İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

VIVADO YÜKSEK SEVİYELİ TASARIM ARACI KULLANARAK KIRMIK ÜSTÜ SİSTEM TASARIMI

LİSANS BİTİRME TASARIM PROJESİ

Barış BİLGİLİ

Ceyhun YAMANEREN

Kubilay VATANSEVER

Umut ÇOLTU

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

MAYIS, 2019

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

VIVADO YÜKSEK SEVİYELİ TASARIM ARACI KULLANARAK KIRMIK ÜSTÜ SİSTEM TASARIMI

LİSANS BİTİRME TASARIM PROJESİ

Barış BİLGİLİ
(040140067)

Ceyhun YAMANEREN
(040140010)

Kubilay VATANSEVER
(040140045)

Umut ÇOLTU
(040150602)

Proje Danışmanı: Doç.Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

MAYIS, 2019

İTÜ, Elektronik ve Haberleşme Mühendisliği Bölümü'nün ilgili Bitirme Tasarım Projesi yönergesine uygun olarak tamamen kendi çalışmamız sonucu hazırladığımız "Vivado Yüksek Seviyeli Tasarım Aracı Kullanarak Kırmık Üstü Sistem Tasarımı" başlıklı Bitirme Tasarım Projesi'nin Ara Raporu'nu sunmaktayız. Bu çalışmayı intihal olmaksızın hazırladığımızı taahhüt eder; intihal olması durumunda bitirme tasarım projesinin başarısız sayılacağını kabul ederiz.

Barış BİLGİLİ ....
(040140067)

Ceyhun YAMANEREN ....
(040140010)

Kubilay VATANSEVER ....
(040140045)

Umut ÇOLTU ....
(040150602)

Proje Danışmanı : Doç.Dr. Sıddıka Berna ÖRS YALÇIN ....

ÖNSÖZ

Bitirme projemiz boyunca bizim yanımızda olan ve her hafta vaktini ayıran, bize olan desteğini hiç esirgemeyen sayın Doç.Dr. Sıddıka Berna ÖRS YALÇIN'a ve bitirme ödevimiz boyunca bize yol gösterip gerektiği anlarda yardımını esirgemeyen Araş. Gör. Yük. Müh. Latif Akçay'a çok teşekkür ederiz.

İstanbul Teknik Üniversitesi lisans eğitimi hayatımız boyunca bize katkı sağlayan, bakış açımızı genişleten değerli hocalarımıza ve arkadaşlarımıza teşekkür ederiz.

Son olarak, her anımızda bize destek olan ve hiçbir şeylerini esirgemeyen değerli ailelerimize sonsuz teşekkürlerimizi sunarız.

Mayıs 2019

Barış Bilgili
Ceyhun Yamaneren
Kubilay Vatansever
Umut Çoltu

İÇİNDEKİLER

KISALTMALAR

AES	: Advanced Encryption Standard
ARM	: Acorn RISC Machine
AXI4	: Advanced eXtensible Interface 4
BRAM	: Block RAM
DES	: Data Encryption Standard
DMA	: Direct Memory Access
DSP	: Digital Signal Processing
FF	: Flip Flop
FIFO	: First In First Out
FPGA	: Field Programmable Gate Array
HDL	: Hardware Description Language
HLS	: High Level Sythesis
I/O	: Input / Output
ILA	: Integrated Logic Analyser
LUT	: Look Up Table
LUTRAM	: Look Up Table RAM
Mbps	: Megabits per second
NIST	: National Institute of Standards and Technology
RAM	: Random Access Memory
RSA	: Rivest Shamir Adleman Algorithm
RTL	: Register Transfer Level
SDK	: Software Development Kit
SHA	: Secure Hash Algorithm
STS	: Station to Station
SURF	: Speeded-Up Robust Features
VHDL	: Very High –Speed Integrated Circuit Hardware Description Language

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

Sayfa

VIVADO YÜKSEK SEVİYELİ TASARIM ARACI KULLANARAK KIRMIK ÜSTÜ SİSTEM TASARIMI

ÖZET

Teknolojinin hızla geliştiği günümüz dünyasında bilgisayarların önemi hızla artmaktadır. Küreselleşen dünya ile birlikte dünyanın her yerindeki verilere ulaşma ve dünyanın her yerindeki alıcılara veri gönderme ihtiyacının artması nedeniyle internet kullanımı günümüzün vazgeçilmez ihtiyaçlarından olmuştur. İnternet kullanımının bu derece artması ile birlikte önemli verilerin sanal ortama geçmesi de kaçınılmaz bir hale gelmiştir. Sanal ortamda birçok önemli verinin transferi söz konusu olduğu için bu ortamda üçüncü kişiler tarafından gerçekleştirilen saldırılar gözlemlenmektedir. Böyle bir ortamda veri transferi gerçekleştirirken bu verilerin korunmasının önemi nedeniyle kriptoloji biliminde büyük bir ilerleme kaydedilmiştir. Teknolojinin gelişimi ve bununla beraber veri elde etmek için yapılan saldırıların da gelişmesi ile birlikte pek çok gelişmiş kriptoloji algoritmaları da ortaya çıkmıştır.

Bu bitirme projesi kapsamında günümüzde yaygın olarak kullanılan Gelişmiş Şifreleme Standardı (Advanced Encryption Standard – AES), Rivest Shamir Adleman Algoritması (Rivest Shamir Adleman Algorithm- RSA), Güvenli Özet Algoritması 3 (Secure Hash Algorithm – SHA 3) kripto algoritmaları gerçeklenmiştir. Bu kripto algoritmalarının gerekli donanımlara implementasyonu gerçekleştirilirken yaygın olarak donanım tanımlama dilleri olan VHDL ve Verilog kullanılarak tasarlanmıştır. Bu projede yapılan gerçekleme esnasında ise yaygın olarak kullanılan yüksek seviyeli diller olan C ve C++ üzerinden tasarım yapılarak gerçeklemenin sağlanacağı bir yol izlenmiştir. Bu dillerle yapılan tasarım esnasında Xilinx firmasının geliştirdiği Vivado Yüksek Seviyeli Sentez (Vivado High Level Synthesis – Vivado HLS) ortamı kullanılmıştır. Bu platform kullanılarak ilgili algoritmaların donanımsal gerçeklemeleri yapılmış ve testleri yine aynı platform üzerinde gerçekleştirilmiştir. Daha sonra ilgili algoritmaları içeren modüller kullanılarak uygun arayüzlerde tasarım yapmıştır. Bu arayüzler seçilirken sistemin gereksinimleri, performansı ve optimizasyonu göz önünde bulundurulmuştur. İlgili modüller daha sonra Xilinx Vivado ortamında donanım tanımlama dilleri ile tekrar tasarlanmıştır. Yüksek seviyeli tanımlama dilleri ile tasarlanan ilgili modüller ve donanım tanımlama dilleri ile tasarlanan ilgili modüllerin birbirleri ile kullandıkları alan, işlemi gerçekleştirdikleri saat döngüsü sayısı, maksimum saat frekansı değerleri bakımından karşılaştırılması gerçekleştirilmiştir. Bu karşılaştırma yapıldıktan sonra yüksek seviyeli diller ile tasarlanan modüller kullanılarak işlemci tabanlı bir sistem oluşturulmuştur. Güvenliği sağlanacak verilerin boyutlarının büyük olması muhtemel olduğundan sistemin yüksek hızlı bir şekilde çalışabilmesi için sistemde direkt hafıza erişimi (Direct Memory Access – DMA) kullanılmıştır. DMA ile oluşturulan işlemci tabanlı sistemde *Station-to-Station* haberleşme protokolüne uygun olarak iki kişi arasında kriptolu iletişim sağlanmıştır. Bu haberleşme protokolü kullanılırken üçüncül kişiler tarafından

yapılabilecek olan saldırılar ve bilgi transferi sırasında yaşanabilecek olan gürültü kayıpları hesap edilmemiştir.

Projenin gerçekleştirilmesi esnasında yapılan çıkarımlar sonucu yüksek seviyeli yazılım dilleri kullanılan Vivado HLS ortamının beklenildiği düzeyde performans gösteremediği, kaynak kullanımı ve zamanlama ile ilgili tahminlerinde donanım tanımlama dilleri kullanılan Vivado geliştirme ortamına kıyasla farklar gösterdiği görülmüştür. İşlemin gerçekleştiği zaman döngüsü sayıları, kaynak kullanımı değerleri açısından iki ortamın farklı sonuçlar verdiği gözlemlenmiştir.

Yüksek seviyeli yazılım dilleri kullanılarak tasarlanan kripto modüllerinin VHDL dili ile tasarlanan kripto modüllerine kıyasla daha geride kaldığı gözlemlenmiştir. VHDL dili ile yazılan kodların daha optimize olarak tasarlandığı ve birçok kısımda yüksek seviyeli yazılım dilleri ile tasarlanan modüllerinden daha üstün sonuçlar verdiği görülmüştür. Bunun nedeni olarak donanım tanımlama dilleri ile tasarım yapılırken kod donanıma yönelik yazıldığı için sonuçların daha optimize çıktığı düşünülmektedir.

SYSTEM ON CHIP DESIGN WITH VIVADO HIGH-LEVEL SYNTHESIS TOOL

SUMMARY

In today's world where technology is developing rapidly, the importance of computers is increasing very quickly. Internet use has become one of the indispensable needs of today due to the increasing need to reach data all over the world and sending data to receivers all over the world with the globalizing world. With this increase in internet usage, the need to enter the vital data in the digital word has become inevitable. Because of the transfer of many important data in the virtual environment, the attacks carried out by third parties are observed in this environment. There has been a great deal of progress in cryptology, due to the importance of protecting these data while transferring data in such an environment. With the development of technology and the development of attacks to obtain data, many advanced cryptology algorithms have emerged.

Within the scope of this graduation project, today’s most commonly used crypto algorithms such as Advanced Encryption Standard (AES), Rivest Shamir Adleman Algorithm (RSA), Secure Hash Algorithm (SHA 3) are designed and used. VHDL and Verilog, which are hardware description languages, are commonly used while implementing these crypto algorithms into a hardware system. During this project, commonly used high level software languages such as C and C++ are used to implement the mentioned algorithms into a hardware. During the design stage of these algorithms with the high level languages, Vivado High Level Synthesis (Vivado HLS) environment developed by Xilinx was used. Using this platform, hardware implementations of related algorithms were made and tests were carried out on the same platform. Then, it has designed the appropriate interfaces by using the modules which include the related algorithms. The interfaces which are tested while designing the modules are AXI4-Lite and AXI4-Stream.

Firstly, AXI4-Lite is tested for the mentioned modules. It is seen that the outcome of the test was successful, however the performance of the system does not satisfy the expected result. Since AXI4-Lite supports a single burst per transfer, the system was working slower than required. Due to this situation, an interface which provides more burst transaction mode is searched. Therefore AXI4-Stream interface is selected for the system. With the help of the AXI4-Stream interface, infinity number of burst transactions could be made. The requirements, performance and optimization of the system were taken into account and because of this, AXI4-Stream is selected for the interface of the designed modules. The related modules are then redesigned with hardware description languages in the Xilinx Vivado environment. The relevant modules designed with high level identification languages and the related modules designed with the hardware description languages were compared with each other in terms of the utilization, the number of clock cycles they performed and the maximum

clock frequency values. After this comparison, a processor based system was created by using modules designed with high level languages.

Direct memory access (DMA) is used in the system for high speed operation of the system, since the size of the data to be secured is likely to be large. DMA has two types of interfaces which are AXI4-Lite and AXI4-Stream. The interface which is connected to the processor is the AXI4-Lite interface and the interface which operates with the modules is AXI4-Stream. DMA provides a high speed transfer by letting the modules accessing the memory. In addition, DMA also provides a help to the processor by sharing the workload. With this phenomenon, the system performs much better compared to the other system which are not DMA based. DMA creates a “last” signal after finishing the desired transfer. The last signal symbolizes the end of the transaction. It is also vital for the DMA module to obtain the last signal after the last pack is reached, otherwise the DMA crushes. The last signal is created with the help of the FIFO modules which are connected to the designed modules’ input and output.

The data transfer which is performed by the DMA is based on byte level transfer. DMA reads by bytes and puts these bytes in an order to fulfill the required bit length which the transfer needs. Due to the byte level operation, the data must be arranged in the order which the modules require to perform their operations. While obtaining data from a FIFO, DMA separates the obtained data to bytes and does the transfer that way. DMA can operate with 8 bit, 16 bit, 32 bit, 64 bit, 128 bit, 256 bit, 512 bit, 1024 bit length.

DMA module was tested separately with the single channel mode. Every modules pins which were connected to the FIFO’s was tested with Integrated Logic Analyzer to observe the data streams. These test was made to see the transfer in live action to witness whether there was a mistake by the transaction or not. After the tests were successful, the cryptographic communication system stage had begun.

Cryptographic communication is simulated between two people with Station-to-Station communication protocol in DMA and processor based system. While this communication protocol is being used, possible attacks carried by third parties and the loss of noise during the transfer of information are not taken into account.

As a result of the conclusions made during the realization of the project, it was seen that the Vivado HLS environment, which uses high level software languages, did not perform at the expected level, and its estimates for resource usage and timing showed differences compared to the Vivado development environment using hardware description languages. It was observed that the two environments gave different results in terms of the number of time cycles in which the process occurred and the values of the resource usage.

Crypto modules designed using high-level software languages have been observed to be lagging behind compared to the cryptographic modules designed with VHDL language. The codes written with VHDL language are designed to be more optimized and give better results than the modules designed with high level language. The reason for this is that it is considered that the results are more optimized because the code is written for the hardware when designing with hardware description languages.

1. GİRİŞ

1.1 Projenin Amacı

Bu projede Gelişmiş Şifreleme Standardı (Advanced Encryption Standard – AES) [3], Rivest Shamir Adleman Algoritması (Rivest Shamir Adleman Algorithm - RSA) [14], Güvenli Özet Algoritması 3 (Secure Hash Algorithm – SHA 3) [4] kripto algoritmalarının Vivado Yüksek Seviyeli Sentez platformu (Vivado High Level Synthesis – Vivado HLS) [19] kullanılarak donanım gerçeklemeleri yapılmıştır. Kriptografi yapı taşlarının donanım gerçeklemeleri ARM işlemcisine (Acorn RISC Machine) hızlandırıcı olarak eklenerek bir güvenlik protokolü Vivado Design Suite ile tasarlanmıştır [18]. Elde edilen modüller birleştirilerek şifreleme ve deşifreleme yapan iki kullanıcının haberleşmesi simüle edilmiştir.

Bu sistemin Vivado HLS üzerinde tasarlanması sayesinde daha önce teknik özelliklerde belirtilen tasarım süresinin kısaltılması, donanım tanımlama dilleri yerine yüksek seviyeli yazılım dilleri (C, C++) kullanılarak işlem seviyesinde tasarımlar yerine uygulanacak olan kripto algoritmasına odaklanılması, optimizasyon yöntemlerinin hızlı bir biçimde uygulanması ile donanım tanımlama dilleriyle oluşturulan tasarımlara kıyasla daha kolay optimizasyon yöntemlerinin uygulanması ve böylelikle performansı daha yüksek bir sistemin elde edilmesi amaçlanmıştır. Literatür araştırmalarında görüldüğü üzere kriptografi algoritmalarının gerçeklenmesi genellikle donanım tanımlama dilleri ile gerçeklenmiştir. Vivado HLS ile gerçeklenen bir kripto algoritmasına literatürde sadece birkaç çalışmaya rastlanmıştır.

Bu proje ile Vivado HLS aracı kullanılarak performans isterleri analizi yapmak sadece donanım tanımlama dilleri kullanılarak yapılan tasarıma göre daha kolaydır. Bu sayede alan ve hız analizleri yapılması amaçlanmaktadır. Gerçeklenecek olan kripto algoritmaları yüksek hızlı tümleşik devreler için donanım tanımlama dili (Very High –Speed Integrated Circuit Hardware Description Language – VHDL) ile de tasarlanarak Vivado HLS ortamında gerçeklenen algoritma ile performans bakımından karşılaştırılması da amaçlanmaktadır [12].

1.1.1 Projenin ikincil amaçları

Birincil amaç olan kriptografi sistemi gerçekleştirilirken kullanılan sisteme uygun arayüzler araştırılmıştır. Elde edilen sonuçlar neticesinde kullanılabilecek arayüzlerin

AXI4-Lite ve AXI4-Stream olması kararlaştırılmıştır [15]. Bu nedenle bu iki arayüz detaylı olarak araştırılmış ve veri transfer protokolleri öğrenilmiştir. Kriptografi sisteminin gerçeklenmesi için kullanılacak her bir modül, bu arayüzlerle tasarlanmış ve denenmiştir.

Kriptografi sistemi oluşturulurken performans göz önünde alındığı için sistemde Direkt Hafıza Erişimi (Direct Memory Access - DMA) [17] kullanılması gerektiği saptanmıştır. Bu doğrultuda DMA'ya uyumlu veri haberleşmesi protokolleri ve arayüzleri araştırılmıştır. Her bir sistem DMA için uyumlu hale getirilmiş ve gerçeklenmiştir.

1.2 Literatür Araştırması

Vivado HLS C, C++ gibi yüksek seviyeli dillerde yazılmış kodları sentezlenebilen yazmaç transfer seviyesinde (register-transfer level – RTL) tasarımlara dönüştürebilen bir araçtır [19]. Donanım ve yazılım arasında bir geçiş sağlar. Donanım tasarımcılarının verimliliğini arttırmaya, yazılım tasarımcıları için sistem performansını arttırmaya, yüksek seviyeli yazılım dilleri düzeyinde tasarım yapmaya olanak sağlayarak harcanan zamanı azaltmaya, performansı arttırmak için optimizasyonu kontrol etmeye yarar.

AES algoritması [5] gerçeklenen bu çalışmada HLS aracı kullanıldığı durum ile RTL tasarım arasında kıyaslamalar yapılmıştır. Sayısal işaret işleme gibi uygulamalarda geniş bir kullanım alanı bulan HLS aracının AES gibi algoritmalarda kullanımı incelenmiştir. İşaret işleme operasyonlarıyla karşılaştırıldığında AES algoritmasındaki işlemler HLS için daha zor olabilmektedir. HDL ile kapı seviyesinde tanımlamalar yapmak kolay olmasına rağmen algoritmaları gerçeklemek yüksek seviyeli dillere göre daha zor olmaktadır [2]. HLS aracının performansı kullanılan yüksek seviyeli dilde yazılmış kaynak koduna bağlılık göstermektedir. HLS kullanarak giriş ve çıkış portları direktiflerle kontrol edilebilmekte olup geniş seçenekler sunabilmektedir. Türkiye'de geliştirilen hızlandırılmış sağlam çehre (Speeded-Up Robust Features – SURF) algoritmasının gerçeklenmesi çalışmasında aşağıdaki sebeplerden ötürü HLS aracının kullanılmasına karar verilmiştir [6].

Vivado HLS programı sayesinde doğrudan yüksek seviyeli yazılım dilleri kullanarak RTL tasarımlar yapılabilmektedir. Bu tasarımlar hızlandırıcı olarak projeye eklenebilir. Bu kodlar yazılırken HLS’de tanımlı direktifler kullanarak tasarım üzerinde optimizasyonlar yapılabilir. Bu sayede kullanılan alanı azaltıp aynı zamanda performansta artış sağlanabilir [18]. Kullanılan direktiflere rağmen eğer daha optimize bir durum sağlanabiliyorsa HLS tasarımı bu şekilde yapar. Bu direktiflerden bazıları aşağıda açıklanmıştır:

İş hattı (pipeline): İş hatları kullanılarak fonksiyonların paralel çalışması sağlanabilir. Bu sayede algoritmanın daha hızlı çalıştırılması mümkün hale gelir.

Dizi tanımlamaları: HLS direktifleri kullanılarak diziler istenilen şekilde birleştirilebilir yatay veya dikey dizi olarak saklanabilir. İsteğe göre rastgele erişilebilir bellek (Random Access Memory- RAM) veya kuyruk (First-In First-Out - FIFO) olarak tanımlanabilir.

Döngü tanımlamaları: Döngüler donanım kullanımını azaltmak için birleştirilebilir, direktifler kullanılarak farklı şekillerde yeniden tanımlanabilir.

HLS ortamında oluşturulan hızlandırıcılar bir işlemci kullanılarak işlemci üzerinde tanımlı kütüphanelerdeki fonksiyonlarla kontrol edilebilir.

2. ÖNBİLGİLER

2.1 Yüksek Seviyeli Tasarım

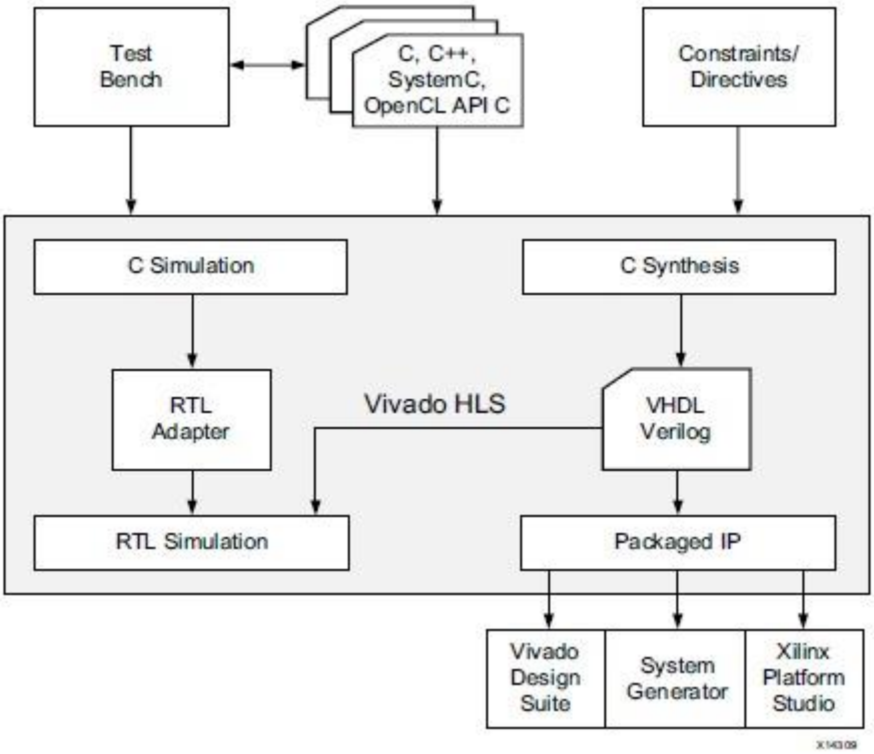
Günümüzde programlanabilir kapı dizileri (Field Programmable Gate Array - FPGA) [1] çok sayıda programlanabilir mantık devreleri, gömülü hesap devreleri, bellek elemanları ve sayısal işaret işleyici (digital signal processing – DSP) bloklarından [14, 15] oluşan ve karmaşık projelerde direk kullanılmak veya prototip üretimi yapmak için fazlaca tecih edilen çok büyük çapta tümleşik devrelerdir. FPGA'ler kullanılarak yapılan donanım tasarımları geleneksel olarak donanım tanımlama dilleri (Hardware Description Language - HDL) ile yapılmaktadır. VHDL ve Verilog çoğunlukla sanayide ve akademik çalışmalarda kullanılan donanım tanımlama dilleridir [12].

FPGA üzerinde donanım gerçeklemesi yapılırken genellikle geleneksel donanım tanımlama dilleri kullanılır. Bu şimdiye kadar kabul edilmiş, sektörde yer etmiş ve birçok ürünün hazır ve işler haline gelmesini sağlamış yoldur. Fakat karmaşıklaşan mimarilerin tasarım süreçlerinde donanım tasarım dillerinin kullanılması oldukça vakit harcamakta ve tasarım süreçlerini bazen zorlaştırmaktadır [19]. Bunun önüne geçilebilmesi maksadıyla alternatif teknolojiler geliştirilmiştir.

Bu alternatif yollardan en güncel ve sektörde kullanım alanı bulan yüksek seviyeli sentezdir (High Level Synthesis - HLS) [19]. Birçok firma, donanım tasarlayan kişilerin işlerini kolaylaştırmak ve yapılacak olan sisteme kapı seviyesinden değil de daha üst bir seviyeden bakılarak tasarımın gerçekleştirilmesi maksadıyla araçlar tasarlamıştır. Bu araçlar C, C++ gibi yüksek seviyeli dillerde yazılmış kodları sentezlenebilen RTL tasarımlarına dönüştürebilen araçlardır. Donanım ve yazılım arasında bir köprü görevi görerek geçiş sağlarlar. Donanım tasarımcılarının verimliliğini arttırmaya, yazılım tasarımcıları için sistem performansını arttırmaya, yüksek seviyeli yazılım dilleri düzeyinde tasarım yapmaya olanak sağlayarak harcanan zamanı azaltmaya, performansı arttırmak için optimizasyonu kontrol etmeye yönelik kullanılırlar.

2.1.1 Vivado HLS ortamı

Bu bitirme projesinde kullanılacak olan yüksek seviyeli tasarım programı, Xilinx firmasının bir aracı olan Vivado HLS'tir [19]. Vivado HLS aracının yüksek seviyeli bir dilden RTL şematiğini çıkarmasına ve bunu test etmesine kadar geçen süreç Şekil 2.1'de gösterilmiştir.



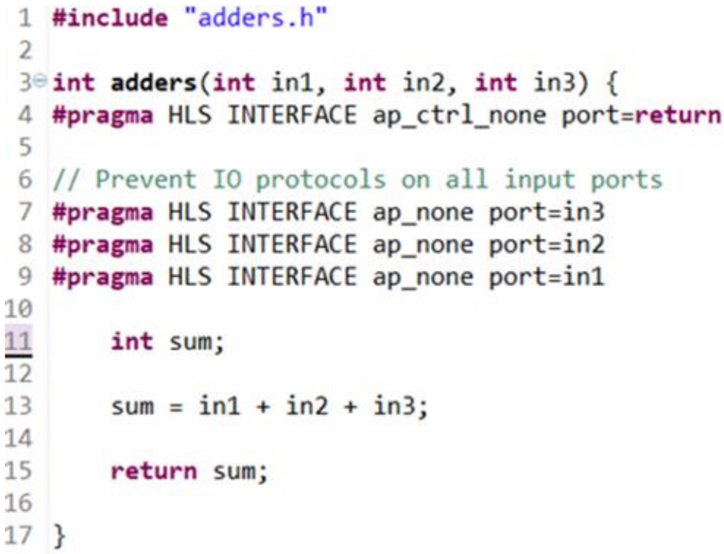
Şekil 2.1 : Vivado HLS aracının tasarım akış şeması [19].

Tasarım akış şemasından da görüleceği üzere, araç öncelikle desteklediği yüksek seviyeli bir programa dili (C, C++, SystemC) ile sayısal bir tasarım yapıldığı gözetilip gözetilmesin fark etmeksizin yazılmış olan kodu ve koda gömülmüş belirli kısıtlar ve direktifleri dikkate alarak bir RTL şematiği oluşturur. Bunun yanı sıra, istenilen kısıtlara, arayüzlere ve birçok özelliğe sahip tasarımın VHDL ve Verilog dillerindeki karşılıklarını da çözüme ekler. Bu oluşturulmuş tasarımın verifikasyonunun yapılması maksadıyla da yine yüksek seviyeli bir dille yazılmış test dosyası ile RTL benzetimini gerçekleştirir.

2.1.1.1 Örnek bir Vivado HLS projesi

Vivado HLS aracının kullanımının ve çalışma ortamının anlatılması için 3 adet değişkenin toplandığı ve sonucun çıkışa verildiği bir tasarımın işlem adımları gösterilecektir.

Öncelikle yeni bir HLS projesi oluşturulur ve *Sources* kısmına C ile yazılacak kodun kaynak dosyası eklenir. *Test Bench* kısmına da RTL şematiği oluşturulduğunda tasarımın doğruluğunun kontrol edilmesi için yine C ile yazılmış bir test kodu eklenir.



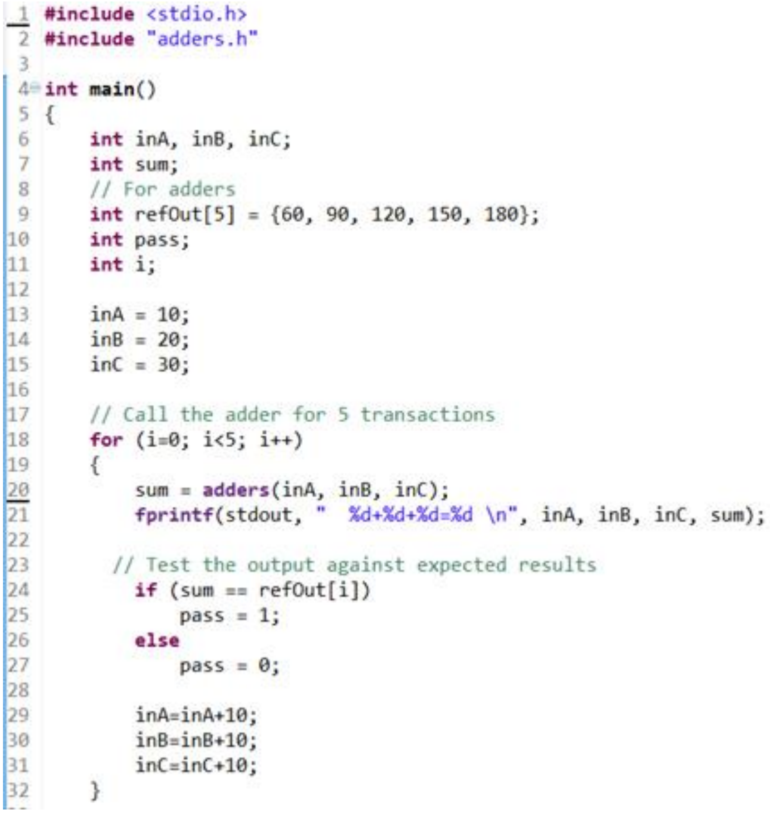
```c
#include "adders.h"

int adders(int in1, int in2, int in3) {
#pragma HLS INTERFACE ap_ctrl_none port=return

// Prevent IO protocols on all input ports
#pragma HLS INTERFACE ap_none port=in3
#pragma HLS INTERFACE ap_none port=in2
#pragma HLS INTERFACE ap_none port=in1

    int sum;

    sum = in1 + in2 + in3;

    return sum;

}
```

Şekil 2.2 : Vivado HLS ile oluşturulan bir C kodu.

Projenin kaynaklarına eklenmiş olan Şekil 2.2'deki C kodu tam sayı olan in1, in2 ve in3 değişkenlerinin değerlerini toplayarak sum isimli değişkene atamaktadır. Vivado HLS'te herhangi bir kod yazılırken aslında sayısal bir tasarım yapıldığı gözetilerek çalışılması programın verimliliği ve çalışabilirliği açısından oldukça önemlidir.

Vivado HLS aracı tasarım yaparken C'de yazılmış olan her bir fonksiyonu aslında bir RTL bloğu olarak modellemektedir. Bir fonksiyon içerisinde çağrılan bir başka fonksiyonu ise VHDL'den örnek vermek gerekirse entity içerisinde bir component olarak çağırmaktadır.

Şekil 2.2'de görülmekte olan farklı #pragma ifadeleri ise HLS içerisinde bulunan ve optimizasyon için kullanılan direktifleri belirtmektedir. Bunlara daha sonra kısaca değinilecektir.



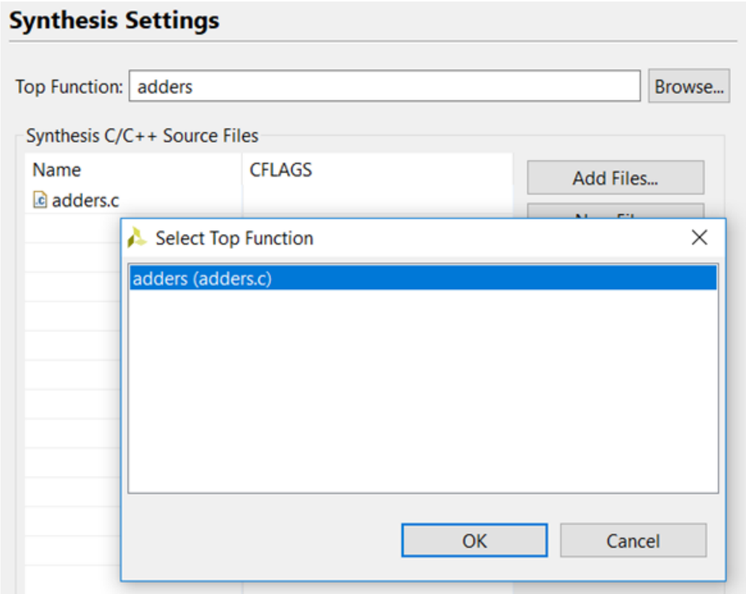
```c
#include <stdio.h>
#include "adders.h"

int main()
{
    int inA, inB, inC;
    int sum;
    // For adders
    int refOut[5] = {60, 90, 120, 150, 180};
    int pass;
    int i;

    inA = 10;
    inB = 20;
    inC = 30;

    // Call the adder for 5 transactions
    for (i=0; i<5; i++)
    {
        sum = adders(inA, inB, inC);
        fprintf(stdout, "  %d+%d+%d=%d \n", inA, inB, inC, sum);

      // Test the output against expected results
        if (sum == refOut[i])
            pass = 1;
        else
            pass = 0;

        inA=inA+10;
        inB=inB+10;
        inC=inC+10;
    }
```

Şekil 2.3 : Projeye eklenmiş bir C kodu.

Şekil 2.3'de projeye eklenmiş ve C'de yazılmış olan test kodu görülmektedir. HLS'te yazılan test kodlarının çalışabilmesi için main fonksiyonunun açılması gereklidir. Ardından kullanılan değişken tanımlamaları ve kod tarafından üretilen sonuçlar ile teorik olan sonuçların karşılaştırılabilmesi maksadıyla teorik sonuçlar dizi olarak tanımlanır.

Bu örnekte 3 adet değişken birbiriyle 5 kere toplanacak ve her bir toplama işleminin sonucunda giriş değişkenlerinin sayı değerleri teker teker 10 ile toplanarak arttırılacaktır. Bu test metoduna uygun olarak bir döngü kurulmuş ve istenen durumlar ifade edilmiştir. Vivado HLS aracının benzetim sonucunun doğru olduğunu anlaması için test fonksiyonunun geri dönüş değerinin '0' olması gerekmektedir. Tam tersi durumda, yani deneysel sonucun teorik sonuç ile uyuşmaması durumunda ise fonksiyonun geri dönüş değeri '1' olarak verilmelidir. Bu kurala uyulmadığı takdirde herhangi bir sorun çıkmayacak fakat HLS aracı benzetim sonucu doğru olsa bile yanlış olduğunu kabul edecektir.

Tasarımın kaynak kodu ve test kodunun yazımı tamamlandıktan sonra araç çubuğundaki *New Solution*’a tıklanarak çözümün üretilmesinin istendiği FPGA platformu listeden seçilir ve kullanılacak olan saat frekansı seçilerek devam edilir. Bu adım tamamlandıktan sonra sol tarafta yer alan *Explorer* sekmesinde *solution1* dosyası açılacaktır.

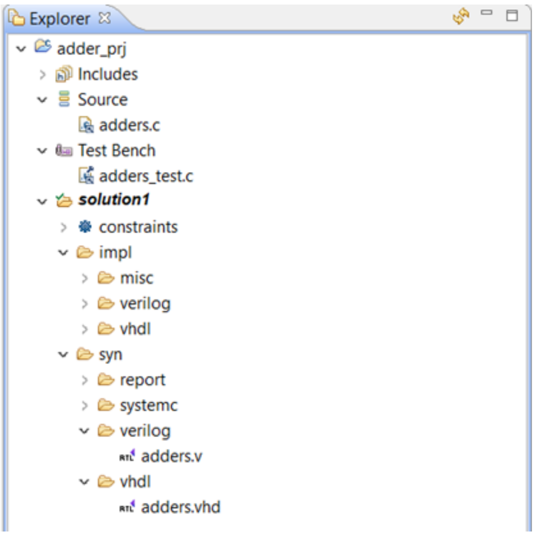


Top Function: adders

Select Top Function: adders (adders.c)

Şekil 2.4 : Üst fonksiyon seçimi.

Diğer sayısal tasarım programlarından aşina olunacağı üzere tasarımın sentezlenebilmesi için bütün modüllerin yer aldığı ve bağlantıların yapıldığı bir üst fonksiyonun seçilmesi gereklidir. Bunun için araç çubuğundaki *Project Settings ->Synthesis* kısmından Şekil 2.4’teki gibi üst fonksiyon seçilmelidir. Tasarım birden fazla fonksiyon içerdiği zaman bunlar liste şeklinde gözükecektir. Vivado HLS’in yazılan C kodundan kendine has RTL şemasını çıkarması, Verilog ve VHDL kodlarını sentezlemesi ve kaynak kullanımı gibi bilgileri hesaplaması için araç çubuğundan *C Synthesis*’e tıklanır.

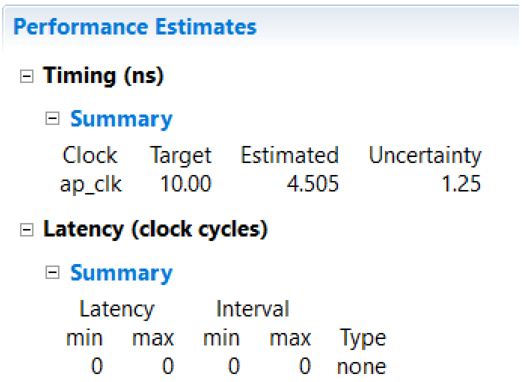


Explorer

- adder_prj
  - Includes
  - Source
    - adders.c
  - Test Bench
    - adders_test.c
  - ***solution1***
    - constraints
    - impl
      - misc
      - verilog
      - vhdl
    - syn
      - report
      - systemc
      - verilog
        - adders.v
      - vhdl
        - adders.vhd

Şekil 2.5 : *C Synthesis* sonucu elde edilen dosyalar.

Sentez işlemi gerçekleştirildikten sonra Şekil 2.5'te görüleceği üzere HLS ortamının sol tarafında yer alan *Explorer* sekmesinde *solution1* altında gerçekleme ve sentezleme sonucunda oluşturulan raporlar ve HDL dosyaları incelenebilir.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	4.505	1.25

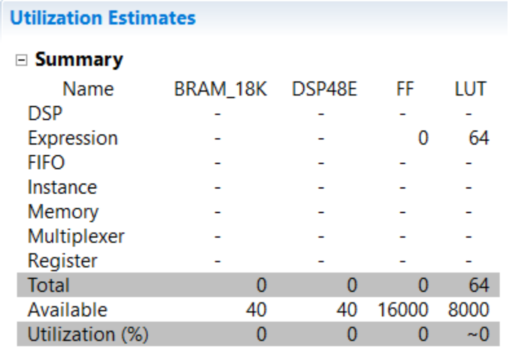
Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Şekil 2.6 : Sentez sonrası performans raporu

Sentez sonrasında Vivado HLS aracı tasarımın raporlarını bir sekme olarak açacaktır. Şekil 2.6'da görüleceği üzere *Performance Estimates* altında en başta belirlenen saat frekansının tasarıma uygun olup olmadığı, uygunsa da yaklaşık olarak hangi frekansta da çalışabileceği gösterilmektedir. Alt tarafta bulunan *Latency* kısmında ise işlemin tamamlanması için gerekli olan toplam saat darbesi bilgisi gösterilmektedir. Örnek tasarımda sadece toplama işlemi yapıldığı için değerler 0 olarak sonuçlanmıştır. Bunun sebebi işlemin tek bir saat darbesi içinde gerçekleştirilmesidir.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	64
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	64
Available	40	40	16000	8000
Utilization (%)	0	0	0	~0

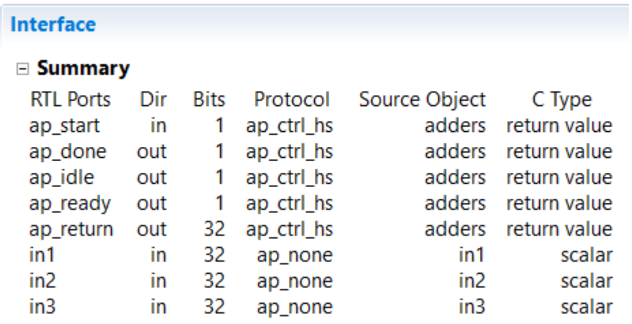
Şekil 2.7 : Sentez sonrası kaynak kullanımı raporu

Ayrıca Şekil 2.7'deki *Utilization Estimates* kısmı incelendiğinde FPGA içerisinde bulunan kaynakların ne kadarının kullanıldığının hem sayısal hem de yüzdesel bilgisi verilmektedir. Tasarım örneğinde görüleceği üzere 64 adet tablo çizelgesi (look-up table - LUT) kullanılmıştır.

Bu bölümün alt kısmında yer alan *Interface* kısmı incelendiğinde ise tasarımda bulunan giriş ve çıkış portlarının hangi protokole göre tanımlandığı, veriyolu genişlikleri, giriş-çıkış yapıları ve isimleri görülebilir. Bu örnek tasarımda;

- **#pragma** HLS INTERFACE ap_none port=in1, in2, in3
- **#pragma** HLS INTERFACE ap_ctrl_none port=**return**

ifadeleri kullanıldı.



Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_start	in	1	ap_ctrl_hs	adders	return value
ap_done	out	1	ap_ctrl_hs	adders	return value
ap_idle	out	1	ap_ctrl_hs	adders	return value
ap_ready	out	1	ap_ctrl_hs	adders	return value
ap_return	out	32	ap_ctrl_hs	adders	return value
in1	in	32	ap_none	in1	scalar
in2	in	32	ap_none	in2	scalar
in3	in	32	ap_none	in3	scalar

Şekil 2.8 : Sentez sonrası giriş-çıkış arayüzleri raporu

İlk ifadenin amacı giriş portlarının HLS tarafından otomatik olarak bir arayüz olarak tanımlanmasını engellemek ve tanımsızlaştırmaktır. İkinci ifade ise HLS'in tasarıma kontrol portlarını eklemesini engellemek amacıyla eklenmiştir. Çünkü koda herhangi bir direktif eklenmediği takdirde standart olarak kontrol arayüzleri eklenecektir. Şekil 2.8'de görüldüğü üzere direktifin eklenmediği takdirde *ap_start* ve *ap_done* gibi sinyaller otomatik olarak eklenmiştir. Bu arayüzlerin eklenmesindeki maksat tasarlanmış olan modülün HLS ortamı dışına çıkarılıp kullanıldığı zaman kontrolünün sağlanması içindir.

2.1.1.2 Direktifler

ARRAY RESHAPE: 8 bitlik diziler 128-bit bloklara yetmediği için bu direktif kullanılarak 128 bit olacak hale dönüştürülmüşlerdir. Bu sayede tek saat döngüsünde 8 bit yerine 128 bitlik veri işlenebilmiştir.

UNROLL: Bu direktif kullanılarak sıralı işlem yapılması yerine farklı döngülere ait operasyonların paralel çalışması sağlanmıştır.

INLINE: Bu direktif alt fonksiyonları tek bir üst fonksiyona entegre ederek gecikmenin azaltılmasını sağlamıştır.

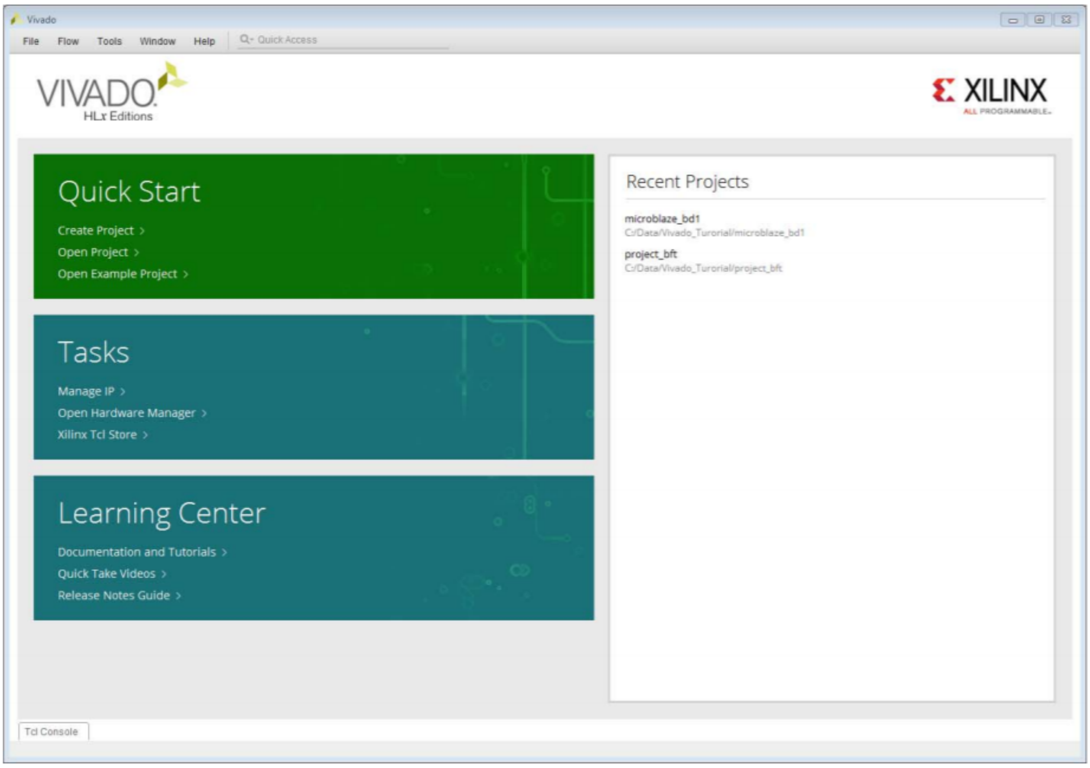
RESOURCE: Kullanılacak kaynakları belirtmek için kullanılmıştır.

INTERFACE: İstenilen şekilde kullanılabilmesi için giriş ve çıkış protokollerini düzenlemeye yarar.

STREAM: AXI4-Stream arayüzünün kullanıldığı portlarda, ilgili portun FIFO'ya bağlanılacağı ya da FIFO olarak sentezleneceği düşünülerek, derinliğin belirlenmesini sağlayan bir direktiftir.

2.2 Xilinx Vivado Ortamı

Vivado sayısal sistem ve donanım tasarımı yapılması için üretilmiş ve 2012 yılında ortaya çıkmış bir geliştirme ortamıdır [18]. Vivado Artix®-7, Kintex®-7, Kintex UltraScale™, Zynq®-7000 üzerinde donanım gerçeklenmesine imkan veren ve arayüzü Şekil 2.9'da gösterilen bir programdır. Eski model FPGA'ler Xilinx ISE [18] adlı program ile kullanılabilirken az önce bahsi geçen FPGA çeşitleri ise sadece Vivado ile kullanılabilmektedir. Vivado ortamında gerekli sayısal modüller Verilog, VHDL gibi donanım tanımlama dilleri ile tasarlanıp bu diller ile yazılan test kodları ile benzetimleri yapılabilmektedir. Modüllerin sentezleme, implementasyon ve FPGA'in programlanması için gerekli olan *bitstream* dosyası Vivado aracılığıyla oluşturulabilmektedir. Vivado geliştirme ortamında blok şeması ile de sistem tasarımı yapılabilmektedir. Bu şekilde yapılan tasarım kullanıcıya görsellik sağlayarak tasarım kolaylığı yaratır. Vivado ortamında varsayılan olarak tanımlı veya kullanıcı tarafından sonradan eklenen modül blokları kullanılabilmektedir. Kullanıcı tarafından modül bloğu oluşturulabilmesi için gerekli olan araçlar Vivado ortamında bulunmaktadır. Ayrıca tasarlanan sistemin başka ortamlara (yazılım geliştirme kiti (Software Development Kit – SDK) vb.) geçişi sağlanabilmektedir.



Şekil 2.9 : Vivado geliştirme ortamı.

Yapılan projede HLS ortamında tasarlanan fonksiyonların sistem üzerinde gerçeklenmesi, SDK ortamına atılabilmesi, benzetimlerinin yapılabilmesi, yapılan tasarımların FPGA üzerinde gerçeklenmesi için Vivado ortamı kullanılmıştır [18].

2.3 AXI4 Arayüzü

Gelişmiş Genişletilebilir Arayüz (Advanced eXtensible Interface - AXI), 1996 yılında geliştirilen bir mikrokontrolör veri yolu ailesi olan ARM Gelişmiş Mikrokontrolör Veri yolu Mimarisi (Advanced Microcontroller Bus Architecture - AMBA)'nin bir parçasıdır [15]. 2003'te yayımlanana AMBA 3.0'da ilk kez yer alan AXI arayüzü 2010'da yayımlanan AMBA 4.0'da AXI ve AXI4 arayüzlerinin ikinci versiyonları ile mevcuttur [15]. AXI arayüzünün kullanım amacı, genel olarak mikroişlemci ve diğer elemanlar arasındaki veri haberleşmesini sağlamaktır. AXI arayüzünün 3 adet türü vardır. Bunlar;

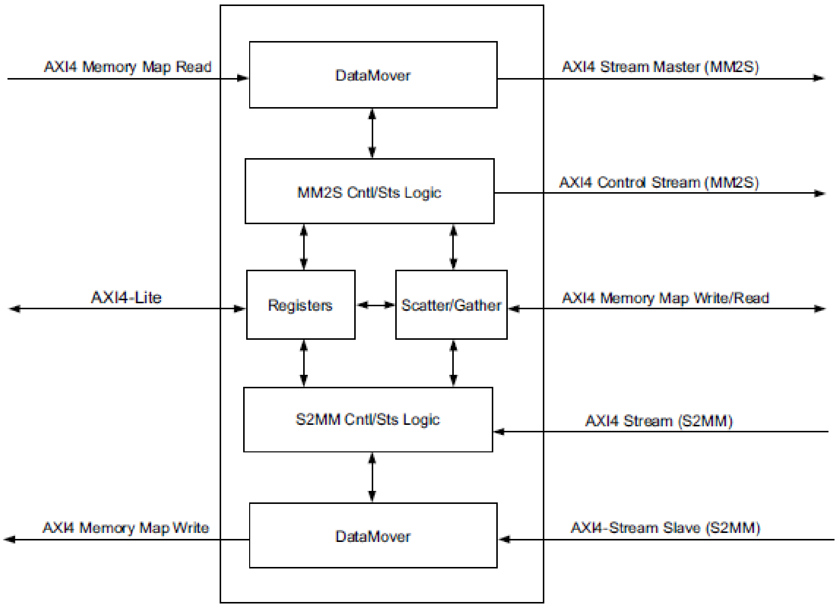
- AXI4
- AXI4-Lite
- AXI4-Stream

AXI4 yüksek performanslı bellek haritalı sistem gereksinimlerde, AXI4-Lite basit ve düşük hacimli veri haberleşmesinde AXI4-Stream ise yüksek hızlı sürekli veri iletiminde kullanılır [15]. AXI, genel olarak komut veren ana (master) ve bir uydu (slave) arasındaki veri alışverişini sağlayan bir arayüzü tanımlar. Fakat Xilinx bünyesindeki modül blokları sayesinde bu protokol birden fazla master ve slave arasında gerçekleşebilir [15].

AXI4 arayüzünün geliştirilmesinin ana amacı gereksiz veri yolu trafiğini azaltmaktır. Çoğuşmalı yöntem (Burst Mode) sayesinde gönderilen bütün veriler tek bir adres bilgisi ile gönderilir. Slave elemanı ise bu adres bilgisini referans alarak sonraki verilerin adresini kendi hesaplar, veri yolu trafiği bu şekilde azaltılır [15]. AXI4 arayüzü tek bir adres ile 256 veri transferi yapabiliyorken AXI4-Lite için bu sayı 1, AXI4-Stream için sınırsızdır [15].

2.4 Direkt Hafıza Erişimi (Direct Memory Access - DMA)

Direkt Hafıza Erişimi (Direct Memory Access - DMA), Vivado ortamında tanımlanan modül blokları kısmında tanıtılan bir modüldür. AXI DMA, AXI4-Stream arayüzlü çevresel birimler ile hafıza elemanları arasında yüksek bant genişlikli veri transfer olanağı sağlar [17]. Modül yapısı Şekil 2.10'da gösterilmiştir.



Şekil 2.10 : AXI DMA modül yapısı [17].

AXI4 DMA, bellekte kayıtlı olan verilerin bit genişliğine bakmaksızın 8 bitlik okuma ve yazma yapmaktadır. Bu özelliğinden dolayı transfer işleminin doğru gerçekleşmesi için DMA modülüne istenilen bit uzunluğu verisi girilmelidir. DMA üzerinden her veri transferi yapılacağı zaman gönderilecek verinin bit uzunluğu da belirtilmelidir. DMA okuduğu 8 bitlik veri paketlerini belirtilen bit genişliğine göre ayarlayarak veri yollarına bu birleşmiş veri paketlerini gönderir. Projede gerçekleşen bit transfer genişliği 32 bit seçildiği için DMA modülü 8'er bitlik 4 paket verisini okuyarak bunları birleştirip 32 bit genişliğinde veri yollarına aktarmaktadır. Bu veri paketleri 32 bit genişliğinde birleştirilirken bunların sırası alıcı modülün girişine hangi sırada verilmek istendiğine bağlı olarak ayarlanır.

AXI4 DMA yaptığı transferlerin sonunda *last* sinyalini kullanır. Bu sinyal, belirtilen bit genişliğinde transferin tamamlandığını göstermektedir. Son paket alıcıya gönderildiği zaman modül bu sinyali aktif hale getirmektedir. Okuma sırasında DMA modülü okumanın tamamlandığını anlaması için aldığı son bit paketi ile birlikte bu sinyalin aktif olup olmadığını kontrol eder. *last* sinyalinin alınmadığı durumda sistemin doğru çalışmadığı gözlemlenmiştir.

Sistemde AXI4 DMA kullanılması, işlemcinin iş yükünü azalttığından sistemin daha performanslı çalışmasını sağlar [17]. AXI4-Stream arayüzü 8, 16, 32, 64, 128, 256, 512, 1024 bit bit genişliğini desteklemektedir [17].

3. RSA ALGORİTMASI

3.1 Giriş

RSA algoritması 1977 yılında Rivest, Shamir ve Adleman tarafından geliştirilmiş ve geliştiricilerinin soyadlarının baş harfleri ile isimlendirilmiş bir kriptografi algoritmasıdır [14]. RSA, asimetrik bir şifreleme algoritması olması sebebiyle simetrik şifreleme algoritmalarına göre yavaş kalmakta fakat büyük sayılarla işlem yaptığı için daha güvenli bir kriptografi sistemi sunmaktadır.

3.2 Matematiksel Altyapı

RSA algoritması ile bir mesajı şifreleme ve şifreli mesajı çözme işlemi şu adımlarla gerçekleştirilmektedir:

Şifrelenecek mesaj M olarak ifade edilsin ve bu veri 0 ile $N-1$ arasında bir tam sayıdır. (E, N) çifti paylaşılan açık anahtarı göstermek üzere şifreleme işlemi aşağıdaki yapılır.

$$C = M^E (mod\ N) \tag{3.1}$$

Denklem 3.1 ile şifrelenmiş veri ifade edilir. Şifre çözme işlemi de aşağıdaki gibi ifade edilir.

$$M = C^D (mod\ N) \tag{3.2}$$

Denklem 3.2'deki (D, N) çifti, kullanıcıya özel üretilen gizli anahtarı ifade eder. İşlemler sonucunda C şifrelenmiş verisi, orijinal mesaj verisinin bulunduğu $0 \leq M \leq N - 1$ tam sayı aralığında kalmaktadır.

Yukarıda şifreleme ve çözme işlemleri sırasında kullanılan genel anahtar (E, N) ve gizli anahtar (D, N) çiftlerinin üretilmesi için ilk olarak N değişkeni, oldukça büyük ve herkesten gizli tutulan rastgele p ve q asal sayılarının çarpımı olarak hesaplanır ve paylaşılır.

$$\emptyset(N) = (p-1).(q-1) \tag{3.3}$$

Ardından denklem 3.3'te hesaplanan $\emptyset(N)$ ifadesi ile aralarında asal olacak büyük bir d tam sayısı seçilir. Seçilen bu D sayısı ve N, gizli anahtar çiftini (D, N) oluşturur. Genel anahtar çiftinde (E, N) bulunan E sayısı ise D'nin $mod\ \emptyset(N)$ de çarpmaya göre tersi olarak ifade edilir ve denklem 3.4 çözülerek elde edilir.

$$E.D \equiv 1\ mod\ \emptyset(N) \tag{3.4}$$

Güvenliğin arttırılması için bu adımların en başında seçilen p ve q asal sayılarının oldukça büyük seçilmesi ile $p.q$ çarpımı sonucunda bulunan n sayısını çarpanlarına ayırmanın hesaplama ile mümkün olmaması sağlanmalıdır [14].

Anahtarların üretimi sağlandıktan sonra en genel hali ile bir gönderici Alice ile alıcı Bob arasında şifrelenmiş iletişim şu şekilde sağlanır:

Şifreleme sürecinde gönderici Alice'in yapacağı işlemler:

- Bob'un kendisi için önceden hesapladığı genel anahtarı kanaldan alır,
- Mesaj verisini ve genel anahtarı şifreleme fonksiyonuna sokar,
- Denklem 3.1 ile şifrelenmiş veriyi Bob'a gönderir.

Şifre çözümleme sürecinde Bob'un yapacağı işlemler:

- Bob kendi gizli anahtarını hesaplar,
- Şifrelenmiş C verisini ve gizli anahtarını şifre çözme fonksiyonuna sokar,
- Denklem 3.2 ile çözümlenmiş olan orijinal M'yi elde eder.

Aynı işlemler, Bob Alice'e veri göndereceği zaman da tekrarlanır.

3.2.1 Kare alma ve çarpma metodu

RSA algoritmasının kullanıldığı kriptografi sistemlerinde temel olarak yapılan matematiksel hesaplamalar üstel modüler işlemlerdir. Denklem 3.1 ele alındığında herhangi bir metot kullanılmadan sonuç hesaplanırsa, $E - 1$ kere modüler çarpım işlemi yapılmalıdır. Fakat E sayısının çok büyük olduğu durumlarda işlem süresi çok uzayacaktır ki sistemin güvenli olması için bu sayının büyük seçilmesi gereklidir.

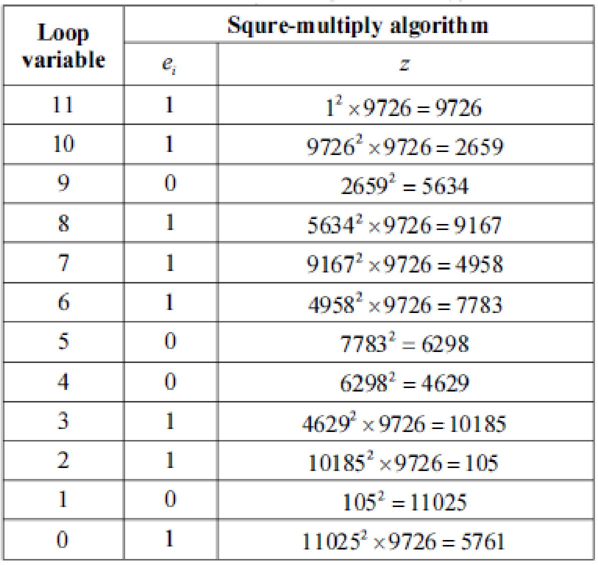
Bu algoritmada üs olan E'nin ikilik sayı tabanında;

$$E = \sum_{i=0}^{b-1} e_i . 2^i \tag{3.5}$$

olarak ifade edilir. Denklem 3.5'teki b sayısı, N tam sayısının ikilik tabanda içerdiği bit sayısıdır. Denklem 3.1 aşağıdaki adımlar takip edilerek gerçekleştirilebilir [8]:

```
Kare alma ve çarpma algoritması:
GİRİŞ: $M = (m_{b-1} \dots m_1 m_0)_2,\ E = (e_{b-1} \dots e_1 e_0)_2$
ÇIKIŞ: $C = M^E\ (mod\ N)$
1. $C \leftarrow 1$.
2. For $i$ from $b-1$ downto 0 do:
    2.1. $C \leftarrow C^2 (mod\ n)$
    2.2. If $e_i = 1\ then$
    2.3. $C \leftarrow C.M\ (mod\ n)$
3. Return (C)
```

Adım **2.2** incelendiğinde üs değeri olan E sayısının içinde bulunan 1 sayısı kadar modüler çarpım işlemi yapılacaktır. Buradan da çıkarılabilir ki minimum b kadar, maksimum da $2.b$ kadar modüler çarpma işlemi yapılacaktır. Bu sayede geleneksel üstel modüler hesabın doğurduğu işlem yükünün önüne geçilerek daha kısa sürede sonuç elde edilebilecektir [8].



Loop variable	Squre-multiply algorithm	
	e_i	z
11	1	$1^2 \times 9726 = 9726$
10	1	$9726^2 \times 9726 = 2659$
9	0	$2659^2 = 5634$
8	1	$5634^2 \times 9726 = 9167$
7	1	$9167^2 \times 9726 = 4958$
6	1	$4958^2 \times 9726 = 7783$
5	0	$7783^2 = 6298$
4	0	$6298^2 = 4629$
3	1	$4629^2 \times 9726 = 10185$
2	1	$10185^2 \times 9726 = 105$
1	0	$105^2 = 11025$
0	1	$11025^2 \times 9726 = 5761$

Şekil 3.1 : Left ro Right Modular Exponantiation için bir örnek.

Şekil 3.1'de $N = 11413, E = 3533$ ve $M = 9726$ için bir örnek verilmiştir. Görüleceği üzere geleneksel hesaplama ile 3532 adımda tamamlanması gereken işlem, 12 adımda tamamlanmıştır. Sonuç olarak da mesajın şifrelenmiş hali $C = 5761$ olarak bulunmuştur [8].

3.2.2 Modüler çarpım algoritması

Square and multiply method algoritmasında yer alan, Adım **2.1** ve **2.3**'teki modüler çarpım işlemleri için, işlem sayısını azaltmak adına aşağıda gösterilen şekilde bir modüler çarpım algoritması kullanılabilmektedir. Denklem 3.6'daki modüler işlem için, $B=(b_{n-1},\ldots,b_1,b_0)_2$ olmak üzere aşağıdaki adımlarla algoritma gerçekleştirilmektedir:

$$C = A * B \ (mod\ M) \tag{3.6}$$

Modüler çarpım algoritması:

GİRİŞ: $M = (m_{n-1} \ldots m_1 m_0)_2,\ A = (a_{n-1} \ldots a_1 a_0)_2,\ B = (b_{n-1} \ldots b_1 b_0)_2$

ÇIKIŞ: $C = A * B \ (mod\ M)$

```
1. $C \leftarrow 0$.
2. For $i$ from $n-1$ downto 0 do:
    2.1. $C \leftarrow C \ll 1$
    2.2. If $C \geq M\ then$
        2.2.1. $C \leftarrow C - M$
    2.3. If $b_i = 1$
        2.3.1. $C \leftarrow C + A$
        2.3.2. $C \geq M$
            2.3.2.1. $C \leftarrow C - M$
3. Return (C)
```

3.2.3 Montgomery modüler çarpım algoritması

Montgomery modüler çarpım algoritması, hem yazılım hem de donanım bakımından gerçeklemek için çok uygun ve işlem zamanı bakımından büyük avantajlar sağlayan bir algoritmadır [1]. Klasik modüler çarpma algoritmasında denklem 3.6 için k-bitlik A, B ve m sayıları göz önünde bulundurulduğunda; çarpma işlemi için k defa k-bitlik toplama işlemleri, k defa k-bitlik karşılaştırma işlemleri ve çıkarma işlemleri yapılması gerekmektedir. Fakat bu algoritma kullanıldığında yapılan işlem sayısı, k'nın 2'nin kuvvetleri eşdeğerinde bir sayıya bölümüne düşmesi sağlanır [1].

a bir tamsayı, A<N; N k-bitlik bir mod olmak üzere, eğer denklem 3.7'deki gösterim yapılırsa, A sayısı için a'nın, r'a göre N-rezidüsü olduğu söylenebilir:

$$\mathrm{AR} = \mathrm{A} * \mathrm{R} \bmod \mathrm{N}, \mathrm{R} = 2^{\mathrm{k}} \tag{3.7}$$

Aynı şekilde verilmiş bir B tamsayısı için b<N olmak üzere, denklem 3.8'deki gibi B sayısı için b'nin, r'a göre N-rezidüsü olduğu söylenebilir:

$$\mathrm{BR} = \mathrm{B} * \mathrm{R} \bmod \mathrm{N}, \mathrm{R} = 2^{\mathrm{k}} \tag{3.8}$$

A ve B'nin Montgomery çarpımları denklem 3.8'deki şekilde fade edilir:

$$\mathrm{R}' = \mathrm{AR} * \mathrm{BR} * \mathrm{R}^{-1} \bmod \mathrm{N} = \mathrm{A} * \mathrm{R} * \mathrm{B} * \mathrm{R} * \mathrm{R}^{-1} \bmod \mathrm{N} = \mathrm{A} * \mathrm{B} * \mathrm{R} \bmod \mathrm{N} \tag{3.8}$$

Görüldüğü üzere, R' sayısı a ve b sayılarının çarpımlarının moduna eşit değildir. Bu sebepten ötürü R' sayısı ile 1 rakamı tekrardan Montgomery işlemine sokularak A ve B sayılarının N'e göre mod alma işlemi sonucu denklem 3.9'dan elde edilmiş olunur.

$$(\mathrm{A} * \mathrm{B} * \mathrm{R}) * 1 * \mathrm{R}^{-1} \bmod \mathrm{N} = \mathrm{A} * \mathrm{B} \bmod \mathrm{N} \tag{3.9}$$

Montgomery algoritması kullanılacak modüler çarpma işlemleri için N sayısının her zaman tek sayı olması gerekmektedir. RSA algoritmasında kullanılacak olan N sayısı, iki adet asal sayının çarpımı ile elde edilen büyük bir sayı olduğundan ve tek bir sayı olduğundan, bu algoritma RSA gerçeklemesi için uygun bir yöntemdir [1].

$$S = A * B * R^{-1}\, mod\, N,\ R = 2^{k}\, mod\, N \tag{3.10}$$

Montgomery modüler çarpım algoritması ile k-bitlik A, B, R, N tamsayıları için denklem 3.10'un hesaplanması aşağıdaki adımlarla gerçekleştirilmektedir [11].

Montgomery modüler çarpım algoritması:

GİRİŞ: $N = (n_{k-1} \dots n_1 n_0)_2,\ A = (a_{k-1} \dots a_1 a_0)_2,\ B = (b_{k-1} \dots b_1 b_0)_2,\ R = 2^{k}\, mod\, N$

ÇIKIŞ: $S = A * B * R^{-1}\, mod\, N$

```
1. $S \leftarrow 0.$
2. For $i = 0$ to $k - 1$ do:
    2.1. $q \leftarrow (S + A * b_i)\ mod\ 2$
    2.2. $S \leftarrow (S + A * b_i + q * N)\ div\ 2$
3. $If(S \geq N)$
    3.1. $S \leftarrow S - N$
4. Return (S)
```

3.3 Tasarım ve Gerçekleme

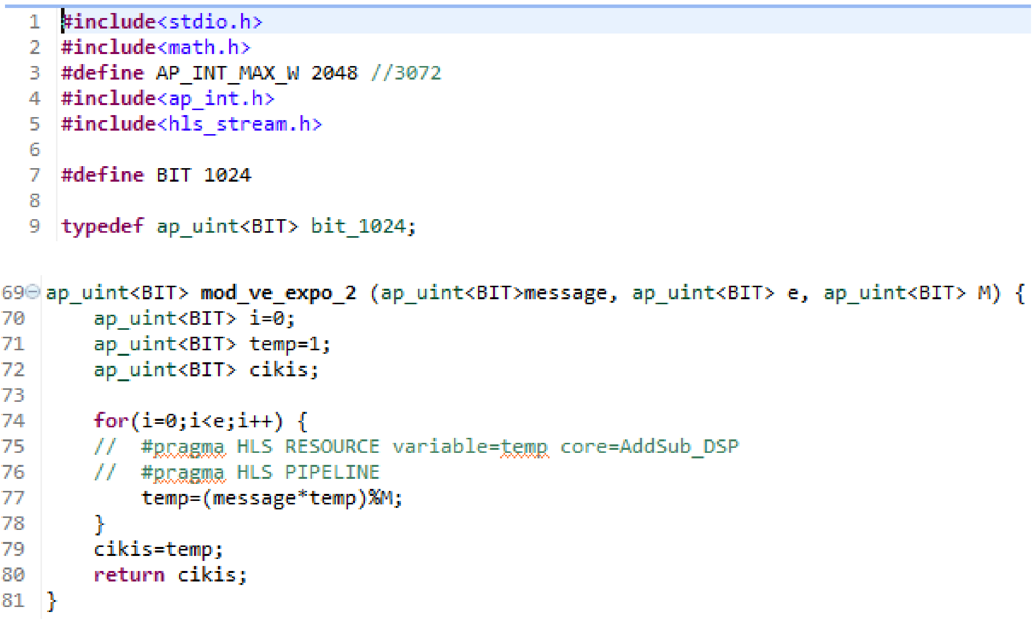
Kararlaştırılmış olan proje isterlerine göre, gerçeklenecek olan RSA algoritmasının 1024 bitlik olması istenmiştir. 1024-RSA algoritması için bir adet üst modül tasarlanmıştır. Modül tasarlanırken ilk olarak RSA'in gerçeklenmesi için gerekli olan algoritmalar, ayrı C++ fonksiyonları ile HLS ortamında tasarlanmış, sentezlenmiş ve test edilmiştir. Sentezlenen fonksiyonların kaynak kullanımlarına, işlem sürelerine bakılarak üst modül için kullanılması gereken fonksiyonlar seçilmiş ve RSA için gerekli olan işlemleri gerçekleyecek modül tasarlanmıştır. Sonrasında tasarlanmış olan RSA fonksiyonu, gerekli HLS direktifleri kullanılarak AXI4-Lite ve AXI4-Stream arayüzü ile tekrardan sentezlenmiştir.

3.3.1 RSA işlevi için tasarlanan fonksiyonlar

3.3.1.1 C++'taki işlem operatörleri kullanılarak mod ve üs alma işlemini yapan fonksiyon

Vivado HLS ortamında, öncelikle denklem 3.1'i (şifreleme ve şifre çözme fonksiyonu) gerçeklemek için bir C++ fonksiyonu yazılmıştır. Şekil 3.2'deki kodun başında, fonksiyonun çalışması için gerekli olan kütüphaneler eklenmiştir. Bu kütüphanelerden *ap_int.h* ve *hls_stream.h* kütüphaneleri, HLS ortamına özel kütüphanelerdir. *hls_stream.h* kütüphanesi bu fonksiyon için kullanılmamakla birlikte, ilerde anlatılacak fonksiyonlarda gerekli olduğu için yazılma gereği duyulmuştur. *ap_int.h* kütüphanesi, istenilen giriş, çıkışların ve yazmaçların, C++'ta tanımlı olan önceden tanımlı veri değişkenlerinin (*int, char* vb.) sabit veri uzunluklarından bağımsız şekilde, kullanıcı tarafından ilgili bit uzunluklarının sağlanması ve donanım tanımlama dillerindeki gibi bit manipülasyonu sağlayacak işlemlerin kolaylıkla yapılabilmesi için gereklidir. *ap_int.h* kütüphanesi kullanıldığında, varsayılan olarak kullanılabilinecek maksimum bit uzunluğu 1024 bit olarak tanımlıdır. Fakat Şekil 3.2'de görüldüğü üzere, *ap_int.h* kütüphanesinin tanımlı olduğu satırdan önce *AP_INT_MAX_W* değişkeni tanımlanarak maksimum bit uzunluğu istenilen sayıda arttırılabilir. Fakat Xilinx tarafından, maksimum bit sayısının kullanılmak istenenden daha fazla arttırılması önerilmemektedir [19].

Şekil 3.2'de görüldüğü üzere, HLS üzerinde istenen donanıma eşdeğer C++ fonksiyonu yazılırken, C++ dili ile tasarım esnasında kullanılan *main* fonksiyonu kullanılmamaktadır. *main* fonksiyonu sadece HLS üzerinde tasarlanan test dosyalarında kullanılabilmektedir.

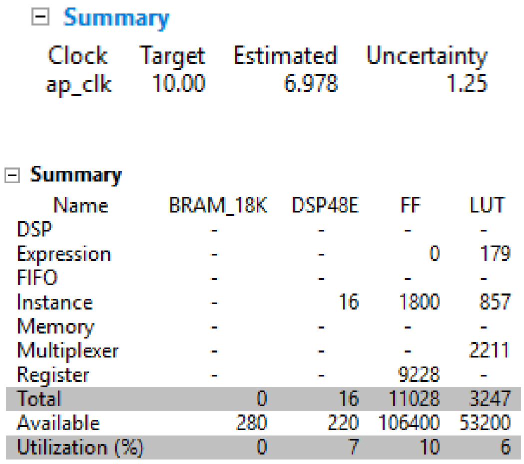


```
#include<stdio.h>
#include<math.h>
#define AP_INT_MAX_W 2048 //3072
#include<ap_int.h>
#include<hls_stream.h>

#define BIT 1024

typedef ap_uint<BIT> bit_1024;
```

```
ap_uint<BIT> mod_ve_expo_2 (ap_uint<BIT>message, ap_uint<BIT> e, ap_uint<BIT> M) {
    ap_uint<BIT> i=0;
    ap_uint<BIT> temp=1;
    ap_uint<BIT> cikis;

    for(i=0;i<e;i++) {
    //  #pragma HLS RESOURCE variable=temp core=AddSub_DSP
    //  #pragma HLS PIPELINE
        temp=(message*temp)%M;
    }
    cikis=temp;
    return cikis;
}
```

Şekil 3.2 : Operatörler ile gerçeklenen RSA fonksiyonu.

Şekil 3.2'de görüldüğü üzere, *return* ile değer döndüren bir fonksiyon tasarlanmıştır. *ap_uint<BIT>* olarak yazılanlar, BIT'e eşdeğer olan sayı kadar bit uzunluğuna sahip bir işaretsiz yazmaç oluşturmak için kullanılmıştır. Görüldüğü üzere, fonksiyonun 1024 bitlik işaretsiz bir çıkışı, üç adet de 1024 bitlik işaretsiz girişi bulunmaktadır. *For* döngüsü ile e sayısı kadar mesaj sayısı çarpılmış ve bunların M sayısına göre modu alınmıştır. Döngü sonunda ortaya çıkan sayı, *cikis* isimli bir yazmaça atanmış ve yazmaç *return* komutu ile çıkışa verilmiştir. İşlemler yapılırken çarpma işlemi için ve mod alma işlemi için, C++'ta tanımlı "*" ve "%" operatörleri kullanılmıştır.



Summary

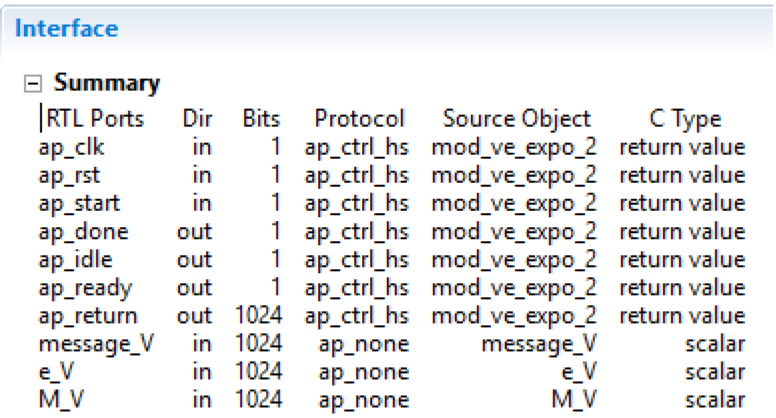
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.978	1.25

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	179
FIFO	-	-	-	-
Instance	-	16	1800	857
Memory	-	-	-	-
Multiplexer	-	-	-	2211
Register	-	-	9228	-
Total	0	16	11028	3247
Available	280	220	106400	53200
Utilization (%)	0	7	10	6

Şekil 3.3 : Sentez sonrası kaynak kullanımı raporu.

Şekil 3.3’den görüldüğü üzere, HLS ortamındaki sentez sonucu (Zedboard ZC702 kartı kullanıldığı varsayılarak), FPGA’in toplam kapasitesine göre yeterince az bir kaynak kullanılmıştır.



Interface

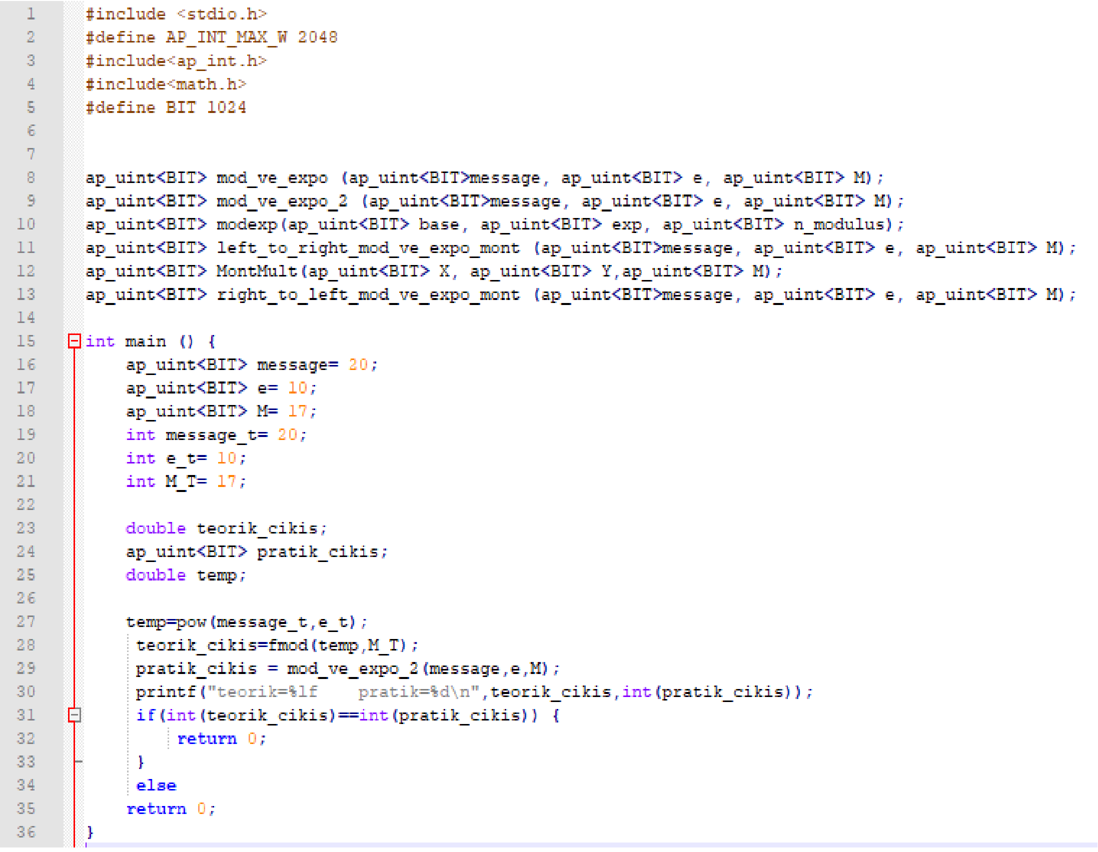
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	mod_ve_expo_2	return value
ap_rst	in	1	ap_ctrl_hs	mod_ve_expo_2	return value
ap_start	in	1	ap_ctrl_hs	mod_ve_expo_2	return value
ap_done	out	1	ap_ctrl_hs	mod_ve_expo_2	return value
ap_idle	out	1	ap_ctrl_hs	mod_ve_expo_2	return value
ap_ready	out	1	ap_ctrl_hs	mod_ve_expo_2	return value
ap_return	out	1024	ap_ctrl_hs	mod_ve_expo_2	return value
message_V	in	1024	ap_none	message_V	scalar
e_V	in	1024	ap_none	e_V	scalar
M_V	in	1024	ap_none	M_V	scalar

Şekil 3.4 : Sentezlenen modülün giriş ve çıkışları.

Şekil 3.4’te görüldüğü üzere fonksiyonda istenilen giriş ve çıkışlar başarı ile sentezlenmiş, ayrıca bunların haricinde farklı giriş ve çıkışlarda sentezlenmiştir. Sentezlenen bu giriş ve çıkışlar modülün kontrolü ve işlem durumu hakkında bilgi sahibi olmak için kullanılan, HLS tarafından otomatik olarak oluşturulan sinyallerdir. Varsayılan olarak bu sinyaller (modülü başlatma, modülü resetleme, ilgili saat sinyali girişi vb.) sentezlenmektedir fakat tasarımcı isterse, ilgili HLS direktiflerini kullanarak bu sinyalleri kaldırabilir, sınırlandırabilir ya da farklı arayüzler kullanarak (AXI4-Lite, *ap_fifo* vb.) tasarladığı modülü sentezleyebilir.

Tasarlanan fonksiyonun test edilmesi için, HLS’in sağladığı yenilikçi özelliklerden biri olan, C++ dili ile test fonksiyonu tasarlanabilmesi özelliğinden yararlanılmış ve tasarlanan test fonksiyonları ile sentez öncesi ve sonrası da benzetim yapma imkanı doğmuştur. Böylece C++ ile yazılmış test fonksiyonları, hem donanım yönünden (sentez sonrası) hem de yazılım yönünden (sentez öncesi) tek bir dosya ile kısa sürede benzetim yapmaya imkan sağlamıştır. HLS’in bu yönü, diğer donanım tanımlama dilleri ile yapılan tasarımların test benzetimlerine göre daha efektif çalıştığını göstermektedir. Fakat daha karmaşık tasarımlarda, donanım tanımlama dilleri ile test kodları yazıp benzetim yapmak, HLS üzerinde test fonksiyonları yazarak benzetim yapmaya göre daha optimize bir çözüm getirebilmektedir.



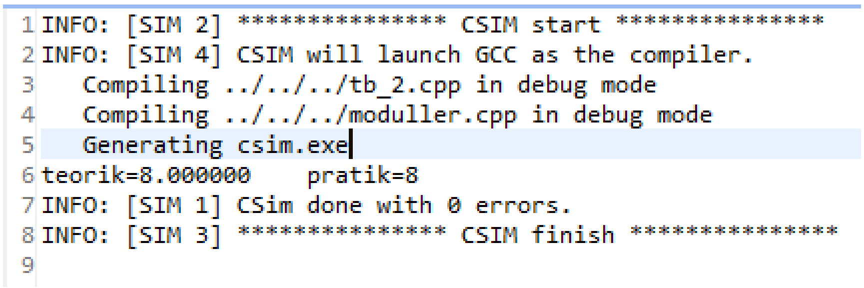
```
#include <stdio.h>
#define AP_INT_MAX_W 2048
#include<ap_int.h>
#include<math.h>
#define BIT 1024


ap_uint<BIT> mod_ve_expo (ap_uint<BIT>message, ap_uint<BIT> e, ap_uint<BIT> M);
ap_uint<BIT> mod_ve_expo_2 (ap_uint<BIT>message, ap_uint<BIT> e, ap_uint<BIT> M);
ap_uint<BIT> modexp(ap_uint<BIT> base, ap_uint<BIT> exp, ap_uint<BIT> n_modulus);
ap_uint<BIT> left_to_right_mod_ve_expo_mont (ap_uint<BIT>message, ap_uint<BIT> e, ap_uint<BIT> M);
ap_uint<BIT> MontMult(ap_uint<BIT> X, ap_uint<BIT> Y,ap_uint<BIT> M);
ap_uint<BIT> right_to_left_mod_ve_expo_mont (ap_uint<BIT>message, ap_uint<BIT> e, ap_uint<BIT> M);

int main () {
    ap_uint<BIT> message= 20;
    ap_uint<BIT> e= 10;
    ap_uint<BIT> M= 17;
    int message_t= 20;
    int e_t= 10;
    int M_T= 17;

    double teorik_cikis;
    ap_uint<BIT> pratik_cikis;
    double temp;

    temp=pow(message_t,e_t);
     teorik_cikis=fmod(temp,M_T);
     pratik_cikis = mod_ve_expo_2(message,e,M);
     printf("teorik=%lf    pratik=%d\n",teorik_cikis,int(pratik_cikis));
     if(int(teorik_cikis)==int(pratik_cikis)) {
         return 0;
     }
     else
    return 0;
}
```

Şekil 3.5 : Tasarlanan fonksiyon için test kodu.

Şekil 3.5’ten görüldüğü üzere, tasarlanan fonksiyonun doğru çalışıp çalışmadığının kontrol edilmesi için bir test dosyası yazılmıştır. Kodda görüldüğü üzere, benzetimi yapılmak istenen fonksiyonlar *main* fonksiyonundan önce belirtilerek, fonksiyonlar test dosyasına tanıtılmıştır. *main* fonksiyonunun içerisinde, öncelikle benzetimi yapılmak istenen fonksiyon için giriş değerleri belirlenmiştir. Değerler teorik olarak, C++ dilinde tanımlı olan *math.h* kütüphanesi içerisindeki fonksiyonlar ile normal bir

C++ fonksiyonu yazar gibi işlemler yapılarak *teorik_cikis* isimli yazmaca atılmış, daha sonra benzetimi yapılmak istenen fonksiyon çalıştırılarak, fonksiyonun sonucu *pratik_cikis* isimli bir yazmaca atanmıştır. Elde edilen iki adet çıkış, *printf* fonksiyonu ile yazdırılarak, sonuçlar kullanıcı tarafından gözlemlenebilmektedir. Ayrıca HLS'in tipik bir özelliği olarak, eğer *main* fonksiyonu, 0 değeri döndürürse, benzetim sonucunun başarılı olduğu anlaşılmakta, eğer sıfırdan farklı bir değer döndürürse benzetimin başarısız olduğu anlaşılmaktadır [19]. **Bölüm 3.3.1** içinde anlatılan denklem 3.1'i gerçekleyen fonksiyonlarda, Şekil 3.5'te gösterilen test dosyası kullanılmıştır.

message=20, *e*=10, *M*=17 değerleri için, tasarlanmış olan fonksiyonun benzetimi yapıldığında beklendiği şekilde Şekil 3.6'daki sonuçlar elde edilmiş ve fonksiyonun doğru çalıştığı gözlemlenmiştir.

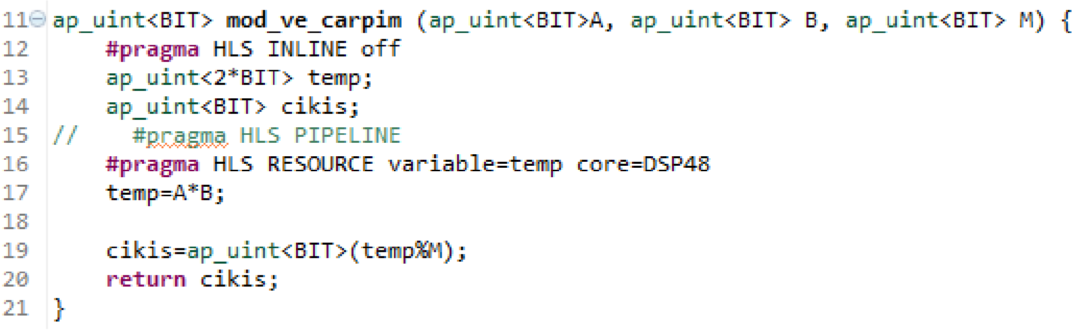


```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../tb_2.cpp in debug mode
   Compiling ../../../moduller.cpp in debug mode
   Generating csim.exe
teorik=8.000000    pratik=8
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Şekil 3.6 : C++ benzetim sonucu.

3.3.1.2 C++ işlem operatörleri kullanılarak tasarlanan modüler çarpım fonksiyonu

Bölüm 3.2.1'de anlatılan kare alma ve çarpma metodu algoritmasında kullanılan modüler çarpım işlemi, Şekil 3.7'de görülen fonksiyon ile HLS ortamında gerçeklenmiştir.

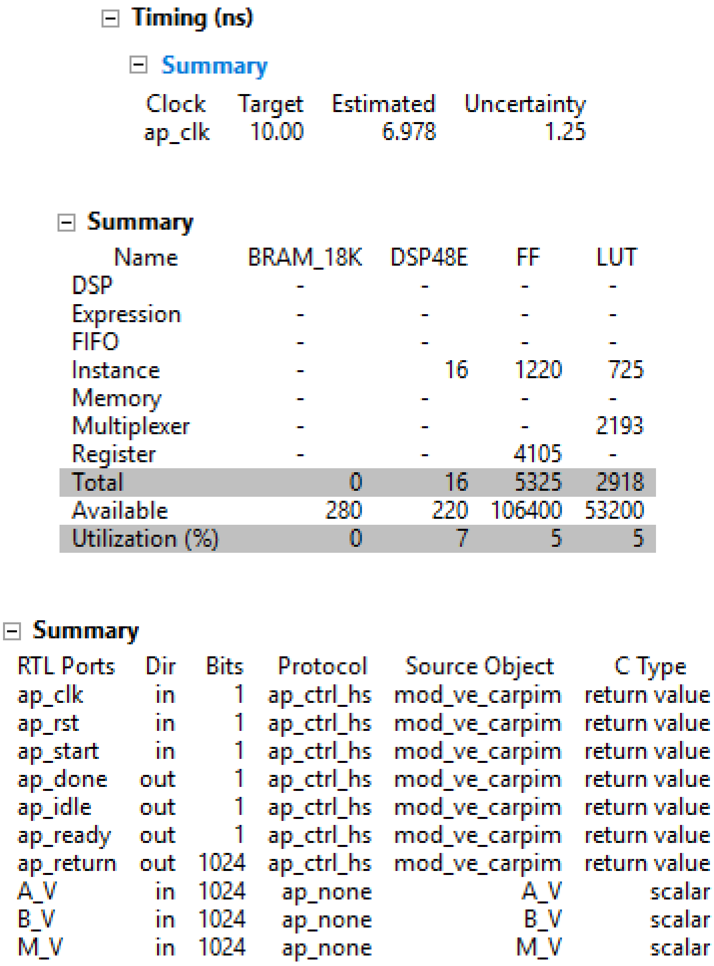


```
ap_uint<BIT> mod_ve_carpim (ap_uint<BIT>A, ap_uint<BIT> B, ap_uint<BIT> M) {
    #pragma HLS INLINE off
    ap_uint<2*BIT> temp;
    ap_uint<BIT> cikis;
//    #pragma HLS PIPELINE
    #pragma HLS RESOURCE variable=temp core=DSP48
    temp=A*B;

    cikis=ap_uint<BIT>(temp%M);
    return cikis;
}
```

Şekil 3.7 : Operatörler ile tasarlanan modüler çarpım fonksiyonu.

Şekil 3.7'de görüldüğü üzere, C++ dilindeki çarpma ve mod alma operatörleri kullanılarak modüler çarpım işlemi gerçeklenmiştir. HLS ortamının sağladığı *RESOURCE* direktifi kullanılarak, çarpma işlemlerinin DSP48 çarpıcıları ile sentezlenmesi istenmiştir.

Tasarlanan fonksiyon sentezlendikten sonra, Şekil 3.8'de görüldüğü üzere sentez raporu elde edilmiştir. Tasarımda hedeflenen 10 ns'lik saat periyodu, sentezlenen modül için kullanabilen bir aralıktadır.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.978	1.25

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	16	1220	725
Memory	-	-	-	-
Multiplexer	-	-	-	2193
Register	-	-	4105	-
Total	0	16	5325	2918
Available	280	220	106400	53200
Utilization (%)	0	7	5	5

Summary

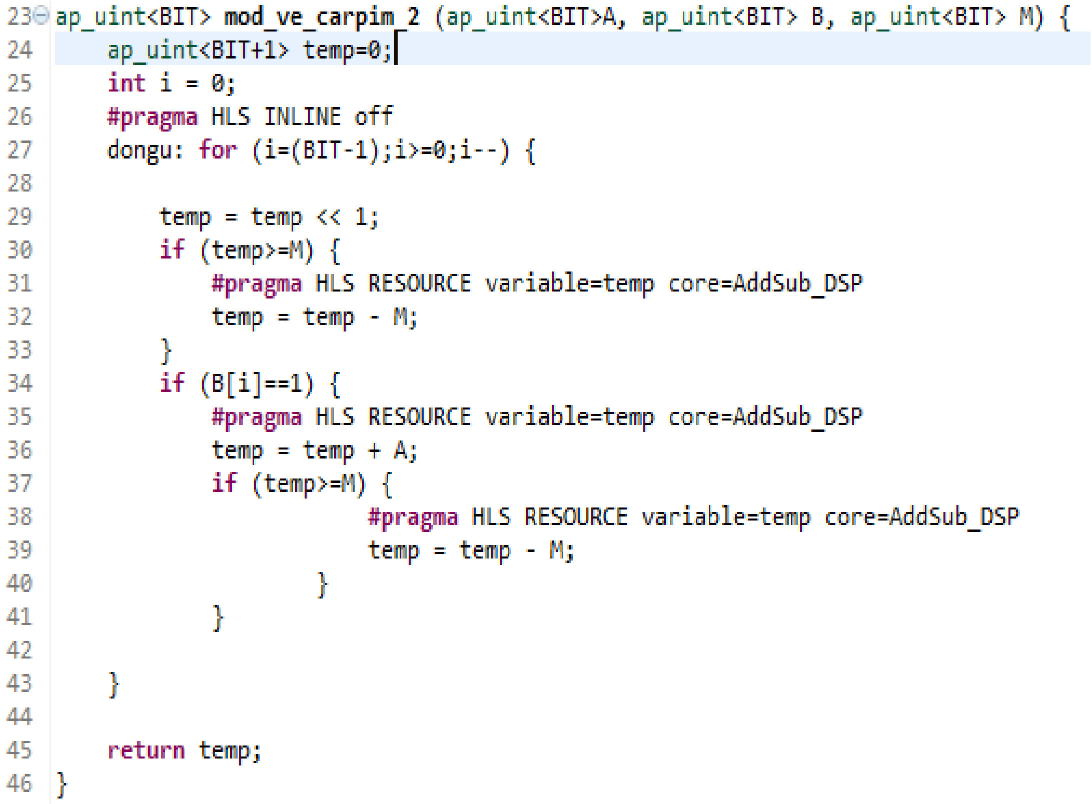
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	mod_ve_carpim	return value
ap_rst	in	1	ap_ctrl_hs	mod_ve_carpim	return value
ap_start	in	1	ap_ctrl_hs	mod_ve_carpim	return value
ap_done	out	1	ap_ctrl_hs	mod_ve_carpim	return value
ap_idle	out	1	ap_ctrl_hs	mod_ve_carpim	return value
ap_ready	out	1	ap_ctrl_hs	mod_ve_carpim	return value
ap_return	out	1024	ap_ctrl_hs	mod_ve_carpim	return value
A_V	in	1024	ap_none	A_V	scalar
B_V	in	1024	ap_none	B_V	scalar
M_V	in	1024	ap_none	M_V	scalar

Şekil 3.8 : Sentez sonrası raporu.

Doğru sentezlenen modül, yazılan test fonksiyonları ile test edilmiş ve girişlerine verilen sayılara göre doğru ve beklenen çıkışlara ulaşılmıştır.

3.3.1.3 Modüler çarpım algoritması kullanılarak tasarlanan modüler çarpım fonksiyonu

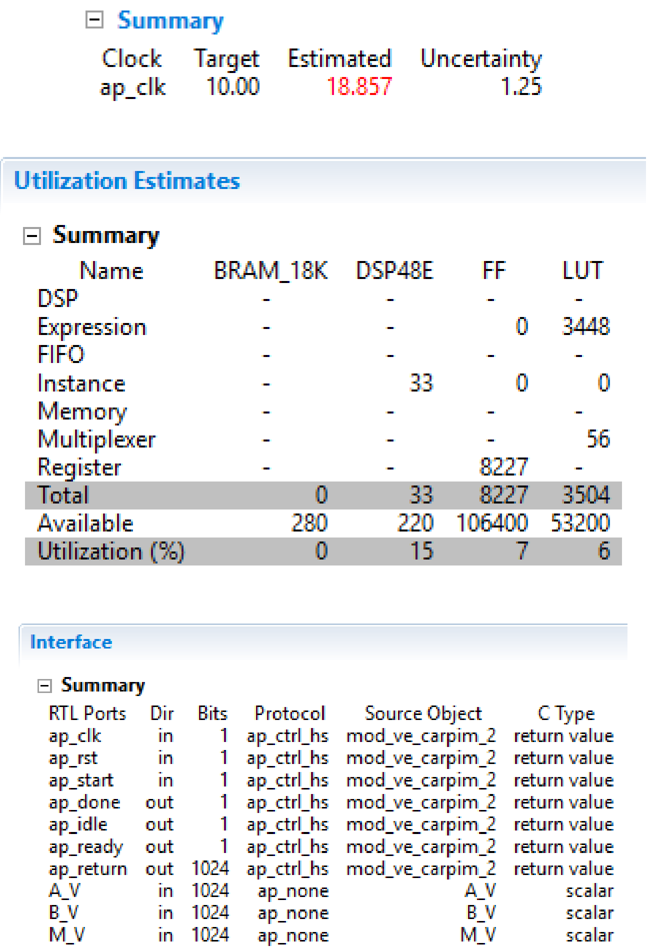
Bölüm 3.2.2’de anlatılan modüler çarpım algoritmasının adımları uygulandığında, Şekil 3.9’da gösterildiği şekilde bir fonksiyon tasarlanmıştır. Görüldüğü üzere, modüler çarpma işlemi için çıkarma, toplama ve kaydırma işlemleri kullanılmıştır. HLS’in *RESOURCE* direktifleri kullanılarak, toplama ve çıkarma işlemlerinin DSP48 bileşenleri ile sentezlenmesi zorlanmıştır. *B[i]* komutu ile, B sayısının tek tek bitlerine erişilebilmektedir. Bu özellik, koda eklenen *ap_int.h* kütüphanesinden kaynaklanmaktadır.



```
ap_uint<BIT> mod_ve_carpim_2 (ap_uint<BIT>A, ap_uint<BIT> B, ap_uint<BIT> M) {
    ap_uint<BIT+1> temp=0;
    int i = 0;
    #pragma HLS INLINE off
    dongu: for (i=(BIT-1);i>=0;i--) {

        temp = temp << 1;
        if (temp>=M) {
            #pragma HLS RESOURCE variable=temp core=AddSub_DSP
            temp = temp - M;
        }
        if (B[i]==1) {
            #pragma HLS RESOURCE variable=temp core=AddSub_DSP
            temp = temp + A;
            if (temp>=M) {
                        #pragma HLS RESOURCE variable=temp core=AddSub_DSP
                        temp = temp - M;
                    }
            }

    }

    return temp;
}
```

Şekil 3.9 : Modüler çarpım algoritması ile tasarlanan modüler çarpım fonksiyonu.

Tasarlanan fonksiyon sentezlendikten sonra, Şekil 3.10’da görüldüğü gibi, sentez sonrası raporu elde edilmiştir. HLS ortamında, 10 ns’lik bir saat periyodu hedeflense de, sentez sonucu HLS tarafından öngörülen gecikmelerden dolayı en az 18.857 ns’lik bir saat periyodu kullanılması gerektiği belirtilmiştir.



⊟ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	18.857	1.25

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	3448
FIFO	-	-	-	-
Instance	-	33	0	0
Memory	-	-	-	-
Multiplexer	-	-	-	56
Register	-	-	8227	-
Total	0	33	8227	3504
Available	280	220	106400	53200
Utilization (%)	0	15	7	6

Interface

⊟ **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	mod_ve_carpim_2	return value
ap_rst	in	1	ap_ctrl_hs	mod_ve_carpim_2	return value
ap_start	in	1	ap_ctrl_hs	mod_ve_carpim_2	return value
ap_done	out	1	ap_ctrl_hs	mod_ve_carpim_2	return value
ap_idle	out	1	ap_ctrl_hs	mod_ve_carpim_2	return value
ap_ready	out	1	ap_ctrl_hs	mod_ve_carpim_2	return value
ap_return	out	1024	ap_ctrl_hs	mod_ve_carpim_2	return value
A_V	in	1024	ap_none	A_V	scalar
B_V	in	1024	ap_none	B_V	scalar
M_V	in	1024	ap_none	M_V	scalar

Şekil 3.10 : Sentez sonrası raporu.

Doğru sentezlenen modül, yazılan test fonksiyonları ile test edilmiş ve girişlerine verilen sayılara göre doğru ve beklenen çıkışlara ulaşılmıştır.

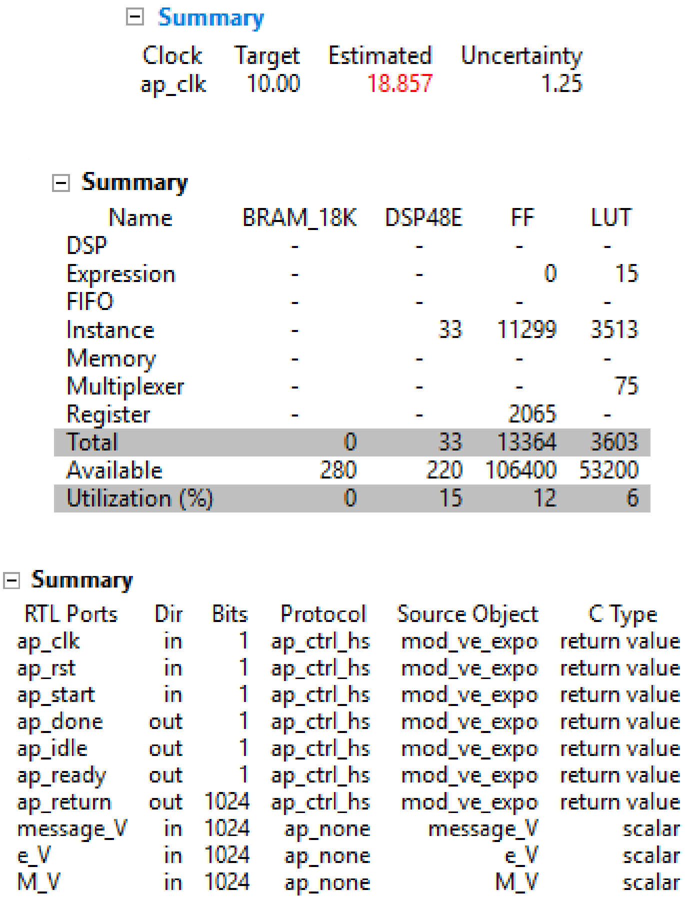
3.3.1.4 Kare alma ve çarpma metodu algoritması ile tasarlanan fonksiyon

Bölüm 3.2.1'de anlatılan kare alma ve çarpma metodu algoritması, **Bölüm 3.3.1.3**'de tasarlanmış modüler çarpım fonksiyonu ile denklem 3.1 işlemi için, Şekil 3.11'de görülen fonksiyon ile HLS ortamında gerçeklenmiştir.

```
ap_uint<BIT> mod_ve_expo (ap_uint<BIT>message, ap_uint<BIT> e, ap_uint<BIT> M) {
    ap_uint<BIT> temp=1;
    int i = 0;
//  ap_uint<BIT-8> zero = 0;
    dongu: for (i=(BIT-1);i>=0;i--) {
        //#pragma HLS PIPELINE
        temp = mod_ve_carpim_2(temp,temp,M);
        //temp = mod_ve_carpim(temp,temp,M);
        if (e[i]==1) {

            temp =  mod_ve_carpim_2(temp,(/*zero,*/message),M);
            //temp =  mod_ve_carpim(temp,(/*zero,*/message),M);
        }

    }

    return temp;
}
```

Şekil 3.11 : Kare alma ve çarpma metodu ile tasarlanan RSA fonksiyonu.

Tasarlanan fonksiyon sentezlendikten sonra, Şekil 3.12'de görüldüğü gibi, sentez sonrası raporu elde edilmiştir. HLS ortamında, 10 ns'lik bir saat periyodu hedeflense de, sentez sonucu HLS tarafından öngörülen gecikmelerden dolayı en az 18.857 ns'lik bir saat periyodu kullanılması gerektiği belirtilmiştir.



⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	18.857	1.25

⊟ **Summary**

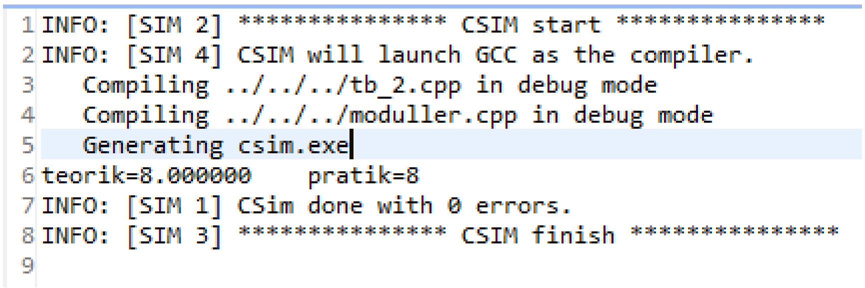
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	15
FIFO	-	-	-	-
Instance	-	33	11299	3513
Memory	-	-	-	-
Multiplexer	-	-	-	75
Register	-	-	2065	-
Total	0	33	13364	3603
Available	280	220	106400	53200
Utilization (%)	0	15	12	6

⊟ **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	mod_ve_expo	return value
ap_rst	in	1	ap_ctrl_hs	mod_ve_expo	return value
ap_start	in	1	ap_ctrl_hs	mod_ve_expo	return value
ap_done	out	1	ap_ctrl_hs	mod_ve_expo	return value
ap_idle	out	1	ap_ctrl_hs	mod_ve_expo	return value
ap_ready	out	1	ap_ctrl_hs	mod_ve_expo	return value
ap_return	out	1024	ap_ctrl_hs	mod_ve_expo	return value
message_V	in	1024	ap_none	message_V	scalar
e_V	in	1024	ap_none	e_V	scalar
M_V	in	1024	ap_none	M_V	scalar

Şekil 3.12 : Sentez sonrası rapor.

message=20, *e*=10, *M*=17 değerleri için, tasarlanmış olan fonksiyonun benzetimi yapıldığında beklendiği şekilde Şekil 3.13'teki sonuçlar elde edilmiş ve fonksiyonun doğru çalıştığı gözlemlenmiştir.

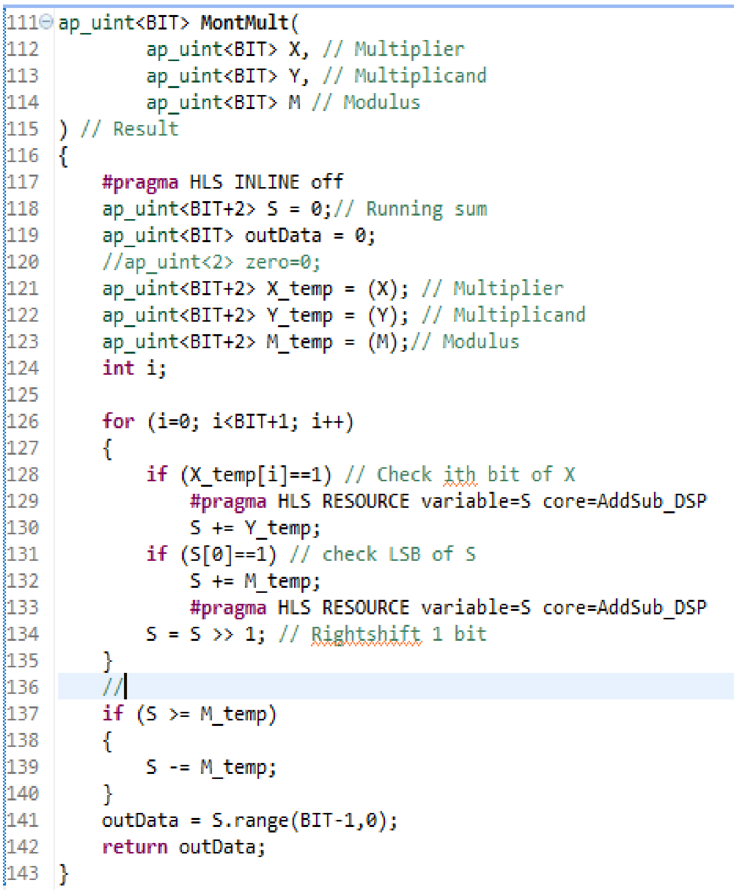


```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../tb_2.cpp in debug mode
   Compiling ../../../moduller.cpp in debug mode
   Generating csim.exe
teorik=8.000000    pratik=8
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Şekil 3.13 : C++ benzetim sonucu.

3.3.1.5 Montgomery modüler çarpma algoritması için tasarlanan fonksiyon

Bölüm 3.2.3'te anlatılan Montgomery modüler çarpım algoritmasının adımları uygulanarak, denklem 3.10'da ki işlem için Şekil 3.14'te gösterilen fonksiyon tasarlanmıştır.

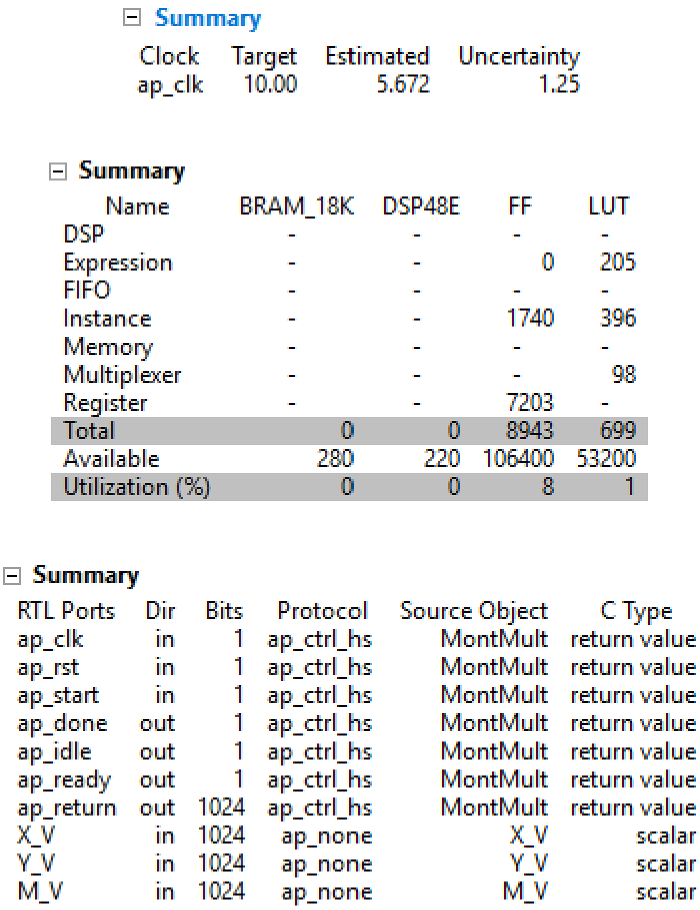


```cpp
ap_uint<BIT> MontMult(
        ap_uint<BIT> X, // Multiplier
        ap_uint<BIT> Y, // Multiplicand
        ap_uint<BIT> M // Modulus
) // Result
{
    #pragma HLS INLINE off
    ap_uint<BIT+2> S = 0;// Running sum
    ap_uint<BIT> outData = 0;
    //ap_uint<2> zero=0;
    ap_uint<BIT+2> X_temp = (X); // Multiplier
    ap_uint<BIT+2> Y_temp = (Y); // Multiplicand
    ap_uint<BIT+2> M_temp = (M);// Modulus
    int i;

    for (i=0; i<BIT+1; i++)
    {
        if (X_temp[i]==1) // Check ith bit of X
            #pragma HLS RESOURCE variable=S core=AddSub_DSP
            S += Y_temp;
        if (S[0]==1) // check LSB of S
            S += M_temp;
            #pragma HLS RESOURCE variable=S core=AddSub_DSP
        S = S >> 1; // Rightshift 1 bit
    }
    //
    if (S >= M_temp)
    {
        S -= M_temp;
    }
    outData = S.range(BIT-1,0);
    return outData;
}
```

Şekil 3.14 : Montgomery modüler çarpım fonksiyonu.

Şekil 3.14’te görüldüğü üzere, HLS direktiflerinden *INLINE* direktifi kullanılarak, ilgili fonksiyonun başka bir fonksiyon altında kullanıldığında, HLS tarafından optimize edilmesi ve hiyerarşinin bozulması engellenmiştir. *RESOURCE* direktifleri ile toplama işlemi için, sentez sırasında *DSP48* bileşenlerinin kullanılması istenmiştir. *S.range* metodu ile, *ap_int.h* kütüphanesinin getirdiği özelliklerden olan S yazmacının belli bir aralıktaki bitlerine ulaşma sağlanmıştır. Ayrıca “[ ]” operatörü kullanılarak, ilgili yazmacın ilgili bitine erişilebilmektedir.

Tasarlanan fonksiyon sentezlendikten sonra, Şekil 3.15’te görüldüğü üzere sentez raporu elde edilmiştir. Tasarımda hedeflenen 10 ns’lik saat periyodu, sentezlenen modül için kullanabilen bir aralıktadır.



⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.672	1.25

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	205
FIFO	-	-	-	-
Instance	-	-	1740	396
Memory	-	-	-	-
Multiplexer	-	-	-	98
Register	-	-	7203	-
Total	0	0	8943	699
Available	280	220	106400	53200
Utilization (%)	0	0	8	1

⊟ **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	MontMult	return value
ap_rst	in	1	ap_ctrl_hs	MontMult	return value
ap_start	in	1	ap_ctrl_hs	MontMult	return value
ap_done	out	1	ap_ctrl_hs	MontMult	return value
ap_idle	out	1	ap_ctrl_hs	MontMult	return value
ap_ready	out	1	ap_ctrl_hs	MontMult	return value
ap_return	out	1024	ap_ctrl_hs	MontMult	return value
X_V	in	1024	ap_none	X_V	scalar
Y_V	in	1024	ap_none	Y_V	scalar
M_V	in	1024	ap_none	M_V	scalar

Şekil 3.15 : Sentez sonrası rapor.

Doğru sentezlenen modül, yazılan test fonksiyonları ile test edilmiş ve girişlerine verilen sayılara göre doğru ve beklenen çıkışlara ulaşılmıştır.

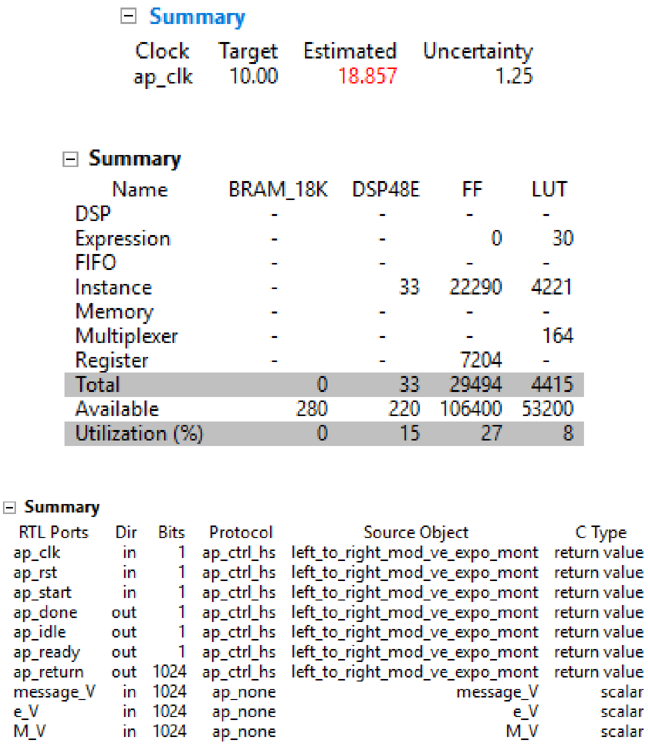
3.3.1.6 Soldan sağa Montgomery modüler çarpımı ile kullanılan RSA fonksiyonu

Bölüm 3.2.1’de anlatılan kare alma ve çarpma metodu algoritması, **Bölüm 3.3.1.5**’te tasarlanmış Montgomery modüler çarpım fonksiyonu, **Bölüm 3.3.1.3**’te ve **Bölüm 3.3.1.4**’te tasarlanan diğer fonksiyonlar ile denklem 3.1’in hesaplanması için, Şekil 3.16’da görülen fonksiyon ile HLS ortamında gerçeklenmiştir. Görüldüğü üzere, kare alma ve çarpma metodunun, kare alarak mod alma işlemleri ile çarparak mod alma işlemleri, Montgomery uzayındaki sayı değerler ile yapılmakta, daha sonra ortaya çıkan sonuç ise Montgomery uzayından çekilerek, denklem 3.1’in sonucuna ulaşılmaktadır. 1024-bitlik RSA işlemi için $R = 2^{1025}$ sayısına ihtiyaç duyulduğundan R sayısını elde etme işlemi, daha önce klasik modüler çarpma algoritmasının kullanıldığı kare alma ve çarpma metodu ile tasarlanan fonksiyon kullanılarak sağlanmıştır.

```
ap_uint<BIT> left_to_right_mod_ve_expo_mont (ap_uint<BIT>message, ap_uint<BIT> e, ap_uint<BIT> M ) {

    ap_uint<BIT> temp;
    ap_uint<BIT> R;
    ap_uint<BIT> R_square;
    ap_uint<BIT> message_mont;
    int i = 0;

    R= mod_ve_expo(ap_uint<BIT>(2),ap_uint<BIT>(BIT+1), M);
    R_square = mod_ve_carpim_2(R, R, M);
    temp=R;
    message_mont= MontMult(message,R_square,M);

    dongu: for (i=(BIT-1);i>=0;i--) {
        //#pragma HLS PIPELINE
        temp = MontMult(temp,temp,M);
        if (e[i]==1) {
            temp =  MontMult(temp,message_mont,M);
        }
    }
    temp =  MontMult(temp,ap_uint<BIT>(1),M);
    return temp;
}
```

Şekil 3.16 : Soldan sağa Montgomery modüler çarpımı ile kullanılan RSA fonksiyonu.

Tasarlanan fonksiyon sentezlendikten sonra, Şekil 3.17’de görüldüğü gibi, sentez sonrası raporu elde edilmiştir. HLS ortamında, 10 ns’lik bir saat periyodu hedeflense de, sentez sonucu HLS tarafından öngörülen gecikmelerden dolayı en az 18.857 ns’lik bir saat periyodu kullanılması gerektiği belirtilmiştir.



Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	18.857	1.25

Summary

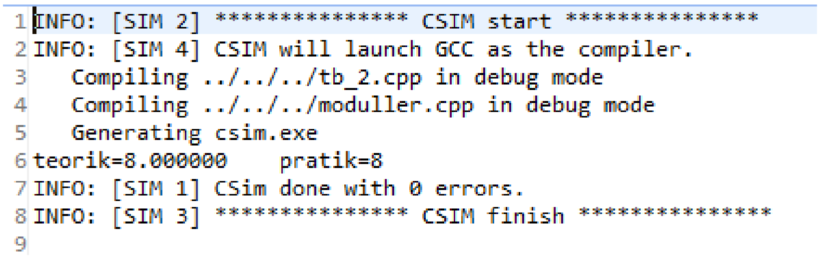
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	30
FIFO	-	-	-	-
Instance	-	33	22290	4221
Memory	-	-	-	-
Multiplexer	-	-	-	164
Register	-	-	7204	-
Total	0	33	29494	4415
Available	280	220	106400	53200
Utilization (%)	0	15	27	8

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
ap_rst	in	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
ap_start	in	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
ap_done	out	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
ap_idle	out	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
ap_ready	out	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
ap_return	out	1024	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
message_V	in	1024	ap_none	message_V	scalar
e_V	in	1024	ap_none	e_V	scalar
M_V	in	1024	ap_none	M_V	scalar

Şekil 3.17 : Sentez sonrası raporu.

M=20, *E*=10, *N*=17 değerleri için, tasarlanmış olan fonksiyonun benzetimi yapıldığında beklendiği şekilde Şekil 3.18’deki sonuçlar elde edilmiş ve fonksiyonun doğru çalıştığı gözlemlenmiştir.



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../tb_2.cpp in debug mode
   Compiling ../../../moduller.cpp in debug mode
   Generating csim.exe
teorik=8.000000    pratik=8
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************

```

Şekil 3.18 : C++ benzetim sonucu.

3.3.1.7 Tasarlanan RSA fonksiyonlarının analizi

Fonksiyonların analizi yapılırken, öncelikle modüler çarpım fonksiyonları karşılaştırılmıştır. Kararlaştırılıp seçilen modüler çarpım fonksiyonunun, kullanıldığı üst RSA fonksiyonu göz önünde bulundurularak, seçilen modüler çarpım fonksiyonunun kullanıldığı üst RSA modülünün tasarımda kullanılmasına karar verilmiştir.

Çizelge 3.1 : Montgomery modüler çarpım fonksiyon analizi.

Bit Sayısı	BRAM	DSP48	FF	LUT	Latency (min)	Latency (max)	Saat Periyodu (ns)
64	0 %0	0 %0	405 %0	342 %0	196	196	10 ns
128	0 %0	0 %0	662 %0	366 %0	388	388	10 ns
256	0 %0	0 %0	2466 %2	369 %0	772	1286	10 ns
512	0 %0	0 %0	3748 %3	368 %0	1540	2566	10 ns
1024	0 %0	0 %0	6310 %5	367 %0	3076	5126	10 ns
1024*	0 %0	0 %0	8362 %7	2392 %4	4101	4101	10 ns

(*)Fonksiyon 1024 bitlik olarak gerçeklenirken, *RESOURCE* direktifi kullanılıp, *DSP48* bileşenleri ile sentezleme zorlandığında, sentezlemede hiçbir *DSP48* bileşeni kullanılmadığı gözükmektedir. Bu direktif kullanılmadığında sebebi bilinmemekle birlikte kaynak kullanımı artmıştır.

Çizelge 3.2 : Klasik modüler çarpım metodu fonksiyonu analizi.

Bit Sayısı	BRAM	DSP48	FF	LUT	Latency (min)	Latency (max)	Saat Periyodu (ns)
64	0 %0	9 %4	342 %0	357 %0	193	193	10 ns
128	0 %0	12 %5	1053 %0	602 %1	641	641	10 ns
256	0 %0	24 %10	2079 %1	1094 %2	1281	1281	11.857ns
512	0 %0	33 %15	4129 %3	1969 %3	2561	2561	18.857ns
1024	0 %0	33 %15	8227 %7	3504 %6	5121	5121	18.857ns

Çizelge 3.1, 3.2 ve 3.3'te görüldüğü üzere kaynak kullanımı bakımından en optimize tasarım Montgomery modüler çarpım modülüdür. İşlem süresi bakımından ikinci sırada olan bir tasarım olsa da, kaynak kullanımının azlığı, Montgomery modüler çarpım modülünün seçilmesini sağlamıştır.

Çizelge 3.3 : Operatörler kullanılarak yapılan modüler çarpım metodu fonksiyonu analizi.

Bit Sayısı	BRAM	DSP48	FF	LUT	Latency (min)	Latency (max)	Saat Periyodu (ns)
64	0 %0	16 %7	1485 %1	1342 %2	136	136	10 ns
128	0 %0	16 %7	1741 %1	1934 %3	264	264	10 ns
256	0 %0	16 %7	2253 %2	2918 %5	520	520	10 ns
512	0 %0	16 %7	3277 %3	2918 %5	1032	1032	10 ns
1024	0 %0	16 %7	5325 %5	2918 %5	2056	2056	10 ns

Seçilen Montgomery çarpım modülünün kullanılabileceği RSA fonksiyonu, soldan sağa Montgomery modüler çarpımı ile kullanılan RSA fonksiyonu olduğundan, RSA işlevini gerçekleştirecek üst modül olarak belirtilen RSA fonksiyonu kullanılmıştır. Çizelge 3.4'te bu RSA üst modülünün kaynak kullanımı ve işlem süresi belirtilmiştir.

Çizelge 3.4 : Kullanılacak RSA üst fonksiyonu analizi.

Bit Sayısı	BRAM	DSP48	FF	LUT	Latency (min)	Latency (max)	Saat Periyodu (ns)
1024	0 %0	33 %15	26861 %25	4083 %7	8413201	21011477	18.857ns

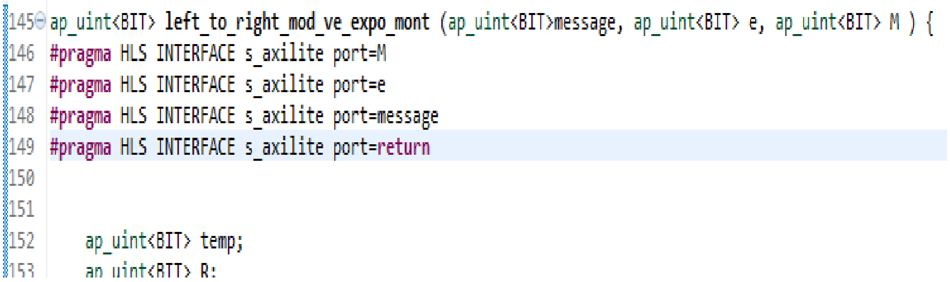
3.3.2 RSA modülünün ilgili arayüzler ile tasarımı

Tasarlanan RSA modülünün, mikroişlemci tabanlı bir sistemde kullanılabilmesi için AXI4-Lite ve AXI4-Stream arayüzleri kullanılarak tekrardan tasarlanmıştır.

3.3.2.1 AXI4-Lite arayüzü eklenen RSA modülü

AXI4-Lite arayüzü, sayısal modüllerin, mikroişlemci ile haberleşmesi ve veri alışverişi gerçekleştirmesi için kullanılan, Xilinx'in ürettiği bir arayüzdür [16]. Tasarlanan RSA modülünün mikroişlemci ile kontrol edilip, ilgili girişlerin modüle gönderilmesi ve RSA modülünün çıkışının mikroişlemci tarafından alınması için AXI4-Lite arayüzü kullanılmıştır.

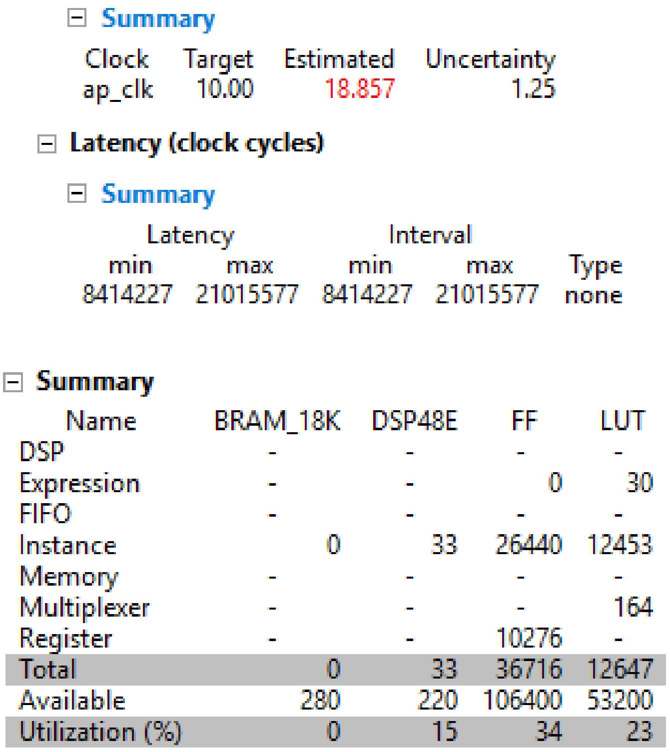
Vivado HLS geliştirme ortamının sağladığı büyük kolaylıklardan birisi, HLS direktifleri ile tasarlanan fonksiyonun giriş ve çıkışlarının farklı arayüzler ile sentezlenebilmesidir [19].



```
ap_uint<BIT> left_to_right_mod_ve_expo_mont (ap_uint<BIT>message, ap_uint<BIT> e, ap_uint<BIT> M ) {
#pragma HLS INTERFACE s_axilite port=M
#pragma HLS INTERFACE s_axilite port=e
#pragma HLS INTERFACE s_axilite port=message
#pragma HLS INTERFACE s_axilite port=return


    ap_uint<BIT> temp;
    ap_uint<BIT> R;
```

Şekil 3.19 : AXI4-Lite arayüzü ile sentezleme yapmak için kullanılan direktifler.

Şekil 3.19'da görüldüğü üzere üç adet giriş ve bir adet çıkış için INTERFACE direktifleri kullanılarak, ilgili giriş ve çıkışlar AXI4-Lite arayüzü ile otomatik olarak sentezlenebilmektedir. AXI4-Lite arayüzünün gerektirdiği tüm donanımsal tasarımlar HLS tarafından otomatik olarak gerçekleştirilmektedir. Ayrıca mikroişlemci yazılımının tasarımı sırasında, modülün kontrolü için kullanılacak sürücü fonksiyonları HLS tarafından otomatik olarak oluşturulmakta ve HLS tarafından Vivado'ya dışa aktarım esnasında aktarılmaktadır. Bu sebeple HLS ortamındaki AXI4-Lite arayüzü kullanımının, donanım tanımlama dilleri ile kullanıma göre daha kolay bir şekilde gerçekleştirilebildiği söylenebilir.



Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	18.857	1.25

Latency (clock cycles)

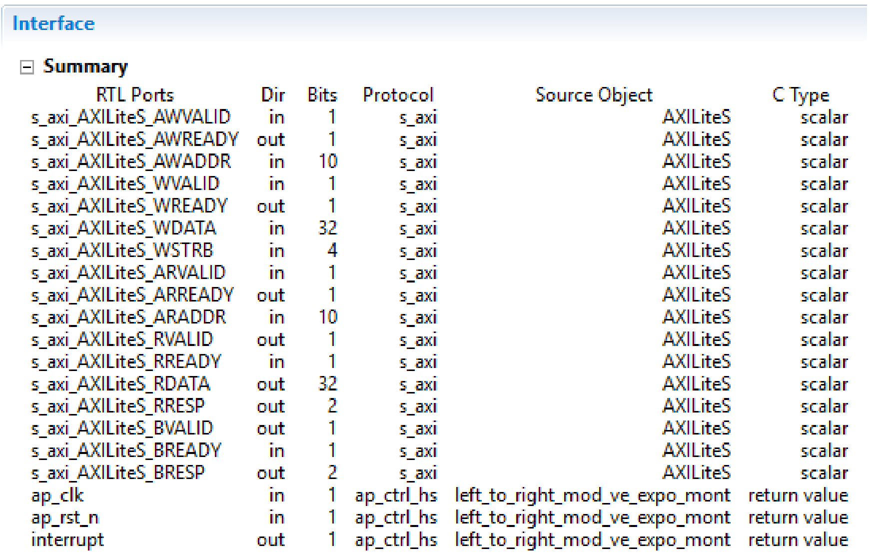
Summary

Latency		Interval		
min	max	min	max	Type
8414227	21015577	8414227	21015577	none

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	30
FIFO	-	-	-	-
Instance	0	33	26440	12453
Memory	-	-	-	-
Multiplexer	-	-	-	164
Register	-	-	10276	-
Total	0	33	36716	12647
Available	280	220	106400	53200
Utilization (%)	0	15	34	23

Şekil 3.20 : AXI4-Lite arayüzü ile sentez sonrası raporu.



Interface

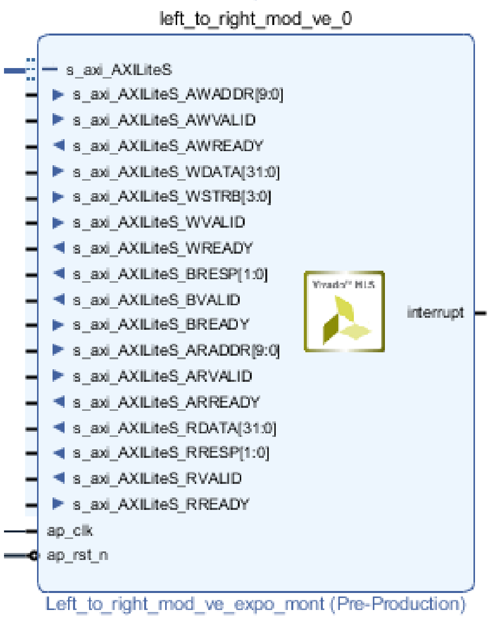
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_AXILiteS_AWVALID	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_AWREADY	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_AWADDR	in	10	s_axi	AXILiteS	scalar
s_axi_AXILiteS_WVALID	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_WREADY	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_WDATA	in	32	s_axi	AXILiteS	scalar
s_axi_AXILiteS_WSTRB	in	4	s_axi	AXILiteS	scalar
s_axi_AXILiteS_ARVALID	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_ARREADY	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_ARADDR	in	10	s_axi	AXILiteS	scalar
s_axi_AXILiteS_RVALID	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_RREADY	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_RDATA	out	32	s_axi	AXILiteS	scalar
s_axi_AXILiteS_RRESP	out	2	s_axi	AXILiteS	scalar
s_axi_AXILiteS_BVALID	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_BREADY	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_BRESP	out	2	s_axi	AXILiteS	scalar
ap_clk	in	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
ap_rst_n	in	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value
interrupt	out	1	ap_ctrl_hs	left_to_right_mod_ve_expo_mont	return value

Şekil 3.20 (devam) : AXI4-Lite arayüzü ile sentez sonrası raporu.

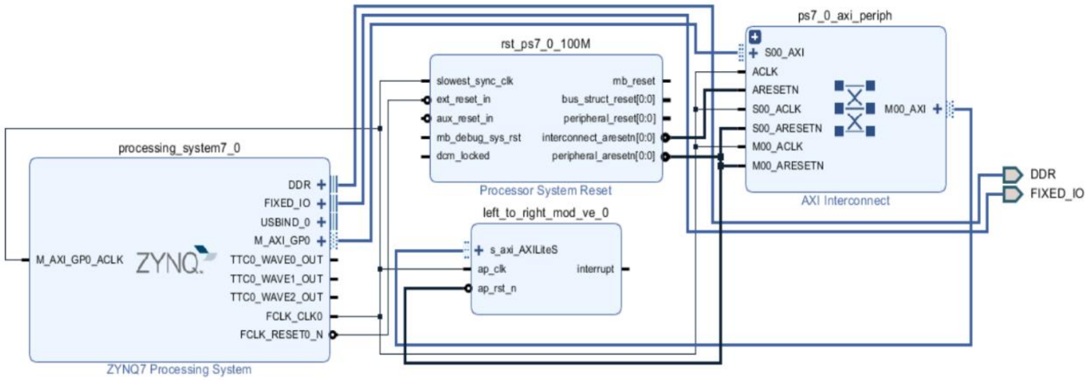
Şekil 3.20’den görüldüğü üzere, Şekil 3.17’de sentezlenmiş fonksiyonun kaynak kullanımına göre kaynak kullanımı artmıştır. Çünkü AXI4-Lite arayüzünün çalışması için gerekli olan devre elemanları, kaynak kullanımını arttırmıştır. Sentezlenen modülün giriş ve çıkış sinyallerinden anlaşılacağı üzere, AXI4-Lite arayüzünün gerektirdiği sinyaller HLS tarafından otomatik olarak oluşturulmuştur.

AXI4-Lite arayüzü ile tekrardan tasarlanan 1024-RSA modülü, HLS ortamından Vivado geliştirme ortamına modül bloğu (Intellectual Property – IP) olarak aktarılmıştır. Aktarılan IP'nin görüntüsü, Şekil 3.21'de gösterilmektedir.



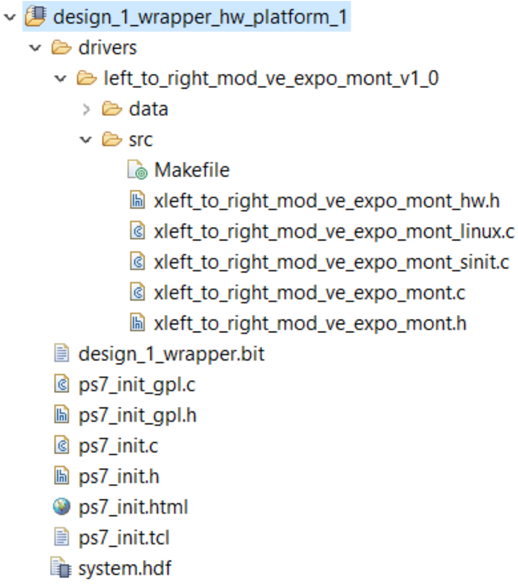
Şekil 3.21 : AXI4-Lite arayüzü eklenmiş RSA modülü.

Vivado geliştirme ortamında oluşturulan blok tasarımına HLS'ten aktarılan IP ve ZYNQ7 Processing System IP'leri eklendi. Geliştirme arayüzünde sağlanan *Run Block Automation* işlevi ile otomatik olarak *Processor System Reset* ve *AXI Interconnect* modülleri de eklenerek gerekli bağlantıları Şekil 3.22'deki gibi sağlandı. ZYNQ7 modülünün ayarlarından programlanabilir lojik saati ayarlaması *FCLK_CLK0* → 10 MHz olacak şekilde belirlendi. Ardından blok tasarımın HDL kodu program tarafından oluşturtularak *bitstream* dosyası hazırlanmıştır. İşlem tamamlandıktan sonra sistemi çalıştırmayı sağlayacak mikroişlemci yazılımını tasarlamak için SDK ortamına geçildi.



Şekil 3.22 : AXI4-Lite arayüzü eklenmiş RSA modülünün ZYNQ işlemciye bağlandığı blok tasarımı.

Vivado geliştirme ortamından SDK ortamına geçildiğinde Şekil 3.23'te görülecektir ki HLS ortamında oluşturulan modülün yazılım geliştirme ortamında işlevsel hale gelebilmesi için gerekli sürücü dosyaları, HLS tarafından otomatik olarak oluşturulmuştur. HLS ortamının burada da tasarımcıya sağlamış olduğu bir kolaylık görülmektedir. Aksi takdirde tasarımcı kendi blokları için sürücü dosyalarını yazmak zorunda kalacaktır.

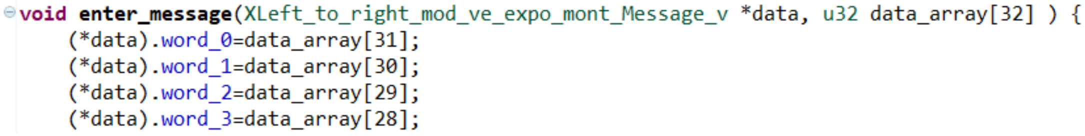


- design_1_wrapper_hw_platform_1
  - drivers
    - left_to_right_mod_ve_expo_mont_v1_0
      - data
      - src
        - Makefile
        - xleft_to_right_mod_ve_expo_mont_hw.h
        - xleft_to_right_mod_ve_expo_mont_linux.c
        - xleft_to_right_mod_ve_expo_mont_sinit.c
        - xleft_to_right_mod_ve_expo_mont.c
        - xleft_to_right_mod_ve_expo_mont.h
  - design_1_wrapper.bit
  - ps7_init_gpl.c
  - ps7_init_gpl.h
  - ps7_init.c
  - ps7_init.h
  - ps7_init.html
  - ps7_init.tcl
  - system.hdf

Şekil 3.23 : AXI4-Lite arayüzü eklenmiş RSA modülünün sürücü dosyaları.

HLS tarafından oluşturulan sürücülerin *header* dosyası incelendiğinde 1024 bitlik giriş kanallarının AXI4-Lite protokolüne uygun olarak 32 bitlik 32 adet kelimeye bölündüğü görülmektedir. Örneğin 1024-RSA modülünün girişi olan *message* için, *XLeft_to_right_mod_ve_expo_mont_Message_v* isimli C veri yapısı oluşturulmuştur. Aksi takdirde protokolün kullanımı mümkün değildir çünkü veri kanalının genişliği 32 bittir.

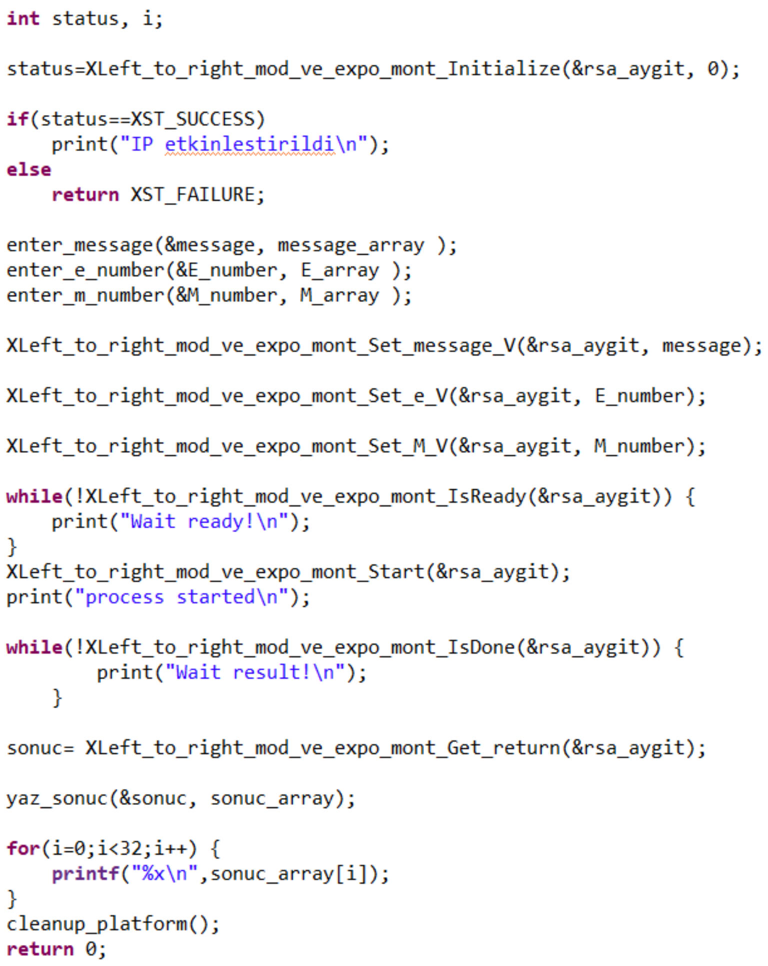
1024-RSA modülünün çalıştırılması için yazılım tasarımına geçildiğinde, modülün girişlerine verilerin gönderilmesi için birkaç fonksiyon oluşturulması gerekli görülmüştür. Örneğin kullanıcı tarafından oluşturulan bir mesaj dizisinin HLS tarafından oluşturulmuş veri yapısına aktarılması için Şekil 3.24'te görülen *enter_message* isimli fonksiyon oluşturulmuştur. Görüldüğü gibi dizideki her 32 bitlik veri kelimelere tek tek aktarılmaktadır.



```
void enter_message(XLeft_to_right_mod_ve_expo_mont_Message_v *data, u32 data_array[32] ) {
    (*data).word_0=data_array[31];
    (*data).word_1=data_array[30];
    (*data).word_2=data_array[29];
    (*data).word_3=data_array[28];
```

Şekil 3.24 : *enter_message* fonksiyonu.

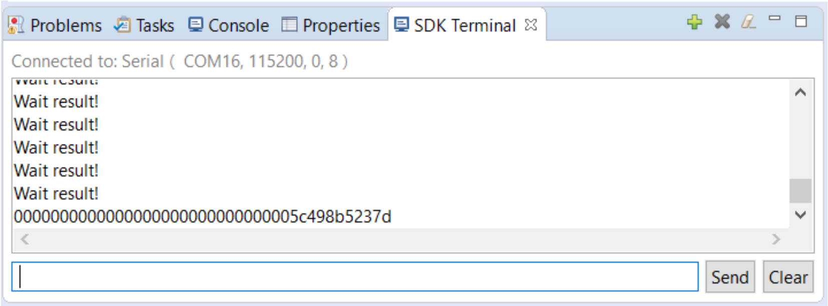
Ardından Şekil 3.25'teki ana fonksiyon yazılmıştır. Görüleceği üzere öncelikle *XLeft_to_right_mod_ve_expo_mont_Initialize* fonksiyonu kullanılarak 1024-RSA modülünün işleme hazır hale getirilmesi sağlanmaktadır. Bu fonksiyonun çıkışı da *status* isimli değişkene atanarak, modülün başarılı biçimde hazırlanıp hazırlanmadığı kontrol edilir. Sonrasında modüle gönderilecek veriler *XLeft_to_right_mod_ve_expo_mont_Set_** fonksiyonları sayesinde modül girişinde bulunan kaydedicilerde tutulur. Modülün mikroişlemci tarafından tetiklenip işleme başlaması için bir döngü içinde modülün hazır olup olmadığı kontrol edildikten sonra *XLeft_to_right_mod_ve_expo_mont_Start* fonksiyonu ile işleme başlaması sağlanır ve tekrardan bir döngü içinde modülün işlemi bitirip bitirmediği kontrol edilir. RSA işlemi gerçekleştikten sonra çıkıştan alınan sonuç *sonuc_array* isimli diziye *enter_message* fonksiyonundaki mantığın tam tersi ile yazılır ve SDK terminalinde gösterilir.



```
int status, i;

status=XLeft_to_right_mod_ve_expo_mont_Initialize(&rsa_aygit, 0);

if(status==XST_SUCCESS)
    print("IP etkinlestirildi\n");
else
    return XST_FAILURE;

enter_message(&message, message_array );
enter_e_number(&E_number, E_array );
enter_m_number(&M_number, M_array );

XLeft_to_right_mod_ve_expo_mont_Set_message_V(&rsa_aygit, message);

XLeft_to_right_mod_ve_expo_mont_Set_e_V(&rsa_aygit, E_number);

XLeft_to_right_mod_ve_expo_mont_Set_M_V(&rsa_aygit, M_number);

while(!XLeft_to_right_mod_ve_expo_mont_IsReady(&rsa_aygit)) {
    print("Wait ready!\n");
}
XLeft_to_right_mod_ve_expo_mont_Start(&rsa_aygit);
print("process started\n");

while(!XLeft_to_right_mod_ve_expo_mont_IsDone(&rsa_aygit)) {
        print("Wait result!\n");
    }

sonuc= XLeft_to_right_mod_ve_expo_mont_Get_return(&rsa_aygit);

yaz_sonuc(&sonuc, sonuc_array);

for(i=0;i<32;i++) {
    printf("%x\n",sonuc_array[i]);
}
cleanup_platform();
return 0;
```

Şekil 3.25 : Mikroişlemci ana fonksiyonu.

Yazılım tamamlandıktan sonra Zedboard ZC702 kartı bilgisayara bağlanıp FPGA programlanmıştır. Yazılımın içerisinde elle girilen veriler ile modülün çalışıp çalışmadığı kontrol edilmiş ve Şekil 3.26'da görüleceği üzere test başarı ile tamamlanmıştır.

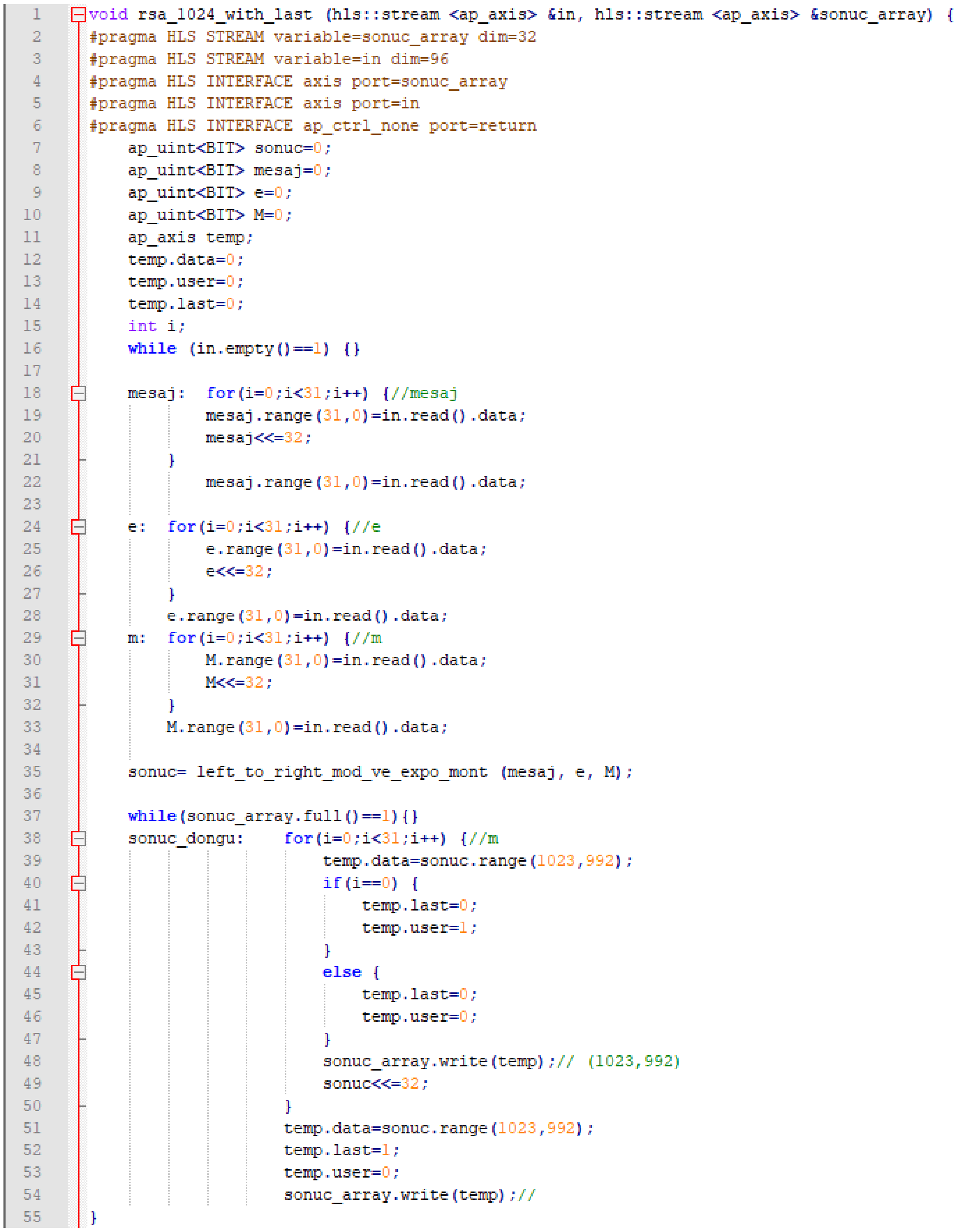


```
Connected to: Serial (  COM16, 115200, 0, 8 )
Wait result!
Wait result!
Wait result!
Wait result!
Wait result!
0000000000000000000000000000005c498b5237d
```

Şekil 3.26 : SDK UART Terminali.

3.3.2.2 AXI4-Stream arayüzü eklenen RSA modülü

Bölüm 3.3.1.6’da tasarlanan 1024-RSA modülünün, AXI4-Stream arayüzü ile tekrardan tasarlanması için Şekil 3.27’deki fonksiyon HLS’de tasarlanmıştır. AXI4-Stream arayüzünün HLS’de kolaylıkla kullanılabilmesi için HLS geliştirme ortamında tanımlı *hls_stream.h* kütüphanesi eklenmiştir. *STREAM* ve *INTERFACE* direktifleri ile giriş ve çıkışların ilgili arayüzde sentezlenmesi sağlanmıştır. *hls_stream.h* kütüphanesinde tanımlı *hls::stream* nesneleri, AXI4-Stream arayüzünün getirdiği özellikleri, C++ için kullanılabilir hale getirmektedir. STREAM direktiflerinde giriş için 32, çıkış için 96 boyut uygulandığı belirtildiğinde, AXI4-Stream FIFO’larının sahip olması gereken en düşük derinlik bulunmuş olunur. Gözlenecek olursa “ap_axis” isminde bir veri yapısı tanımlanmıştır. Bunun sebebi AXI4-Stream arayüzünü kullanacak olan DMA modülünün isterleridir. DMA’nın düzgün çalışabilmesi için veri paketleri gönderildikten sonra, aslında AXI4-Stream protokolünde de bulunan fakat manuel olarak ayarlanması gereken bir sinyal vardır. Bu da *last* sinyalidir. Bu sinyal, DMA’ya veri paketleri gönderilirken en son paket ile birlikte ‘1’ yapılması gereken bir sinyaldir. Aksi takdirde DMA modülü doğru çalışmayacak ve FPGA programlanırken hatalar alınacaktır.

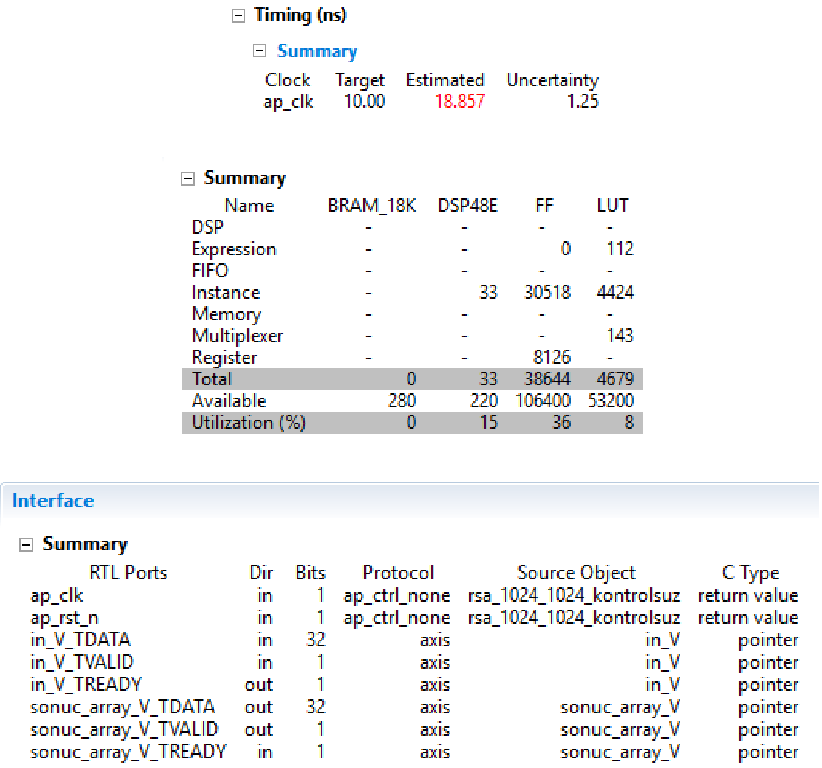


```
void rsa_1024_with_last (hls::stream <ap_axis> &in, hls::stream <ap_axis> &sonuc_array) {
#pragma HLS STREAM variable=sonuc_array dim=32
#pragma HLS STREAM variable=in dim=96
#pragma HLS INTERFACE axis port=sonuc_array
#pragma HLS INTERFACE axis port=in
#pragma HLS INTERFACE ap_ctrl_none port=return
    ap_uint<BIT> sonuc=0;
    ap_uint<BIT> mesaj=0;
    ap_uint<BIT> e=0;
    ap_uint<BIT> M=0;
    ap_axis temp;
    temp.data=0;
    temp.user=0;
    temp.last=0;
    int i;
    while (in.empty()==1) {}

    mesaj:  for(i=0;i<31;i++) {//mesaj
            mesaj.range(31,0)=in.read().data;
            mesaj<<=32;
        }
            mesaj.range(31,0)=in.read().data;

    e:  for(i=0;i<31;i++) {//e
            e.range(31,0)=in.read().data;
            e<<=32;
        }
        e.range(31,0)=in.read().data;
    m:  for(i=0;i<31;i++) {//m
            M.range(31,0)=in.read().data;
            M<<=32;
        }
        M.range(31,0)=in.read().data;

    sonuc= left_to_right_mod_ve_expo_mont (mesaj, e, M);

    while(sonuc_array.full()==1){}
    sonuc_dongu:    for(i=0;i<31;i++) {//m
                        temp.data=sonuc.range(1023,992);
                        if(i==0) {
                            temp.last=0;
                            temp.user=1;
                        }
                        else {
                            temp.last=0;
                            temp.user=0;
                        }
                        sonuc_array.write(temp);// (1023,992)
                        sonuc<<=32;
                    }
                    temp.data=sonuc.range(1023,992);
                    temp.last=1;
                    temp.user=0;
                    sonuc_array.write(temp);//
}
```

Şekil 3.27 : AXI4-Stream arayüzü eklenmiş RSA modülü.

Görüldüğü üzere fonksiyonun bir adet girişi ve bir adet çıkışı bulunmaktadır. Bu giriş ve çıkış 32-bitlik genişliğe sahiptir. Öncelikle *in.empty* metodu kullanılarak, girişteki FIFO’nun boş mu olduğu kontrol edilmektedir. FIFO’nun dolu olduğu algılandığında, *in.read* metodu ile okunan 32-bitlik değer, 1024-bitlik yazmaçın en önemsiz bitlerine yazılır ve daha sonra 1024-bitlik yazmaç 32 kere sola kaydırılır. Böylece FIFO’dan okunan 1024-bitlik mesaj, 32-bitlik paketler halinde modüle sokularak tek bir yazmaça aktarılmış olur. RSA için gerekli olan e ve M sayıları da anlatılan şekilde modül içerisine sokularak üç adet 1024-bitlik bilgi (96 adet 32-bitlik paket) modül içerisine alınmış olur ve RSA işlevini yapacak olan fonksiyona, ilgili paketler sokularak fonksiyonun çalışması sağlanır. RSA işlevini gerçekleştiren fonksiyon işlemini tamamladıktan sonra, yazma işleminin yapılacağı çıkış FIFO'sunun dolu olup olmadığı *sonuc_array.full* metodu ile kontrol edilir. Eğer FIFO boş ise, RSA işlevini yapan fonksiyonun sonucu 32 adet 32-bitlik paketler aracılığıyla çıkış FIFO'suna yazdırılmakta ve son paket gönderildiğinde *last* sinyali de aynı zamanda '1' yapılmaktadır.

Tasarlanan fonksiyonun, diğer fonksiyonların aksine tek giriş ve tek çıkışa sahip olduğu görülmektedir. Bu giriş ve çıkışların bağlı olduğu FIFO'lar olduğu belirtilmiştir. Bu FIFO'lar HLS tarafından sentezlenmemektedir. Dolayısıyla IP'nin herhangi bir sistem içerisinde kullanıldığında FIFO'ların harici olarak RSA modülüne bağlanması gerektiği anlaşılmaktadır.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	18.857	1.25

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	112
FIFO	-	-	-	-
Instance	-	33	30518	4424
Memory	-	-	-	-
Multiplexer	-	-	-	143
Register	-	-	8126	-
Total	0	33	38644	4679
Available	280	220	106400	53200
Utilization (%)	0	15	36	8

Interface

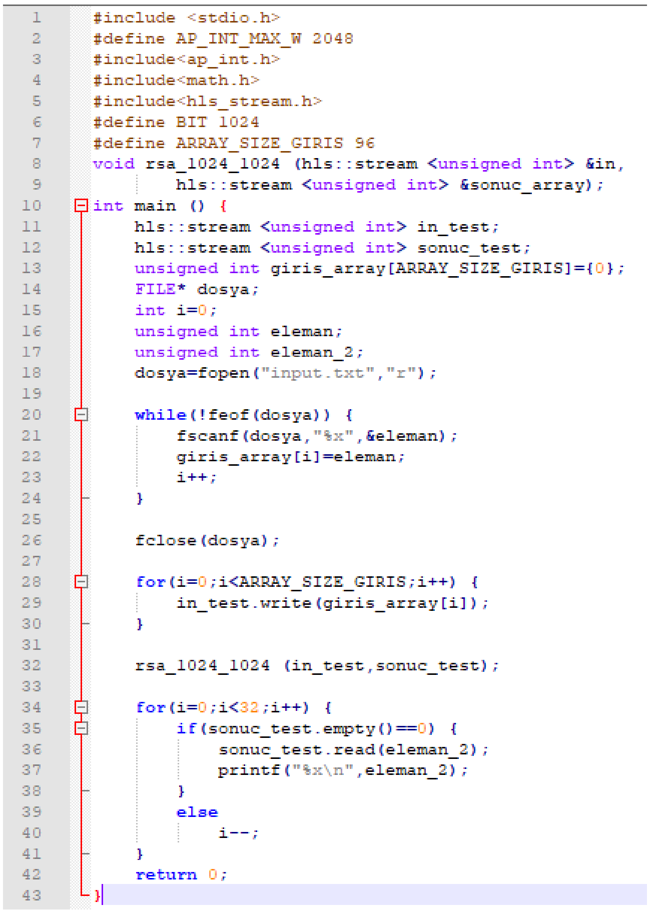
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	rsa_1024_1024_kontrolsuz	return value
ap_rst_n	in	1	ap_ctrl_none	rsa_1024_1024_kontrolsuz	return value
in_V_TDATA	in	32	axis	in_V	pointer
in_V_TVALID	in	1	axis	in_V	pointer
in_V_TREADY	out	1	axis	in_V	pointer
sonuc_array_V_TDATA	out	32	axis	sonuc_array_V	pointer
sonuc_array_V_TVALID	out	1	axis	sonuc_array_V	pointer
sonuc_array_V_TREADY	in	1	axis	sonuc_array_V	pointer

Şekil 3.28 : Sentez sonrası raporu.

Şekil 3.28'ten görüldüğü üzere sentez sonrası raporunda belirtildiği gibi, AXI4-Stream arayüzünün sahip olduğu sinyaller başarı ile sentezlenmiştir. Arayüzün gerçeklenmesi için gerekli olan kaynak kullanımı rapora yansımaktadır. Modül için en az 18.857 ns bir saat periyodu kullanılması gerekmektedir. Kullanılması gereken minimum saat periyodu, modülün FPGA’e implementasyonu sırasında, oluşacak ara bağlantılar yüzünden artış gösterebilir.

Tasarlanan modül için, HLS ortamında bir test dosyası yazılmıştır. Yazılan test dosyası, benzetim için kullanılacak 32-bitlik paketler için bir .txt dosyasından dosya okuması gerçekleştirmektedir. Çekilen paketler bir diziye yazılarak, test fonksiyonunda yer alan *hls::stream”* nesneleri doldurulmuştur. Bu nesneler test fonksiyonlarında FIFO davranışı göstermektedir. Benzetimi yapılacak modüle, ilgili FIFO nesnesi giriş verilerek, fonksiyonun çıkışları ayrı bir FIFO nesnesine aktarılmıştır. Aktarılan FIFO nesnesindeki paketler, *printf* fonksiyonu ile de ekrana yazdırılmaktadır. Benzetim sonucu tasarlanan modülün doğru çalıştığı anlaşılmıştır. Test kodu Şekil 3.29’te görülmektedir.

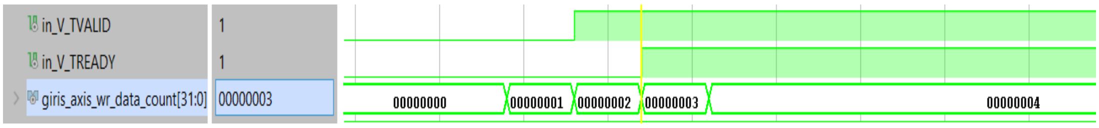


```
#include <stdio.h>
#define AP_INT_MAX_W 2048
#include<ap_int.h>
#include<math.h>
#include<hls_stream.h>
#define BIT 1024
#define ARRAY_SIZE_GIRIS 96
void rsa_1024_1024 (hls::stream <unsigned int> &in,
        hls::stream <unsigned int> &sonuc_array);
int main () {
    hls::stream <unsigned int> in_test;
    hls::stream <unsigned int> sonuc_test;
    unsigned int giris_array[ARRAY_SIZE_GIRIS]={0};
    FILE* dosya;
    int i=0;
    unsigned int eleman;
    unsigned int eleman_2;
    dosya=fopen("input.txt","r");

    while(!feof(dosya)) {
        fscanf(dosya,"%x",&eleman);
        giris_array[i]=eleman;
        i++;
    }

    fclose(dosya);

    for(i=0;i<ARRAY_SIZE_GIRIS;i++) {
        in_test.write(giris_array[i]);
    }

    rsa_1024_1024 (in_test,sonuc_test);

    for(i=0;i<32;i++) {
        if(sonuc_test.empty()==0) {
            sonuc_test.read(eleman_2);
            printf("%x\n",eleman_2);
        }
        else
            i--;
    }
    return 0;
}
```

Şekil 3.29 : AXI4-Stream RSA modülü için test kodu.

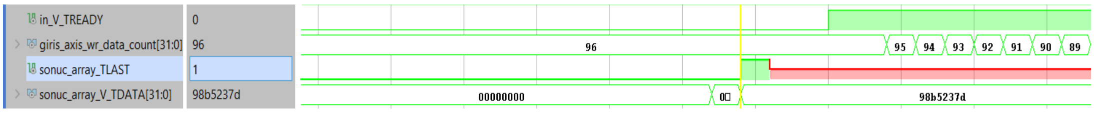
Tasarlanan RSA modülü, Vivado HLS’ten IP olarak Vivado’ya aktarılmıştır. Vivado’ya aktarılan modülün benzetimi için test kodu yazılmıştır. Benzetim için modülün giriş ve çıkışına AXI4-Stream FIFO'lar bağlanmıştır. Bağlanan FIFO'ların derinlikleri giriş için 256, çıkış için ise 128 paket alacak biçimde seçilmiştir. Gerekli modül bağlantıları yapılıp benzetim işlemi yürütüldüğünde RSA modülünün FIFO'dan veriyi düzgün bir biçimde alıp, işleyip, sonucu da çıkışında bulunan FIFO'ya aktardığı görülmüştür.

Şekil 3.30'da görüleceği üzere giriş FIFO'sunun veri almaya hazır olduğu andan itibaren her bir saat darbesinde veriler içine yazılmıştır. RSA modülümüz FIFO'nun boş olmadığı anladığı andan itibaren gerekli işlemleri yaparak *in_V_TREADY* sinyalini '1' yaparak FIFO'dan bulunan verileri çekmeye başlar. Dalga şeklinden de anlaşılacağı üzere modülümüz veri çekmeye başladığından itibaren FIFO içerisindeki veri sayısını belirten sinyal "4" değerinde kalmıştır. FIFO'ya veri yazılırken aynı zamanda modülün de FIFO'yu boşaltması sebebiyle içeride bulunan veri sayısı bütün 96 paket alınana kadar sabit kalacaktır.



Şekil 3.30 : RSA modülünün FIFO'dan veri çekmeye başladığı an.

96 adet paket başarıyla alındıktan sonra RSA işlemi gerçekleştirilecek ve işlem sonunda da Şekil 3.31'de görüldüğü gibi hesaplanmış olan değer veri kanalına verilmiş ve son veri paketinin gönderilmesi ile birlikte aynı anda *sonuc_array_TLAST* sinyali '1' yapılmıştır. RSA modülü tekrardan 3 saat darbesi sonra veri almaya hazır olacağını da *in_V_TREADY* sinyali ile belirtmekte ve görüldüğü gibi giriş FIFO'sundaki 96 paketlik yeni veriyi çekmeye başlayarak aktif hale gelmektedir. Bu sayede AXI4-Stream arayüzü ile tasarlanan RSA modülünün de düzgün bir biçimde çalıştığı test edilmiş ve doğrulanmıştır.

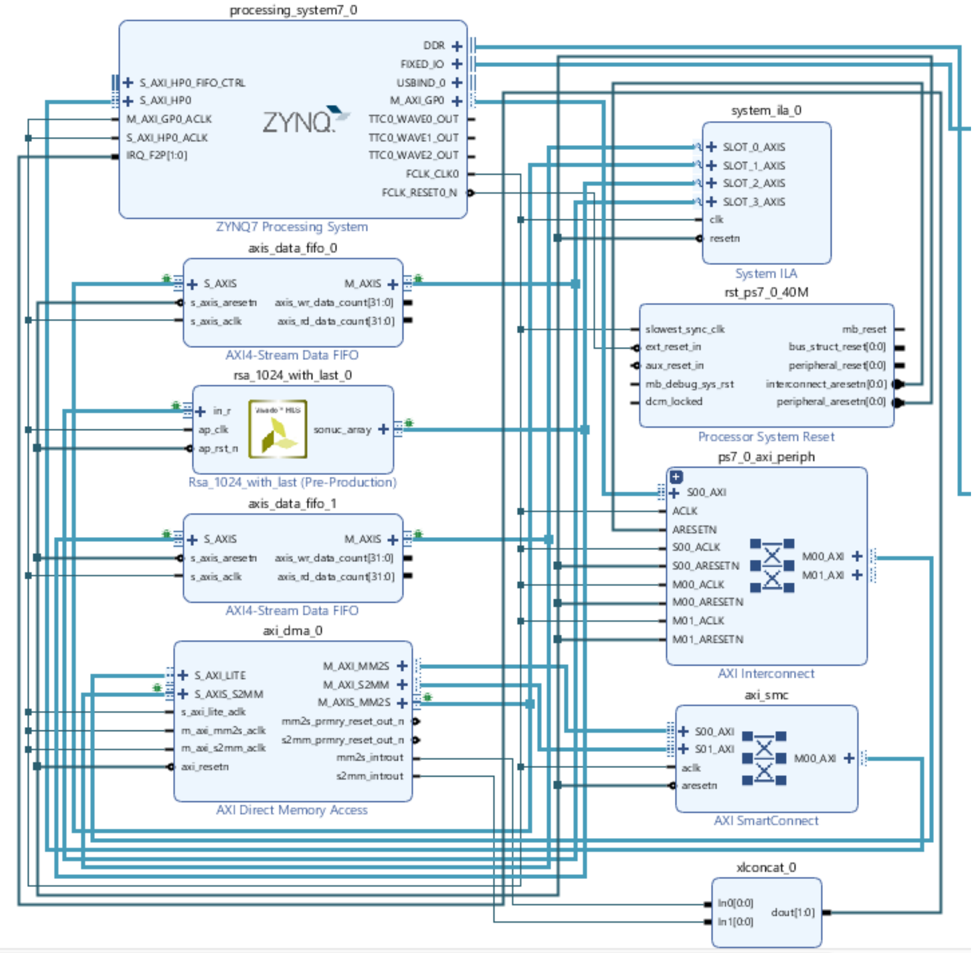


Şekil 3.31 : RSA modülünün çıkış FIFO'suna son paketi gönderdiği an.

DMA ile yapılacak olan tasarımda kullanılacak HLS modüllerinin tasarımında *last* sinyali dikkat edilmesi gereken çok önemli bir unsurdur. Çünkü HLS tarafından bu sinyal otomatik olarak çıkartılmamaktadır. DMA modülünün FIFO'dan düzgün bir biçimde veri alması ve sistemde hataya sebep olmaması için bu duruma özellikle dikkat edilmelidir.

3.3.3 RSA modülünün DMA ile kullanımı

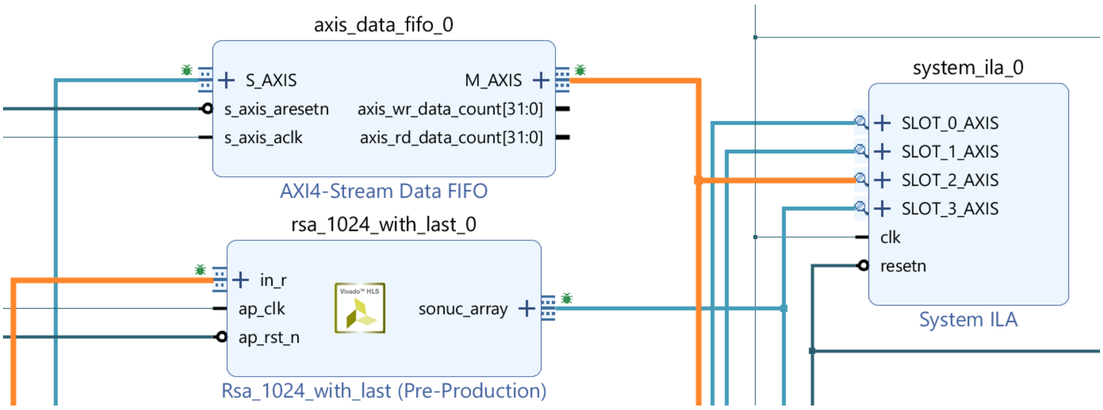
Bölüm 3.3.2.2'de tasarlanan AXI4-Stream arayüzüne sahip RSA modülünün DMA ile uyumlu bir şekilde çalışması için Vivado geliştirme ortamında bir blok tasarım yapılmıştır. Zaten modüle AXI4-Stream arayüzünün eklenme amacı da DMA'nın uyumlu olduğu protokolün AXI4-Stream olmasıdır. Şekil 3.32'de görülen blok tasarımda HLS ortamından aktarılan RSA IP'si ZYNQ7 mikroişlemcisi ile **bölüm 3.3.2.2**'de bağlandığı gibi bağlanmıştır. Ardından Vivado'nun hazır olarak sunduğu *AXI Direct Memory Access* IP'si eklenmiştir.



Şekil 3.32 : RSA modülünün DMA ile kullanıldığı blok tasarım.

DMA'nın mikroişlemci ile birlikte düzgün çalışması için gerekli birkaç ayar yapılmalıdır. Öncelikle ZYNQ7'nin konfigürasyon ayarlarından *S AXI HP0 interface* aktifleştirilmelidir. Bu ayar aktif hale geldiğinde Vivado'nun otomatik bağlama yöntemi ile *AXI SmartConnect* bloğu da tasarıma eklenecek ve ZYNQ işlemci ile DMA arasında bir köprü görevi görecektir. Burada DMA, mikroişlemcinin bir *slave* olarak görev görecektir bu sebeple yüksek performanslı *AXI Interconnect* arayüzüne ihtiyaç duyulmaktadır.

Ardından biri RSA modülünün girişine diğeri de çıkışına bağlanacak FIFO IP'leri tasarıma eklenmiştir. RSA'nın girişine bağlanan FIFO DMA'nın bir *slave* i olacağından *M_AXIS_MM2S* veriyolu ile birbirine bağlanır. Aynı işlem çıkışta bulunan FIFO için de geçerlidir ve bu sefer FIFO'nun çıkışı *M_AXIS_S2MM* veriyoluna bağlanır. DMA'nın kontrol ve durum bilgisi akışını sağlayan ilgili giriş ve çıkışları da deaktif hale getirilmiştir. Çünkü bu veri akışı gereksiz bir bilgi yüküne neden olacaktır ve tasarımda herhangi bir etkisi olmayacaktır.

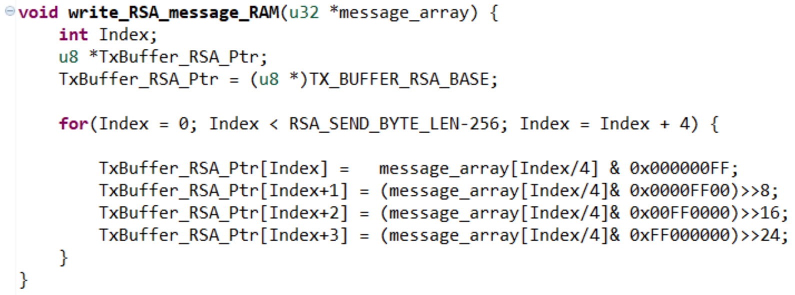


Şekil 3.33 : Analiz edilecek sinyalin ILA modülüne bağlantısı.

Sistemin sağlıklı bir biçimde çalışıp çalışmadığını anlayabilmek için bazen sadece sonuçları incelemek yeterli olmayabilir. Bu sebeple tasarımda bulunan sinyallerin gerçek zamanlı olarak takip edilebilmesi, hataların analizinde oldukça önemli bir yer tutmaktadır. Sinyallerin taşıdığı verilerin anlık analiz edilebilmesi için Vivado geliştirme ortamının sağladığı bir araç olan entegre mantık analizörü (Integrated Logic Analyzer - ILA) kullanılmıştır. Bu araç aslında bir avometre olarak da düşünülebilir çünkü yaptığı aslında bulunan sinyallerin değerlerini FPGA kartında yazılım işletilirken analiz etmektir. Bunu sağlayabilmek için sonradan izlenmek istenen sinyale sağ tıklanıp *Debug* seçeneği seçilip *Run Connection Automation* aracının çalıştırılması yeterli olacaktır. Şekil 3.33'te görüleceği üzere seçilen sinyal *System ILA* isimli bir modül oluşturulup ona otomatik olarak paralel bağlanacaktır. ILA'nın kullanılmasına SDK ortamına geçildikten sonra değinilecektir.

SDK ortamına geçildiğinde DMA'nın kullanılabilmesi için Vivado tarafından örnek teşkil etmesi için hazırlanmış olan *xaxidma_example_simple_poll.c* dosyası referans olarak kullanılmıştır. Yazılım kodunun başında öncelikle DDR belleğin adresleri veya transfer sayısı gibi sabitler tanımlanmaktadır. Donanımla alakalı bütün adresler ve bilgiler yazılım koduna dahil edilen *xparameters.h* içerisinde bulunmaktadır.

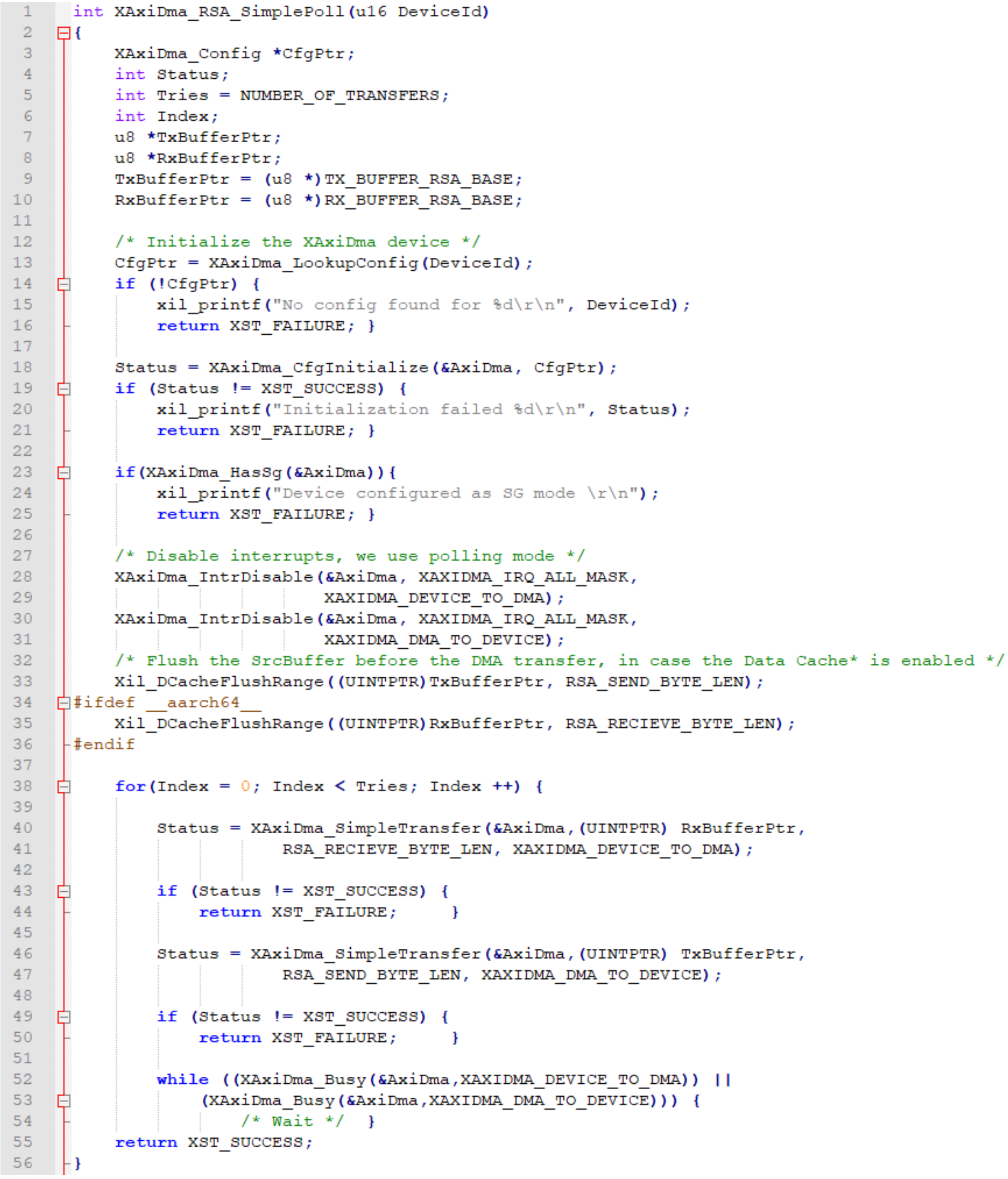
DMA'nın çalışma sistemine uygun bir şekilde RSA modülü ile veri iletimi sağlayabilmek için özel fonksiyonlar yazılmıştır. Örneğin *write_RSA_message_RAM* fonksiyonu, yazılım içinden el ile girilen mesaj verisini DDR belleğe yazmaktadır. Fakat yazma işlemi yapılırken birkaç hususa dikkat edilmiştir. Öncelikle DMA modülünün verileri bellekten çekerken 8 bitlik veri paketleri olarak çektiğinin bilinmesi önem arz etmektedir. Buna dikkat edilmediğinde veriler yanlış çekilecektir. Şekil 3.34'te gösterilen fonksiyon, 32*32 bit boyutlarındaki diziyi *TxBuffer_RSA_Ptr* göstericisine (pointer) her bir 32 bitlik veri paketini 8'er bitlik parçalara ayırarak kaydeder. Döngü 128 defa işleyecek ve *message_array* ın indeksi de en fazla 32'ye kadar gideceğinden bütün mesaj dizisi işlenmiş olacaktır. Aynı şekilde *write_RSA_e_number_RAM* ve *write_RSA_m_number_RAM* fonksiyonları da aynı mantıkta çalışacak fakat tek farkları *TxBuffer_RSA_Ptr* göstericisine her bir bilgi için farklı indekslerle atanacak olmasıdır.



```
void write_RSA_message_RAM(u32 *message_array) {
    int Index;
    u8 *TxBuffer_RSA_Ptr;
    TxBuffer_RSA_Ptr = (u8 *)TX_BUFFER_RSA_BASE;

    for(Index = 0; Index < RSA_SEND_BYTE_LEN-256; Index = Index + 4) {

        TxBuffer_RSA_Ptr[Index] =   message_array[Index/4] & 0x000000FF;
        TxBuffer_RSA_Ptr[Index+1] = (message_array[Index/4]& 0x0000FF00)>>8;
        TxBuffer_RSA_Ptr[Index+2] = (message_array[Index/4]& 0x00FF0000)>>16;
        TxBuffer_RSA_Ptr[Index+3] = (message_array[Index/4]& 0xFF000000)>>24;
    }
}
```

Şekil 3.34 : *write_RSA_message_RAM* fonksiyonu.

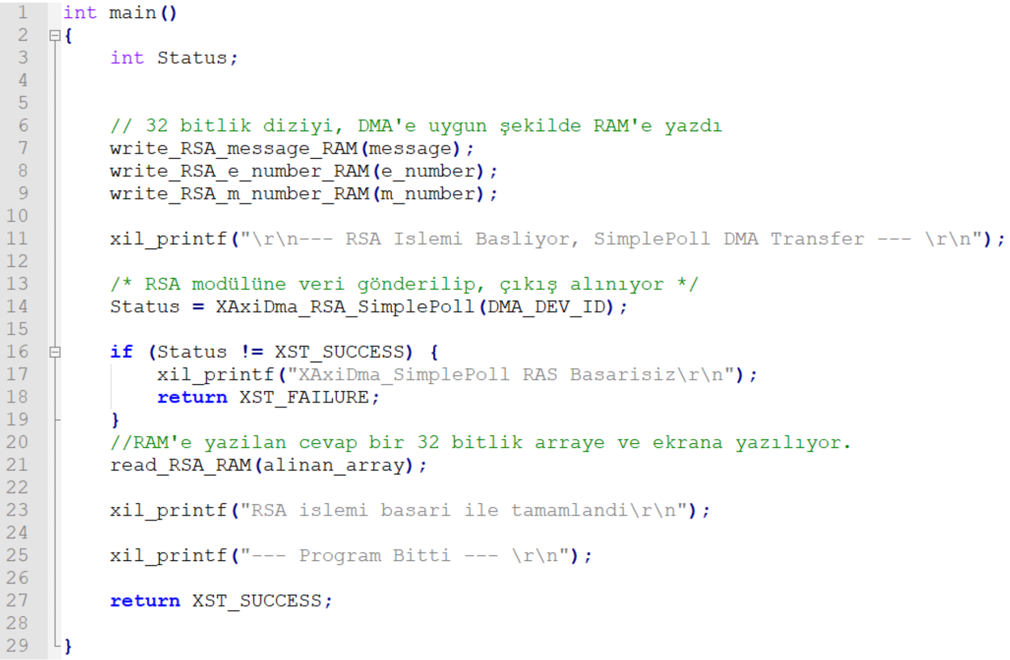
Ardından DMA ile bellekten verinin çekilip RSA modülüne gönderildiği ve RSA işlemi sonucunda alınan verinin tekrardan belleğe yazıldığı temel fonksiyon olan *XAxiDma_RSA_SimplePoll* fonksiyonu Şekil 3.35'te görülmektedir.



```c
int XAxiDma_RSA_SimplePoll(u16 DeviceId)
{
    XAxiDma_Config *CfgPtr;
    int Status;
    int Tries = NUMBER_OF_TRANSFERS;
    int Index;
    u8 *TxBufferPtr;
    u8 *RxBufferPtr;
    TxBufferPtr = (u8 *)TX_BUFFER_RSA_BASE;
    RxBufferPtr = (u8 *)RX_BUFFER_RSA_BASE;

    /* Initialize the XAxiDma device */
    CfgPtr = XAxiDma_LookupConfig(DeviceId);
    if (!CfgPtr) {
        xil_printf("No config found for %d\r\n", DeviceId);
        return XST_FAILURE; }

    Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
    if (Status != XST_SUCCESS) {
        xil_printf("Initialization failed %d\r\n", Status);
        return XST_FAILURE; }

    if(XAxiDma_HasSg(&AxiDma)){
        xil_printf("Device configured as SG mode \r\n");
        return XST_FAILURE; }

    /* Disable interrupts, we use polling mode */
    XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                        XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                        XAXIDMA_DMA_TO_DEVICE);
    /* Flush the SrcBuffer before the DMA transfer, in case the Data Cache* is enabled */
    Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, RSA_SEND_BYTE_LEN);
#ifdef __aarch64__
    Xil_DCacheFlushRange((UINTPTR)RxBufferPtr, RSA_RECIEVE_BYTE_LEN);
#endif

    for(Index = 0; Index < Tries; Index ++) {

        Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) RxBufferPtr,
                    RSA_RECIEVE_BYTE_LEN, XAXIDMA_DEVICE_TO_DMA);

        if (Status != XST_SUCCESS) {
            return XST_FAILURE;     }

        Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) TxBufferPtr,
                    RSA_SEND_BYTE_LEN, XAXIDMA_DMA_TO_DEVICE);

        if (Status != XST_SUCCESS) {
            return XST_FAILURE;     }

        while ((XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)) ||
            (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE))) {
                /* Wait */  }
    return XST_SUCCESS;
}
```

Şekil 3.35 : XAxiDma_RSA_SimplePoll fonksiyonu.

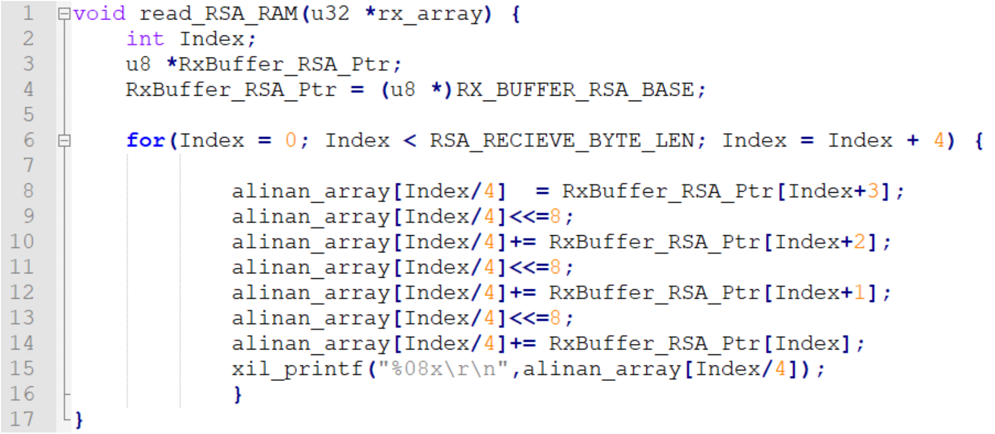
Şekil 3.35'te görülen fonksiyonda öncelikle DMA modülü yapılandırılarak çalışma modu ayarlanmaktadır. Ardından bir *for* döngüsüne girilerek denem sayısı kadar (1 deneme olarak kabul edilmiştir) DMA veri alma ve gönderme işlemi yapmaktadır. *XAxiDma_SimpleTransfer* fonksiyonu ile veri gönderme ve alma işlemleri gerçekleştirilmektedir. Fonksiyonda bulunan *XAXIDMA_DMA_TO_DEVICE* değişkeni DMA modülüne olan doğrultuyu temsil etmektedir. Yani bu değişken kullanıldığı zaman MM2S kanalı aktif edilmektedir. Diğer yön olan *XAXIDMA_DEVICE_TO_DMA* ise *S2MM* kanalını aktif hale getirmektedir. Ayrıca fonksiyonda belirtilmesi gereken değişkenlerden biri de *MAX_PKT_LEN* dir. Bu değişken DMA'ya, işlem yapılacak verinin büyüklüğünün ne kadar olduğunu belirtmektedir. Örneğin yazılımımızda bulunan *RSA_RECIEVE_BYTE_LEN* isimli değişkenin büyüklüğü 128 byte yani 1024 bittir ki bu veri büyüklüğü çıkış FIFO'sundan DMA'ya gönderilecek verinin büyüklüğünü belirtmektedir. Ayrıca bu fonksiyon *Status* isimli bir değişkene de eşitlenerek her adımda herhangi bir hata olup olmadığı kontrol edilmektedir. Son olarak da bir *while* döngüsü içerisinde M2SS ve *S2MM* kanallarının meşgul olup olmadıkları kontrol edilip bütün verilerinin ilgili adreslere gönderildiği tasdiklendikten sonra fonksiyon *XST_SUCCESS* dönütünü vererek başarılı bir biçimde tamamlanmaktadır.



```c
int main()
{
    int Status;


    // 32 bitlik diziyi, DMA'e uygun şekilde RAM'e yazdı
    write_RSA_message_RAM(message);
    write_RSA_e_number_RAM(e_number);
    write_RSA_m_number_RAM(m_number);

    xil_printf("\r\n--- RSA Islemi Basliyor, SimplePoll DMA Transfer --- \r\n");

    /* RSA modülüne veri gönderilip, çıkış alınıyor */
    Status = XAxiDma_RSA_SimplePoll(DMA_DEV_ID);

    if (Status != XST_SUCCESS) {
        xil_printf("XAxiDma_SimplePoll RAS Basarisiz\r\n");
        return XST_FAILURE;
    }
    //RAM'e yazilan cevap bir 32 bitlik arraye ve ekrana yazılıyor.
    read_RSA_RAM(alinan_array);

    xil_printf("RSA islemi basari ile tamamlandi\r\n");

    xil_printf("--- Program Bitti --- \r\n");

    return XST_SUCCESS;

}
```

Şekil 3.36 : DMA kullanılan sistemin ana fonksiyonu.

Son olarak yazılımın ana fonksiyonu tanımlanmıştır. Şekil 3.36'da görüldüğü gibi öncelikle tanımlanmış olan fonksiyonlar yardımı ile RSA protokolüne ait giriş değişkenleri belleğe, DMA'ya uygun biçimde yazılmıştır. Ardından veri alımı ve gönderiminin sağlandığı *XAxiDma_SimpleTransfer* fonksiyonu çağrılmıştır ve görevini düzgün biçimde gerçekleştirip gerçekleştirmediği kontrol edilmiştir. Ardından çağrılan *read_RSA_RAM* fonksiyonu ile belleğe yazılmış sonuç verisi *alinan_array* isimli 1024 bitlik dizide saklanıp terminal ekranında gösterilmiştir. Şekil 3.37'de gösterilen *read_RSA_RAM* fonksiyonu, *write_RSA_e_number_RAM* fonksiyonun tam tersi şekilde işlem yaparak *RxBuffer_RSA_Ptr* göstericisinin işaret ettiği yerdeki sonuç verisini okuma yapabilmek ve düzenli şekilde 32 bitlik paketler haline getirebilmek için yazılmıştır.



```c
void read_RSA_RAM(u32 *rx_array) {
    int Index;
    u8 *RxBuffer_RSA_Ptr;
    RxBuffer_RSA_Ptr = (u8 *)RX_BUFFER_RSA_BASE;

    for(Index = 0; Index < RSA_RECIEVE_BYTE_LEN; Index = Index + 4) {

            alinan_array[Index/4]  = RxBuffer_RSA_Ptr[Index+3];
            alinan_array[Index/4]<<=8;
            alinan_array[Index/4]+= RxBuffer_RSA_Ptr[Index+2];
            alinan_array[Index/4]<<=8;
            alinan_array[Index/4]+= RxBuffer_RSA_Ptr[Index+1];
            alinan_array[Index/4]<<=8;
            alinan_array[Index/4]+= RxBuffer_RSA_Ptr[Index];
            xil_printf("%08x\r\n",alinan_array[Index/4]);
            }
}
```

Şekil 3.37 : *read_RSA_RAM* fonksiyonu.

SDK'da yazılımın tasarlanmasından sonra FPGA programlanmış ve sistemin çalışıp çalışmadığı kontrol edilmek üzere tekrardan Vivado geliştirme platformuna geçiş yapılmıştır. Önceden blok tasarımına eklenmiş olan ILA modülünün kullanımı gerçekleştirilmek üzere donanım yöneticisi (Hardware Manager) açılmıştır. Normalde yapıldığı gibi otomatik bağlanma (Auto Connect) yerine yeni bir donanım hedefi seçilmesi gerekmektedir. Burada dikkat edilmesi gereken noktalardan biri JTAG saat frekansının ILA modülüne bağlı olan saatin frekansından en az iki kat daha küçük olması gerektiğidir. Aksi takdirde doğru ölçüm yapılamayacak ve hata alınacaktır. Ayarlamalar yapılıp ILA ekranında gerekli sinyaller takip edilerek bütün sistemin doğru çalıştığı test edilmiştir.

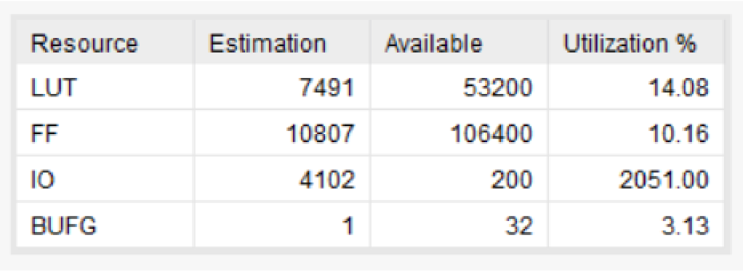
3.3.3 Montgomery çarpım modülü ile VHDL karşılaştırması

Şu ana kadar RSA ile ilgili yapılmış olan bütün tasarımlar HLS ortamında yüksek seviyeli dillerden olan C ve C++ ile yapılmıştır. HLS'te yapılan tasarımların aynı zamanda VHDL kodları da kullanıcıya sunulmaktadır. Yapılan tasarımlarda HLS ortamının kullanılmasının maksadı yapılan tasarımın herhangi bir HDL ile yazılan koddan daha optimize olup olmadığı ve aralarında ne gibi farklılıklar olduğunu (kaynak kullanımı, saat frekansı vb.) araştırmaktır.

Karşılaştırması yapılacak olan iki adet modül, RSA algoritmasının içinde kullanılan ve HLS tasarımı Şekil 3.14'te gösterilen modül ile aynı çarpım işlemini yapan VHDL ile tasarlanmış modüldür. Karşılaştırmalar Vivado tasarım ortamında yapılmıştır. HLS ortamı da tasarım sonrasında belirli tasarım çıktılarını vermektedir fakat fark edilmiştir ki Vivado geliştirme ortamı ile HLS ortamı arasında oldukça büyük farklılıklar bulunmaktadır. Bu sebepten ötürü en doğru sonuçların elde edildiği varsayılan Vivado geliştirme ortamı kullanılmıştır.

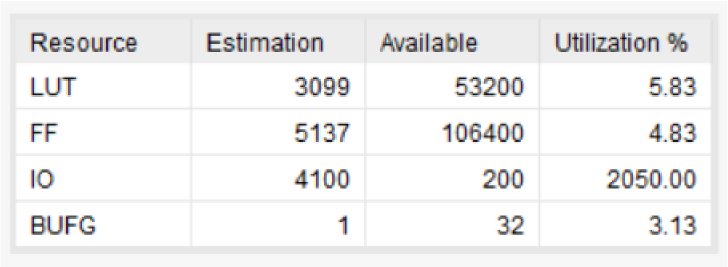
3.3.3.1 Kaynak kullanımı

İki modül de Vivado tasarım ortamında herhangi bir sentez stratejisi uygulanmadan sentezlenmiş ve kaynak kullanımları karşılaştırılmıştır. Şekil 3.38'de görülen veriler HLS tasarımına, Şekil 3.39'daki veriler ise VHDL ile yapılan tasarıma aittir.



Resource	Estimation	Available	Utilization %
LUT	7491	53200	14.08
FF	10807	106400	10.16
IO	4102	200	2051.00
BUFG	1	32	3.13

Şekil 3.38 : HLS tasarımının kaynak kullanımı.



Resource	Estimation	Available	Utilization %
LUT	3099	53200	5.83
FF	5137	106400	4.83
IO	4100	200	2050.00
BUFG	1	32	3.13

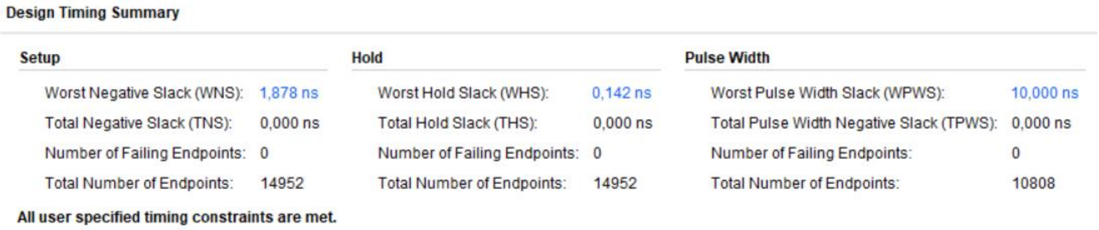
Şekil 3.39 : VHDL tasarımının kaynak kullanımı.

Şekillerde giriş ve çıkışların (IO) planlanmasında %2050 kullanım gözükmektedir. Bu durum herhangi bir sorun teşkil etmeyecektir çünkü modüllerin giriş ve çıkışları 1024 bit olarak tasarlanmıştır. Halihazırda Montgomery çarpımı en üst seviyedeki tasarımın altında kullanılacak bir alt modül olacaktır.

Görüldüğü üzere VHDL ile yapılan tasarımda LUT kullanımında yaklaşık %60'lık, Flip-Flop kullanımında %53'lük bir düşüş gözlenmiştir. Bu karşılaştırma tek başına yeterli olmayacaktır ancak zamanlama analizi ve işlemin gerçekleşmesi için gereken saat darbesi sayısı bulunduğunda anlam kazanacaktır.

3.3.3.2 Zamanlama

İki modülün de en fazla hangi saat frekansında hatasız bir şekilde çalışabileceği analiz edilip bir Montgomery çarpımı işleminin kaç saat darbesi sürdüğü davranışsal benzetim yapılarak analiz edilmiştir. Şekil 3.40 ve Şekil 41'deki zamanlama verileri HLS tasarımına, Şekil 3.42 ve Şekil 3.43'teki veriler ise VHDL tasarımına aittir.



Design Timing Summary

Setup

Worst Negative Slack (WNS): 1,878 ns
Total Negative Slack (TNS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 14952

Hold

Worst Hold Slack (WHS): 0,142 ns
Total Hold Slack (THS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 14952

Pulse Width

Worst Pulse Width Slack (WPWS): 10,000 ns
Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 10808

All user specified timing constraints are met.

Şekil 3.40 : HLS tasarımının zamanlama özeti.

Name	Waveform	Period (ns)	Frequency (MHz)
ap_clk_0	{0.000 10.500}	21.000	47.619

Şekil 3.41 : HLS tasarımının maksimum saat frekansı.

Design Timing Summary

Setup

Worst Negative Slack (WNS): 0,178 ns
Total Negative Slack (TNS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 8226

Hold

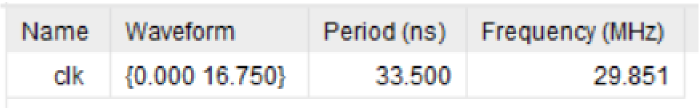
Worst Hold Slack (WHS): 0,132 ns
Total Hold Slack (THS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 8226

Pulse Width

Worst Pulse Width Slack (WPWS): 16,250 ns
Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 5138

All user specified timing constraints are met.

Şekil 3.42 : VHDL tasarımının zamanlama özeti.



Name	Waveform	Period (ns)	Frequency (MHz)
clk	{0.000 16.750}	33.500	29.851

Şekil 3.43 : VHDL tasarımının maksimum saat frekansı.

Zamanlama analizleri yapılırken her iki tasarıma da farklı frekanslarda saat kısıtlamaları eklenip test edilmiş ve en iyi frekans sonuçlarına ulaşılmıştır. Elde edilen veriler Çizelge 3.5'te gösterilmiştir.

Çizelge 3.5 : Farklı şekillerde oluşturulmuş algoritmaların karşılaştırılması.

	Zaman döngüsü sayısı	**Minimum periyot**	**LUT**	**Flip Flop**	**BRAM**
Vivado HLS	4103	≈21 ns	%14.08	%10.16	%0
VHDL	3078	≈33.5 ns	%5.83	%4.83	%0

Elde edilen veriler karşılaştırıldığında, VHDL'de tasarlanan modülün saat frekansının HLS'e göre %40 daha düşük olduğu fakat zaman döngüsü sayısının da %25 daha düşük olduğu görülmüştür. HLS için bir Montgomery çarpımının işlem süresi yaklaşık 86.2 μs, VHDL'de tasarlanan modül için ise 103 μs'dir. Kullanılan kaynak sayısında VHDL modülünün toplamda ortalama %55 daha az kaynak kullandığı belirlenmiştir. Her iki modül de bazı kriterlerde diğerine farklı kullanımlar için daha uygundur. Kullanıcının isteklerine bağlı olarak herhangi biri tercih edilebilir. Fakat VHDL ile yapılan tasarımların bir avantajı daha tasarıma istenildiği gibi müdahale edilebileceğinden yeni algoritmalar ile çok daha performanslı sonuçlar alınabilmesidir.

4. SHA-3 ALGORİTMASI

4.1 Giriş

Kriptografik özetleme fonksiyonları girişindeki rastgele bir uzunluktaki veriyi özetlenmiş bir biçimde sabit bir uzunluktaki çıkışına yansıtan ve girişindeki her değişime bağlı olarak çıkışında da değişimler gözlemlenen fonksiyonlardır [10]. Bu fonksiyonların girişine genelde mesaj, çıkışına ise özet hali denir. Kriptografik özetleme fonksiyonları her farklı giriş için farklı çıkış üretmelidir. Bir kriptografik özetleme fonksiyonu tek yönlü olarak çalışmalıdır, fonksiyonun çıkışındaki veriye bakarak girişindeki veri tahmin edilememelidir. Özetleme fonksiyonları kimlik kanıtlama, sayısal imza oluşturma ve doğrulama, anahtar türetme ve yalancı rastgele bit üretme gibi bilgi güvenliği uygulamalarında çok önemli yer tutmaktadır [4,7,9].

Amerika Ulusal Standartlar ve Teknoloji Enstitüsü (National Institute of Standards and Technology – NIST) tarafından FIPS-202'de standartlaştırılmış SHA-3, permütasyon temelli bir kriptografik fonksiyon takımıdır [4]. 1995'te standartlaştırılan SHA-1 ve 2001'de standartlaştırılan SHA-2'de bulunan güvenlik açıkları nedeniyle 2007 yılında NIST tarafından düzenlenen yarışmada beş finalist arasından seçilen Keccak fonksiyonu SHA-3 fonksiyonlarının temelini oluşturmaktadır [9]. Performans ve güvenlik açısından diğer algoritmalara olan üstünlüğünün yanı sıra donanımsal uygulamalara uygunluğu da Keccak'ın seçilmesinde etkili olmuştur. Bu algoritmanın önceki kriptografik algoritmalardan en önemli farklılıklarından biri kullandığı sünger yapısıdır [4,7,9,10].

SHA-3 fonksiyon takımı altı adet fonksiyondan oluşmaktadır. Bu fonksiyonlardan dördü kriptografik özetleme fonksiyonları olan SHA3-224, SHA3-256, SHA3-384, SHA3-512, ikisi ise genişletilebilir-çıkış fonksiyonları olan SHAKE128, SHAKE256 fonksiyonlarıdır [4]. Kriptografik özetleme fonksiyonlarının sonuna eklenen tire işaretinden sonraki sayıları çıkışlarının uzunluğunu verir. Örneğin, SHA3-256 fonksiyonu 256 bit uzunluğunda çıkış verir. Genişletilebilir-çıkış fonksiyonlarının çıkışları ise istenilen bit uzunluğuna genişletilebilir. Bu fonksiyonların isimlerinin sonundaki sayılar güvenlik derecelerini belirtir. SHAKE128 ve SHAKE256 genişletilebilir-çıkış fonksiyonları SHA-3 fonksiyon takımını oluşturan diğer fonksiyonlardan farklı olsalar da aynı amaçla kullanılarak esneklik sağlayabilmektedir. Yapılan uygulamanın gerekliliği olarak belirtilen kriptografik özetleme fonksiyonlarının çıkış uzunluklarından farklı bir uzunlukta çıkış istenilirse genellikle genişletilebilir-çıkış fonksiyonları kullanılır. SHA-3 fonksiyon takımındaki fonksiyonların sahip olması gereken üç önemli özellik vardır. Bunlar çarpışma dayanıklılığı, öngörüntü saldırılarına dayanıklılık ve ikincil öngörüntü saldırılarına dayanıklılıktır. Çakışma dayanıklılığı, aynı özet değerine sahip iki farklı mesajın bulunamamasıdır. Öngörüntü saldırılarına dayanıklılık, özet değeri bilindiği durumda bile mesajın kendisinin bulunamamasıdır. İkincil öngörüntü saldırılarına dayanıklılık, bilinen bir mesajın özet değerine sahip ikinci bir mesajın bulunamamasıdır. Bu üç duruma olan dayanıklılığın kırılmasının mümkün olmaması gerekir [10].

4.2 Matematiksel Altyapı

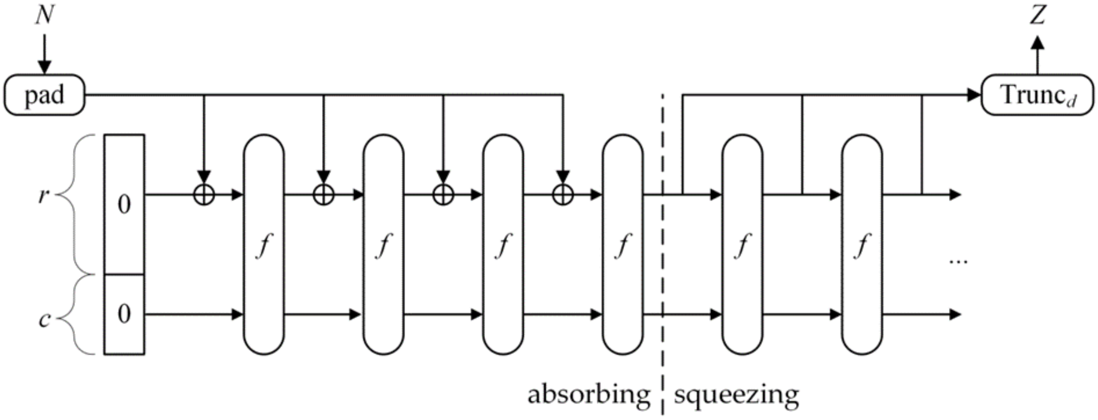
SHA-3 kriptografik özetleme fonksiyonları KECCAK[c] fonksiyonun özel tanımlanmış halleridir. Mesajın sonuna iki bit eklenerek ve çıkışın uzunluğu belirtilerek denklem 4.1'deki gibi tanımlanır [4].

$$\begin{aligned} \text{SHA3-224}(M) &= \text{KECCAK}\,[448]\,(M \,\|\, 01, 224) \\ \text{SHA3-256}(M) &= \text{KECCAK}\,[512]\,(M \,\|\, 01, 256) \\ \text{SHA3-384}(M) &= \text{KECCAK}\,[768]\,(M \,\|\, 01, 384) \\ \text{SHA3-512}(M) &= \text{KECCAK}\,[1024]\,(M \,\|\, 01, 512) \end{aligned} \tag{4.1}$$

Genişletilebilir-çıkış fonksiyonları ise benzer şekilde denklem 4.2'deki gibi tanımlanır.

$$\begin{aligned} \text{SHAKE128}(M, d) &= \text{KECCAK}\,[256]\,(M \,\|\, 1111, d) \\ \text{SHAKE256}(M, d) &= \text{KECCAK}\,[512]\,(M \,\|\, 1111, d) \end{aligned} \tag{4.2}$$

KECCAK[c] fonksiyonları, permütasyonları 1600 bit sayısından oluşmuş KECCAK fonksiyonları olarak tanımlanır. KECCAK fonksiyonları özel bir doldurma kuralı olan sünger fonksiyonu takımıdır ve KECCAK-p permütasyonlarının 1600 bit sayısı ve 24 turdan oluşan hali olan KECCAK-f fonksiyonlarını temel alırlar. Bu doldurma kuralı pad10*1 kuralıdır ve istenilen boyutta çıkışın üretilmesi için ‘0’ bitinin çıkarılması veya yinelenmesi üzerine kuruludur. KECCAK fonksiyonlarının önemli bir özelliği olan sünger yapısı bu doldurma kuralını kullanır. Sünger yapısının iki adet girdisi vardır. Bunlar bir bit dizisi ve bu dizinin uzunluğudur. Bu yapı doldurma, emilme ve sıkma aşamalarından oluşur. Sünger yapısı Şekil 4.1’de görülebilir.



Şekil 4.1 : Sünger Yapısı [4].

Sünger yapısındaki f fonksiyonu b uzunluğundaki bit dizilerini alır. Oran olarak gösterilen r, b sayısından daha küçük bir pozitif sayıdır. Kapasite olarak gösterilen c pozitif bir sayıdır ve denklem 4.3’teki biçimde gösterilir. Mesaj r bitlik bloklara ayrılır. Bu durumda güvenlik seviyesini belirten c genellikle d ile ifade edilen çıkış uzunluğunun iki katıdır ve denklem 4.4’teki biçimde ifade edilir.

$$c = b - r \tag{4.3}$$

$$c = 2 * d \tag{4.4}$$

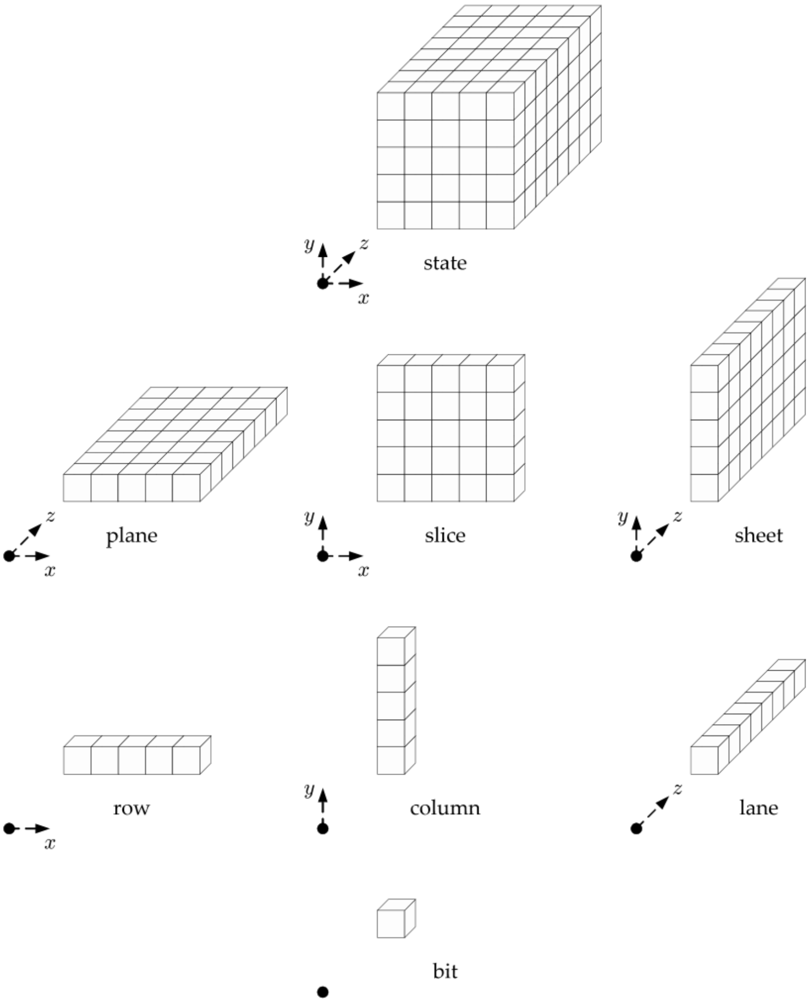
4.2.1 KECCAK-p permütasyonları

KECCAK-p permütasyonları sırası değiştirilecek sabit bir boyut ve yinelenme sayıları ile ifade edilir. Boyut b ile gösterilir ve {25,50,100,200,400,800,1600} değerlerini alabilir. KECCAK-p permütasyonlarının bir yinelemesi basamak eşleme denilen beş adımdan oluşur. Durum adı verilen b bitlik diziler başlangıçta giriş değerlerine eşittir ve bu adımlar sırasında güncellenir [4].

4.2.2 Durum dizisi

Bir durum *b* bitten oluşur, b/25 değerine *w*, $\log_2$(b/25) değerine *l* adı verilir. Buna göre *b*, *w* ve *l* değerlerinin alabileceği 7 farklı değerler vardır. Bu değerler *b* için

{25,50,100,200,400,800,1600}, w için {1,2,4,8,16,32,64}, l için {0,1,2,3,4,5,6} değerleridir [4]. Durumlar 5x5xw şeklinde bit dizileri olarak şeklinde tasvir edilir. Durumu S ile gösterirsek durumu oluşturan bitler denklem 4.5'teki biçimde gösterilebilir. Tanımlanan bu üç boyutlu dizinin indeksleri x, y, z ile olarak ifade edilirse Şekil 4.2'teki gibi gösterilebilir.

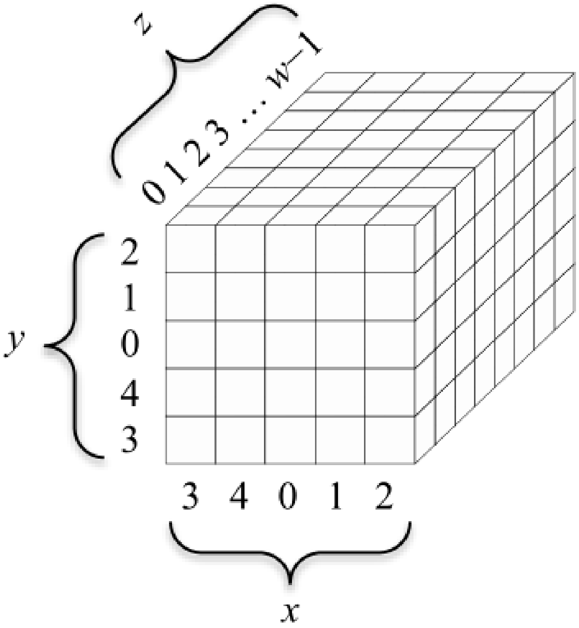


Şekil 4.2 : Durum dizisi [4].

$$S = S\,[0]\;||\;S\,[1]\;||\;\ldots\;||\;S\,[b-2]\;||\;S\,[b-1] \tag{4.5}$$

Şekil 4.2'deki gösterimde x koordinatı sıfırdan beşe kadar, y koordinatı sıfırdan beşe kadar, z koordinatı ise sıfırdan w'ya kadar değerler alır. Üç boyutlu durum dizisinin şekilde görüldüğü gibi üç adet iki boyutlu alt dizisi ve üç adet tek boyutlu alt dizisi vardır.

Durum dizisi Şekil 4.3'te gösterilen şekilde numaralandırılır. Bu numaralandırma basamak eşlemeler sırasında kullanılır.



Şekil 4.3 : Koordinatların numaralandırılması [4].

4.2.3 Basamak eşleme

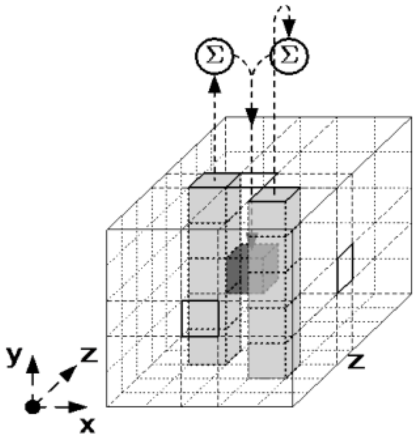
KECCAK-p permütasyonları θ, ρ, π, χ,ι şeklinde tanımlanan beş adet basamak eşleme içerir. Bir yineleme bu beş basamak eşlemenin sırayla uygulanmasıyla oluşur. Rnd ile gösterilen bir yinelemedeki basamaklar denklem 4.6'daki şekilde sıralanır. Buradaki i_r yineleme indeksidir [4].

$$\mathrm{Rnd}(\mathrm{A}, \mathrm{ir}) = \iota(\chi(\pi(\rho(\theta(\mathrm{A})))), \mathrm{ir}) \tag{4.6}$$

4.2.3.1 θ fonksiyonu

θ basamağı durumdaki her biti Şekil 4.4'te gösterildiği gibi ikili sütunlar halinde XOR işlemine tabi tutar. Bu basamak aşağıdaki üç adımdan oluşur [4]:

```
θ fonksiyonu algoritması:
1. $C[x,z] = S[x,0,z] \oplus S[x,1,z] \oplus S[x,2,z] \oplus S[x,3,z] \oplus S[x,4,z]$
2. $D[x,z] = C[(x-1) mod\ 5, z] \oplus C[(x+1) mod\ 5, (z-1) mod\ w]$
3. $S'[x,y,z] = S[x,y,z] \oplus D[x,z]$
```



Şekil **4.4 :** θ'nın görselleştirilmesi [4].

4.2.3.2 ρ fonksiyonu

ρ basamağı her şeritte bulunan bitleri Şekil 4.5'te gösterildiği gibi belirli bir uzunluk boyunca z koordinatında öteler. Bu uzunluk şeridin x ve y koordinatlarına bağlıdır. Bu öteleme uzunlukları Çizelge 4.1'deki gibidir.

Çizelge 4.1: Öteleme uzunlukları [4].

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	153	231	3	10	171
$y=1$	55	276	36	300	6
$y=0$	28	91	0	1	190
$y=4$	120	78	210	66	253
$y=3$	21	136	105	45	15

Her bitin ne kadar öteleneceği tablodaki değerlerin w'ya göre mod işlemine tabi tutulmasıyla bulunur. Örneğin w=4 için y=1 ve x=2 için bu değer 2 olur. ρ basamağı 4 adımdan oluşur. Bu adımlar aşağıda görüldüğü gibidir [4]:

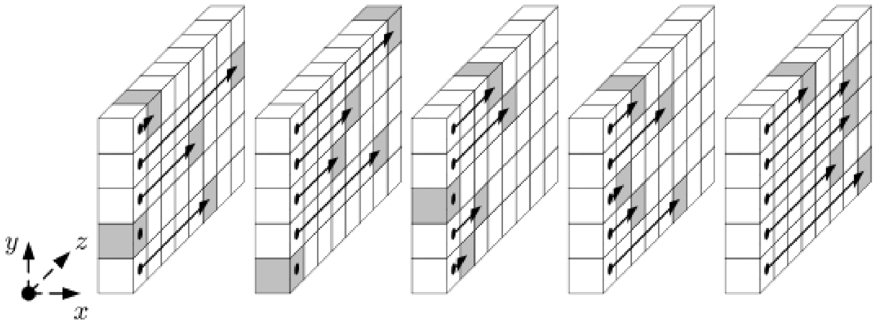
ρ fonksiyonu algoritması:

```
1. $0 \leq z < w, S'[0,0,z] = S[0,0,z]$
2. $(x,y) = (1,0)$
3. t sıfırdan yirmi üçe kadar:
```

$$\boldsymbol{a.\ 0 \le z < w, S'[x,y,z] = S\left[x,y,\left(z - \frac{(t+1)(t+2)}{2}\right) mod\ w\right]}$$

$$\boldsymbol{b.\ (x,y) = (y,(2x + 3y) mod\ 5)}$$

4. S' çıkışa verilir.



Şekil 4.5 : ρ'nun görselleştirilmesi [4].

4.2.3.3 π fonksiyonu

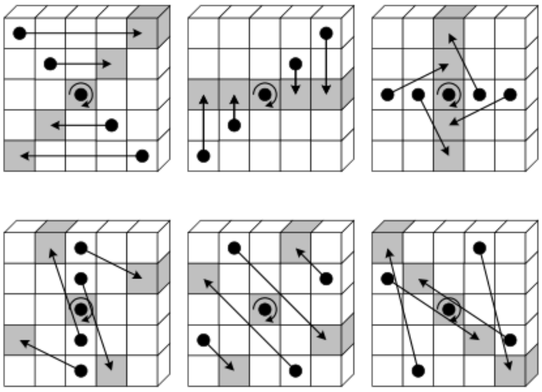
π basamağı Şekil 4.6'da gösterildiği gibi şeritlerin pozisyonunu yeniden düzenler. π basamağı iki adımdan oluşur. Bu adımlar aşağıda gösterildiği gibidir [4]:

π fonksiyonu algoritması:

1. $0 \le x < 5, 0 \le y < 5, 0 \le z < w$

$$S'[x,y,z] = S[(x + 3y) mod\ 5, x, z]$$

2. S' çıkışa verilir.



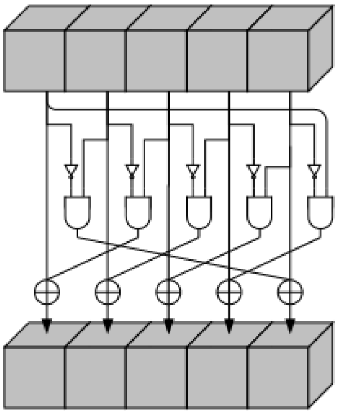
Şekil 4.6 : π’nin görselleştirilmesi [4].

4.2.3.4 χ fonksiyonu

χ basamağı Şekil 4.7’de görüldüğü gibi her biti aynı sıradaki diğer bitlerle XOR işlemine tabi tutar. χ basamağı iki adımdan oluşur. Bu adımlar aşağıda gösterildiği gibidir [4]:

χ fonksiyonu algoritması:

```
1. $0 \leq x < 5, 0 \leq y < 5, 0 \leq z < w$
   $S'[x, y, z] = S[x, y, z] \oplus \left((S[(x + 1) mod\ 5, y, z] \oplus 1) AND S[(x + 2) mod\ 5, y, z]\right)$
2. $S'$ çıkışa verilir.
```



Şekil 4.7 : χ’nin görselleştirilmesi [4].

4.2.3.5 ι fonksiyonu

ι basamağında (0,0) şeridindeki bitler yineleme indeksine bağlı olarak değiştirilir ve diğer şeritler bu basamakta değişiklik göstermez. ι basamağı aşağıdaki gibi 5 adımdan oluşur [4]:

ι fonksiyonu algoritması:

```
1. $0 \leq x < 5, 0 \leq y < 5, 0 \leq z < w$
        $S'[x, y, z] = S[x, y, z]$.
2. $RC = 0^w$
3. $j$ sıfırdan bire kadar, $RC[2j - 1] = rc(j + 7ir)$.
4. $0 \leq z < w$,     $S'[0, 0, z] = S'[0, 0, z] \oplus RC[z]$.
5. $S'$ çıkışa verilir.
```

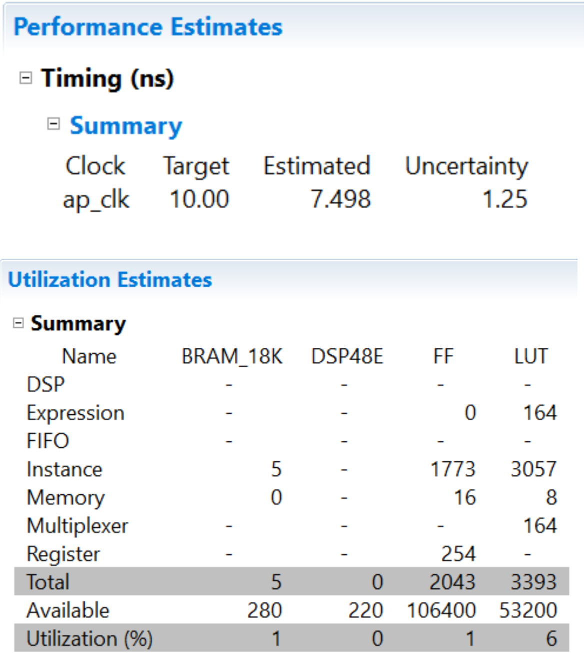
Bu adımlardaki rc fonksiyonu 4 adımdan oluşur ve yineleme sabitini üretir.

```
1. $t \bmod 255 = 0$ ise $1$ çıkışı verir
2. $R = 10000000$
3. $i$ birden $(t \bmod 255)$'e kadar
        a. $R = 0 || R$
        b. $R[0] = R[0] \oplus R[8]$
        c. $R[4] = R[4] \oplus R[8]$
        d. $R[5] = R[5] \oplus R[8]$
        e. $R[6] = R[6] \oplus R[8]$
        f. $R = Trunc_8[R]$
4. $R[0]$ çıkışa verilir.
```

4.3 Tasarım ve Gerçekleme

Tasarıma Vivado HLS ortamında kullanmak üzere C++ dilinde SHA-3 algoritmasının kodu yazılarak başlanmıştır. Tasarım sırasında sıkça kullanılacak olan matematiksel işlemler önce ayrı ayrı fonksiyonlar halinde yazılıp sonra da kullanılacakları şekilleriyle tanımlanarak kolaylık sağlanmıştır. KECCAK-f fonksiyonu, sünger yapısı ayrı ayrı fonksiyonlar şeklinde yazılmıştır ve sünger yapısı yazılırken bu fonksiyonun içinde KECCAK-f fonksiyonu çağırılarak kullanılmıştır.

Tasarımda kullanılmak üzere SHA-3 fonksiyon takımındaki kriptografik özetleme fonksiyonlarından olan SHA3-512 fonksiyonu seçilmiştir. SHA3-512 fonksiyonun özelliklerini sağlayacak şekilde bir üst fonksiyon yazılarak sünger yapısına bu parametreler gönderilmiştir. Bu aşamalardan sonra yazılan kod Vivado HLS’te tekrar yazılmıştır. Kodun doğru çalıştığını test etmek için test dosyası hazırlanmıştır ve yapılan testlerin sonucunda SHA3-512 fonksiyonun doğru bir şekilde çalıştığı görülmüştür. Bu aşamalardan sonra modül sentezlenerek FPGA üzerinde kullandığı alana ve kullanılabilecek minimum saat periyodu incelenmiştir. Sentez sonuçları Şekil 4.8’de gösterilmiştir. Tasarımda kullanılacak diğer modüller olan AES ve RSA modülleriyle beraber kullanılacağından FPGA üzerinde kullandığı alanın yeterince küçük olduğuna karar verilmiştir.



Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.498	1.25

Utilization Estimates

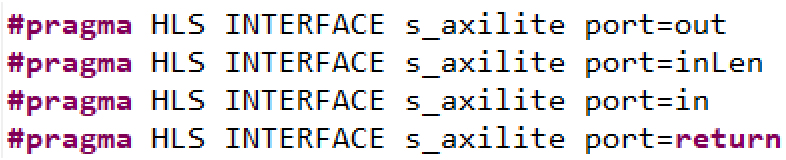
⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	164
FIFO	-	-	-	-
Instance	5	-	1773	3057
Memory	0	-	16	8
Multiplexer	-	-	-	164
Register	-	-	254	-
Total	5	0	2043	3393
Available	280	220	106400	53200
Utilization (%)	1	0	1	6

Şekil 4.8 : SHA-3 algoritması zamanlama değerleri ve kaynak kullanımı.

4.3.1 AXI4-Lite arayüzü ile tasarlanan modül

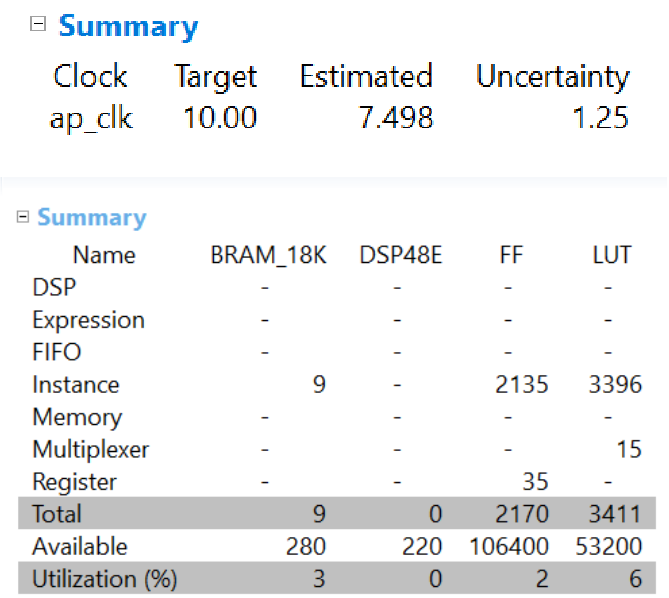
HLS ortamında tasarlanan SHA3-512 modülünün dışarı aktarılarak Vivado ortamında mikroişlemci tabanlı sistemlerde kullanılabilmesi için tasarıma arayüz eklenmesi gerekmektedir. Dışarıya aktarılan modülün mikroişlemci ile haberleşebilmesi için AXI4-Lite arayüzü eklenmiştir. AXI4-Lite arayüzü, HLS ortamında direktifler yazarak eklenebilir. Bu amaçla yazılmış direktifler Şekil 4.9'da gösterilmiştir.



```
#pragma HLS INTERFACE s_axilite port=out
#pragma HLS INTERFACE s_axilite port=inLen
#pragma HLS INTERFACE s_axilite port=in
#pragma HLS INTERFACE s_axilite port=return
```

Şekil 4.9 : AXI4-Lite arayüzü direktifleri.

Şekil 4.9'da yazılan direktifler giriş ve çıkış portlarının AXI4-Lite arayüzü ile sentezlenebilmesini sağlamıştır. Bu direktiflerle yapılan sentezin sonuçları Şekil 4.10'da gösterilmiştir. Sentez sonuçları incelenerek zamanlama değerleri ve kaynak kullanımının uygun olduğu, giriş ve çıkışların doğru şekilde istenilen arayüzle sentezlendiği görüşmüştür. Sentez sonrasında AXI4-Lite arayüzü eklenmesi nedeniyle kaynak kullanımının arttığı gözlemlenmiştir.



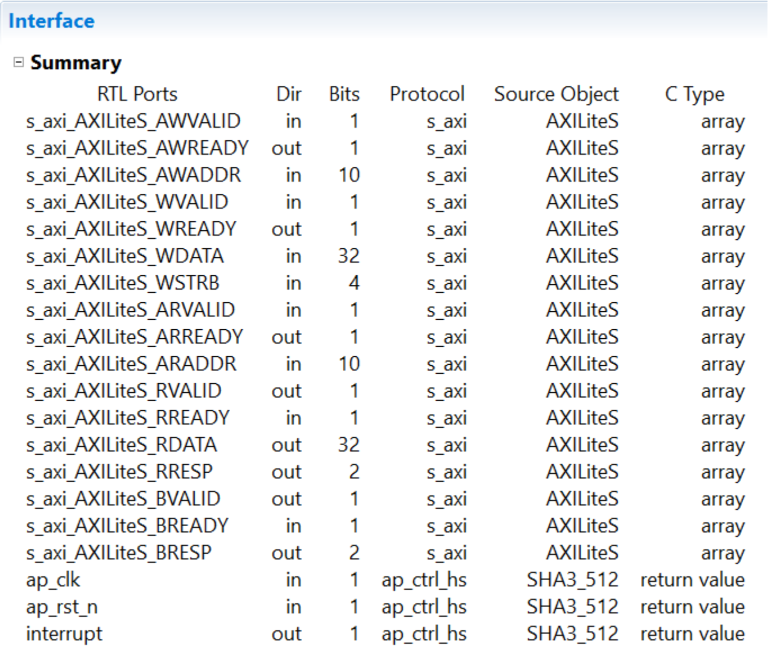
Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.498	1.25

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	9	-	2135	3396
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	35	-
Total	9	0	2170	3411
Available	280	220	106400	53200
Utilization (%)	3	0	2	6

Şekil 4.10 : AXI4-Lite arayüzü sentez sonuçları.



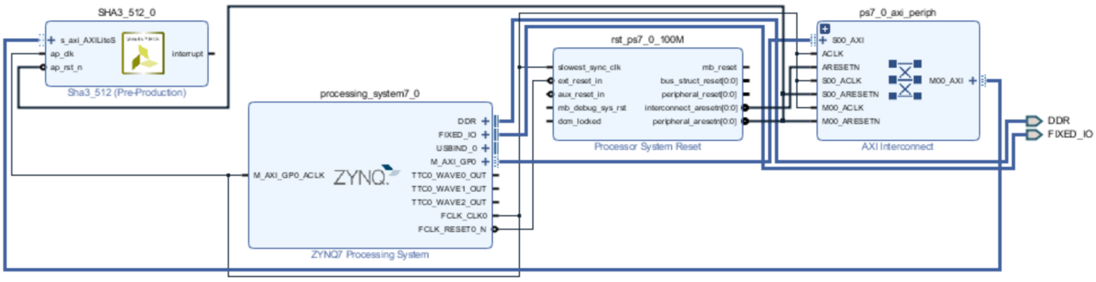
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_AXILiteS_AWVALID	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_AWREADY	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_AWADDR	in	10	s_axi	AXILiteS	array
s_axi_AXILiteS_WVALID	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_WREADY	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_WDATA	in	32	s_axi	AXILiteS	array
s_axi_AXILiteS_WSTRB	in	4	s_axi	AXILiteS	array
s_axi_AXILiteS_ARVALID	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_ARREADY	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_ARADDR	in	10	s_axi	AXILiteS	array
s_axi_AXILiteS_RVALID	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_RREADY	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_RDATA	out	32	s_axi	AXILiteS	array
s_axi_AXILiteS_RRESP	out	2	s_axi	AXILiteS	array
s_axi_AXILiteS_BVALID	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_BREADY	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_BRESP	out	2	s_axi	AXILiteS	array
ap_clk	in	1	ap_ctrl_hs	SHA3_512	return value
ap_rst_n	in	1	ap_ctrl_hs	SHA3_512	return value
interrupt	out	1	ap_ctrl_hs	SHA3_512	return value

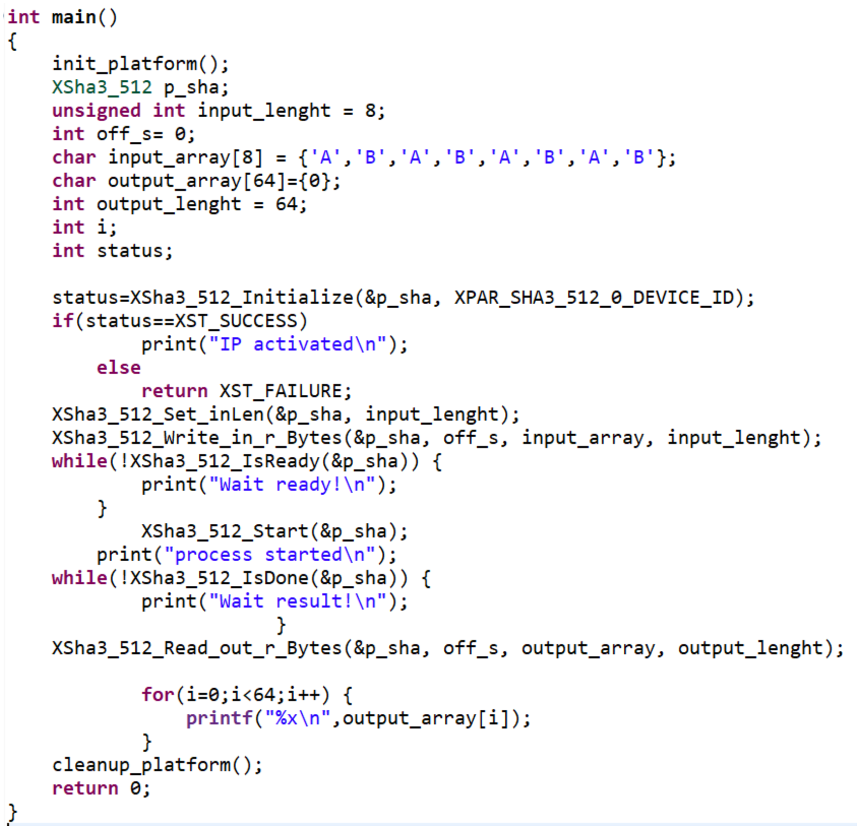
Şekil 4.10 (devam) : AXI4-Lite arayüzü sentez sonuçları.

AXI4-Lite arayüzü eklenmiş SHA3-512 modülü HLS ortamından dışarıya aktarılarak Vivado ortamına modül bloğu olarak eklenmiştir. Vivado ortamında ZYNQ işlemciyi ve aktardığımız SHA3-512 modülünü ekleyebilmek için blok tasarımı oluşturulmuştur. Bu tasarıma eklenebilmesi için SHA3-512 modülü veri deposuna eklenmiştir. Tasarımın son hali Şekil 4.11'de gösterilmiştir.



Şekil 4.11 : AXI4-Lite arayüzü kullanılan blok şeması.

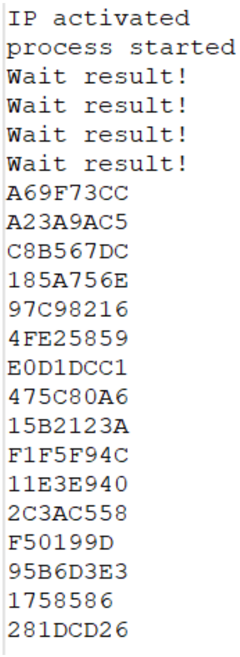
Blok şeması oluşturulduktan sonra ZYNQ işlemci üzerinden SHA3-512 modülünün kontrol edilebilmesi için SDK ortamına geçilmiştir. SDK ortamında yazılan kodlar Şekil 4.12'de gösterilmiştir. SHA3-512 modülünü HLS'ten aktardığımızdan dolayı kontrolü için gereken sürücüler otomatik olarak oluşturulmuş ve Vivado ortamına aktarılmıştır.



```
int main()
{
    init_platform();
    XSha3_512 p_sha;
    unsigned int input_lenght = 8;
    int off_s= 0;
    char input_array[8] = {'A','B','A','B','A','B','A','B'};
    char output_array[64]={0};
    int output_lenght = 64;
    int i;
    int status;

    status=XSha3_512_Initialize(&p_sha, XPAR_SHA3_512_0_DEVICE_ID);
    if(status==XST_SUCCESS)
            print("IP activated\n");
        else
            return XST_FAILURE;
    XSha3_512_Set_inLen(&p_sha, input_lenght);
    XSha3_512_Write_in_r_Bytes(&p_sha, off_s, input_array, input_lenght);
    while(!XSha3_512_IsReady(&p_sha)) {
            print("Wait ready!\n");
        }
            XSha3_512_Start(&p_sha);
        print("process started\n");
    while(!XSha3_512_IsDone(&p_sha)) {
            print("Wait result!\n");
                        }
    XSha3_512_Read_out_r_Bytes(&p_sha, off_s, output_array, output_lenght);

            for(i=0;i<64;i++) {
                printf("%x\n",output_array[i]);
            }
    cleanup_platform();
    return 0;
}
```

Şekil 4.12 : SDK ortamında SHA3-512 modülünün kontrolü için yazılan kod.

Otomatik olarak oluşturulmuş fonksiyonlar kullanılarak modülün girişlerine istenilen değerler gönderilmiş ve doğru çıkış değerinin alınması sağlanmıştır. Başlangıçta kullanılacak değişkenler, giriş dizisinin değerleri, giriş dizisinin uzunluğu ve çıkış tanımlanmıştır. *XSha3_512_Initialize* fonksiyonu ile cihaz kimliği tanımlanmıştır. *XSha3_512_Set_inLen* fonksiyonu modülün doğru çalışması için gerekli olan giriş dizisinin uzunluğunu modüle göndermek amacıyla kullanılmıştır. *XSha3_512_Write_in_r_Bytes* fonksiyonu sayesinde modülün girişine gönderilecek veri dizisi gönderilmiştir. *XSha3_512_IsReady* fonksiyonu kullanılarak modülün hazır olup olmadığı kontrol edilmiştir. *XSha3_512_Start* fonksiyonu ile işlem başlatılmıştır. *XSha3_512_IsDone* fonksiyonu kullanılarak işlem sürdüğü sürece sonucun beklendiği terminalde yazılmıştır. *XSha3_512_Read_out_r_Bytes* fonksiyonu ile modülün çıkışındaki değerler okunmuştur. Okunan değerlerin doğruluğu NIST tarafından paylaşılmış örnek sonuçlarla kıyaslanarak teyit edilmiştir. Elde edilen sonuç Şekil 4.13'te gösterilmiştir.

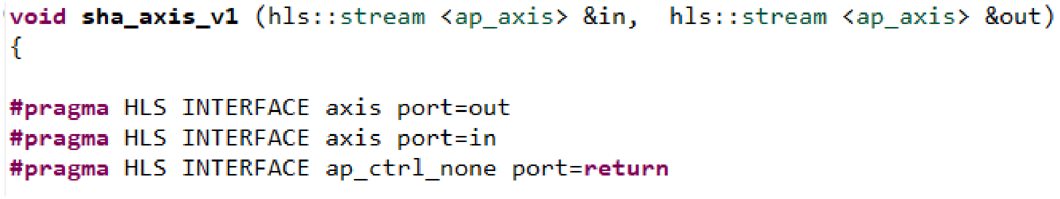


```
IP activated
process started
Wait result!
Wait result!
Wait result!
Wait result!
A69F73CC
A23A9AC5
C8B567DC
185A756E
97C98216
4FE25859
E0D1DCC1
475C80A6
15B2123A
F1F5F94C
11E3E940
2C3AC558
F50199D
95B6D3E3
1758586
281DCD26
```

Şekil 4.13 : SDK ortamında yapılan sistem testi sonucu.

4.3.2 AXI4-Stream arayüzü ile tasarlanan modül

AXI4-Lite arayüzü eklenerek dışarıya aktarılmış olan SHA3-512 modülü AXI4-Stream arayüzü ile yeniden tasarlanmıştır. AXI4-Stream arayüzünü gerçeklemek için HLS ortamında yazılan direktifler Şekil 4.14'te gösterilmiştir. AXI4-Stream arayüzü tasarlanırken giriş ve çıkışlar birer tane olacak şekilde tasarlanmıştır. Çıkış sayısı önceki tasarımda da bir tane olmasına rağmen iki tane olan giriş sayısı bire indirilmiştir.

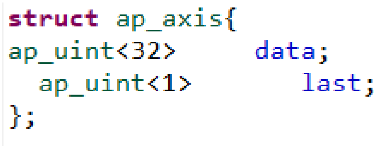


```
void sha_axis_v1 (hls::stream <ap_axis> &in,  hls::stream <ap_axis> &out)
{

#pragma HLS INTERFACE axis port=out
#pragma HLS INTERFACE axis port=in
#pragma HLS INTERFACE ap_ctrl_none port=return
```

Şekil 4.14 : AXI4-Stream arayüzü direktifleri.

Tasarımda HLS nesneleri kullanılarak ilgili kütüphane üzerinden AXI4-Stream arayüzünün özelliklerinin kullanılması sağlanmıştır. AXI4-Stream arayüzü ile kullanmak için yeniden düzenlenen modül girişine ve çıkışına birer FIFO bağlanacak şekilde tasarlanmıştır. Girişindeki FIFO'nun dolu ya da boş olmasını kontrol ederek FIFO boşsa bekleyecek ve FIFO dolduğunda girişinden veri almaya başlayacaktır.

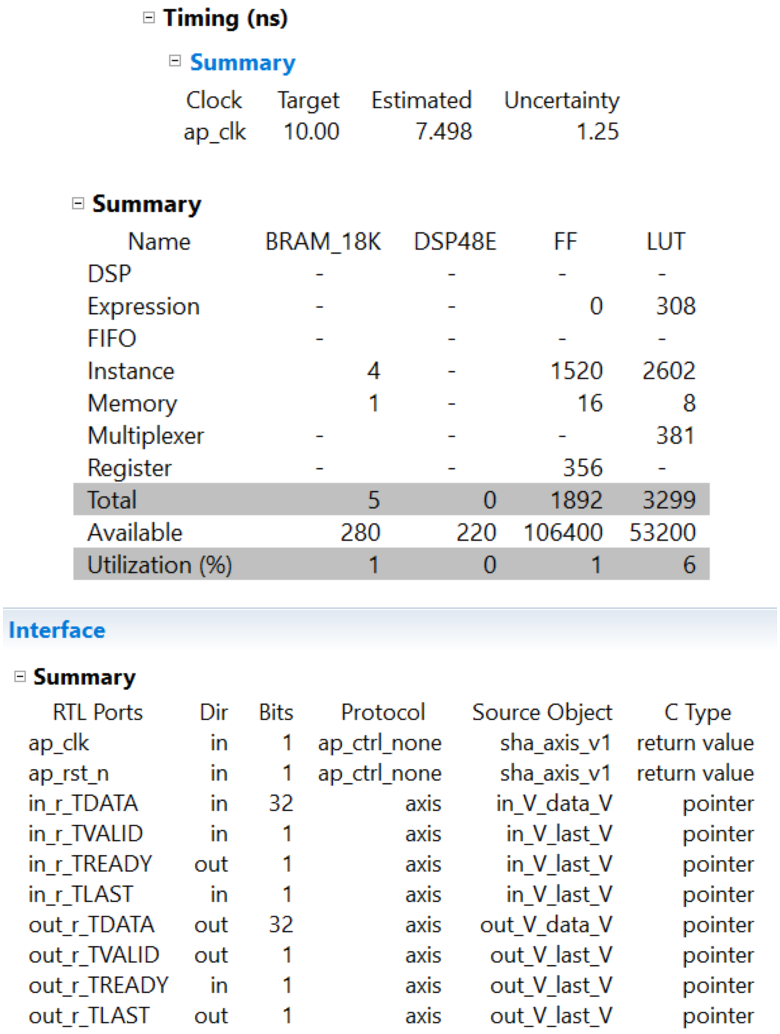
Modülün FIFO’dan okuduğu ilk veri alınacak verinin toplam boyut bilgisini içerecektir. Daha sonrasında FIFO dolu olduğu sürece 32 bitlik paketler halinde girişinden veri okumaya devam edecektir. FIFO boşalana kadar veriyi alıp SHA-3 fonksiyonunun gerçekleştirdiği işlemler bittikten sonra yine 32 bitlik paketler halinde sonuç verisini çıkışındaki FIFO’ya yazmaktadır. AXI4-Stream arayüzü ile tasarlanmış SHA3-512 modülü dışarıya aktarıldıktan sonra kullanılacak sistemde DMA yer aldığı için sentezlemeden önce bu modüle *last* sinyali eklenmesi gerekmektedir. Bu sinyal eklenmediği halde sistem düzgün bir biçimde çalışmayacaktır. Bu sinyalin yapısı ve nasıl eklendiği Şekil 4.15’te gösterilmiştir.



```
struct ap_axis{
ap_uint<32>     data;
  ap_uint<1>       last;
};
```

Şekil 4.15 : Modülün giriş ve çıkışları için kullanılan yapı.

Görüldüğü gibi giriş ve çıkışlardaki sinyallerin sağlanabilmesi için C dilindeki *struct* denilen yapı kullanılmıştır. Tasarlanan modülde çıkıştaki FIFO’ya son 32 bitlik veri paketini yazarken *last* sinyalini vermektedir. Modülün AXI4-Stream özellikleriyle birlikte doğru çalıştığını test etmek için HLS ortamında bir test dosyası oluşturulmuş ve istenilen şekilde çalıştığı gözlemlenmiştir. Bu testler üzerine modül sentezlenmiştir ve sentez sonuçları Şekil 4.16’da gösterilmiştir.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.498	1.25

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	308
FIFO	-	-	-	-
Instance	4	-	1520	2602
Memory	1	-	16	8
Multiplexer	-	-	-	381
Register	-	-	356	-
Total	5	0	1892	3299
Available	280	220	106400	53200
Utilization (%)	1	0	1	6

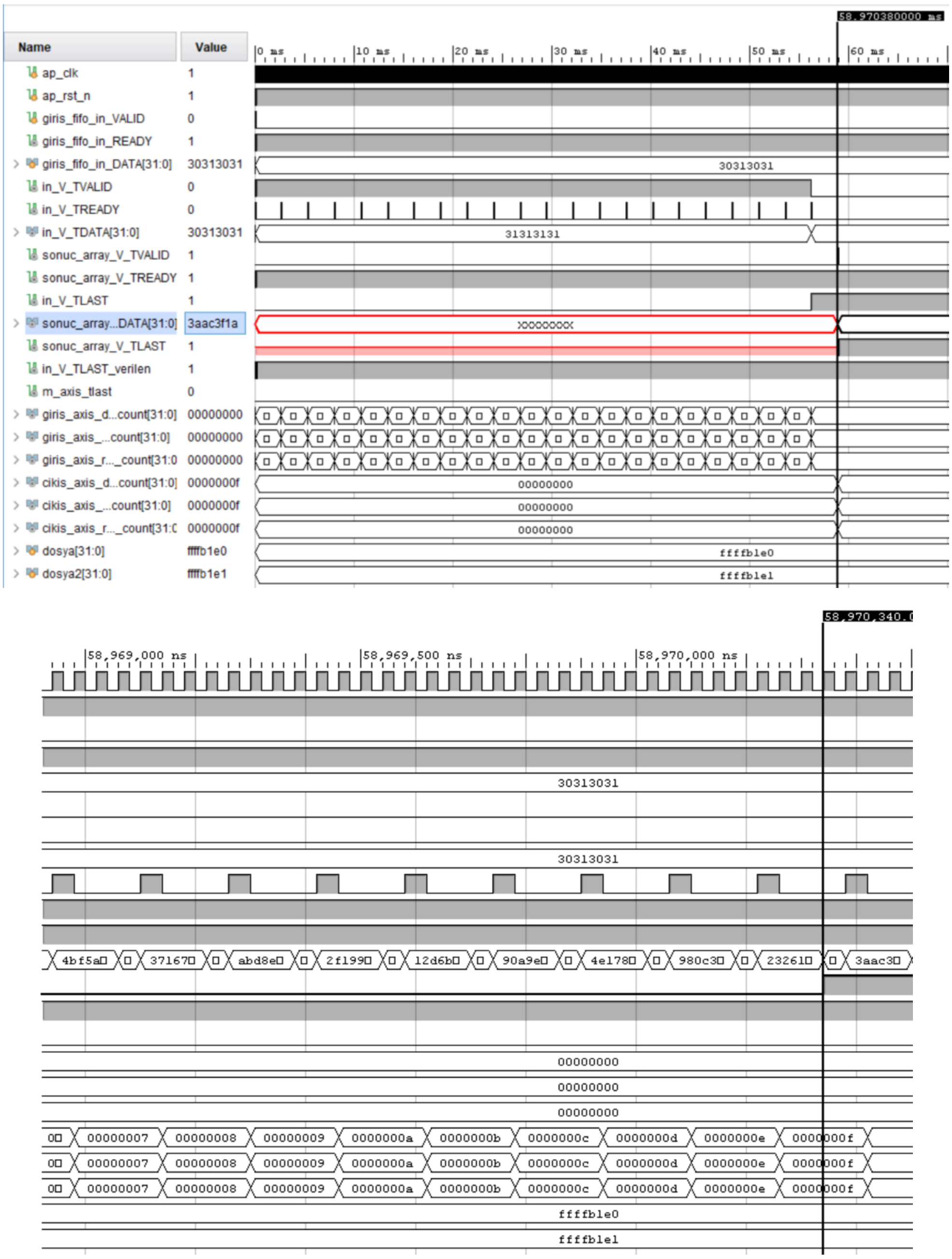
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	sha_axis_v1	return value
ap_rst_n	in	1	ap_ctrl_none	sha_axis_v1	return value
in_r_TDATA	in	32	axis	in_V_data_V	pointer
in_r_TVALID	in	1	axis	in_V_last_V	pointer
in_r_TREADY	out	1	axis	in_V_last_V	pointer
in_r_TLAST	in	1	axis	in_V_last_V	pointer
out_r_TDATA	out	32	axis	out_V_data_V	pointer
out_r_TVALID	out	1	axis	out_V_last_V	pointer
out_r_TREADY	in	1	axis	out_V_last_V	pointer
out_r_TLAST	out	1	axis	out_V_last_V	pointer

Şekil 4.16 : AXI4-Stream arayüzü sentez sonuçları.

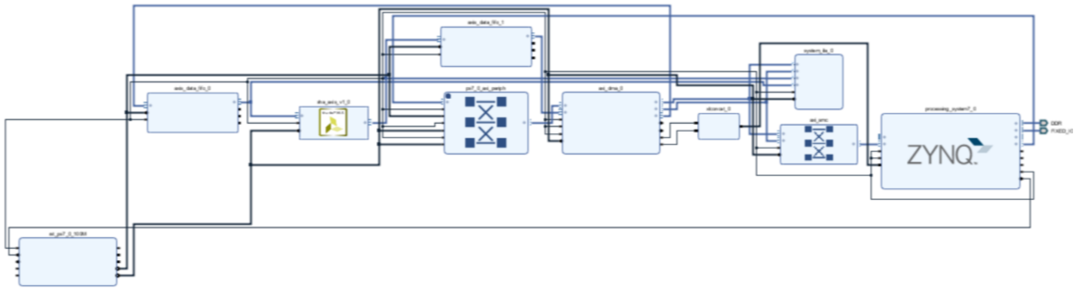
Görüldüğü gibi bir giriş sinyali, bir çıkış sinyali ve bunlara eklenmiş *last* sinyali oluşmuştur. Tasarım amaçlanan AXI4-Stream arayüzüne sahiptir. HLS, FIFO'ları otomatik eklemediği için sonradan Vivado ortamında eklenecektir. Sentezleme işlemi de başarıyla tamamlandıktan sonra modül dışarıya aktarılarak Vivado ortamına geçilmiştir. HLS ortamında doğru çalıştığı test edilen modülün bu sefer Vivado ortamında doğru çalıştığıyla ilgili testler yapmak amacıyla bir test dosyası oluşturulmuştur. Yapılan test sonucunda doğru bir şekilde çalıştığı, sinyallerin istenilen zamanda çıktığı gözlenmiştir. Benzetim sonucu Şekil 4.17'de gösterilmiştir.



Name	Value
ap_clk	1
ap_rst_n	1
giris_fifo_in_VALID	0
giris_fifo_in_READY	1
giris_fifo_in_DATA[31:0]	30313031
in_V_TVALID	0
in_V_TREADY	0
in_V_TDATA[31:0]	30313031
sonuc_array_V_TVALID	1
sonuc_array_V_TREADY	1
in_V_TLAST	1
sonuc_array...DATA[31:0]	3aac3f1a
sonuc_array_V_TLAST	1
in_V_TLAST_verilen	1
m_axis_tlast	0
giris_axis_d...count[31:0]	00000000
giris_axis_...count[31:0]	00000000
giris_axis_r..._count[31:0	00000000
cikis_axis_d...count[31:0]	0000000f
cikis_axis_...count[31:0]	0000000f
cikis_axis_r..._count[31:(	0000000f
dosya[31:0]	ffffb1e0
dosya2[31:0]	ffffb1e1

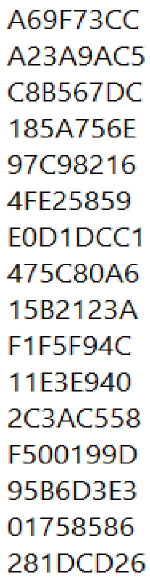
Şekil 4.17 : SHA3-512 AXI4-Stream modülü Vivado benzetim sonucu.

Benzetim sonucunda görüldüğü gibi modülün çıkışındaki FIFO'ya son 32 bitlik paket yazılmadan önce *last* sinyali çıkmıştır ve modülün sistemde DMA ile birlikte doğru şekilde çalışacağına karar verilmiştir. Bu aşamadan sonra dışarıya aktarılmış olan SHA3-512 modülü ZYNQ işlemci tarafından kontrol edilen ve DMA içeren bir sistemde kullanılmak üzere yeni bir Vivado projesi oluşturulmuştur. Bu projeye ZYNQ işlemciyi, DMA'yı, FIFO'ları ve tasarladığımız SHA3-512 modülünü ekleyebilmek için bir blok şeması oluşturulmuştur.



Şekil 4.18 : AXI4-Stream arayüzü kullanılan blok şeması.

Şekil 4.18’de görüldüğü sisteme ZYNQ işlemci, DMA ve tasarlanan modül eklenmiştir. Modülün girişine ve çıkışına FIFO’lar bağlanmıştır. Daha önceden tasarlandığı gibi modül giriş FIFO’su dolu olduğunda çalışmaya başlayacak ve işlemleri gerçekleştirip çıkış FIFO’suna veri yazmaya başlayacaktır. Tüm veri okuma ve veri yazma işlemleri 32 bitlik paketler halinde gerçekleştirilecektir. Çıkış FIFO’suna son veri paketi yazarken *last* sinyalini de çıkartacaktır. Bu sinyal sayesinde sistemde DMA’in doğru çalışması sağlanmıştır. Blok şeması oluşturma işlemi tamamlandıktan sonra sistemin kontrolü için gerekli kodları yazmak için SDK ortamına geçilmiştir. Yazılan kodda öncelikle SHA3-512 modülü kullanarak özetlenmek istenen veri RAM’e yazılmıştır. Özetleme işlemi bittiğinde sonuç verisinin tekrar RAM’e yazılması amaçlanmaktadır. FIFO’lara veriler doğru şekilde geldiği sürece modül kendiliğinden çalışacağından dolayı yazılan kod ile işlemci üzerinden DMA kontrol edilmiştir. DMA üzerinden RAM’e veri yazarken veya RAM’den veri okurken 32 bitlik paketler 8 bitlik paketlere bölünerek aktarılmıştır. Bunun nedeni DMA ile veri aktarılırken 8 bitlik paketler halinde verilmesi gerekliliğindendir. Bu aşamadan sonra sistem FPGA üzerinde çalıştırılmış ve verilen girişlere karşılık NIST tarafından paylaşılmış örnek sonuçlarla kıyaslayarak doğru çıkışların alındığı terminal üzerinden gözlemlenmiştir. Elde edilen sonuç Şekil 4.19’da gösterilmiştir.



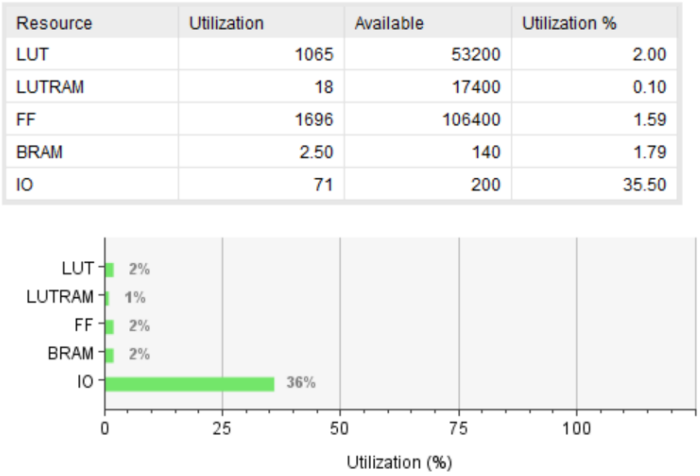
```
A69F73CC
A23A9AC5
C8B567DC
185A756E
97C98216
4FE25859
E0D1DCC1
475C80A6
15B2123A
F1F5F94C
11E3E940
2C3AC558
F500199D
95B6D3E3
01758586
281DCD26
```

Şekil 4.19 : SDK ortamında yapılan sistem testi sonucu.

4.4 HLS modülü ve VHDL modülünün kıyaslanması

SHA3-512 modülü Vivado HLS ortamında C yazılım dili ile yazılmış ve HLS ortamının özelliği sayesinde bu kodlardan VHDL kodları elde edilmiştir. Bu sayede Vivado ortamında modül bloğu olarak kullanılabilmiştir. HLS üzerinden üretilen bu kod doğrudan VHDL dili ile yazılan eşdeğer bir kod karşısında performans ve kaynak kullanımı anlamında Vivado ortamında kıyaslanmıştır. HLS üzerinden üretilen kod için AXI4-Stream arayüzüne ile tasarlanmış modül kullanılmıştır. Bu sefer SHA3-512 yerine SHA3-256 modülüne dönüştürülerek kullanılmıştır. Bu iki modülün farkı çıkış uzunluklarının farklı olmasıdır. Parametreler değiştirilerek bu iki modül arasında kolaylıkla geçiş yapılabilmektedir. Bu modüle karşılık VHDL dili ile yazılmış SHA3-256 modülü kullanılmıştır.

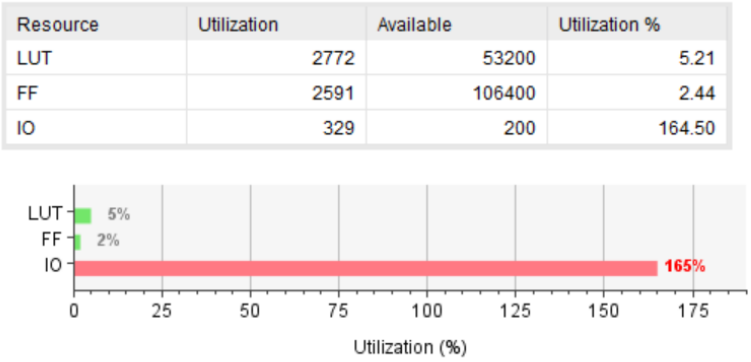
HLS üzerinden üretilmiş modülün Vivado ortamındaki kaynak kullanımı Şekil 4.20'de gösterilmiştir.



Resource	Utilization	Available	Utilization %
LUT	1065	53200	2.00
LUTRAM	18	17400	0.10
FF	1696	106400	1.59
BRAM	2.50	140	1.79
IO	71	200	35.50

Şekil 4.20 : HLS’te yazılan modülün kaynak kullanımı.

Doğrudan VHDL diliyle yazılmış modülün Vivado ortamındaki kaynak kullanımı Şekil 4.21’de gösterilmiştir.



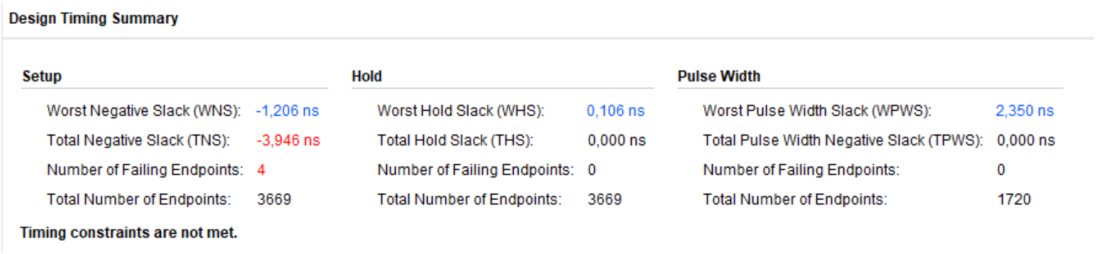
Resource	Utilization	Available	Utilization %
LUT	2772	53200	5.21
FF	2591	106400	2.44
IO	329	200	164.50

Şekil 4.21 : VHDL yazılan modülün kaynak kullanımı.

İki modül kaynak kullanımı açısından kıyaslandığında HLS’te yazılan modülün LUT ve Flip-Flop kullanımının VHDL’de yazılan modüle göre daha az olduğunu ancak buna karşılık VHDL ile yazılmış modülde kullanılmayan LUTRAM ve BRAM’lerin HLS ile yazılmış modülde kullanıldığı gözlemlenmiştir. LUT, mantık hücresinde yer alıp kombinasyonal fonksiyonları uygulayabilirken LUTRAM, hafıza hücresinde yer alır ve bunun yanında kaydıran yazmaç uygulamaları için yapılandırılabilir. Her ikisi de aynı şekilde kombinasyonal fonksiyonlar uygulayabilir. HLS ile yazılmış kodda BRAM kullanılarak bazı LUT’ların kullanımı VHDL ile yazılmış koda göre azaltılmıştır.

HLS üzerinden üretilmiş modül ve VHDL ile tasarlanmış modül için test dosyaları yazılıp sonuçlar incelendiğinde doğrudan VHDL ile tasarlanan modülün sonucu çok daha az saat döngüsünde çıkardığı gözlemlenmiştir. HLS üzerinden üretilmiş modülün girişine veri gönderildikten sonra işlemler yaklaşık 268 saat döngüsü sürmektedir. VHDL ile tasarlanan modülde ise bu işlemler 2 saat döngüsünde tamamlanmaktadır.

HLS ortamında tasarlanan SHA-3 modülü üzerinde saat periyodunu azaltacak stratejiler izlenerek sentezleme yapılmıştır. Bu işlem sonucunda saat periyodu olabilecek en düşük değere çekilmeye çalışılmıştır. Limit olarak 8 ns ve 7 ns saniye denenmiş sonuç olarak 7.2 ns limit koyulmasına karar verilmiştir. 7.2 ns saat döngüsü için sonuçlar Şekil 4.22 ve Şekil 4.23'te gösterilmiştir.

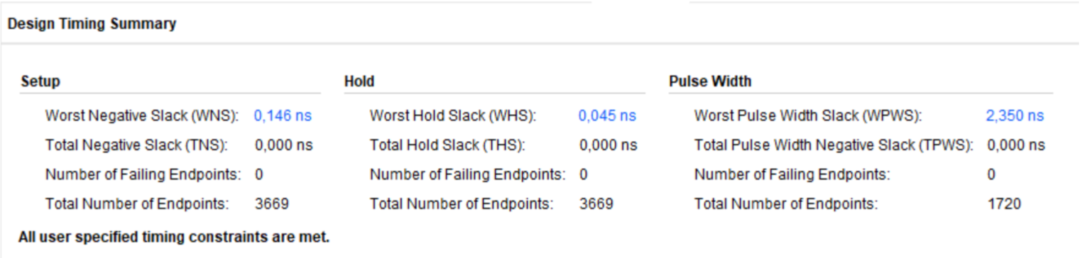


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1,206 ns	Worst Hold Slack (WHS):	0,106 ns	Worst Pulse Width Slack (WPWS):	2,350 ns
Total Negative Slack (TNS):	-3,946 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	4	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	3669	Total Number of Endpoints:	3669	Total Number of Endpoints:	1720

Timing constraints are not met.

Şekil 4.22 : HLS'te yazılan modülün sentez sonrası zamanlama özeti.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,146 ns	Worst Hold Slack (WHS):	0,045 ns	Worst Pulse Width Slack (WPWS):	2,350 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	3669	Total Number of Endpoints:	3669	Total Number of Endpoints:	1720

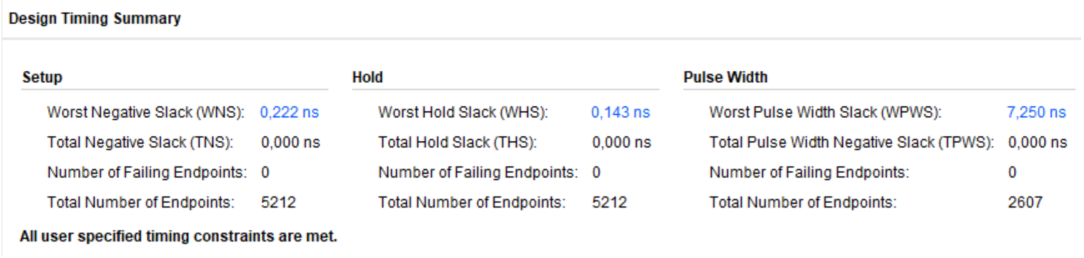
All user specified timing constraints are met.

Şekil 4.23 : HLS'te yazılan modülün implementasyon sonrası zamanlama özeti.

Sentez sonucunda zaman sınırlandırmasını karşılayamamıştır ve en düşük saat periyodunun 8.4 ns mertebesinde olması gerektiği gözlemlenmiştir ancak implementasyon sonucunda ise daha çok deneme yapıldığı için zaman sınırlandırmasını karşılayabilmiş hatta saat periyodunun 7.1 ns mertebesine kadar indirilebileceği gözlemlenmiştir.

VHDL dili ile tasarlanmış modül üzerinde aynı şekilde saat periyodunu azaltacak stratejiler kullanılarak sentezleme yapılmıştır. Limit olarak 15.5 ns uygulanmıştır.

Sentez sonucunda VHDL dili ile tasarlanan modül zaman sınırlandırmasını karşılamıştır. Sentez sonucu Şekil 4.24’te gösterilmiştir.



Design Timing Summary

Setup

Worst Negative Slack (WNS): 0,222 ns
Total Negative Slack (TNS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 5212

Hold

Worst Hold Slack (WHS): 0,143 ns
Total Hold Slack (THS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 5212

Pulse Width

Worst Pulse Width Slack (WPWS): 7,250 ns
Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 2607

All user specified timing constraints are met.

Şekil 4.24 : VHDL dili ile yazılan modülün sentez sonrası zamanlama özeti.

Sonuçlar incelendiğinde minimum saat periyodunun 15.3 ns mertebesine kadar indirilebileceği gözlemlenmiştir. Kıyaslama sonucunda elde edilen sonuçların bir tabloya aktarılmış hali Çizelge 4.2’de mevcuttur.

Çizelge 4.2 : Karşılaştırma sonuçları.

	Zaman döngüsü sayısı	**Minimum periyot**	**LUT**	**LUTRAM**	**Flip Flop**	**BRAM**
Vivado HLS	268	7.1 ns	%2	%1	%2	%2
VHDL	2	15.3 ns	%5	%0	%2	%0

İki modül arasındaki kıyaslamanın sonuçları incelendiğinde minimum saat periyodu bakımından HLS’te tasarlanan modülün daha üstün olduğu görülmüştür. Zaman döngüsü bakımından incelendiğinde ise VHDL ile tasarlanmış modül üstünlük göstermektedir. VHDL ile yazılan modül daha çok Flip-Flop ve LUT kullanmasına rağmen HLS ile tasarlanan modül ek olarak LUTRAM ve BRAM kullanmaktadır. Zaman döngüsü sayısındaki büyük fark nedeniyle minimum saat periyodu daha büyük olmasına rağmen VHDL ile tasarlanmış modül genele bakıldığında bir üstünlük göstermektedir.

5. AES ALGORİTMASI

5.1 Giriş

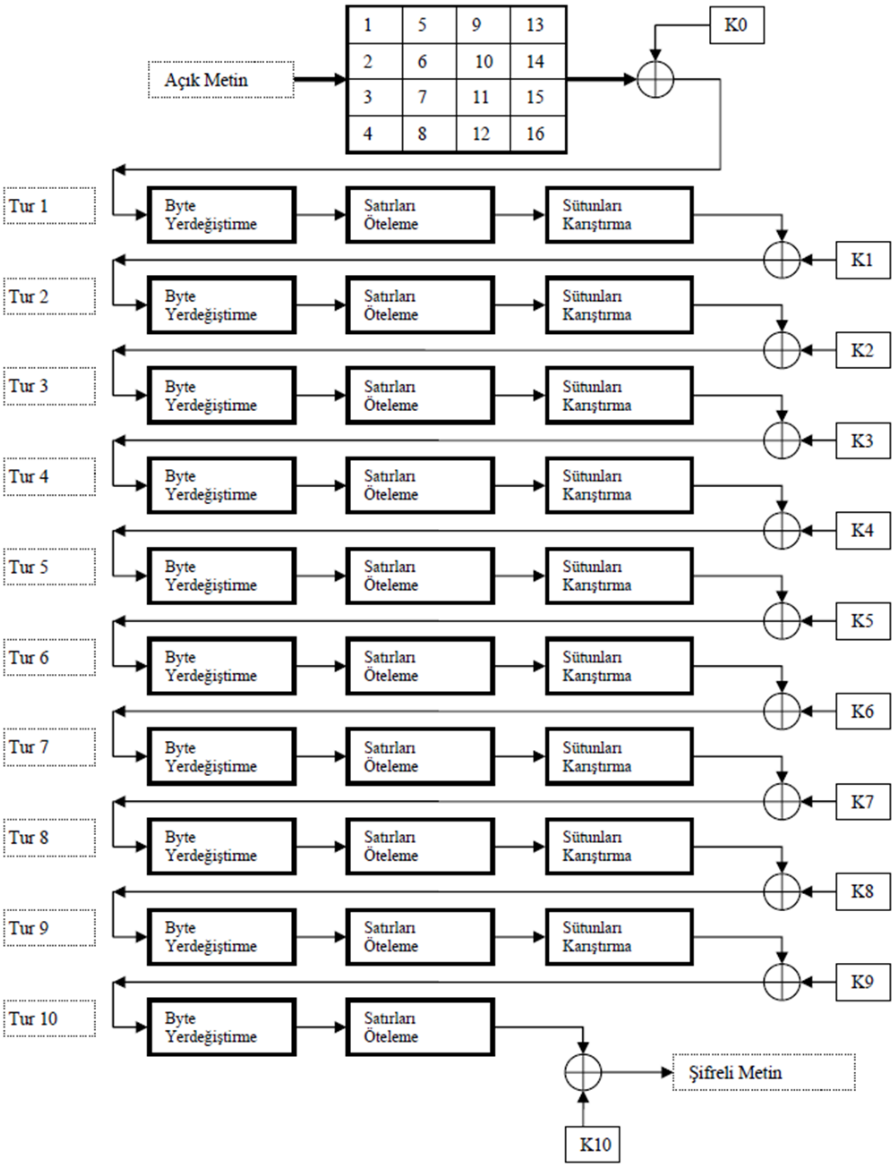
Kriptografide sistemler açık anahtar ve gizli anahtar kullanan olarak iki ana grup altında toplanmaktadır [5]. Asimetrik şifreleme algoritmaları olarak anılan ve şifrelemesinde açık anahtar kullanan sistemlerde şifrelenmiş metin açık anahtar ile şifrelenirken gizli anahtarlar ile çözülür, yani iki adet anahtar kullanılmaktadır [5]. Gizli anahtarlı sistemlerde ise süreç daha farklı ilerlemektedir. Simetrik algoritmalar olarak anılan bu sistemlerde tek bir gizli anahtar bulur ve bu anahtar hem şifrelemede hem de şifreyi çözmede kullanılır [5]. Gelişmiş Şifreleme Standardı (Advanced Encryption Standart - AES), tek bir gizli anahtar kullandığı için simetrik kriptografi algoritması olarak anılmaktadır [5].

NIST, IBM tarafından geliştirilen veri şifreleme standardı (Data Encryption Standard – DES) [5] algoritmasını 1977 yılında kendi standardı olarak kabul etmiştir. Uzun yıllar boyunca güvenli bir şekilde kullanılan bu algoritma, gelişen teknoloji sonucunda iyileşen bilgisayar donanımları yardımıyla DES algoritması kırılmış, bunun sonucunda DES algoritmasının 3 kere üst üste kullanılması ile oluşan üç kere DES (Triple DES – TDES/3DES) geliştirilmiştir [5]. DES ve TDES gelişen teknoloji ortamında zayıf kalması ve güvenilirliğinin sorgulanması sonucu bir yarışma çağrısında bulunmuş, yapılan bir çağrı sonucu Joan Daemon – Vincent Rijmen tarafından “Rijndael” algoritması geliştirilmiştir. 4 sene süren bu yarışmanın galibi olan bu algoritma, daha sonra AES olarak anılmıştır. Günümüzde AES; yazılım, aygıt yazılımı, donanım veya bunların çeşitli birleşimlerin oluşan birimlerde kullanmaktadır [3].

Çizelge 5.1 : Tur sayısı ile anahtar uzunluğu arasındaki ilişki.

AES türü	Anahtar uzunluğu	Veri uzunluğu	Tur sayısı
AES-128	128 bit	128 bit	10
AES-192	128 bit	192 bit	12
AES-256	128 bit	256 bit	14

AES algoritması, kullanacağı anahtarın bit sayısına göre adlandırılmaktadır. 128 bitlik veri blokları, 128, 192 ve 256 bit anahtarla şifrelenebilir. Anahtarın bir sayısı arttıkça tur sayıları da Çizelge 5.1’de de görüldüğü üzere etkilenmektedir. Basitçe açıklanacak olursa, daha fazla anahtar bitine sahip anahtarların kullanıldığı algoritmalar, daha az kullanılanlara göre daha güvenilirdir. Günümüzde sivil haberleşmede AES-128 kullanılmaktadır.



Şekil 5.1 : AES-128 algoritmasının şifrelemesi [5].

AES algoritması, her ne kadar anahtar bit sayısı değişken olsa da aynı yolu izleyerek şifrelemeyi gerçekleştirmektedir. Algoritma, gelen 128 bitlik veri bloğunu 4x4 baytlık matrise dönüştürür. Bu matrisin her hücresi 16 bitlik verilerden oluşur. Daha sonra bu matris alt fonksiyonlarında çeşitli yönlendirmeler sonucu son çıktı olan şifrelenmiş metni oluşturur. Bu alt fonksiyonlar,

- Byte Değişimi
- Satırların Kaydırılması
- Sütunların Karıştırılması
- Tur Anahtarı ile Ayrıcalıklı veya (Exclusive or- XOR) işlemi şeklinde sıralanabilir.

Değişken olan tur sayısı, algoritmada kullanılan alt fonksiyonların kullanım sayısını etkilemektedir. Sivil haberleşmede kullanılan AES-128'in şifreleme algoritması Şekil 5.1'de verilmiştir. Örnek bir açık metin için uygulanan bu şekilde K olarak adlandırılan eklentiler o turda eklenen tur anahtarları göstermektedir.

Bir şifreli metnin şifresinin çözülmesi işleminde ise şifrelendiği algoritmanın tur sayısı kullanılmaktadır, yani eğer AES-128 ile şifrelenmişse bu metnin deşifrelenmesinde yine 10 turluk alt rutin kullanılacaktır. Deşifrelemede kullanılacak alt fonksiyonlarda her ne kadar tur sayısı değişmeyecek olsa da türleri ve sırası değişecektir [3]. Bu fonksiyonlar ise,

- Ters Satır Kaydırma
- Ters Byte Değişimi
- Tur Anahtarı ile Ayrıcalıklı veya XOR işlemi
- Ters Sütün Karıştırma olarak sıralanabilir.

Deşifreleme işleminde kullanılacak anahtar eklemedeki anahtarlar, şifreleme esnasında oluşturulan ve kullanılan anahtarların sonuncusundan başlayarak eklenir. Görüldüğü gibi şifre çözme sürecindeki adımlar, şifreleme sürecindeki adımlarla birçok benzerlik göstermektedir. Her işlem şifreleme sürecindeki işlemlerin tersi olarak oluşturulmuştur.

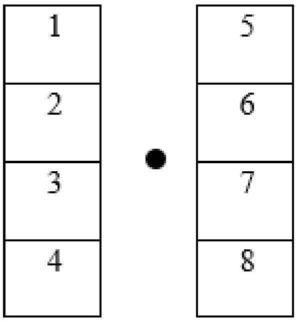
5.2 Matematiksel Altyapı

Her şifreleme algoritması gelen bit dizileri üzerinde çeşitli manipülasyonlar uygulayarak şifrelenmiş metinleri oluşturmaktadır. AES algoritmasında kullanılan bit manipülasyonları genel olarak XOR ve kaydırma işlemleri olarak söylenebilir. Kaydırma işlemlerinde, satırlar kaydırma işlemleri ile sağa veya sola ötelenirken XOR fonksiyonlarında ise bu kapı kullanılır. Matematiksel uygulamaları daha iyi anlamak adına fonksiyonlarda kullanılan bit çarpımlarını iyi kavramak gerekmektedir.

5.2.1 Nokta çarpım

AES algoritmasında, ilk önce şifrelenecek olan bilgi, 128 bitlik veri bloğu 4x4 boyutundaki matrise aktarılır. Matriste olan bu veri, sütun karıştırma işleminde sütun olarak işlem göreceği için ilk olarak nokta çarpımın nasıl olduğunu ve bit çarpımlarının nasıl yapıldığını bilmek gerekir.

Nokta çarpım, iki vektörün çarpımı sonucunu yönsüz bir değere eşitleme işlemidir. Bir vektörü oluşturan sayılar, karşılık gelen diğer vektörün sayılarıyla çarpılır ve bu çarpımlar toplanır. Şekil 5.2'de işlem gösterilmiş olup denklem 5.1'de işlem gerçekleştirilmiştir.



Şekil 5.2 : Nokta çarpım işlemi.

$$(1 * 5) + (2 * 6) + (3 * 7) + (4 * 8) = 70 \tag{5.1}$$

5.2.2 Polinom çarpımı

Bit dizilerinde işlemler yapılırken anlaşılırlığını arttırmak için polinom şeklinde işlemleri yapılır. Bir bit dizisi, bit değerlerinin derecesi baz alınarak bir değişken katsayısı atanır, a. bit için x^a değeri kullanılır. Bu değişken katsayısının önündeki çarpan ise bit değeri tarafından belirlenir, 1 ise 1 çarpanı 0 ise 0 çarpanı yerleştirilir. Şekil 5.3'te $(D2)_h$ sayısının polinom gösterimi verilmiştir.

$$(D2)_h = 1101\ 0010$$

$$1 \cdot x^7 + 1 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0$$

Şekil 5.3 : Bit dizisinin polinom gösterimi

Bu düzeyde çarpım işlemi ise polinom şeklinde verilen sayıların karşılıklı çarpılmasıyla elde edilir. Çarpım sonucunda sayıların katsayılarının mod2'si alınır, yani 2'ye bölümünden kalan değeri yazılır. Örnek çarpım işlemi Şekil 5.4'te verilmiştir.

$$\begin{array}{r} (x^6 + x^4 + x^2 + x + 1) * (x^7 + x^5 + x + 1) \\ \hline \rightarrow x^7 + x^5 + x + 1 \; + \\ x^8 + x^6 + x^2 + x \; + \\ x^9 + x^7 + x^3 + x^2 \; + \\ x^{11} + x^9 + x^5 + x^4 \; + \\ x^{13} + x^{11} + x^7 + x^6 \; + \\ \hline x^{13} + 2x^{11} + 2x^9 + x^8 + 3x^7 + 2x^6 + 2x^5 + x^4 + x^3 + 2x^2 + 2x + 1 \\ \hline\hline \end{array}$$

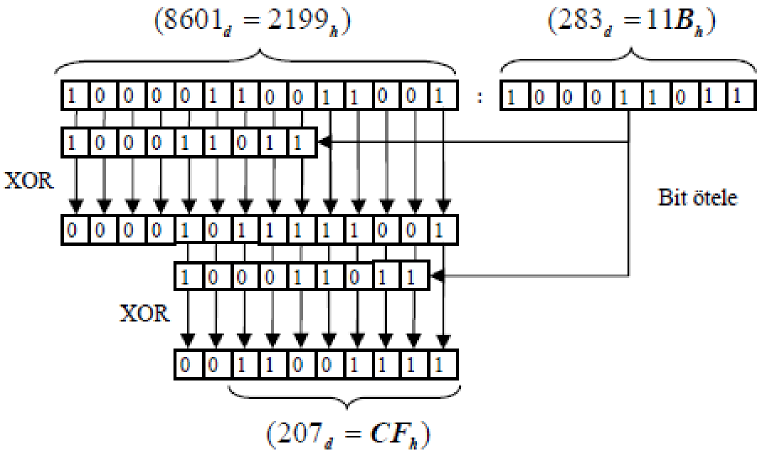
Şekil 5.4 : Polinom çarpımı [5].

Görüldüğü üzere verilen çarpma işleminde katsayılar 2 tabanında değerlendirilmemiştir. Bu katsayılarda gereken işlemler yapıldığında elde edilen sonuç Şekil 5.5'te verilmiştir.

$$x^{13} + x^8 + x^7 + x^4 + x^3 + 1$$

Şekil 5.5: Polinom çarpım sonucu [5].

Derecesi daha yüksek olan polinom değerlerinin derecesi indirgenmesi gerekir. Bu işlem ise elde edilen sonucun en anlamlı bitleri kesişecek bir biçimde başlayarak $m(x)=(x^8+ x^4+ x^3+x+1)$ XOR'lanmaya başlanır. Elde edilen sonuç m(x) polinom değer elde edilene kadar devam edilir. Bu işlem aslında m(x) değerine göre mod alma işlemi sürecidir. Örnek işlemler, Şekil 5.6'da verilmiştir [5].



Şekil 5.6 : Bir polinomun indirgenmesi [5].

5.2.3 AES şifreleme alt fonksiyonları

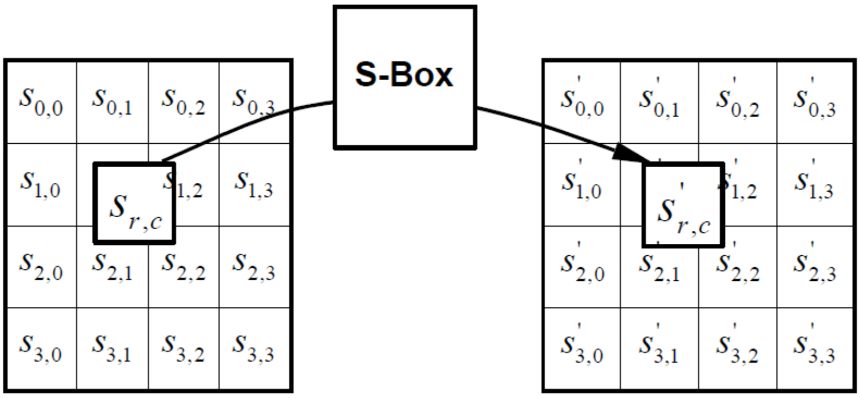
AES şifreleme işlemleri, 128 bit veri bloğunun girişte 4x4'luk durum matrislerine dağıtımından sonra 4 alt fonksiyonları aynı durum matrisi üzerinde kullanarak şifrelenmiş verinin elde edilmesini sağlar. Bu süreç gerçekleşirken yine alt fonksiyonlarda kullanılmak üzere anahtar değerleri üretilir. Bu anahtar değerleri anahtar planlama fonksiyonunda gerçekleşirken AES'te gerçekleşen her tur için farklı bir değer almaktadır [3, 5].

AES şifreleme sürecinde kullanılan 4 alt fonksiyon şunlardır

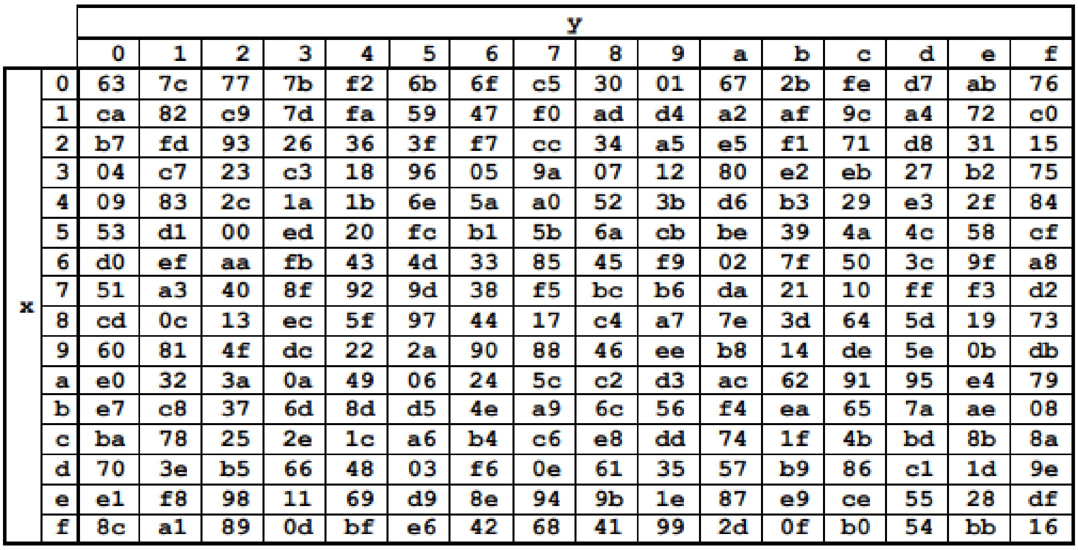
- Byte Değişimi
- Satırların Kaydırılması
- Sütunların Karıştırılması
- Tur Anahtarı ile XOR işlemi [3, 5]

5.2.3.1 Byte değişimi

128 bit veri bloğunun girişte 4x4'luk durum matrislerine dağıtımından sonra oluşan durum matrisinde byte değişim işlemi, mevcut durum matrisinin hücredeki hegza sayı değerinin S kutu (Substitution Box) diye belirtilen tablo değerleri ile değiştirilme işlemidir. 16 bitlik bir veri için ilk hex değeri yatay birleşeni ikincisi ise dikey birleşeni oluşturacaktır ve kesişimleri yeni değeri verecektir. Bu işlem ve S kutu tablosu Şekil 5.7 ve Şekil 5.8'de sırasıyla verilmiştir [3, 5].



Şekil 5.7 : Hücre değişimi işlemi [3].

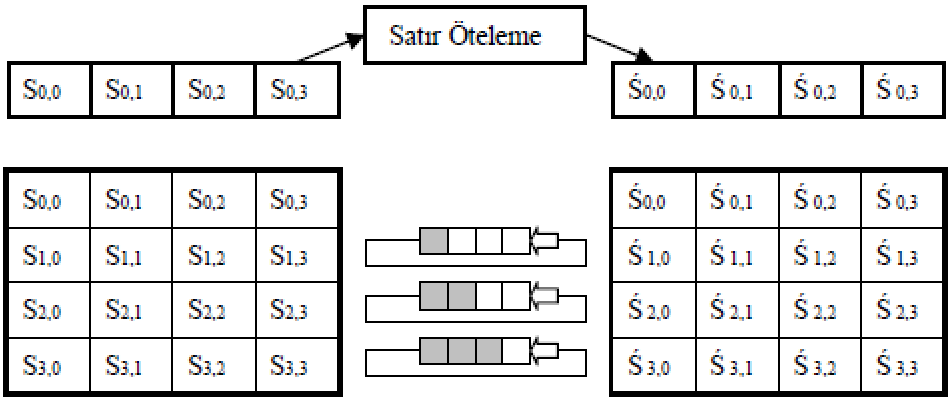


		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Şekil 5.8 : S-Kutusu tablosu [3].

5.2.3.2 Satırların kaydırılması

Bu alt fonksiyonda durum matrisi satırları sola doğru kaydırılır, en soldaki kaydırılması durumunda o hücre değeri en sağ kısma yazılır. İlk satıra dokunulmazken 2. satır 1, 3. satır 2 ve 4. satır ise 3 kere kaydırılır. İşlem Şekil 5.9'da gösterilmiştir.



S0,0	S0,1	S0,2	S0,3

Ś0,0	Ś 0,1	Ś 0,2	Ś 0,3

S0,0	S0,1	S0,2	S0,3
S1,0	S1,1	S1,2	S1,3
S2,0	S2,1	S2,2	S2,3
S3,0	S3,1	S3,2	S3,3

Ś0,0	Ś 0,1	Ś 0,2	Ś 0,3
Ś 1,0	Ś 1,1	Ś 1,2	Ś 1,3
Ś 2,0	Ś 2,1	Ś 2,2	Ś 2,3
Ś 3,0	Ś 3,1	Ś 3,2	Ś 3,3

Şekil 5.9 : Satır kaydırma işlemi [5].

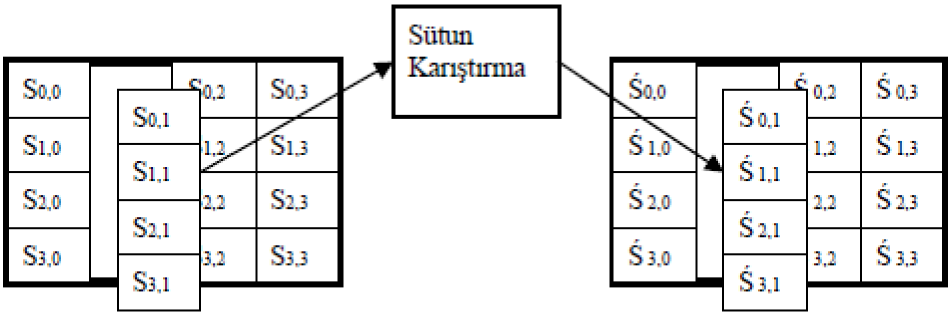
5.2.3.3 Sütunların karıştırılması

Yeni elde edilen durum matrisindeki hücre değerleri denklem 5.2'de verilen sabit bir matrisle çarpılarak elde edilir. Sabit matrisle durum matrisi, üstte açıklanan nokta çarpımı ile işleme sokulur ve her çarpım daha sonra kendi arasında XOR'lanır. Yeni elde edilen sütun için işlemler de denklem 5.2 ve denklem 5.3'te verilmiştir. Nb, oluşan matrisin sütun sayısını belirtmektedir.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad 0 \le c \le Nb\ için \qquad (5.2)$$

$$\begin{aligned} s'_{0,c} &= (\{02\}\bullet s_{0,c}) \oplus (\{03\}\bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\}\bullet s_{1,c}) \oplus (\{03\}\bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\}\bullet s_{2,c}) \oplus (\{03\}\bullet s_{3,c}) \\ s'_{3,c} &= (\{03\}\bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\}\bullet s_{3,c}) \end{aligned} \qquad (5.3)$$

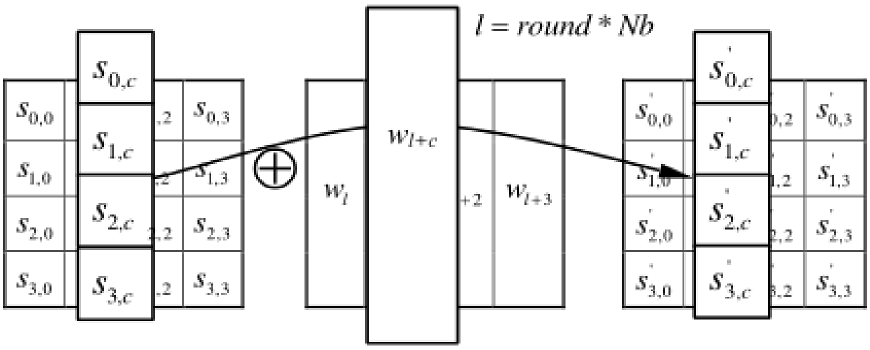
Bu işlemler Şekil 5.10'daki gibi her sütun için uygulanır ve böylece sütunlar karıştırılmış olur.



Şekil 5.10 : Sütunların değişimi [5].

5.2.3.4 Tur anahtarı ile XOR işlemi

Bütün şifreleme işleri durum matrisinde gerçekleşirken ilk girişteki anahtar değerinden farklı bir bit dizisi ile XOR işlemine sokulur. Bu anahtar değerleri “Anahtar Planlama” adı altında bir dizi işlem sonucunda oluşur. Anahtar planlama işlemlerinden sonra elde edilen dizi elde edilen son durum matrisi ile XOR işlemi gerçekleştikten sonra işlem biter.



Şekil 5.11 : Tur anahtarı ekleme [3].

Sütunların işlemleri yukarıda verilen Şekil 5.11’deki gibi oluşurken bu işlemlerin vektör şeklindeki gösterimi denklem 5.4’teki gibi yapılır.

$$\begin{gathered}[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [\mathrm{w}_{\boldsymbol{tur*Nb+c}}] \\ 0 \le c \le Nb \text{ } için\end{gathered} \tag{5.4}$$

5.2.4 Anahtar planlama

Anahtar planlama işlemi, girişte alınan anahtar bit dizisinin manipüle edilmesidir. Her tur için farklı bit dizileri oluşturularak bu dizilerin durum matrisleri ile tur sonlarında XOR işlemi uygulanarak güvenilirliğin sağlanması amaçlanmıştır. Bu anahtar değerleri daha sonra şifrelerin çözülmesinde de kullanılacaktır. Bu işlemler için kullanılan fonksiyonlar,

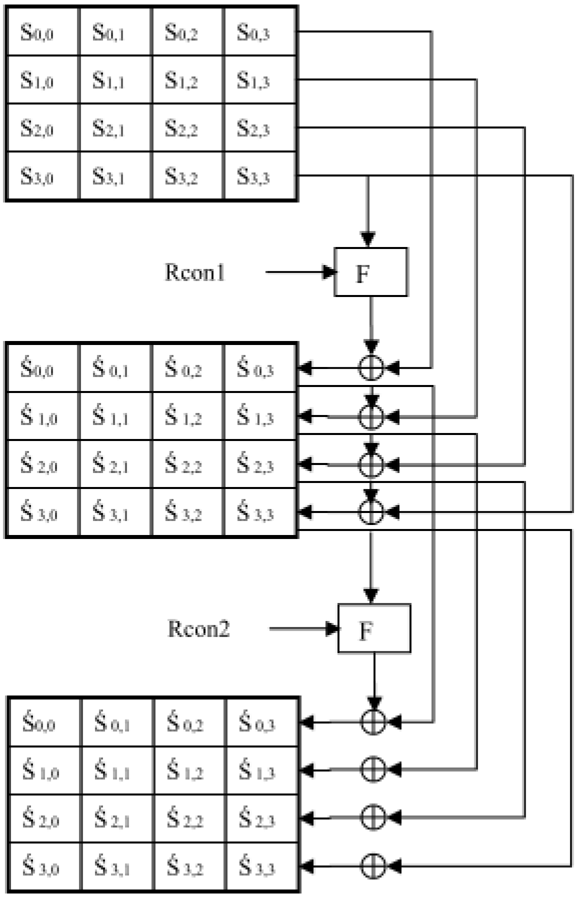
- Rot_word (Sütunun döndürülmesi)
- S-Kutusu ile byte değişimi
- Rcon (Sütunun Rcon matrisi ile çarpılması)
- Sıralı XOR işlemleri olarak sıralanabilir.

İlk anahtar değeri tur 0da direkt olarak veri matrisi ile XOR işlemine sokulur, fakat daha sonraki turlar için anahtar üretilmesi gerekir. Anahtar üretimi ilk olarak mevcut anahtar matrisinin son sütunun ilk hücresi olan elemanın en sona kaydırılarak her hücrenin bir üst hücreye kaydırılmasını sağlamaktır. Böylece elde edilen yeni sütun daha sonra Şekil 5.8’de verilen S-kutusu ile her hücre değeri tablo değeri ile değiştirilir. Bunu takip eden Rcon işleminde ise yeni sütun önceki turda kullanılan anahtar değerinin ilk sütunu ve Çizelge 5.2’de gösterilen Rcon tablosunun ilk sütunu ile XOR işlemine sokulur. Böylece yeni anahtar değerimizin ilk sütunu elde edilmiş olur [3, 5].

Çizelge 5.2 : Rcon tablosu.

01	02	04	08	10	20	40	80	1B	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Yeni anahtar değerinin diğer sütunları (2., 3. ve 4.) için ise sıralı bir şekilde XOR işlemi uygulanır. Bir önceki anahtar değeri matrisinin 2. sütunu ile elde edilen yeni matrisin ilk sütunu XOR işlemine sokulur ve böylece yeni anahtar değerinin 2. sütunu üretilir. Aynı işlem eski matrisin 3. sütunu ve yeni matrisin 2. Sütunu XOR işlemine sokulur ve bir önceki turun 4. sütunu alınır ve yeni anahtar matrisinin 3. sütunu XOR işlemine sokulup yeni matrisin son sütunu elde edilir. Burada dikkat edilmesi gereken husus, Rcon tablosunun, byte değişiminin ve hücre kaydırma işlemlerinin sadece yeni anahtar değerinin ilk sütunun elde edilmesinde kullanıldığıdır.



Şekil 5.12 : Genel hatlarıyla tur anahtarı üretimi [5].

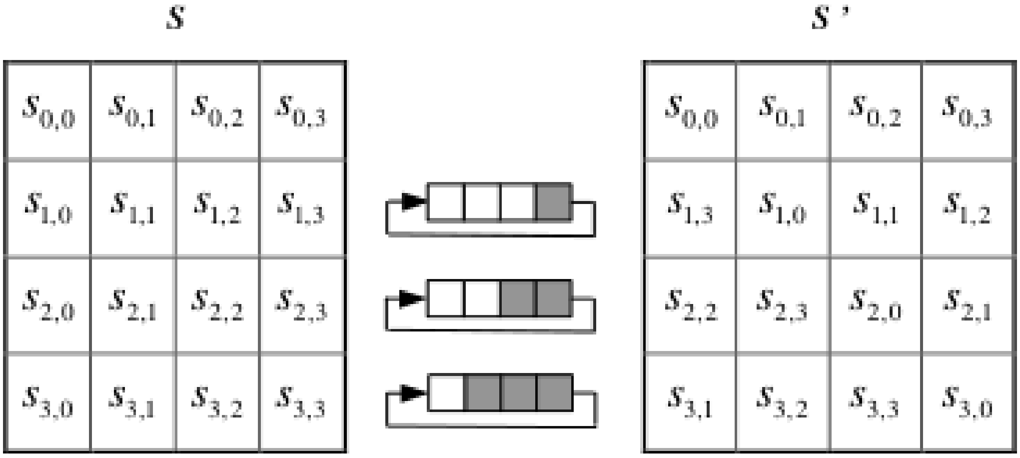
5.2.5 AES deşifreleme alt fonksiyonları

Şifre çözme işlemi, şifreleme işleminde yapılan adımların ters olarak gerçekleştirilmesi üzerine kurulmuştur. Anahtar planlama kısımları eş olup en son üretilen anahtar değerinden başlayarak mevcut durum matrisi ile XOR işlemi gerçekleştirilerek şifre çözme işlemi gerçekleştirilir. Deşifreleme işleminin gerçekleşmesi için 4 alt fonksiyon kullanılır. Bunlar,

- Ters Satır Kaydırma
- Ters Byte Değişimi
- Tur Anahtarı XOR işlemi
- Ters Sütün Karıştırma olarak sıralanabilir [3].

5.2.5.1 Ters satır kaydırma

Deşifreleme durum matrisinin anahtar matrisi ile tur0'da XOR işlemine sokulmasından sonrasında başlar. Daha sonra şifrelemenin aksine satır kaydırma ile devam eder. Ters satır kaydırma, şifrelemedeki satır kaydırma işlemi ile aynı işlemi yapar, tek farkı yönüdür.

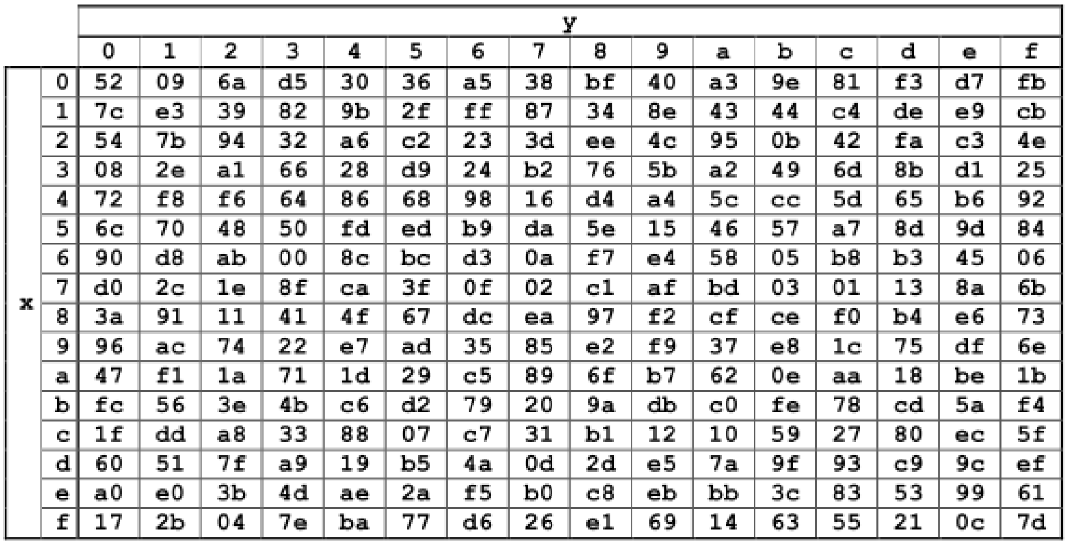


Şekil 5.13 : Ters satır kaydırma işlemi [3].

İlk satıra işlem uygulanmazken sonrasındaki satırlar sırayla 1, 2 ve 3 kere sağa kaydırılır. En soldaki hücre sağ kısımdan satıra dahil edilir. İşlemlerin gösterimi Şekil 5.13'te mevcuttur.

5.2.5.2 Ters byte değişimi

Ters Byte Değişimi, esasen şifreleme algoritmasındaki byte değişimi ile aynı işlemi yapmaktadır. Mevcut hücre değeri Ters S- kutusu tablosundaki değer ile değiştirilmektedir. Hücredeki 16 bitlik verinin ilk hegza değeri satırı, ikincisi ise sütunu işaret etmektedir. Ters S-kutusundaki satır ve sütunun kesiştiği tablo değeri ile mevcut hücredeki veri değiştirilir. Mevcut durum hücresindeki her kutu, Ters S-kutusu tablosu yardımı ile yeni değeri ile değiştirilir. Ters S-kutusu Şekil 5.14'te verilmiştir.



		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Şekil 5.14 : Ters S- kutusu tablosu [3].

5.2.5.3 Ters sütun karıştırma

Ters sütun karıştırma işlemi, AES şifreleme algoritmasında yer alan Sütun Karıştırma işleminin tam tersidir. Aynı işlemler uygulanmaktadır, ancak kullanılan matris değeri farklıdır. Uygulanan matris ve işlemler denklem 5.5 ve denklem 5.6'da verilmiştir.

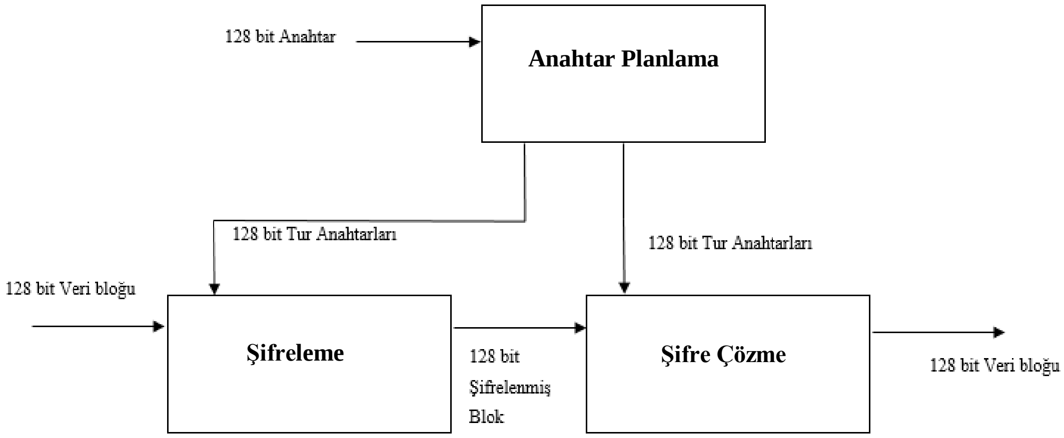
$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad 0 \le c \le Nb \text{ } için \tag{5.5}$$

$$\begin{aligned} s'_{0,c} &= (\{0e\}\bullet s_{0,c}) \oplus (\{0b\}\bullet s_{1,c}) \oplus (\{0d\}\bullet s_{2,c}) \oplus (\{09\}\bullet s_{3,c}) \\ s'_{1,c} &= (\{09\}\bullet s_{0,c}) \oplus (\{0e\}\bullet s_{1,c}) \oplus (\{0b\}\bullet s_{2,c}) \oplus (\{0d\}\bullet s_{3,c}) \\ s'_{2,c} &= (\{0d\}\bullet s_{0,c}) \oplus (\{09\}\bullet s_{1,c}) \oplus (\{0e\}\bullet s_{2,c}) \oplus (\{0b\}\bullet s_{3,c}) \\ s'_{3,c} &= (\{0b\}\bullet s_{0,c}) \oplus (\{0d\}\bullet s_{1,c}) \oplus (\{09\}\bullet s_{2,c}) \oplus (\{0e\}\bullet s_{3,c}) \end{aligned} \tag{5.6}$$

5.3 Tasarım ve Gerçekleme

AES şifreleme/deşifreleme algoritması tam kavrandıktan sonra tasarım aşamasına geçilebilir. Tasarımın başlangıcı bir Vivado HLS projesi oluşturmakla başlanmıştır.

Bahsedildiği üzere Vivado HLS yüksek seviyeli C dillerini (C, C++, SystemC) kullanarak donanım kodlaması yapmaya ve bunun yanında bu kodların değişik durumlar için ve isteklere göre optimizasyon yapılmasına olanak vermektedir. Bunun yanı sıra çeşitli haberleşme protokoller için de kolaylık sağlayan Vivado HLS, oluşturulan modüllerin arayüzlerin hazırlanması için de kolaylık sağlamaktadır.
Her ne kadar C dilleri kullanılsa da bazı özel noktalarda Vivado HLS klasik yazım tarzını kabul etmemektedir. Ne kadar yüksek seviyeli diller kullanılsa da optimizasyon veya dizi yazımlarında Vivado HLS devreye girerek yazılan dilin söz dizimi kurallarını kabul etmemektedir. Bunun yanı sıra bazı optimizasyon yönlendirmeleri de her yapı için kullanılmamaktadır. Kullanılan dilin yapısı ve izin verdiği kütüphane seçenekleri yüzünden de sınırlandırılan optimizasyon yöntemleri Vivado HLS programının sunduğu özgürlüğü bazen kısıtlamaktadır. Tasarımın başlangıcında sonuçlanması gereken çıktı modüllerin çizimi Şekil 5.15'de gösterilmiştir.



Şekil 5.15 : Tasarlanan modüller.

Şifreleme, deşifreleme ve anahtar planlama ile ilgili işlemler 3 farklı modülde gerçeklenmiştir. İlgili modüllerin ayrı ayrı oluşturulmasının amacı en az düzeyde hafıza kaplamadır. Şifreleme ve Deşifreleme algoritmalarında aynı anahtarlar kullanılmaktadır. Sonuç olarak aynı işlemi gerçekleştiren algoritmanın iki kere kullanılarak sınırlı bir hafızayı harcamasının önlenmesi için anahtar üretimi ayrı bir modül olarak tasarlanmıştır.

Tasarlanacak olan modüllerde kullanılacak olan fonksiyonların yazımından önce her fonksiyonda kullanılması ve kirlilik yaratmamak adına bir tanım dosyası (header file) hazırlanmıştır. Her fonksiyonda kullanılacak alt modüller ve tanımlar her defasında belirtilip karmaşıklığa neden olmaması için düzenlenen bu header dosyası algoritmanın daha anlaşılır oluşmasını sağlamıştır. Header dosyası içerisinde her fonksiyonun tanımını, boyutlarını ve bit tanımlarını içermektedir.

Header dosyasının yanı sıra her fonksiyon yine ayrı ayrı dosya şeklinde tanımlanmıştır. Bunun anlamı her dosya gerçekleştirdiği işlemin düzenli ve sistematik oluşturulduğudur. Kontrol amaçlı yazılacak olan bir test dosyası ile kolay çağrılma ve istenildiği zaman her fonksiyonun ileriki tasarımlarda gereken donanım tanımlama kodunun (VHDL/Verilog) kolay oluşturulmasını sağlamaktır. İlgili işlem yapılması istenen fonksiyon kolayca üst fonksiyon (top function) seçilerek test edilebilir ya da donanım tanımlama kodu çıkarılabilir.

Alt fonksiyon dosyalarının haricinde şifrelemede/deşifrelemede kullanılacak olan tablolar da ayrı dosyalar şeklinde oluşturulmuştur. Farklı boyutlarda işlem gerektirdiği için S-kutusu anahtar planlamada ve şifrelemede/deşifrelemede ayrı şekillerde oluşturulmuştur.

Modüllerin kendi arasında haberleşmesinde kullanılmak üzere arayüzler oluşturulmuştur. Vivado HLS programının özelliği olan “pragma” direktifi kullanılarak AXI4-Stream arayüzü oluşturulmuştur. Bu arayüzün seçilmesinin nedeni seri veri akışlarının sağlanmak istenmesidir. FIFO (First In First Out) modülleri, tasarlanan modüllerin girişine ve çıkışına bağlanarak bu verilerin modüle girişini/çıkışını sağlamaması için kullanılacaktır. Normal şartlarda Vivado HLS, AXI4-Stream arayüzü oluştururken FIFO eklentilerini koyma özelliğine sahiptir, fakat bu bazı kodlar için çalışmamaktadır.

Tasarlanan modüllerde arayüz tanımlaması haricinde pragma direktifi kullanılmamıştır. Mevcut sistemin FPGA üzerinde çalıştırılmasında boyut açısından sıkıntı doğmayacağı düşünülmüştür.

AES 128 algoritmasının şifreleme modülünde test dosyasının oluşturulması için anahtar planlama fonksiyonları içerilse de üst fonksiyonun sentezlendiği için bu dosyalar esasen kullanılmamıştır. Vivado HLS programının sentezlediği üst fonksiyonun içerdiği dosyaları kullandığı için sentez dosyasının özetinde yer alan kullanımlarında üst fonksiyon içerisinde yer almayan ve kullanılmayan fonksiyonların etkisi yoktur.

Şifreleme algoritması tasarlanırken ilk amaç hızın yanında olabildiğince az yer kaplamasının sağlanmasıdır, zira bu sadece AES 128 için tasarlanan 3 modülün yanı sıra SHA-3 ve RSA modülleri de aynı FPGA modülünün içine yerleşeceği için LUT ve Flip-flop kullanımı göz önüne alınmıştır.

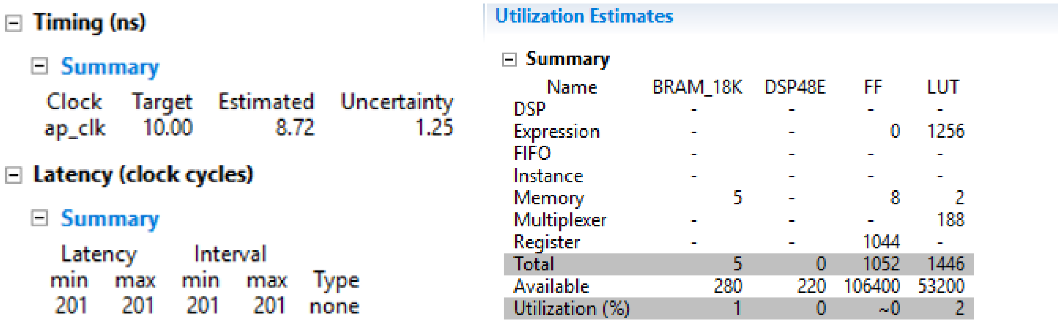
Sütunların karıştırılması işlemi ilk olarak LUT üzerinden gerçekleştirilmiştir. Bu gerçekleştirme sonucunda daha hızlı kullanım söz konusudur, fakat bunun için hazırlanan kodun fazladan tablolar barındıracağı gerçeği göz önüne alındığında daha optimize seçim olarak işlemlerin gerçekleştirilmesi düşünülmüştür. Bu sayede algoritmanın LUT kullanımı düşürülmüştür, fakat daha yavaş çalışmasına neden olmuştur. LUT değerinin düşük tutulması nedeni ise kodun mevcut durumda en fazla kullanımı olan bu değerin azaltılmasıdır.

Üst fonksiyon tanımlanırken giriş çıkışlar en başta belirtilen Şekil 5.15'deki gibi oluşturulmuştur. Anahtar girişi bir dizi olarak alınarak her tur için ayrı bir elemanı kullanılmaktadır. Üst fonksiyonda pragma direktifi kullanılarak oluşturulan arayüz oluşumunda istenilen sonuç alınmadığı için fazladan bir dosya oluşturularak bu durum ortadan kaldırılmıştır. İlk yazılan fonksiyonda çıkış olarak görünmesi gereken şifrelenmiş metin sentez özetinde giriş olarak belirtilmesinden dolayı yapılan bu değişiklik sonucunda istenilen giriş/çıkış tanımları sağlanmıştır.

AES 128 deşifreleme algoritmasında da şifrelemedeki gibi anahtar planlama dosyaları mevcut sistemin kontrolü için eklenmiştir Çok benzer yapılara sahip olan AES şifreleme ve deşifreleme algoritmalarının oluşumu kadar yazılma işlemi de benzer şekilde oluşturulmuştur. Fonksiyonların içerikleri farklı görünse de bu farkın nedeni tasarım şeklini değiştirmemiştir.

AES 128 algoritmalarında kullanılmak üzere tasarlanan Anahtar Planlama algoritması ise benzer yapıları içerdiği için boyutlar değiştirilerek benzer fonksiyonlar kullanılmıştır. Üst fonksiyon şifreleme ve şifre çözme işlemlerinde kullanılan fonksiyonları içermediği için ve bu diğer algoritmalara göre daha az işlem yaptığı için kaynak kullanımı daha az ve hızı daha yüksektir. Arayüz ayarlaması yapılırken giriş-çıkış kanallarında istenilen tanımlamalar sentez sonunda elde edildiği için ek olarak bir üst fonksiyon hazırlanma ihtiyacı hissedilmiştir.

Tasarlanan 3 modül için pragma direktifi yardımı ile hazırlanan arayüzler ileriki kullanımlar için oluşturulmuştur. Mevcut kaynak kullanımları ve işlem süreleri her modül için Şekil 5.16, Şekil 5.17 ve Şekil 5.18'de verilmiştir.



⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.72	1.25

⊟ Latency (clock cycles)

⊟ Summary

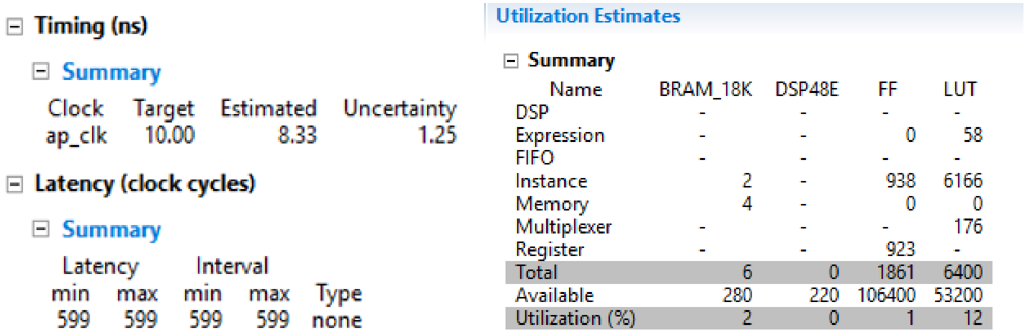
Latency		Interval		
min	max	min	max	Type
201	201	201	201	none

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1256
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	5	-	8	2
Multiplexer	-	-	-	188
Register	-	-	1044	-
Total	5	0	1052	1446
Available	280	220	106400	53200
Utilization (%)	1	0	~0	2

Şekil 5.16 : Anahtar planlama algoritmasının zamanlama değerleri ve kaynak kullanımı.



⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.33	1.25

⊟ Latency (clock cycles)

⊟ Summary

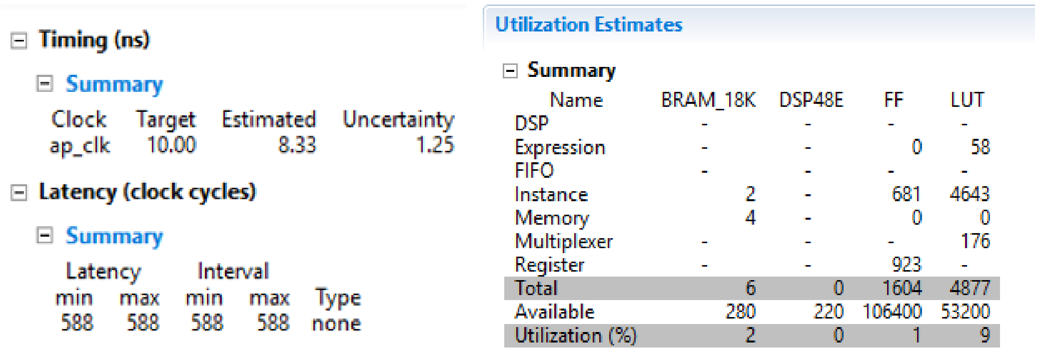
Latency		Interval		
min	max	min	max	Type
599	599	599	599	none

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	58
FIFO	-	-	-	-
Instance	2	-	938	6166
Memory	4	-	0	0
Multiplexer	-	-	-	176
Register	-	-	923	-
Total	6	0	1861	6400
Available	280	220	106400	53200
Utilization (%)	2	0	1	12

Şekil 5.17 : Şifre çözme algoritmasının zamanlama değerleri ve kaynak kullanımı.



⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.33	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
588	588	588	588	none

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	58
FIFO	-	-	-	-
Instance	2	-	681	4643
Memory	4	-	0	0
Multiplexer	-	-	-	176
Register	-	-	923	-
Total	6	0	1604	4877
Available	280	220	106400	53200
Utilization (%)	2	0	1	9

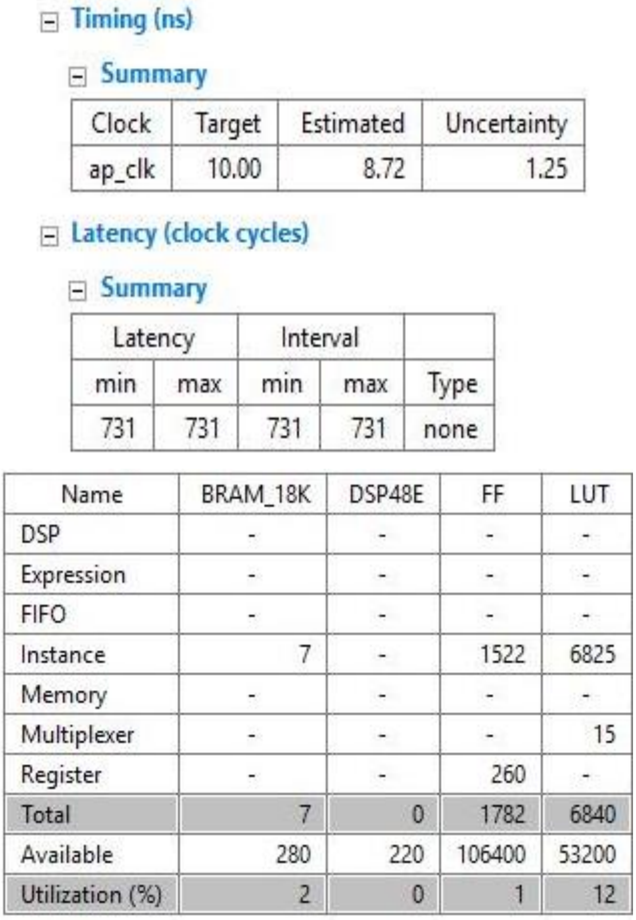
Şekil 5.18 : Şifreleme algoritmasının zamanlama değerleri ve kaynak kullanımı.

5.3.1 AXI4-Lite arayüzlü modül

Projede kullanılacak olan ZedBoard Zynq™-7000 Geliştirme kartında bulunan Zynq modülü, kontrol etmesi gereken modüllerde AXI4-Lite arayüzü şartı aramaktadır. Düşünülen Zynq tabanlı sistemde ise yazılmış modüllerin kontrolü Zynq üzerindeki işlemci tarafından kontrol edilmesi öngörüldüğü için ilk başta hazırlanan AXI4-Stream arayüzü AXI4-Lite arayüzüne çevrilmiştir. Bu çevirmenin yanı sıra, tasarlanan sistemdeki modüllerin tek giriş ve tek çıkış şartı istendiği için AES 128 şifreleme ve deşifreleme modüllerinin içerisine Anahtar Planlama modülü eklenerek kullanılacak modüllerin sayısı ikiye ve giriş kanalları ise teke düşürülmüştür. Bu uygulama sayesinde giriş teke düşürülmüştür fakat kullanılan toplam alan AES 128 kısmı için artmıştır. Giriş tek kanal ile 256 bit bilgi alırken 128 bit bilgi dışarı vermektedir. Giriş kanalında verilen 256 bitlik dizinin ilk 128 bitlik kısmı anahtar iken kalan sondaki 128 bitlik kısmı ise mesaj metnidir. Çıkış ise yine 128 bitlik kriptografi mesajıdır. Deşifrelemede ise aynı şekilde anahtar iletimi gerçekleşirken ikinci 128 bitlik kalan kısım kriptografi bitleri iken çıkış ise mesaj metnidir.

AXI4-Lite arayüzüne geçiş için ise basit bir pragma direktifi kullanıldı. Port tanımlarında *#pragma HLS INTERFACE s_axilite port* direktifleri kullanılırken *“#pragma HLS INTERFACE s_axilite port=return* direktifi ise oluşacak olan başlama ve benzeri pinlerin yok edilmesi için kullanılacaktır. Bunun nedeni ise AES şifreleme/deşifreleme modülünün girişlerine bağlanacak olan FIFO’ların dolmasına ve boşalmasına göre çalışması istenmesidir.

Arayüzlerin ve Anahtar Planlama modülünün şifreleme/deşifreleme modüllerine bağlanması sonucunda elde edilen yeni modüllerin kaynak kullanımları ve zamanlamaları Şekil 5.19 ve Şekil 5.20’de verilmiştir.



⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.72	1.25

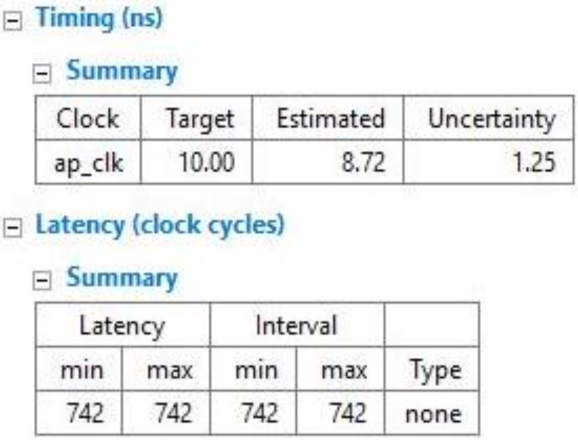
⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
731	731	731	731	none

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	7	-	1522	6825
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	260	-
Total	7	0	1782	6840
Available	280	220	106400	53200
Utilization (%)	2	0	1	12

Şekil 5.19 : Yeni AES-128 şifreleme modülünün zamanlaması ve kaynak kullanım oranları



⊟ Timing (ns)

⊟ Summary

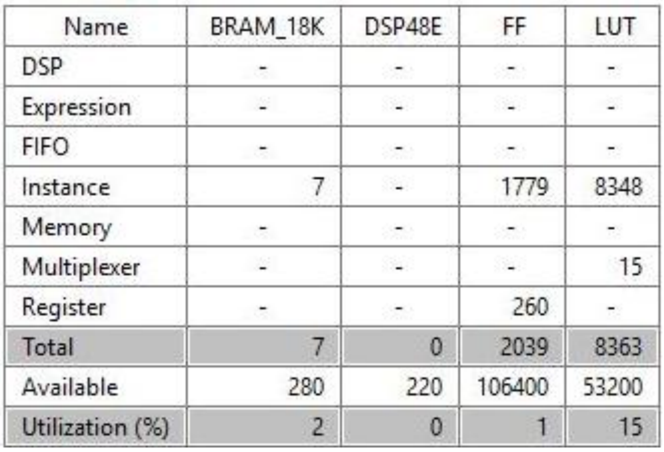
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.72	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
742	742	742	742	none

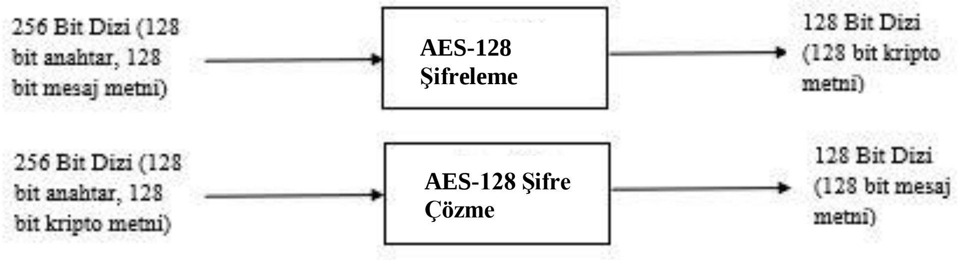
Şekil 5.20 : Yeni AES-128 deşifreleme modülünün zamanlaması ve kaynak kullanım oranları



Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	7	-	1779	8348
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	260	-
Total	7	0	2039	8363
Available	280	220	106400	53200
Utilization (%)	2	0	1	15

Şekil 5.20 (devam) : Yeni AES-128 deşifreleme modülünün zamanlaması ve kaynak kullanım oranları

Yeni oluşturulan modüllerin çalışma şekli ise aşağıdaki Şekil 5.21'deki blok şemada verilmiştir.

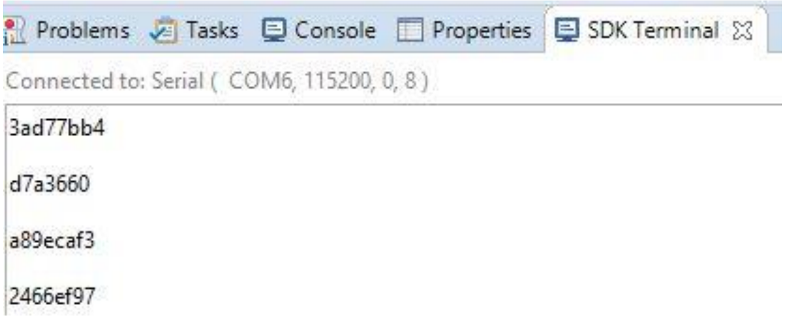


Şekil 5.21 : Yazılan yeni modüllerin blok şemaları.

Oluşturulan bu modüller daha sonra Vivado programına aktarılarak bu programın yardımı ile arayüzün test işlemi için gerekli blok tasarımları oluşturdu. Blok tasarımda ilk olarak Zynq işlemci eklendi ve frekans süresine bakıldı. Yaşanacak gecikmelerde sıkıntı yaşanmayacağı için bir varsayılan değerinde bırakıldı. Daha sonra *Bitsream* dosyası oluşturuldu ve tasarım tamamlandı. Oluşturulan tasarım SDK programı yardımı ile mikroişlemci kodları yazıldıktan sonra FPGA'e atılarak test edildi.

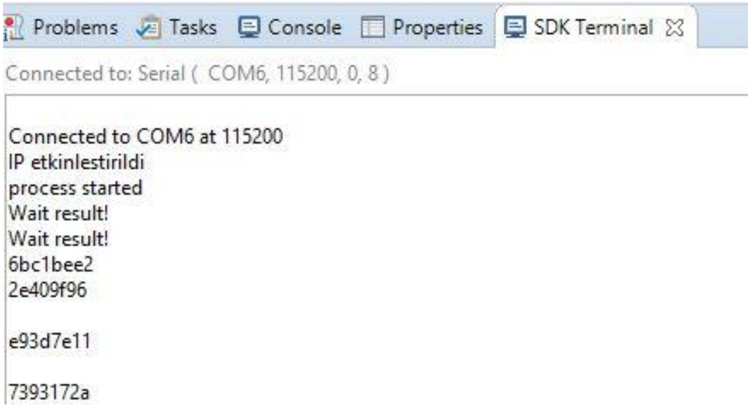
Test işlemi için açılan ekranda yeni bir proje oluşturulmasından sonra sürücü dosyasından gelen kütüphane dosyası açılarak fonksiyon tespit işlemi ile başladı. Oluşan sürücü dosyaları Vivado HLS tarafından otomatik olarak oluşturulmuştur. Kütüphane dosyasının içeriğinde yazmaç elemanlarının adresleri ve değişkenleri isimlerle atandığı, fonksiyon yapıları oluştuğu görülmüştür. Oluşan fonksiyonlardan

Set ve *Get* isimli olanlar *slave* durumundaki yazmaç yapısından oluşmuş olup bu kısmıma kodun yazılması ya da alınması komutlarını içermektedir. Bunun yanı sıra modülün içeriğindeki bit girişleri için farklı adres değişkenleri atandığı görüldü. Sistem kodu yazılırken farklı kütüphanelerden elde edilen adres kodları ve sürücü kütüphaneleri ile sistemin yazma ve okuma geçişleri ayarlanmış ve gerekli adres atamaları yapılmıştır. Şifrelenecek mesaj modülün girişine uyacak şekilde ayarlandı ve *device id* sayesinde kullanılacak modül seçimi yapılmıştır. Sistem daha sonra FPGA'e atılarak test edilmiş ve mesaj giriş olarak NIST'in örnek denemeleri yapılmıştır. Sistem aynı şekilde deşifreleme modülünde de işlediği için bütün işlemler aynı şekilde bu modülde de uygulanmıştır. İki modül için de elde edilen sonuçlar NIST'in kendi makalesinde yayınladığı örnekler üzerinden yapıldı ve doğru şekilde çalıştığı gözlemlendi. Elde edilen sonuçlar Şekil 5.22 ve Şekil 5.23'te mevcuttur.



```
Connected to: Serial ( COM6, 115200, 0, 8 )

3ad77bb4

d7a3660

a89ecaf3

2466ef97
```

Şekil 5.22 : SDK ortamında AXI4-Lite arayüzlü şifreleme sisteminin test sonucu.

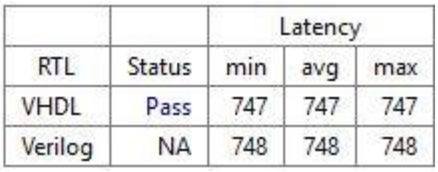


```
Connected to: Serial ( COM6, 115200, 0, 8 )

Connected to COM6 at 115200
IP etkinlestirildi
process started
Wait result!
Wait result!
6bc1bee2
2e409f96

e93d7e11

7393172a
```

Şekil 5.23 : SDK ortamında AXI4-Lite arayüzlü deşifreleme sisteminin test sonucu.

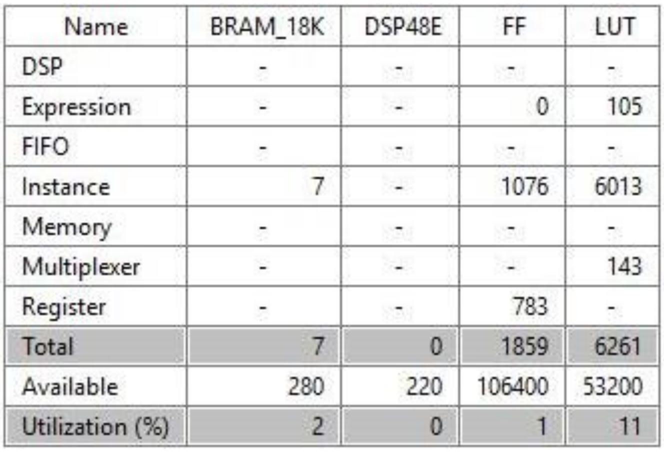
5.3.2 AXI4-Stream arayüzlü yeni modül

AXI4-Lite ile oluşturulan sistemde her işlemin mikroişlemci üzerinden yürümesinden dolayı düşünülen sistem DMA ile kullanılan bir Zynq tabanlı sistem olmuştur. Zynq modülünün kontrol ettiği DMA modülü sayesinde oluşturulan modüllerin haberleşmesi sağlanacaktır. Bu yüzden FIFO'lar arasında daha farklı bir arayüz yapılması düşünülmüş ve genellikle hızlı veri paketlerin aktarılmasında kullanılan AXI4-Stream arayüzü düşünülmüştür. Eski yazılan modülün iki girişli olmasından dolayı yeniden tasarlanan modülde farklı olarak AXI4-Lite arayüzü ile tasarlanmış modüldeki gibi başlama, durdurma vb. kontrol girişleri yok edilmiştir. Yeni oluşturulan bir üst modülde işleyiş bir *HLS::STREAM* için tanımlı *struct* yapısı sayesinde gerçekleşmiştir. Bu yapıda oluşan veri paketlerine ek *last* sinyal işareti de eklenmiş ve bu şekilde modülün girişine ve çıkışına bağlanacak olan FIFO'ların kontrolü sağlanması amaçlanmıştır. AXI4-Stream arayüzü için pragma direktiflerinin yanı sıra girişler de arayüze uygun olacak şekilde 32 bit olacak şekilde üst modülde tasarlandı. 256 bit olacak şekilde 8 paket veri alınacak olan üst modülde ilk 128 bit anahtar olarak tanıtılacakken sonraki 128 bitlik dizi ise mesaj olacaktır, yani ilk 4 paket veri anahtar bilgisini taşırken geriye kalan dosyalar mesaj içeriğini barındıracaktır. *Last* sinyali 1 olmadıkça gelen her dosya aynı anahtar bilgisi ile şifrelenecektir. Her şifreleme için 4 paket mesaj bilgisi, yani 128 bit değerinde veri gereklidir. Bit paketlerinin birbiri ile birleşimi alındıktan sonra kaydırma işlemi sayesinde dizilmesi ile oluşur. Çıkış ise 4 adet 32 bitten oluşan mesaj paketleridir. Üst modülde yazılan yöntem aynı şekilde deşifreleme için de uygulanmıştır. Tasarımlar tamamlandıktan sonra gerekli port tanımları pragma direktifleri kullanıldı ve tamamlandı. Tasarımların kaynak kullanımları ve zamanlamaları Şekil 5.24, Şekil 5.25, Şekil 5.26 ve Şekil 5.27'de verilmiştir.



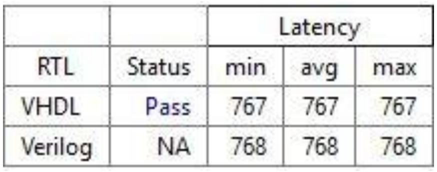
		Latency		
RTL	Status	min	avg	max
VHDL	Pass	747	747	747
Verilog	NA	748	748	748

Şekil 5.24 : Yeni AXI4-Stream arayüzlü AES-128 şifreleme modülünün zamanlaması (Verilog ve VHDL).



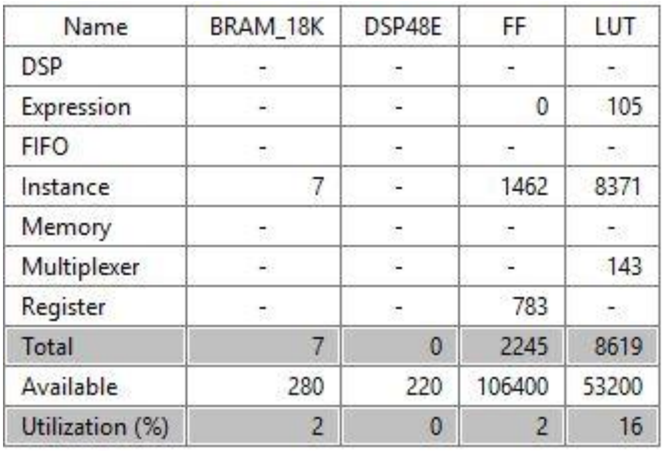
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	105
FIFO	-	-	-	-
Instance	7	-	1076	6013
Memory	-	-	-	-
Multiplexer	-	-	-	143
Register	-	-	783	-
Total	7	0	1859	6261
Available	280	220	106400	53200
Utilization (%)	2	0	1	11

Şekil 5.25 : Yeni AXI4-Stream arayüzlü AES-128 şifreleme modülünün kaynak kullanım oranı.



		Latency		
RTL	Status	min	avg	max
VHDL	Pass	767	767	767
Verilog	NA	768	768	768

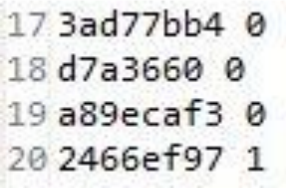
Şekil 5.26 : Yeni AXI4-Stream arayüzlü AES-128 deşifreleme modülünün zamanlaması (Verilog ve VHDL).



Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	105
FIFO	-	-	-	-
Instance	7	-	1462	8371
Memory	-	-	-	-
Multiplexer	-	-	-	143
Register	-	-	783	-
Total	7	0	2245	8619
Available	280	220	106400	53200
Utilization (%)	2	0	2	16

Şekil 5.27 : Yeni AXI4-Stream arayüzlü AES-128 deşifreleme modülünün kaynak kullanım oranı.

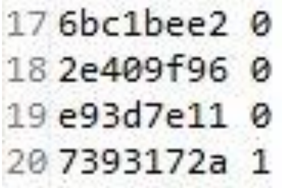
Kodun Vivado HLS üzerinde tanımlanmasından sonra yine aynı ortamda bir test dosyası oluşturulmuştur. Bu test dosyası oluşturulurken okunacak anahtar ve mesaj değeri bir dosyadan, *last* sinyalleri ise diğer dosyadan okunmuştur. *Stream* için yazılan *struct* yapısı sayesinde oluşturulan değişkene mesaj paketlerine ek bitişlere *last* sinyali ile verilmiş ve çalışma durumunda sinyalin doğru çıkıp çıkmadığı test edilmiştir. Modülün girişinde ve çıkışında istenilen sonuçlar görülmüştür. Test sonucu Şekil 5.28'de verilmiştir. Sonuçta görüldüğü üzere şifrelenmiş metin ile bitişi simgeleyen *last* sinyal çıkışı görülmektedir.



```
3ad77bb4 0
d7a3660 0
a89ecaf3 0
2466ef97 1
```

Şekil 5.28 : Vivado HLS ile tasarlanan şifreleme algoritmasının test dosyası sonucu.

Aynı işlemlerin deşifreleme algoritması için uygulanmıştır. Deşifreleme algoritmasının çıkışı ise Şekil 5.29'da verilmiştir. İlgili görselde deşifrelenmiş metnin çıkışı ve işlemin bittiğine işaret eden *last* sinyal çıkışları görülmektedir.

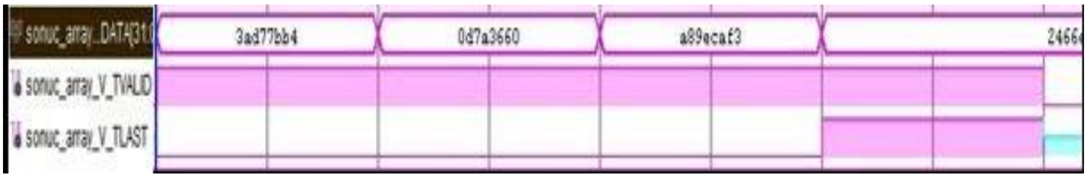


```
6bc1bee2 0
2e409f96 0
e93d7e11 0
7393172a 1
```

Şekil 5.29 : Vivado HLS ile tasarlanan şifreleme algoritmasının test dosyası sonucu.

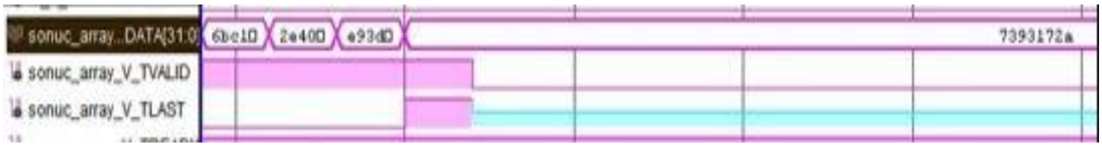
Oluşturulan yeni modüllerin Vivado HLS testinden sonra FIFO'dan doğru veri alıp almadığı Vivado üzerinden yazılan bir verilog test dosyası üzerinden test edilmiştir. Vivado üzerinden yapılan bu testte oluşturulan modüllerin giriş ve çıkışlarına FIFO'lar bağlanarak veri akışı sağlanmıştır. FIFO'lar modül bloğu kataloğu ile blok dizayn oluşturulmadan eklenmiştir. AXI4-Stream arayüzüne uygun FIFO modülü seçilmiştir ve 32 bitlik dizi alabilmesi için veri büyüklükleri ayarlanmış ve *last* sinyal çıkış portu aktif hale getirilmiştir. FIFO'ların içeriği bir dosya ile okunmuş ve doldurulmuştur. Şifreleme ve deşifreleme modülleri ise girişteki FIFO'dan 256 bit veri çekerek anahtar ve mesaj metinlerini oluşturmakta ve çıkışa işe 128 bit çıkış bit dizisi verilmektedir. Her giriş ve çıkış 32 birlik paketler şeklinde oluşmaktadır, yani girişte 8 adet paket bulunurken çıkışta ise 4 adet veri paketi bulunmaktadır. Verilerin doğruluğu ve düzgünlüğü için ise modüllerin *last* çıkışları kullanılmıştır. Alınan ilk bit paketi ile yanan *valid* sinyali sayesinde geçerli dosyanın iletilmek istenen veri olduğu belirtilir. En son veri paketi gönderildiği an ise *last* sinyali yanarak yapılan işlemin bittiği belirtilir. Test esnasında iki modül için de görülen sinyal çıkışları aşağıda Şekil 5.30, Şekil 5.31, Şekil 5.32 ve Şekil 5.33'te verilmiştir.

Şekil 5.30 : AXI4_Stream arayüzlü AES-128 şifreleme algoritmasının veri çekme testi.



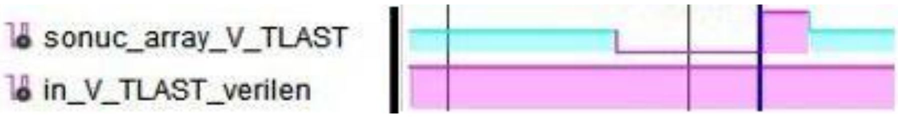
Şekil 5.31 : AXI4_Stream arayüzlü AES-128 şifreleme algoritmasının veri çıkarma testi.

Şekil 5.32 : AXI4_Stream arayüzlü AES-128 deşifreleme algoritmasının veri çekme testi.



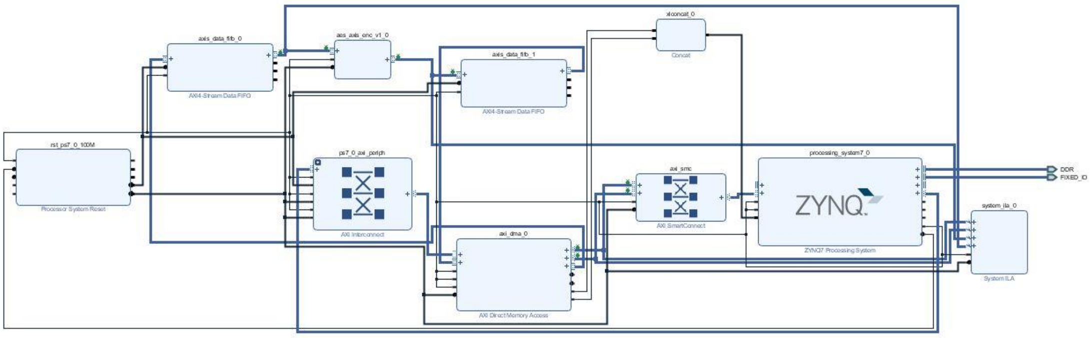
Şekil 5.33 : AXI4_Stream arayüzlü AES-128 deşifreleme algoritmasının veri çıkarma testi.

Bu çıkışların yanı sıra her iki modülde de görülen çıkış *last* sinyali normal durumlarda belirsiz olarak görülse de 1 olması gereken konumda belirli duruma geçmektedir. Bu durum Şekil 5.34’te görülmektedir. Bunun kaynağı Vivado HLS olarak düşünülmektedir. Bu belirsiz durum oluşmaması için FIFO’lar üzerinden veri aktarımı yürütülmektedir.

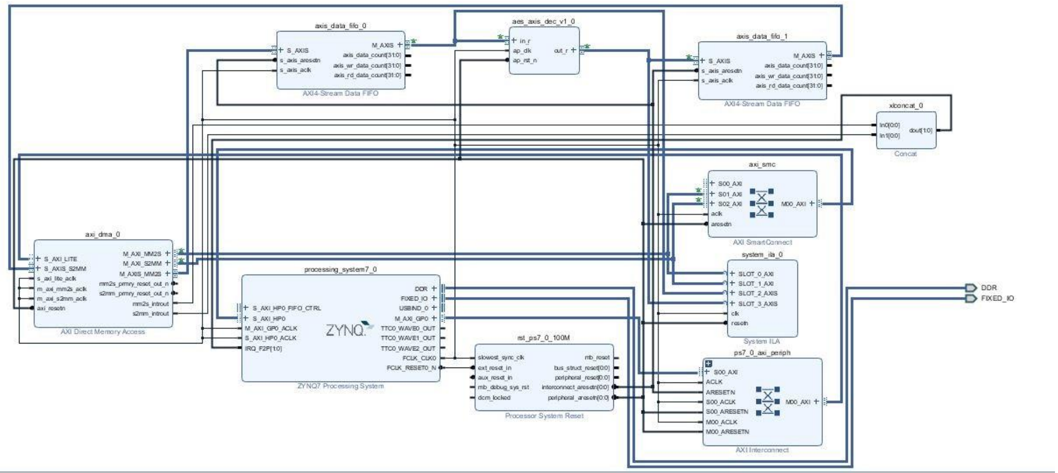


Şekil 5.34 : AXI4_Stream arayüzlü AES-128 deşifreleme algoritmasının çıkış *last* sinyali.

Yazılan test dosyasının davranışsal simülasyonunun istenildiği gibi elde edilmesinden sonra DMA modülünün tek modüllü sistem tasarımına geçildi. Bu sistem için ilk blok dizayn oluşturuldu ve burada Zynq işlemci sisteme yerleştirildi. Özelleştirilmesinde ilk olarak *scatter-gather* özelliği kapatıldı ve gerekli master-slave çıkışları aktif hale getirildi. Daha sonra dizayna DMA kontrolörü eklendi ve bu kontrolör bağlantısı otomatik olarak yapıldı. Sonrasında sisteme eklenen FIFO’lar ve modül ile bağlantıları yapıldıktan sonra tasarım tamamlandı. Tasarımda FIFO’ların girişini ve çıkışını canlı akış içerisinde kontrol etmek amacıyla bu uçlarına ILA bağlanmıştır. Gerekli dizayn tamamlandıktan sonra blok dizaynın *wrapper* komutuna takılır ve *bitstream* dosyası üretilir. Bu aşamalar tamamlandıktan sonra ise SDK ortamına geçilir. Oluşturulan blok dizayn Şekil 5.35’te gösterilmiştir. Aynı sistem deşifreleme için de oluşturulmuştur. Deşifreleme modülünün şeması Şekil 5.36’de gösterilmiştir.



Şekil 5.35 : DMA ile çalışan sistemin tek modülle tasarlanan sistemin blok şeması.



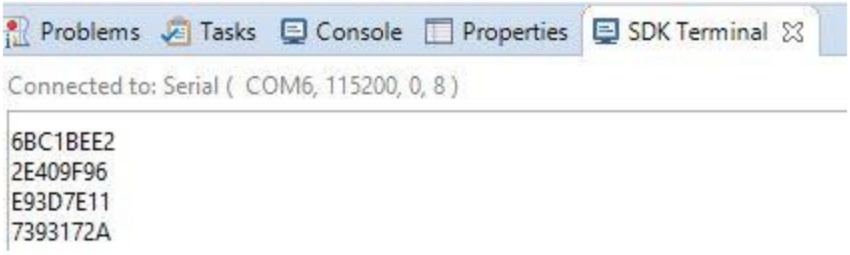
Şekil 5.36 : DMA ile çalışan sistemin tek modülle tasarlanan sistemin blok şeması.

SDK ortamında yazılan C kodu ile DMA kontrolü sağlanması ve gerekli modüllere verilerin gönderilmesi amaçlanmaktadır. Mikroişlemci kodunda DMA kontrolörünün kontrol edilmesi amaçlanmıştır, bu yüzden *device_id* kısımlarında bu elemanların adres bilgisine yer verilmiştir. İlk olarak girilmesi istenen mesaj ve anahtar bilgileri yazılan bir fonksiyonla RAM üzerine yazılmıştır. RAM üzerinden alınan ve verilen veriler DMA özelliğinden kaynaklandığı için 8 bitlik diziler halinde gerçekleşmektedir. Bellek bilgisi alınıp şifreleme/deşifreleme modülünün girişindeki FIFO'ya doldurulur ve FIFO'nun dolmasıyla şifreleme/deşifreleme modülü harekete geçer. Elde ettiği çıkışı ise çıkışındaki FIFO'ya aktarır. Çıkış FIFO'sunda ise içeriğindeki bilgi yine Belleğe yazılır. NIST'in yayınlanan örnek dosyaları SDK ortamında yazılan kodun sonrasında FPGA'e atılarak modüllerden istenilen çıkışın elde edildiği gözlemlenmiştir. Testlerin sonuç ekranı Şekil 5.37 ve Şekil 5.38'de mevcuttur.

Connected to: Serial (COM6, 115200, 0, 8)

```
3AD77BB4
0D7A3660
A89ECAF3
2466EF97
```

Şekil 5.37 : SDK ortamında DMA ile çalışan şifreleme sisteminin test sonucu.



```
Connected to: Serial ( COM6, 115200, 0, 8 )
6BC1BEE2
2E409F96
E93D7E11
7393172A
```

Şekil 5.38 : SDK ortamında DMA ile çalışan deşifreleme sisteminin test sonucu.

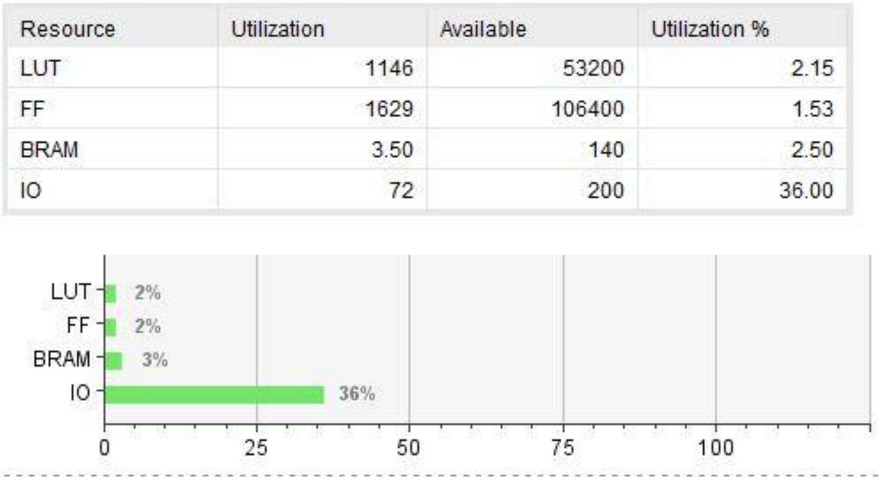
5.4 VHDL Kodu ile Karşılaştırma

Projede yazılan kodların her birinin tasarımında yüksek seviyeli diller kullanılmıştır. Bu dillerden yapılan tasarımlar ile VHDL kodları Vivado HLS yardımı ile elde edilmiştir. Bunun yanı sıra aynı kriptografi algoritmaların VHDL karşılıkları da Vivado programı kullanılarak hazırlanmıştır. VHDL dili ve yüksek seviyeli diller kullanılarak yapılan tasarımların tasarım süreleri tasarımcıya göre değişse de bu kodların karşılıklı karşılaştırmaları ileriki bölümlerde verilmiştir.

AES için şifreleme ve deşifreleme algoritmalarının algoritma yapıları itibari ile çok az değişiklikler gösterdiği için karşılaştırma sadece şifreleme algoritması üzerinden yapılmıştır. Tasarlanan modüllerden AXI4-Stream arayüzüne sahip olan modül kullanılmıştır. Yapılacak olan karşılaştırmalar ise kaynak kullanımı ve zamanlama olarak hesaplanmıştır. Vivado HLS üzerindeki kaynak kullanımı ve zamanlamalar Vivado ortamında farklı görüldüğü için referans ortamı Vivado olarak alınmıştır.

5.4.1 Kaynak kullanımı

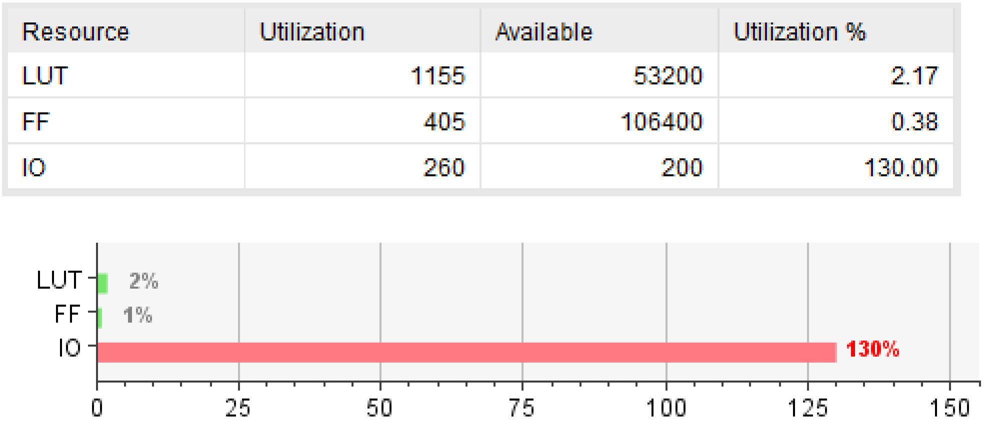
Vivado HLS üzerinden tasarlanan yapının Vivado ortamına aktarılması sonrasında o ortamda da sentezlenmesi sonucu elde edilen kaynak kullanımları Şekil 5.39’da verilmiştir. Vivado’daki sonuçlara göre incelendiğinde LUT kullanımı daha az görünürken Flip-Flop ve BRAM kullanımında %1’lik artış görülmektedir.



Resource	Utilization	Available	Utilization %
LUT	1146	53200	2.15
FF	1629	106400	1.53
BRAM	3.50	140	2.50
IO	72	200	36.00

Şekil 5.39 : Vivado HLS ortamında tasarlanan modülün kaynak kullanım oranları.

VHDL kodu ile yazılmış olan AES şifreleme algoritmasının kaynak kullanımı Şekil 5.40'da verilmiştir.



Resource	Utilization	Available	Utilization %
LUT	1155	53200	2.17
FF	405	106400	0.38
IO	260	200	130.00

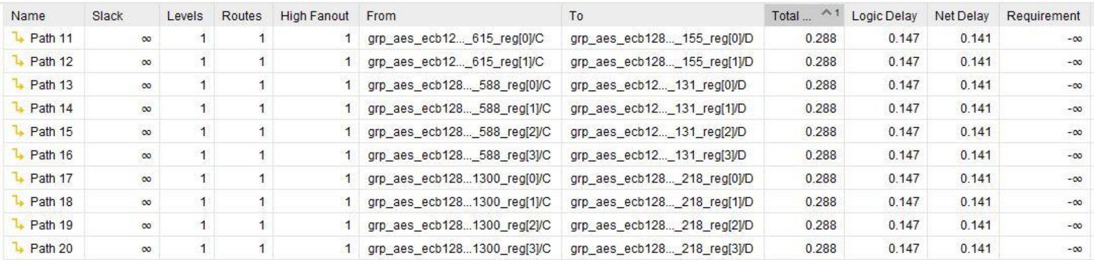
Şekil 5.40 : Vivado HLS ortamında tasarlanan modülün kaynak kullanım oranları.

5.4.2 Zamanlama

Önceden bahsedilen zaman kullanımlarına ek olarak Vivado ortamında elde edilen zaman gecikmeleri Şekil 5.41 ve Şekil 5.42'de verilmiştir.

Name	Slack	Levels	Routes	High Fanout	From	To	Total ...	Logic Delay	Net Delay	Requirement	Source Clock
Path 4	∞	5	5	96	i_1_fu_80_reg[4]/C	p_Val2_1_fu_84_reg[100]/R	4.615	1.145	3.470	∞	
Path 5	∞	5	5	96	i_1_fu_80_reg[4]/C	p_Val2_1_fu_84_reg[101]/R	4.615	1.145	3.470	∞	
Path 6	∞	5	5	96	i_1_fu_80_reg[4]/C	p_Val2_1_fu_84_reg[102]/R	4.615	1.145	3.470	∞	
Path 7	∞	5	5	96	i_1_fu_80_reg[4]/C	p_Val2_1_fu_84_reg[103]/R	4.615	1.145	3.470	∞	
Path 8	∞	5	5	96	i_1_fu_80_reg[4]/C	p_Val2_1_fu_84_reg[104]/R	4.615	1.145	3.470	∞	
Path 9	∞	5	5	96	i_1_fu_80_reg[4]/C	p_Val2_1_fu_84_reg[105]/R	4.615	1.145	3.470	∞	
Path 10	∞	5	5	96	i_1_fu_80_reg[4]/C	p_Val2_1_fu_84_reg[106]/R	4.615	1.145	3.470	∞	
Path 3	∞	3	3	11	exitcond2_reg_392_reg[0]/C	out_r_TLAST[0]	4.944	3.357	1.587	∞	
Path 2	∞	3	4	11	out_r_TREADY	out_r_TVALID	5.253	3.654	1.599	∞	input port clock
Path 1	∞	3	4	14	in_r_TVALID	in_r_TREADY	5.255	3.656	1.599	∞	input port clock

Şekil 5.41 : Vivado HLS ortamında tasarlanan şifreleme modülünün kurulma zamanları.



Name	Slack	Levels	Routes	High Fanout	From	To	Total ...	Logic Delay	Net Delay	Requirement
Path 11	∞	1	1	1	grp_aes_ecb12..._615_reg[0]/C	grp_aes_ecb128..._155_reg[0]/D	0.288	0.147	0.141	-∞
Path 12	∞	1	1	1	grp_aes_ecb12..._615_reg[1]/C	grp_aes_ecb128..._155_reg[1]/D	0.288	0.147	0.141	-∞
Path 13	∞	1	1	1	grp_aes_ecb128..._588_reg[0]/C	grp_aes_ecb12..._131_reg[0]/D	0.288	0.147	0.141	-∞
Path 14	∞	1	1	1	grp_aes_ecb128..._588_reg[1]/C	grp_aes_ecb12..._131_reg[1]/D	0.288	0.147	0.141	-∞
Path 15	∞	1	1	1	grp_aes_ecb128..._588_reg[2]/C	grp_aes_ecb12..._131_reg[2]/D	0.288	0.147	0.141	-∞
Path 16	∞	1	1	1	grp_aes_ecb128..._588_reg[3]/C	grp_aes_ecb12..._131_reg[3]/D	0.288	0.147	0.141	-∞
Path 17	∞	1	1	1	grp_aes_ecb128...1300_reg[0]/C	grp_aes_ecb128..._218_reg[0]/D	0.288	0.147	0.141	-∞
Path 18	∞	1	1	1	grp_aes_ecb128...1300_reg[1]/C	grp_aes_ecb128..._218_reg[1]/D	0.288	0.147	0.141	-∞
Path 19	∞	1	1	1	grp_aes_ecb128...1300_reg[2]/C	grp_aes_ecb128..._218_reg[2]/D	0.288	0.147	0.141	-∞
Path 20	∞	1	1	1	grp_aes_ecb128...1300_reg[3]/C	grp_aes_ecb128..._218_reg[3]/D	0.288	0.147	0.141	-∞

Şekil 5.42 : Vivado HLS ortamında tasarlanan şifreleme modülünün bekleme zamanları.

Vivado HLS üzerinden tasarlanan algoritmanın yazılan test dosyası incelendiğinde bir şifreleme işleminin yaklaşık olarak 719 saat döngüsü devam etmektedir. Bu sayı üstte belirtilen Vivado HLS programının tahmin sonuçları ile fark göstermektedir, çünkü Vivado HLS bu işlemlerin yaklaşık olarak 767 saat döngüsü süreceğini öngörmüştür. Sentez sırasında ise saat periyodu azaltacak şekilde strateji seçimleri yapılmıştır. Periyot süresini en minimuma indirmek amacıyla saat limitleri konulmuş ve 4.5 ns olarak belirtilen bu limitlemede Şekil 5.43'teki sonuçlar elde edilmiştir Elde edilen sonuç donanıma daha yakın sonuç veren implementasyon aşamasındandır. Sentez sonucu yaklaşık olarak 4.8ns olarak limitlemiş olsa da implementasyon aşamasında daha çok denemeler sonucu bu sonucu elde etmiştir.

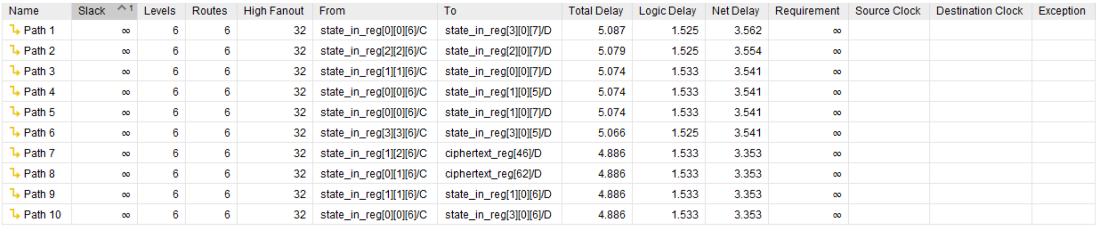
Design Timing Summary

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	0,214 ns	Worst Hold Slack (WHS):	0,031 ns	Worst Pulse Width Slack (WPWS):	1,750 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	3614	Total Number of Endpoints:	3614	Total Number of Endpoints:	1640

All user specified timing constraints are met.

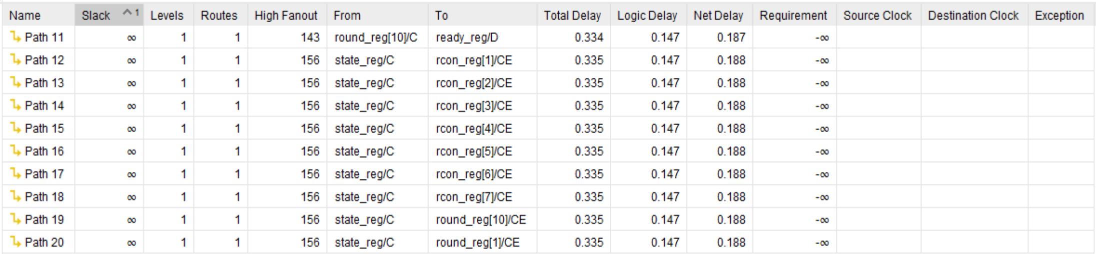
Şekil 5.43 : Vivado HLS ortamında tasarlanan şifreleme modülünün zamanlama özeti

Görüldüğü üzere 4.3ns’lere kadar inilebileceği saptanmıştır. VHDl ile yazılan algoritmanın ise kurulma zamanlarında gösterdiği gecikmeler ve bekleme zamanındaki gecikmeler Şekil 5.44 ve Şekil 5.45’te gösterilmiştir. Vivado HLS ile elde edilen kodlarda yaşanan net gecikme en fazla 3,470 ns mertebesinde iken VHDL ile elde edilen kodda bu sayı 3,562 ns mertebesindedir.



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 1	∞	6	6	32	state_in_reg[0][0][6]/C	state_in_reg[3][0][7]/D	5.087	1.525	3.562	∞			
Path 2	∞	6	6	32	state_in_reg[2][2][6]/C	state_in_reg[2][0][7]/D	5.079	1.525	3.554	∞			
Path 3	∞	6	6	32	state_in_reg[1][1][6]/C	state_in_reg[0][0][7]/D	5.074	1.533	3.541	∞			
Path 4	∞	6	6	32	state_in_reg[0][0][6]/C	state_in_reg[1][0][5]/D	5.074	1.533	3.541	∞			
Path 5	∞	6	6	32	state_in_reg[0][0][6]/C	state_in_reg[1][0][7]/D	5.074	1.533	3.541	∞			
Path 6	∞	6	6	32	state_in_reg[3][3][6]/C	state_in_reg[3][0][5]/D	5.066	1.525	3.541	∞			
Path 7	∞	6	6	32	state_in_reg[1][2][6]/C	ciphertext_reg[46]/D	4.886	1.533	3.353	∞			
Path 8	∞	6	6	32	state_in_reg[0][1][6]/C	ciphertext_reg[62]/D	4.886	1.533	3.353	∞			
Path 9	∞	6	6	32	state_in_reg[1][1][6]/C	state_in_reg[1][0][6]/D	4.886	1.533	3.353	∞			
Path 10	∞	6	6	32	state_in_reg[0][0][6]/C	state_in_reg[3][0][6]/D	4.886	1.533	3.353	∞			

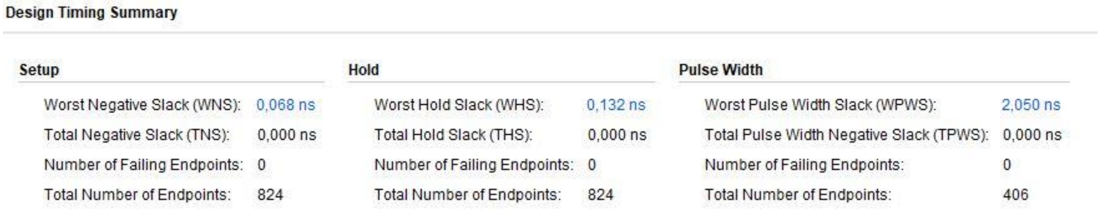
Şekil 5.44 : VHDL dili ile tasarlanan şifreleme modülünün kurulma zamanları.



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 11	∞	1	1	143	round_reg[10]/C	ready_reg/D	0.334	0.147	0.187	-∞			
Path 12	∞	1	1	156	state_reg/C	rcon_reg[1]/CE	0.335	0.147	0.188	-∞			
Path 13	∞	1	1	156	state_reg/C	rcon_reg[2]/CE	0.335	0.147	0.188	-∞			
Path 14	∞	1	1	156	state_reg/C	rcon_reg[3]/CE	0.335	0.147	0.188	-∞			
Path 15	∞	1	1	156	state_reg/C	rcon_reg[4]/CE	0.335	0.147	0.188	-∞			
Path 16	∞	1	1	156	state_reg/C	rcon_reg[5]/CE	0.335	0.147	0.188	-∞			
Path 17	∞	1	1	156	state_reg/C	rcon_reg[6]/CE	0.335	0.147	0.188	-∞			
Path 18	∞	1	1	156	state_reg/C	rcon_reg[7]/CE	0.335	0.147	0.188	-∞			
Path 19	∞	1	1	156	state_reg/C	round_reg[10]/CE	0.335	0.147	0.188	-∞			
Path 20	∞	1	1	156	state_reg/C	round_reg[1]/CE	0.335	0.147	0.188	-∞			

Şekil 5.45 : VHDL dili ile tasarlanan şifreleme modülünün bekleme zamanları.

VHDL ile yazılan kodun sentez sonucu Şekil 5.46’de verilmiştir. Sentez aşamasında saat limitleri konmuş ve elde edilen minimum periyot zamanı 5.1 ns olarak belirlenmiştir.



Design Timing Summary

Setup

Worst Negative Slack (WNS): 0,068 ns
Total Negative Slack (TNS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 824

Hold

Worst Hold Slack (WHS): 0,132 ns
Total Hold Slack (THS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 824

Pulse Width

Worst Pulse Width Slack (WPWS): 2,050 ns
Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 406

Şekil 5.46 : VHDL ile tasarlanan şifreleme modülünün zamanlama özeti

Zamanlama özetinde bakıldığı zaman VHDL ile sentezlenen sonucun her türlü 5 ns yukarısında kaldığı görülmektedir.

Elde edilen bütün sonuçlar bir tabloya aktarılmış hali Çizelge 5.3’te mevcuttur.

Çizelge 5.3 : Farklı şekillerde oluşturulmuş algoritmaların karşılaştırılması.

	Zaman döngüsü sayısı	**Minimum periyot**	**LUT**	**Flip Flop**	**BRAM**
Vivado HLS	719	≈4.3 ns	%2	%2	%3
VHDL	23	≈5.1 ns	%2	%1	%0

Minimum periyotlarda az bir fark söz konusu iken zaman döngüsü kısmında büyük farklar mevcuttur. Kullanılan LUT sayıları yaklaşık olarak aynı iken Flip-Flop ve BRAM kullanımında farklar vardır. Tablo incelendiğinde VHDL ile üretilen kodun minimum periyot süresinin fazla olmasına rağmen zaman döngüsündeki ciddi fark yüzünden daha tercih edilebilir olduğu söylenebilir. Bunun yanı sıra Vivado HLS üzerinden gösterilen özet bilgilerinin Vivado sentezi sonucu ile uyuşmadığı görülmektedir. Vivadonun sunduğu kaynak değerleri LUT haricinde yakınken işlemin sürdüğü zaman döngü sayısı ve LUT değerinde kayda değer farklar olduğu gözlemlenmiştir. Bütün farklılıkların nedeni VHDL dilinin donanıma dönük bir yazım şeklinin olmasından dolayı daha optimal bir sonuç üretmesidir.

6. DMA KULLANILARAK HAZIRLANAN HABERLEŞME SİSTEMİ

6.1 Kullanılan Haberleşme Protokolü

Sistemde kullanılan haberleşme protokolü basit *Station-to-Station (STS)* protokolüdür ve Şekil 6.1'de genel hatlarıyla gösterilmiştir. Bu protokolde haberleşen kişiler A ve B olarak adlandırılabilir. Bu kişiler arasında ilk olarak imza işlemleri ile kendilerini tanıtım ve bu işlem başarı ile sonuçlandığında veri aktarımı gerçekleşecektir.

Haberleşme protokolü başlamadan önce birkaç olay varsayılır. Bunlar kişilerin aynı α ve p sayılarına sahip olduğudur. Bunun yanı sıra iki kişi de kendilerine ait birer gizli anahtara ve X sayısına sahiptir, bunlar kimseyle paylaşılmaz. N_a, N_b ve kişilerin açık anahtarları da bilgileri her iki kişi tarafından biliniyordur [13].

Protokolün ilk basamağı kişilerin kendilerini tanıtımıdır. Gönderici olan kişi karşı tarafın kim olduğunu anlaması adına rastgele bir r sayı dizisi gönderir ve B'nin bu sayı dizisini imzalamasını ister. B de bu sayı değerini SHA-3 algoritmasını kullanarak özetler ve bu özete RSA algoritmasını kullanarak denklem 6.1'deki işlemi gerçekleştirir.

$$(SHA3(r))^{(gizli\ anahtar\ B)} mod N_b = Y_b \tag{6.1}$$

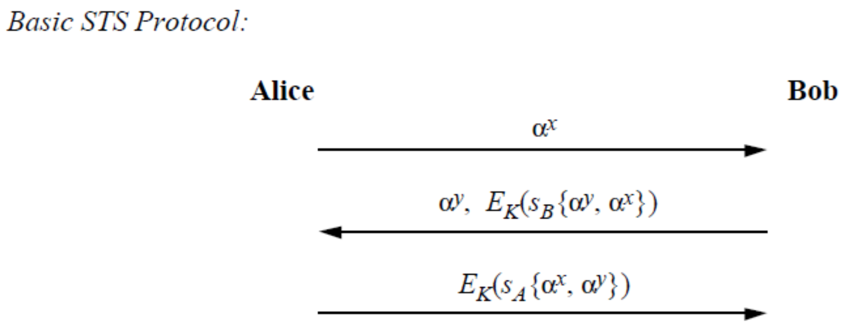
Karşı taraf bu sonucu alır ve sonucu alır ve yine RSA algoritmasını kullanarak denklem 6.2'deki işlemi gerçekleştirir.

$$(Y_b)^{(açık\ anahar\ B)} mod N_b \tag{6.2}$$

A kişisi de SHA-3 ile özetleme yapar ve bu ikisi sonucunun eş olup olmadığına bakar. Eğer bu değerler eşit çıkarsa karşı taraftaki kişinin varsayılan kişi olduğu anlaşılır. Böylece tanıtım işi biter ve anahtar paylaşım aşamasına geçilir. Kişi tanıtımı çoklu sistemlerde bir liste üzerinden işler. Birçok kişinin açık anahtarı bulunan bir liste tek tek işleme sokulur ve özetleme sonucu ile aynı olup olmadığı kontrol edilir. Eğer sonuçlar eşleşirse, kişinin o açık anahtarın sahibi olduğu anlaşılır. Eşleşme durumu yaşanmazsa kişinin sistemde kaydı olmadığı ve tanınmadığı anlaşılır [13].

Anahtar paylaşımı işi için A kişisi işlem öncesinde kendisi için kimseyle paylaşmadığı bir X_A değeri seçer, aynı şekilde de B kişisi de bir X_B sayısı belirler. Önceden varsayılan α ve p değerleri için A kişisi $\alpha^{X_A} mod\ p$ işlemini gerçekleştirip bu bilgiyi B kişisine aktarır. Aynı şekilde B kişisi de kendisine sakladığı X_B bilgisini ortak olarak iki kişide de bulunan α ve p değerlerini de kullanarak $\alpha^{X_B} mod\ p$ işlemini gerçekleştirerek bu bilgiyi A kişisine yollar. Bu işlemlerde yine RSA algoritması kullanılır. A kişisi elde ettiği verinin x_A kuvvetini alır, yani $(\alpha^{X_B}\, mod\, p)^{X_A}$ işlemini gerçekleştirir. Bu işlemin amacı AES için kullanılması gereken anahtarın eldesidir. Aynı şekilde de B kişisi gelen bilgilerin x_B kuvvetini alarak aynı anahtarı üretir. $(\alpha^{X_A}\, mod\, p)^{X_B}$ işlemine bakıldığında üs değişimi özelliği yüzünden $(\alpha^{X_B}\, mod\, p)^{X_A}$ ile aynı olduğu görülür. Bu şekilde bir sonraki aşama için gereken anahtar paylaşımı da gerçekleşmiş olur. Anahtar üretimi sonrasında x_A ve x_B sayıları silinir.

Kullanılacak AES algoritması için 128 bit anahtar gereklidir, fakat anahtar paylaşımı ile elde edilen değer 1024 bittir. Bunun için iki tarafın da önceden bildiği şekilde bu 1024 bit parçalanır ve gerekli 128 bit elde edilir. İki taraf da bu sayede aynı anahtara sahip olur. Bu aşamalar tamamlandıktan sonra A kişisi göndermek istediği mesajı AES algoritmasına sokar, anahtar olarak da önceki işlemden elde edilen anahtar kullanılır. Şifrelenmiş metin B kişisine yollanır. B kişisi de elinde bulunan aynı anahtar ile şifreyi çözer ve gönderilmek istenen metni elde eder [13].



Şekil 6.1 : STS protokolü şeması [13].

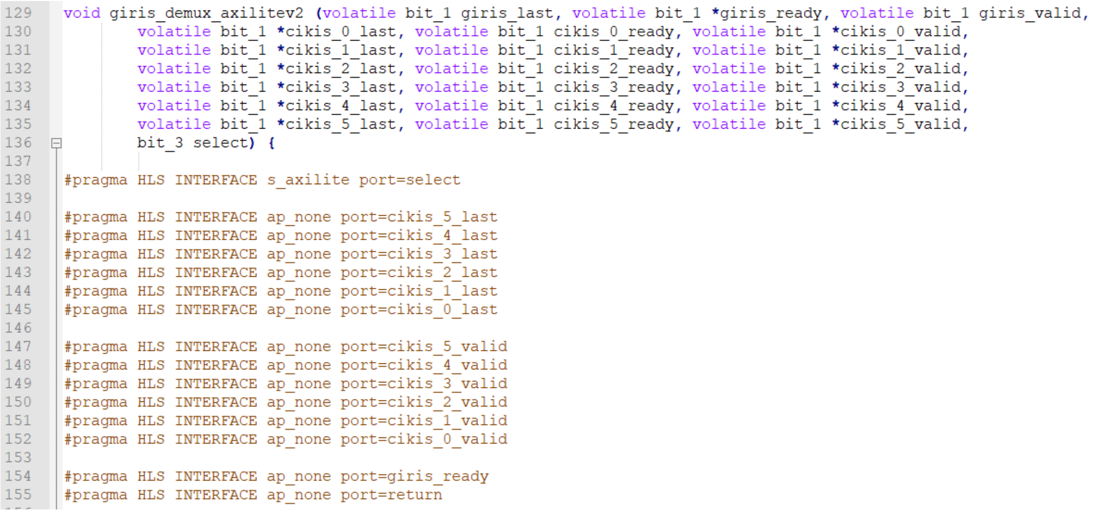
6.2 ZYNQ Tabanlı DMA ile Çalışan Sistem

Projenin önceki aşamalarında tasarlanan modül blokları belirtilen haberleşme protokolüne uygun olarak kriptografi sistemini gerçeklemek amacıyla birleştirilmiştir. Kriptografi sistemi ZYNQ işlemci tabanlı olup AXI DMA kullanarak modüller arasındaki veri haberleşmesini sağlamaktadır. Sisteme yerleştirilen AXI DMA modülü tek kanallı olacak şekilde ayarlanmıştır. Bu tek kanallı yapı ile birden fazla modülün haberleşmesi sağlanacağından bu modüllerin giriş ve çıkışlarını kontrol etmek amacıyla çoğullayıcı ve çoğullama çözücü yapısına sahip iki yeni modül tasarlanmıştır. Bu modüllerin tasarımı Vivado HLS ortamında gerçekleşmiştir ve AXI4-Stream arayüzü ile tasarlanmışlardır.

6.2.1 Vivado HLS ile tasarlanan çoğullayıcı ve çoğullama çözücü modüller

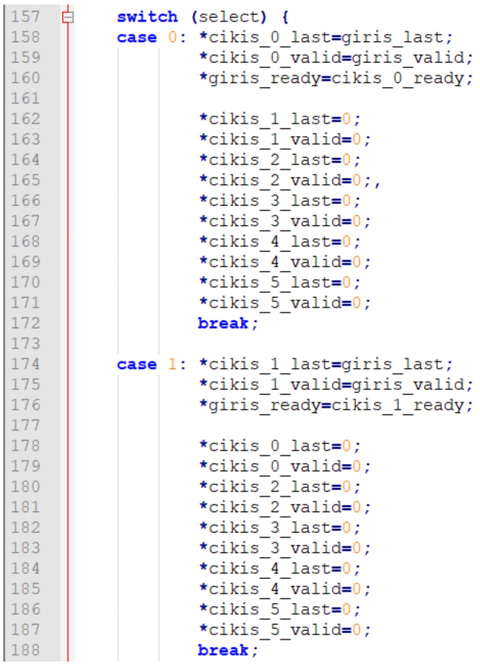
Gerçeklenecek olan protokolde tek bir adet DMA modülü ve 4 farklı tasarlanmış IP olacağından, aradaki veriyolu bağlantılarını sağlamak için çoğullayıcı ve çoğullama çözücüye ihtiyaç vardır. Örneğin DMA ile 4 farklı IP arasına bir adet çoğullama çözücü konulması gerekmektedir. Bu çoğullama çözücünün seçim bitlerini kontrol edecek olan modül ise mikroişlemci olacaktır. Bu sayede mikroişlemci istediği zaman veriyolunu 4 farklı IP'nin girişlerine bağlı olan FIFO'larına bağlayabilecektir.

Çoğullama çözücü Şekil 6.1 ve Şekil 6.2'de gösterildiği üzere HLS ortamında tasarlanmıştır. *giris_demux_axilitev2* isimli fonksiyonun giriş çıkışları tanımlanırken *volatile* değişkeni kullanılmıştır. Bunun sebebi çoğullama çözücünün asenkron bir devre olması ve bağlı olan bütün sinyallerin anlık değişimlerinin saat darbesine bağlı olmadan, değişen değerleri algılamasının istenmesindendir. HLS programının hangi değişkenlerin çıkış hangilerinin giriş olarak tanımlandığını algılayamaması ve buna özel bir tanımlamanın bulunmaması sebebiyle, modülün çıkışı olan değişkenler *pointer* olarak tanımlanmıştır.



```
void giris_demux_axilitev2 (volatile bit_1 giris_last, volatile bit_1 *giris_ready, volatile bit_1 giris_valid,
        volatile bit_1 *cikis_0_last, volatile bit_1 cikis_0_ready, volatile bit_1 *cikis_0_valid,
        volatile bit_1 *cikis_1_last, volatile bit_1 cikis_1_ready, volatile bit_1 *cikis_1_valid,
        volatile bit_1 *cikis_2_last, volatile bit_1 cikis_2_ready, volatile bit_1 *cikis_2_valid,
        volatile bit_1 *cikis_3_last, volatile bit_1 cikis_3_ready, volatile bit_1 *cikis_3_valid,
        volatile bit_1 *cikis_4_last, volatile bit_1 cikis_4_ready, volatile bit_1 *cikis_4_valid,
        volatile bit_1 *cikis_5_last, volatile bit_1 cikis_5_ready, volatile bit_1 *cikis_5_valid,
        bit_3 select) {

#pragma HLS INTERFACE s_axilite port=select

#pragma HLS INTERFACE ap_none port=cikis_5_last
#pragma HLS INTERFACE ap_none port=cikis_4_last
#pragma HLS INTERFACE ap_none port=cikis_3_last
#pragma HLS INTERFACE ap_none port=cikis_2_last
#pragma HLS INTERFACE ap_none port=cikis_1_last
#pragma HLS INTERFACE ap_none port=cikis_0_last

#pragma HLS INTERFACE ap_none port=cikis_5_valid
#pragma HLS INTERFACE ap_none port=cikis_4_valid
#pragma HLS INTERFACE ap_none port=cikis_3_valid
#pragma HLS INTERFACE ap_none port=cikis_2_valid
#pragma HLS INTERFACE ap_none port=cikis_1_valid
#pragma HLS INTERFACE ap_none port=cikis_0_valid

#pragma HLS INTERFACE ap_none port=giris_ready
#pragma HLS INTERFACE ap_none port=return
```

Şekil 6.2 : *giris_demux_axilitev2* modülünün HLS tasarımı.



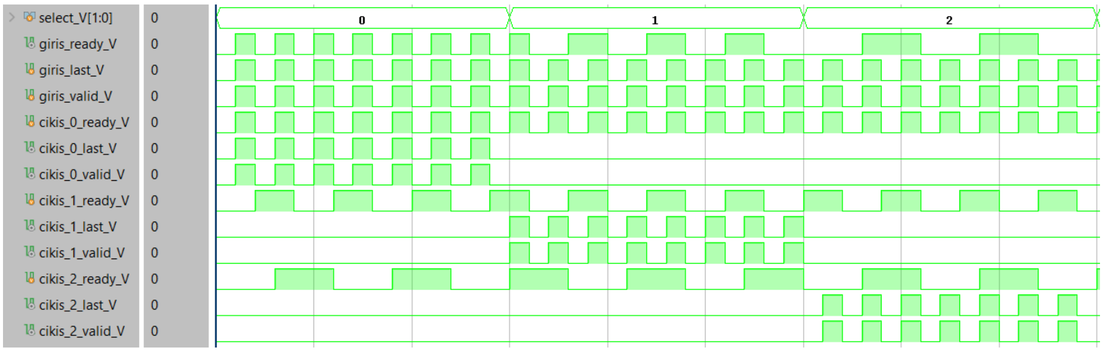
```
    switch (select) {
    case 0: *cikis_0_last=giris_last;
            *cikis_0_valid=giris_valid;
            *giris_ready=cikis_0_ready;

            *cikis_1_last=0;
            *cikis_1_valid=0;
            *cikis_2_last=0;
            *cikis_2_valid=0;,
            *cikis_3_last=0;
            *cikis_3_valid=0;
            *cikis_4_last=0;
            *cikis_4_valid=0;
            *cikis_5_last=0;
            *cikis_5_valid=0;
            break;

    case 1: *cikis_1_last=giris_last;
            *cikis_1_valid=giris_valid;
            *giris_ready=cikis_1_ready;

            *cikis_0_last=0;
            *cikis_0_valid=0;
            *cikis_2_last=0;
            *cikis_2_valid=0;
            *cikis_3_last=0;
            *cikis_3_valid=0;
            *cikis_4_last=0;
            *cikis_4_valid=0;
            *cikis_5_last=0;
            *cikis_5_valid=0;
            break;
```

Şekil 6.3 : *giris_demux_axilitev2* modülünün HLS tasarımın seçim kısmı.

Fonksiyon bir çoğullama çözücü işlevi göreceğinden, *switch-case* yapısı kullanılmıştır. Çoğullama çözücünün girişi DMA’dan çıkan ve giren AXI4-Stream sinyallerinden oluşacak, çıkışı ise FIFO’lara bağlanacak olan AXI4-Stream protokolüne ait olan sinyaller olacaktır. Bu sinyaller DMA’dan geliyor ise *valid* (giriş), *last* (giriş) ve *last* (çıkış) olarak tanımlanacak, tasarlanan IP’lerin girişine bağlı FIFO’lara gidiyor ise *valid* (çıkış), *last* (çıkış) ve *last* (giriş) olarak tanımlanacaktır. Aslında buradaki amaç AXI4-Stream protokolünün sinyallerini sadece birbirine bağlamaktır.

Durum değiştirme koşulu olarak *select* 3 bitlik değişkeni tanımlanmıştır. Bu değişkenin alacağı değere göre, örneğin ‘1’ olması durumunda, DMA’dan gelen sinyaller SHA modülünün girişindeki FIFO’ya bağlanacaktır ve geri kalan çıkış sinyalleri ‘0’ yapılacaktır. Bu sayede *select* değişkenine göre istenilen FIFO aktif hale getirilebilecek ve DMA’dan veri gönderilebilecektir.

HLS tarafından oluşturulan IP, Vivado geliştirme ortamına aktarılmış ve test kodu yazılmıştır. Ardından benzetim gerçekleştirilmiş ve Şekil 6.3’te görülen dalga formu elde edilmiştir. Burada görüleceği üzere *select_V* sinyali ‘0’ seçildiğinde *giris_last_V* ve *giris_valid_V* sinyalinin dalga formu, bağlantının kurulduğu *cikis_0_last_V* ve *cikis_0_valid_V* sinyalinde gözlenebilir. Aynı şekilde *select_V* değiştikçe *giris_ready_V* sinyalinin sırayla *cikis_0_ready_V*, *cikis_1_ready_V* ve *cikis_2_ready_V* sinyallerinin dalga formunun iletildiği görülebilmektedir.



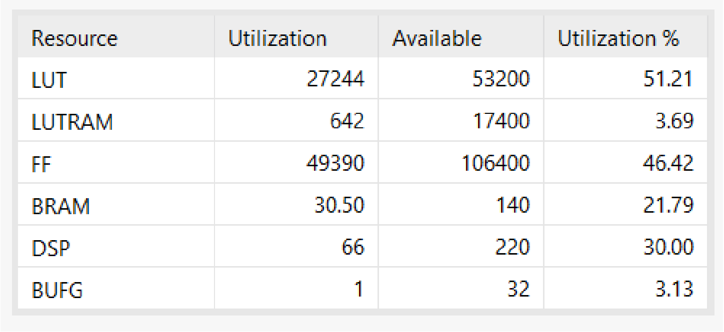
Şekil 6.4 : *giris_demux_axilitev2* modülünün benzetim sonucu.

Çoğullama çözücüsünün tasarımının tam tersi mantığı çoğullayıcıda bulunmaktadır. Çoğullayıcı, modüllerin çıkışlarında bulunan FIFO'larının AXI4-Stream sinyallerini DMA'nın ilgili giriş ve çıkışlarına aynı mantıkla bağlayacaktır.

6.2.2 Kriptografi sistemi

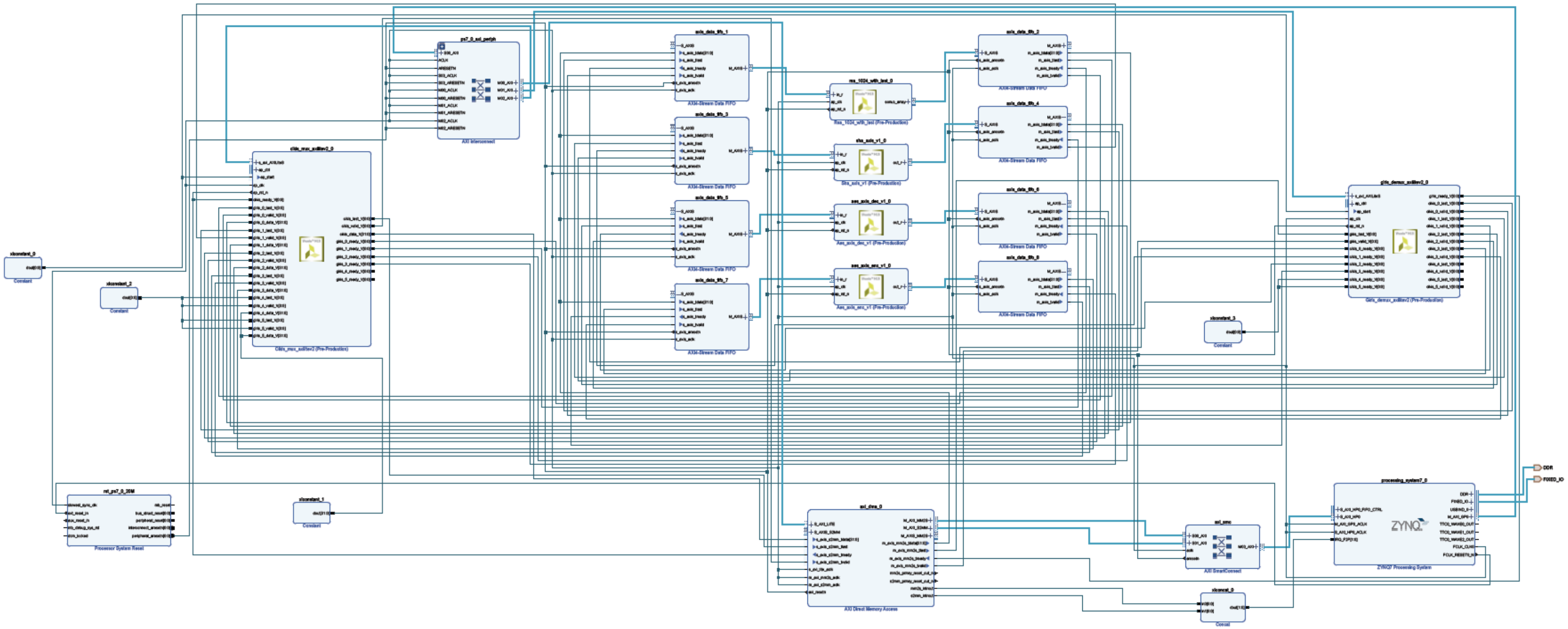
Station to station haberleşme protokolüne sahip kriptografi sistemini kurmak için Vivado ortamında ZYNQ işlemci tabanlı bir sistem gerçeklenmiştir. Bu sistemde kullanılmak üzere HLS ortamında uygun arayüzlerle tasarlanmış modüller dışarıya aktarılarak Vivado ortamında veri deposuna eklenmiştir. Sistemin gerçeklenebilmesi için bir blok tasarımı oluşturulmuştur. Blok tasarım ekranı açıldıktan sonra tasarıma ZYNQ işlemcisi eklenmiştir. Bunu takiben AXI DMA modülü de eklenerek gerekli bağlantılar sağlanmıştır. Oluşturulan AXI DMA modülünün çoklu modül sistemine erişebilmesi için sistem için özel tasarlanan çoğullayıcı ve çoğullama çözücü modül blokları sisteme eklenmiştir. Bu blokların sürekli çalışıp kombinazonsal devre gibi görev yapması için *start* girişlerine sabit 1 değeri atanmıştır. Kullanılmayacak girişlere de sabit 0 değeri atanmıştır. Daha sonra ise kriptografi sisteminde kullanılacak olan RSA, SHA-3, AES-şifreleme ve AES-deşifreleme modül blokları blok tasarıma eklenmiştir. Bu modüllerin giriş ve çıkışlarına birer FIFO eklenmiştir. FIFO'lar eklenirken veri genişlikleri 32 bit olarak ayarlanmış ve *last* sinyalleri aktif hale getirilmiştir. Bu sinyallerin aktif hale getirilmesinin amacı DMA'ya gönderilecek olan verinin son paketinin transfer edildiğini belirtebilmektir. Gerekli bağlantılar oluşturulduktan sonra Vivado ortamının özelliği kullanılarak tasarım doğrulaması yapılmıştır. Oluşturulan blok tasarım Şekil 6.6'da verilmiştir.

Buraya kadar eklenen bütün IP'ler ve yapılan bağlantılar önceki bölümlerde gerektiği gibi detaylı şekilde ele alınmıştır. Vivado geliştirme ortamında blok tasarımın implementasyonu gerçekleştirilmiş ve Şekil 6.5’te görüldüğü gibi kaynak kullanımı analiz edilmiştir.



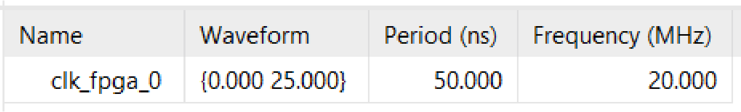
Resource	Utilization	Available	Utilization %
LUT	27244	53200	51.21
LUTRAM	642	17400	3.69
FF	49390	106400	46.42
BRAM	30.50	140	21.79
DSP	66	220	30.00
BUFG	1	32	3.13

Şekil 6.5 : STS kriptografi sisteminin kaynak kullanımı.



Şekil 6.6 : Kriptografi sisteminin blok tasarımı

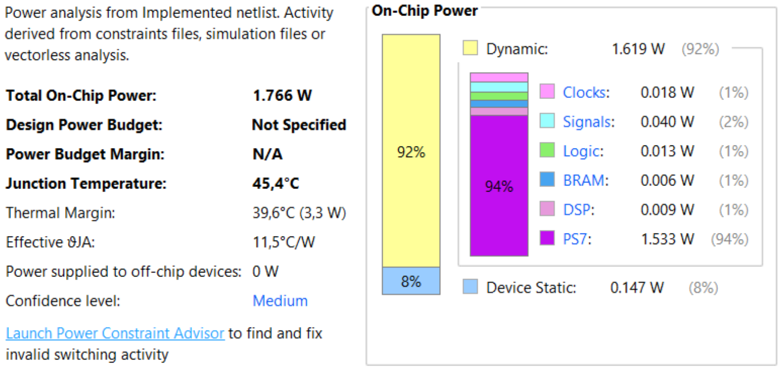
Görüldüğü üzere kaynak kullanımı sebebi ile bir verici bir de alıcı için kaynak yetersizliğinden dolayı iki adet FPGA geliştirme kartı gerekecektir. Fakat bulunan imkanlar sebebiyle tek bir FPGA kartı üzerinde hem alıcı hem de verici modeli tasarlanmıştır. Bütün sistemin saat frekansı ZYNQ işlemcinin ayarlarından Şekil 6.7'de görüldüğü üzere herhangi bir veri kaybına ve zamanlama hatasına sebebiyet vermemek için 20 MHz olarak seçilmiştir.



Name	Waveform	Period (ns)	Frequency (MHz)
clk_fpga_0	{0.000 25.000}	50.000	20.000

Şekil 6.7 : Kriptografi sisteminin saat frekansı.

Ayrıca ek bilgi olarak sistemin harcamış olduğu toplam güç 1.766W olarak sentez aracı tarafından Şekil 6.8'deki gibi analiz edilmiştir.



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.766 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**45,4°C**
Thermal Margin:	39,6°C (3,3 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨 Dynamic:	1.619 W	(92%)
🟪 Clocks:	0.018 W	(1%)
🟦 Signals:	0.040 W	(2%)
🟩 Logic:	0.013 W	(1%)
🟦 BRAM:	0.006 W	(1%)
🟪 DSP:	0.009 W	(1%)
🟪 PS7:	1.533 W	(94%)
🟦 Device Static:	0.147 W	(8%)

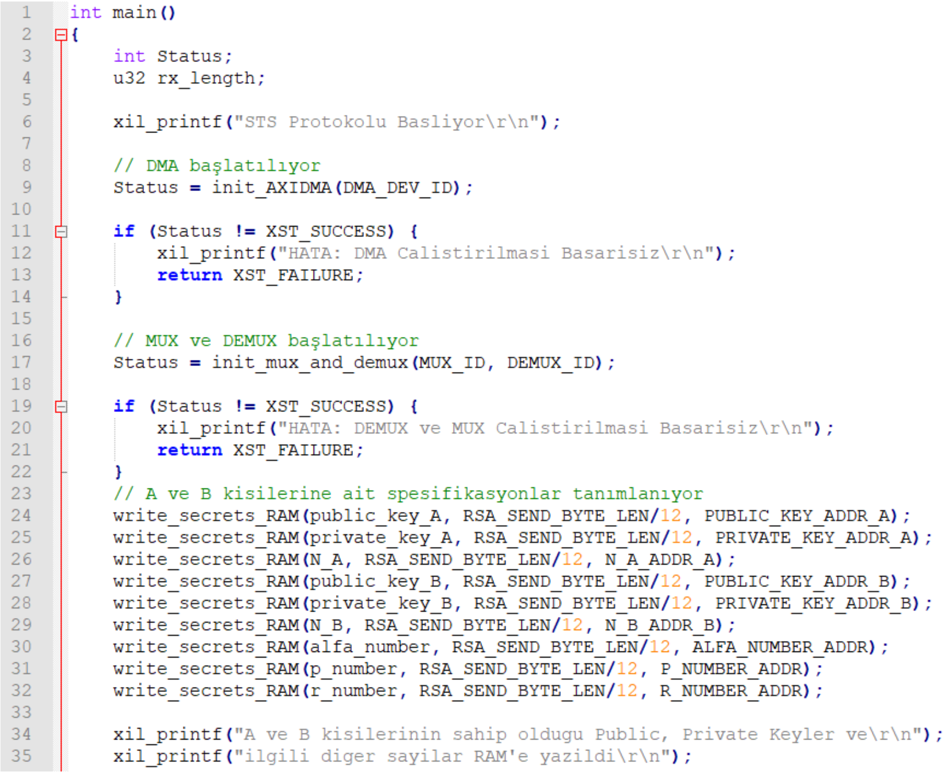
Şekil 6.8 : Kriptografi sisteminin tüketilen güç analizi.

Blok tasarımın *bitstream* dosyası oluşturulmuş ve mikroişlemci kodunun yazılması için Xilinx SDK ortamına geçilmiştir. Modüllere gönderilecek olan bütün veriler öncelikle belleğe aktarılmak için Şekil 6.9'daki gibi uygun uzunlukta veri paketleri olarak dizilere kaydedilmiştir. Burada A ve B kişisi aynı sistem içerisinde bulunan iki kişi olarak kabul edilmiştir. Kişisel anahtarlar sistem tarafından hesaplanmayıp önceden belirlenerek belleğe yazılmaktadır.

```c
// RAM'e yazılmak istenen girişler
// A kişisi
u32 public_key_A[RSA_SEND_BYTE_LEN/12]={0x00000000,0x00000000,0x00000000,0x00000000,
        0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,
        0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,
        0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,
        0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,
        0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x01305413};
u32 private_key_A[RSA_SEND_BYTE_LEN/12]={0x7f23930b ,0xa5abbfd1 ,0x9a658315 ,0x92223165 ,0xd0e930e2
        ,0x2590f42f ,0xb5d26280 ,0x45e46140 ,0x2f8199b9 ,0x3b768d8c
        ,0xbc527698 ,0x913f72b ,0x1b359399 ,0x3c326fec ,0x5deb8c6
        ,0x37f83e14 ,0x76c7c9dd ,0x4a04be49 ,0xd71b0c15 ,0xf39d4795
         ,0x1c079d7 ,0x55014e3e ,0x26303ace ,0xfe64c337 ,0x6953a5c1
         ,0xf5888e4 ,0xc8625ffe ,0x48d97697 ,0x1e6894fa ,0x17425b7
         ,0x8446bf1 ,0xa320d93f};

u32 message_decr_AES[AES_MESSAGE_BYTE_LEN/4]={0x3AD77BB4,0x0D7A3660,0xA89ECAF3,0x2466EF97};
u32 message_encr_AES [AES_MESSAGE_BYTE_LEN/4]={0x6BC1BEE2,0x2E409F96,0xE93D7E11,0x7393172A};
```

Şekil 6.9 :Yazılım kodundaki örnek dizi tanımlamaları.

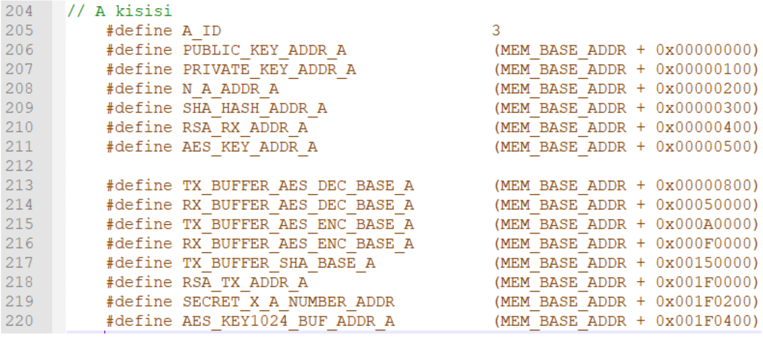
Şekil 6.10'daki ana fonksiyonun ilk bölümü incelenecek olursa öncelikle DMA modülü **Bölüm 3.3.3**'te açıklandığı şekilde *init_AXIDMA* fonksiyonu ile aktif hale getirilmektedir. Ardından DMA için kanal ayarlamasını sağlayacak olan çoğullayıcı ve çoğullama çözücü modüller *init_mux_and_demux* ile aktifleştirilmektedir. Bu işlemlerin yapılmasının amacı ilgili modülün işlemciden AXI4-Lite arayüzü üzerinden kurulum dosyalarını almasını sağlamaktır. Sonrasında tekrar **Bölüm 3.3.3**'te bahsedildiği gibi bit manipülasyon işlemlerinin uygulandığı *write_secrets_RAM* fonksiyonu ile A ve B kişisine ait özel veriler belleğe yüklenmektedir.



```c
int main()
{
    int Status;
    u32 rx_length;

    xil_printf("STS Protokolu Basliyor\r\n");

    // DMA başlatılıyor
    Status = init_AXIDMA(DMA_DEV_ID);

    if (Status != XST_SUCCESS) {
        xil_printf("HATA: DMA Calistirilmasi Basarisiz\r\n");
        return XST_FAILURE;
    }

    // MUX ve DEMUX başlatılıyor
    Status = init_mux_and_demux(MUX_ID, DEMUX_ID);

    if (Status != XST_SUCCESS) {
        xil_printf("HATA: DEMUX ve MUX Calistirilmasi Basarisiz\r\n");
        return XST_FAILURE;
    }
    // A ve B kisilerine ait spesifikasyonlar tanımlanıyor
    write_secrets_RAM(public_key_A, RSA_SEND_BYTE_LEN/12, PUBLIC_KEY_ADDR_A);
    write_secrets_RAM(private_key_A, RSA_SEND_BYTE_LEN/12, PRIVATE_KEY_ADDR_A);
    write_secrets_RAM(N_A, RSA_SEND_BYTE_LEN/12, N_A_ADDR_A);
    write_secrets_RAM(public_key_B, RSA_SEND_BYTE_LEN/12, PUBLIC_KEY_ADDR_B);
    write_secrets_RAM(private_key_B, RSA_SEND_BYTE_LEN/12, PRIVATE_KEY_ADDR_B);
    write_secrets_RAM(N_B, RSA_SEND_BYTE_LEN/12, N_B_ADDR_B);
    write_secrets_RAM(alfa_number, RSA_SEND_BYTE_LEN/12, ALFA_NUMBER_ADDR);
    write_secrets_RAM(p_number, RSA_SEND_BYTE_LEN/12, P_NUMBER_ADDR);
    write_secrets_RAM(r_number, RSA_SEND_BYTE_LEN/12, R_NUMBER_ADDR);

    xil_printf("A ve B kisilerinin sahip oldugu Public, Private Keyler ve\r\n");
    xil_printf("ilgili diger sayilar RAM'e yazildi\r\n");
```

Şekil 6.10 : Ana fonksiyonun ilk bölümü.

Veriler belleğe yüklenirken *header* dosyasında önceden belirtilmiş bellek adresleri kullanılmaktadır. Şekil 6.11’de görüleceği üzere A kişisinin kişisel anahtarı, genel anahtarı, RSA sonuç tutucusu ve AES anahtar tutucusu gibi bilgilerin hangi adreslerde tutulacağı önceden bellidir. A kişisinden B kişisine bir veri gönderileceği zaman aslında yapılan A’nın sahip olduğu bilgilerin bellekten çekilip işleme sokularak B’nin verilerinin bulunduğu adreslerden uygun olanına yazmaktır.



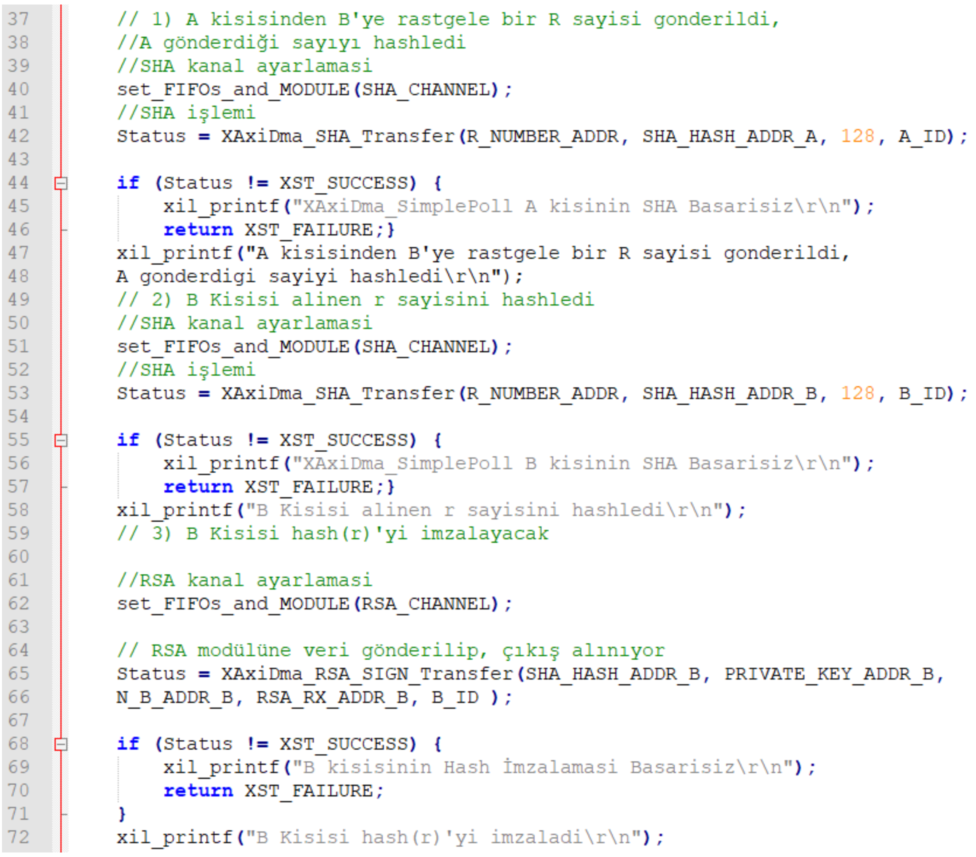
```
// A kisisi
    #define A_ID                            3
    #define PUBLIC_KEY_ADDR_A               (MEM_BASE_ADDR + 0x00000000)
    #define PRIVATE_KEY_ADDR_A              (MEM_BASE_ADDR + 0x00000100)
    #define N_A_ADDR_A                      (MEM_BASE_ADDR + 0x00000200)
    #define SHA_HASH_ADDR_A                 (MEM_BASE_ADDR + 0x00000300)
    #define RSA_RX_ADDR_A                   (MEM_BASE_ADDR + 0x00000400)
    #define AES_KEY_ADDR_A                  (MEM_BASE_ADDR + 0x00000500)

    #define TX_BUFFER_AES_DEC_BASE_A        (MEM_BASE_ADDR + 0x00000800)
    #define RX_BUFFER_AES_DEC_BASE_A        (MEM_BASE_ADDR + 0x00050000)
    #define TX_BUFFER_AES_ENC_BASE_A        (MEM_BASE_ADDR + 0x000A0000)
    #define RX_BUFFER_AES_ENC_BASE_A        (MEM_BASE_ADDR + 0x000F0000)
    #define TX_BUFFER_SHA_BASE_A            (MEM_BASE_ADDR + 0x00150000)
    #define RSA_TX_ADDR_A                   (MEM_BASE_ADDR + 0x001F0000)
    #define SECRET_X_A_NUMBER_ADDR          (MEM_BASE_ADDR + 0x001F0200)
    #define AES_KEY1024_BUF_ADDR_A          (MEM_BASE_ADDR + 0x001F0400)
```

Şekil 6.11 : *header* dosyasındaki bellek adres atamaları.

Sistemdeki bütün işlemler yapılırken işlemlerin doğru yapılıp yapılmadığına dair kontroller *Status* değişkeni yardımıyla yapılmaktadır ve herhangi bir başarısız durumda sistem hatayı belirtip çalışmayı durduracak şekilde tasarlanmıştır. Veriler belleğe yazıldıktan sonra STS protokolünün işletilmesine SHA modülünün ve kullanılması ile başlanmaktadır. Fakat modülün başlatılması için önce DMA kanalı ile bağlantının çoğullama çözücü ile sağlanması gerekir. Şekil 6.12’den görüleceği üzere bu kanal ayarlarını yapacak olan fonksiyon *set_FIFOs_and_MODULE*’dir. Fonksiyon değişkeni SHA’nın aktifleştirilmesi için ‘1’, RSA için ‘0’, AES_DEC için ‘2’ ve AES_ENC için ‘3’ tür. Bu değişken bilgileri kullanılarak çoğullayıcının *select_V* girişinin değeri değiştirilerek kanal ayarlaması yapılmaktadır.

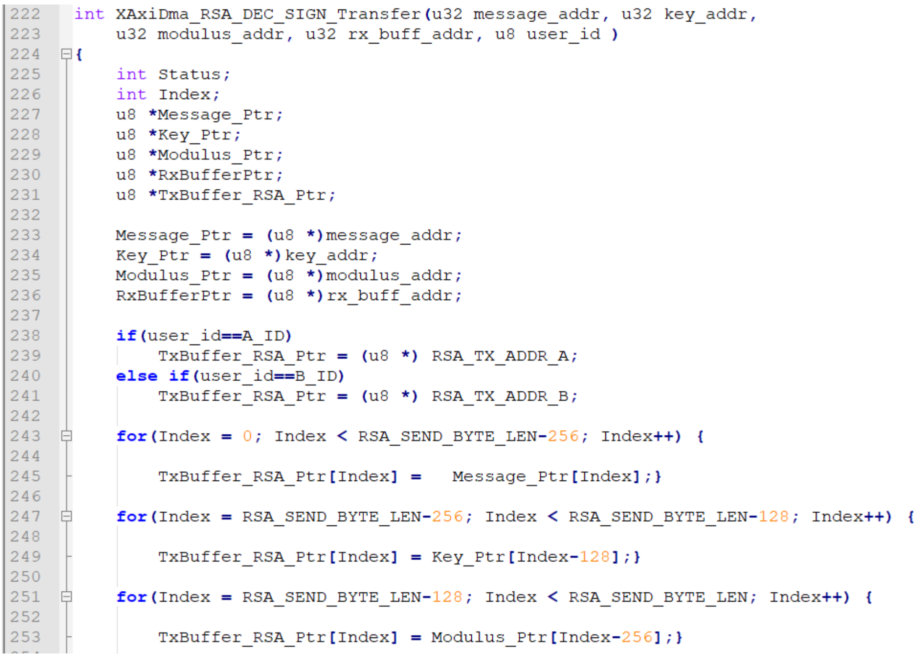
SHA, AES ve RSA işlemlerini gerçekleştiren fonksiyonlar farklıdır fakat mantıkları neredeyse aynı denilebilir. Bu sebeple örnek teşkil etmesi amacıyla Şekil 6.13’teki RSA işlemini gerçekleştiren *XAxiDma_RSA_SIGN_Transfer* fonksiyonu açıklanacaktır. Fonksiyonun ilk değişkeni olan *message_addr* RSA modülünün mesaj olarak kabul edeceği verinin bulunduğu adresi belirtmektedir. İkinci ve üçüncü değişken olan *key_addr* ve *modulus_addr* ise sırasıyla mod alma işleminin üssü ve mod alma işleminin tabanının bilgisinin bulunduğu bellek adresini belirtir. *rx_buff_addr* değişkeni RSA sonucunda hesaplanan verinin hangi adrese yazılacağının bilgisini ve *user_id* de A veya B kişilerinden hangisinin işlemi gerçekleştirdiği bilgisini taşımaktadır.

Fonksiyon içerisinde yapılan ilk işlem *TxBuffer_RSA_Ptr* işaretçisinin belirttiği ilk 128 bytelık adrese mesaj verisini, sonraki 128 byteına mod alma işleminin üssü verisini ve sonraki 128 byteına da mod alma işleminin tabanının bilgisini yazmaktır. *TxBuffer_RSA_Ptr* nin belirttiği adreslerdeki veriler aslında DMA'nın RSA modülüne göndereceği verilerdir. A'nın göndereceği veriler *RSA_TX_ADDR_A*'te, B'nin göndereceği veriler ise *RSA_TX_ADDR_B*'de saklanmaktadır. Ardından önceki bölümlerde anlatılan DMA transferi işlemi gerçekleştirilmekte ve sonuçlar eğer işlem yapan A ise *RSA_RX_ADDR_A*'da, işlemi yapan B ise *RSA_RX_ADDR_B*'de tutulmaktadır. Bu sayede iki kişinin de imzaladıkları verilerin karşılıklı doğruluğunu kontrol etmeleri için yalnızca bellekteki bu belirli adreslere bakmaları yeterli olmaktadır.



```
// 1) A kisisinden B'ye rastgele bir R sayisi gonderildi,
//A gönderdiği sayıyı hashledi
//SHA kanal ayarlamasi
set_FIFOs_and_MODULE(SHA_CHANNEL);
//SHA işlemi
Status = XAxiDma_SHA_Transfer(R_NUMBER_ADDR, SHA_HASH_ADDR_A, 128, A_ID);

if (Status != XST_SUCCESS) {
    xil_printf("XAxiDma_SimplePoll A kisinin SHA Basarisiz\r\n");
    return XST_FAILURE;}
xil_printf("A kisisinden B'ye rastgele bir R sayisi gonderildi,
A gonderdigi sayiyi hashledi\r\n");
// 2) B Kisisi alinen r sayisini hashledi
//SHA kanal ayarlamasi
set_FIFOs_and_MODULE(SHA_CHANNEL);
//SHA işlemi
Status = XAxiDma_SHA_Transfer(R_NUMBER_ADDR, SHA_HASH_ADDR_B, 128, B_ID);

if (Status != XST_SUCCESS) {
    xil_printf("XAxiDma_SimplePoll B kisinin SHA Basarisiz\r\n");
    return XST_FAILURE;}
xil_printf("B Kisisi alinen r sayisini hashledi\r\n");
// 3) B Kisisi hash(r)'yi imzalayacak

//RSA kanal ayarlamasi
set_FIFOs_and_MODULE(RSA_CHANNEL);

// RSA modülüne veri gönderilip, çıkış alınıyor
Status = XAxiDma_RSA_SIGN_Transfer(SHA_HASH_ADDR_B, PRIVATE_KEY_ADDR_B,
N_B_ADDR_B, RSA_RX_ADDR_B, B_ID );

if (Status != XST_SUCCESS) {
    xil_printf("B kisisinin Hash İmzalamasi Basarisiz\r\n");
    return XST_FAILURE;
}
xil_printf("B Kisisi hash(r)'yi imzaladi\r\n");
```

Şekil 6.12 : Ana fonksiyonun devamı.



```c
int XAxiDma_RSA_DEC_SIGN_Transfer(u32 message_addr, u32 key_addr,
    u32 modulus_addr, u32 rx_buff_addr, u8 user_id )
{
    int Status;
    int Index;
    u8 *Message_Ptr;
    u8 *Key_Ptr;
    u8 *Modulus_Ptr;
    u8 *RxBufferPtr;
    u8 *TxBuffer_RSA_Ptr;

    Message_Ptr = (u8 *)message_addr;
    Key_Ptr = (u8 *)key_addr;
    Modulus_Ptr = (u8 *)modulus_addr;
    RxBufferPtr = (u8 *)rx_buff_addr;

    if(user_id==A_ID)
        TxBuffer_RSA_Ptr = (u8 *) RSA_TX_ADDR_A;
    else if(user_id==B_ID)
        TxBuffer_RSA_Ptr = (u8 *) RSA_TX_ADDR_B;

    for(Index = 0; Index < RSA_SEND_BYTE_LEN-256; Index++) {

        TxBuffer_RSA_Ptr[Index] =   Message_Ptr[Index];}

    for(Index = RSA_SEND_BYTE_LEN-256; Index < RSA_SEND_BYTE_LEN-128; Index++) {

        TxBuffer_RSA_Ptr[Index] = Key_Ptr[Index-128];}

    for(Index = RSA_SEND_BYTE_LEN-128; Index < RSA_SEND_BYTE_LEN; Index++) {

        TxBuffer_RSA_Ptr[Index] = Modulus_Ptr[Index-256];}
```

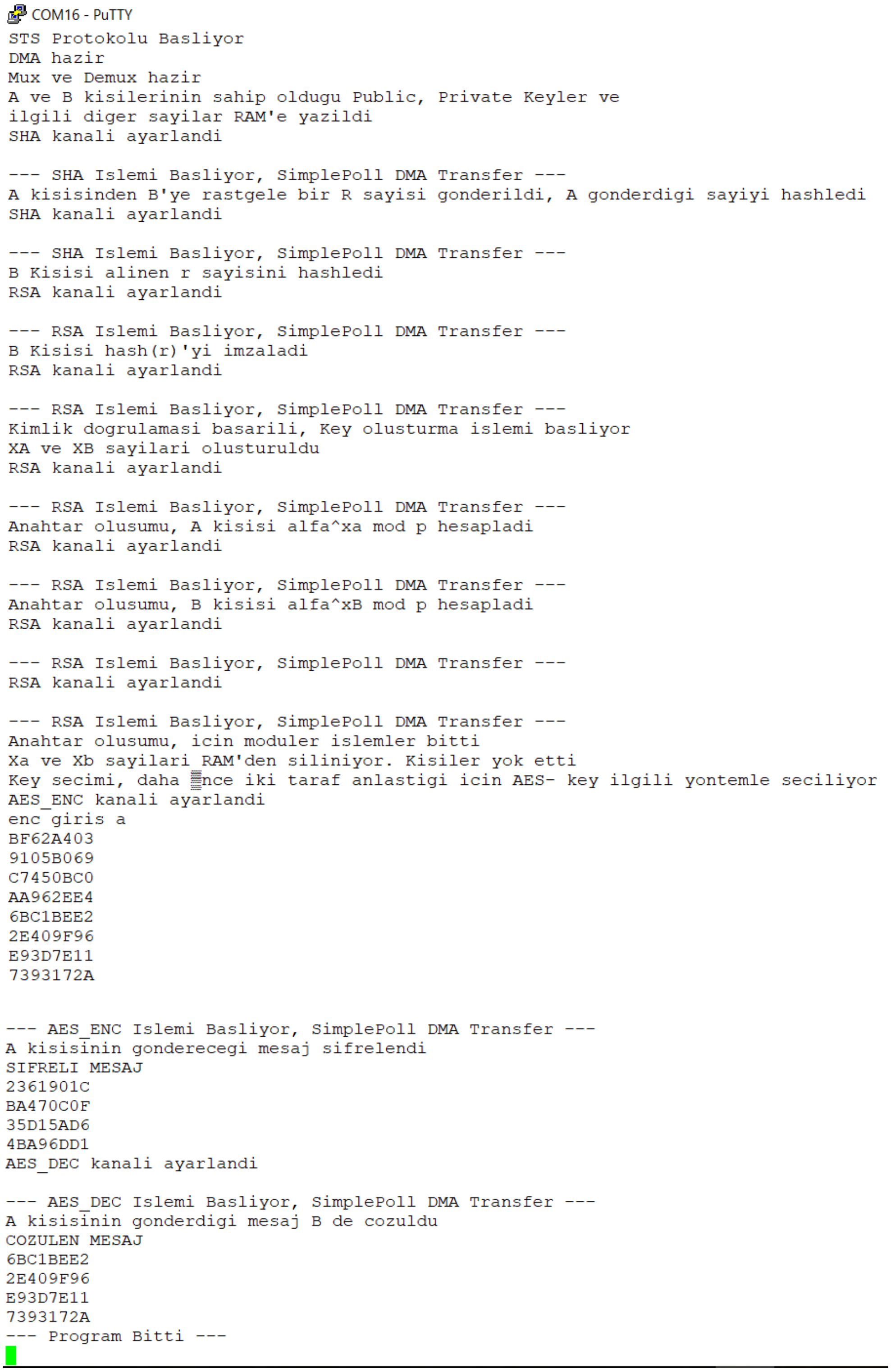
Şekil 6.13 : *XAxiDma_RSA_SIGN_Transfer* fonksiyonu.

A kişisinden B kişine yollanan r sayısı özetlenmesi amacıyla SHA-3 modülüne gönderilir. SHA-3 modülüne gönderim için modülün girişlerine ek olarak işlem yapan kişi belirtilir. Kişi belirtilmesinin sebebi işlem sonucunun yazılması gereken bellek bölümünün seçimidir. Tasarlanan fonksiyon ile DMA gerekli bilgiyi SHA-3 modülüne iletir ve işlem tamamlanıp modülün çıkışındaki FIFO dolduğunda veriyi alıp belleğe yazar. FIFO'ların meşgul olduğu zamanlarda ise fonksiyon sürekli olarak FIFO'ları kontrol eden bir döngü içinde kalır. SHA-3 modülünde özetlenen değer daha sonra RSA veri yolunun aktif hale getirilmesi ile haberleşme protokolü gereği B kişisi tarafından imzalanır ve bu değer A kişine yollanır. Bu imzalama işleminin girişinin 1024 bit olması gerekmesine rağmen SHA-3 modülünün çıkışının 512 bit olmasından dolayı kalan 512 bit 0 olarak atanır. Daha sonra A kişisi tarafından elde edilen özetleme değeri, B kişisi tarafından gönderilen özetleme değeri ile karşılaştırılır. Karşılaştırmanın başarısız olduğu durumda sistem hata mesajı vererek çalışmayı durdurur, aksi takdirde işlemin başarılı olduğu bilgisi verilir ve anahtar oluşturma işlemine başlanır.

Anahtar oluşturma işlemi XA ve XB değerlerinin oluşturulması ile başlar. *generate_and_write_Xa_Xb_RAM* isimli fonksiyon ile oluşturulan değerler bu uygulamaya özel olarak el ile girilmiş değerlerden oluşmaktadır. Çünkü bu tür veriler normal şartlar altında sayısal devreler ile değil de yazılım ile oluşturulmakta ve belleğe gönderilmektedir. Fonksiyonun görevi oluşturulan verileri ilgili bellek adresine yazmaktır.

Anahtar üretme işleminin başlatılması için haberleşme protokolünde belirtilen işlemler RSA modülü üzerinden gerçekleştirilir. Bu aşamadaki imzalama işlemleri daha önce gerçekleştirilen imzalama işlemleri ile aynı şekilde yapılır. Anahtar oluşturma işlemi tamamlandıktan sonra x_A ve x_B değerleri bellekten silinir. Bu silme işlemi veriye sahip olan ilgili kişiler tarafından yapılır. Oluşturulan anahtar 1024 bit uzunluğunda olduğu için bu bit dizisinin parçalanması gerekir. Tasarlanan sistemde oluşturulan 1024 bitlik bit dizisinin en önemsiz 128 biti seçilerek ayrıştırılır ve protokolde kullanılmak üzere AES modüllerinin kullanacağı anahtar değerine karşılık gelir. Daha sonra AES kanalı çoğullama çözücü tarafından aktif hale getirilir ve bellekten gelen mesaj değeri, oluşturulan anahtar ile şifrelenerek A kişisi tarafından B kişisine gönderilir. Bu gönderim sırasında mesaj boyutunun yazılmasının sebebi DMA protokolünden kaynaklanmaktadır. İşlem başarıyla tamamlandıktan sonra terminal ekranına şifrelenen mesaj yansıtılır. Mesajı alan B kişisi deşifreleme kanalı aktif hale getirilerek ürettiği aynı anahtar ile mesajı deşifre eder ve terminal ekranına yansıtır. Böylece haberleşme sistemi tamamlanmış olur. Sistemin sonuç ekranı Şekil 6.14'te görüldüğü gibidir.

Sonuçlar analiz edildiğinde bütün adımların doğru çalıştığı ve elde edilen verilerin doğru olduğu anlaşılmıştır.



```
STS Protokolu Basliyor
DMA hazir
Mux ve Demux hazir
A ve B kisilerinin sahip oldugu Public, Private Keyler ve
ilgili diger sayilar RAM'e yazildi
SHA kanali ayarlandi

--- SHA Islemi Basliyor, SimplePoll DMA Transfer ---
A kisisinden B'ye rastgele bir R sayisi gonderildi, A gonderdigi sayiyi hashledi
SHA kanali ayarlandi

--- SHA Islemi Basliyor, SimplePoll DMA Transfer ---
B Kisisi alinen r sayisini hashledi
RSA kanali ayarlandi

--- RSA Islemi Basliyor, SimplePoll DMA Transfer ---
B Kisisi hash(r)'yi imzaladi
RSA kanali ayarlandi

--- RSA Islemi Basliyor, SimplePoll DMA Transfer ---
Kimlik dogrulamasi basarili, Key olusturma islemi basliyor
XA ve XB sayilari olusturuldu
RSA kanali ayarlandi

--- RSA Islemi Basliyor, SimplePoll DMA Transfer ---
Anahtar olusumu, A kisisi alfa^xa mod p hesapladi
RSA kanali ayarlandi

--- RSA Islemi Basliyor, SimplePoll DMA Transfer ---
Anahtar olusumu, B kisisi alfa^xB mod p hesapladi
RSA kanali ayarlandi

--- RSA Islemi Basliyor, SimplePoll DMA Transfer ---
RSA kanali ayarlandi

--- RSA Islemi Basliyor, SimplePoll DMA Transfer ---
Anahtar olusumu, icin moduler islemler bitti
Xa ve Xb sayilari RAM'den siliniyor. Kisiler yok etti
Key secimi, daha ▒nce iki taraf anlastigi icin AES- key ilgili yontemle seciliyor
AES_ENC kanali ayarlandi
enc giris a
BF62A403
9105B069
C7450BC0
AA962EE4
6BC1BEE2
2E409F96
E93D7E11
7393172A


--- AES_ENC Islemi Basliyor, SimplePoll DMA Transfer ---
A kisisinin gonderecegi mesaj sifrelendi
SIFRELI MESAJ
2361901C
BA470C0F
35D15AD6
4BA96DD1
AES_DEC kanali ayarlandi

--- AES_DEC Islemi Basliyor, SimplePoll DMA Transfer ---
A kisisinin gonderdigi mesaj B de cozuldu
COZULEN MESAJ
6BC1BEE2
2E409F96
E93D7E11
7393172A
--- Program Bitti ---
```

Şekil 6.14 : Haberleşme protokolünün sonuç ekranı.

7. GERÇEKÇİ KISITLAR, SONUÇLAR VE ÖNERİLER

7.1 Çalışmanın Uygulama Alanı

Günümüzün gerekliliklerinden dolayı insan faktörünün olduğu her alanda, yapılan işlemlerin güvenliği, bilgilerin güvenilirliği ve kişisel bilgilerin korunması gibi konularda kriptoloji bilimi ve uygulamaları kendilerine oldukça büyük bir etki alanı bulmaktadır. Projede gerçekleştirilen sistemin kriptografik yapı taşları veri güvenliğinin sağlanması gereken her konuda temel teşkil etmektedir. Örnek verilecek olursa, kredi kartlarında ve kullanıldığı cihazlarda (bankamatik, POS cihazı vb.), internet bankacılığında, nesnelerin interneti uygulamalarında, kablolu veya kablosuz haberleşme sağlayan neredeyse bütün cihazlarda, internet ortamında yapılan ve güvenlik gerektiren her türlü işlemde farklı protokollere sahip kriptografi sistemleri kullanılır. Örneğin projede gerçeklenen kriptografi protokolü ve protokolün içerisinde bulunan algoritmalar, kablosuz bağlantı alanlarının güvenliği sağlamak gibi farklı alanlarda kullanılan bütün güvenlik sistemlerinin temelini oluşturmaktadır.

7.2 Gerçekçi Tasarım Kısıtları

7.2.1 Maliyet

Proje yürütülürken Vivado ve Vivado HLS ortamında tasarlanacak kriptografi protokolü sistemlerinin pratik olarak gerçeklenmesi için 1 adet FPGA Geliştirme Kartı kullanılmıştır. Kullanılan ZedBoard Zynq-7000 ARM/FPGA kırmık üstü sistem geliştirme kartının ücreti yaklaşık 3700 ₺ etmektedir. Haftada 4 gün, günde 4 saat, saatlik 20 ₺ üzerinden toplamda 30 hafta olmak üzere 9600 ₺ işçilik maliyeti vardır. Belirtilen bu maliyetler projenin yürütülmesi için gerekli olan toplam 13300 ₺'lik maliyeti oluşturmaktadır.

7.2.2 Standartlar

Projede yapılacak çalışmalar IEEE ve NIST standartlarına uygun olarak yürütülmüştür.

7.2.3 Sosyal, çevresel ve ekonomik etki

Günümüzde yaşadığımız toplum, bilgi toplumu olarak isimlendirilmektedir. Gerek devlet ve ordu nezdinde gerekse sosyal medyadaki kişisel veya toplumsal bilgiler paylaşılmaktadır. Bu bilgilerin, kötü amaçlı olsun veya olmasın üçüncü şahıslar tarafından elde edilebilme olasılığı bu bilgilerin korunması ve şifrelenmesi gerekliliğini ortaya çıkarır. Bu sağlıklı ve güvenli paylaşım ortamını sağlayacak ve kriptografi protokollerini gerçekleyecek donanımların tasarlanması büyük önem arz etmektedir. Bu donanımların hızlı ve etkili biçimde tasarlanarak performanslı biçimde çalıştırılmaları bilgi paylaşımını daha güvenli ve sağlıklı kılacaktır. Ayrıca donanım tasarlayacak mercilerin tasarım süresi ve maliyetlerinin en aza indirgenmesi, araştırma ve geliştirme faaliyetlerine daha fazla zaman ayrılabilmesine olanak tanıyacaktır

7.2.4 Sağlık ve güvenlik riskleri

Proje herhangi bir sağlık ve güvenlik riski içermemektedir.

7.3 Sonuçlar

Bu projede donanımlar arası güvenli iletişimi sağlayan kriptografi algoritması, yüksek seviyeli bir dil olan C programlama dilinde, Vivado HLS'in donanıma dönüştürme özellikleri kullanarak yazılmıştır. VHDL ile tasarlanan protokol ile Vivado HLS yardımıyla tasarlanan protokol performans açısından karşılaştırılmıştır. Projedeki kaygı güvenliği maksimum derecede sağlamak değil, sistemin temel yapı taşlarını oluşturmak ve oluştururken izlenebilecek farklı yolları analiz etmektir. Çalışma, güvenli haberleşme sistemleri üreten ve tasarlayan savunma sanayi, bankacılık ve telekomünikasyon gibi sektörlerdeki üreticilerin, sistem oluştururken izleyebileceği efektif tasarım yollarını sunmayı hedeflemiştir. Tasarımın daha üst seviyelerinde kalınarak (yazmaç transfer seviyesine inilmeden) düşük seviyede tasarım yapıldığında elde edilen performans kriterlerine ulaşılamasa da makul sistem gerçeklemeleri yapılabildiği görülmüştür. Yazılımsal ve donanımsal ortak tasarımın yapıldığı sistemlerde kullanılan arayüzlerin sisteme entegrasyonunun daha basit şekilde sağlanması ve bu vesileyle tasarım süresinin azaltılarak kolay hale getirilmesi ile algoritma tasarımına daha çok odaklanmaya fırsat sağlanmıştır.

7.4 Geleceğe Yönelik Öneriler

Yüksek seviyede tasarım yaparken esnekliğin ve donanım üzerindeki kontrolün daha fazla arttırılması ile birlikte performans bakımından günümüzdeki sistemlere kıyasla daha üstün sistemler gerçekleştirilebilir. Yüksek seviyeli tasarım ortamında tasarımın performansıyla ilgili yapılan analizler, alt seviyeli tasarım ortamında yapılan analizlere göre daha az kesinlik sağladığından dolayı yüksek seviyeli tasarım ortamında yapılan bu analizlerin geliştirilmesi tasarımcıya daha çok kolaylık sağlayacaktır. Alt seviyelerde tasarım yapmadan da aynı performansın yakalanabilmesi adına bu alanda çalışmaların devam etmesi gerekmektedir.

KAYNAKLAR

[1] **D. B. Gümüş**, “POWER EFFICIENT FPGA IMPLEMENTATION OF RSA ALGORITHM,” Y. L. Tezi, İstanbul Teknik Üniversitesi, Maslak, İST, 2010.

[2] **E. Homsirikamol and K. Gaj**, “Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study,” *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14),* 2014.

[3] **FIPS, P.,** 2001. 197, Advanced encryption standard (AES), National Institute of Standards and Technology.

[4] **FIPS, P.,** 2015. 202. SHA-3 Standard, National Institute of Standards and Technology.

[5] **H. Kayış,** “AES UYGULAMASI’NIN FPGA GERÇEKLEMELERİNE KARŞI GÜÇ ANALİZİ SALDIRISI,” Y. L. Tezi, İstanbul Teknik Üniversitesi, Maslak, İST, 2006.

[6] **H. Özdemir**, “FPGA Platformu İçin VIVADO HLS Tabanlı SURF Algoritmasının Gerçeklenmesi”, Akdeniz Üniversitesi, Ocak 2018

[7] **H. S. Jacinto, L. Daoud, and N. Rafla,** “High level synthesis using vivado HLS for optimizations of SHA-3” 08, 2017.

[8] **H. Wang, Z. Song, X. Niu and Q. Ding**, "Key generation research of RSA public cryptosystem and Matlab implement," *PROCEEDINGS OF 2013 International Conference on Sensor Network Security Technology and Privacy Communication System,* Nangang, 2013, pp. 125-129. doi: 10.1109/SNS-PCS.2013.6553849

[9] **K. E. Ahmed, M. M. Farag,** “Hardware/software co-design of a dynamically configurable SHA-3 System-on- Chip” 12, 2015.

[10] **M. A. Özkan,** “IMPLEMENTATION OF A LIGHTWEIGHT TRUSTED PLATFORM MODULE”, Y. L. Tezi, İstanbul Teknik Üniversitesi, Maslak, İST, 2014.

[11] **M. Shieh, J. Chen, H. Wu and W. Lin,** "A New Modular Exponentiation Architecture for Efficient Design of RSA Cryptosystem," in *IEEETransactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 9, pp. 1151-1161, Sept. 2008. doi: 10.1109/TVLSI.2008.2000524

[12] **N. Botros**, HDL with digital design. Dulles, Virginia: Mercury Learning and Information, 2015.

[13] **P. C. van Oorschot, M. J. Wiener** "Authentication and Authenticated Key Exchanges" 3,1992.

[14] **R.L. Rivest, A. Shamir and L. Adleman,** "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM 21, 2,* Feb., pp. 120-126, 1978.

[15] **Xilinx.** 2011. AXI Reference Guide, UG761 (v13.1) Mart 7, 2011

[16] **Xilinx.** 2016. AXI4-Lite IPIF v3.0 LogiCORE IP Product Guide, PG155 Nisan 6, 2016

[17] **Xilinx.** 2018. AXI DMA v7.1 LogiCORE IP Product Guide, PG021 Nisan 4, 2018

[18] **Xilinx.** 2018. Vivado Design Suite User Guide Getting Started, UG910 (v2018.2) Nisan 4, 2018

[19] **Xilinx.** 2018. Vivado Design Suite User Guide High-Level Synthesis, UG902 (v2018.2) Temmuz 2, 2

ÖZGEÇMİŞ

Ad-Soyad **:** Barış Bilgili

Doğum Tarihi ve Yeri **:** İstanbul, 11.10.1995

E-posta **:** bilgiliba@gmail.com

Eğitim Durumu (Kurum ve Yıl) Lise : Hüseyin Avni Sözen Anadolu Lisesi (2009-2014)

Lisans : İstanbul Teknik Üniversitesi – Elektronik ve Haberleşme Mühendisliği (2014-2019)

Ad-Soyad **:** Ceyhun Yamaneren

Doğum Tarihi ve Yeri **:** İstanbul, 17.08.1995

E-posta **:** cyamaneren@gmail.com

Eğitim Durumu (Kurum ve Yıl) Lise : Hüseyin Avni Sözen Anadolu Lisesi (2009-2014)

Lisans : İstanbul Teknik Üniversitesi – Elektronik ve Haberleşme Mühendisliği (2014-2019)

Ad-Soyad **:** Kubilay Vatansever

Doğum Tarihi ve Yeri **:** Baar/İsviçre, 02.05.1994

E-posta **:** vatanseverkubilay@gmail.com

Eğitim Durumu (Kurum ve Yıl) Lise : Hüseyin Avni Sözen Anadolu Lisesi (2009-2014)

Lisans : İstanbul Teknik Üniversitesi – Elektronik ve Haberleşme Mühendisliği (2014-2019)

Ad-Soyad **:** Umut Çoltu

Doğum Tarihi ve Yeri **:** İstanbul**,** 14.04.1997

E-posta **:** ucoltu@gmail.com

Eğitim Durumu (Kurum ve Yıl) Lise : Adnan Menderes Anadolu Lisesi (2011-2015)

Lisans : İstanbul Teknik Üniversitesi – Elektronik ve Haberleşme Mühendisliği (2015-2019)