FPGA Implementation of RSA Encryption System

Semester Project
Design and Implementation Report

by
Kamran Ali 13100174
Muhammad Asad Lodhi 13100175
Ovais bin Usman 13100026

Advisor
Dr. Shahid Masud
[smasud@lums.edu.pk]

Reader

December 26, 2012
Department of Electrical Engineering
Syed Babar Ali School of Science and Engineering
Lahore University of Management Sciences, Pakistan

Contents

1 Introduction **1**
- 1.1 Cryptography 1
  - 1.1.1 Symmetric Encryption 2
  - 1.1.2 Public Key (Asymmetric) Encryption . 2

2 RSA Public Key Encrption **2**
- 2.1 Mathematics behind RSA 2
  - 2.1.1 Prime Numbers 2
  - 2.1.2 Modular Arithematics 2
- 2.2 RSA Encrytion/Decryption Algorithm 2
  - 2.2.1 Encryption 3
  - 2.2.2 Decryption 3
- 2.3 Example 3

3 Implementation **3**
- 3.1 Random Number Generation 3
- 3.2 Detection of primes : Miller-Rabin Primality Test 3
- 3.3 Extended Euclidean algorithm [7] 4
- 3.4 VGA and Keyboard interfacing 4
- 3.5 Block wise implementation of modules 4

4 Testing Methodology **4**

5 Conclusion **5**

1 Introduction

Means and amount of communicated information has changed a lot since last two or three decades. Specially, the amount of information communicated electronically has grown and is growing exponentially fast. It is very much important to develop new ways to guarantee security over the communication channels. So, in order to deal with this large a amount of data, a high performance *encryption* system is needed which processes the data and and provides security to the overall electronic information system.

Cryptographic algorithms, being the core component of most security systems, are usually based upon the fact that their complexity is superior to present computing power. According to the Moore's Law computing power doubles every 1.5 years [1], the comlexity of cryptographic computations needs to grow at least at the same rate to provide a consistent level of security. But this does also mean, that the actual workload of data processing, meaning encryption and decryption, increases. As a consequence, the demanded amount of computing power of a secured information system increases at the same speed as cryptographic complexity and integration level of its hardware components [2].

One of the algorithms which the above mentioned problems is *RSA* which is the most widely used public key algorithm. A vast numbers and wide varieties of works have been done on this particular field of hardware implementation of *RSA* encryption algorithm. Today, *RSA* is used in IP data security (IPSEC/IKE), transport data security (TLS/SSL) , email security (PGP), terminal connection security (SSH), conferencing service security (SILC) and so on. *The security of RSA revolves around the difficulty of factoring a number into two prime factors. Since RSA encryption and finding prime numbers* are both computationally intensive tasks, we thought it would be interesting to implement them in hardware on the FPGA to see how these algorithms can be implemented in an application specific integrated circuit. In addition, many companies such as Oracle and Intel have added on-chip hardware support for encryption such as AES to their products. This project may help us understand the complexities involved in such implementations.

We will develop our system based on the implementations by *Benedikt*, et al. in [2] and *Khalil*, et al. in [4] who used *Mongomery* Multiplier (which will be explained in the report) to do *modular multiplication* and *exponentiation* for determining primes and for general encryption/decryption. This report will explain the procedure of accelerating an embedded *cryptographic* system by specially designed hardware from the evaluation of the involved algorithms down to designing the accelerator using VHDL.

1.1 Cryptography

Cryptography, thus, literally means the art of secret writing. The art of hiding information therefore is not as modern as one might guess but is known to be some thousand years old. Cryptography provides, amongst others, means of hiding and recovering information called encryption schemes. In general an encryption scheme consists of a set of encryption and decryption operations each associated with a key, which is supposed to be kept secret. There are two main categories of encryption: *symmetric*

or *public key* (asymmetric) encryption mentioned as follows:

1.1.1 Symmetric Encryption

An encryption scheme is related to symmetric cryptography, when it is computationally easy to discover the second key, knowing one of them. In most practical cases the two keys will be identical, which is illustrated by the word symmetric, the shared key is reffered to as *secret key* [3].

A disadvantage is, that all parties involved in the communication process have to share a common secret, the secret key. This implicates more difficulties, than might be obvious at first sight.

"A common image to explain the idea of symmetric encryption is a safe. All participating parties own an identic copy of the key to the safe, so every party can open the safe to either put something inside (encryption), or to get something out (decryption)."[2]

1.1.2 Public Key (Asymmetric) Encryption

An encryption scheme is said to be public key encryption, when it is impossible to compute the second key, knowing one of them. In this context the encryption operation, using the encryption key, can be regarded as a trapdoor one way function, with the decryption key being the trapdoor, that allows easy message recovery. Message recovery without knowledge of the decryption key is computationally infeasible [3].

A major advantage of a scheme belonging to this category is the fact, that one cannot compute the decryption key knowing only the encryption key. This allows distribution of a party's encryption key over insecure channels, which simplifies the process of key distribution. Therefore the encryption key is referred to as **public key** while the decryption key is called **private key**. One of the disadvantages of public key encryption is its bad performance in terms of throughput. In order to keep the decryption key secure, even though the encryption key is available in public, the encryption scheme needs to be more complex than a symmetric one. This denotes that the operations being performed become more complex and time consuming.

"To get an idea of Public Key Encryption, one can imagine a simple mailbox. Anyone can put a letter into the mailbox (public encryption key), but only the owner of the mailbox's key can get the letters out of it (private decryption key)."[2]

2 RSA Public Key Encrption

The problems with private key encryption would be resolved if the decoding mechanism could not be (easily) obtained from the encoding mechanism. But how do you get something like that? The answer is to make breaking the code depend on being able to solve a hard problem, like the factorization of a large number. The RSA cryptosystem is by far the most used public key encryption system. Its name is an abbreviation of the names of R. Rivest, A. Shamir and L. Adleman, who published it in 1978 [5]. This is the most commonly used public-key cryptographic algorithm, and it is considered secure when sufficiently long keys are used. The security of RSA depends on the difficulty of factoring large integers [6].

This section provides a short introduction to the underlying mathematical principles, a detailed look at RSA encryption and decryption operations.

2.1 Mathematics behind RSA

2.1.1 Prime Numbers

An integer p larger than 1 is called a prime number if its only divisors are 1 and p, e.g. p = 2, 3, 5, 7, 11, 13... . There exists an infinite set of prime numbers and there are several well known algorithms of generating prime numbers. Two integers a and b are called *relatively prime*, if their greatest common divisor is 1, e.g. 3 and 4 are relatively prime. Prime numbers play an important role in public key encryption as will be seen later on.

2.1.2 Modular Arithematics

Although most people would say they do not know *modular arithmetic* or *modular reduction* they use it in everyday life [2]. *Modular reduction* means that the set of integer numbers available is limited, the limit is set by the so called *modulus*, denoted by n. Modular arithmetic with the modulus being 5 means, that the set of available numbers consists of $\{0, 1, 2, 3, 4\}$. Any number bigger than or equal to the modulus has to be reduced by the modulus by subtraction until it equals a number within the set of available numbers, this operation is called *modular reduction*.

"*Modular addition* is defined by an ordinary addition followed by a modular reduction operation in order to keep the result within the set of available numbers. Let n = 5, a = 3 and b = 2 then $a + b \equiv 0$ (mod n) since a + b = 5 and $5 \equiv 0$ (mod n). People often use modular reduction when talking about time as 21:00 is referred to as 9:00, which is nothing else than $21 \equiv 9$ (mod 12) " [2].

Modular multiplication, *exponentiation* and *inversion* are the most used and important operations. *Modular multiplication* works exactly the same way as addition: it is an ordinary multiplication followed by a modular reduction operation. $a \cdot b \equiv 1$ (mod 5) since $a \cdot b = 6$ and $6 \equiv 1$ (mod 5). *Modular exponentiation* works slightly different, it can be computed as a series of multiplications followed by a modular reduction operation. a^b (mod n) = $a \cdot a \cdot a$... (mod n). "In practice the modular reduction operation will be performed after each multiplication to keep the intermediate results as small as possible in order to save memory and to avoid unnecessary big inputs to the next multiplication" [2]. a^b (mod n) $\equiv (((a \cdot a)$ (mod n) $) \cdot a$ (mod n)...).

The *multiplicative inverse* of a (mod n) is a number within the set of available integers satisfying $a \cdot b \equiv 1$ (mod n). If b exists, it is unique and denoted by $b = a^{-1}$ (mod n). b exists, if a and n are relatively prime. In the example, a is invertible and the multiplicative inverse of a modulo n is b, as $3 \cdot 2 = 6 \equiv 1$ (mod 5). The well known *Extended Euclidean Algorithm* can be used to compute the greatest common divisor of a and n. If it is 1, the algorithm computes the multiplicative inverse of a at the same time.

2.2 RSA Encrytion/Decryption Algorithm

In order to set up an RSA encryption scheme, several numbers have to be either randomly chosen or computed. **Every party** that wants to participate in RSA secured communication has to set up an own scheme based on following:

- Generate two large random (and distinct) primes p and q, each roughly the same size.
- Compute $n = pq$ and $\phi = (p-1)(q-1)$.
- Select a random integer $e;\ 1 < e < \phi$, such that $gcd(e;\ \phi) = 1$.
- Use the *Extended Euclidean Algorithm* to compute the unique integer $d;\ 1 < d < \phi$, such that $ed \equiv 1 \pmod{\phi}$.
- The public key is (n, e), the private key is d. [3]

2.2.1 Encryption

Following is the RSA public key encryption - key generation algorithm. "In order to encrypt a message m for Alice, Bob" should follow these steps [2]

- Obtain Alice's authentic public key (n, e).
- Represent the message as an integer m in the interval $[0, n-1]$.
- Compute $c = m^e \pmod{n}$.
- Send the cipher-text c to A.

2.2.2 Decryption

If Alice wants to read the received message, she should decrypt the cipher- text according to these steps [2]

- Use the private key d to recover $m = cd \pmod{n}$.

2.3 Example

- I pick $p = 17, q = 11, n = 17 \times 11 = 187$.
- I pick $e = 3, d = 107$ $(ed = 321 = 2 \cdot 16 \cdot 10 + 1)$.
- I post $187;\ 3$.
- You encode the letter J as 10, and put $M = 10$; then $C \equiv M^e \equiv 10^3 \equiv 1000 \equiv 65 \pmod{187}$ and so you send me 65.
- I compute $C^d \pmod{n}$, and find $C^d \equiv 65^{107} \equiv 10 \pmod{187}$.

3 Implementation

3.1 Random Number Generation

For encryption purposes we want a high degree of randomness in the selection and detection of our p's and q's (the two large prime numbers as mentioned in example above). The security of the whole scheme depends upon it. So for generation of large primes with high degree of randomness we needed to have a robust random number generator the pattern of which repeats after 6-7 years. So, we the random numbers were generated using a random number generator from *Stochastic Chemical Reaction Simulation* by *Bruce Land* [7]. The VHDL code is provided with the other project files.

```
Miller-Rabin Primality Test Pseudocode

Input: target accuracy k
Input: an odd integer n to check for primality
Output: prime == 0 if n is composite, otherwise prime == 1 (maybe prime)
LOOP: repeat k times
  compute d and s such that n-1 is 2^s * d
  generate a 32 bit base number by grabbing two 16 bit outputs from the RNG
  regenerate this number if it is less than 2
  // scale the num into the range [2, n-2]
  while the base number is > n-2, shift it right by 1
  compute x = base^d mod n
  if x == 1 or x == n-1 go to next LOOP
  for r = 1 to s-1
    compute x = x^2 mod n
    if x == 1 return composite (prime == 0)
    if x == n-1 go to next LOOP
  return composite (prime == 0)
return maybe prime (prime == 1)
```

Figure 1: Miller-Rabin Primality Test [7]

3.2 Detection of primes : Miller-Rabin Primality Test

The *Miller-Rabin primality* test is based on the properties of strong *pseudoprimes* and relies on a series of inequalities that hold true for composite numbers. These inequalities are used to check if a number is composite. If some of these tests fail, the number is maybe prime. If many of these tests fail, we become more convinced that the number is prime. Therefore, by trying a larger number of these tests, we can gain more confidence in a numbers primality, although we can never be completely sure of it. On the other hand, if a test passes, we immediately know that it is composite and can mark it as such [**?**].

Monier (1980) and Rabin (1980) showed that a composite number passes the test for at most of the possible bases. Thus, if N independent tests are performed on a composite number, the probability that it passes each test is $1/4^N$ or less.

The inequalities rely on square roots of unity. Suppose that x is a square root of 1 mod p, where p is a prime greater than 2. The following must be true: $x^2 \equiv 1$ (mod p), which results in $(x-1)(x+1) = 0$ (mod p). This means that x is either (1 mod p) or (-1 mod p).

Now suppose that n is an odd prime. Then n-1 is an even number and can be written as $2^s * d$ with s and d as positive integers and d odd. If we choose an a in $(Z/nZ)^*$, then: $a^d \equiv 1$ (mod n) or $a^d \equiv -1$ (mod n) for some $0 \leq r \leq s-1$.

Fermats Little Theorem says: $a^n - 1 = 1$ (mod n).

If we repeatedly take square roots of $a^n - 1$, we will get either 1 or -1. This means that if $a^d! \equiv 1$ (mod n) or $a^d! \equiv -1$ (mod n) for some $0 \leq r \leq s-1$, then n is not prime. Thus, if a is chosen and the test passes, we are sure of ns compositeness. We can call a a witness for the compositeness of n. Otherwise, a can be called a strong *liar*, and we can call n a strong probable prime. We can generate our a randomly in order to probabilistically determine ns primality [7].

In figure 1 is the pseudocode for this algorithm [7],

"We feed in odd numbers and a desired accuracy and wait for a finish signal from the module. When the finish signal is detected, we check the prime wire to see whether our number is maybe prime or definitely composite. If the number is prime, we save it and if the user presses the correct key, this number is used in the RSA encryption algorithm. The number is sent to the VGA module to be viewed by the user" [7].

```
extended_euclidean_main(p,q)
    e := 1
    (gcd, d) := (0,0)
    while (gcd != 1 || d < 0)
    begin
        e := e + 2
        (gcd, d) := extended_euclidean_loop((p - 1)(q - 1), e)
    end
    return (e,d)

extended_euclidean_loop(a,b)
    (y, y_prev) := (1, 0)
    while b != 0
    begin
        (y, y_prev) := (y_prev - a/b*y, y)
        (a, b)      := (b, a mod b)
    end
    return (a, y_prev) {gcd((p-1)(q-1),e) is a, inverse is y_prev}
```

Figure 2: Extended Euclidean algorithm [7]

3.3 Extended Euclidean algorithm [7]

The *extended Euclidean algorithm* is used to find d. In our implementation, we iterate through values of e, starting from $e = 3$, until the extended Euclidean algorithm indicates that the greatest common divisor of e and $(p-1)(q-1)$ is 1, indicating that they are relatively prime, and computes a positive value for d.

In figure 2 is the pseudocode for our implementation. Unlike the standard algorithm, we do not compute x and $x - prev$ as their values are not needed for our project.

This module reads values of p and q and computes $(p-1)$ and $(q-1)$. The module then performs the *extended Euclidean algorithm* to find the greatest common divisor of e and (p-1)(q-1) as well as the modular inverse of e mod (p-1)(q-1). If the greatest common divisor is 1 (indicating that e and $(p-1)(q-1)$ are relatively prime to each other) and the *modular multiplicative inverse* is positive, the module returns the values of e and the *modular multiplicative inverse* d to the us. Otherwise, e is incremented by 2 and the algorithm executed again, this is repeated until a value of e which results in a greatest common divisor of 1 and a positive inverse is found. e will be used as the encryption exponent and d as the decryption exponent.

3.4 VGA and Keyboard interfacing

The Complete VHDL code of the VGA and keyboard interfacing with rest of the project modules is provided with this report. It took a lot of time on our part to create a fully functional and efficient interface. Figure 3 shows the FONT ROM and CHARACTER ROM, the two basic blocks for the character display module. The code for **Timing Generation**, etc. is provided along with codes for other modules. *Moreover, the scanned copies of the calculations and notes for both Keyboard and VGA control are also provided.*

We interfaced PS/2 Keyboard afterwards. Again it was a tedious task as We received keystrokes from the keyboard and created two registers that keep track of the current "Write" position of characters: one register for the character column and one register for the character row. When a new character is received from the PS/2 receiver, we write the value of the character into the character memory. Also, increment the column register so the next character is written in the next column position. If the character is written into the last column (79), set the column address to 0 and increment the row register (i.e., have the characters wrap around once you have finished a complete line).

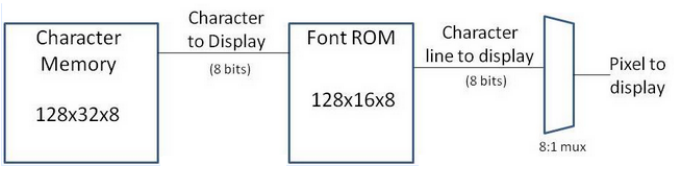

Figure 3: VGA blocks

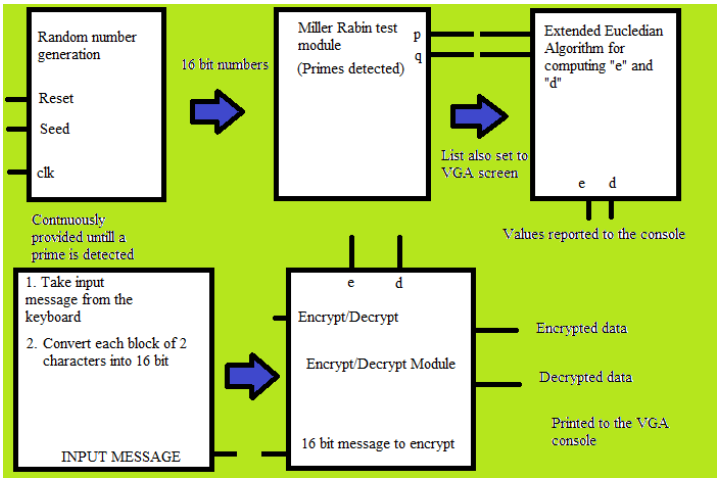

Figure 4: Extended Euclidean algorithm [7]

3.5 Block wise implementation of modules

Figure 4 shows the general block diagram of our RSA implementation. There are 4 major blocks as you can easily see in the diagram.

Figure 5 and 6 shows timing and clock constraints of the implementation. Figure 7 shows the project status and Device Utilization Summary. Figure 8 is showing the HDL Synthsis report.

4 Testing Methodology

As described in previous section, we were as efficient as we could in interfacing our PS/2 keyboard and the VGA monitor.So, we did not need to use any test bench. We wrote our code and checked the outputs on the screen. Following two figures show how we tested our *random number generator*. At the top corner you can see a random number of 8 bits **ASCII** is being gener-

	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	Autotimespec constraint for clock net clk25	SETUP		8.561ns		0
			HOLD	0.927ns		0	0
2	Yes	Autotimespec constraint for clock net clk50_in_BUFGP	SETUP		3.945ns		0
			HOLD	1.542ns		0	0
3	Yes	Autotimespec constraint for clock net KEYBRD/QOUT<18>	SETUP		1.905ns		0
			HOLD	1.220ns		0	0
4	Yes	Autotimespec constraint for clock net KEYBRD/QOUT<1>	SETUP		5.739ns		0
			HOLD	1.048ns		0	0

Figure 5: Timing Constraints

	Clock Net	Routed	Resource	Locked	Fanout	Net Skew(ns)	Max Delay(ns)
1	clk25	ROUT...	BUFGMUX_X1...	No	152	0.085000	0.203000
2	clk50_in_BUFGP	ROUT...	BUFGMUX_X2...	No	11	0.038000	0.170000
3	KEYBRD/QOUT<...	ROUT...	BUFGMUX_X1...	No	15	0.013000	0.154000
4	KEYBRD/QOUT<...	ROUT...	Local		2	0.000000	1.868000

Figure 6: Clock Constraints

clockmodule Project Status (12/25/2012 - 20:12:03)			
Project File:	Character_VGA.xise	**Parser Errors:**	No Errors
Module Name:	clockmodule	**Implementation State:**	Programming File Generated
Target Device:	xc3s500e-4fg320	**•Errors:**	No Errors
Product Version:	ISE 14.2	**•Warnings:**	26 Warnings (7 new)
Design Goal:	Balanced	**•Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**•Timing Constraints:**	All Constraints Met
Environment:	System Settings	**•Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	211	9,312	2%	
Number of 4 input LUTs	207	9,312	2%	
Number of occupied Slices	258	4,656	5%	
Number of Slices containing only related logic	258	258	100%	
Number of Slices containing unrelated logic	0	258	0%	

Figure 7: Utilization Summary and Project Status

ated. When we press a push button (which stops the update of random number on the screen) on the Nexys 2 board, the we see a random ASCII character on the screen.

The testing of the **VGA Text generation** with Keyboard interfacing is also shown in the figures below.

5 Conclusion

The VHDL code for RSA Encryption/Decryption algorithm is developed block wise. Optimized and Synthesizable VHDL code for each block synthesized using Xilinx ISE 14.2 and verified that functionally correct. The maximum clock frequency is found to be 50 MHz. Since the device require more than 100% resources, it is difficult to implement in FPGA.

Due to time limits we were able to reach Prime numbers generation part of the project where we almost implemented the Miller Rabin Test and Extended Eucledian algorithm. We found the *Modular Arithmatic* part of the project very challenging which is why we weren't able to achieve our final goal. *The incomplete VHDL codes for the remaining modules are also provided.* However, **the purpose of the course project was met as we implemented a fully interfaced VGA controller with keyboard utility which can be thought of as a handy Notepad application. Moreover the random number generator we implemented is one of very robust implementations of PRNG's available.**

Our future plans include not only the completion of this project but also to develop user friendly GUI (adding **SD CARD** interface as well) using which a user will be able to encrypt a file in his SD CARD and decrypt it whenever he wants.

References

[1] Moore, Gordon E. (1965): *Cramming more components onto integrated circuits* Electronics, Volume 38, Number 8

HDL Synthesis Report

Macro Statistics
RAMs : 1
4096x8-bit dual-port RAM : 1
ROMs : 2
16x7-bit ROM : 1
2048x8-bit ROM : 1
Adders/Subtractors : 3
10-bit subtractor : 2
4-bit subtractor : 1
Counters : 4
10-bit up counter : 2
2-bit up counter : 1
21-bit up counter : 1
Registers : 39
1-bit register : 16
11-bit register : 2
12-bit register : 1
3-bit register : 2
4-bit register : 2
8-bit register : 16
Comparators : 10
10-bit comparator greatequal : 2
10-bit comparator greater : 2
10-bit comparator less : 4
5-bit comparator greater : 1
7-bit comparator greater : 1
Multiplexers : 3
1-bit 4-to-1 multiplexer : 1
1-bit 8-to-1 multiplexer : 1
4-bit 4-to-1 multiplexer : 1
Xors : 16
1-bit xor2 : 16

Figure 8: HDL Synthesis Report

Figure 9: Working

Figure 10: Working

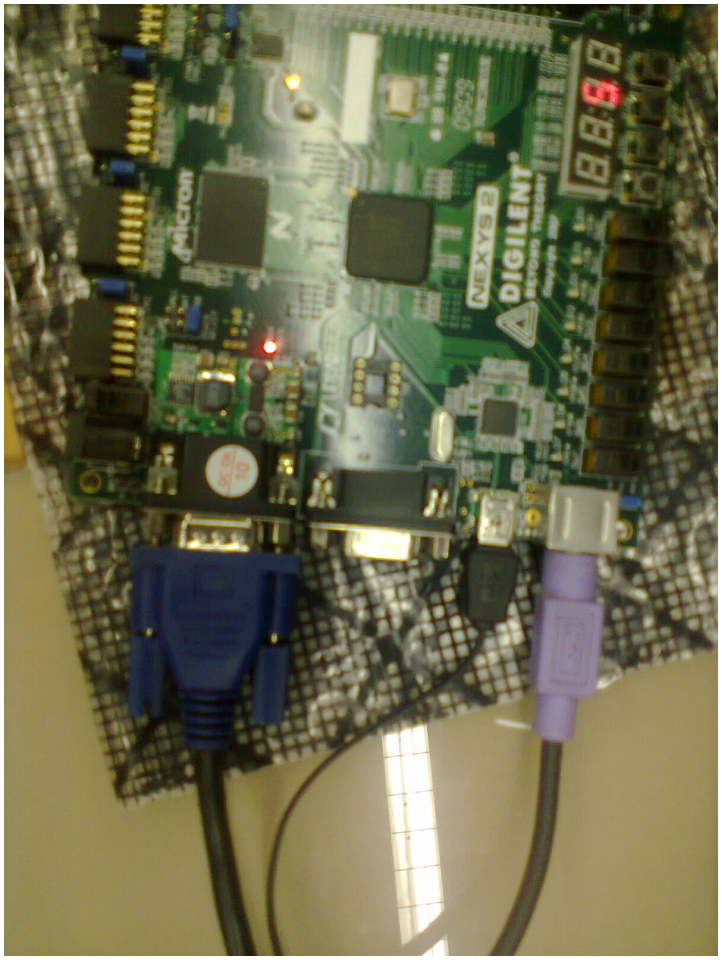

Figure 11: Working

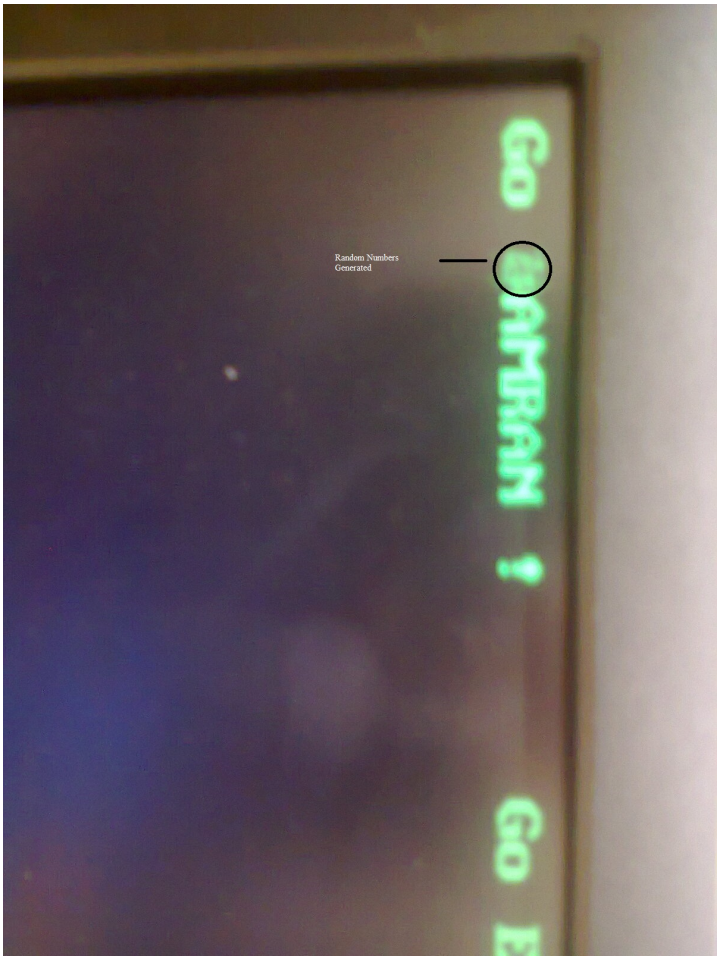

Figure 12: Working

Figure 13: Working

[2] Benedikt Gierlichs: *Hardware-Software Co-Design: A Case Study on an accelerated Implementation of RSA*

[3] Menezes, van Oorshot, Vanstone (1997) *Handbook of applied cryptography, CRC Press*

[4] M.K. Hani, T.S. Lin, N. Shaikh-Husin: *FPGA Implementation of RSA Public-Key Cryptographic Coprocessor* Proceedings of TENCON, vol. 3, pp. 6-11, Kuala Lumpur, Malaysia, 2000

[5] R.L. Rivest, A. Shamir, and L. Adleman (1978): *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems* Communications of the ACM 21,2, pp. 120-126

[6] Ankit Anand, Pushkar Praveen: *Implementation of RSA Algorithm on FPGA* Centre for Development of Advanced Computing, (CDAC) Noida, India

[7] Bruce Land: *people.ece.cornell.edu*

[8] : Modular Exponentiation: *wikipedia, Modular Exponentiation*

[9] : Linear feedback shift register: *wikipedia, Linear feedback shift register*

[10] : Extended Euclidean algorithm: *wikipedia, Extended Euclidean algorithm*

[11] : LFSR: *wikipedia, LFSR*