ISTANBUL TECHNICAL UNIVERSTY

ELECTRICAL – ELECTRONICS ENGINEERING FACULTY

A SOFTWARE-HARDWARE IMPLEMENTATION OF A SECURED DATA COMMUNICATION PROTOCOL USING TEA ALGORITHM

BSc Thesis by

Arif GENCOSMANOGLU

Department: Electronics and Communication Engineering

Programme: Electronics Engineering

Supervisor: Assoc. Prof. Dr. Siddika Berna Ors Yalcin

MAY 2014

ACKNOWLAGEMENT

First of all, I really want to give my endless thanks to my supervisor, Assoc. Prof. Dr. Siddika Berna Ors Yalcin for sparing her precious time, giving advice and sharing her knowledge with me. It was really necessary to finish my thesis.

Also I would like to thank to Research Assis. Emre GÖNCÜ and my friend Ahmet JORGANXHI for giving their help when I mostly needed.

Last of all, I really appreciate all the support came from my family that they always stood behind my decisions.

MAY 2014 Arif GENCOSMANOGLU

INDEX

ABBREVIATIONS

DH	**:** Diffie Helman
EDK	**:** Embedded Development Kit
FPGA	**:** Field Programmable Gate Array
ISE	**:** Integrated Software Environment
LUT	**:** Look Up Table
RCA	**:** Ripple Carry Adder
RTL	**:** Register Transfer Level
SDK	**:** Software Development Kit
TEA	**:** Tiny Encryption Algorithm
UART	**:** Universal Asynchronous Receiver/Transmitter
UCF	**:** User Constraints File
USB	**:** Universal Serial Bus
XPS	**:** Xilinx Platform Studio

FIGURE LIST

A SOFTWARE-HARDWARE IMPLEMENTATION OF A SECURED DATA COMMUNICATION PROTOCOL USING TEA ALGORITHM

SUMMARY

By using Diffie–Hellman (DH) key exchange protocol a key is created by the sides which is about to communicate. In order to realize this method field programmable gate arrays (FPGA) are being used. By implementing DH, some mathematical operations like multiplication and summation are needed. So FPGAs are used to calculate these mathematical operations. Karatsuba multiplier and Ripple carry adder modules are used to calculate multiplication and summation to succeed DH key exchange method.

Two different FPGAs are used to communicate each other. After sending some information to each other several times, which is particularly explained in the thesis, both sides have a common key which is kept in secret. One of the sides uses this key to crypt a message with tiny encryption algorithm (TEA) and send it to the other side. The other side receives the message than decrypts the message with the same key. At the end, two sides are able to communicate confidentially on an unsecure line.

Applying this crypto procedure, all the algorithms are calculated with 128 bit numbers. Such size of numbers are used in these days because they are highly enough to be used in crypto algorithms.

GÜVENLİ BİR VERİ HABERLEŞME PROTOKOLÜNÜN TEA ALGORİTMASI İLE DONANIM-YAZILIM ORTAK OLARAK GERÇEKLENMESİ

ÖZET

Diffie–Hellman (DH) anahtar değişim protokolünden yararlanarak, haberleşmesi istenilen iki taraf arasında bir anahtar oluşturulmuştur. Bu yöntemi gerçeklemek için Sahada Programlanabilinir Kapı Dizileri (Field Programmable Gate Array, FPGA) kullanılmıştır. DH metodunu uygularken, toplama ve çarpma gibi bir takım matematiksel işlemlere ihtiyaç vardır. Bu işlemlerin hesaplanması için FPGA kullanılmıştır. Çarpma ve toplama işlemleri için, Karatsuba Çarpa algoritması ve Zincirleme Elde Toplayıcısı (Ripple Carry Adder, RCA) modülü kullanılarak DH anahtar değişim yöntemi başarı ile gerçekleştirilmiştir.

Haberleşmek adına iki farklı FPGA kullanılmıştır. Tezde detaylı olarak anlatılan şekilde iki taraf birbirlerine bir kaç kere bilgi yollayarak ortak bir anahtar elde ederler. Bir taraf bu anahtarı kullanarak yollamak istediği mesajı Küçük Kriptolama Algoritması (Tiny Encryption Algorithm, TEA) ile şifreleyerek öteki tarafa gönderir. Daha sonra öteki taraf bu mesajı aynı anahtar ve algoritma ile deşifre eder. Sonuç olarak, tehlikeli bir hat üzerinde güvenli bir şekilde bilgi aktarımı yapılmış olunur.

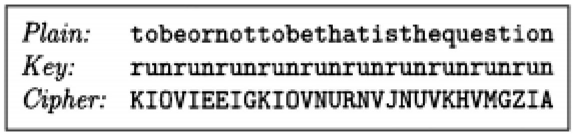
Kriptolama prosedürü uygulanırken, tüm algoritmaların hesaplanmasında 128 bitlik sayılar kullanılmıştır. Bu büyüklükteki sayılar, kriptolama algoritmalarının gerçeklemesinde yeterli olarak kabul edildiğinden dolayı, günümüzde de böyle kullanılmaktadır.

1. INTRODUCTION

Cryptography is the science of using mathematics to encrypt and decrypt data. Cryptography enables you to store sensitive information or transmit it across insecure networks (like the Internet) so that it cannot be read by anyone except the intended recipient. [16]

To overcome such problems by using crypto methods. Crypto algorithms allows to crypt a data that is to be sent and after it is sent, the receiver decrypts the received data to understand it. But first of all, both sides has to know how the data is encrypted. The art of protecting information by transforming it (encrypting it) into an unreadable format, called cipher text. Only those who possess a secret key can decipher (or decrypt) the message into plain text. Encrypted messages can sometimes be broken by cryptanalysis, also called code breaking, although modern cryptography techniques are virtually unbreakable.

In Figure 1, for example, we see “kiov” repeated after nine letters, and “nu” after six. Since three divides both six and nine, we might guess a keyword of three letters. It follows the chipper text letters one, four, seven and so and all enciphered under the same key letter; so we can use frequency analysis techniques to guess the most likely values of this letter, then repeat the process for the second and third letters of the key [1].



Plain:	tobeornottobethatisthequestion
Key:	runrunrunrunrunrunrunrunrunrun
Cipher:	KIOVIEEIGKIOVNURNVJNUVKHVMGZIA

Figure 1 Example of an encrypted data [1]

In this project encryption is made by TEA (Tiny Encryption Algorithm) and the key is produced by Diffie Helman (DH) key exchange method. The key that will be produced from DH key exchange method is 128 bit length therefore the TEA algorithm will use a key with length of 128 bit and encrypt a message using this key.

2. Foreknowledge

2.1. Diffie Helman Key Exchange Protocol

Diffie–Hellman (D-H) key exchange algorithm is a specific method of exchanging secret keys. It is one of the earliest practical examples of key exchange implemented within the field of cryptography. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher. [2]

DH key exchange protocol is used to share a key secretly from the observer who is listening to the communication line. For this key to be produced, some mathematical operations are need to be applied. These operations uses some non-secret variables that are known by everyone including the observer, and some secret variables that are only know by the producer.

This key exchange method uses constant numbers ‘**p**’ and ‘**g**’ where p is a prime number and g is a primitive root mod p. These two variables are known preliminary and are non-secret variables. There are also secret variables that is used. Each sides produce their own secret random variable ‘**a**’ and ‘**b**’ which are only known by themselves. The process begins after both sides to communicate produces their own secret random number.

All the variables are 128 bit long which is enough for a safe in communication. Explaining DH key exchange method, is easier by using an example.

1. Alice and Bob agree to use a prime number p = 23 and base g = 5.

2. Alice chooses a secret integer a = 6, then sends Bob (g^a mod p):

   $5^6 = 15625 \equiv 8 \pmod{23}$.

3. Bob chooses a secret integer b = 15, then sends Alice (g^b mod p):

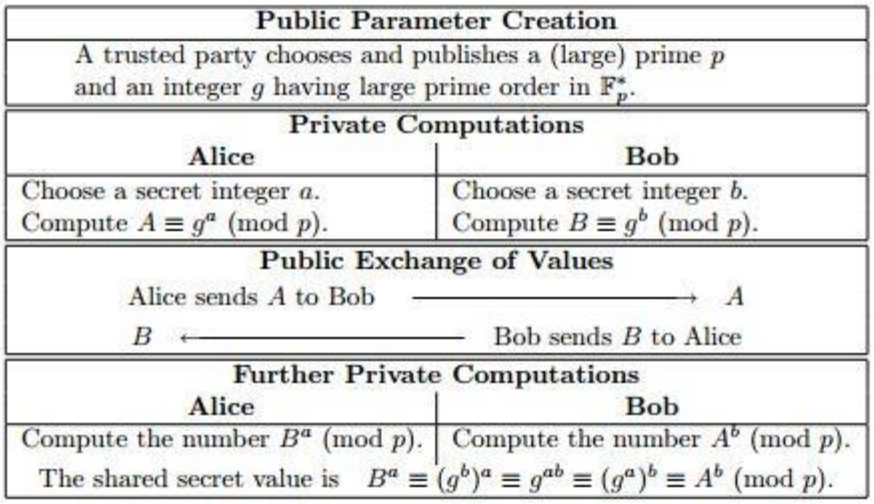
   $5^{15} = 30517578125 \equiv 19 \pmod{23}$.

4. Alice computes (g^b mod p)^a mod p:

   $19^6 = 47045881 \equiv 2 \pmod{23}$.

5. Bob computes (g^a mod p)^b mod p:
   $8^{15} = 35184372088832 \equiv 2 \pmod{23}$.

After the Diffie-Hellman key exchange both Alice and Bob have agreed on the key 2. [13]



Public Parameter Creation	
A trusted party chooses and publishes a (large) prime p and an integer g having large prime order in $\mathbb{F}_p^*$.	
Private Computations	
Alice	**Bob**
Choose a secret integer a. Compute $A \equiv g^a \pmod{p}$.	Choose a secret integer b. Compute $B \equiv g^b \pmod{p}$.
Public Exchange of Values	
Alice sends A to Bob $\longrightarrow$	A
B $\longleftarrow$	Bob sends B to Alice
Further Private Computations	
Alice	**Bob**
Compute the number $B^a \pmod{p}$.	Compute the number $A^b \pmod{p}$.
The shared secret value is $B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \pmod{p}$.	

Figure 2 Diffie Helman Key Exchange Protocol [12]

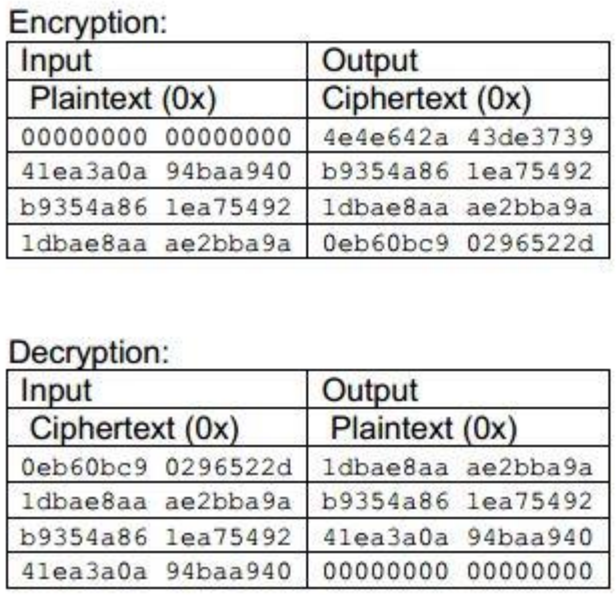
Both A and B have agreed on the same value, because $(g^a)^b$ and $(g^b)^a \bmod p$ are equal. The essential point is only a, b, and $(g^{ab} \bmod p = g^{ba} \bmod p)$ are kept as secret whereas all the other variables – p, g, $g^a \bmod p$, and $g^b \bmod p$ – are sent in the clear. Once side A and B compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.

2.2. Tiny Encription Algorithm (TEA)

The TEA (Tiny Encryption Algorithm) is a symmetric (private) key encryption algorithm created by David Wheeler and Roger Needham of Cambridge University and published in 1994. It was designed for simplicity and performance, while seeking an encryption strength on par with more complicated and resource-intensive algorithms such as DES (Data Encryption Standard). [14]

TEA uses two 32-bit unsigned integers and a 128-bit key. Also, it has a simple key schedule, mixing all of the key material in exactly the same way for each cycle. After the key is obtained from Diffie Helman key exchange protocol, it will be used to encrypt and decrypt a message.

The message entering to TEA encryption module is called as "**plaintext**". After plaintext is encrypted by the key, module produces an output which is called as "**chipertext**". In order to regain encrypted plaintext back, chipertext is given as input to TEA decryption module.



Encryption:

Input	Output
Plaintext (0x)	Ciphertext (0x)
00000000 00000000	4e4e642a 43de3739
41ea3a0a 94baa940	b9354a86 1ea75492
b9354a86 1ea75492	1dbae8aa ae2bba9a
1dbae8aa ae2bba9a	0eb60bc9 0296522d

Decryption:

Input	Output
Ciphertext (0x)	Plaintext (0x)
0eb60bc9 0296522d	1dbae8aa ae2bba9a
1dbae8aa ae2bba9a	b9354a86 1ea75492
b9354a86 1ea75492	41ea3a0a 94baa940
41ea3a0a 94baa940	00000000 00000000

Figure 3 TEA module input output example [9]

The simple conclusions (and answers to my opening questions) are: yes, TEA/XTEA is easy to implement, fast, efficient, and cryptographically strong. When implemented and used correctly, TEA can be an excellent choice, particularly for encrypting and decrypting small, short-lived data in resource constraint devices. [14]

2.3. Xilinx XPS

Xilinx is a company whom aids us to use Spartan - FPGAs. Not only FPGAs, this company supports us with several softwares which can be used to program FPGA or to configure and control its unique microcontroller Microblaze that takes part inside FPGA. The interface softwares that are used in the thesis project are; Xilinx Integrated Software Environment (ISE), Xilinx Embedded Development Kit (EDK) and Xilinx Software Development Kit (SDK)

2.3.1. Field Programmable Gate Array

A Field programmable gate array (FPGA) is a logic device that contains a two-dimensional array of generic logic cells and programmable switches. The conceptual structure of an FPGA device is shown in Figure 4. A logic cell can be configured (i.e., programmed) to perform a simple function, and a programmable switch can be customized to provide interconnections among the logic cells. A custom design can be implemented by specifying the function of each logic cell and selectively setting the connection of each programmable switch. Once the design and synthesis is completed, we can use a simple adaptor cable to download the desired logic cell and switch configuration to the FPGA device and obtain the custom circuit. Since this process can be done "in the field" rather than "in a fabrication facility (fab)," the device is known as field programmable. [3]

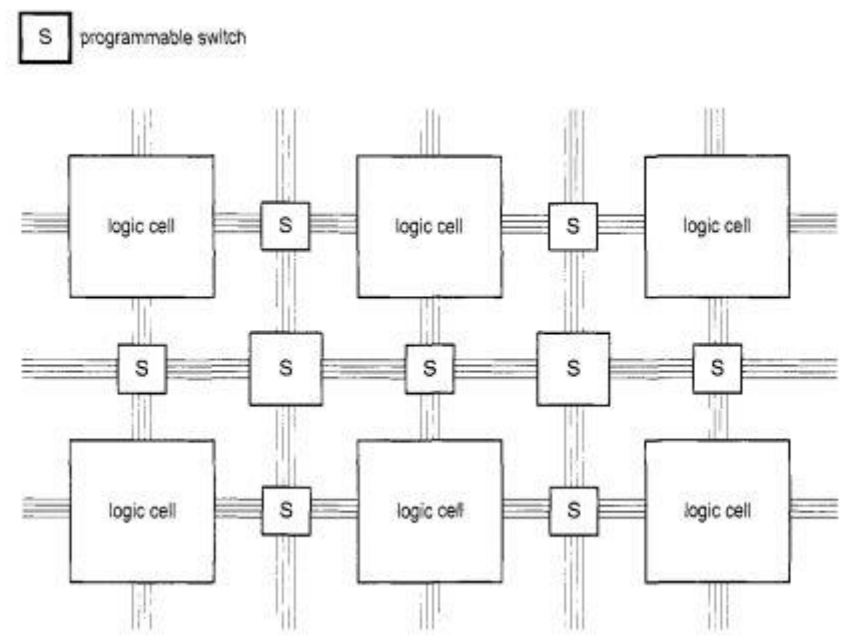


Figure 4 Logic celles inside FPGA [4]

In general, a logic block consists of a few logical cells. A typical cell consists of a 4-input LUT, a Full adder (FA) and a D-type flip-flop, as shown below. The LUTs are in this figure split into two 3-input LUTs. In normal mode those are combined into a 4-input LUT through the left mux. In arithmetic mode, their outputs are fed to the FA. The selection of mode is programmed into the middle multiplexer. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In practice, entire or parts of the FA are put as functions into the LUTs in order to save space. [4]

FPGAs are capable with parallel programming, so they can be used to build fast systems. Additionally, because microprocessors are logic circuits they can be programmed inside FPGAs according to the needs of the user. Therefore in a single integrated circuit as a control unit, it is possible to configure the hardware particularly for the user and the processor. FPGAs are fast also because all the system is on the same place, which reduces the delay caused by the connections. Because of all these features FPGA offers, it is decided to be used in this project.

2.3.1.1. Xilinx Spartan-6 LX45 FPGA

The project aims to send an encrypted message by a FPGA to another FPGA. Xilinx FPGA Spartan-6 LX45 is used as a transmitter and also as the receiver. This FPGA includes 4, LUTs with 6 inputs and 6,822 slices which has 8 flip-flops inside one of it. Also, the clock frequency can speed up to 500 MHz. [5]

2.3.1.2. Verilog Hardware Description Language

The modules that are used for the DH key exchange method and TEA algorithm has been designed in Xilinx ISE by using Verilog hardware description language. The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development. Like C, Verilog is case-sensitive and has a basic preprocessor. Its operator precedence is compatible with C. Because of these properties of Verilog, developers mostly do prefer to use this language.

2.3.1.3. Xilinx ISE Environment

Integrated Software Environment (ISE) is an interface software program that is developed by Xilinx Company. This software serves to configure Xilinx FPGAs. ISE has couple of properties which provide ease when designing a digital system. Hardware description languages can be used to define logic systems whereas can provide schematic information which certainly comes in handy. Also ISE gives the opportunity to test the designed modules on the test-bench simulation, moreover it is possible to see the delay based on the wires from place and root simulation. Last of all ISE is able to give information about space consumption and FPGA layout vision.

However these designed modules are not enough for the process to continue. Microblaze, which is a soft processor, is used to assure all the controls between these modules.

2.3.2. Microblaze Processor

Microblaze processor, is produced by programming the proper blocks on a FPGA. This processor is controlled using FPGA by software and it is used in embedded systems. Microblaze, gives the opportunity to choose peripheral units, memory and interface properties on a single FPGA. This properties yields for the user to build a flexible embedded system.

The Microblaze soft core processor is highly configurable, allowing you to select a specific set of features required by your design.

The fixed feature set of the processor includes:

• Thirty-two 32-bit general purpose registers

• 32-bit instruction word with three operands and two addressing modes

• 32-bit address bus

• Single issue pipeline

In addition to these fixed features, the MicroBlaze processor is parameterized to allow selective enabling of additional functionality. Older (deprecated) versions of MicroBlaze support a subset of the optional features described in this manual. Only the latest (preferred) version of MicroBlaze (v7.10) supports all options. [7]

2.3.2.1. Xilinx EDK Environment

Microblaze which is explained in the previous topic, is part of the Embedded Processor Development Kit (EDK). This platform is used when a microprocessor based digital system is aimed to build. This platform has some utilities which provides convenience for the user. These utilities are adjusting systems addressing, defining communication protocols and setting the hardware connections. Thus the user can be focused on designing the hardware and software.

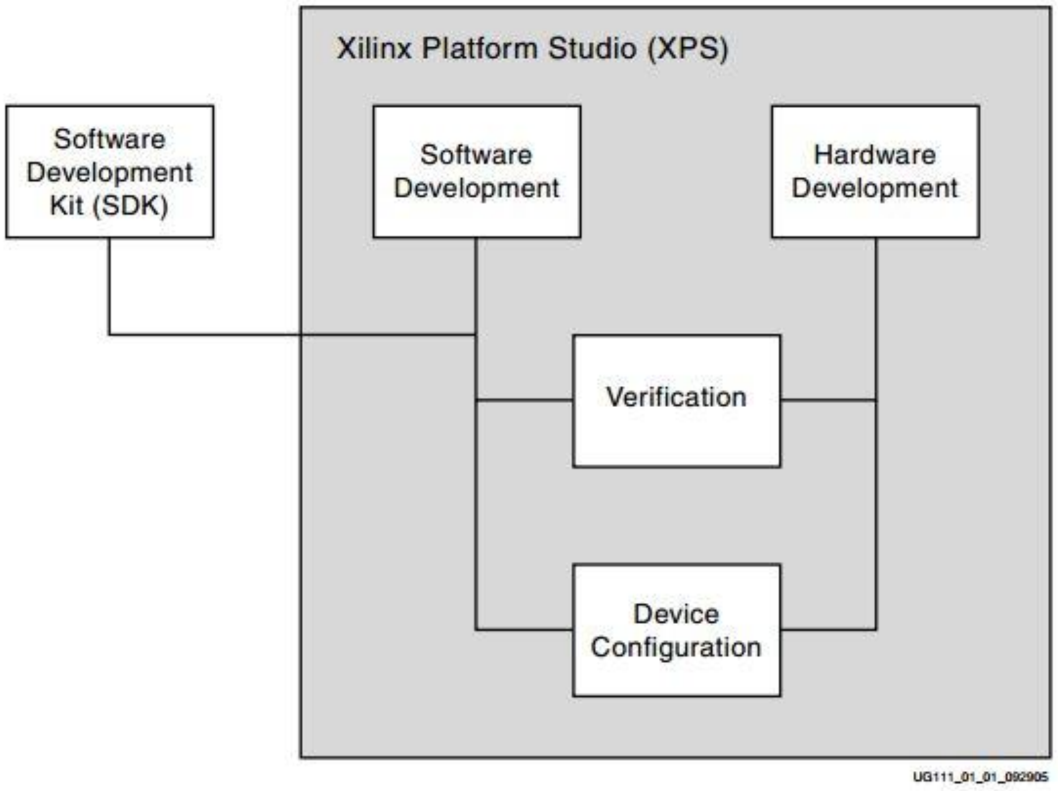


Figure 5 Basic Embedded Design Process Flow [8]

EDK includes:

• The Xilinx Platform Studio (XPS) Interface

• The Embedded System Tools suite

• Embedded processing Intellectual Property (IP) cores such as processors (also called pcores) and peripherals

• The Platform Studio SDK (Software Development Kit), based on the Eclipse open source framework, which you can use (optionally) to develop your embedded software application. [8]

2.3.2.2. Xilinx SDK Environment

Software Development Kit (SDK) takes part in EDK and is an interface where the processor based digital system software design is done. On XPS's previous versions SDK was involved in XPS. However after version ISE 13.0 Xilinx decided to spate SDK from XPS, which can be seen from Figure 5. This means XPS is used on purpose of designing a systems hardware section whereas SDK is only responsible for the software design section. EDK creates the necessary libraries that belongs to user logic hardware and peripheral units. Inside SDK, these created libraries are defined and used in order to design the desired system. These libraries are used to control Microblaze processor as it is needed.

SDK features include:

- Feature-rich C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic Makefile generation
- Error navigation
- Well-integrated environment for seamless debugging and profiling of embedded targets
- Source code version control. [6]

3. IMPLEMENTED PROTOCOLS AND ALGORITHMS

All the following protocols and algorithms are implemented on Xilinx Spartan 6 FPGA. All the modules are designed as state machines which means they all have a clock input. In order to work together all the modules have a reset and start input also a done flag output. By using these input and outputs, all the modules have been able to work together.

3.2. Implementation of DH Key Exchange Protocol

As it is explained above, DH key exchange protocol requires some mathematical operations to be done. These operations are simply multiplication and summation. The key exchange is fundamentally made by calculating A^B mod C which corresponds to the result of the square and multiply algorithm.

Both FPGAs are set to calculate A^B mod C using their own secret variable and a constant 128 bit number. After they make this calculation they send the result to each other. Finally both the FPGAs calculates the same operation again using their secret variable and the result came from the other FPGA. Since $(g^a)^b$ and $(g^b)^a$ are equal, the final result will be equal on both FPGAs and will be used as a key to crypt a message. If it is explained step by step;

1st step: FPGA 1 calculates X ^ a mod P using square and multiply algorithm. Here a is FPGA 1's secret variable

2nd step: FPGA 2 calculates X ^ b mod P using square and multiply algorithm. Here b is FPGA 2's secret variable.

3rd step: FPGA 1 sends the result **R1,** to FPGA 2 using UART protocol.

4th step: FPGA 2 sends the result **R2**, to FPGA 1 using UART protocol.

5th step: FPGA 1 calculates R1 ^ a mod P and obtains the key, K.

6th step: FPGA 2 calculates R2 ^ b mod P and obtains the same key, K.

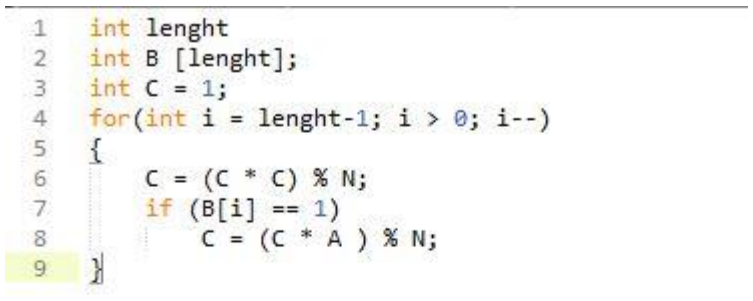
The produced key is used to encrypt and decrypt a message by Tiny Encryption Algorithm. This message will be encrypted on FPGA 1. After being sent to the other side the encrypted message will be decrypted at FPGA 2.

All the steps above would be controlled by using C code on SDK and will be explained in detail on "**Microblaze – Hardware Relationship**" topic.

3.3. Square and Multiply Algorithm

This module is designed to calculate the operation A^B mod C. In mathematics and computer programming, exponentiation by squaring is a general method for fast computation of large positive integer powers of a number, or, more generally of an element of a ring, like a polynomial or a square matrix. Some variants are commonly referred to as square-and-multiply algorithms or binary exponentiation. These can be of quite general use, for example in modular arithmetic or powering of matrices. This method is also used for exponentiation in groups. For groups for which additive notation is commonly used, like elliptic curves used in cryptography, this method is also referred to as double-and-add.

This algorithm for square and multiply written in C language can be seen at the figure below:



```
int lenght
int B [lenght];
int C = 1;
for(int i = lenght-1; i > 0; i--)
{
    C = (C * C) % N;
    if (B[i] == 1)
        C = (C * A ) % N;
}
```

Figure 6 Square and Multiply C code

As it is seen from the algorithm square and multiply uses some fundamental operations. These are multiplication and taking mod operation. For these two operations some extra modules are designed separately. Karatsuba multiplication algorithm is designed for the multiplication operation and it is being called by the top module (square and multiply module) when multiplication is needed.

Taking mod is being computed by making subtraction. A mod B is calculated by subtraction B from A, until A is smaller than B. Every single subtraction is made on one clock cycle. Therefore obtaining the result from the mod operation depends on the gap between A and B. If A is very large than B it may take too long to reach the result. However this will not cause a problem on calculation A ^ B mod C because C will be chosen as 128 bit length constant number so, the maximum calculation time will be made on 128 clock cycles.

To make the subtraction, mod operation calls a subtraction module which is designed by using Ripple Carry Adder (RCA) module. RCA is used for the reason of gain from space. This may cause reaching to the result in a longer state but gain from space was more important that will be explained on the topic OPTIMIZATION in detail.

RTL schematic which is obtained from the implemented square and multiply module is given as below:

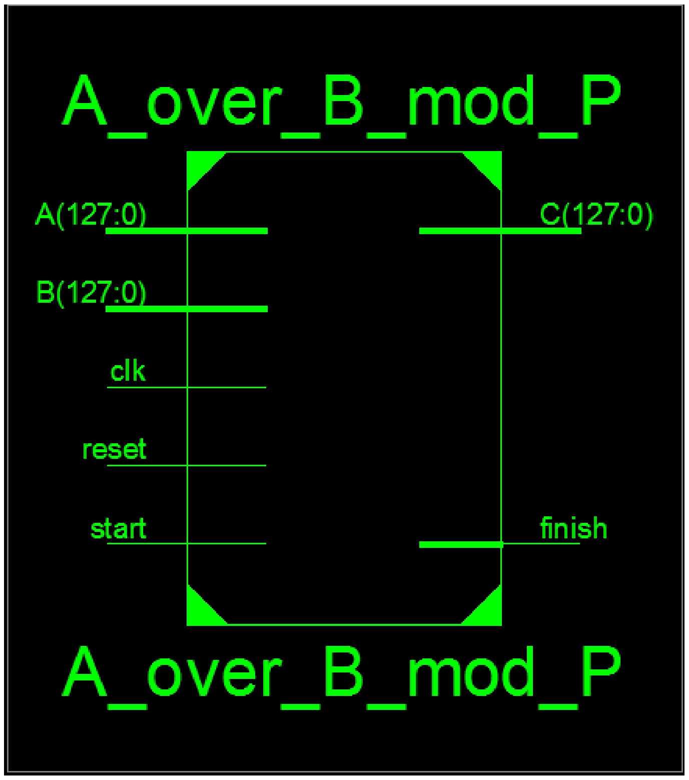


Figure 7 Square and Multiply modüle RTL schematic

As its seen from Figure 7, the module has a start and a reset bit as input and a done bit flag as output to work with other modules in harmony.

On the first state, the initial conditions are set and the module waits for the start bit to arrive. The module starts after the start bit is logic 1 and gives both of the inputs for multiplication as C, in order to compute C^2. When the calculation is done the multiplication will return the done flag which will lead the top module to the next state. After all the multiplication calculation are complete the module have to be reset to repeat the compute when the inputs are changed. On the next state the mod operation is made which the final result will be C^2 mod P. As the program flows the calculation of A^B mod P is made by changing the input of the multiplication and resetting it until it reaches to the end. The module finishes when the calculation counter becomes 0 as the counter initial value is bit size of B (128).

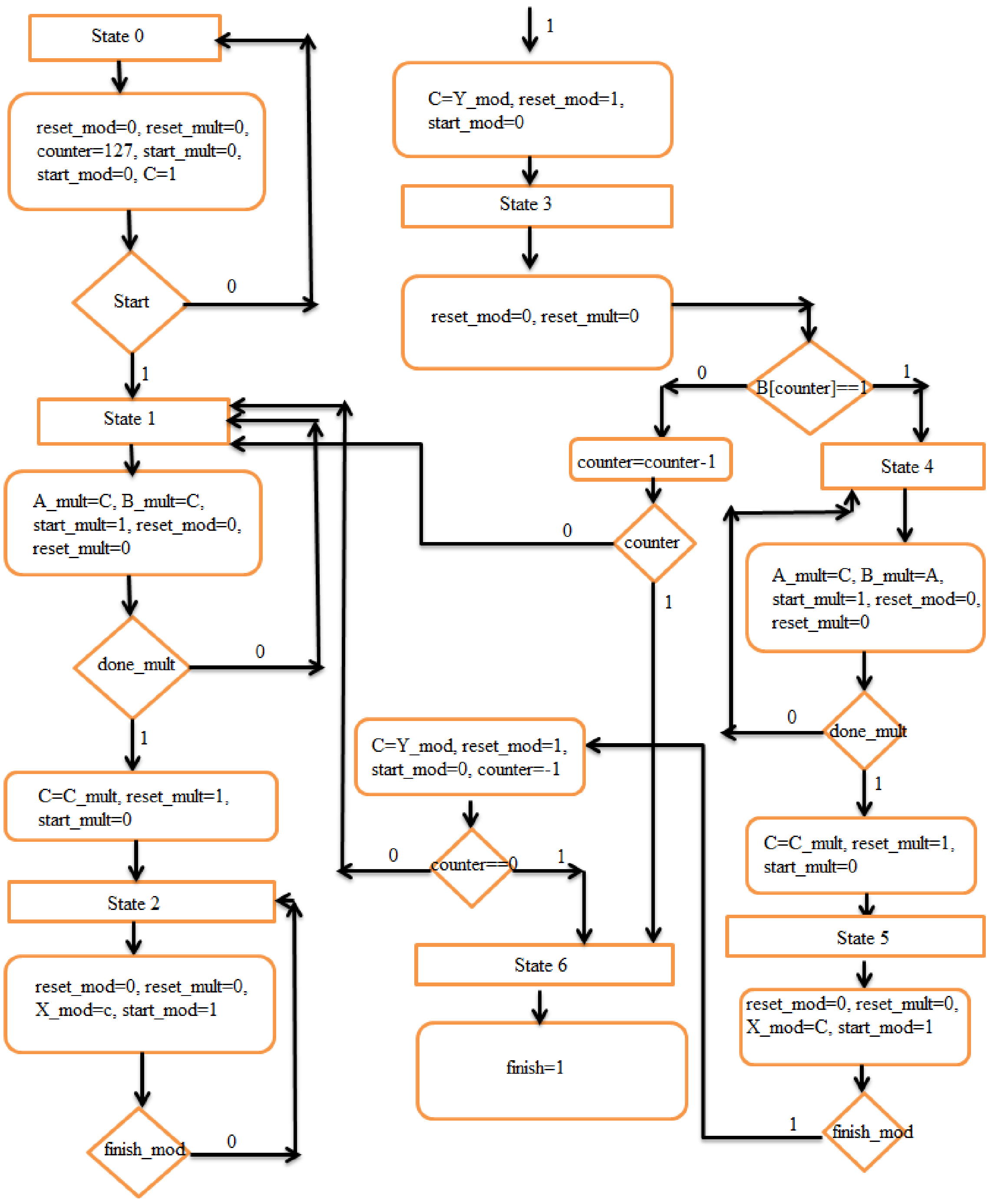

State 0
reset_mod=0, reset_mult=0, counter=127, start_mult=0, start_mod=0, C=1
Start
0
1
State 1
A_mult=C, B_mult=C, start_mult=1, reset_mod=0, reset_mult=0
done_mult
0
1
C=C_mult, reset_mult=1, start_mult=0
State 2
reset_mod=0, reset_mult=0, X_mod=c, start_mod=1
finish_mod
0
1
C=Y_mod, reset_mod=1, start_mod=0
State 3
reset_mod=0, reset_mult=0
B[counter]==1
0
1
counter=counter-1
counter
0
1
State 4
A_mult=C, B_mult=A, start_mult=1, reset_mod=0, reset_mult=0
done_mult
0
1
C=C_mult, reset_mult=1, start_mult=0
State 5
reset_mod=0, reset_mult=0, X_mod=C, start_mod=1
finish_mod
1
C=Y_mod, reset_mod=1, start_mod=0, counter=-1
counter==0
0
1
State 6
finish=1

Figure 8 Square and Multiplier asm diagram

Square and multiply module test-bench simulation is made as below;

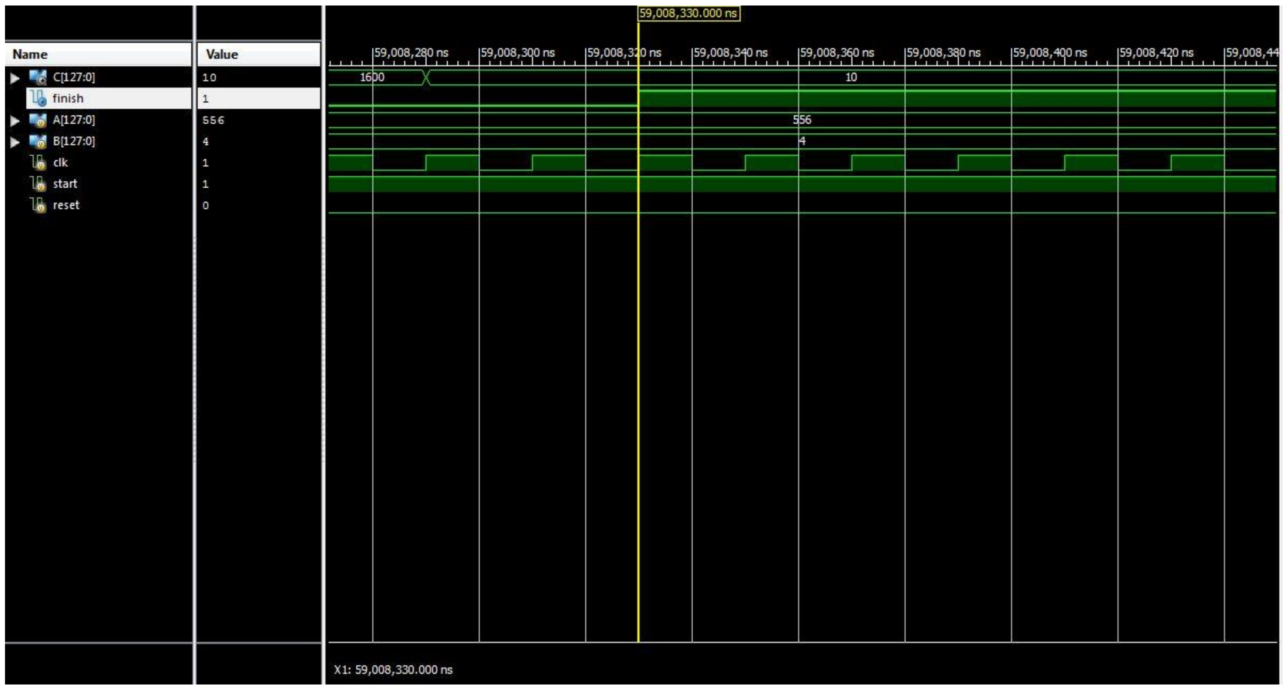


Figure 9 Square and Multiply module test-bench simulation image

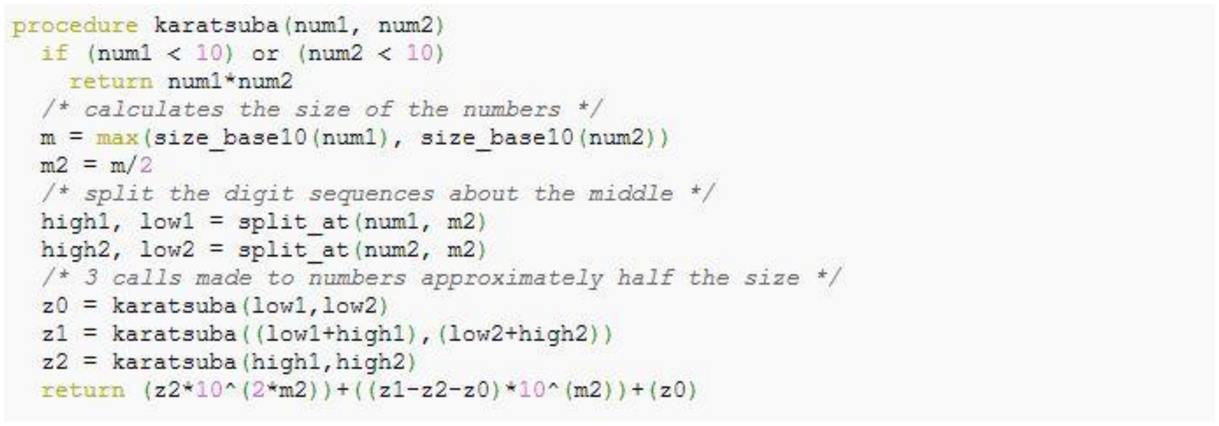
Module’s calculation speed depends on the number of logic 1’s that the second input has. This is because Karatsuba multiplier is called on every B input bit of logic 1 as it can be seen from the state diagram. For this simulation the time passed to reach the result is 59ms whereas the clock period is 20 ns. Therefore this simulation needs 59,000,000 / 20 = **2,950,000** clock cycles for the finish bit to rise.

3.4. Karatsuba Algorithm

This module is designed for the multiplication operation that is used in square and multiply module. The input bit length is 128 bit whereas the output bit length is 256 bit. Karatsuba multiplication has the advangtage to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts.

To calculate an input size of 128 bit, Karatsuba is divided in to smaller modules which means 64, 32, 16 and 8 bit length Karatsuba modules are also designed to perform the calculation properly.

Psuedo algoritm for Karatsuba can be seen at the figure below:



```
procedure karatsuba(num1, num2)
  if (num1 < 10) or (num2 < 10)
    return num1*num2
  /* calculates the size of the numbers */
  m = max(size_base10(num1), size_base10(num2))
  m2 = m/2
  /* split the digit sequences about the middle */
  high1, low1 = split_at(num1, m2)
  high2, low2 = split_at(num2, m2)
  /* 3 calls made to numbers approximately half the size */
  z0 = karatsuba(low1,low2)
  z1 = karatsuba((low1+high1),(low2+high2))
  z2 = karatsuba(high1,high2)
  return (z2*10^(2*m2))+((z1-z2-z0)*10^(m2))+(z0)
```

Figure 10 Psuedo code for Karatsuba Multiplier

After designing the module in verilog language on Xilinx ISE, it is implemented and the RTL shcmatic of the module can be seen on the following figure:

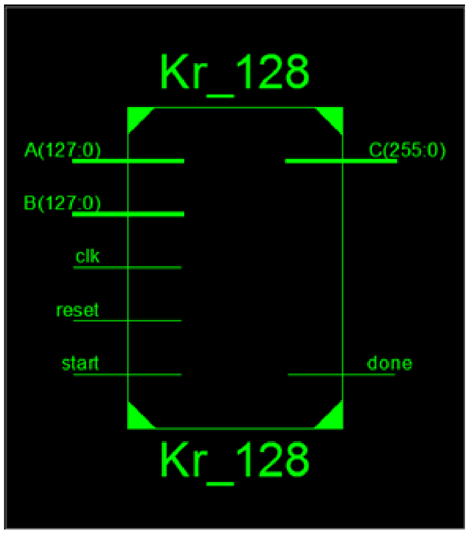


Figure 11 Karatsuba multiplier module RTL schematic

Simply expressing the Karatsuba verilog module, at the beginning the module waits for the start bit to become logic 1. After the start bit comes, the code flows through asm diagram which can be found on below;

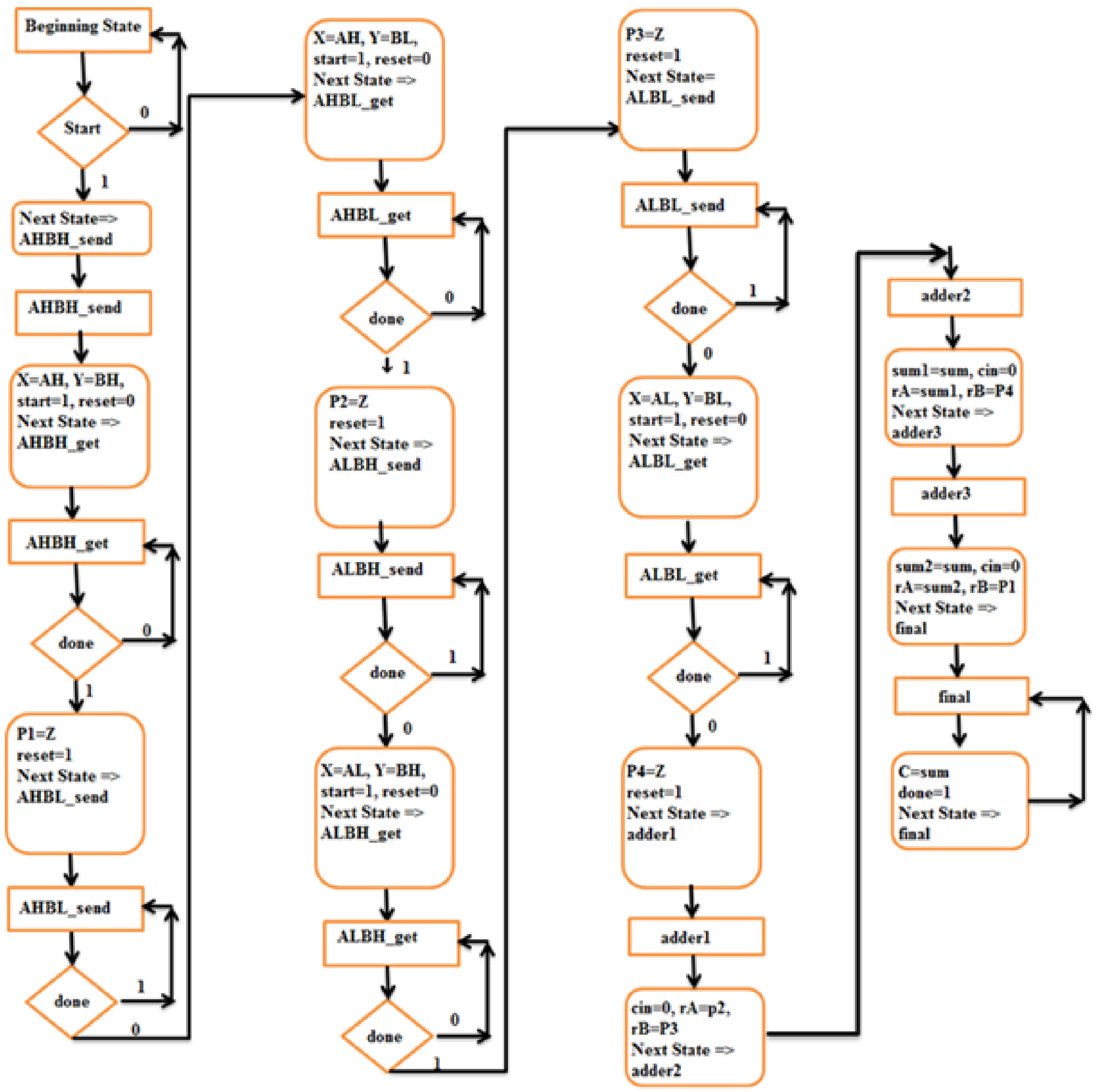

Beginning State
Start
0
1
Next State=> AHBH_send
AHBH_send
X=AH, Y=BH, start=1, reset=0 Next State => AHBH_get
AHBH_get
done
0
1
P1=Z reset=1 Next State => AHBL_send
AHBL_send
done
1
0
X=AH, Y=BL, start=1, reset=0 Next State => AHBL_get
AHBL_get
done
0
1
P2=Z reset=1 Next State => ALBH_send
ALBH_send
done
1
0
X=AL, Y=BH, start=1, reset=0 Next State => ALBH_get
ALBH_get
done
0
1
P3=Z reset=1 Next State= ALBL_send
ALBL_send
done
1
0
X=AL, Y=BL, start=1, reset=0 Next State => ALBL_get
ALBL_get
done
1
0
P4=Z reset=1 Next State => adder1
adder1
cin=0, rA=p2, rB=P3 Next State => adder2
adder2
sum1=sum, cin=0 rA=sum1, rB=P4 Next State => adder3
adder3
sum2=sum, cin=0 rA=sum2, rB=P1 Next State => final
final
C=sum done=1 Next State => final

Figure 12 Karatsuba multiplier asm flow chart [15]

As it appears, after the calculation is over the done flag is set to logic 1. So that it can be used synchronously with the other modules. Because of square and multiply module is calling Karatsuba module several times, its reset button is used to make the calculation all over again.

Lastly, Karatsuba needs to use summation to compute the output. As it is mentioned on square and multiply module, the summation is calculated by using RCA summation module.

Karatsuba module test-bench simulation is made as below;

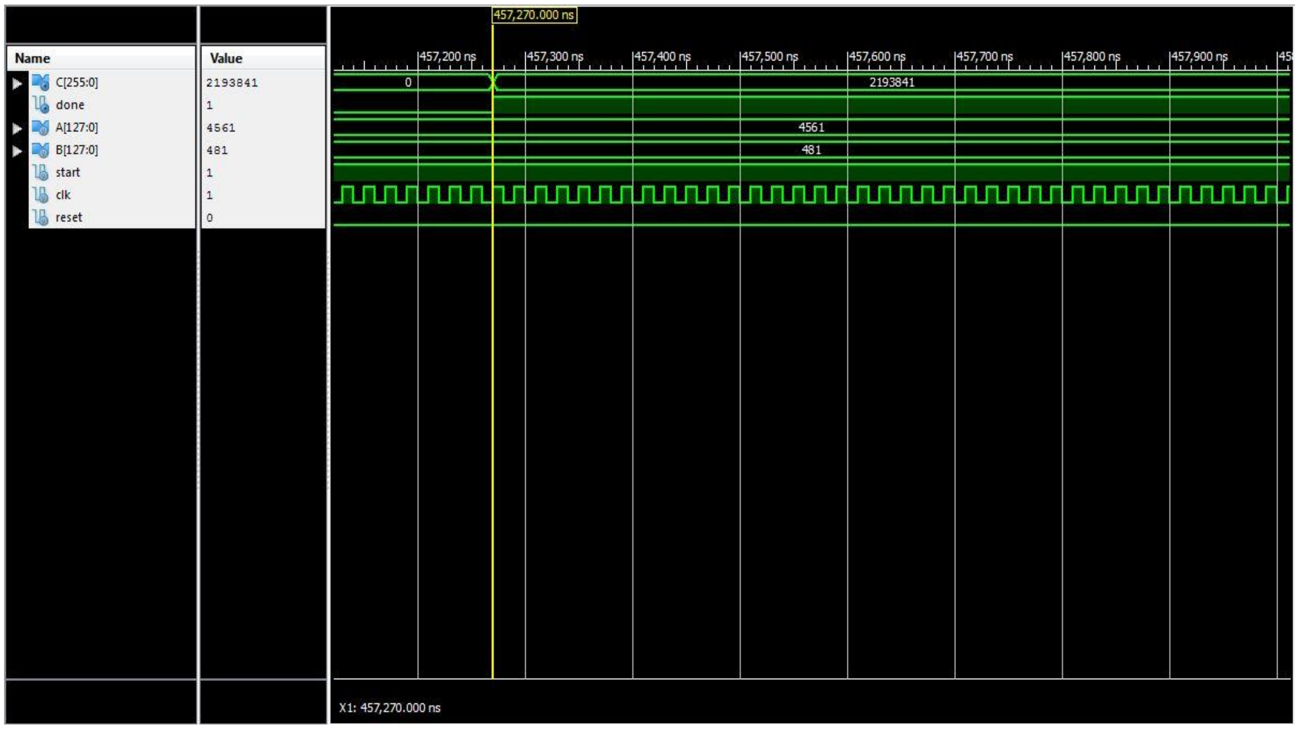
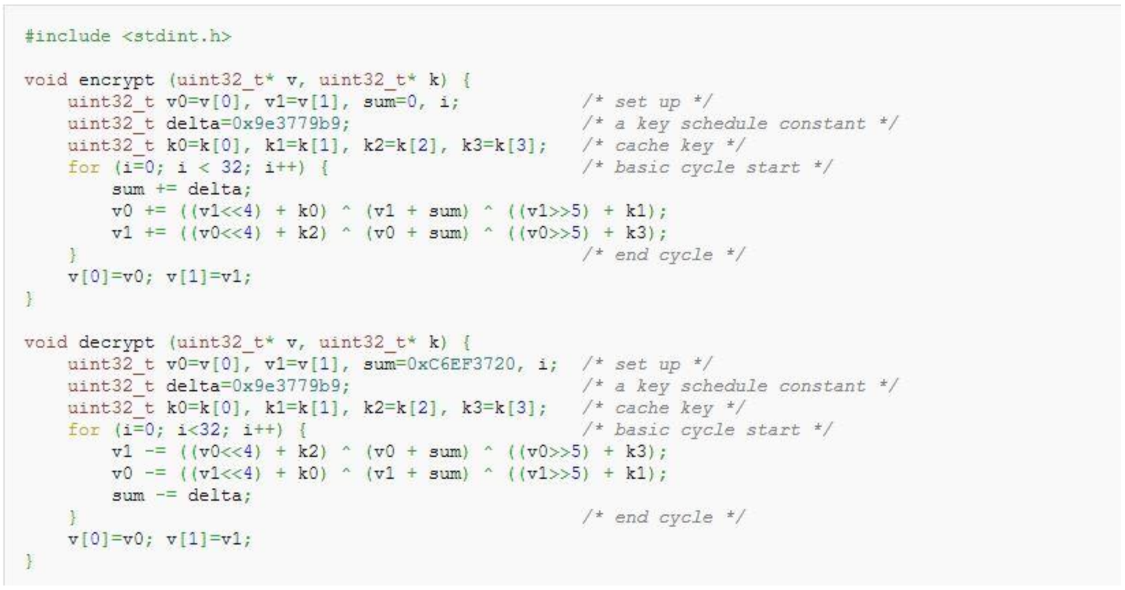


Figure 13 Karatsuba Multiplier test-benc simulation image

3.5. Implementation TEA Cryptography Algorithm

Following module is an adaptation of the reference encryption and decryption routines in C, released into the public domain by David Wheeler and Roger Needham:



```c
#include <stdint.h>

void encrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0, i;           /* set up */
    uint32_t delta=0x9e3779b9;                     /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i < 32; i++) {                       /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}

void decrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0xC6EF3720, i;  /* set up */
    uint32_t delta=0x9e3779b9;                     /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i<32; i++) {                         /* basic cycle start */
        v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        sum -= delta;
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

Figure 14 Tiny Encrytion Algorithm written in C code

As it can be seen from the c algorithm, TEA module has two 32 bit length inputs. One of them is for the message to be sent and the other one is the key gathered from DH key exchange method. The module is asynchronous so that as soon as the input changes, the output is immediately acquired.

The following notation is necessary for understanding TEA algorithm.

Shift Operator: The logical shift of x by y bits is denoted by x << y. The logical right shift of x by y bits is denoted by x >> y.

Rotation Operator: A left rotation of x by y bits is denoted by x <<< y. A right rotation of x by y bits is denoted by x >>> y.

Exclusive-OR: The operation of addition of n-tuples over the fields denoted by $x \oplus y$.

Integer Addition: The operation of integer addition modulo is denoted by $x \boxplus y$.

The flow chard of TEA using these operators can be seen in the figure below:

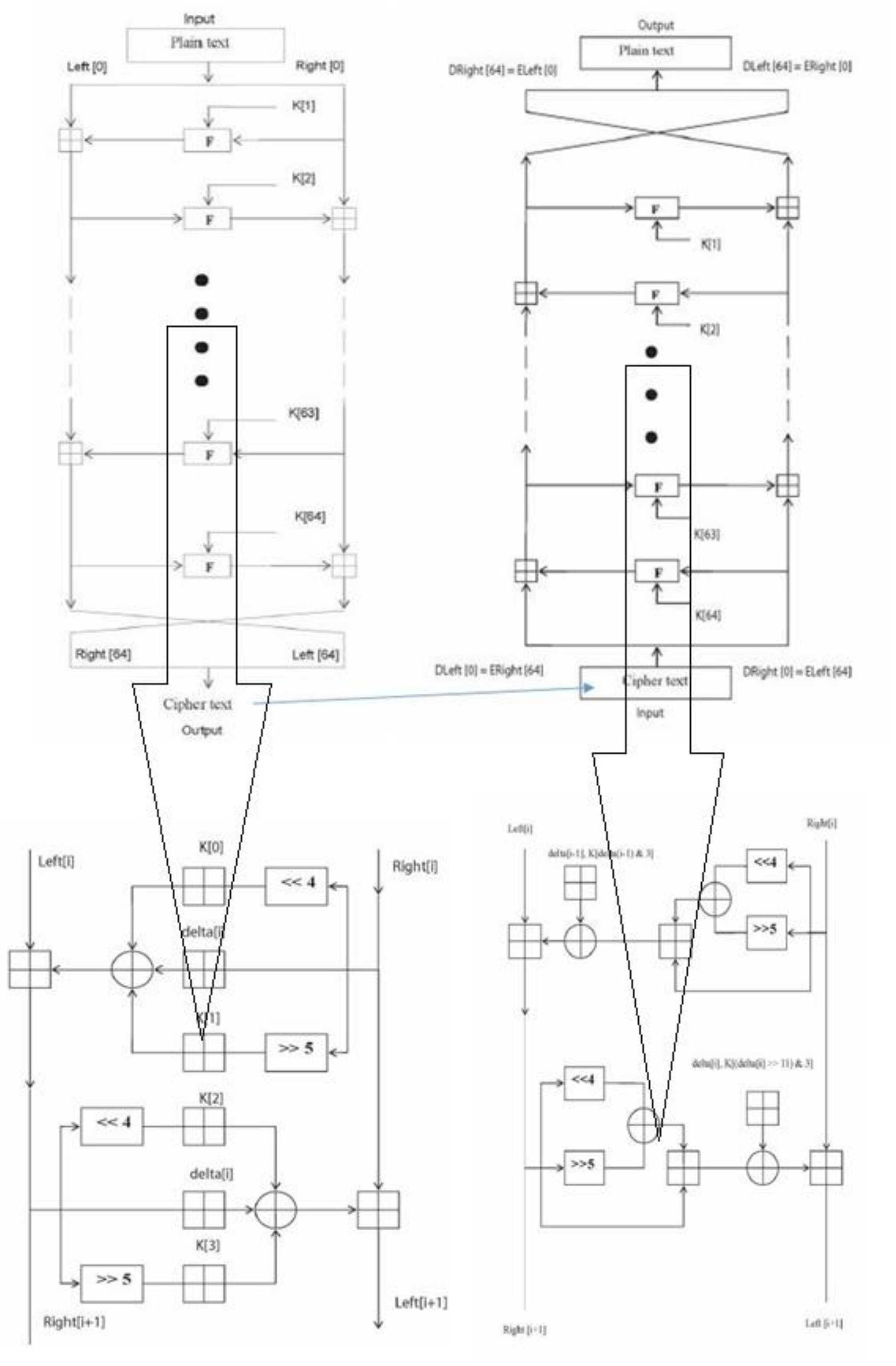


Figure 15 Encrytion and decryption process of TEA

Figure 14 shows how TEA module encrypts and decrypts a plain text briefly. Here ‘K‘ is the key which is obtained from DH key exchange protocol. The encryption and decryption module RTL schematics can be on Figure 15.

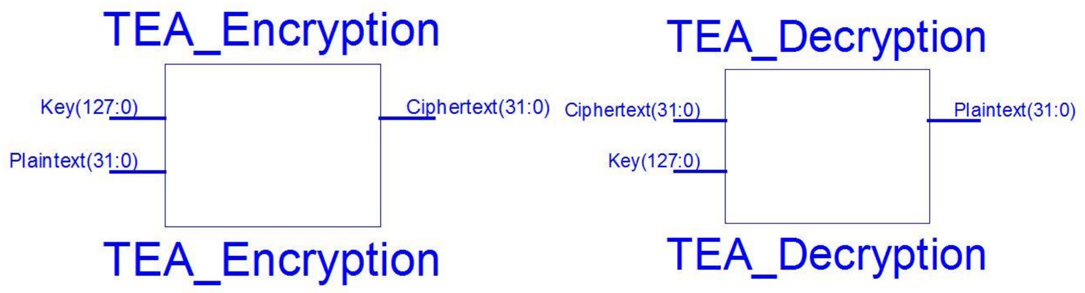


Figure 16 TEA Encryption and Decryption module RTL schematics

4. HARDWARE DESING

4.1. Communication Between Two FPGA

As it is mentioned on the implementation topic, two different FPGAs are planned to communicate. This communication method is chosen to be Universal Asynchronous Receiver/Transmitter (UART) protocol. Generally, UART communication is made by using RS232 cable. This cable is used at serial communication transmission of data. This cable can be connected to a computers Universal Serial Bus (USB) socket.

Spartan 6 FPGA is does not have any inputs for RS232 therefore the communication is made by using Rx and Tx ports to send and receive data.

4.1.1. UART protocol

The UART performs serial-to-parallel conversions on data received from a peripheral device and parallel-to-serial conversion on data received from the CPU. The CPU can read the UART status at any time. The UART includes control capability and a processor interrupt system that can be tailored to minimize software management of the communications link. [10]

Sending serial data on a single line accurately needs some control to be applied. Therefore UART protocol, except the data bits, has a parity bit which is optional, a stop bit and a start bit. When there is no data to send the line is in idle case. Idle case terminates when the start bit is seen and sends data until the stop bit is raised. Pairty bit is optional that comes before the end bit. Tx line is used to transmit data and Rx is to receive data. Every single bit of data is sent according to the Baudrate which is the rate of data sent per second. Baudrate is generally set to 9600.

4.1.2. Configurations Made on FPGA to Communicate

In the case of communicating two Spartan 6 FPGAs, for the reason RS232 cannot be used, UART protocol is made manually. Thus, Rx and Tx ports are defined on both FPGAs. Baudrate is set to 9600 and non-parity mode is selected.

First FPGA’s sending port is set as the second FPGA’s receive port which means, Tx1 is connected to Rx2 and Rx1 is connected to Tx2. So, both sides are able to send and receive data. The pins “**T3**” and “**R3**” are used as Tx and Rx. The adjustments and changes on User Constraints File (UCF) for UART communication are made on XPS.

4.2. Microblaze – Hardware Relationship

Microblaze is used to conclude DH key exchange steps that is explained on “**Implementation of DH Key Exchange Protocol**” topic. Programming Microblaze is made on Xilinx SDK but before passing to SDK, firs all the verilog modules have to be embedded to EDK. In this project the necessary modules are the ones used to compute A over B mod C and the TEA algorithm module. After these modules are embedded to EDK, libraries are produced for these modules so Microblaze can control them by being programmed on SDK.

Accordingly, to send data using UART protocol on SDK, the codes were written in C language. To send data from Tx, ***outbyte()*** function is being used whereas to receive data from Rx, ***inbyte()*** function is used. Also, all the module calculations were made by the special functions which were inside of the produced libraries.

Another thing that Microblaze is responsible about producing random numers for FPGA 1 and FPGA 2. For this random function generator which belongs to the library “**time.h**” is used. So both FPGA has their own secret number.

By using the properties of debugging all the steps have been able to be observed. Being able to see all the steps makes any mistake to be solve easily.

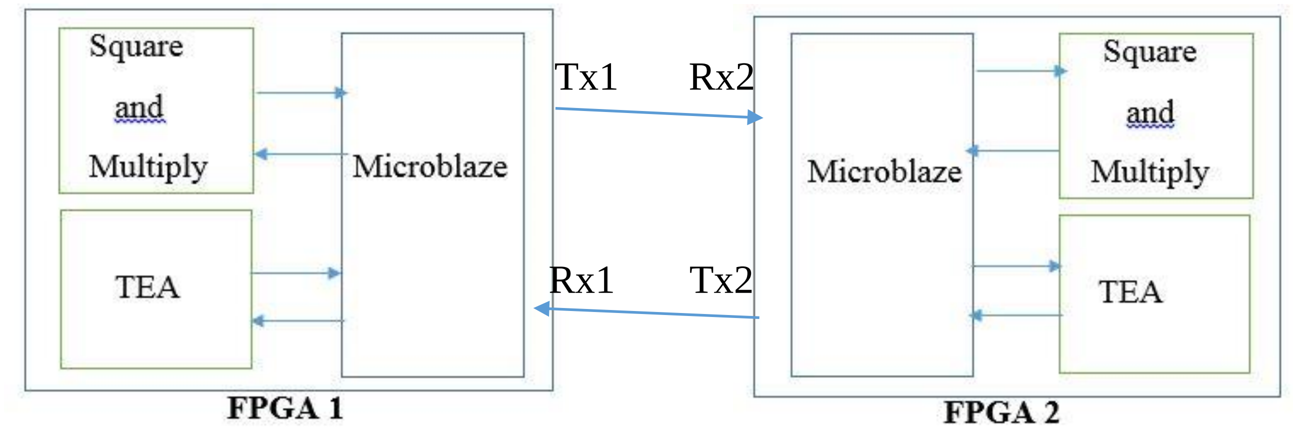


Figure 17 Overall system connections

Figure 16 explains the interior and exterior overall hardware system connection.

5. OPTIMIZATION

At the very beginning, the modules to realize DH key exchange, were designed in a different way. Karatsuba algorithm was asynchronous and square and multiply module was completely different. square and multiply module was different because of the calculation of A ^ B mod C. The module named "**A_over_B_mod_C**" was implemented in order to compute the following equation.

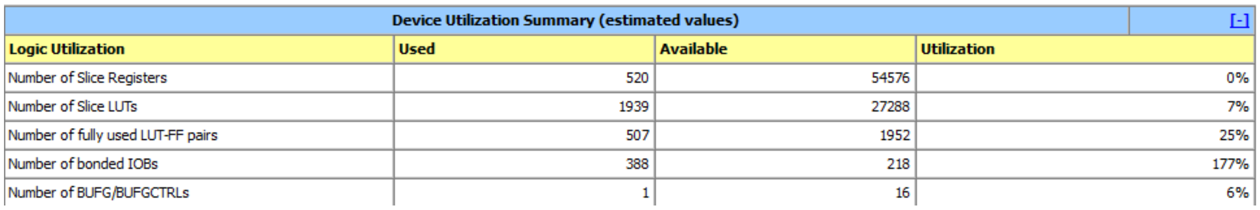
$$A \wedge B \,\%\, C = \underbrace{(A * A \,\%\, C) * A \,\%\, C) * A \,\%\, C)\ldots * A \,\%\, C)}_{\text{B times}}$$

This module had changed for the reason of time consumption. Whenever B is produced as a big number (since B is 128 bit), the computing time proportionally increases. In the case of producing a 128 bit length number, obtaining the result takes 2 ^ 128 clock cycle as time. Consequently, calculation of A ^ B mod C is impossible for huge B values.

Asynchronous Karasuba algorithm possesses problem of space consumption. For this module, gain of time ends up loss on space. Because of implementing expansion of Karatsuba equations, the module demands enormously many FPGA IO-block amount. This module was not able to fit in Spartan-6 FPGA. After several tests, the results showed the module was only able to fit in of Xilinx FPGA Virtex-6.

Only one single Virtex-6 was idly for usage, thus single FPGA is not convenient for this project (reason for communication). The module has decreased to Spartan – 6 by the loss of time and gain of space.

A ^ B mod C module was the top module that had to be fit in Spartan – 6 FPGA. After implementing the module before optimization, this design summary table was acquired.



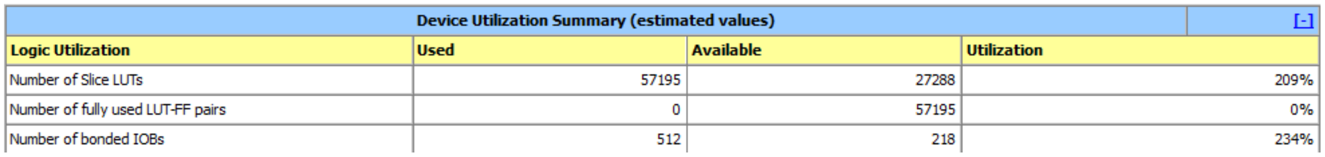
Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	520	54576	0%
Number of Slice LUTs	1939	27288	7%
Number of fully used LUT-FF pairs	507	1952	25%
Number of bonded IOBs	388	218	177%
Number of BUFG/BUFGCTRLs	1	16	6%

Figure 18 A over B mod C module design summary before optimization

As it is seen from Figure 17, the module is not able to fit inside Spartan – 6 FPGA because of using too much IOBs. After optimization the top module which is the square and multiply module, is successfully reduced to fit in Spartan – 6 FPGA. The design summary belongs to this module takes place under the “**Results**” topic in detail.

6. RESULTS

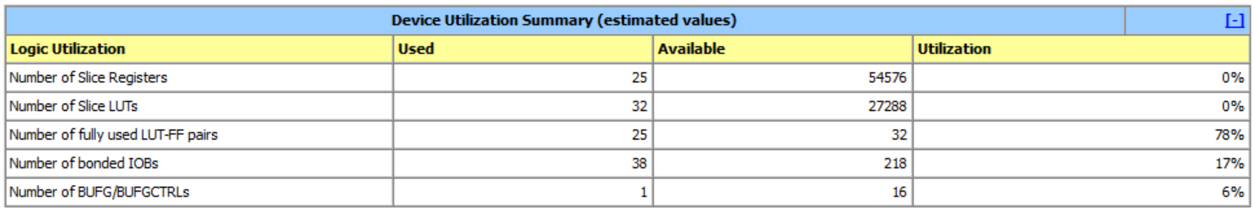
Space consumption for Karatsuba module can be seen as below;



Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice LUTs	57195	27288	209%
Number of fully used LUT-FF pairs	0	57195	0%
Number of bonded IOBs	512	218	234%

Figure 19 Karatsuba Multiplier design summary after optimization

Space consumption for square and multiply module can be seen as below;

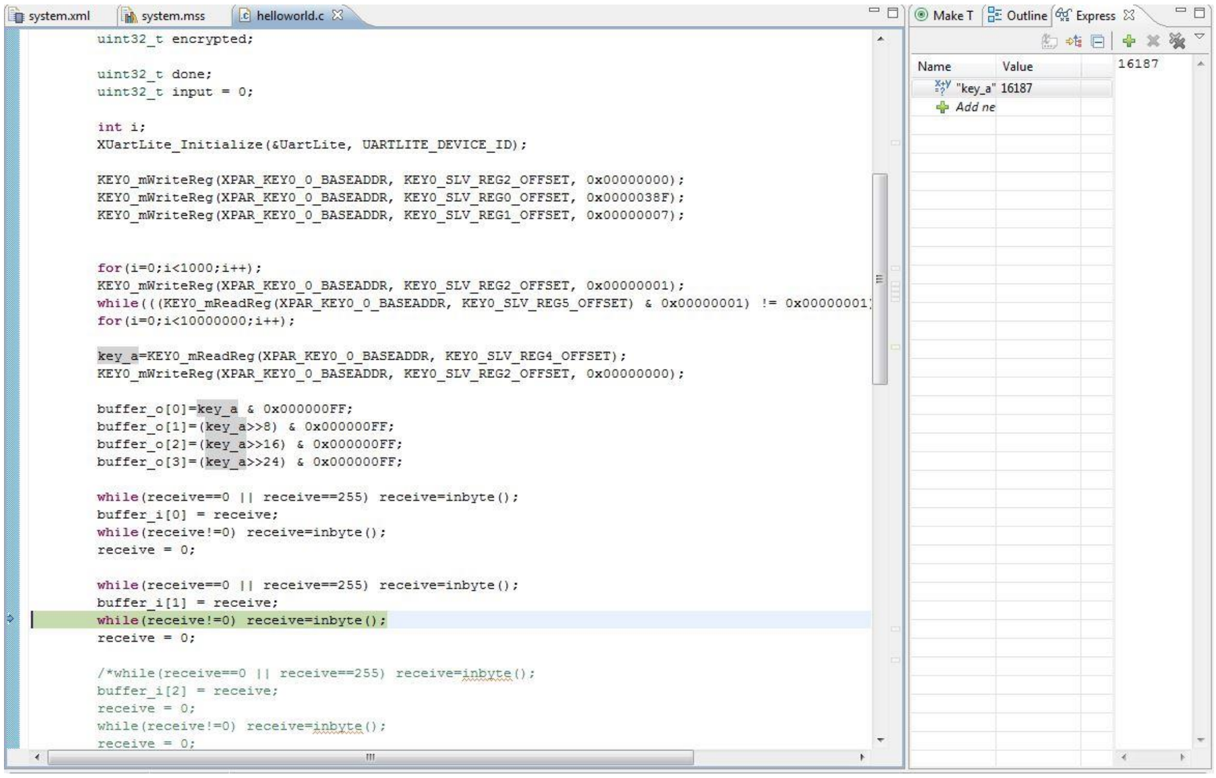


Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	25	54576	0%
Number of Slice LUTs	32	27288	0%
Number of fully used LUT-FF pairs	25	32	78%
Number of bonded IOBs	38	218	17%
Number of BUFG/BUFGCTRLs	1	16	6%

Figure 20 Square and Multiply design summary after optimization

Considering Figure 18, Karatsuba multiplier module is not able to fit inside Spartan – 6 FPGA. However when this module is called by the top module which is square and multiply, IOB size decreases so that the module is able to fit in the desired FPGA. Top module (square and multiply) is able to fit in Spartan – 6 FPGA as it is seen from Figure 19.

Simulation obtaining key, using DH key exchange method on SDK can be seen on Figure 20. The information is gathered by using SDK debugging feature.



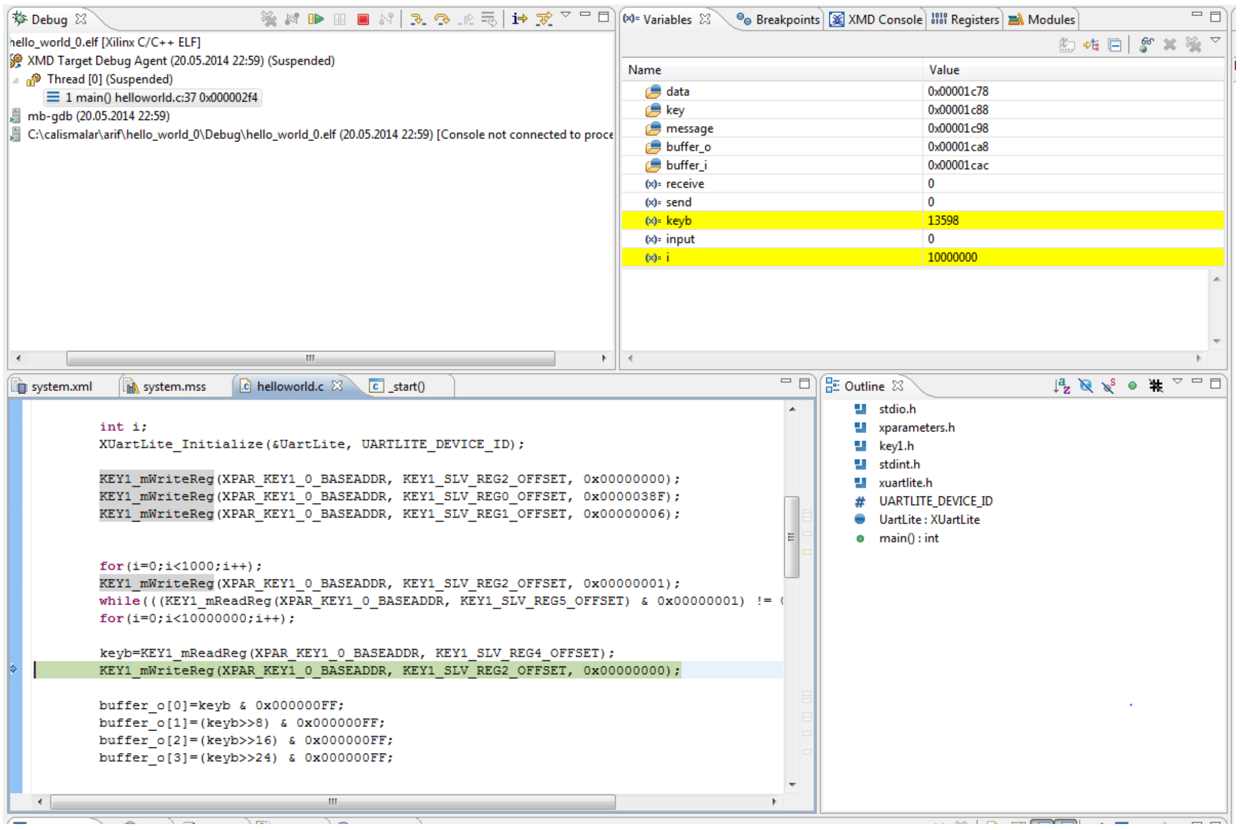
```
uint32_t encrypted;

uint32_t done;
uint32_t input = 0;

int i;
XUartLite_Initialize(&UartLite, UARTLITE_DEVICE_ID);

KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000000);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG0_OFFSET, 0x0000038F);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG1_OFFSET, 0x00000007);


for(i=0;i<1000;i++);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000001);
while(((KEY0_mReadReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG5_OFFSET) & 0x00000001) != 0x00000001
for(i=0;i<10000000;i++);

key_a=KEY0_mReadReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG4_OFFSET);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000000);

buffer_o[0]=key_a & 0x000000FF;
buffer_o[1]=(key_a>>8) & 0x000000FF;
buffer_o[2]=(key_a>>16) & 0x000000FF;
buffer_o[3]=(key_a>>24) & 0x000000FF;

while(receive==0 || receive==255) receive=inbyte();
buffer_i[0] = receive;
while(receive!=0) receive=inbyte();
receive = 0;

while(receive==0 || receive==255) receive=inbyte();
buffer_i[1] = receive;
while(receive!=0) receive=inbyte();
receive = 0;

/*while(receive==0 || receive==255) receive=inbyte();
buffer_i[2] = receive;
receive = 0;
while(receive!=0) receive=inbyte();
receive = 0;
```

Name	Value
"key_a"	16187
Add ne	

```
16187
```

Figure 21 Obtaining the result on debug screen from FPGA1

On Figure 20, the first FPGA has produced its own key using its own secret variable. The debug screen shows which result is computed. This result is sent to the other FPGA to produce a key in common. The result from square and multiply algorithm is found as **16187**.



```
hello_world_0.elf [Xilinx C/C++ ELF]
XMD Target Debug Agent (20.05.2014 22:59) (Suspended)
  Thread [0] (Suspended)
    1 main() helloworld.c:37 0x000002f4
mb-gdb (20.05.2014 22:59)
C:\calismalar\arif\hello_world_0\Debug\hello_world_0.elf (20.05.2014 22:59) [Console not connected to proce
```

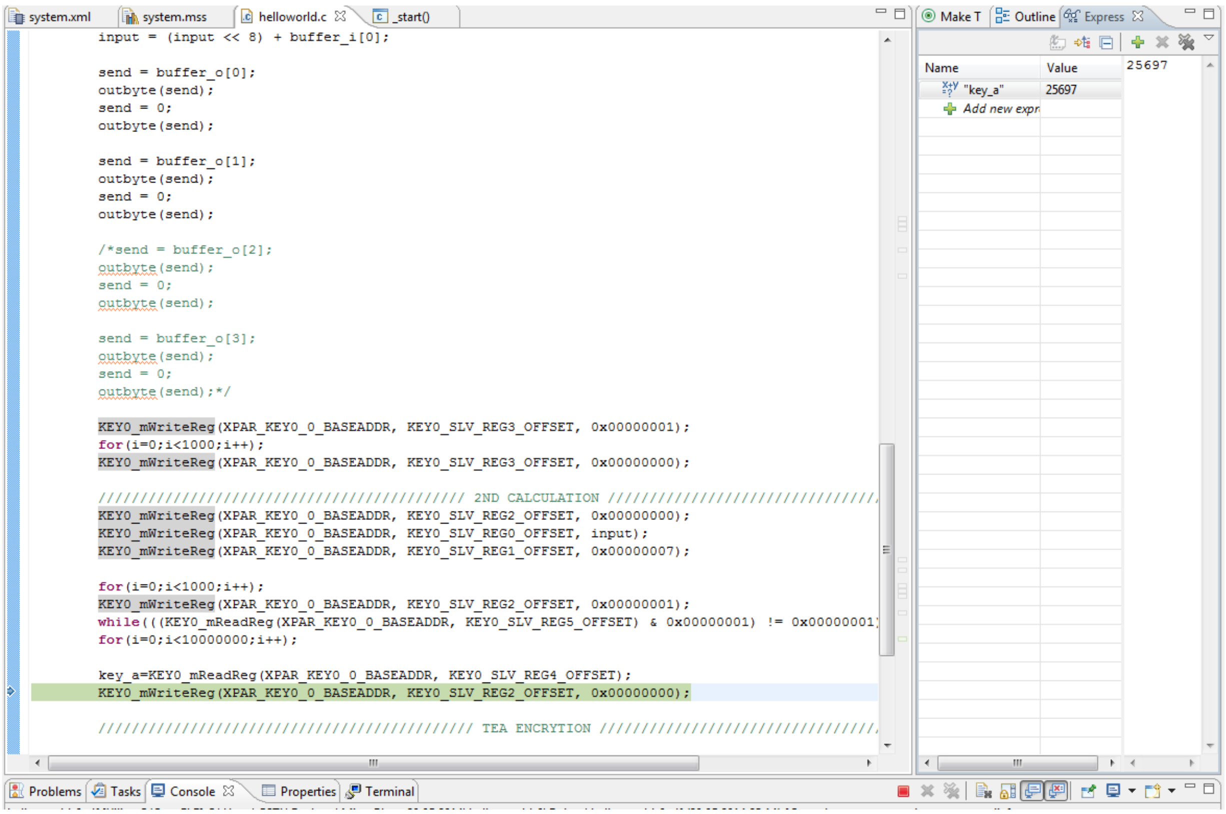
Name	Value
data	0x00001c78
key	0x00001c88
message	0x00001c98
buffer_o	0x00001ca8
buffer_i	0x00001cac
receive	0
send	0
keyb	13598
input	0
i	10000000

```
        int i;
        XUartLite_Initialize(&UartLite, UARTLITE_DEVICE_ID);

        KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG2_OFFSET, 0x00000000);
        KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG0_OFFSET, 0x0000038F);
        KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG1_OFFSET, 0x00000006);


        for(i=0;i<1000;i++);
        KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG2_OFFSET, 0x00000001);
        while(((KEY1_mReadReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG5_OFFSET) & 0x00000001) != (
        for(i=0;i<10000000;i++);

        keyb=KEY1_mReadReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG4_OFFSET);
        KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG2_OFFSET, 0x00000000);

        buffer_o[0]=keyb & 0x000000FF;
        buffer_o[1]=(keyb>>8) & 0x000000FF;
        buffer_o[2]=(keyb>>16) & 0x000000FF;
        buffer_o[3]=(keyb>>24) & 0x000000FF;
```

```
stdio.h
xparameters.h
key1.h
stdint.h
xuartlite.h
UARTLITE_DEVICE_ID
UartLite : XUartLite
main() : int
```

Figure 22 Obtaining the result on debug screen from FPGA2

Next, the second FPGA has produced its own key using its own secret variable. The debug screen shows which result is computed. This result is sent to the other FPGA to produce a key in common. As it can be seen on Figure 21, the result from square and multiply algorithm is found as **13598**.



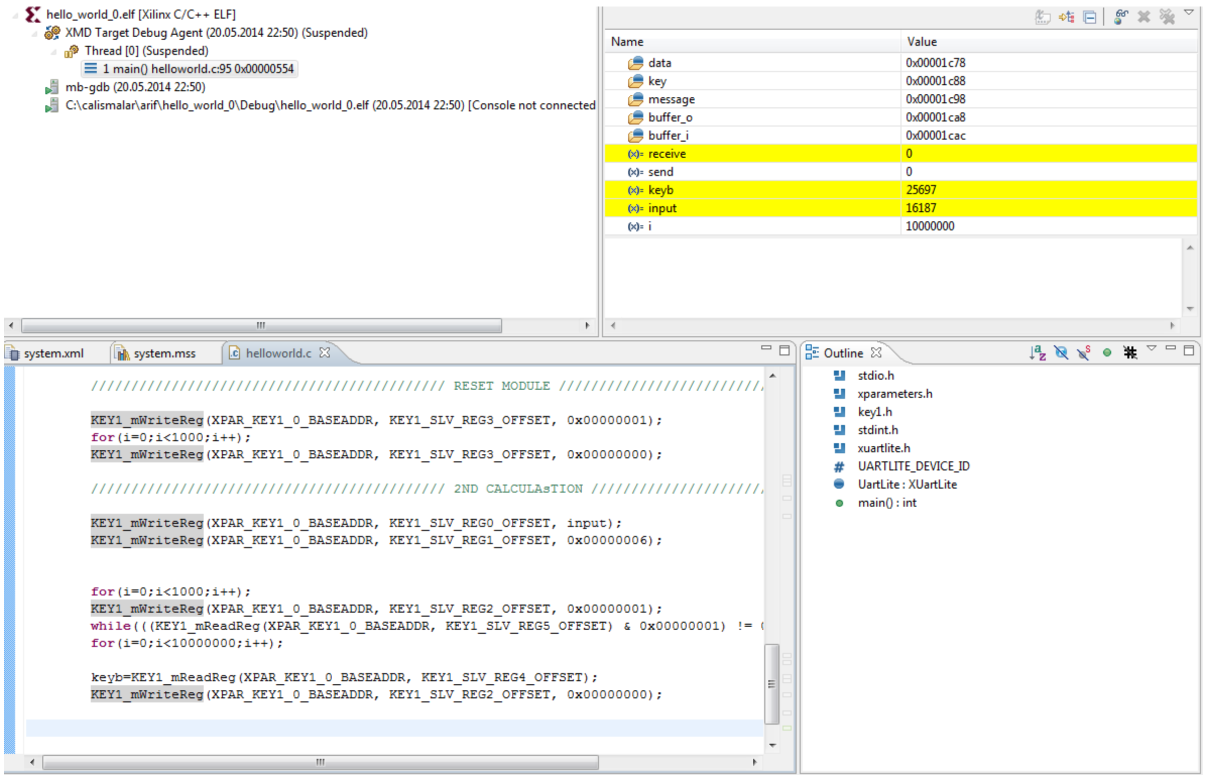
```
input = (input << 8) + buffer_i[0];

send = buffer_o[0];
outbyte(send);
send = 0;
outbyte(send);

send = buffer_o[1];
outbyte(send);
send = 0;
outbyte(send);

/*send = buffer_o[2];
outbyte(send);
send = 0;
outbyte(send);

send = buffer_o[3];
outbyte(send);
send = 0;
outbyte(send);*/

KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG3_OFFSET, 0x00000001);
for(i=0;i<1000;i++);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG3_OFFSET, 0x00000000);

///////////////////////////////////////////// 2ND CALCULATION ///////////////////////////////
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000000);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG0_OFFSET, input);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG1_OFFSET, 0x00000007);

for(i=0;i<1000;i++);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000001);
while(((KEY0_mReadReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG5_OFFSET) & 0x00000001) != 0x00000001)
for(i=0;i<10000000;i++);

key_a=KEY0_mReadReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG4_OFFSET);
KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000000);

////////////////////////////////////////////// TEA ENCRYTION ////////////////////////////////
```

Name	Value
"key_a"	25697
Add new expr	

25697

Figure 23 Producing the key on FPGA1

After first FPGA sends the result (16187) to the second FPGA, this result enters as an input to compute the shared key. The key computed key on the first FPGA is **25697** as it is seen from Figure 22.



```
hello_world_0.elf [Xilinx C/C++ ELF]
  XMD Target Debug Agent (20.05.2014 22:50) (Suspended)
    Thread [0] (Suspended)
      1 main() helloworld.c:95 0x00000554
  mb-gdb (20.05.2014 22:50)
  C:\calismalar\arif\hello_world_0\Debug\hello_world_0.elf (20.05.2014 22:50) [Console not connected
```

Name	Value
data	0x00001c78
key	0x00001c88
message	0x00001c98
buffer_o	0x00001ca8
buffer_i	0x00001cac
receive	0
send	0
keyb	25697
input	16187
i	10000000

```
    ///////////////////////////////////////// RESET MODULE ///////////////////////////

    KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG3_OFFSET, 0x00000001);
    for(i=0;i<1000;i++);
    KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG3_OFFSET, 0x00000000);

    ///////////////////////////////////////// 2ND CALCULAsTION /////////////////////

    KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG0_OFFSET, input);
    KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG1_OFFSET, 0x00000006);


    for(i=0;i<1000;i++);
    KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG2_OFFSET, 0x00000001);
    while(((KEY1_mReadReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG5_OFFSET) & 0x00000001) != (
    for(i=0;i<10000000;i++);

    keyb=KEY1_mReadReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG4_OFFSET);
    KEY1_mWriteReg(XPAR_KEY1_0_BASEADDR, KEY1_SLV_REG2_OFFSET, 0x00000000);
```

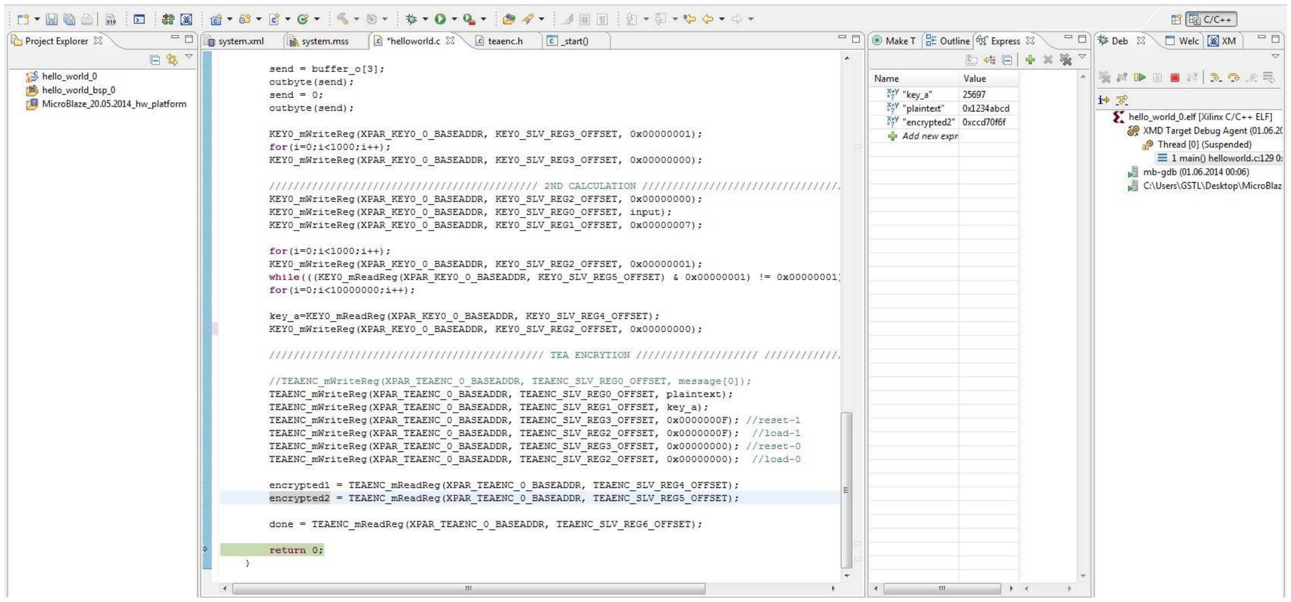
```
Outline
  stdio.h
  xparameters.h
  key1.h
  stdint.h
  xuartlite.h
  UARTLITE_DEVICE_ID
  UartLite : XUartLite
  main() : int
```

Figure 24 Producing the key on FPGA2

As well as the second FPGA does the same procedure with the first one, they both compute the same value which can be seen on Figure 22 and Figure 23, the computed key is same on both sides. Both FPGA's were able to share a key in secret, for this example the key is calculated as **25697.**

Finally, the produced key is used as encryption and decryption on a message to be sent. Therefore, FPGA 1 includes a TEA encryption module while TEA decryption module is embedded in FPGA 2.

FPGA 1 has gathered encrypted value by using the common key (25697) and produced a ciphered data of a message which is given as an input to the encryption module. Figure 24 shows the ciphered message data.



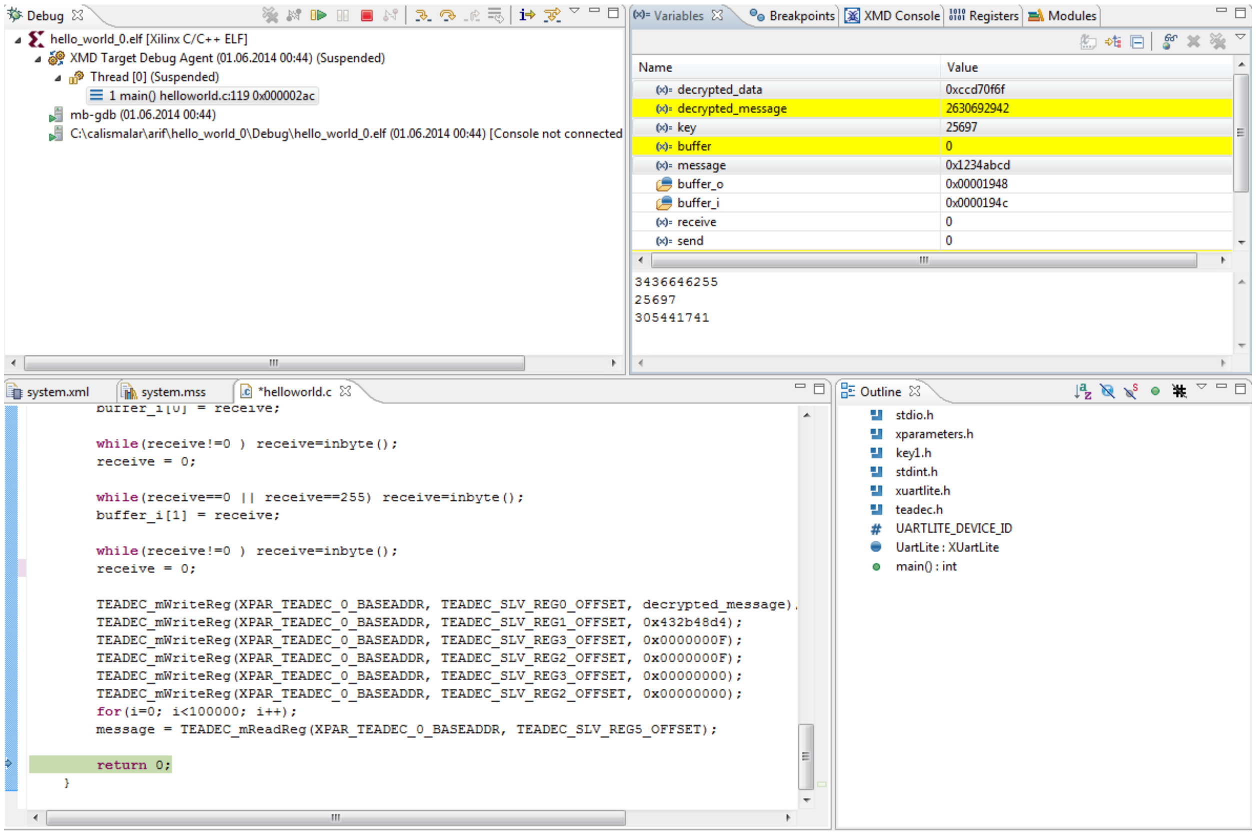
```
        send = buffer_o[3];
        outbyte(send);
        send = 0;
        outbyte(send);

        KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG3_OFFSET, 0x00000001);
        for(i=0;i<1000;i++);
        KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG3_OFFSET, 0x00000000);

        ///////////////////////////////////////////// 2ND CALCULATION ////////////////////////////////
        KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000000);
        KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG0_OFFSET, input);
        KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG1_OFFSET, 0x00000007);

        for(i=0;i<1000;i++);
        KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000001);
        while(((KEY0_mReadReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG5_OFFSET) & 0x00000001) != 0x00000001)
        for(i=0;i<10000000;i++);

        key_a=KEY0_mReadReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG4_OFFSET);
        KEY0_mWriteReg(XPAR_KEY0_0_BASEADDR, KEY0_SLV_REG2_OFFSET, 0x00000000);

        ///////////////////////////////////////////// TEA ENCRYTION //////////////////// ////////////

        //TEAENC_mWriteReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG0_OFFSET, message[0]);
        TEAENC_mWriteReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG0_OFFSET, plaintext);
        TEAENC_mWriteReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG1_OFFSET, key_a);
        TEAENC_mWriteReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG3_OFFSET, 0x0000000F); //reset-1
        TEAENC_mWriteReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG2_OFFSET, 0x0000000F);  //load-1
        TEAENC_mWriteReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG3_OFFSET, 0x00000000); //reset-0
        TEAENC_mWriteReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG2_OFFSET, 0x00000000);  //load-0

        encrypted1 = TEAENC_mReadReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG4_OFFSET);
        encrypted2 = TEAENC_mReadReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG5_OFFSET);

        done = TEAENC_mReadReg(XPAR_TEAENC_0_BASEADDR, TEAENC_SLV_REG6_OFFSET);

        return 0;
    }
```

Name	Value
"key_a"	25697
"plaintext"	0x1234abcd
"encrypted2"	0xccd70f6f
Add new expr	

```
hello_world_0.elf [Xilinx C/C++ ELF]
  XMD Target Debug Agent (01.06.2(
    Thread [0] (Suspended)
      1 main() helloworld.c:129 0:
  mb-gdb (01.06.2014 00:06)
  C:\Users\GSTL\Desktop\MicroBlaz
```

Figure 25 TEA encryption module output

The message that is about to be sent to the other FPGA is chosen as "**0x1234ABCD**". This message is given as input to the encryption module according to the key calculated previously which is **25697.** Output of this module is the chippered value "**0xCCD70F6F**" which can be seen from Figure 25.



```
hello_world_0.elf [Xilinx C/C++ ELF]
  XMD Target Debug Agent (01.06.2014 00:44) (Suspended)
    Thread [0] (Suspended)
      1 main() helloworld.c:119 0x000002ac
  mb-gdb (01.06.2014 00:44)
  C:\calismalar\arif\hello_world_0\Debug\hello_world_0.elf (01.06.2014 00:44) [Console not connected
```

Name	Value
decrypted_data	0xccd70f6f
decrypted_message	2630692942
key	25697
buffer	0
message	0x1234abcd
buffer_o	0x00001948
buffer_i	0x0000194c
receive	0
send	0

```
3436646255
25697
305441741
```

```c
        buffer_i[0] = receive;

        while(receive!=0 ) receive=inbyte();
        receive = 0;

        while(receive==0 || receive==255) receive=inbyte();
        buffer_i[1] = receive;

        while(receive!=0 ) receive=inbyte();
        receive = 0;

        TEADEC_mWriteReg(XPAR_TEADEC_0_BASEADDR, TEADEC_SLV_REG0_OFFSET, decrypted_message);
        TEADEC_mWriteReg(XPAR_TEADEC_0_BASEADDR, TEADEC_SLV_REG1_OFFSET, 0x432b48d4);
        TEADEC_mWriteReg(XPAR_TEADEC_0_BASEADDR, TEADEC_SLV_REG3_OFFSET, 0x0000000F);
        TEADEC_mWriteReg(XPAR_TEADEC_0_BASEADDR, TEADEC_SLV_REG2_OFFSET, 0x0000000F);
        TEADEC_mWriteReg(XPAR_TEADEC_0_BASEADDR, TEADEC_SLV_REG3_OFFSET, 0x00000000);
        TEADEC_mWriteReg(XPAR_TEADEC_0_BASEADDR, TEADEC_SLV_REG2_OFFSET, 0x00000000);
        for(i=0; i<100000; i++);
        message = TEADEC_mReadReg(XPAR_TEADEC_0_BASEADDR, TEADEC_SLV_REG5_OFFSET);

        return 0;
    }
```

Figure 26 TEA decryption module output

After the chippered key is sent to the other FPGA, this value enters to the decryption TEA module and decrypts the chippered message according to the common key. So, the input for decryption module is "**25697**" which is the key and the chippered message "**0xCCD70F6F**". As it can be seen from Figure 26 the output is the original text which has entered the encryption module which is "**0x1234ABCD**".

REFERENCES

[1] **Anderson, R.,** 2001. Security Engineering: A Guide to Building Dependable Distributed
Systems, pp.75.
[2**] Ibrahim, Mahmood K.,** 2012. "Modification of Diffie–Hellman Key Exchange
Algorithm for Zero Knowledge Proof ", Eng.&Tech. Journal, VoL,30. No.3, 2012
pp. 444.
[3] **Chu, Pong P.,** 2008. FPGA Prototyping by VHDL Examples. Wiley-
Interscience, New Jersay, pp. 12.
[4] **Xilinx,** 2008. Virtex-4 FPGA User Guide.
[5] **Xilinx**, 2011. Spartan-6 Family Overview.
[6] **Xilinx,** Xilinx Software Development Kit Help Contents, [Citation Date: 10 May 2014],
http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/SDK_Doc/index.html.
[7]**Xilinx,** 2008. MicroBlaze Processor Reference Guide, pp.10.
[8] **Xilinx,** 2002 – 2008. Embedded System Tools Reference Manual, pp. 19-20.
[9] **Grabowski, J.; Keurian, J.,** 2010. "Tiny Encryption Algorithm**"**, Team Garrett
– Cryptography Submission, pp. 3.
[10] **Andem, Vikram R.,** 2003, "CRYPTANALYSIS OF THE TINY
ENCRYPTION ALGORITHM", Master Thesis, pp. 6-8-10.
[11] **Texas Instrument,** 2010. Universal Asynchronous Receiver/Transmitter
(UART), User Guide.
[12] **Hoffstein, J.,** 2008. An Introduction to Mathematical Cryptography, pp. 66.
[13] **Ritter, E.,** 2013. "Asymmetric Ciphers and Public Key Cryptography",
Handout, pp. 32.
[14] **Williams, D.,** 2008. The Tiny Encryption Algorithm (TEA), pp. 1-7.
[15] **Jorganxhi, A.,** 2014. "A Software-Hardware Common Implementation of a
Secured data communication protocol using AES algorithm", Thesis.
[16] **Network Associates,** 1999. An Introduction to Cryptography, pp. 11.

RESUME

Name Surname: Arif Gencosmanoglu

Birth Place and Date: Maine/USA, 1990

HighSchool: Ihlas Collage; 2004 – 2006

BSc: Istanbul Technical University, Electronics Engineering; 2009-20014