metot sunmuştur. 1990 yılında Leslie ve Singh bu problemleri ortadan kaldırmak için yeni bir mimari önermişlerdir [10].

Günümüzde sayısal sistemlerinin kullanıldığı alanlardaki hızlı gelişmeler, üretim tamamlandıktan sonra da esnek, genel amaçlı olacak şekilde tasarlanan işlemcileri ortaya çıkarmıştır. VLSI teknolojisi, yüksek yoğunluklu sayısal devrelerin yüksek hızlı ve kolay gerçekleştirilmesini sağlamıştır [11]. Sayısal devrelerin tasarımında işlevleri değiştirilebilen programlanabilir aygıtların kullanımı sayısal sistemlere oldukça üstünlük sağlamıştır. Alan programlanabilir kapı dizisi (Field Programmable Gate Arrays, FPGA) geleneksel programlanabilir aygıtlardan daha üstün özelliklere sahiptir [12]. Bu özelliklerden bazıları yeniden programlanabilir yapısı ve FPGA teknolojisindeki ilerleme, tüm sistem ve alt sistemleri içeren tek aygıt haline getirilebilmesidir [5, 13].

FPGA'nın yapılandırılması için yüksek seviyeli donanım tanımlama dillerinden (Hardware Description Language, HDL), VHDL'nin kullanımı yaygınlaşmaktadır. Bu diller hem yüksek soyutlama ile tasarım sunma hem de geleneksel metotlarla karşılaştırıldığında tasarım süresinde önemli ölçüde azaltma sağlama yeteneğine sahiptir [14]. FPGA üreticilerinden olan Xilinx firmasının Sayısal Sinyal İşlemciler (DSP) için geliştirdiği *MATLAB*/*Simulink* araç kutusu olan *System Generator for DSP* genellikle tasarımların tamamı ya da bir kısmını gerçeklemek için kullanılır [15-18]. *System Generator for DSP* araç kutusu, görsel modüllerin kullanılmasını imkan sağlayarak, HDL'nin detaylarının bilinmesi gerekmeden donanımsal tasarımın kolay bir şekilde gerçeklenmesini sağlar. *System Generator for DSP* hızlı prototipleme aracıdır. *System Generator for DSP* kullanılarak tasarlanan donanım *System Generator for DSP* modu olan *Hardware Co-Simulation* kullanılarak test edilir [15-18].

Bu tez çalışmasında, ikinci bölümde SDM'ler hakkında bilgi verilmişitr. Üçüncü bölümde FPGA anlatılmış, dördüncü bölümde VHDL ve *System Generator for DSP*

incelenmiştir. Beşinci bölümde ise SDM'nin FPGA ile gerçeklenmesi ve tasarlanan SDM'nin test edilmesi anlatılmış ve altıncı bölümde elde edilen sonuçları verilmiştir.

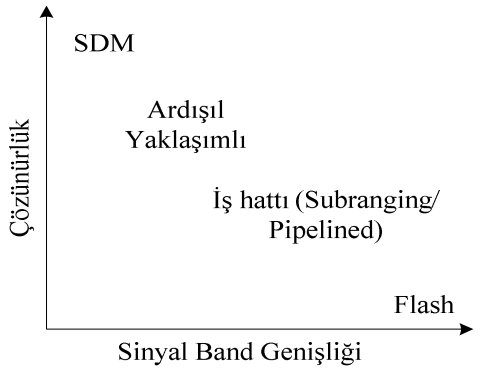
2. SİGMA-DELTA MODÜLATÖRLER

Literatür incelendiğinde, sinyalleri sürekli zamandan ayrık zamana çevirmek için kullanılan birçok yönteme rastlanabilinir. Bu yöntemler örnekleme oranına göre iki ana gruba ayrılır. Birincisi; Nyquist oranı çeviriciler ikincisi ise aşırı örneklemeli çeviricilerdir [19, 20]. Nyquist örnekleme frekansı Denklem 2.1'de verilmiştir.

$$f_{Ns} = 2f_{\max} \tag{2.1}$$

Denklem 2.1'de f_{Ns} nyquist örnekleme frekansını, f_{max} ise sinyalin en yüksek frekanslı bileşenini (temel bant) temsil eder. Nyquist oranı çeviriciler klasik çeviriciler olarak adlandırılır. Nyquist çeviricilere, ardışıl yaklaşımlı çeviriciler, iş hattı (pipeline), flash çeviriciler örnek olarak verilebilinir. OSR çeviricilerde örnekleme frekansı; klasik çeviricilerden çok daha büyüktür [15-17]. SDM'ler aşırı örneklemeli çeviricilerdir [7, 8, 19].

SDM devrelerinde kullanılan elektronik devre elemanları geleneksel çeviricilere oranla daha toleranslı olabilir. SDM'ler sürekli zamanlı devrelerin hızından ve sayısal yöntemlerin doğruluğundan faydalanırlar. Bu çeviriciler, düşük bant genişliğinde en yüksek çözünürlüğü sağlarlar. Şekil 2.1'de farklı çeviriciler için Bant genişliği-Çözünürlük grafiği verilmiştir [19,22].

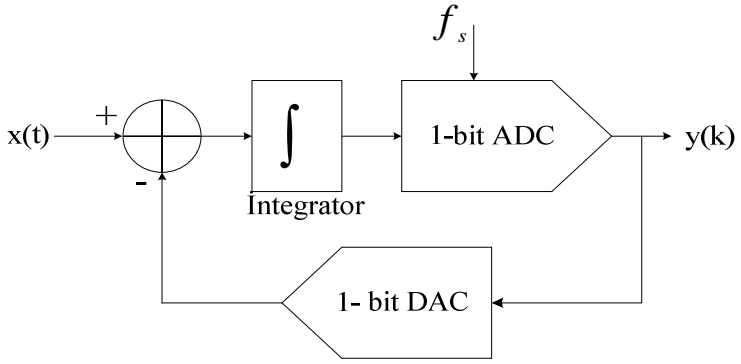


Şekil 2.1. Bant genişliğine karşı çözünürlük değişimi [19].

Şekil 2.1'den de görülebileceği gibi SDM'ler 4Khz bant genişliği ve 14 bit çözünürlüğe gereksinim duyulan konuşma uygulamalarında sıklıkla tercih edilir [19].

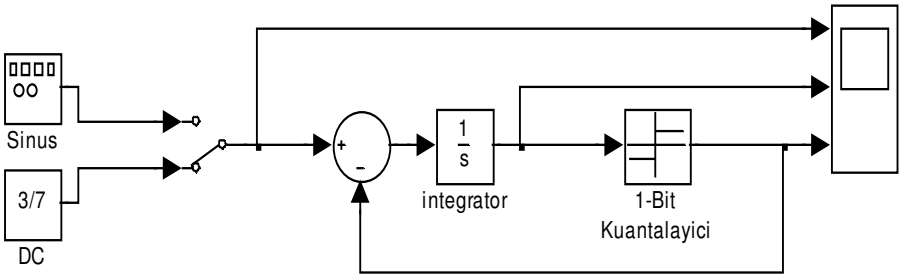
2.1. Sigma-Delta Modülatörlerin Genel Yapısı

Birinci derece SDM blok diyagramı Şekil 2.2.'de verilmiştir. Şekil 2.2'deki 1-bitlik ASÇ yerine, genellikle iki seviyeli niceleyici kullanılır.

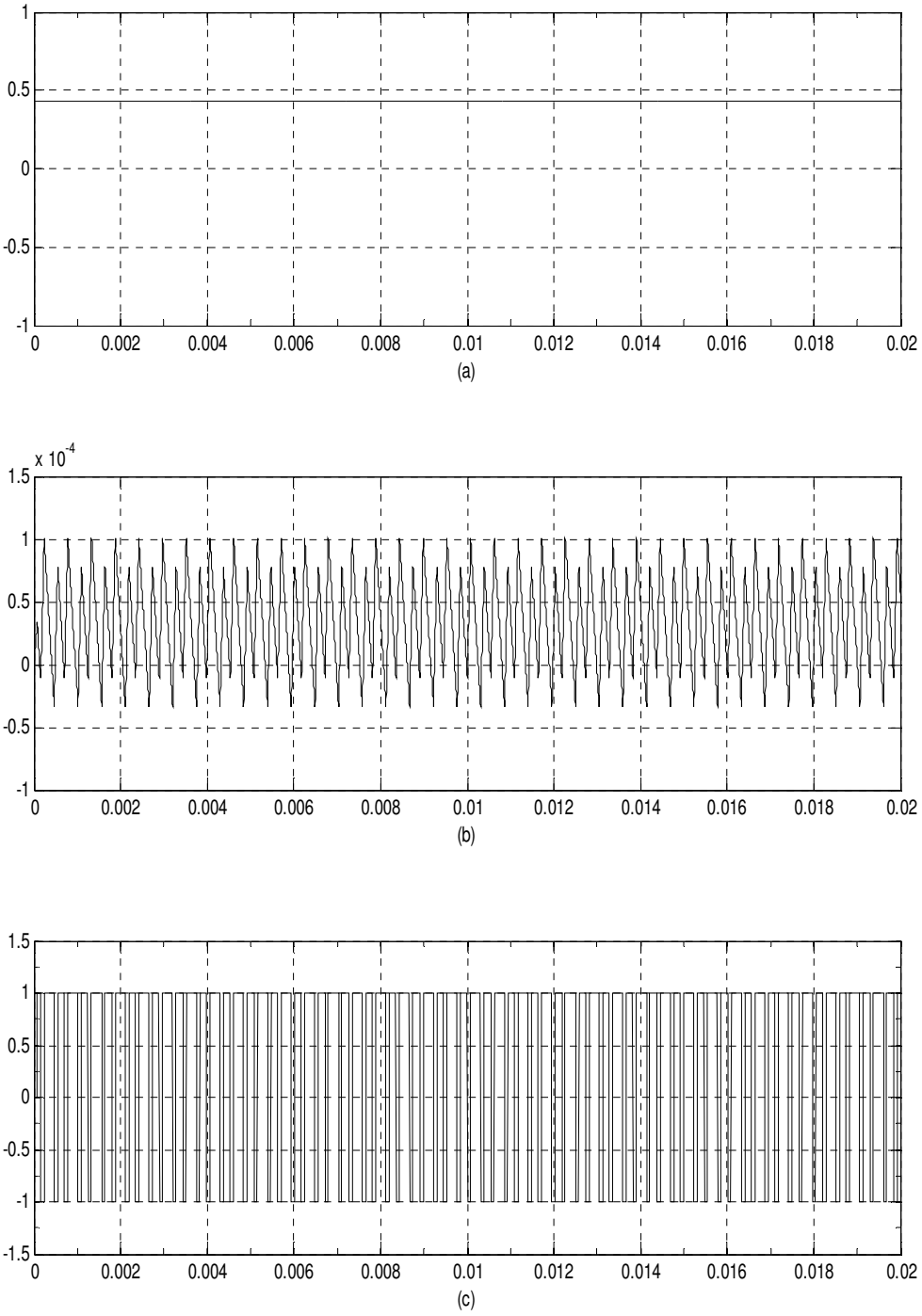


Şekil 2.2. Birinci Dereceden SDM Blok Diyagramı

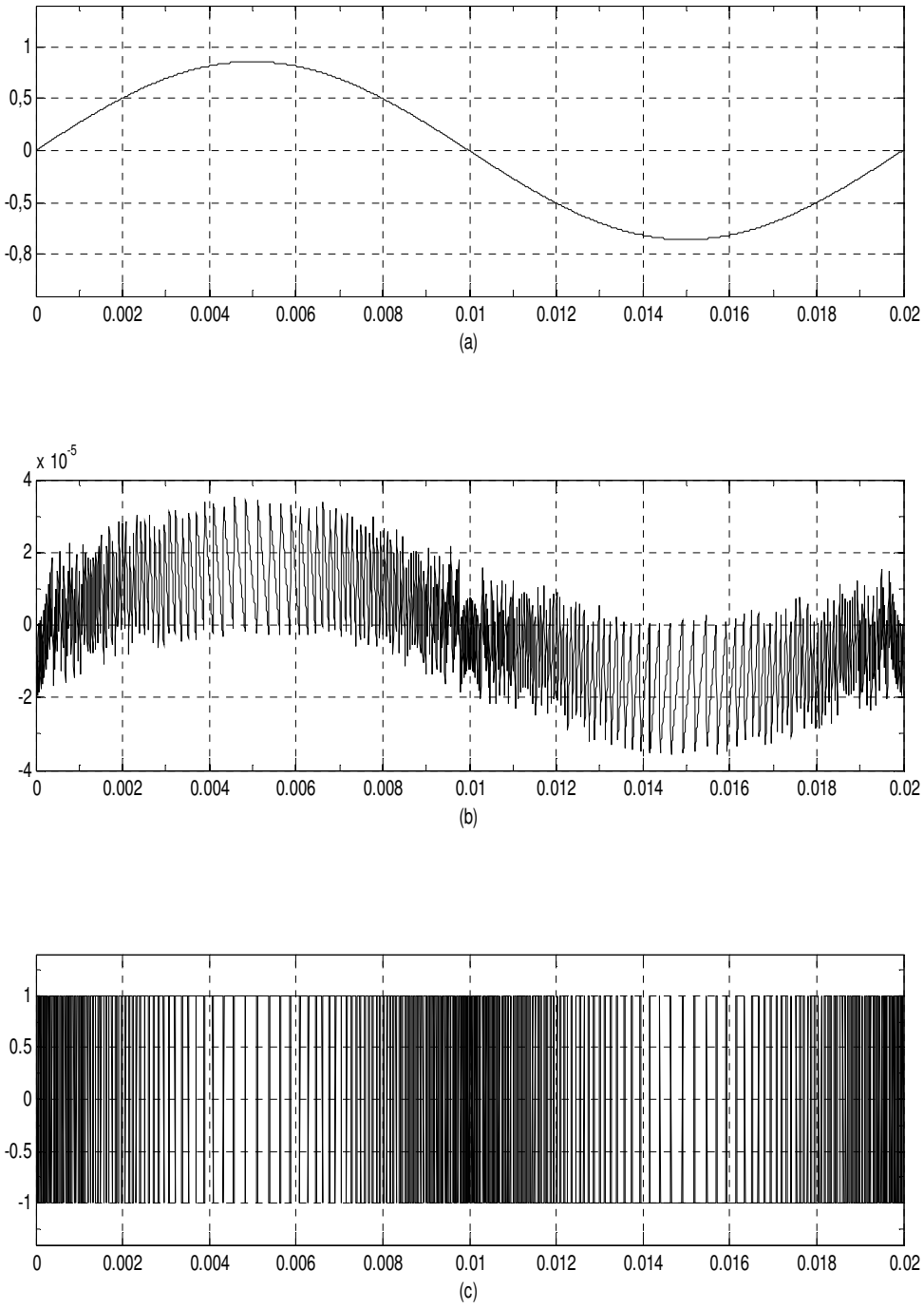
Şekil 2.2'de f_s örnekleme frekansını temsil eder. Birinci derece SDM'de giriş sinyali, nicemleyiciye integral devresi üzerinden aktarılır ve nicemlenmiş çıkış, giriş sinyalinden çıkartılarak geri besleme sağlanır. Geri besleme sayesinde; nicemlenmiş sinyalin ortalama değeri, giriş sinyalinin ortalama değerini izler. Bunların arasındaki fark integral devresi tarafından toplanarak düzeltilir [22]. Çıkışta 1 bitlik darbe dizisi (pulse stream) vardır. Konunun daha iyi anlaşılması için *MATLAB/Simulink* ortamında birinci derece SDM'nin zaman bölgesi modeli (Şekil 2.3) oluşturulmuştur. OSR 128 seçilerek 3/7V sabit ve 0.85V 50Hzlik sinüs giriş sinyalleri için benzetim gerçekleştirilmiştir. Elde edilen benzetim sonuçları Şekil 2.4 ve Şekil 2.5'te sırası ile verilmiştir.



Şekil 2.3. Birinci derece SDM modeli



Şekil 2.4. 3/7V sabit giriş uygulanmış SDM dalga formları, (a) giriş sinyali, (b) integratör çıkışı, (c) SDM'nin çıkışı



Şekil 2.5. 0.85V 50Hz Sinüs giriş uygulanmış SDM dalga formları, (a) giriş sinyali, (b) integratör çıkışı, (c) SDM'nin çıkışı

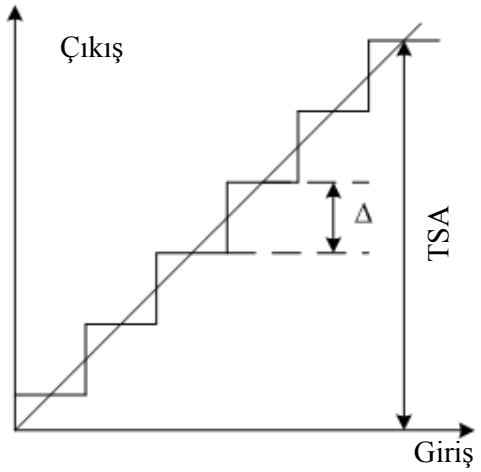
2.2. Nicemleme Gürültüsü ve Aşırı Örnekleme

2.2.1. Nicemleme Gürültüsü

Nicemleme gürültüsü nicemleme hatası olarak da isimlendirilir. Aşırı örnekleme prensiplerinin incelenebilmesi için, nicemleme hatası kavramının açık bir şekilde anlaşılması gerekir. Kısaca; sonsuz sayıda genlik değerine sahip sürekli zamanlı sinyal, önceden tanımlanmış sonlu sayıda sayısal genlik seviyelerine çevrilir. Denklem 2.2'te nicemleyicinin doğrusal olmayan matematiksel ifadesi verilmiştir.

$$y = g_q i + e \tag{2.2}$$

Denklem 2.2'de y nicemleyici çıkışı, i nicemleyici girişi, g_q nicemleyici kazancı ve e ise nicemleme hatasıdır. Şekil 2.6'da düzgün (uniform) nicemleme işlemi gösterilmiştir.



Şekil 2.6 Düzgün nicemleme [19]

Şekil 2.6'da TSA, tam skala aralığını (TSA), Delta (Δ) ardışıl seviyeler arasındaki adım büyüklüğünü gösterir. Şekil 2.6'da görüldüğü gibi sürekli zamanlı sinyal daha önceden TSA'da belirlenen tam sayılara nicemlenir. Nicemleme hatası çeviricinin çözünürlüğüne bağlıdır. Nicemleme hatası sürekli zamanlı giriş ile ayrık zamanlı çıkış sinyali arasındaki fark olarak tanımlanabilir. Nicemleme hatası nicemleme seviyesi sayısı

artırılarak azaltılabilir. Böylece çeviricinin çözünürlüğü artar. Δ'nin matematiksel ifadesi denklem 2.3'de verilmiştir.

$$\Delta = \frac{TSA}{2^{N-1}} \tag{2.3}$$

Denklemde *N* nicemleyici bit sayısını gösterir. Eğer giriş TSA aralığına sınırlanırsa, nicemleme hatası *(e)*, *±Δ/2* aralığında sınırlandırılır. Eğer nicemleme hatasının Δ üzerine düzgün dağıldığı varsayılırsa, gürültü olasılık fonksiyonu;

$$P(y) = \frac{1}{\Delta} \tag{2.4}$$

şeklinde ifade edilir. Denklem 2.4'te *P* gürültü olasılık fonksiyonudur. Nicemleme hatası *(e)*, *±Δ/2* aralığında eşit olasılıkla dağılıyorsa, nicemleme hatasının *(e)*'nin efektif değeri

$$e_{rms}^2 = \frac{1}{\Delta}\int_{-\frac{\Delta}{2}}^{+\frac{\Delta}{2}} e^2 de = \frac{\Delta^2}{12} \tag{2.5}$$

Denklem 2.5'deki gibi bulunur ve ifadenin karekökü alınması ile nicemleme gürültüsünün büyüklüğü Denklem 2.6'da verildiği gibi bulunur.

$$e_{rms} = \frac{\Delta}{\sqrt{12}} \tag{2.6}$$

N-bitlik ideal nicemlenmiş sinüs sinyalinin, sinyal gürültü oranını (SNR) hesaplayabilmek için Denklem 2.6'te hesaplanan nicemleme gürültüsü gerilimine ek olarak Denklem 2.7'da verilen sinüs sinyalin efektif değeri de hesaba katılmalıdır.

$$s_{rms} = \frac{2^N \Delta}{2\sqrt{2}} \tag{2.7}$$

Böylece SNR Denklem 2.8'deki gibi düzenlenir.

$$SNR = \frac{s_{rms}}{e_{rms}} = \frac{\frac{2^N \Delta}{2\sqrt{2}}}{\frac{\Delta}{\sqrt{12}}} = 2^N \sqrt{1.5} \tag{2.8}$$

Denklem 2.8'in logaritması alınarak SNR'nin desibel olarak değeri bulunur.

$$(SNR)_{dB} = 6.02N + 1.76 \tag{2.9}$$

Denklem 2.9'da SNR ile nicemleyici bit sayısı arasında ilişki oluştuğuna dikkat edilmelidir. Verilen bu eşitlikler Nyquist oranı çeviriciler için geçerlidir. Önceden de belirtildiği gibi SDM'ler nicemleme gürültüsünü düşürüp çözünürlüğü artırmak için aşırı örnekleme tekniğini kullanırlar. Bu teknik örnekleme frekansı bilgisini hesaba katarak e_{rms} formülünü değiştirir ve sonuç olarak SNR'yi ilerletir [2, 9, 19, 21].

2.2.2. Aşırı Örnekleme

Nyquist teoremine göre; aşırı örnekleme oranı, örneklenmiş sinyale bilgi eklemez ama geniş görünge için kullanımı gereksizdir. Görünge genişlediğinde nicemleme güç yoğunluğu azalır çünkü örnekleme oranı ne olursa olsun nicemleme gürültüsü gücü sabit kalır. Denklem 2.5'te verilen nicemleme hatası efektif değeri, aşırı örneklenmiş sinyalin bant genişliği temel bant genişliğine sınırlandırılmış olduğu ve örnekleme frekansının işleme katılması ile nicemleme hata gerilimi nicemleyici bit sayısına bağlı olur. Böylece Denklem 2.10 elde edilmiş olur.

$$e_{rms}^2 = \frac{\Delta^2}{12OSR} \tag{2.10}$$

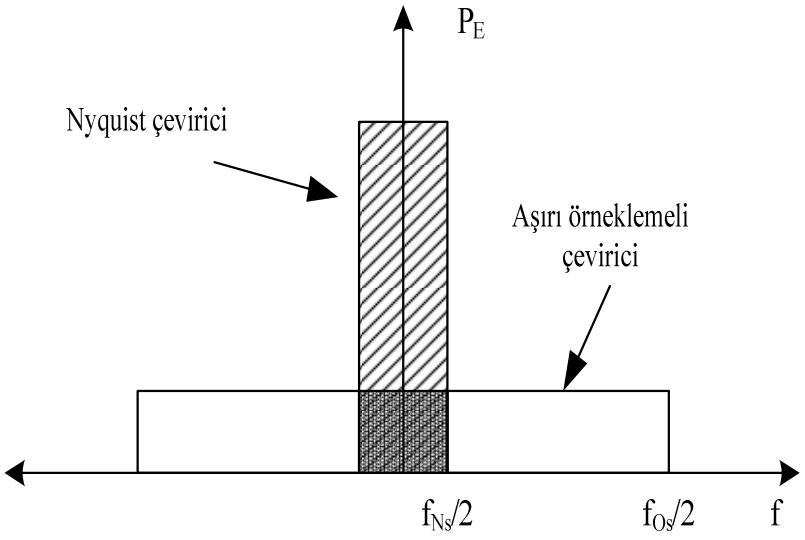
Aşırı örnekleme oranı (OSR) ifadesi Denklem 2.11'de verilmiştir.

$$OSR = \frac{f_s}{2f_{\max}} = \frac{f_s}{2f_{Ns}} \tag{2.11}$$

Aşırı örnekleme ilgilenilen bantta yer alan gürültü miktarını azaltır, zayıf gürültü ise SNR'yi artırır. Örneğin OSR değeri iki seçilirse Denklem 2.11'den f_s ve nicemleme gürültüsünün bant genişliği 4 kat artar ve buna rağmen, nicemleme gürültüsünün genliği ise dörde bölünür. Ardından aşırı örneklenmiş sinyal temel bant süzgeçden geçirilirse, nicemleme gürültüsünün gücü dört kat azaltılır. Denklem 2.9'un aşırı örneklemeli çeviriciler için değiştirilmiş versiyonu olan Denklem 2.12 düşünülürse, nicemleme gürültüsü gücünde ki dört kat azalma SNR oranında 6 dB'lik iyileşme sağlar.

$$(SNR)_{dB} = 6.02N + 1.76 + 3.01(OSR) \tag{2.12}$$

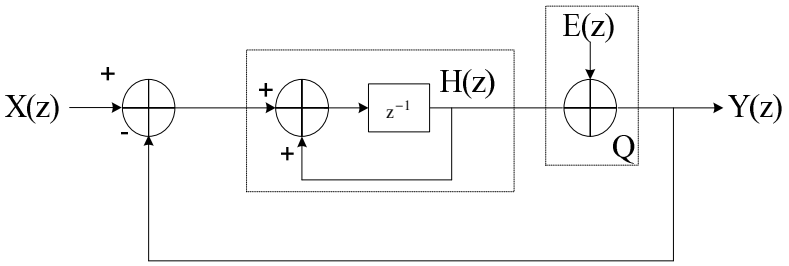
Denklem 2.12'ten görüleceği gibi OSR'deki her 2 kat artış 6 dB'lik iyileşme (bu nicemlenmiş sinyale 1 bit ekleme ile aynı) sağlar. Bu ilke çeviricilerin çok önemli karakteristiği olan doğruluğunu artırır. Aşırı örneklemenin sonucu olan bu iyileşme Şekil 2.7'de gösterilmiştir [2, 9, 19, 21].



Şekil 2.7. Nyquist çevirici ile aşırı örneklemeli çevirici karşılaştırması [19].

2.2.3. Gürültü Şekillendirme

Gürültü şekillendirme; sinyalin temel bandında düşük gürültü gücü bırakmak için gürültünün bir kısmını yüksek frekanslara kaydırarak gerçekleştirilen bir süzme işlemidir. Bu yöntem aşırı örnekleme tekniğinin kullanımını ve bant genişliğini artırma riskini almaksızın SNR'de iyileştirme yeteneği sağlar.



Şekil 2.8. Birinci derece SDM'nin lineerleştirilmiş blok diyagramı [23].

Şekil 2.8'de gösterildiği gibi nicemleme gürültüsü ayrı bir giriş olarak kabul edilir. Devrenin transfer fonksiyonu Denklem 2.13'te verilmiştir

$$Y(z) = S_{TF} X(z) + N_{TF} E(Z) \tag{2.13}$$

Denklem 2.13'de S_{TF} sinyal transfer fonksiyonunu, N_{TF} ise gürültü transfer fonksiyonunu ifade eder. Bunlar sırası ile Denklem 2.14 ve 2.15'de verilmiştir.

$$S_{TF} = \frac{H(z)}{1+H(z)} \tag{2.14}$$

$$N_{TF} = \frac{1}{1+H(z)} \tag{2.15}$$

Şekil 2.8'deki çevrim süzgeci, *H(z)*, giriş sinyaline alçak geçiren süzgeç ve gürültüyü şekillendirmek için nicemleme gürültüsüne yüksek geçiren süzgeç gibi davranır. Böylece nicemleme gürültüsü arzu edilen frekanstan bandından uzaklaştırılır. Gürültü ve sinyal transfer fonksiyonu sırasıyla 0 ve 1 düşünülürse, *H(z), z=1* için sonsuz değere yaklaşmalıdır. İstenen etkiyi oluşturmak için çevrim süzgeci yerine integratör kullanılmalıdır. İntegratörün transfer fonksiyonu Denklem 2.16'da verilmiştir.

$$H(z) = \frac{z^{-1}}{1 - z^{-1}} \tag{2.16}$$

Böylece Denklem 2.14 ve 2.15 sırasıyla Denklem 2.17 ve 2.18 dönüşür.

$$S_{TF} = \frac{H(z)}{1 + H(z)} \tag{2.17}$$

$$N_{TF} = 1 - z^{-1} \tag{2.18}$$

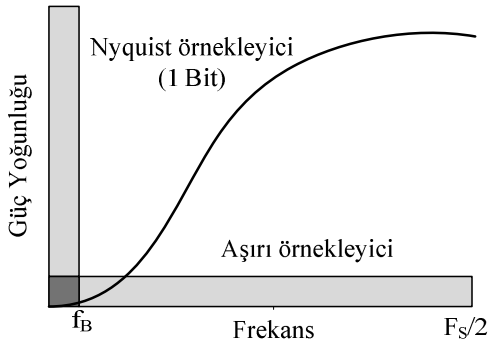
Böylece devrenin transfer fonksiyonu Denklem 2.19'daki düzenlenir.

$$Y(z) = X(z)z^{-1} + E(Z)(1 - z^{-1}) \tag{2.19}$$

Ayrıca Denklem 2.19 aşağıdaki gibi gösterilebilir.

$$y[n] = x[n-1] + e[n] - e[n-1] \tag{2.20}$$

Sonuç olarak SDM giriş sinyaline alçak geçiren süzgeç ve nicemleme gürültüsüne yüksek geçiren süzgeç gibi davranır. çıkışın ortalama değeri giriş sinyaline eşit olur. Şekil 2.9'da verilen grafikten anlaşılacağı gibi gürültü güç yoğunluğu yüksek frekanslara taşınarak arzu edilen temel bant genişliğinde görülen gürültü güç yoğunluğu düşürülmüş olur [2, 9, 19, 21].



Şekil 2.9. Birinci derece SDM'nin görüngesi[19].

3. ALAN PROGRAMLANABİLİR KAPI DİZİSİ (FPGA)

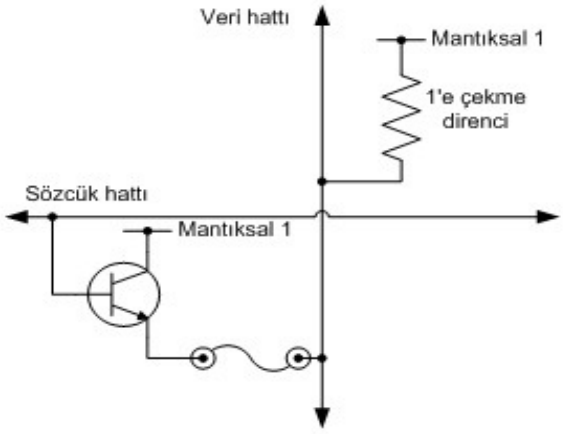
Alan programlanabilir kapı dizisi (Field Programmable Gate Arrays, FPGA) kullanıcı tarafından serbest olarak yeniden programlanabilen lojik-yapı taşları olarak tanımlanabilir. FPGA yaygın olarak kullanılan, geniş uygulama alanlarına sahip programlanabilir yongalardır [24]. Konunun daha iyi anlaşılması için Bölüm 3.1'de programlanabilir lojik aygıtlar anlatıldıktan sonra Bölüm 3.2'de FPGA incelenmiştir.

3.1. Programlanabilir Lojik Aygıtlar

Günümüzde VLSI tasarımı için farklı seçenekler sunulmaktadır. Programlanabilir lojik aygıtlar (PLDs) VLSI tasarım için en çok kullanılan yapılardır. PLD kısaca kullanıcı tarafından programlanabilen tüm devreler için kullanılan genel bir kavramdır. PLD günümüz gelişmiş karmaşık yapısına ulaşmak için uzun bir evrim süreci geçirmiştir. Ürün miktarı ve fiyat dikkate alındığında eğer performans gereksinimleri de karşılanıyorsa PLD'nin kullanılması tasarım sürecini kısaltmaktadır. Klasik tasarım sürecinde tasarım bittikten sonra ürünün ortaya çıkması aylar sürmekte ve tasarımda herhangi bir yanlışın olması durumunda ise harcanan maliyet ve süre tekrar ödenir. Gerçekleştirilmesi istenen lojik devreye bağlı olarak metal tabaka üretici firma tarafından son aşamada oluşturulmaktadır. Ancak bu yöntemde tasarım esnekliğini tamamen kullanıcıya bırakmamaktadır [25, 26]. PLD'ler; programlanabilir sadece okunabilir hafızalar (PROM), basit programlanabilir lojik aygıt (SPLD), karmaşık programlanabilir lojik aygıt (CPLD), maske programlanabilir kapı dizileri (MPGA), alan programlanabilir kapı dizileri (FPGA) olmak üzere beş alt bölümde incelenebilir [5].

3.1.1. Programlanabilir Sadece Okunabilir Hafızalar (PROM)

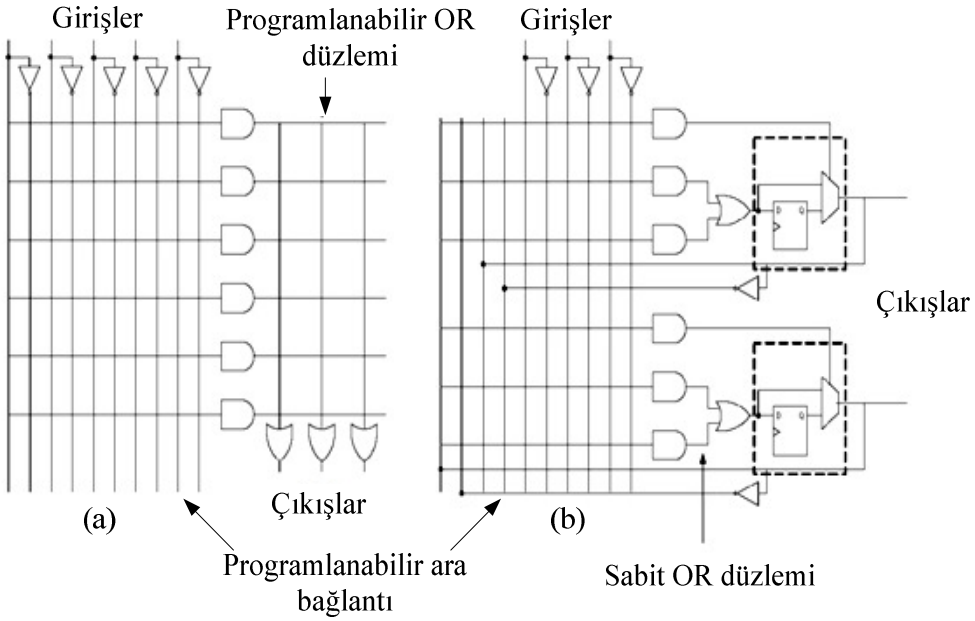
PROM'lar ilk programlanabilir lojik aygıtlarıdır [23]. Şekil 3.1.'de bir PROM hücresinin yapısı gösterilmektedir. Bu aygıtta, belleğin adres girişleri (sözcük hattı), lojik devrenin girişlerine, adreslenmiş gözdeki bilgiler (veri hattı), lojik fonksiyonun çıkışlarına karşılık düşmektedir. Genellikle karmaşık lojik fonksiyonlar için verimsizdirler [5].



Şekil 3.1. PROM hücresi

3.1.2. Basit Programlanabilir Lojik Aygıt (SPLD)

SPLD, basit özelliklere sahip olmasının yanında maliyeti de esas tercih edilme sebeplerinden biridir. Basit işleve sahip devrelerde oldukça sık kullanılmaktadır. SPLD'ler programlanabilir lojik dizi (PLA) ve programlanabilir dizi lojik (PAL) olmak üzere iki kısma ayrılır.

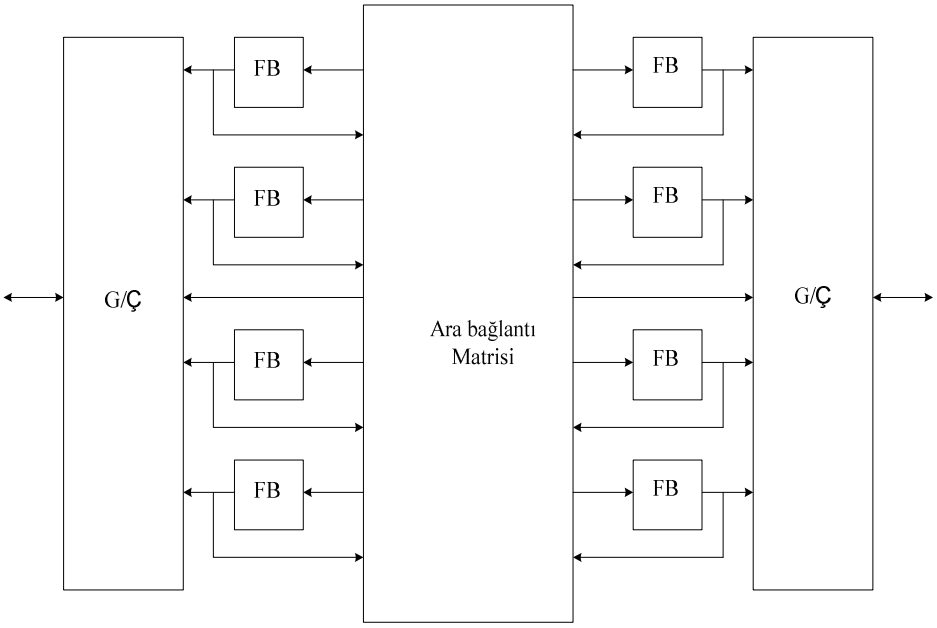


Şekil 3.2. (a) PLA yapısı, (b) PAL Yapısı [29].

PLA programlanabilir bir OR (VEYA) düzlemi ile programlanabilir ve düzlemi, PAL ise sabit bir OR düzlemi ile programlanabilir ve AND düzlemi içerir. PAL küçük ölçekli lojik devrelerde kullanılmaktadır [28].

3.1.3. Karmaşık Programlanabilir Lojik Aygıt (CPLD)

SPLD'lerin sınırlı kapasiteleri daha yüksek kapasiteli programlanabilir devreler olan CPLD'lerin doğmasına neden olmuştur. CPLD'ler, SPLD benzeri birçok bloğun bir araya getirilmesiyle oluşmuş yapılardır [30]. Bir CPLD, on ile birkaç yüz makro hücre içerir. Sekizden on altıya kadar makro hücreler birleşerek daha büyük bir fonksiyon bloğu (FB) oluşturur. Genel olarak bir fonksiyon bloğu içerisinde yer alan makro hücreler birbirine tamamen bağlıdır. Hangi blokların nasıl bağlandığı üreticiye ve ait olduğu aileye göre değişmektedir [29]. Şekil 3.3' de genel CPLD yapısı görülmektedir.



Şekil 3.3. CPLD mimarisi.

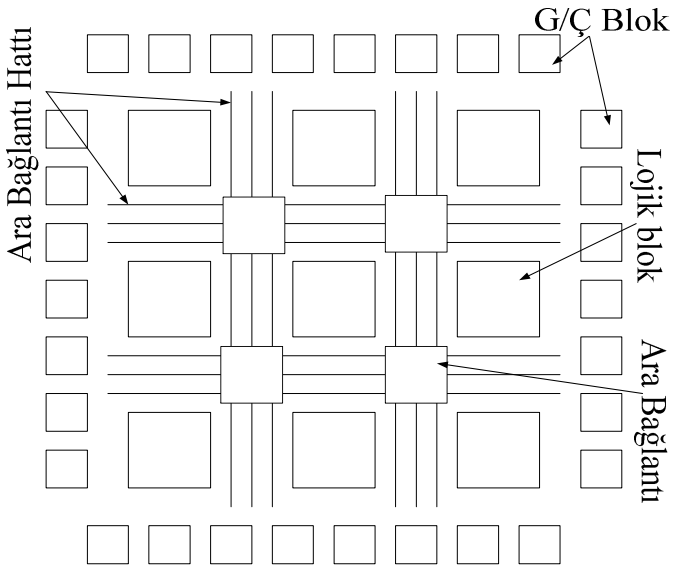
3.1.4. Maske Programlanabilir Kapı Dizisi (MPGA)

Daha büyük lojik devreleri oluşturabilmek için MPGA yongalar üretilmiştir. Genel bir MPGA, istenen lojik devreyi oluşturabilmek için birbirleri ile bağlanmış transistor satırları içerir. Kullanıcı tarafından tanımlanmış bağlantılar, hem satır içerisinde hem de

satır aralarında bulunur. Bu yöntem, lojik kapıları oluşturabilme ve oluşturulmuş lojik kapıları birbirine bağlayabilme imkânı sağlarken, metal tabakalar üretici tarafından tanımlandığı için üretim sayısı arttıkça fiyat ve zamandan kazandırır [28].

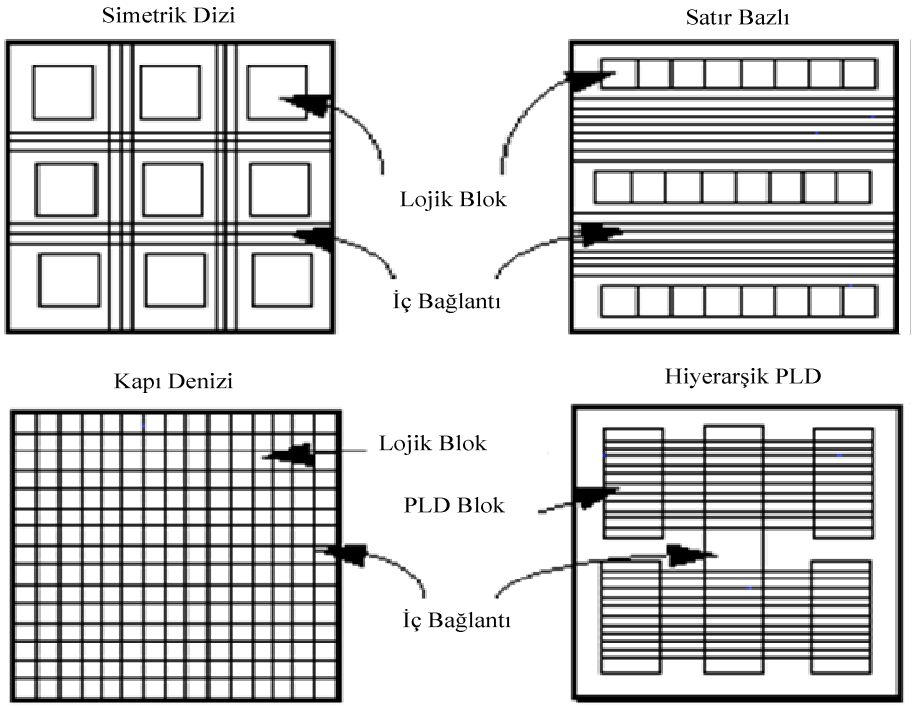
3.2. FPGA

FPGA, Xilinx tarafından 1984'de piyasaya sürülmüştür[27, 31, 32]. FPGA'lar bir sayısal devre ya da sistem olmak için elektriksel olarak programlanabilen yarı-hazır (pre-build) silikon aygıtlardır. Programlanabilme terimi; silikon aygıtın üretildikten sonra programlanabilme yeteneğini gösterir [33]. Bir lojik devrenin FPGA üzerinde; düşük geliştirme maliyetle geliştirilmesi ve doğasında olan alan programlama esnekliği FPGA teknolojisinin göz alıcı biçimde büyümesine ve VLSI aygıtlar arasında popüler olmasına yol açmıştır [34, 35]. Ağ oluşturma, sayısal sinyal işleme gibi pek çok uygulama alanlarında çok yönlülüğü gösterilmiştir [31, 32]. FPGA iç yapısı Şekil 3.4.'de verilmiştir



Şekil 3.4. FPGA'nın içyapısı.

Şekil 3.5'de gösterildiği gibi FPGA'lar bağlantı çeşitleri dikkate alınarak simetrik dizi, sıra tabanlı, hiyerarşik PLD ve kapı denizi olmak üzere dört grupta sınıflandırılırlar [24, 36].



Şekil 3.5. FPGA'nın bağlantı şekillerine göre sınıflandırılması [37].

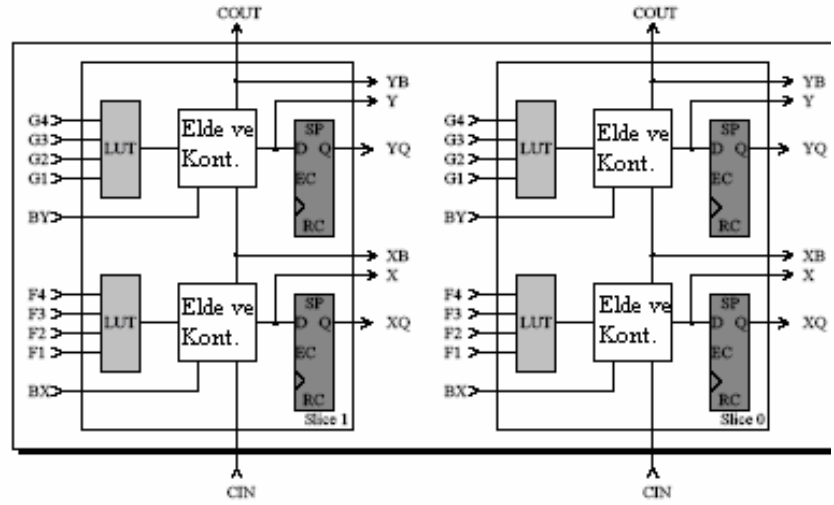
3.2.1. FPGA Mimarisi

FPGA mimarisi Şekil 3.4'de gösterildiği gibi üç bileşenden oluşur [21, 33]. Bunlar;

- Düzenlenebilir lojik blok (Configurable Logic Blocks, CLB)
- Giriş/Çıkış blokları (Input/Output Blocks, IOB)
- Ara bağlantılar (Anahtarlama matriksi)

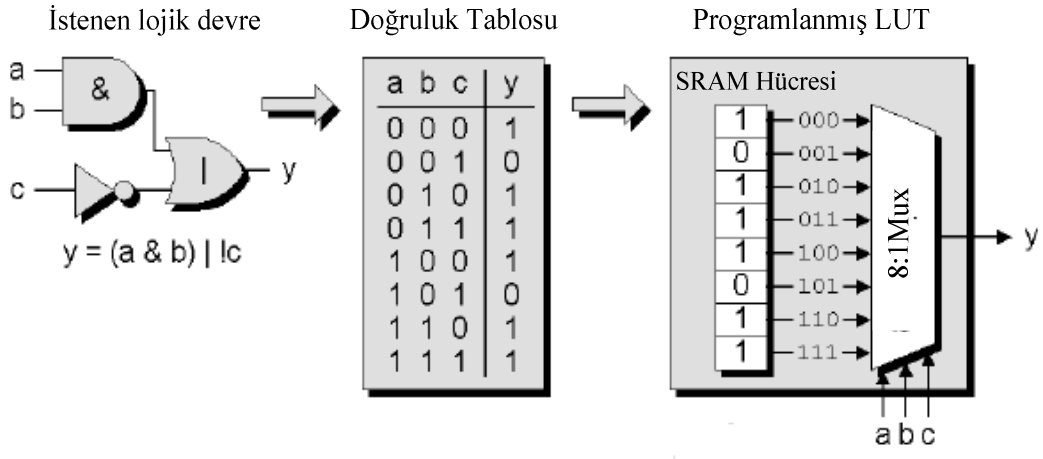
Düzenlenebilir lojik blok (CLB): CLB tasarımcının oluşturmak istediği lojik devre için programlanabilen fonksiyonel aygıtlardır. FPGA'lar çok sayıda bu CLB'lerden oluşmaktadır. Yongadaki her lojik blok farklı bir fonksiyonu gerçekleştirmek için uygun SRAM (farklı bir teknolojide olabilir) programlama hücreleri vasıtasıyla yapılandırılabilir [25, 37]. Şekil 3.6'de bir CLB bloğunun iç yapısı gösterilmiştir. Bir CLB bloğu bir çift Flip-Flop (FF) ve iki adet dört girişli fonksiyon üreticisi (üretici firmaya göre farklılık gösterebilir) içerir. Bu fonksiyon üreticileri dört girişten az olan fonksiyonları üretilebilme esnekliğine sahiptirler. Bu bloklar FPGA yongaya uygulanacak olan lojik devrenin büyük

bir kısmını oluştururlar. CLB mimarisinin sahip olduğu esneklik ve simetri özellikleri uygulamaların kolaylıkla yerleştirilmesine ve yönlendirmesine olanak sağlar [37].



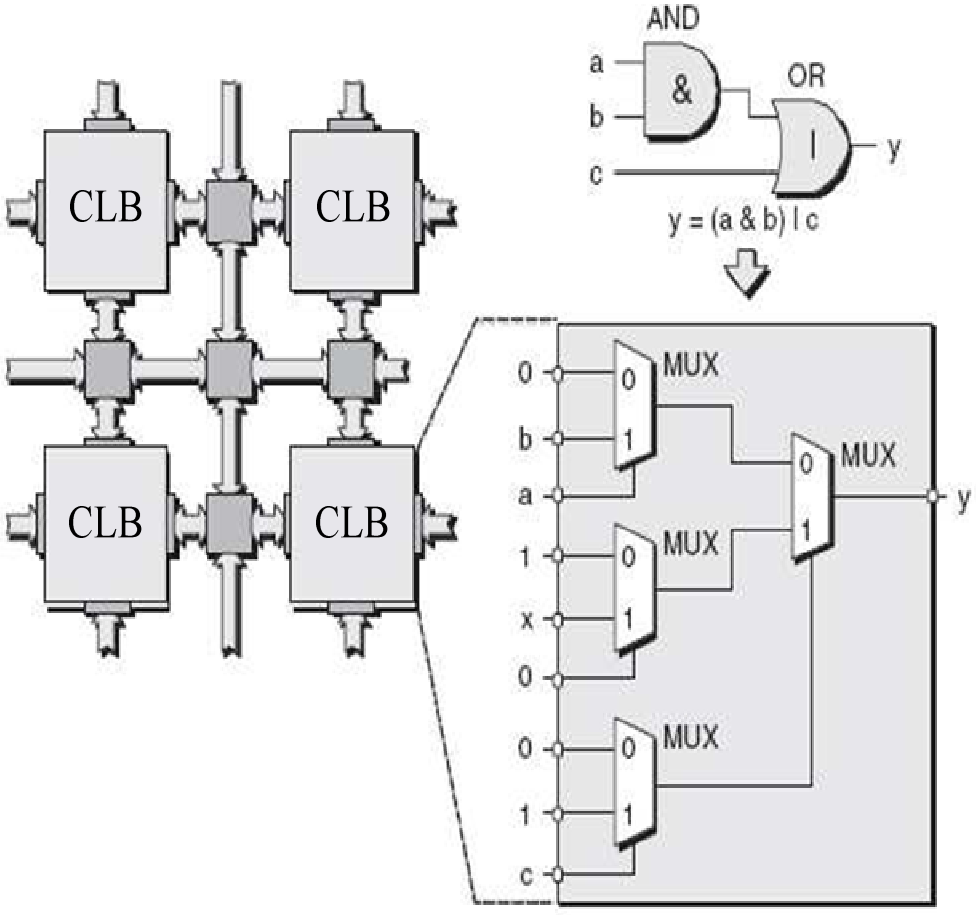
Şekil 3.6. CLB'nin İç yapısı [37].

CLB'ler mimarisi doğruluk tablosu (Look-Up Table, LUT) tabanlı ya da çoklayıcı (Mux) tabanlı yapıdan oluşurlar [25, 37]. Bazılarında ise ardışıl devrelerin de gerçeklenebilmesi amacıyla FF kullanılmıştır [25, 37]. LUT tabanlı yapının temel bloğu m (3-6 arasında sabit sayı) değişkenli her Boole fonksiyonunu gerçekleyebilen devredir. Şekil 3.7'de bir lojik fonksiyonun gerçeklenmesi için LUT' un yapılandırılması gösterilmiştir [25].



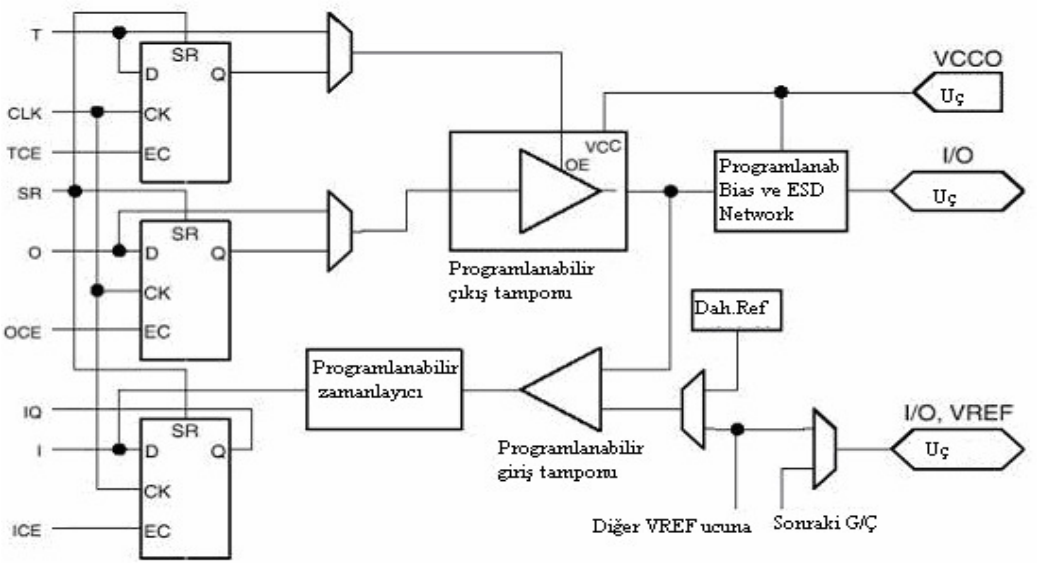
Şekil 3.7. LUT'un yapılandırılması[25].

Çoklayıcı tabanlı yapının temel bloğu çeşitli yapılandırmalardan ve olabildiğince az lojik kapılarıdan (VE ve VEYA gibi) oluşur. Bu yapıdaki FPGA'ların içinde tutucu (latch) ve (FF) bulunmadığından çoklayıcı kullanılarak CLB gerçeklenir. Şekil 3.8'de bir lojik fonksiyonun Mux tabanlı yapı ile gerçeklenmesi gösterilmiştir [38].



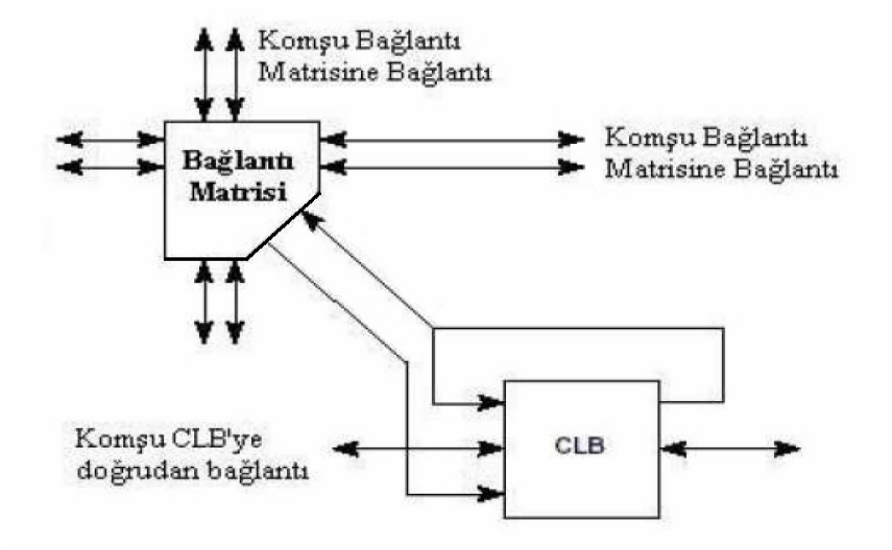
Şekil 3.8. Bir lojik fonksiyonun MUX tabanlı yapı ile gerçeklenmesi [25].

Giriş/Çıkış Birimi (IOB): IOB'ları, kılıf bacaklarıyla tasarım için kullanılan birimler (CLB, Blok RAM) arasında bağlantı kurar. FPGA'lerin giriş-çıkış blokları; giriş, çıkış veya giriş-çıkış olarak 3 farklı şekilde tanımlanır [25]. Şekil 3.9'da Xilinx Firmasının ürettiği FPGA'ya ait giriş-çıkış bloklarının yerleşimi ve bir giriş-çıkış bloğunun ayrıntılı seması gösterilmiştir [37].



Şekil 3.9. Xilinx FPGA IOB yapısı [37].

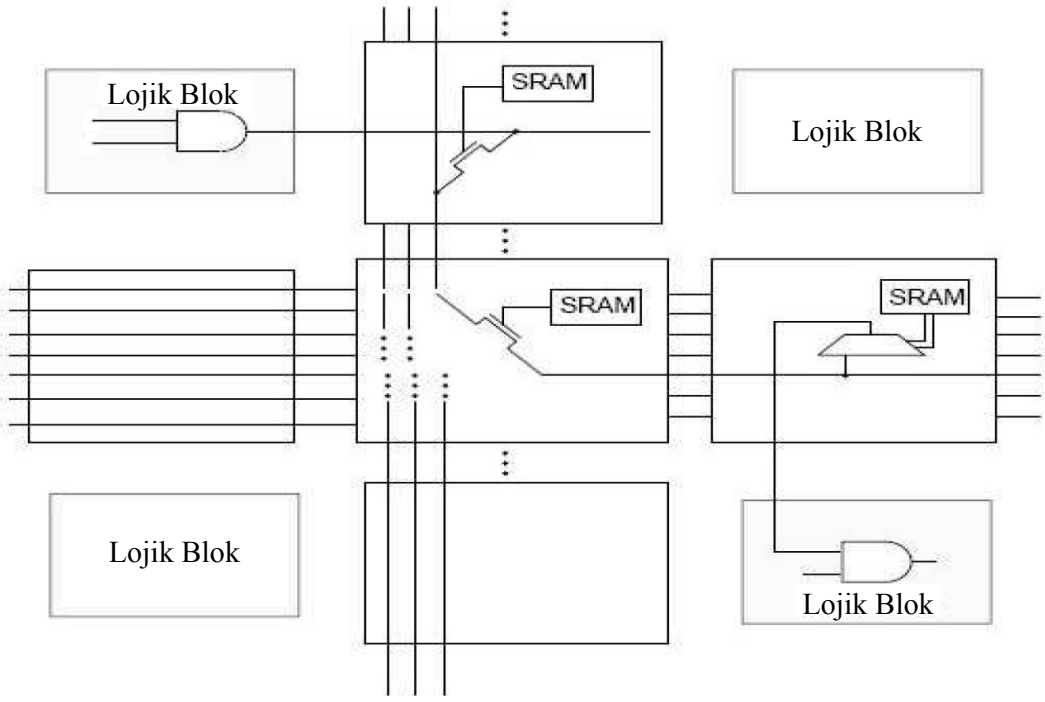
Ara Bağlantılar: IOB'lar ile istenen lojik devre için hazırlanan CLB'ler arasındaki bağlantılar, ara bağlantı aygıtları ile sağlanır. FPGA içerisindeki ara bağlantı aygıtları, CLB' ler arasına satırlar ve sütunlar halinde yerleştirilmiş bağlantı hatları ve bu hatların kesişim noktalarına yerleştirilmiş bağlantı matrislerinden oluşur. Şekil 3.10'da bir FPGA yongası içerisindeki bağlantı elemanları gösterilmiştir [25].



Şekil 3.10. FPGA Ara bağlantı birimi [25].

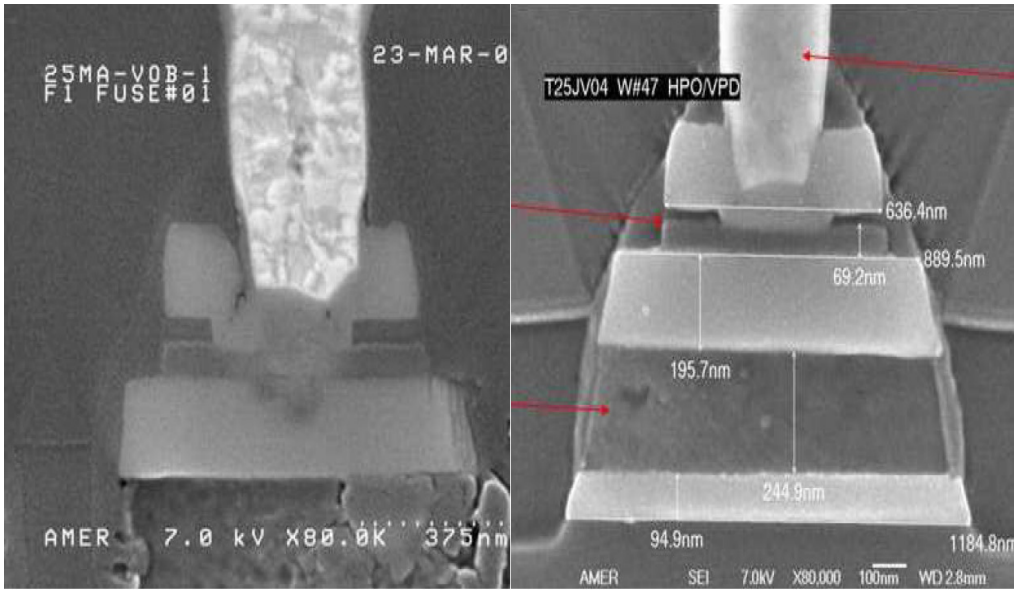
3.2.2. FPGA Programlama Teknolojileri

CLB'ler ve programlanabilir ara bağlantılar sayesinde FPGA'lar programlanabilme yeteneği kazanır. Programlanabilir ara bağlantıların teknolojisi FPGA'nın programlama teknolojisini belirler [39]. Üreticiler tarafından, kendi FPGA mimarilerinin programlanabilirlik özelliğini sağlamak için SRAM, EPROM, EEPROM, sigorta (Fuse) ve anti sigorta(Anti-Fuse) gibi farklı programlama teknolojileri kullanılmaktadır [35]. Günümüzde SRAM ve karşıt sigorta (Anti-Fuse) öne çıkmaktadır. Bu iki teknolojinin birbirlerine göre olumlu ve olumsuz yönleri bulunmaktadır. SRAM programlama teknolojisinde normal işlem süresince program statik hafıza hücresinde tutulur. Yonga üzerinde SRAM hafıza hücreleri için ayrılmış yer yoktur. SRAM hücreleri kontrol ettikleri lojik elemanlar arasında dağıtılarak yerleştirilmiştir. SRAM programlama teknolojisinin öne çıkan olumsuzlukları geniş alan kaplaması ve uçuculuğudur (elektrik kesilince silinmesi). SRAM programlama teknolojisi hızlı tekrar programlanabilirlik ve hatta karmaşık devreler için bile düşük güç tüketimi gibi iki önemli üstünlüğe sahiptir [39]. Şekil 3.11'de SRAM hücrelerinin kullanılması verilmiştir.



Şekil 3.11. SRAM kullanımı [25].

Bir anti-sigorta yüksek empedans konumunda iken (100M-Ohm ile 1G-Ohm arası değerlidir) programlama gerilimi uygulanarak düşük empedans (sigortalanmış) durumlarına programlanabilir. Şekil 3.12'de anti sigorta teknolojisiyle programlama verilmiştir.



Şekil 3.12. Anti sigorta teknolojisiyle programlama [25].

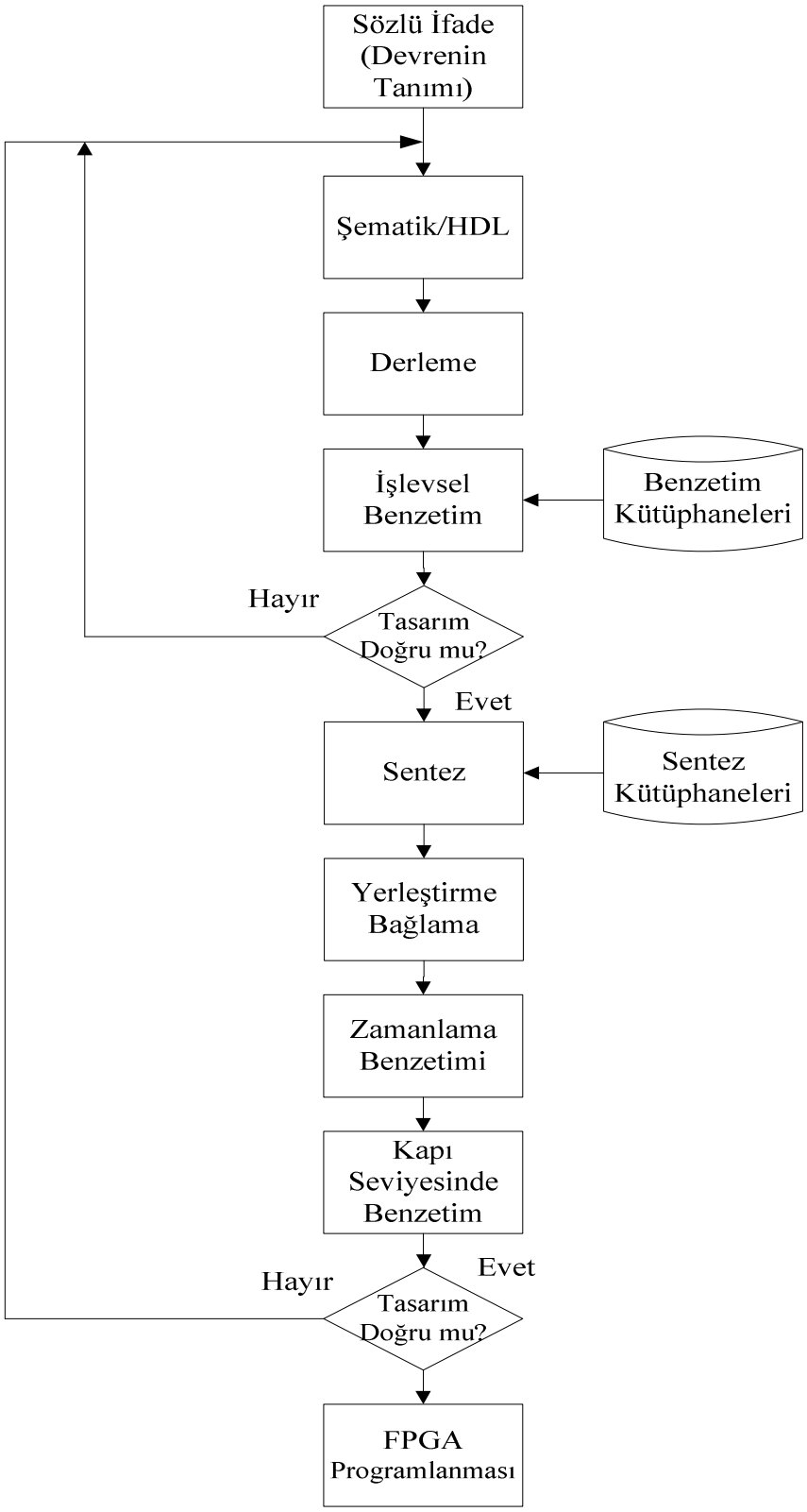
SRAM teknolojisinden daha düşük maliyeti olmasına karşın bir kere programlanabilen teknoloji olduğundan ilk örnek üretim için pahalı bir çözüm olmaktadırlar. Anti sigorta tabanlı yongalar programlanırken SRAM tabanlı yongaların aksine, özel bir programlayıcı kullanarak, yonga sistemde yok iken (offline) olarak programlanır. Anti sigorta tabanlı FPGA'ların enerji kesildiğinde bile yapılandırma verisinin (bit stream kodu) kalıcı olması gibi bir üstünlüğü vardır. Yapılandırma kalıcı olduğundan saklamak için hafıza gerektirmez. Böylece kart üzerinde yerden tasarruf sağlanır. Bu yongaların ara-bağlantı yapıları *Radiation Hardening (elektronik devre bileşenlerinin radyasyon girişimlerine karşı dirençlendiren fiziksel yapı)*'dır. Bu yapı, yongaların radyasyonun etkilerine karşı bağışıklığı olduğu anlamına gelir. Bu sayede radyasyona maruz kalınması durumunda yonganın yapılandırması bozulmaz. Bu özelliği sebebiyle askeri ve uzay uygulamalarında tercih edilirler [25].

3.2.3. FPGA Programlanması

Bir FPGA programlanmasında aşağıdaki adımlar izlenir [25].

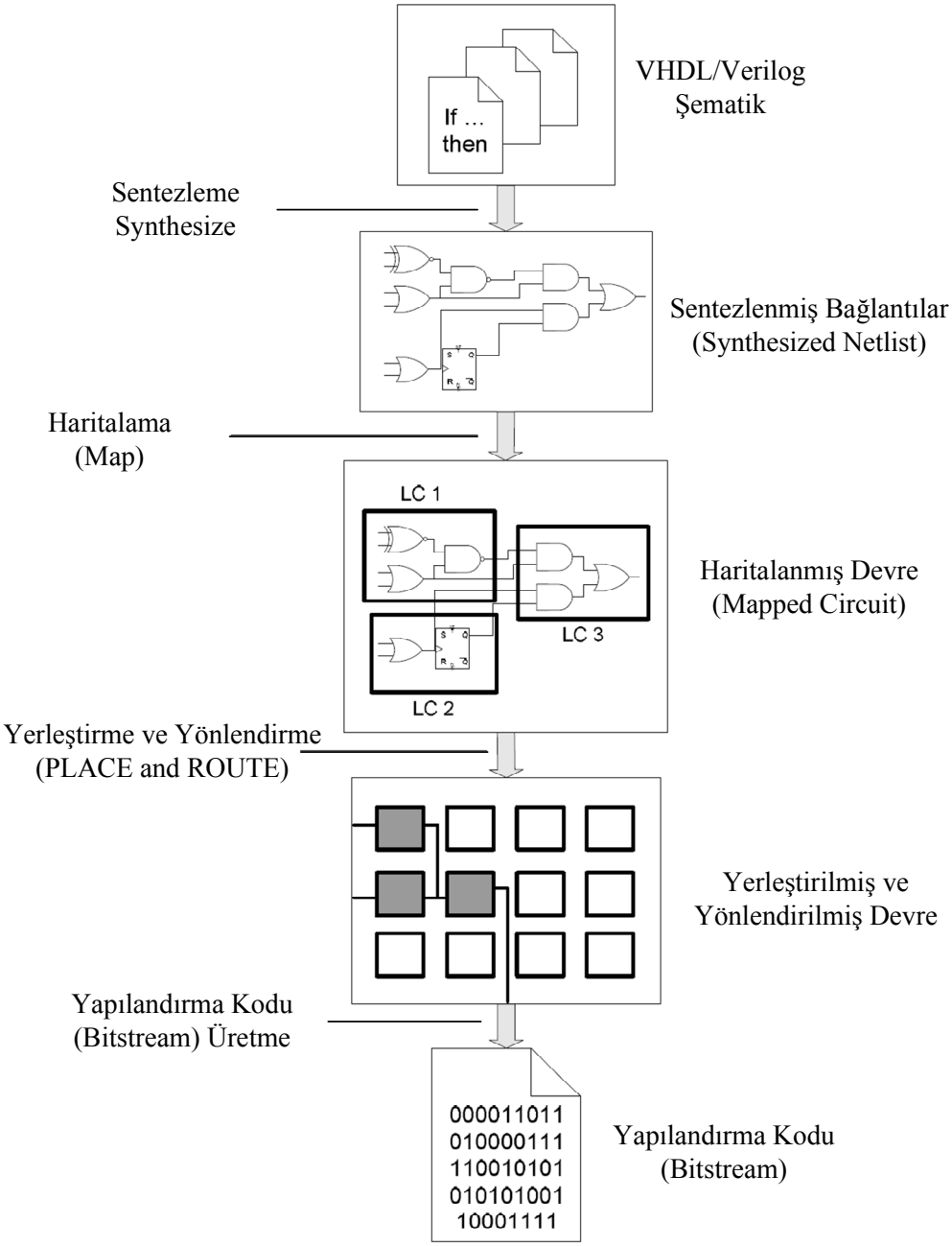
- Devrenin sözle tanımı
- Şematik veya HDL (VHDL, Verilog, vb.) kullanılarak tasarımı hazırlama
- Devreye ait standart bağlantı listesi (Netlist)
- Fonksiyonel benzetim (Functional Simulation)
- Lojik sentezleme
- FPGA seçimi
- Sentezleme işleminde varsa, devreye ait lojik kısıtlamalar (I/O, zamanlama, yerleştirme, saat frekansı, kritik yollar,..) kullanıcı kısıtlama dosyası (Xilinx FPGA için User constraints file)
- Lojik sentezleyici ile istenilen fonksiyonların gerekli lojik indirgemeleri (logic optimization)
- Elde edilen lojik fonksiyonlar FPGA içerisindeki lojik bloklarla eşleştirilmesi işlemi (technology mapping) kapı seviyesinde bir bağlantı listesi
- Teknoloji haritalaması sırasında, kullanıcı kısıtlama dosyası da kullanılarak zamanlama gereksinimi karşılanmak amacıyla gerekirse daha fazla lojik eleman kullanımı.
- Sentezleme sonrasında, yerleştirme ve yönlendirme (Placement and Routing)
- Bu aşamadan sonra kapı seviyesinde benzetimin gerçekleştirilmesi uygun olacaktır.

Yerleştirme ve yönlendirme sonrası benzetimlerde istenilen sonuçlar elde edildikten sonra FPGA' nın programlanması aşamasında kullanılacak bit dizisi, üreticinin sağladığı yazılımla elde edilir. Bu süreç izlenerek oluşturulan yapılandırma veri dosyası (bit stream kodu) FPGA üreticisi firmanın geliştirdiği uygun donanım (kablo) kullanılarak programlanması ile tasarım tamamlanır. Bu programlama adımlarına ilişkin tasarım akış algoritması Şekil 3.13'de verilmiştir.



Şekil 3.13. FPGA tasarım algoritması

Şekil 3.13'teki tasarım algoritmasının akış şeması Sekil 3.14'te verilmiştir.



Şekil 3.14. FPGA tasarım akışı

4. DONANIM TANIMLAMA DİLİ ve VHDL

Yongaların kapasitelerinin artmasıyla ve sayısal devreleri gerçekleştirmeden önce mantık diyagramlarını çizmek oldukça zorlaşmıştır. Bu sebeple, geleneksel "tasarımı yap" ve "devreyi kurarak denemeleri yap" yöntemlerinin yerini "tanımla ve sentezle" yöntemleri almıştır. HDL'nin "tanımla ve sentezle" yönteminde önemli bir rolü vardır. HDL dilleri bir elektronik sistemin tanımlanmasında, test edilmesinde ve sentezlenmesinde kullanılır. Pek çok donanım tanımlama dillerinin arasında VHDL dili en yaygın kullanılanıdır. VHDL dili ilk olarak 1980'lerin başında geliştirilmiştir. Bu dil sayesinde çok karmaşık devre tasarımlarını bile çok hızlı şekilde gerçekleştirmek mümkündür. Çünkü VHDL dili kütüphanedeki mevcut mantık elemanlarını kullanarak sistemler oluşturma yerine komutlar yardımıyla gerçekleştirir. Ayrıca VHDL dili ile yazılmış bir program, FPGA değiştirilmek istenildiğinde yazılmış program değiştirilmeden yeni FPGA'da kullanılabilir. VHDL dili gelişmiş bir programlama dili olduğundan diğer dillerde hazırlanmış fonksiyonların kullanımına imkan tanır [5, 30, 40].

4.1. VHDL Temel Bileşenleri

Bu bölümde VHDL temel bileşenleri;

- Entity (Entity)
- Mimari (Architecture)
- Paket (Package)
- Bileşen (Component)
- İşlem (Process)

4.1.1. Varlık (Entity)

Bir tasarımın en temel bloğudur. Verilen bir mantık fonksiyon için bütün giriş ve çıkışları yani mantık fonksiyonun dış dünya ile bağlantısını tanımlar. Her VHDL tasarım mutlaka en az bir entity içerir. Port tanımlamaları farklı biçimlerde olabilir. Burada port tanımlaması ile işaretin giriş/çıkış olduğu ve veri tipi belirtilir. Örnek bir entity tanımlaması sonraki sayfada verilmiştir [5, 41].

```
ENTITY entity-adı IS
PORT(giris: in STD_LOGIC;
Cikis: out STD_LOGIC_VECTOR(3 downto 0));
END entity-adı;
```

4.1.2. Mimari (Architecture)

Mimari entity'nin davranışını tanımlar. Bir entity birden fazla mimariye sahip olabilir. Bir mimari davranışsal (behavioral) tanımlama, yapısal (structural) tanımlama, veri akışı (data flow) olmak üzere üç farklı biçimde kullanılabilir. Sırasıyla genel tanımlama biçimleri aşağıda verilmiştir [5, 40, 41, 42].

Davranışsal Tanımlama

```
ARCHITECTURE architecture-adı OF entity-adı IS
Sinyal tanımlamaları;
Fonksiyon tanımlamaları;
Procedure tanımlamaları;
BEGIN
PROCESS bloğu;
Eş zamanlı işlemler;
END architecture-adı;
```

Yapısal Tanımlama

```
ARCHITECTURE architecture-adı OF entity-adı IS
Component tanımlamaları;
Sinyal tanımlamaları;
BEGIN
Anlık-adı: PORT MAP ifadeleri;
Eş zamanlı işlemler ifadeleri;
END architecture-adı;
```

Veri Akışı Tanımlama

ARCHITECTURE architecture-adı OF entity-adı IS
Sinyal tanımlamaları;
BEGIN
Eş zamanlı işlemler;
END architecture-adı;

4.1.3. Paket (Package)

Paket; entity üniteleri tarafından kullanılan tanımlamaları bir grup haline getirir ve paylaştırır. Kullanımı aşağıdaki verilmiştir [30, 42]

PACKAGE paket-adı IS
Tip tanımlaması;
Alt tip tanımlaması;
Sinyal tanımlamaları;
Değişken tanımlamaları;
Sabit tanımlamaları;
Component tanımlamaları;
Fonksiyon tanımlamaları;
Procedure tanımlamaları;
END paket-adı;

4.1.4. Bileşen (Component)

Bileşen yapısı; devre tanımlamasında bir alt devre gibi kullanılan bileşenin adını ve ara yüzünü tanımlar. Aşağıdaki gibi kullanılır [42].

COMPONENT bileşen–adı IS
PORT(port-adı listeleri ve tipleri);
END COMPONENT;

4.1.5. İşlem (Process)

İşlem bloğu sıralı şekilde gerçekleştirilecek durumları içerir. Bir mimaride birden fazla işlem bloğu anlık gerçekleştirilir. Yani işlem blokları aynı anda başlar ve her bir işlem bloğu kendi içinde satır satır sıralı olarak gerçekleştirilir. Kullanımı aşağıda gibidir [42].

işlem-adı: PROCESS(Hassasiyet-listesi)
Değişken tanımlamaları;
BEGIN
Sıralı-deyimler;
END PROCESS işlem-adı

4.2. Veri Tipleri

VHDL nesnel özelliklere sahip olan bir dildir. Nesnelerin davranış ve işlevlerine göre fonksiyonellik kazanır[42].

4.2.1. Sinyal (Signal)

Sinyaller entity, mimari ve paket içinde kullanılabilir. Sinyallerin paket içinde kullanımı çok önemlidir. Çünkü paket genel bir ifade olduğundan dolayı işaretler burada kullanıldığı zaman diğer entity veya mimariler tarafından çağrılarak kullanılabilir. Sinyallere başlangıç değeri atanabilir ve işlem gerçekleşirken bu ifade yenilenebilir [30]. Kullanımı, aşağıdaki verilir.

SIGNAL sinyalin-adı: sinyal_tipi [:= başlangıç değeri];

4.2.2. Değişken (Variable)

Genel olarak geçici değerleri kullanmak için tercih edilir. Böylece programın daha hızlı çalışması sağlanabilir. Fakat gerçek zamanlı işlemlerde özellikle tasarımın FPGA' de gerçeklenmesinde değişken kullanımının sakıncaları bulunmaktadır [30].

VARIABLE değişkenin-adı: değişkenin tipi [:= başlangıç değeri];

4.2.3. Sabit (Constant)

Eğer tasarımda değişmez verilere ihtiyaç duyulursa sabitler kullanılır. Bu durum, tasarımın daha iyi gözlemlenebilmesi ve anlaşılabilmesi için önemli bir faktördür. Özellikle tasarım içinde sık sık kullanılan değeri aynı olan ifadeler için vazgeçilmez bir veri tipidir. Kullanımı aşağıdaki gibidir [30].

CONSTANT sabitin-adı: sabitin tipi [:= başlangıç değeri];

4.2.4. Ön tanımlamalı veri tipleri:

Bu veri tipleri aşağıda verilmiştir. STD_LOGIC ve STD_LOGIC_VECTOR tiplerinin kullanımı için, VHDL tasarım dosyası "std_logic_1164" paketini içermelidir [30].

Standart Mantık Tip: STD_LOGIC, STD_LOGIC_VECTOR
Bit Tip: BIT, BIT_VECTOR
Tam Sayı Tipi: INTEGER
Kayan noktalı Tip: REAL
Fiziksel Tip: TIME
Liste Tip: BOOLEAN, CHARACTER

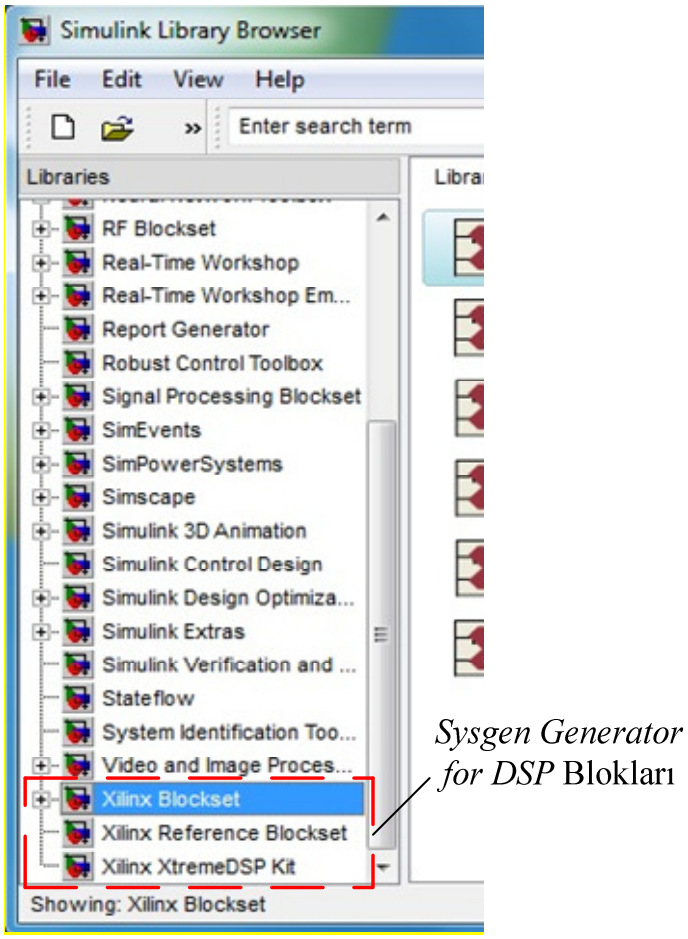
4.3. System Generator for DSP

System Generator FPGA üzerinde donanım tasarımını kolaylaştıran sistem düzeyinde modelleme aracıdır. *System Generator* donanım tasarımı için yüksek seviyeli soyutlama yoluyla modelleme yapar ve bir buton aracılığıyla tasarlanan sistemin FPGA'ya derlenmesini sağlar. Araç kutusu aynı zamanda düşük seviyeli soyutlama yoluyla FPGA kaynaklarının çok verimli olarak kullanılmasına olanak tanır. Bir tasarım için gerekli olan birçok öğe *System Generator for DSP* araç kutusunda mevcuttur. Hazırlanan tasarımı Oluştur (Generate) düğmesine basılarak oluşturulan modeli HDL koduna (VHDL, Verilog,

vb.) dönüştürür. Derlenen kod, yükleme aracı (Xilinx FPGAları için IMPACT) ile FPGA yapılandırılır. Derlenen kodlar; HDL kodlarını, tasarım için gerek duyulan saat darbeleri ve alt sistemler için saat darbelerini (aktif etme, resetleme), tasarım test sinyallerini (testbench), sentezleme araçları için proje dosyalarını içerir [16-18].

4.3.1. System Generator for DSP Blokları

System Generator for Dsp blokları *Simulink* kütüphane tarayıcısı (*Simulink* Library Browser) içerisinde (Şekil 4.1) bulunmaktadır.



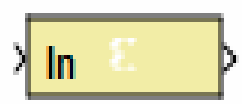
Şekil 4.1. *Simulink* kütüphane tarayıcısı içinde *System Generator for Dsp* konumu

System Generator for Dsp blokları *Simulink* blokları gibi kullanılırlar. Bu bloklar matematiksel mantık, hafıza ve DSP fonksiyonları için soyutlama sağlarlar. Ayrıca bu

bloklar diğer yazılım araçları (FDATool, Modelsim) ile bağlantı kurabilirler. Bazı *System Generator for Dsp* blokları açıklanmıştır [16-18].

Gateway In Bloğu

Tasarımın *System Generator for Dsp* bloklarından oluşan bölümüne giriş için kullanılır. Şekil 4.2'de *Gateway In* bloğu verilmiştir [16-18].



Şekil 4.2. *Gateway In* bloğu

Gateway In blokları aşağıdaki fonksiyonları gerçekleştirir;

- *Simulink* tamsayı, çift tamsayı (double) sabit noktalı (fixed point) veri tiplerinden *System Generator* sabit noktalı veri tipine dönüştürür.
- *System Generator* tarafından derlenen HDL kodunda giriş portu oluşturur.
- *System Generator* bloğunda test sinyalleri (Testbench) oluşturma seçilmiş ise test sinyalleri uyaranları tanımlanır.

Gateway Out Bloğu

Tasarımın *System Generator* bölümünün çıkış bloğudur. Şekil 4.3'de *Gateway Out* bloğu gösterilmiştir [16-18].

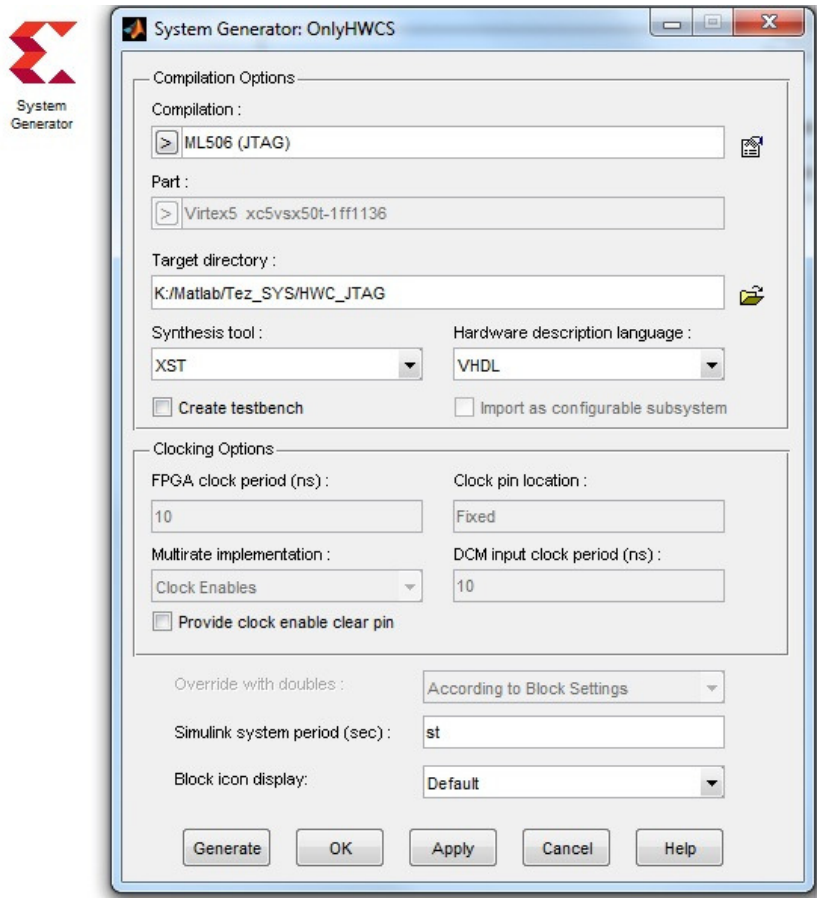
Şekil 4.3. *Gateway Out* bloğu

Gateway Out bluğu

- *System Generator* sabit noktalı veri tipinden *Simulink* çift tamsayı (double) veri tipine dönüşüm sağlar.
- *System Generator* tarafından derlenen HDL kodunda çıkış portu oluşturur.
- *System Generator* bloğunda test sinyalleri (Testbench) oluşturma seçilmiş ise test sinyalleri uyaranları tanımlar.

***System Generator* Bloğu**

Her tasarım çalışmasında en az bir tane *System Generator* bloğu bulunmalıdır.Bu blok sistemin ve benzetim parametrelerinin kontrolünü sağlar. Parametreleri özelleştirerek kod üretimini ve benzetimi yürütür. Şekil 4.4'de *System Generator* bloğu ve sistem parametrelerinin ayarlandığı özellikler penceresi gösterilmiştir [16-18].



Şekil 4.4. *System Generator* bloğu ve özellikler penceresi

Bu blokla Derleme seçenekleri bölümünde aşağıdaki parametreler aşağıdaki sistem parametreleri ayarlanır.

- Derleme (Compilation) paremetresi derleme tipi seçilir. Derleme türleri []' de detaylı olarak verilmiştir.
- Aygıt (Part) parametresi gerçeklemede kullanılacak FPGA aygıtını belirler.
- Hedef dizin (Target directory) parametresi derleme dosyalarının kaydedileceği dizini tanımlar.
- Sentezleme aracı (Synthesis tool) parametresi ile sentezleme yapılarken kullanılacak sentezleme aracı seçilir.
- Donanım tanımlama dili (Hardware Description Language) paremetresi kodların derleneceği dil seçilir.
- Test sinyali (Create testbench) oluşturulan HDL'nin test edilmesi için sinyaller oluşturur.

Saatlama seçenekleri (Clocking options) bölümünde Sistem saat darbesi hızı ve aygıt üzerinde hangi bacak üzerine bağlanacağı tanımlanır ve derleme işlemnde *Xilinx ISE* projesi olan kullanıcı sınırlama dosyasına (.ucf) işlenir. Ayrıca alt sistemler ve aktif etme uçları için saat darbesi tanımlamaları da ayarlanır. *Simulink* sistem periyotu (*Simulink* system period) ile *Simulink* sistem periyotu saniye biriminde ayarlanır [16-18].

Constant Bloğu

Bu blok sabit noktalı, ikili (Boolean) ya da DSP48 veritipindesabit bir değer üretir. Şekil 4.5'te *constant* bloğu gösterilmiştir [16-18].

Şekil 4.5. *Constant* bloğu

Bu blok genel olarak veri tipi, sabit değeri, örnekleme sabiti parametrelerine sahiptir.

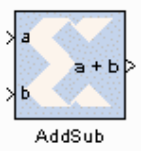
Convert Bloğu

Bu blok girişini arzulanan aritmetik tipine örnekler. Bu blok aritmetik tipi, nicemleme, taşma, aktifleştirme ucu seçme gibi parametrelere sahiptir. Şekil 4.6'da *Convert* blok şekli verilmiştir [16-18].

Şekil 4.6. *Convert* bloğu

AddSub Bloğu

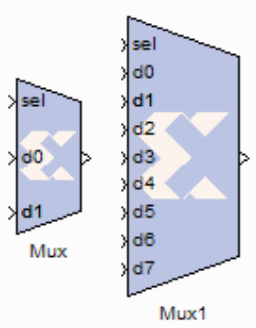
AddSub bloğu toplama /çıkarma işlemini gerçekleştririr. Bu blok parametrelerinde işlem türü, Taşma girişi, taşma çıkışı, aygıtın verimli kullanımı için HDL davranışsal benzetiminin kullanılması gibi parametrelere sahiptir. Şekil 4.7'de *AddSub* bloğu gösterilmiştir [16-18].



Şekil 4.7. *AddSub* bloğu

Mux Bloğu

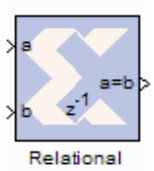
Bu blok çoğullayıcı işlemini gerçekleştirir. *Mux* bloğu bir seçme girişi ve kullanıcı tanımlı (2-1024 adet) veri girişlerine sahiptir. Şekil 4.8'de *Mux* bloğu şekli verilmiştir [16-18].



Şekil 4.8. *Mux* bloğu

***Relational* Bloğu**

Bu blok karşılaştırıcı bloğudur. Şekil 4.9'da *Relational* bloğu gösterilmiştir bu bloğun çıkışı ikili veri tipine sahiptir [16-18].



Şekil 4.9. *Relational* bloğu

Gerçekleştirdiği karşılaştırma işlemleri aşağıda verilmiştir.

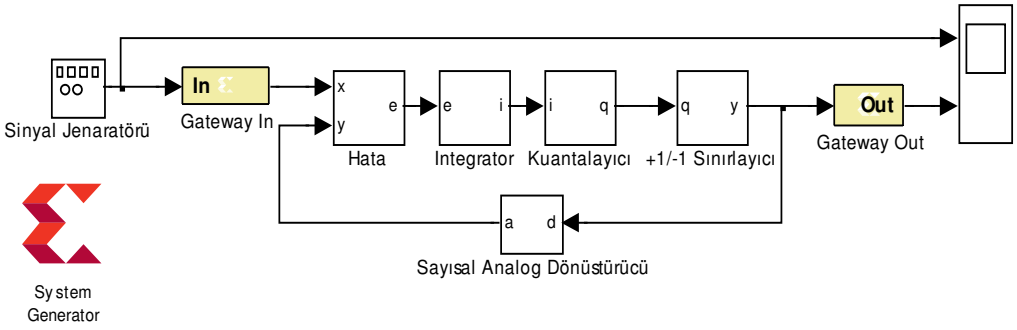
- Eşit (a = b)
- Eşit değil (a != b)
- Küçük (a < b)
- Büyük (a > b)
- Küçük eşit (a <= b)
- Büyük eşit (a >= b)

4.3.2 Hardware Co-Simulation Modu

Bu mod *Simulink* çalışan bir benzetimine FPGA'yı doğrudan dahil eder. *System Generator* derleme işlemi esnasında hedef aygıt için *bitstream* (.bit, FPGA yapılandırma dosyası) oluşturur ve *Hardware Co-Simulation* bloğuna bu dosyayı ilişkilendirir. Tasarım benzetimi gerçekleştirildiğinde derlenmiş kısmın aygıt üzerinde kaplayacağı alan hesaplanır. Bu derlenmiş tasarımın gerçek aygıtta test edilmesine olanak tanır ve benzetimi çalışmalarını hızlandırır. FPGA geliştirme kartı ile *Hardware Co-Simulation* bağlantı genel olarak Ethernet ve JTAG kablosu üzerinden yapılabilir. Kullanılan kart bu modu desteklemelidir [16-18].

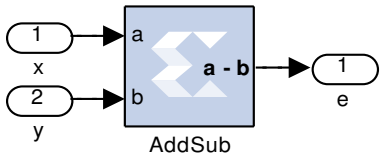
5. DENEYSEL ÇALIŞMA

Deneysel çalışmada ”System Generator for DSP” kullanılarak birinci derece SDM FPGA üzerinde gerçeklenmiştir. Oluşturulan sistem Şekil 5.1’de verilmiştir.



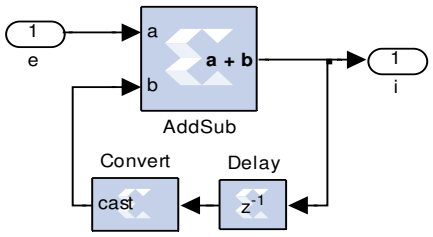
Şekil 5.1. Birinci derece SDM

Şekil 5.1’de verilen hata bloğu iç yapısı Şekil 5.2’deki verildiği gibi oluşturulmuştur.



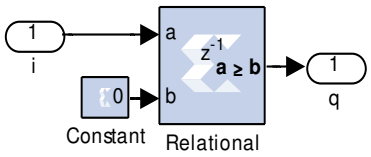
Şekil 5.2. Hata bloğu iç yapısı

Şekil 5.2’deki hata bloğunda, dönüştürülecek giriş sinyali ile geri besleme yolundan gelen çıkış sinyalinin farkı alınır. Elde edilen fark sinyali Şekil 5.3’de gösterilen integratör girişine uygulanır.

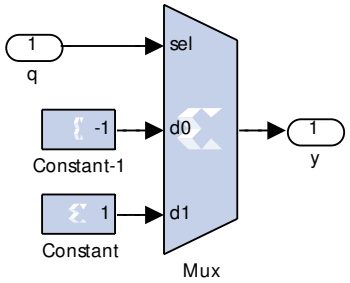


Şekil 5.3. İntegratör bloğu iç yapısı

Ayrık integratör yapısında yer alan Cast bloğu integrasyon işlemindeki bit uzunluğu uyumunu sağlamak için kullanılmıştır. Kullanılan nicemleyici yapısı, Şekil 5.4'te verilmiştir. SDM çıkışını +1/-1 arasında sınırlandırmak için hazırlanan Sınırlayıcı bloğu yapısı Şekil 5.5'te verilmiştir.

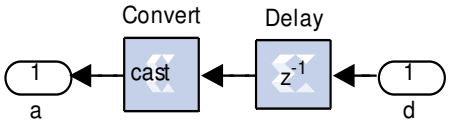


Şekil 5.4. Nicemleyici bloğu iç yapısı



Şekil 5.5. +1/-1 Sınırlayıcı bloğu yapısı

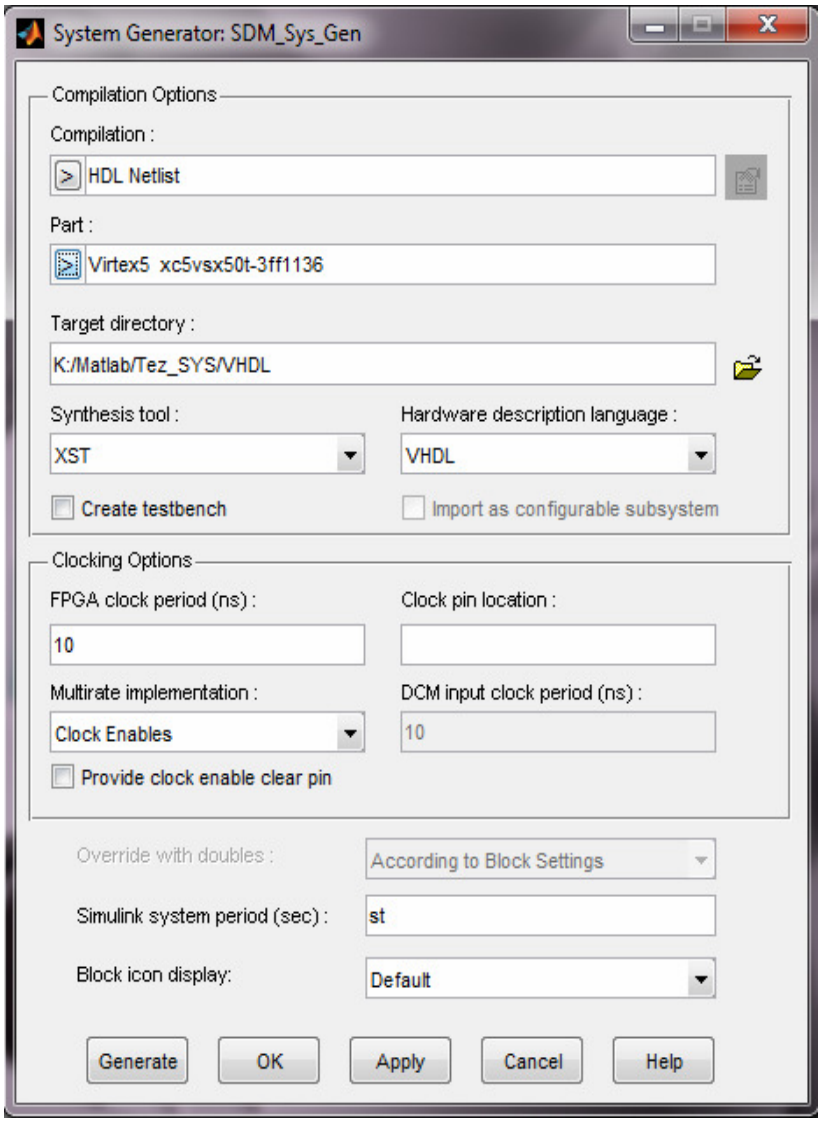
Şekil 5.6'da geri besleme yolunda kullanılan 1-bitlik Sayısal/Analog dönüştürücü yapısı gösterilmiştir.



Şekil 5.6. Sayısal/Analog dönüştürücü

Tasarlanan Birinci derece SDM'nin FPGA gerçeklemesi için gerekli olan VHDL kodları *System Generator Token* araç kutusu kullanılarak derlenmiştir. Derleme işlemi Şekil 5.7'de gösterilen *System Generator Token* özelliklerinden Derleme Seçenekleri (Compilation Options) bölümünden derleme tipi (Compilation) HDL Netlist, Aygıt (Part) kısmından kullanılacak aygıt modeli (FPGA), Hedef Dizin (Target Directory) bölümünden derlenecek dosyalar için hedef dizin konumu, Sentezleme aracı (Synthesis Tool) bölümünden Xilinx Sentezleme aracı (XST; Xilinx Synthesis Tool), Donanım Tanımlama

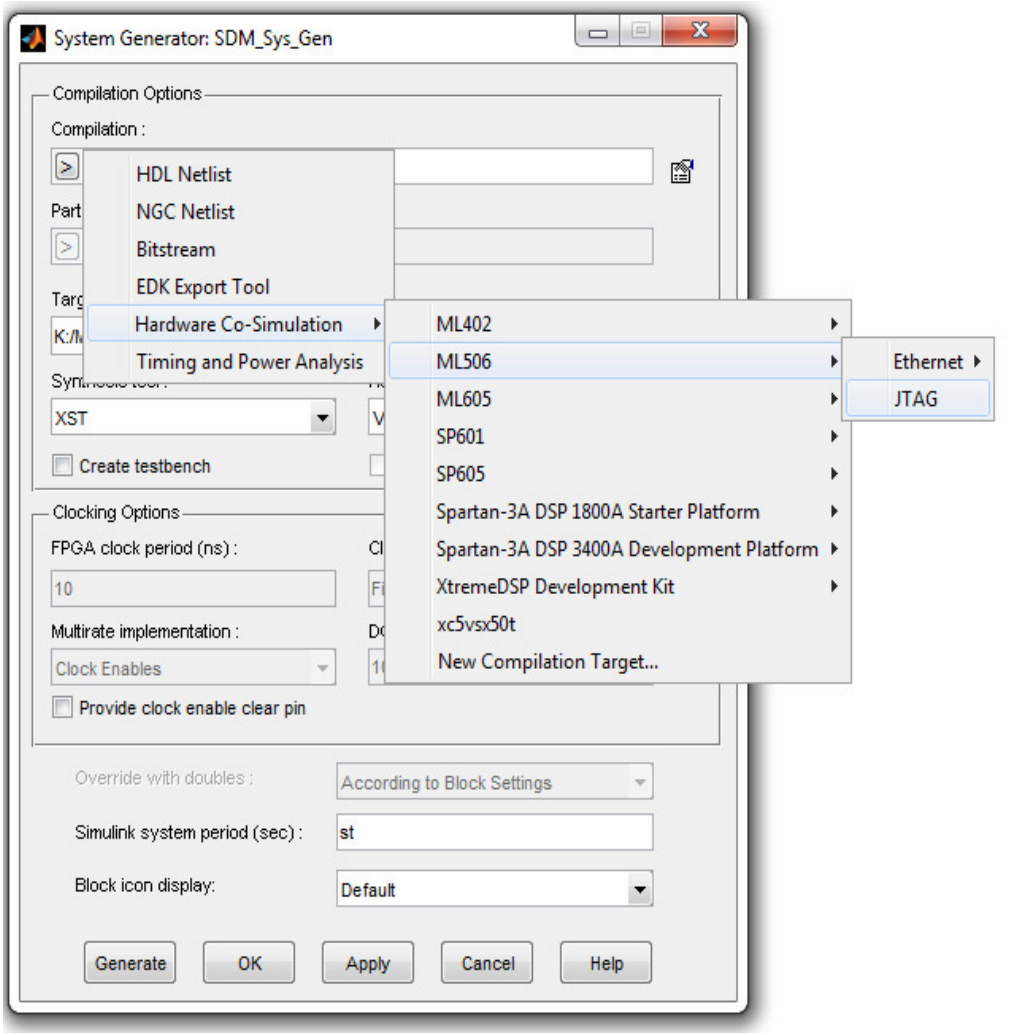
Dili (Hardware Desciription Language) seçeneğinden VHDL, sistem için belirlenen saat darbelerinin hızı süre olarak ve bacak (pin) konumu ve son olarak *Simulink* sistem periyot (*Simulink* system period) seçeneğinden örnekleme periyodu olarak seçilir. Oluştur (Generate) butonu ile VHDL kodları ürettirilir. Üretilen programlar belirtilen klasöre Xilinx ISE projesi olarak kaydedilir (Ek A'da derlenen VHDL kodu verilmiştir).



Şekil 5.7. System Genetor Token özellikleri

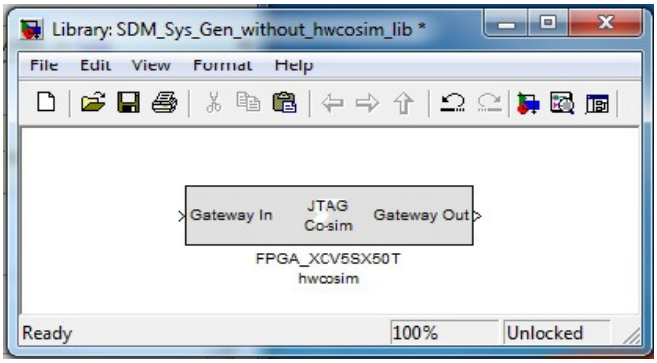
Tasarlanan SDM' nin test edilmesi için *System Generator'un Hardware Co-Simulation* modu kullanılarak tasarıma ve kullanılacak aygıta özgü *Hardware Co-Simulation* bloğu hazırlanır. Hazırlanmasında yukarıda olduğu gibi *System Generator*

Token özellik penceresinden Derleme Seçenekleri (Compilation Options) bölümünden Derleme (Compilation) seçeneğinden, Şekil 5.8'de gösterildiği gibi ML506 (JTAG) seçilir, Aygıt kısmı ve saat darbesi seçenekleri aktif değildir. Hedef dizin bölümünden *Hardware Co-Simulation* bloğu ve Xilinx ISE proje dosyalarının oluşturulacağı klasör tanımlanır. Sentezleme aracı bölümünden XST Xilinx Sentezleme aracı, Donanım tanımlama dili seçeneğinden (Hardware Desciription Language) VHDL, *Simulink* periyot süresi seçilip oluştur butonu ile *Hardware Co-Simulation* bloğu ürettirilir.



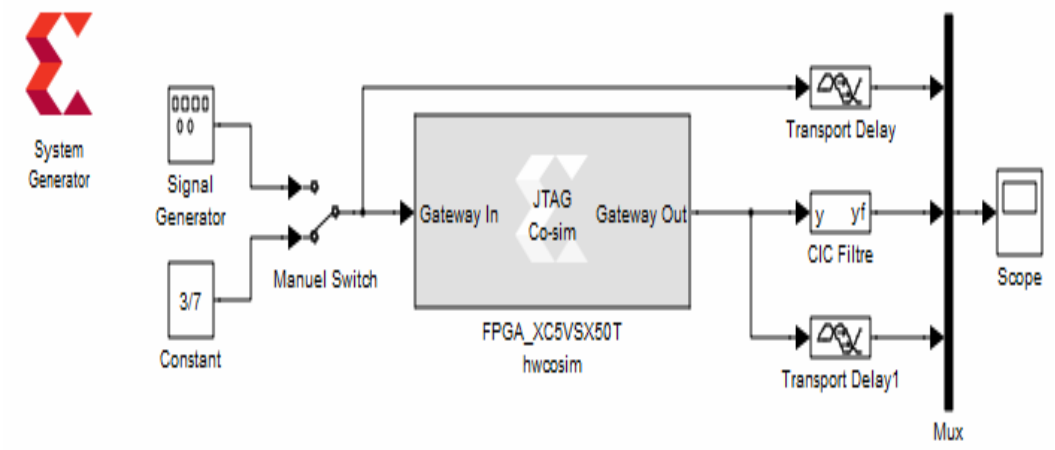
Şekil 5.8. Hardware Co-Simulation Modu için derleme tipinin seçilmesi

Derleme sonucunda oluşan ve FPGA'yı temsil eden blok Şekil 5.9'da verilmiştir.



Şekil 5.9. Birinci derece SDM için derlenen Hardware Co-Simulation bloğu

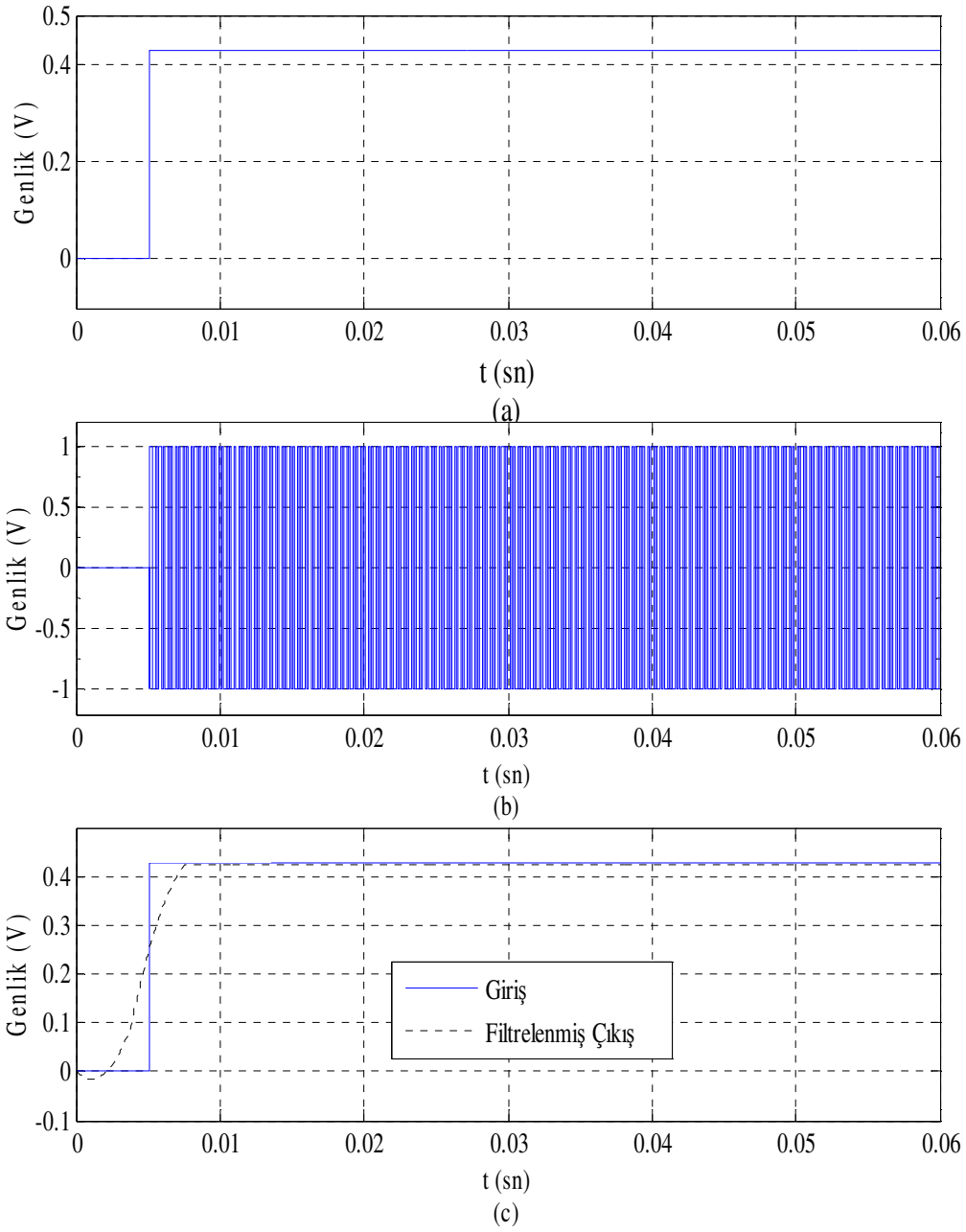
Şekil 5.1'de verilen birinci derece SDMnin, *System Generator Hardware Co-Simulation* eşdeğeri Şekil 5.10'da verilmiştir.



Şekil 5.10. Birinci derece SDM'nin *Hardware Co-Simulation* eşdeğeri

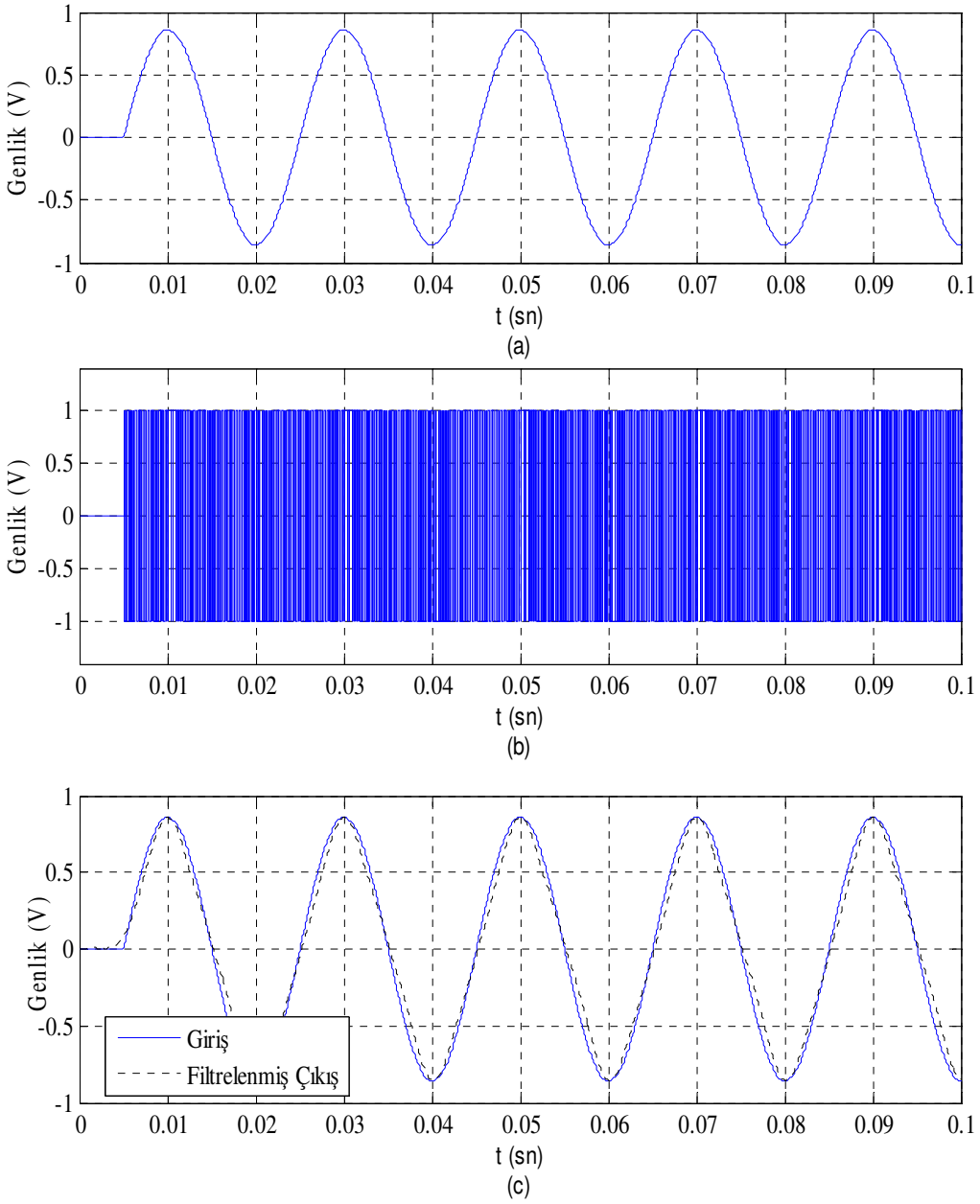
Şekil 5.10'da Sinyal Üreteci bloğu sinüs sinyali, Constant bloğu DC sinyali üretmek için kullanılır. Çıkış sinyallerini incelerken süzgeçlerdeki gecikmelerden faz gecikmesini ihmal etmek için iletim gecikmesi (Transport Delay) bloklarından faydalanılmıştır. Modülatörün çıkışı Seri birleştirilmiş tarak süzgeci (SBT(CIC)) kullanılarak tekrar sürekli zamana dönüştürülmüştür.

Gerçeklenen SDM'nin doğruluğunu test etmek için kullanılacak olan Şekil 5. 10'daki *Hardware Co Simulation* bloğunun girişine 3/7V sabit uygulanmıştır. Örnekleme frekansı 25.6 kHz alındığında oluşan SDM çıkış sinyali Şekil 5.11'de verilmiştir.



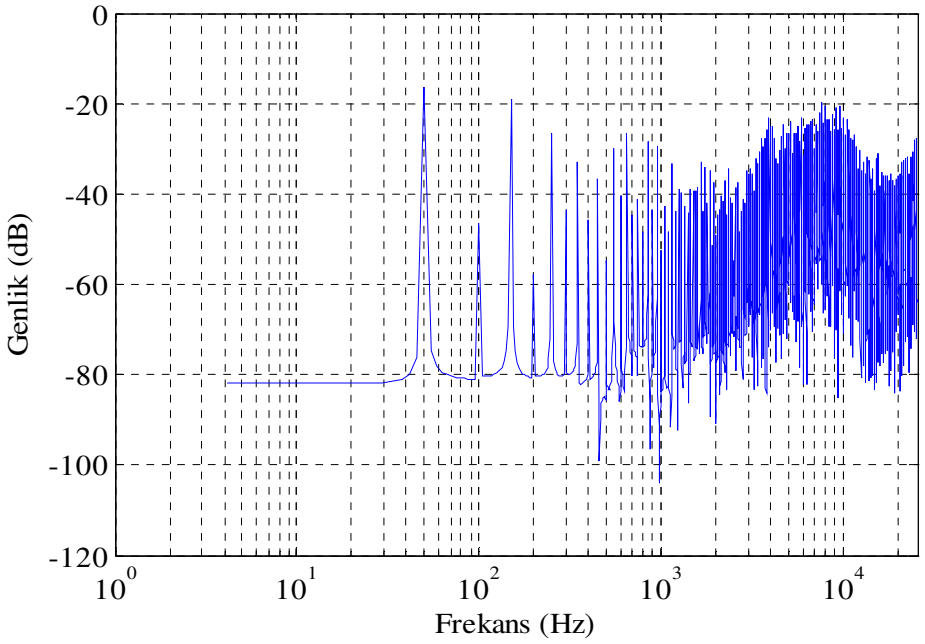
Şekil 5.11. 3/7V Sabit giriş için birinci derece SDM'nin a) Giriş Sinyali, b) SDM çıkışı c) Giriş ve Süzgeçlenmiş SDM çıkışı

Şekil 5.10'daki bloğun girişine 0.85V 50Hz'lik sinüs sinyali uygulandığında oluşan SDM sinyalleri Şekil 5.12'de verilmiştir. Aşırı Örnekleme Oranı, 512 olarak seçilmiştir. Şekil 5.12 a'da giriş sinyali, b'de çıkış sinyali, c'de ise uygulanan giriş sinyali ile SBT süzgeci kullanılarak elde edilmiş sürekli zamanlı sinyal gösterilmiştir.

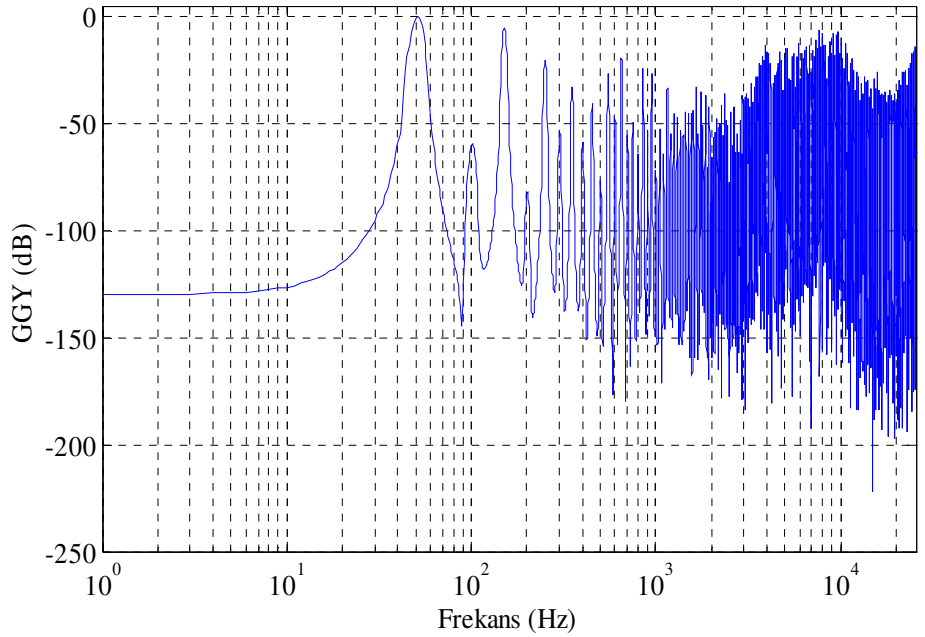


Şekil 5.12. 0.85V 50Hz'lik Sinüs giriş için birinci derece SDM'nin a) Giriş Sinyali, b) SDM çıkışı c) Giriş ve Süzgeçlenmiş SDM çıkışı

SDM çıkışının OSR 512 için elde edilen görüngesi Şekil 5.13'de, güç görünge yoğunluğu (GGY) ise Şekil 5.13'de verilmiştir.

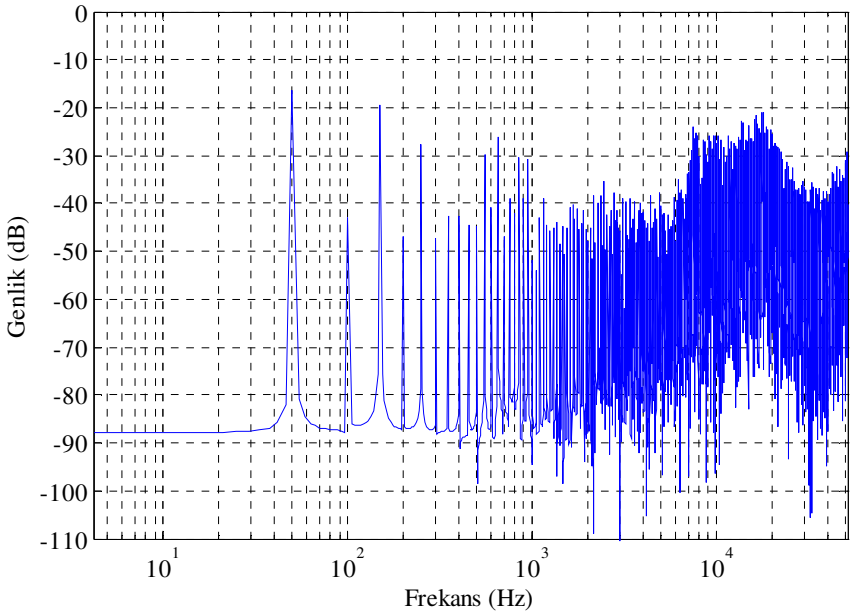


Şekil 5.13. SDM Çıkışının görüngesi

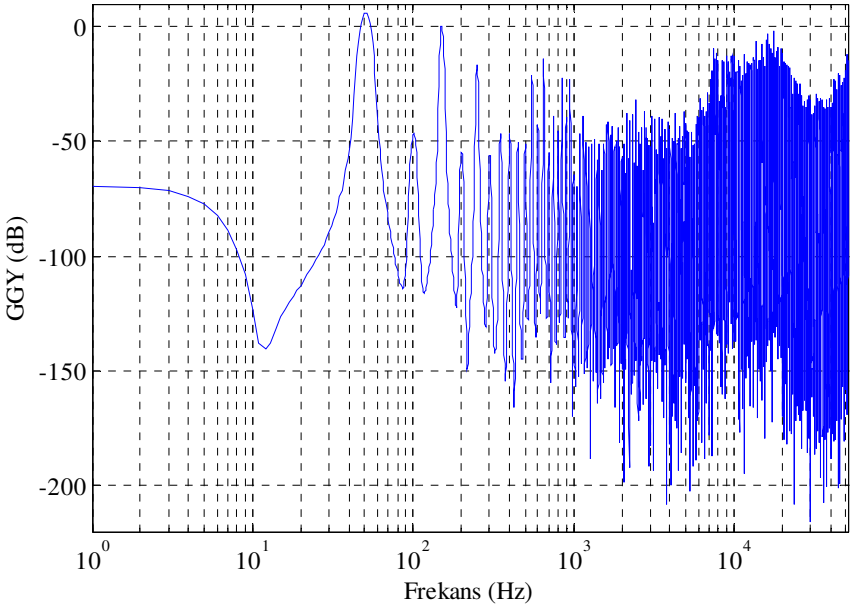


Şekil 5.14. SDM Çıkışının güç görünge yoğunluğu

SDM çıkışının OSR 1024 için elde edilen görüngesi Şekil 5.15'de, güç görünge yoğunluğu ise Şekil 5.16'da verilmiştir.



Şekil 5.15. SDM Çıkışının görüngesi



Şekil 5.16. SDM Çıkışının güç görünge yoğunluğu

6. SONUÇ

Bu tez çalışmasında sürekli zamanlı sinyalleri ayrık zamanlı forma yüksek çözünürlükte çevirmek amacı ile kullanılan SDM'ler incelenmiştir. Bu amaçla birinci derece ayrık zamanlı SDM, FPGA kullanılarak hem benzetim hemde pratik olarak gerçeklenmiştir.

İlk olarak birinci derece sürekli ve ayrık zamanlı SDM devrelerinin *MATLAB* ortamında benzetimleri yapılmıştır. Anlatılan teorik bilgiler Bölüm 2 ve Bölüm 5'te yapılan uygulamalar üzerinde açıklanarak konuların daha iyi anlaşılması amaçlanmıştır.

Daha yüksek dereceli SDM yapıları birinci derece SDM'ler kullanılarak yapılabileceğinden birinci derece SDM, SDM analiz ve sentezinde temel rol oynar.

Bu çalışmadan elde edilen sonuçlar ileride yapılacak olan SDM tasarım çalışmalarında kullanılacaktır. Ayrıca ses ve görüntü işleme gibi farklı alanlarda kullanılacaktır. Yüksek çözünürlükte sinyal işleme yapılacağından sistemlerin performansının artacağı düşünülmektedir.

KAYNAKLAR

[1]. **Altınok, G. D**., 2009. [Medikal ultrason görüntüleme uygulaması amacıyla GUI ile yürütülen sigma-delta modülatör tasarım ve ölçüm aracı, *Yüksek Lisans Tezi,* Boğaziçi Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

[2]. **Yetik, Ö**., 2007. Sigma-delta kipleyiciler için eleman idealsizliklerini de hesaba katan bir otomatik mimari geliştirici, *Yüksek Lisans Tezi,* Boğaziçi Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

[3]. **Sağlamdemir, M. O**., 2007. Sigma-delta kipleyicilerin gerçeklenmesi ve başarımlarının değerlendirilmesi, *Yüksek Lisans Tezi,* Boğaziçi Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

[4]. **Iyappan, P., Jamuna, P. and Vijayasamundiswary, S**., 2009. Design of Analog to Digital Converter Using CMOS Logic, *International Conference on Advances in Recent Technologies in Communication and Computing'09, 27-28* October 2009, 74–76, (in India).

[5]. **Sırmaçek, B**., 2007. FPGA ile mobil robot için öğrenme algoritması modellenmesi, *Yüksek Lisans Tezi,* Yıldız Teknik Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

[6]. **Foo, S., Moss, P., Norton, T. and Stafford, D**., 2004. Fifth-Order Sigma-Delta Modulator with Decimation, System Theory, *Proceedings of the Thirty-Sixth Southeastern Symposium on* , pp.522 – 526.

[7]. **Abeysekera, S. S**., 1996. Stability Analysis of Sigma - Delta Modulators Using a Non-Linear Technique, *Signal Processing and Its Applications, Fourth International Symposium on ,* vol: 1, pp. 242 – 245.

[8]. **Al-Alaoui, M. A., and Ferzli, R**., 2006. An Enhanced First-Order Sigma-Delta Modulator With a Controllable Signal-to-Noise Ratio, *Circuits and Systems I: Regular Papers, IEEE Transactions on,* vol: 53 , pp. 634 – 643.

[9]. **Hanbay, D.,** 2003. İkinci dereceden sigma-delta modülatörünün pratik olarak gerçekleştirilmesi ve incelenmesi, *Yüksek Lisans Tezi,* Fırat Üniversitesi, Fen Bilimleri Enstitüsü, Elazığ.

[10]. **Norsworthy, S. R., Schreier, R., Temes, G.C.,** 1996**.** *Delta-Sigma Data Converters: Theory, Design, and Simulation,* IEEE Press The Institute of Electrical and Electronics Engineers,Inc.,New York.

[11]. **Temes. G. C., Candy, J. C**., 1990. A Tutorial Discussion of the Oversampling Method for A/D and D/A Conversion, *Circuits and Systems, IEEE International Symposium on* , vol.2 pp.910 – 913.

[12]. **Wang, H., Brennan, P. V and Jiang, D**., 2007. FPGA Implemention of Sigma-Delta Modulators in Fractional-N Frequency Synthesis, *Signals, Circuits and Systems, ISSCS International Symposium on,* vol:1, pp.1 – 4.

[13]. **Tagzout, S., Achour, K., Djekoune, O.,** 2001. Hough transform algorithm for FPGA implementation, *Signal Processing*, 81, 1295-1301.

[14]. **Kuusilinna,K., Hamalainen, T., Saarinen, J.,** 1999. Practical VHDL optimization for timing critical FPGA applications, *Microprocessors and Microsystems,* 23, 459–469.

[15]. **Saidani ,T., Dia, D, Elhamzi, W, Atri, M and Tourki, R.,** 2009. Hardware Co-simulation For Video Processing Using Xilinx System Generator, *Proceedings of the World Congress on Engineering 2009 Vol I WCE 2009,* July 1 - 3, 2009, London, U.K.

[16]. System Generator for DSP User Guide UG640 (v11.4), http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/sysgen_user.pdf, 01 Mayıs 2011.

[17]. System Generator for DSP Getting Started Guide UG639 (v 12.3), http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/sysgen_gs.pdf, 01 Mayıs 2011.

[18]. System Generator for DSP Reference Guide UG638 (v 12.3), http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/sysgen_ref.pdf, 01 Mayıs 2011.

[19]. **Gökdel, Y. D**., 2007. [Yüksek performanslı uyarlamalı sıgma delta modülatör tasarımları, Boğaziçi Üniversitesi, *Yüksek lisans tezi*, Fen Bilimleri Enstitüsü, İstanbul.

[20]. **Genç, N**., 2007.Sigma-delta modülatörlerde dijital filtre tasarımı, hata modellemesi ve düzeltilmesi, Boğaziçi Üniversitesi, *Yüksek lisans tezi,* Fen Bilimleri Enstitüsü, İstanbul.

[21]. **Kurşunoğlu, S**., 2009. Yüksek performanslı uyarlanabilir sigma delta modülatör tasarımları, Boğaziçi Üniversitesi, *Yüksek lisans tezi,* Fen Bilimleri Enstitüsü, İstanbul.

[22]. **Bilge ,H. Ş**., 2003. Delta-sigma örneklemeli altdizilim işlemeye dayalı bir demetleme yöntemi, Başkent Üniversitesi, *Yüksek lisans tezi,* Fen Bilimleri Enstitüsü, Ankara.

[23]. **Aziz, P.M., Sorensen, H.V., vn der Spiegel, J.,** 1996**.**An Overview of Sigma- Delta Converters, Signal Processing Magazine, IEEE, Volume: 13, pp. 61 – 84.

[24]. **Gacar, A**., 2009. FPGA tabanlı görüntü işleme arabirimi, *Yüksek Lisans Tezi,* Ege Üniversitesi, Fen Bilimleri Enstitüsü, İzmir.

[25]. **Güdenler, İ**., 2010. Saklayıcı – bellek mimarisinde, 16 – bitlik, gömülü sistem (mikroişlemci) tasarımı ve sentezlenmesi, *Yüksek Lisans Tezi,* Dumlupınar Üniversitesi, Fen Bilimleri Enstitüsü, Kütahya.

[26]. **Zeidman, B**., 2002. *Designing with FPGAs and CPLDs*, CMP Books, Kansas.

[27]. **Maxfield, C.,** 2004.*The Design Warrior's Guide to FPGA's*, Newness Elsevier Inc., Burlington,USA.

[28]. **Yılmaz, N.,** 2008. Alan programlamalı kapı dizileri (FPGA) Üzerinde bir YSA'nın tasarlanması ve donanım olarak gerçekleştirilmesi, , Selçuk Üniversitesi, *Yüksek lisans tezi* Fen Bilimleri Enstitüsü, Konya.

[29]. **Parnell, K., and Mehta N**., Programmable Logic Design Quick Start Handbook, http://webserv.kmitl.ac.th/~taweepolsuesut/logic.pdf, 20 Aralık 2010.

[30]. **Dikmeşe, Ş.,** 2007. Kablosuz haberleşme sistemlerinde FPGA uygulaması, Kocaeli Üniversitesi, *Yüksek lisans tezi,* Fen Bilimleri Enstitüsü, Kocaeli.

[31]. **Cofer, R.C., and Harding, B. F.,** 2006. *Rapid System Prototyping with FPGA's*, Elsevier Inc., Burlington.

[32]. **Jong-Ru Guoa, J-R., Youa, C., Zhoua, K., Chua, M., Currana, P., F., Diaoa, J., Godab, B., Krafta, R., P., McDonalda, J., F**., 2005. A 10GHz 4:1 MUX and 1:4 DEMUX implemented by a Gigahertz SiGe FPGA for fast ADC, *Integration, the VLSI Journal*, 38, 525-540.

[33]. **Kuon, I., Tessier, R. and Rose, J.,** 2008. *FPGA Architecture: Survey and Challenges*, the essence of knowledge, Boston.

[34]. **Doumar, A., Ito, H.,** 2001. FPGAs and Fault Tolerance, *The 13th International Conference on Microelectronics, Morocco*, 222-225.

[35]. **Renovell, M., Portal, J. M., Figueras, J. and Zorian, Y.,** 1997. Test Pattern and Test Configuration Generation Methodology for the Logic of RAM-Based FPGA, *ATS'97, Japan*, November 199, 254-259.

[36]. Virtex-II Pro and Virtex-II Pro X FPGA User Guide, http://www.xilinx.com/support/documentation/user_guides/ug012.pdf , 20 Aralık 2010.

[37]. **Öcal, F**., 2006. Güvenli iletişim için FPGA kullanarak şifreleme sistemi tasarımı ve gerçekleştirilmesi, *Yüksek Lisans Tezi*, Gazi Üniversitesi, Fen Bilimleri Enstitüsü, Ankara.

[38]. **Yılmaz, N.,** 2008. Alan programlamalı kapı dizileri (FPGA) üzerinde bir yapay sinir ağları (YSA)'nın tasarlanması ve donanım olarak gerçekleştirilmesi,Selçuk Üniversitesi, *Yüksek lisans tezi,* Fen Bilimleri Enstitüsü, Konya.

[39]. **Sırmaçek, B**., 2007. FPGA ile mobil robot için öğrenme algoritması modellenmesi, *Yüksek Lisans Tezi,* Yıldız Teknik Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

[40]. **Özbey, R. S.,** 2004. Bilgisayar aritmetik ünitelerinin tasarımı için VHDL tabanlı kütüphane geliştirilmesi, *Yüksek Lisans Tezi,* İstanbul Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

[41]. **Douglas, L. P.,** 1991, *VHDL*, McGraw-Hill Inc., California.

[42]. **Enoch O. Hwang, E. O.,** 2006, *Digital logic and microprocessor design with VHDL*,Thomson, California.

EKLER

EK A

System Generator DSP kullanarak birinci derece Sigma-Delta modülatörün FPGA gerçeklenmesi için derlenen VHDL kodları aşağıda verilmiştir.

```
-------------------------------------------------------------------
-- System Generator version 12.3 VHDL source file.
--
-- Copyright(C) 2010 by Xilinx, Inc.  All rights reserved.  This
-- text/file contains proprietary, confidential information of
Xilinx,
-- Inc., is distributed under license from Xilinx, Inc., and may be
used,
-- copied and/or disclosed only pursuant to the terms of a valid
license
-- agreement with Xilinx, Inc.  Xilinx hereby grants you a license to
use
-- this text/file solely for design, simulation, implementation and
-- creation of design files limited to Xilinx devices or
technologies.
-- Use with non-Xilinx devices or technologies is expressly
prohibited
-- and immediately terminates your license unless covered by a
separate
-- agreement.
--
-- Xilinx is providing this design, code, or information "as is"
solely
-- for use in developing programs and solutions for Xilinx devices.
By
-- providing this design, code, or information as one possible
-- implementation of this feature, application or standard, Xilinx is
-- making no representation that this implementation is free from any
-- claims of infringement.  You are responsible for obtaining any
rights
-- you may require for your implementation.  Xilinx expressly
disclaims
-- any warranty whatsoever with respect to the adequacy of the
-- implementation, including but not limited to warranties of
-- merchantability or fitness for a particular purpose.
--
-- Xilinx products are not intended for use in life support
appliances,
-- devices, or systems.  Use in such applications is expressly
prohibited.
--
-- Any modifications that are made to the source code are done at the
user's
-- sole risk and will be unsupported.
--
-- This copyright and support notice must be retained as part of this
-- text at all times.  (c) Copyright 1995-2010 Xilinx, Inc.  All
rights
-- reserved.
```

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

------------------------------------------------------------------
-- System Generator version 12.3 VHDL source file.
--
-- Copyright(C) 2010 by Xilinx, Inc.  All rights reserved.  This
-- text/file contains proprietary, confidential information of
Xilinx,
-- Inc., is distributed under license from Xilinx, Inc., and may be
used,
-- copied and/or disclosed only pursuant to the terms of a valid
license
-- agreement with Xilinx, Inc.  Xilinx hereby grants you a license to
use
-- this text/file solely for design, simulation, implementation and
-- creation of design files limited to Xilinx devices or
technologies.
-- Use with non-Xilinx devices or technologies is expressly
prohibited
-- and immediately terminates your license unless covered by a
separate
-- agreement.
--
-- Xilinx is providing this design, code, or information "as is"
solely
-- for use in developing programs and solutions for Xilinx devices.
By
-- providing this design, code, or information as one possible
-- implementation of this feature, application or standard, Xilinx is
-- making no representation that this implementation is free from any
-- claims of infringement.  You are responsible for obtaining any
rights
-- you may require for your implementation.  Xilinx expressly
disclaims
-- any warranty whatsoever with respect to the adequacy of the
-- implementation, including but not limited to warranties of
-- merchantability or fitness for a particular purpose.
--
-- Xilinx products are not intended for use in life support
appliances,
-- devices, or systems.  Use in such applications is expressly
prohibited.
--
-- Any modifications that are made to the source code are done at the
user's
-- sole risk and will be unsupported.
--
-- This copyright and support notice must be retained as part of this
-- text at all times.  (c) Copyright 1995-2010 Xilinx, Inc.  All
rights
-- reserved.
------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.conv_pkg.all;
-- synopsys translate_off
```

```
library unisim;
use unisim.vcomponents.all;
-- synopsys translate_on
entity xlclockdriver is
  generic (
    period: integer := 2;
    log_2_period: integer := 0;
    pipeline_regs: integer := 5;
    use_bufg: integer := 0
  );
  port (
    sysclk: in std_logic;
    sysclr: in std_logic;
    sysce: in std_logic;
    clk: out std_logic;
    clr: out std_logic;
    ce: out std_logic;
    ce_logic: out std_logic
  );
end xlclockdriver;
architecture behavior of xlclockdriver is
  component bufg
    port (
      i: in std_logic;
      o: out std_logic
    );
  end component;
  component synth_reg_w_init
    generic (
      width: integer;
      init_index: integer;
      init_value: bit_vector;
      latency: integer
    );
    port (
      i: in std_logic_vector(width - 1 downto 0);
      ce: in std_logic;
      clr: in std_logic;
      clk: in std_logic;
      o: out std_logic_vector(width - 1 downto 0)
    );
  end component;
  function size_of_uint(inp: integer; power_of_2: boolean)
    return integer
  is
    constant inp_vec: std_logic_vector(31 downto 0) :=
      integer_to_std_logic_vector(inp,32, xlUnsigned);
    variable result: integer;
  begin
    result := 32;
    for i in 0 to 31 loop
      if inp_vec(i) = '1' then
        result := i;
      end if;
    end loop;
    if power_of_2 then
      return result;
```

```
  else
    return result+1;
  end if;
end;
function is_power_of_2(inp: std_logic_vector)
  return boolean
is
  constant width: integer := inp'length;
  variable vec: std_logic_vector(width - 1 downto 0);
  variable single_bit_set: boolean;
  variable more_than_one_bit_set: boolean;
  variable result: boolean;
begin
  vec := inp;
  single_bit_set := false;
  more_than_one_bit_set := false;
  -- synopsys translate_off
  if (is_XorU(vec)) then
    return false;
  end if;
   -- synopsys translate_on
  if width > 0 then
    for i in 0 to width - 1 loop
      if vec(i) = '1' then
        if single_bit_set then
          more_than_one_bit_set := true;
        end if;
        single_bit_set := true;
      end if;
    end loop;
  end if;
  if (single_bit_set and not(more_than_one_bit_set)) then
    result := true;
  else
    result := false;
  end if;
  return result;
end;
function ce_reg_init_val(index, period : integer)
  return integer
is
   variable result: integer;
 begin
    result := 0;
    if ((index mod period) = 0) then
        result := 1;
    end if;
    return result;
end;
function remaining_pipe_regs(num_pipeline_regs, period : integer)
  return integer
is
   variable factor, result: integer;
begin
    factor := (num_pipeline_regs / period);
    result := num_pipeline_regs - (period * factor) + 1;
    return result;
```

```
  end;

  function sg_min(L, R: INTEGER) return INTEGER is
  begin
      if L < R then
            return L;
      else
            return R;
      end if;
  end;
  constant max_pipeline_regs : integer := 8;
  constant pipe_regs : integer := 5;
  constant num_pipeline_regs : integer := sg_min(pipeline_regs,
max_pipeline_regs);
  constant rem_pipeline_regs : integer :=
remaining_pipe_regs(num_pipeline_regs,period);
  constant period_floor: integer := max(2, period);
  constant power_of_2_counter: boolean :=
    is_power_of_2(integer_to_std_logic_vector(period_floor,32,
xlUnsigned));
  constant cnt_width: integer :=
    size_of_uint(period_floor, power_of_2_counter);
  constant clk_for_ce_pulse_minus1: std_logic_vector(cnt_width - 1
downto 0) :=
    integer_to_std_logic_vector((period_floor - 2),cnt_width,
xlUnsigned);
  constant clk_for_ce_pulse_minus2: std_logic_vector(cnt_width - 1
downto 0) :=
    integer_to_std_logic_vector(max(0,period - 3),cnt_width,
xlUnsigned);
  constant clk_for_ce_pulse_minus_regs: std_logic_vector(cnt_width -
1 downto 0) :=
    integer_to_std_logic_vector(max(0,period -
rem_pipeline_regs),cnt_width, xlUnsigned);
  signal clk_num: unsigned(cnt_width - 1 downto 0) := (others =>
'0');
  signal ce_vec : std_logic_vector(num_pipeline_regs downto 0);
  attribute MAX_FANOUT : string;
  attribute MAX_FANOUT of ce_vec:signal is "REDUCE";
  signal ce_vec_logic : std_logic_vector(num_pipeline_regs downto 0);
  attribute MAX_FANOUT of ce_vec_logic:signal is "REDUCE";
  signal internal_ce: std_logic_vector(0 downto 0);
  signal internal_ce_logic: std_logic_vector(0 downto 0);
  signal cnt_clr, cnt_clr_dly: std_logic_vector (0 downto 0);
begin
  clk <= sysclk;
  clr <= sysclr;
  cntr_gen: process(sysclk)
  begin
    if sysclk'event and sysclk = '1'  then
      if (sysce = '1') then
        if ((cnt_clr_dly(0) = '1') or (sysclr = '1')) then
          clk_num <= (others => '0');
        else
          clk_num <= clk_num + 1;
        end if;
    end if;
```

```
    end if;
  end process;
  clr_gen: process(clk_num, sysclr)
  begin
    if power_of_2_counter then
      cnt_clr(0) <= sysclr;
    else
      if (unsigned_to_std_logic_vector(clk_num) =
clk_for_ce_pulse_minus1
          or sysclr = '1') then
        cnt_clr(0) <= '1';
      else
        cnt_clr(0) <= '0';
      end if;
    end if;
  end process;
  clr_reg: synth_reg_w_init
    generic map (
      width => 1,
      init_index => 0,
      init_value => b"0000",
      latency => 1
    )
    port map (
      i => cnt_clr,
      ce => sysce,
      clr => sysclr,
      clk => sysclk,
      o => cnt_clr_dly
    );
  pipelined_ce : if period > 1 generate
      ce_gen: process(clk_num)
      begin
          if unsigned_to_std_logic_vector(clk_num) =
clk_for_ce_pulse_minus_regs then
              ce_vec(num_pipeline_regs) <= '1';
          else
              ce_vec(num_pipeline_regs) <= '0';
          end if;
      end process;
      ce_pipeline: for index in num_pipeline_regs downto 1 generate
          ce_reg : synth_reg_w_init
              generic map (
                  width => 1,
                  init_index => ce_reg_init_val(index, period),
                  init_value => b"0000",
                  latency => 1
                  )
              port map (
                  i => ce_vec(index downto index),
                  ce => sysce,
                  clr => sysclr,
                  clk => sysclk,
                  o => ce_vec(index-1 downto index-1)
                  );
      end generate;
      internal_ce <= ce_vec(0 downto 0);
```

```
  end generate;
  pipelined_ce_logic: if period > 1 generate
      ce_gen_logic: process(clk_num)
      begin
          if unsigned_to_std_logic_vector(clk_num) =
clk_for_ce_pulse_minus_regs then
              ce_vec_logic(num_pipeline_regs) <= '1';
          else
              ce_vec_logic(num_pipeline_regs) <= '0';
          end if;
      end process;
      ce_logic_pipeline: for index in num_pipeline_regs downto 1
generate
          ce_logic_reg : synth_reg_w_init
              generic map (
                  width => 1,
                  init_index => ce_reg_init_val(index, period),
                  init_value => b"0000",
                  latency => 1
                  )
              port map (
                  i => ce_vec_logic(index downto index),
                  ce => sysce,
                  clr => sysclr,
                  clk => sysclk,
                  o => ce_vec_logic(index-1 downto index-1)
                  );
      end generate;
      internal_ce_logic <= ce_vec_logic(0 downto 0);
  end generate;
  use_bufg_true: if period > 1 and use_bufg = 1 generate
    ce_bufg_inst: bufg
      port map (
        i => internal_ce(0),
        o => ce
      );
    ce_bufg_inst_logic: bufg
      port map (
        i => internal_ce_logic(0),
        o => ce_logic
      );
  end generate;
  use_bufg_false: if period > 1 and (use_bufg = 0) generate
    ce <= internal_ce(0);
    ce_logic <= internal_ce_logic(0);
  end generate;
  generate_system_clk: if period = 1 generate
    ce <= sysce;
    ce_logic <= sysce;
  end generate;
end architecture behavior;
library IEEE;
use IEEE.std_logic_1164.all;
use work.conv_pkg.all;

entity default_clock_driver is
  port (
```

```
    sysce: in std_logic;
    sysce_clr: in std_logic;
    sysclk: in std_logic;
    ce_1: out std_logic;
    clk_1: out std_logic
  );
end default_clock_driver;

architecture structural of default_clock_driver is
  attribute syn_noprune: boolean;
  attribute syn_noprune of structural : architecture is true;
  attribute optimize_primitives: boolean;
  attribute optimize_primitives of structural : architecture is
false;
  attribute dont_touch: boolean;
  attribute dont_touch of structural : architecture is true;

  signal sysce_clr_x0: std_logic;
  signal sysce_x0: std_logic;
  signal sysclk_x0: std_logic;
  signal xlclockdriver_1_ce: std_logic;
  signal xlclockdriver_1_clk: std_logic;

begin
  sysce_x0 <= sysce;
  sysce_clr_x0 <= sysce_clr;
  sysclk_x0 <= sysclk;
  ce_1 <= xlclockdriver_1_ce;
  clk_1 <= xlclockdriver_1_clk;

  xlclockdriver_1: entity work.xlclockdriver
    generic map (
      log_2_period => 1,
      period => 1,
      use_bufg => 0
    )
    port map (
      sysce => sysce_x0,
      sysclk => sysclk_x0,
      sysclr => sysce_clr_x0,
      ce => xlclockdriver_1_ce,
      clk => xlclockdriver_1_clk
    );

end structural;
library IEEE;
use IEEE.std_logic_1164.all;
use work.conv_pkg.all;

entity sdm_filt64_2_cw is
  port (
    ce: in std_logic := '1';
    clk: in std_logic; -- clock period = 10.0 ns (100.0 Mhz)
    gateway_in: in std_logic_vector(23 downto 0);
    gateway_out: out std_logic_vector(23 downto 0);
    gateway_out1: out std_logic_vector(23 downto 0);
    gateway_out_x0: out std_logic_vector(23 downto 0)
```

```
  );
end sdm_filt64_2_cw;

architecture structural of sdm_filt64_2_cw is
  component xlpersistentdff
    port (
      clk: in std_logic;
      d: in std_logic;
      q: out std_logic
    );
  end component;
  attribute syn_black_box: boolean;
  attribute syn_black_box of xlpersistentdff: component is true;
  attribute box_type: string;
  attribute box_type of xlpersistentdff: component is "black_box";
  attribute syn_noprune: boolean;
  attribute optimize_primitives: boolean;
  attribute dont_touch: boolean;
  attribute syn_noprune of xlpersistentdff: component is true;
  attribute optimize_primitives of xlpersistentdff: component is
false;
  attribute dont_touch of xlpersistentdff: component is true;

  signal ce_1_sg_x6: std_logic;
  attribute MAX_FANOUT: string;
  attribute MAX_FANOUT of ce_1_sg_x6: signal is "REDUCE";
  signal clkNet: std_logic;
  signal clk_1_sg_x6: std_logic;
  signal gateway_in_net_x3: std_logic_vector(23 downto 0);
  signal gateway_in_net_x4: std_logic_vector(23 downto 0);
  signal mux1_y_net_x4: std_logic_vector(23 downto 0);
  signal mux1_y_net_x5: std_logic_vector(23 downto 0);
  signal persistentdff_inst_q: std_logic;
  attribute syn_keep: boolean;
  attribute syn_keep of persistentdff_inst_q: signal is true;
  attribute keep: boolean;
  attribute keep of persistentdff_inst_q: signal is true;
  attribute preserve_signal: boolean;
  attribute preserve_signal of persistentdff_inst_q: signal is true;

begin
  clkNet <= clk;
  gateway_in_net_x3 <= gateway_in;
  gateway_out <= mux1_y_net_x4;
  gateway_out1 <= gateway_in_net_x4;
  gateway_out_x0 <= mux1_y_net_x5;

  default_clock_driver_x0: entity work.default_clock_driver
    port map (
      sysce => '1',
      sysce_clr => '0',
      sysclk => clkNet,
      ce_1 => ce_1_sg_x6,
      clk_1 => clk_1_sg_x6
    );

  persistentdff_inst: xlpersistentdff
```

```
    port map (
      clk => clkNet,
      d => persistentdff_inst_q,
      q => persistentdff_inst_q
    );

  sdm_filt64_2_x0: entity work.sdm_filt64_2
    port map (
      ce_1 => ce_1_sg_x6,
      clk_1 => clk_1_sg_x6,
      gateway_in => gateway_in_net_x3,
      gateway_out => mux1_y_net_x4,
      gateway_out1 => gateway_in_net_x4,
      gateway_out_x0 => mux1_y_net_x5
    );

end structural;
```

ÖZGEÇMİŞ

Doğum Tarihi	:	23/02/1983
Doğum Yeri	:	Malatya
Lise	:	Doğanşehir Çok Programlı Lise (1997–2000)
Lisans	:	Fırat Üniversitesi Teknik Eğitim Fakültesi Elektronik ve Bilgisayar Öğretmenliği (2003–2007)
Yüksek Lisans	:	Fırat Üniversitesi Fen Bilimleri Enstitüsü Elektronik ve Bilgisayar Eğitimi Anabilim Dalı (2008-Devam)
İş Tecrübesi	:	Fırat Üniversitesi Araştırma Görevlisi(2009-Devam)