T.C.
İSTANBUL ÜNİVERSİTESİ-CERRAHPAŞA
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

FPGA TABANLI ÇOK KANALLI SAYISALLAŞTIRICI TASARIMI

İDRİS CELİL SİNCANLI

DANIŞMAN

Doç. Dr. Koray GÜRKAN

ELEKTRİK ELEKTRONİK
MÜHENDİSLİĞİ

Elektrik-Elektronik Mühendisliği, Tezli Yüksek Lisans Programı

Mayıs 2024

TEZ KABUL VE ONAYI

İDRİS CELİL SİNCANLI tarafından, Doç. Dr. KORAY GÜRKAN danışmanlığında hazırlanan "FPGA TABANLI ÇOK KANALLI SAYISALLAŞTIRICI TASARIMI" başlıklı bu çalışma, jürimiz tarafından 10/05/2024 tarihinde yapılan sınav sonucunda oy birliği ile başarılı bulunarak Yüksek Lisans Tezi olarak kabul edilmiştir.

Tez Jürisi

		İmza	**Sonuç**
DANIŞMAN	Doç. Dr. Koray GÜRKAN İstanbul Üniversitesi-Cerrahpaşa Elektrik- Elektronik Mühendisliği Anabilim Dalı		☒ Kabul ☐ Ret
ÜYE	Doç. Dr. Erkan ATMACA İstanbul Üniversitesi-Cerrahpaşa Elektrik- Elektronik Mühendisliği Anabilim Dalı		☒ Kabul ☐ Ret
ÜYE	Dr. Öğr. Üyesi Fatih ADIGÜZEL Yıldız TeknikÜniversitesi Kontrol ve Otomasyon MühendisliğiAnabilim Dalı		☒ Kabul ☐ Ret

Aileme ithaf ediyorum…

TEŞEKKÜR

Yüksek lisans eğitimim boyunca değerli desteğini almaktan çekinmediğim, planlanmasından yazılmasına kadar tüm aşamalarında yardımlarını esirgemeyen, beni teşvik eden, aynı titizlikte beni yönlendiren değerli danışmanım Doç. Dr. Koray GÜRKAN'a ve her zaman desteklerini esirgemeden arkamda duran, bana inanan ve her zaman yanımda olan Anneme, Babama ve Kardeşime teşekkürü bir borç bilirim.

Mayıs 2024 İDRİS CELİL SİNCANLI

İÇİNDEKİLER

Sayfa No

ŞEKİL LİSTESİ

Sayfa No

TABLO LİSTESİ

Sayfa No

SİMGE VE KISALTMA LİSTESİ

Kısaltmalar **Açıklama**

FPGA : Field Programmable Gate Array
ADC : Analog-Digital Converter
RAM : Random Access Memory
FIFO : First in First Out
LIFO : Last In First Out
UART : Universal Asynchronous Receiver Transmitter
CLB : Configurable Logic Blocks
LUT : Look Up Table
MUX : Multiplexer

ÖZET

YÜKSEK LİSANS TEZİ

FPGA TABANLI ÇOK KANALLI SAYISALLAŞTIRICI TASARIMI

İDRİS CELİL SİNCANLI

İstanbul Üniversitesi-Cerrahpaşa

Lisansüstü Eğitim Enstitüsü

Elektrik- Elektronik Mühendisliği Anabilim Dalı

Elektrik-Elektronik Mühendisliği, Tezli Yüksek Lisans Programı

Danışman : Doç. Dr. Koray GÜRKAN

Son bir kaç yıldaki teknolijideki atılımlarla birlikte, farklı elektronik donanım sistemlerinde merkezi olmayan bir biçimde çalışan alt sistemlerin kullanımı artmıştır, özellikle yüksek hızlar gerektiren ölçüm işlemleri için ihtiyaç duyulan donanım sistemleri ile görseleştirme ve arayüz sistemleri için daha düşük hızlardaki donanımlar farklılık gösterebilmektedir ve çesitlilik sağlamaktadır. Ancak bu sistemlerde, farklı hızlarda çalışan birimler arasında veri aktarımı önemli bir gerekliliktir. Doğru ve güvenilir bir veri iletimi için farklı hızlardaki birimler arasında bir arabellek kurmak en etkili yol olarak kabul edilmektedir. Bu tezin amacı, FPGA üzerinde farklı saat hızlarında çalışan sayısallaştırıcı ve seri iletişim birimleri arasında en az veri kaybı ile ölçüm yapmak ve haberleşmeyi sağlamak için düşük maliyetli bir sayısal dönüştürücü tasarlamaktır. Tez kapsamında, FPGA üzerinde FIFO arabellek mimarileri incelenmiş ve uygun bir yapı kurulmuştur. Ayrıca hem düşük maliyet gözetilerek hem de ADC hızı yeterli görülen bir donanım seçilmiştir. Bunun sonucunda 8 kanaldan 16 bit ADC ile 0-3.3V arası ölçümler alınmış ve 20 ms periyotlar halinde UART aracılığıyla görselleştirici

cihaza aktarılmıştır. ADC ve UART arasındaki Farklı saat hızlarından kaynaklanabilecek senkronizasyon sorunlarını önlemek amacıyla da FIFO kullanılmıştır.

Mayıs 2024, 67 sayfa.

Anahtar kelimeler: FPGA, ADC, FIFO, UART, Çoklu Kanal ADC

ABSTRACT

M.Sc. THESIS

DESIGN OF FPGA BASED MULTI-CHANNEL DIGITIZER

İDRİS CELİL SİNCANLI

Istanbul University-Cerrahpaşa

Institute of Graduate Studies

Department of Electrical and Electronics Engineering

Electrical and Electronics Engineering M.Sc. Programme

Supervisor : Assoc. Prof. Dr. Koray GÜRKAN

With recent technological advancements over the past few years, the utilization of subsystems operating in a decentralized manner in various electronic hardware systems has increased. Particularly for measurement processes requiring high speeds, hardware systems differ from visualization and interface systems that operate at lower speeds, providing diversity. However, in these systems, data transfer between units operating at different speeds is a crucial requirement. Establishing a buffer between units operating at different speeds is considered the most effective way for accurate and reliable data transmission. The purpose of this thesis is to design a low-cost digital converter on FPGA to perform measurements and facilitate communication with minimal data loss between a digitizer and serial communication units operating at different clock speeds. Within the scope of the thesis, FIFO buffer architectures on FPGA have been examined, and a suitable structure has been established. Additionally, hardware has been selected considering both low cost and sufficient ADC speed. As a result, measurements in the range of 0-3.3V were obtained using an 8-channel 16-bit ADC, and the data were transmitted to a visualization device via UART in 20 ms intervals.

FIFO was also employed to prevent synchronization issues that may arise from different clock speeds between ADC and UART.

May 2024, 67 pages.

Keywords: FPGA, ADC, FIFO, UART, Multi-Channel ADC

1 GİRİŞ

Gömülü sistemler, belirli bir amaca yönelik tasarlanır ve bu amacı düşük güç tüketimiyle en hızlı şekilde gerçekleştirmek üzere optimize edilir. FPGA'ların icadı, bu gömülü sistemlerin tasarım ve test süreçlerini önemli ölçüde kolaylaştırmıştır.

FPGA'ler, 1984 yılında, yarı iletken alanında uzman, Xilinx'in kurucuları Ross ve Vonderschmitt tarafından icat edilmiştir. Freeman ve Vonderschmitt, geleneksel mantık devrelerinin tasarım ve üretiminin karmaşık ve pahalı olduğunu fark ettiler ve bu sorunu çözmek için FPGA'leri geliştirdiler [1-2].

FPGA'lerin icadıyla birlikte, tasarımcılar, devreleri daha hızlı ve daha kolay bir şekilde tasarlayabilir hale geldiler. FPGA'ler, çeşitli uygulamalarda kullanılmaya başlandı, bunların arasında: Telekomünikasyon, elektronik cihazlar, otomotiv, havacılık, askeri alanlar bulunmaktadır.FPGA'lerin icadı, elektronik tasarım endüstrisinde devrim yarattı. FPGA'ler, elektronik cihazların tasarımını ve üretimini daha verimli ve ekonomik hale getirmeye yardımcı oldu. Alan programlanabilir kapı dizisi (FPGA), programlanabilir elektrik devreleri vasıtasıyla bağlanmış, fiziksel olarak yapılandırılabilir mantık blokları (CLB) matrisine dayanan yarı iletken donanımlardır. Sayısal lojik tasarımı geliştiricilerinin gereksinim gördüğü lojik fonksiyonları gerçekleştirmek amacıyla üretilmiştirdir.

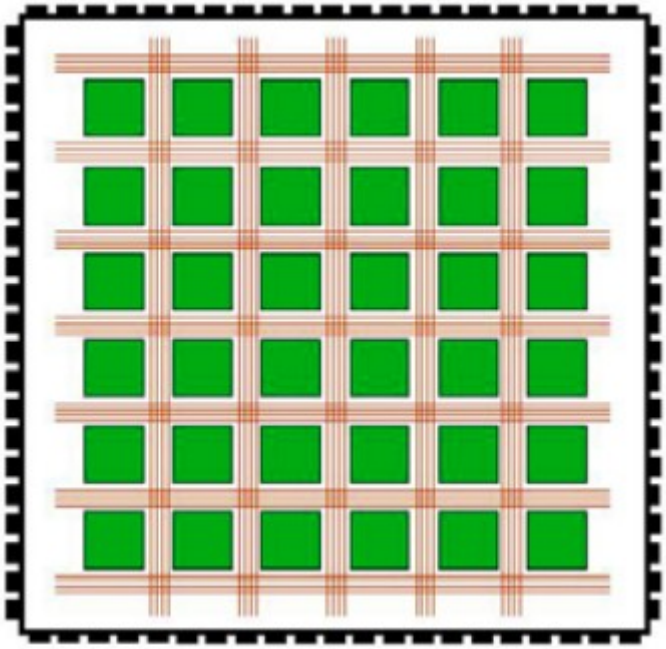
Sayısal Donanım Geliştiricisinin gerek duyduğu mantık kapılarıyla çalışan bir donanım amacına yönelik dizayn edilmiştir. Bu sebebten dolayı her bir mantık fonksiyonu geliştirici eliyle düzenlenebilir haldedir. FPGA ile temek mantık kapılarının ve içeriği dolayısıyla daha kompleks olan devre elemanlarının fonksiyonelliği arttırılmıştır. Alanda programlanabilir ifadesinin verilme sebebi, mantık bloklarının ve sinyal bağlantılarının üretim aşamasından sonra da değiştirilebilir olmasıdır [3].

FPGA'ler, geleneksel mantık devrelerine göre birçok avantaja sahiptir. Bunlar arasında: Esneklik; FPGA'ler, tasarımcının ihtiyaçlarına göre yeniden programlanabilir. Bu, FPGA'leri yeni uygulamalara uyarlamak için ideal hale getirir. Performans; FPGA'ler, geleneksel mantık devrelerine göre daha yüksek performans sağlayabilir. Bu, FPGA'leri yüksek hızlı uygulamalar için

ideal hale getirir. Maliyet; FPGA'ler, geleneksel mantık devrelerine göre daha düşük maliyetli olabilir. Bu, FPGA'leri küçük hacimli üretimler için ideal hale getirir.

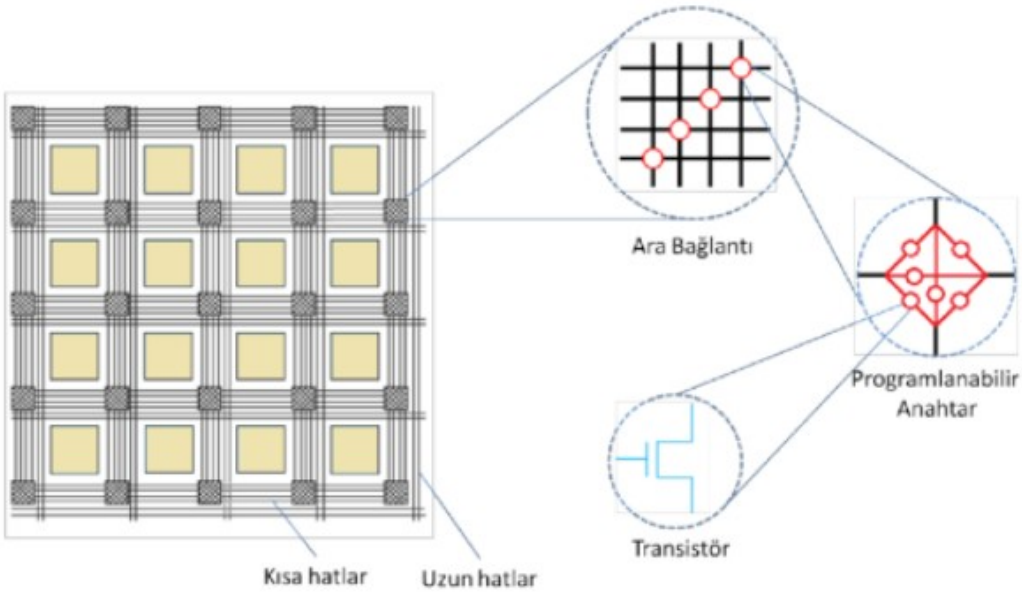
2. KAVRAMSAL ÇERÇEVE

FPGA teknolojisi, üç ana bölümden oluşmaktadır. Bu bölümler; giriş-çıkış birimleri için kullanılan I/O blokları, veri yollarının belirlendiği ve bağlantıların yapıldığı ara bağlantı blokları, ayrıca çarpma DSP, hafıza işlemleri ve saat üretimi gibi yapıları içeren konfigüre edilebilir bloklardan (CLB) oluşmaktadır. I/O blokları, hem giriş hem çıkış işlemleri için kullanılabilmektedir. Konfigüre edilebilir bloklar, çeşitli işlemleri gerçekleştirmek üzere tasarlanmış yapılardan oluşmaktadır. Bu iki temel bileşenin birleşimi, ara bağlantı blokları aracılığıyla gerçekleştirilmektedir. Şekil 2.1.'de gösterilen genel FPGA yapısı, bu üç ana bileşenin etkileşimini göstermektedir.



Şekil 2.1. FPGA Yapısı

FPGA'lerin temelini oluşturan lojik blokları yapı taşları, LUT (Look Up Table), çoklayıcı (MUX) ve Flip-Flop'lardan meydana gelir. Mantık hücrelerinin yapısı, ara bağlantı blokları sayesinde matris mantığıyla gerçekleştirilen veri akışlarını, giriş-çıkış blokları arasındaki bağlantıları içerir. Şekil 2.2.'de, bu veri yollarının bir şeması bulunmaktadır. Bu şemada, kısa hatlar iç hatlarda kullanılırken, uzun hatlar genellikle saat, set/reset gibi global işlemler için kullanılır. Bu hatlar üzerindeki kesişim noktalarındaki transistör yapıları, hangi hattın nasıl bir şekilde birbirine bağlanacağını belirler [4].



Şekil 2.2. FPGA İç Mimarisi [4].

FPGA ile fiziksel tasarımların oluşturulması ve işleyişlerinin ifade edilmesi için kullanılan dil ailesine "HDL (Hardware Description Language)" denir. HDL, donanım tasarımını açıklamak ve tanımlamak için kullanılan bir dil sınıfıdır ve 1987 yılında IEEE tarafından standartlaştırılmıştır. Günümüzde, Verilog ve VHDL en yaygın kullanılan HDL dillerindendir. Bu diller, devre tasarımlarının davranışlarını ifade etmenin yanı sıra test ve simülasyon özellikleri kazandırır. Verilog, donanım tanımlama dilinin bir örneğidir ve FPGA tasarımları, ASIC tasarımları ve dijital sistem tasarımları için yaygın olarak kullanılır.

Verilog, modüler bir yapıya sahiptir ve tasarımların bileşenlere ayrılmasını ve yeniden kullanılmasını sağlar. Hem yapısal hem de davranışsal tasarım yaklaşımlarını destekler ve devrelerin mantıksal davranışlarını açıklamak için kullanılır. Ayrıca, testbench tasarımları için de idealdir, bu sayede tasarımların simülasyon ve doğrulama süreçleri kolaylıkla gerçekleştirilebilir.

FPGA'lerde FIFO (First In, First Out) bellek, ADC (Analog-to-Digital Converter) ve UART (Universal Asynchronous Receiver-Transmitter) modülleri, genellikle veri işleme ve iletişim uygulamalarında önemli roller oynar. Bu üç bileşenin bir araya getirilmesi, geniş bir uygulama yelpazesi içinde hızlı ve güvenilir veri akışını sağlamak için kullanışlıdır.

FIFO bellek, özellikle veri transferinin hızlı ve düzenli bir şekilde gerçekleştirilmesi gereken uygulamalarda önemli bir yer tutar. Genellikle çeşitli veri kaynaklarından gelen verileri depolamak ve bu verileri belirli bir sırayla okumak için kullanılır. FPGA'lerde FIFO bellek, yüksek hızlı veri

akışlarını düzenlemenin yanı sıra farklı saat hızları arasında denge sağlama yeteneği ile öne çıkar [5].

ADC, analog sinyalleri dijital verilere dönüştürerek FPGA içinde işlenebilir hale getirir. FPGA tabanlı ADC'ler, genellikle çok kanallıdır ve birden çok analog girişten gelen veriyi eşzamanlı olarak sayısal verilere dönüştürme yeteneğine sahiptir. Bu sayede çeşitli sensörlerden gelen analog verilerin işlenmesi ve kontrol sistemlerinde kullanılması mümkün olur.

UART ise, FPGA'nın diğer cihazlarla seri iletişim kurmasını sağlayan bir araçtır. Bilgisayarlar, mikrodenetleyiciler veya diğer FPGA'lar gibi harici cihazlarla veri alışverişi için kullanılır. FPGA içindeki veri, UART modülü tarafından seri bir veri akışına dönüştürülerek iletim veya alım işlemleri gerçekleştirilir.

Bu üç bileşenin entegrasyonu, örneğin bir FPGA tabanlı veri toplama ve iletim sistemi tasarımında, sensör verilerinin analog olarak alınması (ADC), bu verilerin FIFO bellekte geçici olarak depolanması ve ardından belirli bir düzenle (örneğin, UART üzerinden) dış dünyaya aktarılması gibi senaryolarda sıkça görülür. Bu tür uygulamalarda, FPGA'lar hızlı ve paralel veri işleme yetenekleri ile öne çıkar ve özelleştirilebilir tasarım esnekliği sayesinde çeşitli uygulamalara uyarlanabilir.

3. YÖNTEM

Ultrason sinyallerini düşük maliyetli bir FPGA tabanlı çok kanallı sayılaştırıcı tasarımıyla görüntüleyebilmek, bu tezin temel amacını oluşturmaktadır. Bu çalışma, öncelikle 8 kanaldan gelen analog sinyal verilerinin yüksek hızlı analog-dijital dönüştürücü (ADC) kullanılarak sayısallaştırılmasını ve Ardışık Alan Programlanabilir Kapı Dizisi (FPGA) modüllerinin pin bağlantı sayıları ve durumları gözetilerek bu verilerin işlenmesini amaçlamaktadır. Tasarım aşamasında dikkate alınan bir diğer önemli faktör ise, yüksek hızda örneklenen ölçüm değerlerinin 50Hz periyotlarla (burst) görüntüleme cihazı üzerinden UART ile paylaşılması ve bilgisayar arayüzünde görselleştirilmesidir.

Bu tezin sınırları, tasarımın düşük maliyetli olması gerekliliğini içermekte ve bu sayede geniş bir kullanıcı kitlesi tarafından kolayca erişilebilir olmalıdır. Ayrıca, tasarımın yüksek hızlı (>1MSPS) örnek toplama kapasitesine sahip olması ve tüm kanallardan gelen ölçüm değerlerini 20 ms içinde UART üzerinden başarıyla iletebilmesi gerekmektedir. Bu sınırlar, tasarımın pratikte kullanılabilirliğini artırmayı hedeflemekte ve ultrason görüntüleme sistemlerine uygun bir çözüm sunmayı amaçlamaktadır.

Bu tez, ultrason sinyallerini düşük maliyetli bir FPGA tabanlı çok kanallı sayılaştırıcı tasarımı aracılığıyla etkili bir şekilde görüntüleme amacını taşımaktadır. Amacın gerçekleştirilmesi için tasarımın detaylı bir şekilde incelenmesi ve performansının değerlendirilmesi, bu tezin ana hedeflerini oluşturmaktadır.

ADC'den veri okuma işlemi genellikle bir miktar zaman alır ve çok kanallı ADC kullanıldığında bu zaman maliyeti daha da artabilir. Bu durum, hızlı ve etkili bir veri aktarımının sağlanması için önemli bir zorluk oluşturabilir. FPGA, bu tür zaman maliyeti sorunlarını çözmek ve düşük maliyetli yüksek hızlı veri aktarımını mümkün kılmak için özellikle avantajlı bir platformdur. FPGA'lar, paralel işleme yetenekleri sayesinde çoklu işlemleri eşzamanlı olarak gerçekleştirebilirler. Bu özellik, çok kanallı ADC'nin neden olduğu zaman maliyetini azaltmada etkili olabilir. FPGA üzerinde paralel veri işleme yeteneği, her kanaldan gelen verilerin eşzamanlı olarak işlenmesini ve daha hızlı bir şekilde toplanmasını sağlayarak verimliliği artırabilir.

Ayrıca, FPGA'lar genellikle özel olarak tasarlanmış donanım modüllerine sahiptir, bu da özel veri işleme görevlerini gerçekleştirmelerini sağlar. Bu sayede, FPGA üzerindeki özel tasarım, veri işleme süreçlerini daha hızlı veoptimize edilmiş bir şekilde gerçekleştirebilir. Bu tasarım özellikleri, ADC'den gelen verilerin hızlı bir şekilde işlenmesini ve daha sonra UART aracılığıyla bilgisayarla

iletilmesini sağlamak için kullanılabilir. Bu bağlamda, FPGA'nın çok kanallı ADC'lerle çalışırken zaman maliyetini düşürme ve düşük maliyetli yüksek hızlı veri aktarımını sağlama yetenekleri, bu tezin anahtar motivasyonunu oluşturmaktadır.

Şekil 3.1. FPGA ile ADC veri okuma ve aktarma

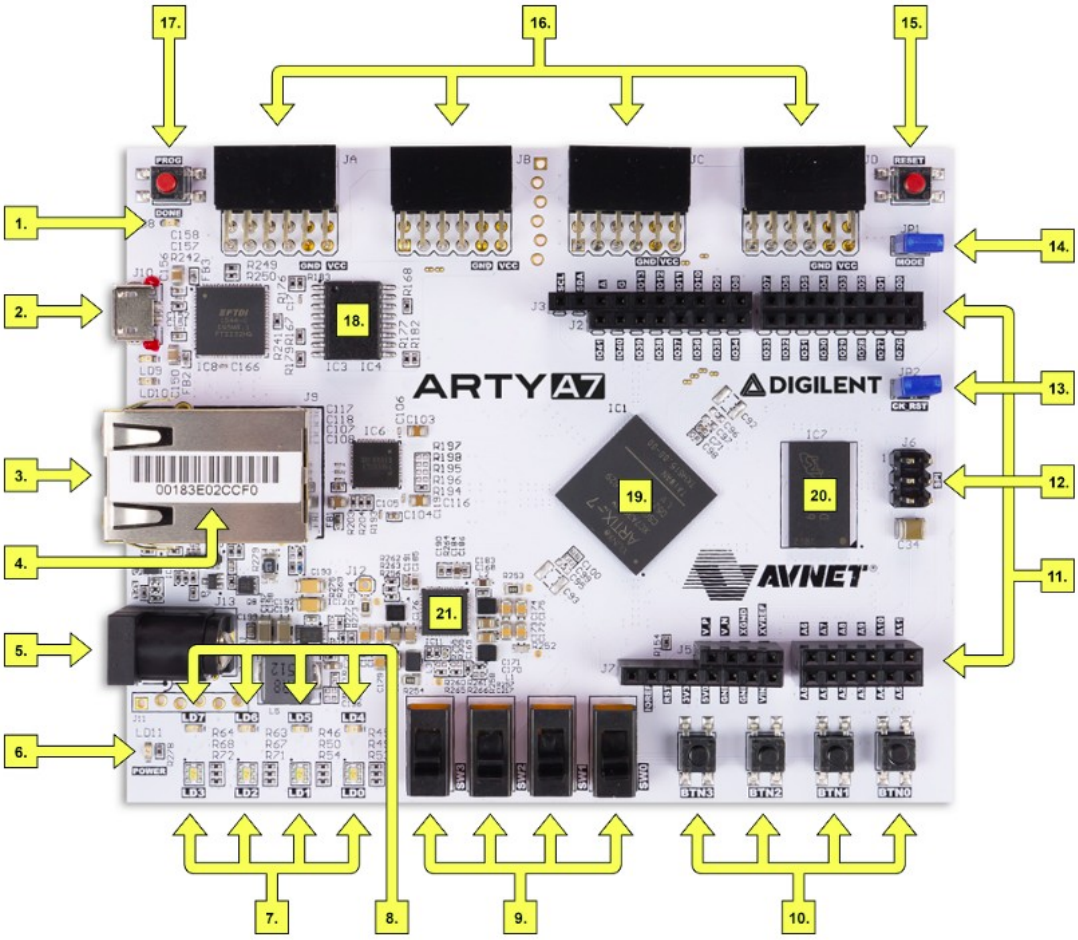
3.1. SAYISAL TASARIM TEST AYGITI: FPGA

3.1.1 SİSTEM DONANIM ALTYAPISI

Gerçek zamanlı simüle edebilmek için Arty A7 35t serisi FPGA barındıran Artix-7 geliştirme kartı ve içinde barındırdığı XADC modülü kullanılmıştır. Analog giriş ve çıkışları yapabilmek için bir arayüz kartı daha kullanılıp, seviye ayarlamalarını yapabilmek için sinyal üretici kullanılmıştır. Bu ekipmanlar dışında bilgisayar bağlantısı yapmak ve sinyal uygulamak için bir adet USB izalatör ünitesi de kullanılmıştır.

3.1.2 Artix-7 FPGA Geliştirme Kartı

Sistemimizin merkezinde bulunan ve FPGA yongasını barındıran Artix-7 geliştirme kartı, üzerinde intel (Xilinx) Arty A7 35t SoC serisi bir FPGA bulunmaktadır. Xilinx serisi FPGA yongaları donanım geliştirmede düşük güç gereksinimi ve bütçe açısından kısıtlı uygulamalarda kullanılan bir yonga ailesidir. SoC türünde imal edilenlerde hem programlanabilir mantık blokları hem de HPS (Hard Processor System) olarak adlandırılan sabit donanımsal RISC tabanlı işlemci bulunmaktadır.



Şekil 3.2. Artix-7 FPGA Geliştirme Kartı ön yüzü

Arty A7, önceden Arty olarak bilinen bir geliştirme platformudur ve Xilinx'in Artix-7™ FPGA'sı etrafında tasarlanmıştır. Özellikle MicroBlaze Soft İşlem Sistemi olarak kullanılmak üzere tasarlanan Arty A7, projenizin ihtiyaçlarına adapte olabilen son derece esnek bir işlem platformudur. Diğer Tek Kart Bilgisayarların aksine, Arty A7, tek bir set işlem periferine bağlı değildir. Bir anda UART'lar, SPI'lar, IIC'ler ve bir Ethernet MAC ile dolu bir iletişim gücüne sahip olabilir, bir sonraki anda ise on iki adet 32-bit zamanlayıcıya sahip bir zaman tutucu olabilir.

Arty A7'nin temel özellikleri:

• Xilinx Artix-7 FPGA

• 5200 dilim içeren dört 6 girişli LUT ve 8 flip-flop (15850 dilim)

• 1800 Kbit hızlı blok RAM (4860 Kbit)

• 5 saat yönetimi bloğu (CMT) içeren, her biri bir faz kilidi ve karışık mod saat yöneticisi bulunan 6 CMT

• 90 DSP dilimi (240 DSP dilimi)

• Dahili 450MHz'den yüksek saat hızlarına sahip analogdan dijitale çevirici (XADC)

• JTAG ve Quad-SPI Flash üzerinden programlanabilir

• 256MB DDR3L, 16-bit veriyolu @ 333MHz

• 16MB Quad-SPI Flash

• USB-JTAG Programlama devresi

• USB veya herhangi bir 7V-15V kaynaktan beslenebilir

• 10/100 Mbps Ethernet ve USB-UART Köprüsü gibi sistem bağlantı özellikleri

• 4 anahtar, 4 düğme, 1 sıfırlama düğmesi, 4 LED ve 4 RGB LED gibi etkileşim ve algılama cihazları

• 4 Pmod konektörü ve Arduino/ChipKit Shield konektörü gibi genişleme bağlantı noktaları bulunur.

Şekil 3.2.'de gösterilen Artix FPGA geliştirme kartında bulunan komponentler Tablo 3.1.'deki gibidir.

Numara	**Tanım**	**Numara**	**Tanım**	**Numara**	**Tanım**
1	FPGA programalama tamamlandı LED'i	8	Kullanıcı Ledleri	15	ChipKIT işlemci resset tuşu
2	Paylaşımlı USB JTAG/UART portu	9	Kullanıcı kızaklı switchleri	16	Pmod bağlantıları
3	Ethernet bağlantısı	10	Kullanıcı butonları	17	FPGA programlama reset butonu
4	MAC adres bilgisi	11	Kullanıcı butonları	18	FPGA programlama reset butonu

5	Opsiyonel güç kaynağı yuvası	12	Arduino/ chipK IT bağlantısı	19	SPI flaş hafıza
6	Güç var LED’i	13	Arduino/ chipK IT SPI bağlantısı	20	Artix FPGA
7	Kullanıcı RGB LEDleri	14	ChipKIT işlemcisi reset jumperı	21	Micron DDR3 hafıza

Tablo 3.1. Artix Geliştirme Kartı elemanları

Arty A7, bank 35'teki MRCC girişine bağlı olan tek bir 100 MHz kristal osilatör içermektedir. Giriş saati, tasarımın çeşitli frekanslarda ve bilinen faz ilişkilerinde saatlere ihtiyaç duyduğu MMCMs veya PLL'leri sürmek için kullanılabilir. Ancak, bazı kurallar 100 MHz'lik giriş saatinin hangi MMCMs ve PLL'leri sürmesine izin verir. Bu kuralların tam açıklaması ve Artix-7 saat kaynaklarının yetenekleri için, Xilinx'in "7 Serisi FPGA Saat Kaynakları Kullanıcı Kılavuzu"na başvurulmalıdır [6].

3.2 FIFO Yapısı

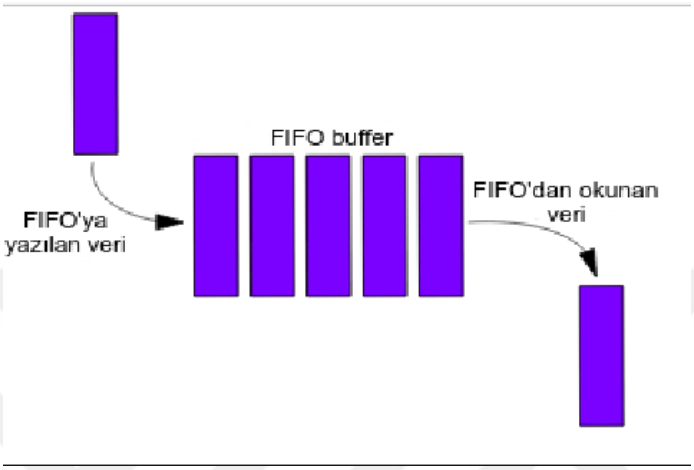
Bildiğimiz gibi, veri transferi günümüz dijital sistemlerinin çoğunda sıkça kullanılmaktadır. Sistemimizin hızından daha yüksek bir hızda veri transferi durumunda, veri kaybı gibi bazı sorunlar ortaya çıkar. Ayrıca, sistem veriyi düzensiz bir şekilde işler veya toplu veri transferi benzer sorunlara neden olabilir. Bu durumda, bir tampon ve depolama oluşturmak gereklidir. Günlük yaşamımızda benzer yaklaşımları gözlemlemekteyiz.

Örneğin posta dağıtım merkezinde çalışan posta görevlileriyle karşılaşabiliriz. Bu görevliler, gelen posta ve paketleri düzenli bir şekilde sıralamak ve hedef adreslere ulaştırmakla sorumludur. Ancak, postaneye gelen gönderilerin miktarı ve boyutu her gün değişiklik gösterir.

İşte bu noktada, posta görevlileri gelen gönderileri düzenli bir şekilde işleyebilmek için bir birim oluştururlar. Bu birim, postaneye gelen büyük bir yığılmayı absorbe eder ve dağıtım sürecini düzenler. Gönderiler, sıralama yapıldıktan sonra sırasıyla hedef adreslere ulaştırılır. Bu sistem sayesinde, posta dağıtımında düzensizlikleri önlemek ve gönderilerin kaybolmasını engellemek

mümkün olur. Benzer bir şekilde, bu düzenleme sayesinde gelen müşteriler,posta görevlilerinden aldıkları takip numarası gibi bir referansla sıra bekler ve bu şekilde ilk gelen ilk hizmet alır.

FIFO, özel bir buffer türüdür. FIFO, ilk giren ilk çıkar anlamına gelir. Bu nedenle buffera yazılan veri, Şekil 3.3'te gösterildiği gibi ilk önce bufferdan çıkar,. Diğer bir buffer türü ise paylaşılan veya yığın bellek, başka diğer bir deyişle LIFO yapısıdır. LIFO son giren ilk çıkar anlamına gelir, yani buffera en son yazılan veri bufferdan ilk çıkar. Hangi tür bufferın seçilmesi uygulamaya bağlıdır.



Şekil 3.3. FIFO buffer kuyruk yapısı

Yazılım veya donanım tabanlı bir FIFO oluşturmak mümkündür. Uygulamaya ve istenilen özelliklere bağlı olarak FIFO'nun yazılım veya donanım tabanlı yapısı seçilir. Yazılım FIFO'nun esnek yapısı sayesinde FIFO'yu değiştirmek oldukça kolaydır. Bunun için sadece kodu gözden geçirmek yeterlidir. Ancak donanım tabanlı bir yapıda, yeni bir kart düzeni oluşturmak gereklidir. Donanım FIFO'nun avantajı, yazılım FIFO'sundan daha yüksek bir hızda çalışabilme yeteneğidir.

3.2.1. FIFO Tipleri

FIFO, yani ilk giren ilk çıkar prensibine dayanan bir veri yapısıdır. FIFO buffer, bu prensibi takip eden bir veri depolama mekanizmasıdır. FIFO buffer'ların farklı tipleri olabilir, ve bunlar genellikle kullanım amaçlarına ve özelliklerine bağlı olarak değişir.

Dizi Tabanlı FIFO Buffer: Bu tip FIFO buffer, genellikle bir diziyi temel alır ve bu dizide verileri depolar. Elemanlar sırayla eklenir ve çıkarılır, bu nedenle FIFO prensibi korunur. Dizi tabanlı FIFO buffer'lar genellikle basit ve hızlıdır.

Bağlı Liste Tabanlı FIFO Buffer: Bağlı liste, düğümlerden oluşan bir veri yapısıdır. FIFO buffer için kullanıldığında, her düğüm bir veriyi temsil eder. Bağlı liste tabanlı FIFO buffer, verileri eklemek ve çıkarmak için düğümleri kullanır.

Halka (Ring) Buffer: Halka buffer, bellekte bir dizi gibi davranan bir veri yapısıdır.Buffer'ın başı ve sonu birbirine bağlıdır, bu da buffer'ın dolu olduğunda yeni verilerin eklenemeyeceği ve boş olduğunda verilerin çıkarılamayacağı anlamına gelir.Bu özellik, döngüsel bir yapıya sahip olduğu için "halka buffer" olarak adlandırılır.

Önceden Belirlenmiş Boyutlu FIFO Buffer: Bu tip FIFO buffer, bir maksimum kapasiteye sahip olan ve bu kapasiteyi aşan veri eklemelerini önleyen bir yapıya sahiptir. Veri eklemesi yapıldığında, buffer doluysa en eski veri otomatik olarak çıkarılır.

Paket Tabanlı FIFO Buffer: Bu tip FIFO buffer, belirli boyutlardaki veri paketlerini depolar. Her bir paket, FIFO prensibine göre eklenir ve çıkarılır. Özellikle iletişim protokollerinde ve ağ uygulamalarında kullanılabilir.

Bunlar FIFO buffer'ların genel tipleridir ve belirli kullanım senaryolarına bağlı olarak daha spesifik implementasyonlar ortaya çıkabilir.

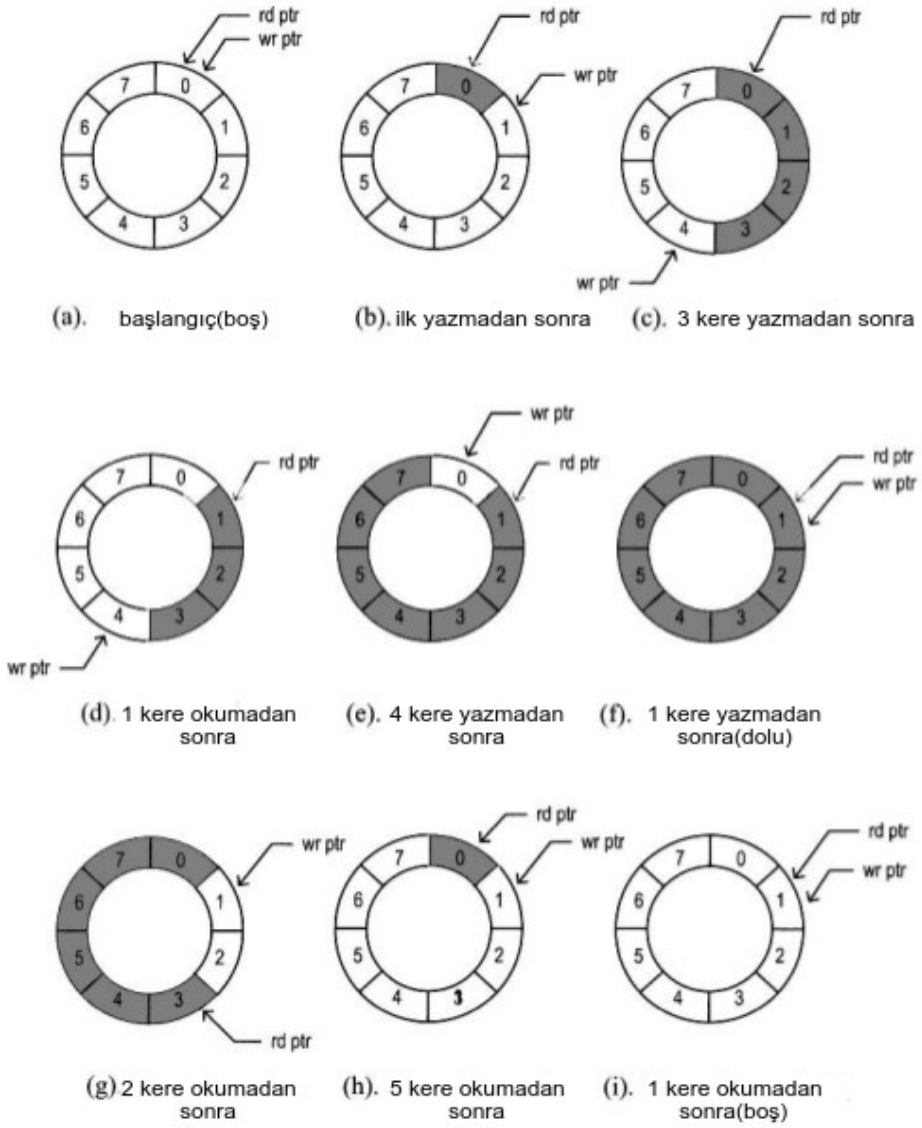
3.2.2. Döngüsel FIFO buffer

Döngüsel (halka) buffer ve standart (sıralı) buffer, veri depolama yapıları arasında farklılıklar gösterir. Standart bufferlar belirli bir kapasiteyle sınırlıdır ve kapasite dolduğunda yeni veri eklenemez, bu da potansiyel veri kaybına yol açabilir. Döngüsel bufferlar ise kapasite dolu olduğunda en eski veriyi otomatik olarak üstüne yazarak sürekli bir veri akışını sürdürür. Standart bufferlar genellikle belirli bir sıra içinde veri depolamak için kullanılırken, döngüsel bufferlar sürekli ve döngüsel veri akışlarını daha etkili bir şekilde işlemek için tercih edilebilir. Her iki yapı da özellikle belirli uygulama senaryolarına bağlı olarak avantajlar sunar.

Gömülü donanım yapılarında sıkça görülen statik veri yapılarından bir tanesi Döngüsel (Ring) buffer’ dir. Döngüsel buffer devamlı olan (continuous), statik boyutlu, çevrimli (circular) ve FIFO özelliğine sahip yaygın bir veri yapısıdır. Çoğunlukla sabit boyutlu bir kuyruk (queue) şeklinde kullanılır. Sabit boyutlu bir sıra dizisi alternatif olarak yeri geldiğinde boyutları değişen yükseltilip düşürülebilen bir kuyruk dizayn da yapılabilmektedir. Fakat gerçek zamanlı gömülü donanımlar

kapsamında dinamik bellek yönetimi donanımın çalışma zamanı istikrarına müdahale etmemek için çoğunlukla sadece başlatma esnasında tercih edilebilmektedir.

Döngüsel buffer tasarımında belirgin olarak yazma (head) ve okuma (tail) denilen iki işaretçi vardır. Kuyruk işaretçisi veri eklemede yani okuma işlemine bağlı haraket yapılmak istendiğinde değeri değişir. Baş işaretçi ise veri okunduğunda değeri değişir. Bunun sayesinde bu iki işaretçi vesilesiyle döngüsel buffer'in dolu ya da atıl olup olmadığını ve aynı zamanda da döngüsel buffer içerisinde ne boyutta atıl alan kaldığı gibi bilgileri edinebilmemiz mümkün olabilmektedir.



Şekil 3.4. Döngüsel FIFO buffer yapısı

Döngüsel buffer tasarımında belirli bir zamanda dögüsel dizinin dolu ya da atıl olup olmadığını anlamak önemlidir. Şayet yazma ve okuma işaretçileri denk ise buffer'ın atıl kaldığı söylenebilir. Bu şart hali hazırda döngüsel buffer'i başlattıgımızda iki işaretçiyi de 0 değeri atayarak sağlanmış olunur. Ancak daha sonrasında veri yazma ve okuma yapıldığında bu şartı sağlamak amacıyla dizinin bir elemanı atıl şekilde bırakılmalıdır. Bunun sebebi ise buffer tamamıyla dolu olduğunda yine de yazma ve okuma işaretcileri ortak kutucuğa gelebileceği için buffer'in dolu mu yoksa boş mu olduğunu ayırt etmenin olanaksız olmasıdır.

Gerçekleşebilmesini sağlamak istendiğinde de daha karmaşık bir dizayn yapmak gerekecektir. Bu sebepten dolayı pek çok döngüsel buffer dizaynında sıralı dizin içerisinde bir eleman boş bırakılır ve buraya veri yazılmaz. Eğer okuma işaretcisi yeniden yazma işaretcisinin bulunduğu alan ile aynı noktaya geliyor ise döngüsel buffer tamamen dolmuş demektir. Özetle döngüsel buffer'in dolu ya atıl olduğunu anlayabilmek için şu şartlar gözden geçirilelidir:

• Yazma ve okuma işaretçisi denk olduğunda döngüsel buffer boştur.

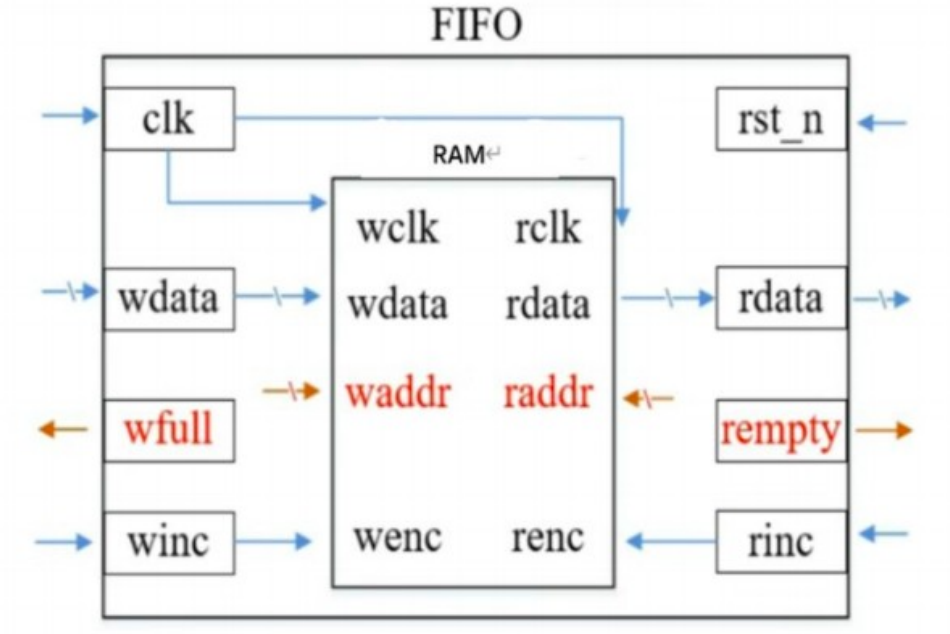
• Okuma işaretcisi ve yazma işaretcisi bir dahaki sefere eşit olacaksa döngüsel buffer dolmuştur.

Eğer döngüsel buffer'a yeni bir eleman eklenecekse, öncelikle buffer'da yer olup olmadığından emin olunması gerekmektedir. Eğer dolu ise farklı tasarıma göre eski verinin yerine atanabilecek veya tasarımcıya bir biçimde bunu bildirebilecek işlemlerin yapılması gerekebilir. Benzer bir ifadeyle, buffer boş olduğunda da okuma işlemi yapıldığında buffer'da veri kalmadığının bildirilmesi gerekmektedir.

Bir FIFO tasarlamak farklı saat darbeleriyle çalışan alanlar arasında veri transfer etmenin en yaygın tekniklerinden biridir [7]. FIFO buffer yapısı, veriyi paralel veya seri iletişimde transfer etmek için kullanılabilir [3]. Veriyi bir saat sinyali ile buffera depolamak ve başka bir saat sinyali ile veriyi bufferdan almak, farklı saat alanları arasında veri transferini gerçekleştirmek için ideal ve kolay bir yöntem gibi görünmektedir. Ancak, boş ve dolu durum kontrol devresini tasarlamak zorlayıcı olabilir [7]. Asenkron ve senkron FIFO'lar, veri iletimi ve işleme yöntemleri açısından farklılık gösterir. İki tür de FIFO prensibini takip eder, ancak verilerin nasıl eklenip çıkarıldığı ve senkronizasyon ile ilgili olan farklılıkta ortaya çıkar.

3.2.3. Senkron FIFO buffer

Senkron FIFO'lar, veri transferinin bir saat sinyali (clock) tarafından kontrol edildiği sistemlerde kullanılır. Bu tür FIFO'lar, her veri transferinin bir saat darbesi tarafından senkronize edildiği bir ortamda çalışır. Senkron FIFO'lar, belirli bir saat periyoduna göre düzenlenmiş, eşzamanlı veri transferi sağlar [8]. Örnek olarak, bir mikroişlemci veya FPGA üzerinde çalışan senkron FIFO, bir dijital veri işleme uygulamasında kullanılabilir. Örneğin, bir ses işleme uygulamasında ses örnekleri düzenli aralıklarla bir veri yolu üzerinden bir işlem birimine aktarılabilir. Tipik bir senkron FIFO üç ana bölümden oluşur: (1) FIFO yazma kontrol mantığı, (2) FIFO okuma kontrol mantığı ve (3) FIFO depolama birimleri. Senkron FIFO'nun ana gövdesi, çift portlu bir RAM parçasından oluşur. RAM'ın okuma ve yazma saatleri (wclk, rclk), FIFO tarafından sağlanır. İkinci olarak, FIFO'nun yazma verisi (wdata) doğrudan RAM'a iletilir ve okuma verisi doğrudan RAM'dan okunur (rdata). FIFO'nun yazma etkinleştirme sinyali (winc), RAM'ın yazma etkinleştirme sinyali (wenc) ile aynıdır ve FIFO'nun okuma etkinleştirme sinyali (rinc), RAM'ın okuma etkinleştirme sinyali (renc) ile aynıdır. Son olarak, doluluk mantığı (wfull) ve boşluk mantığı (rempty) bulunmaktadır.



Şekil 3.5. Senkron FIFO yapısı [5]

3.3. Evrensel Asenkron Alıcı/Verici (Universal Asynchronous Receiver Transmitter)

Universal Asenkron Alıcı/Verici (UART), seri iletişimde yaygın olarak kullanılan bir iletişim protokolüdür. Kısa mesafe iletişimi sağlar ve istikrar, güvenilirlik ve kullanım kolaylığı açısından birçok avantaja sahiptir. UART kullanılarak iletişim, bir pin üzerinden (yarı çift yönlü) veya iki pin üzerinden (tam çift yönlü) gerçekleşebilir. Bu protokol asenkron bir yapıya sahiptir ve iletişim için herhangi bir senkronize saat sinyali gerektirmez. UART protokolünde iletişim, Baud Oranı olarak bilinen önceden belirlenmiş bir hızda gerçekleşir. UART'ta senkronize saat sinyali bulunmadığından, hata olmadan iletişim kurabilmek için veri bitleriyle birlikte UART çerçevesine ekstra bitler eklenir [9].

UART modülünün mimarisi üç ana bloktan oluşur: Baud Oranı Oluşturucu, Verici modülü ve Alıcı modülü. Verici modülü, paralel veriyi seri veriye dönüştürür ve ardından çıkış pininden seri olarak iletim sağlar. Alıcı modülün görevi ise cihaz tarafından belirlenen bir hızda seri veri bitlerini almak olarak tanımlanmıştır [10].

Alıcı modü UART veri paketine eklenen ekstra bitleri kaldırır ve paketten 8 bit (değişkenlik gösterebilir) anlamlı veri sağlar. UART iletişim protokolünde, bir UART paketinde genellikle 1 Başlangıç biti, 5-8 Veri biti, 1 Teklik (parity) biti (isteğe bağlı) ve 1-2 Durma biti içerir. Bu tasarımda önerilen UART modülü, UART paketinde 1 Başlangıç biti, 8 Veri biti ve 1 Durma biti kullanır (Şekil 3.6.'da gösterildiği gibi). Veri bitleri Amerikan Bilgi Değişim Standart Kodu (ASCII) formatında ters sırayla iletilir. UART iletişim protokolü, bilgisayar ile FPGA arasında tam çift yönlü iletişim için kullanılabilir.

Şekil 3.6. UART Veri Paketi

Baud hızı, bit başına ölçülen birimlerle (bps) ifade edilir ve bir bitin gönderilip alınma süresini belirler. Baud hızı parametresi 300, 1200, 2400, 4800, 9600, 19200, 115200 gibi standart değerler olabilir. Ayrıca günümüzde donanımların desteklemesi halinde 1 Mbps 2 Mbps ya da 3Mbps değerlerinde veri iletişimi de mümkündür. Gönderen ve alıcıdaki baud hızı aynı olmalıdır. Böylece gönderilen veri alıcı tarafında doğru bir şekilde alınabilir. Bazı cihazlar baud hızını otomatik olarak

belirleyebilir. Hız, UART paketindeki diğer bitlerini de içerdiği için etkili veri hızı, bit hızından daha düşüktür. Örneğin, 8N1 formunda, sadece bitlerin %80'i veri için kullanılabilir.

Veri bitlerinin sayısı 5, 6, 7, 8 (en yaygın) veya 9 olabilir. Daha yeni uygulamalarda genellikle 8 veri biti neredeyse evrensel olarak kullanılır. 5 veya 7 bit, genellikle daha eski cihazlarla çalışır. Çoğu seri iletişim tasarımı, bitleri byte halinde gönderir, en az anlamlı bitin (LSB) ilk gönderilen olur. Bu standart aynı zamanda "little endian" olarak adlandırılır. Nadiren kullanılan bir "big endian" standartı da vardır, başka bir değişle en anlamlı bit (MSB) ilk gönderilendir.

Parity, iletim sırasında meydana gelen hataları tespit etmek için kullanılır. Veri bitleri ile birlikte ek bir bit gönderilir. Bu bit, her karakterdeki 1 bit sayısının her zaman tek veya her zaman çift olmasını sağlamak üzere ayarlanmıştır. Eğer parity yanlışsa, bu byte'ın bozuk olduğu anlamına gelir. Eğer parity doğruysa, hata yoktur veya çift sayıda hata vardır. Her bitin XOR (özel veya) işlemine tabi tutularak elde edilir. Sonuç 1 ise tek parity, sonuç 0 ise çift parity anlamına gelir. Çift parity, verinin tüm bitleri üzerinde XOR işleminden sonra işlem yapmadan hesaplanır. Tek parity, verinin tüm bitleri üzerinde XOR işlemi gerçekleştirilerek daha sık kullanılır. Bunun nedeni bir durum geçisi meydana gelir ve böylece daha güveniilir hale gelmiş olmasıdır.

Her iletilen mesaj byte'ın sonunda alıcıyı tekrar senkronize edebilmek için durma bitleri gönderilmelidir. Haberleşme cihazlarında çoğunluka bir adet kullanılır. Nadiren, düşük saat hızlı cihazlarda bir buçuk veya iki durma bitine ihtiyaç duyulabilir.

3.4. Analog-Digital Dönüştürücü (ADC)

ADC (Analog-Digital Converter - Analog-Dijital Dönüştürücü), analog sinyalleri dijital verilere dönüştüren bir elektronik bileşendir. Bu dönüştürücü, gerçek dünyadaki sürekli analog sinyalleri, mikrodenetleyiciler, bilgisayarlar veya diğer dijital sistemler tarafından işlenebilecek dijital formatlara çevirir.

ADC'nin temel görevi, sensörlerden veya analog sinyal kaynaklarından gelen analog voltaj veya akım gibi sürekli sinyalleri, dijital bilgisayarlar veya mikrodenetleyiciler tarafından anlaşılabilir dijital formata çevirmektir. Bu, analog sinyallerin ölçüm, kontrol veya diğer dijital işlemler için kullanılmasını sağlar.

ADC'ler, çeşitli uygulamalarda yaygın olarak kullanılır, örneğin ses işleme, sensör verilerinin dijital kontrol sistemlerine dönüştürülmesi, tıbbi cihazlar, iletişim ekipmanları ve daha birçok alanda. ADC'ler, analog sinyallerin dijital dünyada işlenebilmesini ve kullanılabilmesini mümkün kılar.

Bu teze konu olan FPGA için özel tasarlanmış olan XADC kullanılmmıştır. Xilinx'in Analog-to-Digital Converter (Analog-Dijital Çevirici) olarak kısaltılan XADC'si, özellikle Xilinx FPGA (Field Programmable Gate Array, Alanda Programlanabilir Kapı Dizisi) cihazları için tasarlanmış bir entegre birimin adıdır. Bu entegre birim, FPGA içerisinde bulunan analog sinyalleri dijital verilere dönüştürme görevini üstlenir. XADC, genellikle FPGA projelerinde analog sensör verilerini işlemek veya çeşitli ölçümleri dijital bir ortamda kullanmak için kullanılır. Arty A7'nin Xilinx XADC'si (Analog-to-Digital Converter), birden fazla giriş kanalına sahip olup çeşitli sensörleri ve analog sinyal kaynaklarını takip etme imkanı sunar. 12-bit çözünürlük ile yüksek hassasiyetli ölçümler gerçekleştirebilir. Ayrıca, entegre sıcaklık sensörü sayesinde FPGA'nın çalışma sıcaklığını ölçebilir ve dahili referans gerilimi ile ölçümler için bir referans noktası sağlar. Arty A7 XADC, programlanabilir çalışma modları ve geniş ölçüm yetenekleri ile kullanıcılara esneklik sunar. Bu özellikleri sayesinde, FPGA projelerinde analog verilerin işlenmesi ve ölçülmesi için ideal bir çözüm sunar.

Bağlı Pin	XADC Kanalı	Kanal Tipi
A0	Kanal 4	Tek uçlu
A1	Kanal 5	Tek uçlu
A2	Kanal 6	Tek uçlu
A3	Kanal 7	Tek uçlu

A4	Kanal 15	Tek uçlu
A5	Kanal 0	Tek uçlu
A6(+)/A7(-)	Kanal 12	Diferansiyel
A8(+)/A9(-)	Kanal 13	Diferansiyel
A10(+)/A11(-)	Kanal 14	Diferansiyel

Tablo 3.2. FPGA XADC Kanal Bağlantıları [10]

3.5 UART Modülün Tasarımı

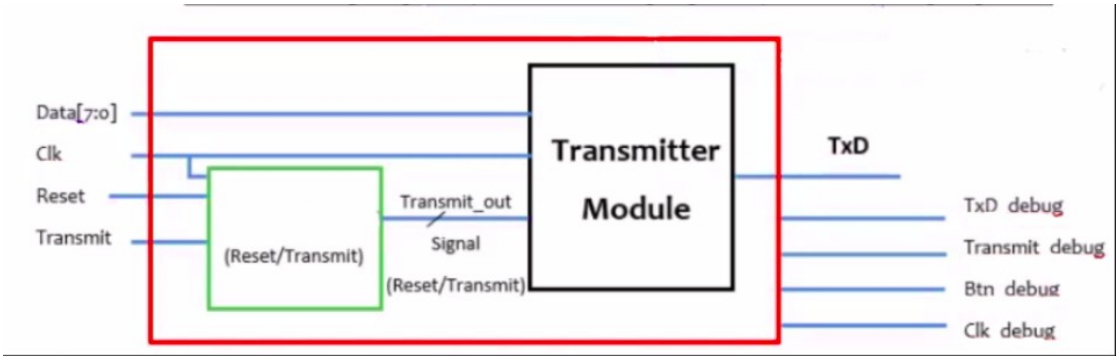
Bu modül, FPGA tabanlı sistemlerde kullanılmak üzere tasarlanmış bir UART verici modülü oluşturan detaylı bir tasarımı içermektedir. Modül, temelde dört ana bölümden oluşmaktadır. İlk olarak, modülün giriş ve çıkışlarını tanımlayan bir bölüm bulunmaktadır. Bu tanımlamalar arasında, UART iletişimini kontrol eden bir saat sinyali (clk), sistem sıfırlamasını tetikleyen bir sıfırlama sinyali (reset), UART iletişimini başlatan bir sinyal (transmit), iletilen 8-bit veriyi içeren bir veri girişi (data), ve son olarak seri iletişim çıkışını sağlayan bir TxD çıkışı yer almaktadır.

İkinci bölüm, içsel değişkenleri ve UART iletim mantığını içermektedir. Bu kısımda, baud hızını hesaplamak için bir sayıcı kullanılır. Belirli bir sayıda saat darbesi geçtikten sonra, durumlar arasında geçiş yaparak veri iletimini gerçekleştirir. İletim sırasında, sağa kaydırma kaydındaki veri bitleri sağa kaydırılarak seri iletim yapılır.

Üçüncü bölüm, bir durum makinesi kullanarak iletimin kontrolünü sağlar. İletim başlatıldığında durumlar arasında geçiş yapar, bit sayısını takip eder ve iletim tamamlandığında gerekli sıfırlamaları gerçekleştirir. Bu sayede, her bir iletimin başında ve sonunda durumları yöneterek veri iletimini düzenler.

Son olarak, kodun dördüncü bölümü, modülün genel kontrolünü sağlamak üzere bir ana durum makinesi içermektedir. İletim başlatıldığında ve tamamlandığında durumları günceller, gerekli sinyalleri kontrol eder ve iletim sürecini yönetir.

Bu tasarım, mikrodenetleyici sistemlerinde seri iletişim gereksinimlerini karşılamak üzere özenle oluşturulmuş bir UART verici modülü sunmaktadır. Kodun modüler ve açıklamalı yapısı, bu modülün anlaşılabilirliğini artırarak entegrasyonu kolaylaştırmaktadır. Baud oranı 2 Mbps olan bir UART kullanılmak için ekler bölümünde de gösterildiği gibi counter değeri 50’ye ayarlanmıştır.



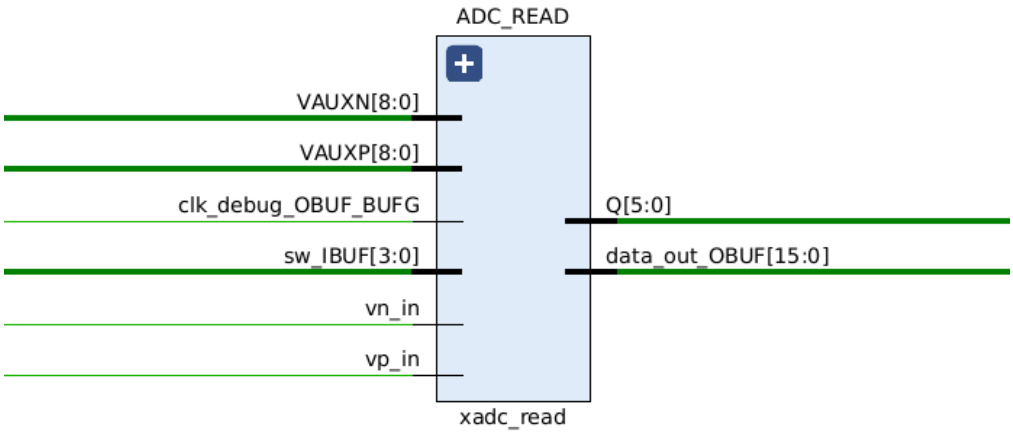
Şekil 3.7. Kurulan UART Modül yapısı

3.6 ADC Modulün Tasarımı

Bu FPGA tabanlı modül, bir Xilinx XADC (Analog-Dijital Dönüştürücü) modülünü kontrol etmek ve kullanıcının analog sinyalleri ölçebilmesi için bir arayüz sunmak üzere oluşturulmuştur. "xadc_read" adlı modül, 100 MHz'lik bir saat sinyali (CLK100MHZ), analog kanalları temsil eden ck_an ve ck_an , diferansiyel analog girişler (vp_in ve vn_in), 4-bit anahtarlama sinyali (sw), ve çeşitli çıkışları içermektedir. Modül, Xilinx XADC Wizard aracı tarafından oluşturulan özel bir XADC örneği olan "xadc_wiz_0" modülünü içerir. Bu modül, FPGA üzerinde analog ölçümler yapabilmek için gerekli konfigürasyonları ve kontrol sinyallerini sağlayan bir XADC modülüdür. "xadc_read" modülü, bu alt modülü kullanarak analog sinyalleri ölçer ve sonuçları işler. Örneğin, seçilen analog kanala bağlı olarak belirli bir LED göstergeyi ayarlar ve bu sayede kullanıcıya ölçüm sonuçlarını görsel olarak sunar.

Modül, içsel olarak bir dizi kontrol sinyali ve durumu yönetir. enable ve ready sinyalleri, XADC modülü ile etkileşimde bulunarak ölçüm başlatma ve tamamlanma durumunu kontrol eder. Ayrıca, Address_in sinyali, hangi analog kanalın ölçüleceğini belirlerken, bad_address sinyali geçersiz bir kanal seçildiğinde kontrolü sağlar.

Bu tasarım, FPGA tabanlı sistemlerde analog sinyalleri ölçmek ve bu ölçümleri işlemek için güçlü ve modüler bir çözüm sunar. Kullanıcı, sw anahtarı aracılığıyla ölçülecek analog kanalı seçebilir ve "xadc_read" modülü, Xilinx XADC modülü ile etkileşimde bulunarak ölçüm sonuçlarını işleyerek kullanıcıya sunar. Bu tasarım, özellikle analog ölçümlerin yapıldığı FPGA tabanlı uygulamalarda kullanıcı dostu bir çözüm sağlamak üzere tasarlanmıştır.



Şekil 3.8. Kurulan ADC Modül yapısı

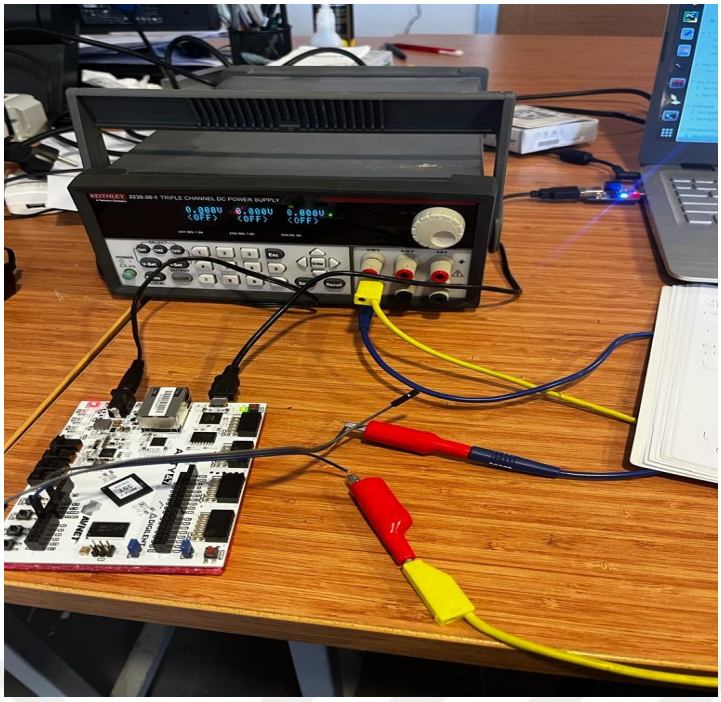
3.7 FIFO Entegrasyonu ve Lojik Yapı

Bu FPGA tabanlı tasarım, bir Xilinx XADC modülü aracılığıyla ölçülen çok kanallı analog verileri seri iletişim (UART) protokolü kullanarak iletmek üzere tasarlanmıştır. "uart_module" adlı modül, bir dizi giriş ve çıkış aracılığıyla analog sensörlerden alınan verileri işler ve bu verileri bir bilgisayar veya başka bir cihaza UART aracılığıyla aktarılmıştır.

Modül, 100 MHz'lik bir saat sinyali (clk) ve çeşitli kontrol sinyalleri ile beslenmiştır. Analog veriler, Xilinx XADC modülü tarafından xadc_read alt modülü aracılığıyla ölçülmüştür. Bu alt modül, diferansiyel analog girişleri (ck_an ve ck_an), tek-sonlu analog girişleri (vp_in ve vn_in), bir anahtarlama anahtarlama sinyali (sw), ve çıkışları (LED, data_out) içerir. "uart_module" modülü, ölçülen verileri depolamak ve iletmek için bir FIFO (First In, First Out) yapısı kullanır. Bu FIFO yapısı, uart_fifo_buffer adında bir dizi ile temsil edilmiştir ve her bir ölçüm verisi, yüksek ve düşük bayt olarak ayrılarak bu diziye eklenir. Ardından, belirlenen zaman aralıklarında UART protokolüne uygun bir şekilde bu veriler seri olarak iletilmiştir.

Modül ayrıca bir dizi gösterge LED'ini kontrol ederek kullanıcıya işlem durumunu görsel olarak bildirir. Ayrıca, iletimin başladığını belirten bir başlangıç baytı da işaret bayrağı olarak kullanılır. (uart_start_flag).

Bu tasarım, FPGA tabanlı sistemlerde çok kanallı analog veri ölçümü ve iletimi için güçlü ve esnek bir çözüm sunar. Hem Xilinx XADC modülü hem de UART iletişim protokolü, kullanıcıya hassas ölçümler yapabilme ve bu verileri harici cihazlara iletebilme imkanı sağlar.



Şekil 3.9. FPGA ile test ölçüm düzeneği

3.8 Verilerin Görselleştirme

FPGA (Field-Programmable Gate Array) tabanlı bir sistemden elde edilen ADC (Analog-Digital Converter) verilerini seri iletişim (UART) yoluyla bilgisayara aktararak gerçek zamanlı olarak görselleştirmek amacıyla geliştirilmiştir. Kodun teknik ayrıntıları şu şekildedir:

1. Veri Yapıları ve Sınıflar:

• CircularBuffer sınıfı, belirli bir maksimum boyuta sahip döngüsel bir veri bufferı oluşturur. Bu tampona veri eklemek, ortalama almak ve zaman farkını hesaplamak için yöntemler sunar.

• ArtyReadADC sınıfı, FPGA'dan gelen UART verilerini işler. Veri işleme sürecinde, belirli bir protokolü takip ederek ADC değerlerini çözer ve bir dizi işlevi içerir.

2. Veri İletişimi:

• ArtyReadADC sınıfı, UART üzerinden iletişim kurar ve belirli bir işaret byte'ını ("42" değeri) algılayarak veri alma işlemini başlatır. Bu işaret, bir çerçevenin başlangıcını belirtir.

• Veri alındığında, ADC değerleri çözülerek uygun bir formata dönüştürülür ve bir buffer'a eklenir.

3. Görselleştirme:

• TemperaturePlotter sınıfı, gerçek zamanlı veri görselleştirmesi için gerekli ayarları yapar. Veriler, bir matplotlib grafik aracılığıyla canlı olarak çizilir.

• Görselleştirme, bir X ve Y ekseni üzerinde sürekli olarak güncellenen verileri içerir. Bu, FPGA tarafından ölçülen analog verilerin zaman içinde nasıl değiştiğini gözlemlemeyi sağlar.

4. Platform Bağımsızlık:

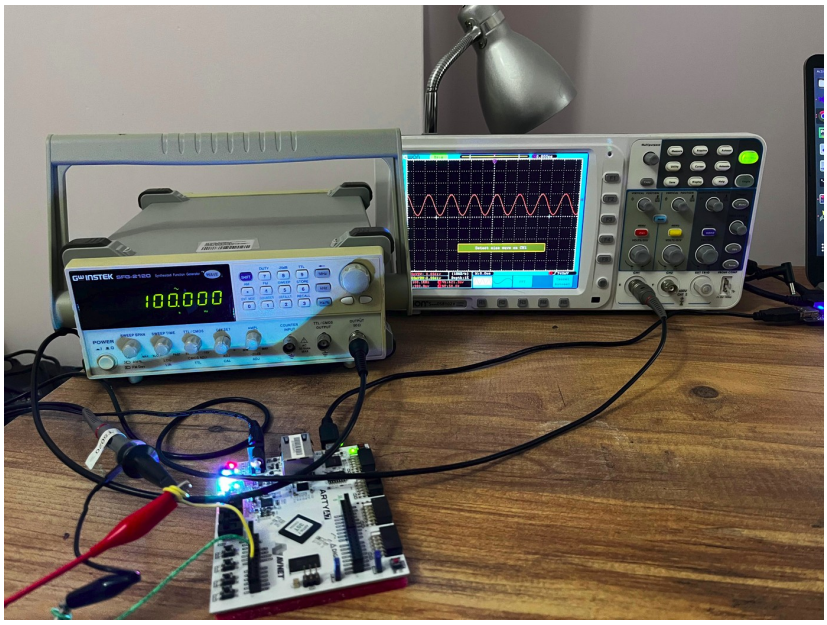
• Kod, Python dilini kullanarak yazılmıştır ve bu nedenle platform bağımsızdır. Python, birçok işletim sistemi üzerinde çalışabilir ve geniş bir kullanıcı kitlesine sahiptir.

• Seri iletişim kütüphanesi olan pyserial kullanılarak, farklı bilgisayar ve FPGA platformları arasında güvenilir bir veri iletişimi sağlanır.

FPGA tabanlı sistemlerde sensör verilerinin gerçek zamanlı olarak izlenmesi ve görselleştirilmesi için genel bir çözüm sunar. Ayrıca, Python'un kullanımı kolay ve geniş kütüphane desteği sayesinde çeşitli platformlarda uygulanabilirliği yüksektir.

3.9.Test Düzeneği

Geliştirilen algoritmanın test edilebilmesi için kurulan düzenek Şekil 3.10’da verilmiştir. ADC girişlerine uygun seviyede ve değişken frekanslı sinüzoidal işaret Instek SFG-2120 20 MHz DDS işaret üreteci ile sağlanmıştır. İşaret doğruluğu ve görüntüleme için Owon SDS8102-V1 100 MHz 2-kanal DSO 2 G/s sayısal osiloskop kullanılmıştır.



Şekil 3.10. FPGA ADC kanallarına uygulanan sinusoidal dalga test ortamı

4. BULGULAR

Geliştirilen donanım mimarisine ait sentez sonuçları Tablo 4.1'de verilmiştir. Vivado 2023.1 sentez raporuna göre, dilim LUT'ların %8.83'ü (1837/20800) kullanılmış olup, bu oran tamamen mantık hücresi olarak kullanılan LUT'lardan oluşmaktadır. Bellek olarak ise LUT'lar kullanılmamıştır. Mantık hücrelerinin %5.16'sı (2146/41600) kullanılmış ve bu kayıtların tamamı flip-flop olarak kullanılmıştır; latch olarak kullanılan hücre ise yoktur. F7 Mux'larının %1.57'si (256/16300) ve F8 Mux'larının %1.57'si (128/8150) kullanılmıştır. Blok RAM ve DSP'ler hiç kullanılmamıştır. Bağlı IOB'lerin %31.43'ü (66/210) kullanılmış olup, bu oldukça yüksek bir kullanım oranıdır. BUFGCTRL'lerin %3.13'ü (1/32) ve XADC'nin %100'ü (1/1) kullanılmıştır. Özellikle XADC'nin tamamen kullanılması önemlidir. Bu oranlar, tasarımın belirli bileşenlerinin etkin kullanımını ve mevcut kaynakların nasıl değerlendirildiğini göstermektedir.

Alan Türu	Kullanılan	Düzeltilen	Kısıtlanan	Mevcut	Kullanım (%)
Slice LUTs	1837	0	0	20800	8.83
LUT as Logic	1837	0	0	20800	8.83
LUT as Memory	0	0	0	9600	0.00
Slice Registers	2146	0	0	41600	5.16
Register as Flip Flop	2146	0	0	41600	5.16
Register as Latch	0	0	0	41600	0.00
F7 Muxes	256	0	0	16300	1.57
F8 Muxes	128	0	0	8150	1.57

Tablo 4.1. Mantık hücrelerinin kullanım raporu

Güç tüketimi açısından, tasarımın düşük güç tüketimini sağlamak için optimize edilmiştir. Geliştirilen donanım mimarisine ait güç tüketim sonuçları Tablo 4.2'de verilmiştir.

Blok RAM ve DSP Kullanımı: Kullanılmayan Blok RAM döşemeleri ve DSP'ler, güç tüketimini düşürmeye yardımcı olmaktadır.

IOB Kullanımı: Bağlı IOB'lerin %31.43'lük yüksek kullanım oranı, belirli alanlarda güç tüketiminin arttığını gösterebilir.

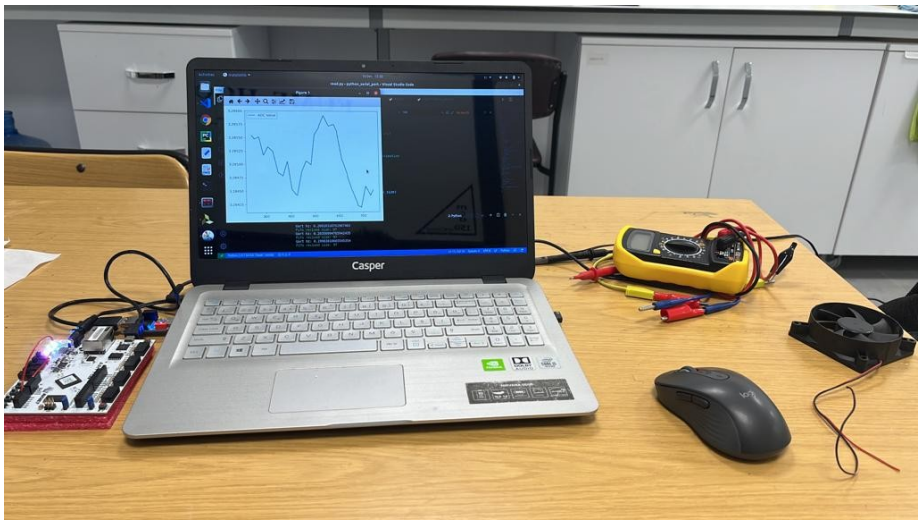
Clocking: BUFGCTRL'lerin %3.13'lük kullanımı, orta seviyede güç tüketimi anlamına gelir.

XADC Kullanımı: XADC'nin %100 kullanımı, bu bileşenin tasarımda önemli bir rol oynadığını ve potansiyel olarak yüksek güç çekişine katkıda bulunabileceğini göstermektedir.

Genel olarak, tasarımın verimli bir şekilde güç tüketimini yönettiği, ancak belirli bileşenlerin daha düşük güç tüketimi için optimize edilebileceği ifade edilmektedir. Mevcut oranlar, tasarımın belirli alanlarında yapılan optimizasyonların ve mevcut kaynakların etkin kullanımının güç tüketimi üzerindeki etkilerini ortaya koymaktadır.

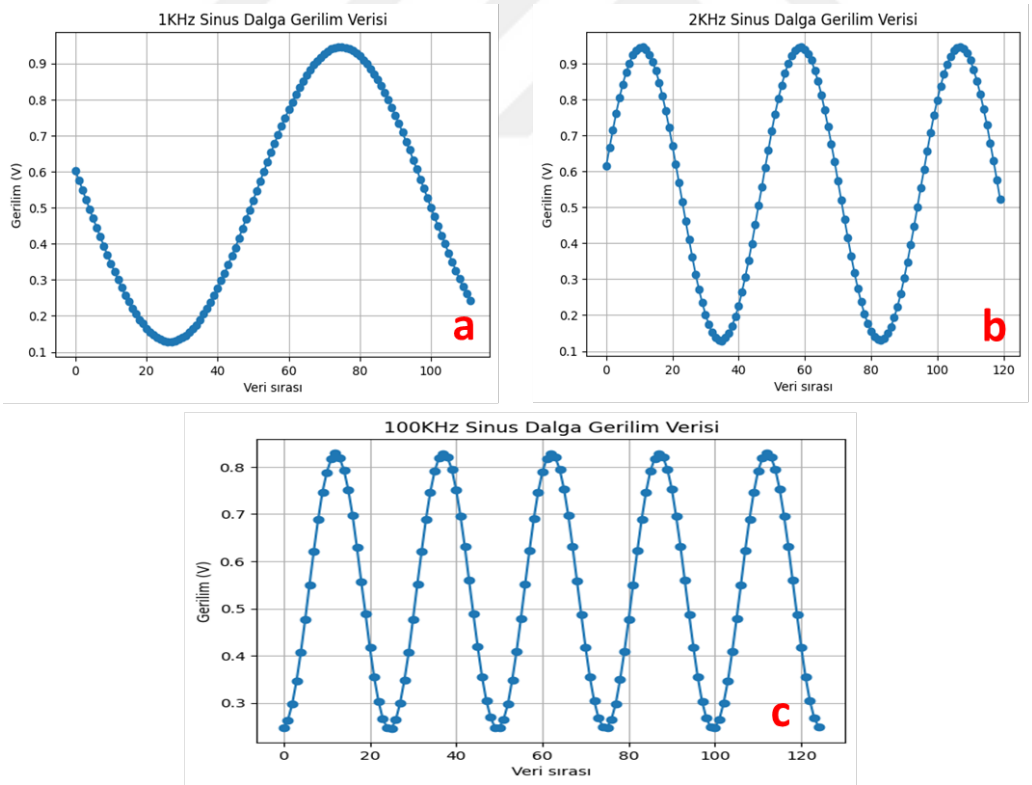
Bileşen	**Güç Tüketimi (W)**	**Kullanım (%)**
Slice Logic	0.600	---
LUT as Logic	0.460	8.83
CARRY4	0.061	0.32
F7/F8 Muxes	0.040	1.18
Register	0.032	5.16
BUFG	0.006	3.13
Others	0.000	---
Signals	0.872	---
I/O	0.691	31.43
XADC	0.121	---
Statik Güç Tüketimi	0.076	---
Toplam	2.360	---

Tablo 4.2. Güç tüketimi raporu



Şekil 4.1. FPGA'den gelen verilerin anlık olarak görselleştirilmesi

Arty A7 FPGA ile yapılan testlerde ADC girişine sinusoidal dalga uygulanmış ve bu dalga biçimı örneklenmeye çalışılmıştır. Buna ek olarak alınan veriler 50 Hz veri paylaşım hızı ile UART üzerinden başka bir bilgisayara aktarılarak görselleştirilmeye çalışılmıştır. Bu süreç sonucunda elde edilen verilere ait bazı örneklemeler Şekil 4.2'deki gibidir.



Şekil 4.2. Farklı frekanslar için elde edilen sayısallaştırma grafikleri

5. TARTIŞMALAR

Bu çalışmada kullanılan veriler direkt olarak güç kaynağından FPGA'nin ADC portlarına yapılan bağlantılar ile toplanmıştır. Ön hazırlık aşaması Vivado 2023.1 üzerinden yazılımsal olarak gerçekleştirilmiştir. Yani her bir modül için ayrı bir test bench oluşturularak modüllerın çalışma modelleri test edilmiş ve tüm modüller üst bir modülde toplanarak cihaz üzerinde koşan bir yazılıma çevrilmiştir. Klasik analog verilerin dijitalize edilme işleminde mikroişlemcilerle daha düşük örnekleme ve iletim hızlarda benzer veriler elde edilir. Literatürdeki bir çok çalışmada olduğü gibi bu çalışmada da ölçüm ve iletim işlemlerı arasındaki saat darbesi farkının önüne geçilebilmek için bir FIFO buffer kullanılmıştır. Böylece iki işlem arasındaki farktan ortaya çıkabilecek bir veri kaybının önüne geçilmeye çalışılmıştır. Ayrıca 8 kanaldan alınan veriler ile hem ölçülen verinin doğrulğunda artış gözlenmiştir hem de iletim modelinde kullanılan buffer modeli sayesinde yüksek hızda yüksek boyutlu veri paylaşımı sağlanmıştır. Günümüzde daha yüksek hız ve hafıza kapasıteli donanımlar ile çok daha hassa ve hızlı ölçümler yapılabilmektedir. Ancak bu tezin amaçlarından bir tanesi olan düşük maaliyetlilik ilkesi gereğince seçilen donanım ile alınancak maksimum verim gözetilmiştir. Sonuç olarak 20ms (50Hz) periyotlarla hem kanaldan herbir kanaldan, 10MHz ADC örnekleme hızıyla, en az 30 örnek barındıracak şekilde ölçumler yapılmış ve aynı süre içerisinde toplam 240 byte büyüklüğündeki veri servis cihazına aktarılmış ve anlık olarak değişim gözlemlenebilir halde sunulmuştur. Bu yapılan tez çalışmasının hedef noktası yalnızca dijital sayısallaştırma olsa da, ortaya konan çalışma daha geniş yelpazede ele alınarak her türlü ölçüm yapılabilecek dahili ve harici sensör ve türleri için de elbette kullanılabilir.

Örneğin, bir tıbbi cihazda kullanılmak üzere ultrasonik sinyal [11] veya doğrudan sanayi ya da üretimde malzeme kalınlıklarının tespit etmek için [12] veya düsük güç tüketimli bir gps sistemi ile iletişimi sağlamak isterken de benzer bir çözüm kullanılanbilmektedir [13].

Ayrıca bu çalışmada önerilen metot, FPGA tabanlı bir sentezlenebilir tasarım içermektedir, bu da çip akışını takip edildiğinde kolayca ASIC (Application-Specific Integrated Circuit) yapısına dönüştürülebilme özelliği taşımaktadır. Bu durum, tasarımın boyutunu önemli ölçüde azaltacağı için güç tüketiminde de belirgin bir düşüş sağlayacaktır. Aynı zamanda, küçük boyutu sayesinde taşınabilir cihazlara entegre edilebilme potansiyelini gösterir, bu da tasarımın geniş bir uygulama yelpazesi içinde kullanılabileceği anlamına gelir.

FPGA kartında bulunan 100 MHz kristal saat üreteci, tüm sistemin tetikleyici sinyali olarak kullanılmıştır. Ancak ADC 10 MHz ile alabilmektedir. Ayrıca 240 byte anlamlı veri içeren FIFO da 50 Hz ile görselleştirilmenin yapıldığı bilgisayara veri aktarabilmektedir. Tüm bu kısıtlardan dolayı

100 kHz’e kadar, Şekil 4.7’de de görüldüğü gibi, anlamlı bir sinus dalga şekli edilmiştir. Bu kısıtlardan kurtulmak için öncelikle seri iletişimin hızlandırılması gerekmektedir. Bunun için aynı FPGA içinde bulunan ethernet altyapısı kullanılabilir [14]. Böylelikle daha yüksek frekanslarda örneklenen sinyaller hızlı bir şekilde görselleştirmenin yapılacağı cihaza aktarılabilir.

6. SONUÇ VE ÖNERİLER

Bu araştırma, FPGA tabanlı bir sistemde yüksek hızlı analog-dijital dönüştürücü (ADC) tarafından örneklelenen sinyallerin görece daha düşük hızla bir bilgisayara iletilmesi amacıyla tasarlanmıştır. FIFO bufferı kullanarak yüksek hızlı ve düşük hızlı saat alanları arasındaki veri iletim sorunları ele alınmıştır. Elde edilen sonuçlar, FIFO'nun bu tür uygulamalarda başarılı bir şekilde kullanılabileceğini göstermiştir. Bu çalışmanın sonuçlarına dayanarak, FPGA tabanlı sistemlerde ADC verilerinin düşük hızlı bir ortama güvenli bir şekilde iletilmesi için FIFO'nun etkili bir çözüm olduğu söylenebilir. Bu çözüm, veri bütünlüğünü korurken aynı zamanda verilere esneklik ve güvenilirlik kazandırmaktadır.

Bu tezin sonuçlarına dayanarak, gelecekteki çalışmalar için bazı öneriler sunulabilir:

1. FPGA kartlarının özelliklerinin artırılması durumunda, çift saatli FIFO kullanarak daha yüksek hızlarda veri iletimi için yeni deneyler yapılabilir.

2. FPGA kartında işlenen veri miktarını artırmak için mikroişlemci kullanımı daha detaylı bir şekilde incelenebilir.

3. Bu çalışmada kullanılan iletişim protokollerinin ve çözümlerinin başka uygulamalara uygunluğu değerlendirilebilir. Eğer daha yüksek hızlar gerekiyorsa ethernet üzerinden bir yapı kurma değerlendirilebilir.

KAYNAKLAR

[1] Ross Freeman, Bernie Vonderschmitt, “Field-Programmable Gate Arrays” IEEE Journal of Solid-State Circuits, vol. 19, no. 4, pp. 452-458, Nisan 1984.

[2] Ross Freeman, Bernie Vonderschmitt, “Field-Programmable Gate Arrays: A New Direction in VLSI”, Proceedings of the IEEE, vol. 72, no.12, pp. 1681-1691, Aralık 1984.

[3] Ahmet Eraslan, “Alan Programlanabilir Kapı Dizilerinin Tasarım ve Uygulamaları”, Elektronik ve Haberleşme Mühendisliği Bülteni, cilt 22, sayı 2, s. 129-142, 2015.

[4] N. A. Atasoy and A. Çavuşoğlu, “Detection of Eye Motion Direction In Real Time Video Image”, 2014 22nd Signal Processing and Communications Applications Conference (SIU), Trabzon, Turkey, 2014, pp. 1487-1490, doi: 10.1109/SIU.2014.6830522.

[5] E. Xie and J. Zhou, “Analysis and Comparison of Asynchronous FIFO and Synchronous FIFO”, 2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA), Changchun, China, 2023, pp. 260-264, doi: 10.1109/EEBDA56825.2023.10090586.

[6] Arty A7 Reference Manual, https://digilent.com/reference/programmable-logic/arty-a7/reference-manual, Son Erişim Tarihi 23.05.2024.

[7] Cummings, Clifford E. “Synthesis and scripting techniques for designing multi-asynchronous clock designs”, In SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers. 2001.

[8] Zhu, Yiyao, Yongquan Chen, and Zhang-Hua Fu. “Knowledge-Guided Two-Stage Memetic Search for The Pickup and Delivery Traveling Salesman Problem With FIFO Loading”, Knowledge-Based Systems 242 (2022): 108332.

[9] G. B. Wakhle, I. Aggarwal and S. Gaba, “Synthesis and Implementation of UART using VHDL Codes”, 2012 International Symposium on Computer, Consumer and Control, Taichung, 2012, pp. 1-3. doi: 10.1109/IS3C.2012.10.

[10] Y. Fang and X. Chen, “Design and Simulation of UART Serial Communication Module Based on VHDL”, 2011 3rd International Workshop on Intelligent Systems and Applications, Wuhan, 2011, pp. 1- 4. doi: 10.1109/ISA.2011.5873448.

[11] Lan, Xiaolong, et al. “Study and Implementation of Ultrasonic Generator Based on FPGA”, 2020 IEEE International Conference on Applied Superconductivity and Electromagnetic Devices (ASEMD). IEEE, 2020.

[12] Shrisha, M. R., et al. “FPGA based Ultrasonic thickness measuring device”, 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, 2018.

[13] Agarwal, Vivek, Hemendra Arya, and S. H. I. V. A. R. A. M. Bhaktavatsala. "Design and development of a real-time DSP and FPGA-based integrated GPS-INS system for compact and low power applications”, IEEE Transactions on Aerospace and Electronic Systems 45.2 (2009): 443-454.

[14] Türk, Serhat, and Serdar Süer Erdem, “Implementation and Performance Analysis of Multi Channel ARINC 429 Controller on FPGA”, 2023 10th International Conference on Electrical and Electronics Engineering (ICEEE). IEEE, 2023.

EKLER

Ek A: FPGA Tabanlı Çok Kanallı Sayısallaştırıcı Tasarımı Verlog kodları.

- UART Modülü:

```
module transmitter(
input clk, //UART input clock
input reset, // reset signal
input transmit, //btn signal to trigger the UART communication
input [7:0] data, // data transmitted
output reg TxD // Transmitter serial output. TxD will be held high during reset, or when no transmissions are taking place.
);

//internal variables
reg [3:0] bitcounter; //4 bits counter to count up to 10
reg [9:0] counter; //14 bits counter to count the baud rate, counter = clock / baud rate
reg state,nextstate; // initial & next state variable
// 10 bits data needed to be shifted out during transmission.
//The least significant bit is initialized with the binary value 0 (a start bit) A binary value 1 is introduced in the most significant bit
reg [9:0] rightshiftreg;
reg shift; //shift signal to start bit shifting in UART
reg load; //load signal to start loading the data into rightshift register and add start and stop bit
reg clear; //clear signal to start reset the bitcounter for UART transmission

//UART transmission logic
```

```
always @ (posedge clk)
begin
if (reset)
begin // reset is asserted (reset = 1)
state <=0; // state is idle (state = 0)
counter <=0; // counter for baud rate is reset to 0
bitcounter <=0; //counter for bit transmission is reset to 0
end
else begin
counter <= counter + 1; //counter for baud rate generator start counting
if (counter >= 49) //if count to 50 (from 0 to 50 --> 2Mbps) clock 100Mhz, baudrate = 2Mbps
begin
state <= nextstate; //previous state change to next state
counter <=0; // reset couter to 0
if (load) rightshiftreg <= {1'b1,data,1'b0}; //load the data if load is asserted
if (clear) bitcounter <=0; // reset the bitcounter if clear is asserted
if (shift)
begin // if shift is asserted
rightshiftreg <= rightshiftreg >> 1; //right shift the data as we transmit the data from lsb
bitcounter <= bitcounter + 1; //count the bitcounter
end
end
end
end
```

// Description of state machine

```
always @ (posedge clk) //trigger by positive edge of clock,
//always @ (state or bitcounter or transmit)
begin
load <=0; // set load equal to 0 at the beginning
shift <=0; // set shift equal to 0 at the beginning
clear <=0; // set clear equal to 0 at the beginning
TxD <=1; // set TxD equals to 1 during no transmission
case (state)
0: begin // idle state
if (transmit) begin // assert transmit input
nextstate <= 1; // Move to transmit state
load <=1; // set load to 1 to prepare to load the data
shift <=0; // set shift to 0 so no shift ready yet
clear <=0; // set clear to 0 to avoid clear any counter
end
else begin // if transmit not asserted
nextstate <= 0; // next state is back to idle state
TxD <= 1;
end
end
1: begin // transmit state
if (bitcounter >=10) begin // check if transmission is complete or not. If complete
nextstate <= 0; // set nextstate back to 0 to idle state
clear <=1; // set clear to 1 to clear all counters
```

```
end
else begin // if transmisssion is not complete
nextstate <= 1; // set nextstate to 1 to stay in transmit state
TxD <= rightshiftreg[0]; // shift the bit to output TxD
shift <=1; // set shift to 1 to continue shifting the data
end
end
default: nextstate <= 0;
endcase
end

endmodule
```

ADC Modülü:

```
module xadc_read(
input CLK100MHZ,
input [8:0] ck_an_p,
input [8:0] ck_an_n,
input vp_in,
input vn_in,
input [3:0] sw,
output reg [7:0] LED,
output reg [15:0] data_out,
output reg adc_ready
);
wire enable;
```

```
reg bad_address = 0;
wire ready;
wire [15:0] data;
reg [6:0] Address_in;

//xadc instantiation connect the eoc_out .den_in to get continuous conversion

xadc_wiz_0 xadc
(
.daddr_in(Address_in), // Address bus for the dynamic reconfiguration port
.dclk_in(CLK100MHZ), // Clock input for the dynamic reconfiguration port
.den_in(enable & ~bad_address), // Enable Signal for the dynamic reconfiguration port
.di_in(0), // Input data bus for the dynamic reconfiguration port
.dwe_in(0), // Write Enable for the dynamic reconfiguration port
.reset_in(0), // Reset signal for the System Monitor control logic
.busy_out(), // ADC Busy signal
.channel_out(), // Channel Selection Outputs
.do_out(data), // Output data bus for dynamic reconfiguration port
.eoc_out(enable), // End of Conversion Signal
.eos_out(), // End of Sequence Signal
.alarm_out(), // OR'ed output of all the Alarms
.drdy_out(ready), // Data ready signal for the dynamic reconfiguration port
.vp_in(vp_in), // Dedicated Analog Input Pair
.vn_in(vn_in),
.vauxp4(ck_an_p[0]), // Auxiliary channel 0
.vauxn4(ck_an_n[0]),
```

```
.vauxp5(ck_an_p[1]), // Auxiliary channel 5
.vauxn5(ck_an_n[1]),
.vauxp6(ck_an_p[2]), // Auxiliary channel 5
.vauxn6(ck_an_n[2]),
.vauxp7(ck_an_p[3]), // Auxiliary channel 6
.vauxn7(ck_an_n[3]),
.vauxp15(ck_an_p[4]), // Auxiliary channel 7
.vauxn15(ck_an_n[4]),
.vauxp0(ck_an_p[5]), // Auxiliary channel 12
.vauxn0(ck_an_n[5]),
.vauxp12(ck_an_p[6]), // Auxiliary channel 13
.vauxn12(ck_an_n[6]),
.vauxp13(ck_an_p[7]), // Auxiliary channel 14
.vauxn13(ck_an_n[7]),
.vauxp14(ck_an_p[8]), // Auxiliary channel 15
.vauxn14(ck_an_n[8])
);
reg _ready = 0;
always@(posedge CLK100MHZ)
_ready <= ready;
//led visual dmm
always @(posedge CLK100MHZ) begin
if (ready == 1 && _ready == 0) begin
data_out <= data;
adc_ready <= 1;
case (data[15:13])
1: LED <= 8'b11;
```

```
2: LED <= 8'b111;
3: LED <= 8'b1111;
4: LED <= 8'b11111;
5: LED <= 8'b111111;
6: LED <= 8'b1111111;
7: LED <= 8'b11111111;
default: LED <= 8'b0;
endcase
end
else begin
adc_ready <= 0;
end
end
//switch driver to choose channel
always @(posedge CLK100MHZ) begin
if (ready == 0 && _ready == 1) begin
case(sw)
0: Address_in <= 8'h14; //A0
1: Address_in <= 8'h15; //A1
2: Address_in <= 8'h16; //A2
3: Address_in <= 8'h17; //A3
4: Address_in <= 8'h1F; //A4
5: Address_in <= 8'h10; //A5
6: Address_in <= 8'h1C; //A6 - A7 differential
7: Address_in <= 8'h1D; //A8 - A9 differential
8: Address_in <= 8'h1E; //A10 - A11 diferential
default: Address_in <= 8'h14; //A0
                    $
```

```
endcase
if (sw >= 4'h9) // switch default case
bad_address <= 1'b1;
else
bad_address <= 1'b0;
end
end
endmodule
```

• **FIFO ve lojik modülü:**

```
module uart_module(
input fpga_clk,
input [8:0] ck_an,
input [8:0] ck_an,
input transmit_btn,
input vn_input,
input vp_input,
input [3:0] switch_arr,
input reset_btn,
output reg led0_b,
output reg led0_r,
output reg led0_g,
output reg led1_b,
output reg led1_r,
output reg led1_g,
output reg led2_b,
output reg led2_r,
output reg led2_g,
```

```
output reg led3_b,
output reg led3_r,
output reg led3_g,
output reg [3:0] led,
output uart_txd,
output uart_txd_debug,
output transmit_debug,
output btn_debug,
output clock_debug,
output [7:0] LED,
output [15:0] data_out
);

parameter FRAME_FIFO_BUFF_SIZE = 120; // 96*2
parameter FRAME_ELEMENTS = 2;
// Intervals
parameter PERIOD_MS = 2000000; // 20ms
parameter UART_INTERVAL = 5000; // 0.05ms
wire transmit_out;
reg [7:0] data_send;
reg [7:0] high_byte;
reg [7:0] low_byte;
reg [7:0] uart_fifo_buffer [0:((FRAME_ELEMENTS*FRAME_FIFO_BUFF_SIZE))];
reg [7:0] test_fifo_buffer [0:19];
reg [7:0] uart_fifo_head;
reg [7:0] uart_fifo_tail;
```

```
reg [7:0] test_fifo_tail;
reg [15:0] adc_value;
reg transmit_flag;
reg fifo_flag;
reg fifo_send_flag;
reg uart_start_flag;
wire adc_flag;
wire uart_flag;
// Counters
reg [33:0] period_counter;
reg [24:0] uart_counter;
initial begin
uart_fifo_head = 0;
uart_fifo_tail = 0;
transmit_flag = 0;
fifo_flag = 1;
led0_b = 0;
led0_g = 0;
led0_r = 0;
led1_b = 0;
led1_r = 0;
led1_g = 0;
led2_b = 0;
led2_r = 0;
led2_g = 0;
led3_b = 0;
led3_r = 0;
```

```
led3_g = 0;
led = 4'b0000;
data_send = 8'b00000000;
test_fifo_buffer[0] <= 8'b11011101; // "."
test_fifo_buffer[1] <= 8'b01011111; // "_"
test_fifo_buffer[2] <= 8'b01011100; // "\"
test_fifo_buffer[3] <= 8'b00101010; // "*"
test_fifo_buffer[4] <= 8'b00101011; // "+"
test_fifo_buffer[5] <= 8'b00101100; // ","
test_fifo_buffer[6] <= 8'b00101101; // "-"
test_fifo_buffer[7] <= 8'b00101110; // "."
test_fifo_buffer[8] <= 8'b00101111; // "/"
test_fifo_buffer[9] <= 8'b00110000; // "0"
test_fifo_buffer[10] <= 8'b00110001; // "1"
test_fifo_buffer[11] <= 8'b00110010; // "2"
test_fifo_buffer[12] <= 8'b00110011; // "3"
test_fifo_buffer[13] <= 8'b00110100; // "4"
test_fifo_buffer[14] <= 8'b00110101; // "5"
test_fifo_buffer[15] <= 8'b00110110; // "6"
test_fifo_buffer[16] <= 8'b00110111; // "7"
test_fifo_buffer[17] <= 8'b00111000; // "8"
test_fifo_buffer[18] <= 8'b00111001; // "9"
test_fifo_buffer[19] <= 8'b00111010; // ":"
test_fifo_tail = 0;
fifo_send_flag = 0;
period_counter = 34'b0;
uart_counter = 24'b0;
```

```
uart_start_flag = 1;
end
assign uart_flag = uart_txd;
assign uart_txd_debug = uart_txd;
assign transmit_debug = transmit_out;
assign btn_debug = reset_btn;
assign clock_debug = fpga_clk;
assign data_wire = data_out;
// ADC read data
xadc_read ADC_READ(fpga_clk, ck_an, ck_an, vp_input, vn_input, switch_arr, LED, data_out, adc_flag);

// Transmit logic definiton
always @(posedge clk) begin
period_counter <= period_counter + 1;
// Transmit Logic
if(period_counter <= PERIOD_MS) begin
if(!fifo_send_flag)begin
if(adc_flag)begin
if (fifo_flag) begin
// Fill the uart buffer with its own protocol
uart_fifo_buffer[uart_fifo_head] <= data_out[15:8]; // Store the current data_out high byte
uart_fifo_buffer[uart_fifo_head + 1] <= data_out[7:0]; // Store the current data_out low byte
led[0] <= data_out[0];
led[1] <= data_out[1];
led[2] <= data_out[2];
```

```
led[3] <= data_out[3];
led0_b <= data_out[4];
led0_g <= data_out[5];
led0_r <= data_out[6];
led1_b <= data_out[7];
led1_g <= data_out[8];
led1_r <= data_out[9];
led2_b <= data_out[10];
led2_g <= data_out[11];
led2_r <= data_out[12];
led3_b <= data_out[13];
led3_g <= data_out[14];
led3_r <= data_out[15];
// Update FIFO head
uart_fifo_head <= uart_fifo_head + FRAME_ELEMENTS;
if(uart_fifo_head   >=   ((FRAME_ELEMENTS*FRAME_FIFO_BUFF_SIZE)))
begin
uart_fifo_buffer[uart_fifo_head] <= 8'b00101010;
uart_fifo_head <= 0;
fifo_flag <= 0;
end
end
end
if(!fifo_flag) begin // Check the adc filled the fifo
if(uart_counter <= UART_INTERVAL) begin
uart_counter <= uart_counter + 1;
if(!TxD) begin
```

```
transmit_flag <= 0;
end
led[0] <= 1;
end // UART Period
else begin
uart_counter <= 0;
transmit_flag <= 1;
data_send <= uart_fifo_buffer[uart_fifo_tail];
uart_fifo_tail <= uart_fifo_tail + 1;
if(uart_fifo_tail >= ((FRAME_ELEMENTS*FRAME_FIFO_BUFF_SIZE)+1))
begin // Check the tail value hit the fifo size
uart_fifo_tail <= 0;
fifo_flag <= 1;
fifo_send_flag <= 1;
end
end // UART Period
if(uart_start_flag) begin // Send communication has been started sign byte (only
once)
uart_start_flag <= 0;
transmit_flag <= 1;
data_send <= 8'b01000011;
end // uart_start_flag
end // fifo_flag
else begin
transmit_flag <= 0;
end // fifo_flag
end // fifo_send_flag
```

```
else begin
transmit_flag = 0;
end // fifo_send_flag
end // 20ms Period
else begin
period_counter <= 0;
fifo_flag = 1;
uart_fifo_head = 0;
uart_fifo_tail = 0;
transmit_flag = 0;
fifo_send_flag = 0;
end // 20ms Period
end // always

// Instantiate the transmitter module
transmitter T1(.clk(fpga_clk), .reset(reset_btn), .transmit(transmit_flag),
.data(data_send), .uart_txd(uart_txd));

endmodule
```

Ek B: Seri Haberleşme Verilerinin Görselleştirilmesi Python Kodları.

```
import serial
import time

import matplotlib.pyplot as plt
from collections import deque
import numpy as np
from colorama import init as colorama_init
from colorama import Fore
from colorama import Style
import threading
from collections import deque
from matplotlib.animation import FuncAnimation
import statistics
from itertools import count
import random

class CircularBuffer:
def __init__(self, max_size):
self.max_size = max_size
self.buffer = deque(maxlen=max_size)

def append(self, value):
self.buffer.append(value)

def get_buffer(self):
return list(self.buffer)

circuler_buffer = CircularBuffer(max_size=48)
index = count()
```

```
class ArtyReadADC(object):


def __init__(self, Adc_max_v, Adc_resolution, Port, Baudrate, Parity, Stop_bits,
Byte_size, Time_out):
self.max_v = Adc_max_v
self.resolution = Adc_resolution
self.buffer_size = 50 # Uart_hz_buff_size
self.buffer = [0] * self.buffer_size
self.index = 0
self.count = 0
self.temp_count = 0
self.uart_hz = 0
self.fifo_flag = False
# Initialize last read time
self.last_read_time = time.time()
self.fifo_size_counter = 0
self.byte_counter = 0
self.byte_array = []
# Initialize serial port
self.ser = serial.Serial(
port=Port,
baudrate=Baudrate,
parity=Parity,
stopbits=Stop_bits,
bytesize=Byte_size,
timeout=Time_out
)
```

```
if not self.ser.isOpen():
self.ser.open()
def readData(self):
try:
data = self.ser.read()
if(data.hex() == hex(42)[2:]):
time_since_last_read = self.timeDiff()
self.bufferAddElement(time_since_last_read)
self.uart_hz = self.bufferMean()
self.last_read_time = time.time()
print(f"Uart hz: {self.bufferMean()}")
print(f"{Fore.GREEN}Fifo recived size: {self.fifo_size_counter+1}
{Style.RESET_ALL}")
self.fifo_size_counter = 0
self.fifo_flag = True

else:
self.fifo_size_counter += 1
if(self.fifo_flag and self.fifo_size_counter > 0):

data_integer = int.from_bytes(data, byteorder='big') # Use 'little' if the byte
order is little-endian)
self.byte_array.append(data_integer)
self.byte_counter += 1
if(self.byte_counter == 2):
self.byte_counter = 0
raw_temp_value = self.getPackData(self.temp_count, self.byte_array)
```

```
temp_value = raw_temp_value * self.max_v / (pow(2, self.resolution))
circuler_buffer.append(temp_value)
print(f"Voltage: {temp_value} V")
self.temp_count += 2
if(self.temp_count >= 240):
self.fifo_flag = False
self.temp_count = 0
self.byte_array.clear()
return data
except serial.SerialException as e:
print(f"Error reading data: {e}")

def bufferAddElement(self, value):
self.buffer[self.index] = value
self.index = (self.index + 1) % self.buffer_size
if self.count < self.buffer_size:
self.count += 1
def bufferMean(self):
if self.count == 0:
return None # Return None if the buffer is empty
return 1/(sum(self.buffer) / self.count)

def timeDiff(self):
# Calculate the time difference since the last read
return time.time() - self.last_read_time

def getPackData(self, arr_index, byte_arr):
```

```
return (byte_arr[arr_index] << 8) | byte_arr[arr_index+1]


class TemperaturePlotter:
def __init__(self,Adc_max_v, Adc_resolution, Port, Baudrate, Parity, Stop_bits, Byte_size, Time_out):
# Form a deque with a maximum capacity of 50 elements.
self.adc_reader = ArtyReadADC(Adc_max_v, Adc_resolution, Port, Baudrate, Parity, Stop_bits, Byte_size, Time_out)
self.mean = 0
self.x_vals = deque(maxlen=500)
self.y_vals = deque(maxlen=500)
self.last_animation_time = time.time()
# Configure the plot settings
plt.xlabel('Index')
plt.ylabel('Value')
plt.legend()


def create_x_axis(self,input_array):
return [i for i in range(len(input_array))]
def getBuffer(self,circular_buffer):
self.graph_buffer = circular_buffer


def animate(self,i):
# Append the new value to the deque
# Calculate the period


self.adc_reader.readData()
```

```
if not len(circuler_buffer.get_buffer()) == 0:
self.mean = statistics.mean(circuler_buffer.get_buffer())
else:
self.mean = 0
if len(self.x_vals) <= 500:
self.x_vals.append(next(index))
self.y_vals.append(self.mean)
plt.cla()
plt.plot(self.x_vals, self.y_vals, label='ADC Value')
plt.legend(loc='upper left')
plt.tight_layout()

def main():

graph_visualizer = TemperaturePlotter(Adc_max_v=3.3, Adc_resolution=16, Port='/dev/ttyUSB1', Baudrate=115200, Parity='N', Stop_bits=1, Byte_size=8, Time_out=0.01)
ani = FuncAnimation(plt.gcf(), graph_visualizer.animate, interval=1)
plt.show()
if __name__ == "__main__":
main()
```