International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:4, No:3, 2010

Hardware Prototyping of an Efficient Encryption Engine

Muhammad I. Ibrahimy, Mamun B.I. Reaz, Khandaker Asaduzzaman and Sazzad Hussain

***Abstract*—**An approach to develop the FPGA of a flexible key RSA encryption engine that can be used as a standard device in the secured communication system is presented. The VHDL modeling of this RSA encryption engine has the unique characteristics of supporting multiple key sizes, thus can easily be fit into the systems that require different levels of security. A simple nested loop addition and subtraction have been used in order to implement the RSA operation. This has made the processing time faster and used comparatively smaller amount of space in the FPGA. The hardware design is targeted on Altera STRATIX II device and determined that the flexible key RSA encryption engine can be best suited in the device named EP2S30F484C3. The RSA encryption implementation has made use of 13,779 units of logic elements and achieved a clock frequency of 17.77MHz. It has been verified that this RSA encryption engine can perform 32-bit, 256-bit and 1024-bit encryption operation in less than 41.585us, 531.515us and 790.61us respectively.

***Keywords*—**RSA, FPGA, Communication, Security, VHDL.

I. Introduction

THE recent explosion of electronic data communications and computer networks have made it very much important to develop new ways to guarantee their security. With this increasing demand of security in the communication channel, the development of a new and efficient hardware security module has started to get the primary preference.

A vast numbers and wide varieties of works have been done on this particular field of hardware implementation of RSA encryption algorithm. A hardware implementation of RSA encryption scheme has been proposed by Khalil, et al. in [1], where they use Montgomery algorithm with modular multiplication and systolic array architecture. A similar approach has been taken by Kim, et al. in [2]. This design scheme focuses on the implementation of a 1024-bit RSA cryptographic processor. But both these designs have the drawback of a slower processing time, though some of them use a faster clock. Shand, et al. have proposed a software implementation of RSA cryptography in [3]. A different approach has been taken by Chris, et al. in [4] for implementing RSA cryptographic scheme. But, it does not provide the flexibility of using many practical applications as it can only be implemented with a fixed key size.

This work approaches hardware implementation of RSA encryption scheme using the modular exponentiation operation. Simple nested loop addition and subtraction has been used to implement the modular exponentiation operation [4], which helps to reduce the processing time. In this design, it is possible to change the key size of RSA according to the application requirement.

M.I. Ibrahimy, M.B.I. Reaz and S. Hussain are with the International Islamic University Malaysia, 53100 Gombak, Kuala Lumpur, Malaysia (phone: 603-61964504; fax: 603-61964488; e-mail: ibrahimy@iiu.edu.my).

K. Asaduzzaman is with the Multimedia University, 75450 Bukit Beruang, Melaka, Malaysia.

II. Design Overview

An exceptional feature that can be found in the RSA algorithm [5] is that it allows most of the components used in encryption to be re-used in the decryption process, which can minimize the resulting hardware area. In RSA, a plaintext block M is encrypted to a cipher text block C by:

$$C = M^e \bmod n \tag{1}$$

The plaintext block is recovered by:

$$M = C^d \bmod n \tag{2}$$

RSA encryption and decryption are mutual inverses and commutative as shown in equation (1) and (2), due to symmetry in modular arithmetic. One of the potential application for which this design of RSA has been targeted is the Secured Voice Communication. In this application, the voice input is fed into an Analog to Digital Converter (ADC). The ADC changes the continuous analog voice input to a continuous stream of binary bits. The purpose of the compression module in this process is to supply the compressed data to the encryption module for a fast operation. The encryption module takes care of the security. The process at the receiving end is same as the process that has been followed at the sending end except that the sequence of the module is reverse. The encryption engine covers both the operation of Encryption and Decryption.

A. Basic RSA Operations

The RSA algorithm requires computation of the modular exponentiation, which is broken into a series of modular multiplications by the application of exponentiation heuristics. Recall that the RSA encryption process is only the mathematical operation, $c = m^e \bmod n$ [5]. This mathematical operation has involved a few modular operations; *modular exponentiation*, *modular multiplication*, *modular addition*,

and *subtraction* operations on large integers [6]. Detail algorithms of the above operations for hardware implementation have been stated in the following sections.

B. Modulus Exponentiation Operation

The modular exponentiation operation is simply an exponentiation operation where multiplication and squaring operations are modular. The exponentiation heuristics developed for computing M^e are applicable for computing M^e *(mod n)*. In the domain of hardware implementation, an intelligent algorithm is needed in order to reach a higher efficiency. Hence, exponentiation is achieved by performing a number of squaring and multiplications.

Given the integers M, e, and n, the e has to be changed to binary in order to start the algorithm to compute M^e. There are two variations which depend on the direction by which the bits of e are scanned: Left-to-Right (LR) and Right-to- Left (RL). The LR binary method is more widely known which has been listed in pseudo codes shown in Fig. 1 [6].

Left-to-Right Method

Output $C = M^e$
(e contains h-bits)
1. if $e_{h-1} = 1$, *then C:=M else C:=1*
2. for i = h-2 down to 0
2a. $C := C \times C$
2b. if $e_i = 1$, *then* $C := C \times M$
3. return C

Fig. 1 Algorithm for Modulus Exponentiation Operation

Let us assume $e = 43 = 101011_2$. So the $h = 6$ (e contains 6 bits). Using Left-to-Right method, as $e_5 = 1$, $C = M$ algorithm starts as the following table 1.

TABLE I
LR METHOD OF COMPUTING EXPONENTIATION

i	e_i	Step 2a	Step 2b
4	0	M^2	M^2
3	1	$(M^2)^2$	$M^4 \times M$
2	0	$(M^5)^2$	M^{10}
1	1	$(M^{10})^2$	$M^{20} \times M$
0	1	$(M^{21})^2$	$M^{42} \times M = M^{43}$

C. Modulus Multiplication Operation

The modular multiplication problem is defined as the computation of $P = A \times B \pmod n$, given the integers A, B, and n. It is usually assumed that A and B are positive integers with $0 \le A, B < n$.

The modulus multiplication operation is needed after the separation of exponentiation into a number of squaring and multiplication. There are basically four general approaches for computing the product P [1, 2, 3, 6]: Multiply and then divide, Interleaving multiplication and reduction, Brickell's method and Montgomery's method.

All approaches above have a common disadvantage that it doubles up the number of bits for each multiplication. For example, when two 32-bit numbers are multiplied together will cost a 64-bit result and hence a large register is needed to store this result.

A modified algorithm is used in this design which will be discussed in *Section E*. The modified algorithm overcomes the problem by separating the multiplication operation into a number of modular addition operations.

D. Modulus Addition Operation

The modular addition problem is defined as the computation of $S = A + B \pmod n$ given the integers A, B, and n. It is usually assumed that A and B are positive integers with $0 \le A, B < n$. The most common method of computing S is as follows:

1. Compute $S = A + B$.
2. Then $S = S - n$.
3. If $S \ge 0$, then repeat step 2, else $S = S$.

Note that modular addition involves subtraction operation in step 2.

E. Complete Algorithm

The difficult part of RSA encryption/decryption lies on the modulus calculation of $c = m^e \bmod n$, to get the encrypted message "c". To calculate the encrypted message "c", it involves exponentiation that requires large amount of combinational logic, which increases exponentially with the number of bits being multiplied.

The only realistic way to accomplish this is by using a sequential circuit, which realizes the exponentiation as a series of multiplications, and multiplication could also be realized as a series of shifts and conditional additions [4] which has already been illustrated in *Section B* where an exponentiation is separated into a number of multiplications and squaring. Each multiplication can be realized by a series of additions which has been discussed in *Section D*. Note that to reduce the hardware size, modulus (*mod n*) is performed as a number of subtractions inside the multiplication loops. Based on the algorithm described above, a Finite State Machine (FSM) has been developed as shown in Fig. 2. Later, with the reference of this FSM, the VHDL code for the design has been developed.

In conclusion, RSA encryption requires a large loop to perform exponentiation, with a smaller inner loop to perform the multiplication. Within the multiplication loop, additions are used to substitute the multiplication. For each loop in addition, the divisor or modulus is subtracted from the working result whenever the working result becomes larger than the divisor, and leaving the modulus when encryption is done.

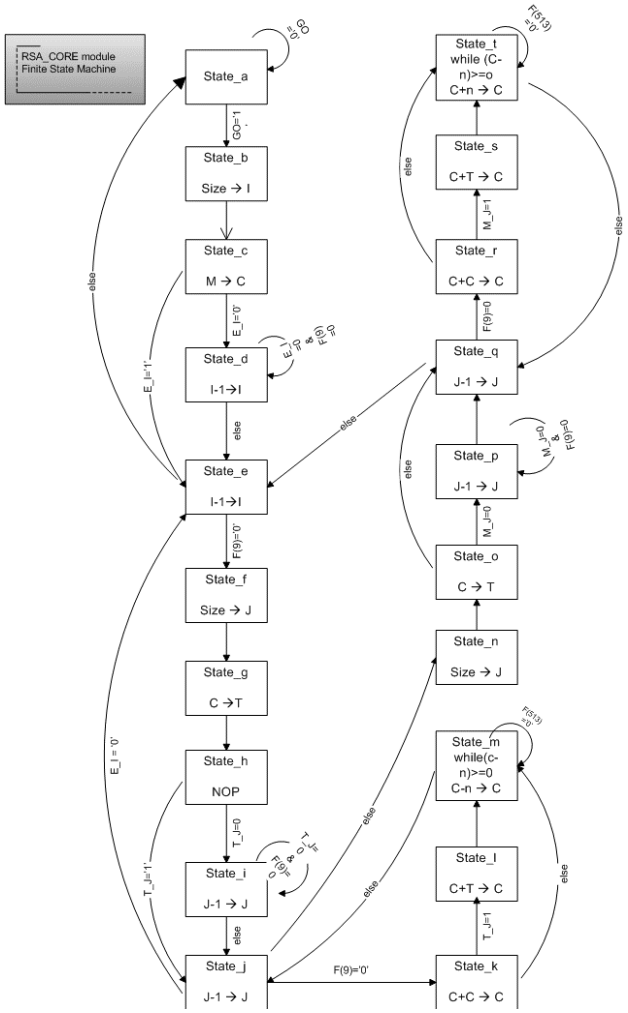

Fig. 2 Finite State Machine (FSM) of RSA Core Module

III. VHDL Modeling

The combined RSA module consists of 5 sub-modules as shown in Fig. 3.

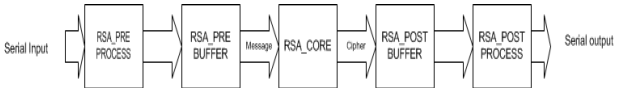

Fig. 3 Overview of RSA Hardware Design

All these modules including with the RSA combine module has been described briefly in the following sections:

A. RSA PREPROCESS Module

RSA_PREPROCESS in Fig. 3 converts serial incoming bits to parallel. Fig. 4 shows the hardware model layout.

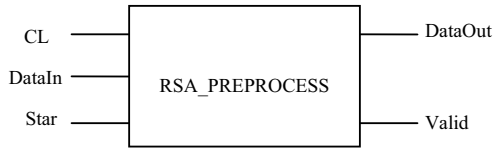

Fig. 4 RSA PREPROCESS Hardware Module Layout

DataIn and *DataOut* are the serial input and parallel output respectively. *Start* pulse starts the conversion process. The *Valid* signal goes high when the conversion is completed.

B. RSA PREBUFFER Module

RSA_PREBUFFER temporarily stores the incoming message block and with a valid signal from the RSA_CORE module, it passes the message block to the CORE module for further process.

C. RSA CORE Module

RSA_CORE module performs the encryption and decryption process. This module has been implemented based on the algorithm listed in the previous section. The VHDL coding of this module is written based on the FSM diagram described in the previous section as well.

An Arithmetic Logic Unit (ALU) has been used in the RSA_CORE module to perform the necessary additions, subtractions and others. Registers have been used to hold the results of the ALU operation temporarily and a state machine described earlier is used to control the operation of the ALU, route the desired signals to the ALU inputs, and latch the ALU output to the desired register. The ALU accepts 32/64/128/512/1024 bits data as input, and produces an output with the same bit length. The input is stored temporarily in a register (34/66/130/514/1026 bits) where arithmetic operations are performed. The result is then moved to the output port when the operation is done. The design uses four registers (each 34/66/130/514/1026 bits) to hold the results, and 2 registers (5/6/7/9/10 bits) to hold the loop variables. The extra 2 bits are used in order to prevent overflowing during add operations. The layout of the RSA_CORE module is shown in Fig. 5.

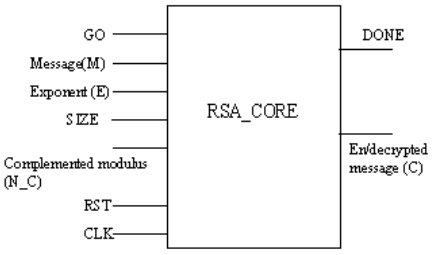

Fig. 5 RSA CORE Module Hardware Layout

In Fig. 5:

- *M* is the plaintext for encryption, or the cipher text for decryption.
- *E* and *N_C* are public key (*e*, *n*) used for encryption, or private key (*d*, *n*) used for decryption.

- *SIZE* provides the key size that is needed for encryption/decryption process.
- *RST* sets the state machine implemented in RSA_CORE architecture to the initial idle state.
- *GO* switches the state machine from idle state to the next state.
- *C* is the cipher text produced by encryption, or the plaintext recovered by decryption.
- *DONE* is high when the encryption or decryption operation is completed, otherwise it is always low.

D. RSA POSTBUFFER Module

RSA_POSTBUFFER module is used to store the incoming message blocks temporarily to avoid data missing. Then the message block transferred to the POSTPROCESS module.

E. RSA POSTPROCESS Module

RSA POSTPROCESS module performs the reverse process of the RSA PREPROCESS module. The module is a parallel to serial converter. The hardware model layout is shown in Fig. 6.

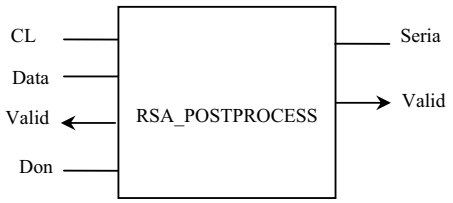

Fig. 6 RSA POSTPROCESS Hardware Module Layout

Here, *DataIn* is a parallel input, and *Serial* is a serial output. With a *Done* pulse, the conversion process starts and complete the conversion with a high *Valid* signal.

F. RSA COMBINE Module

RSA_COMBINE is the combination of 5 different modules: CORE, PREPROCESS, PREBUFFER, POSTBUFFER and POSTPROCESS. These five modules are linked together as illustrated in the Fig. 7.

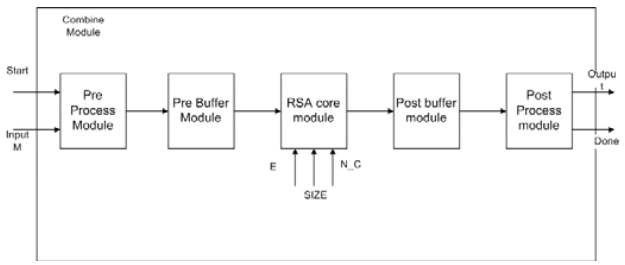

Fig. 7 Design Model for the Combine RSA Module

The VHDL coding of each and every module is coded as a separate entity and finally combined together with a top level design.

IV. Simulation, Synthesis and Discussion

A. Theoretical Result of the Simulation Process

To begin the testing of encryption /decryption process, a set of RSA parameters has calculated. Then the calculated parameters have been fed into the RSA_CORE module and the results have been compared with the theoretical values.

Assumed that $p = 7$, $q = 9$. To generate the two keys, the product is computed as, $n = p \times q = 63$**.** The Euler's Totient function of n is computed as, $\varphi = 48$. Public key e is randomly chosen such that e and φ is relatively prime. Here, $e = 5$ has been chosen. Finally, the Extended Euclid's algorithm is used to compute the decryption key, *d.* The private key for decryption obtained as $d = 29$. Thus, public key = (5, 63), private key = (29).

The encryption process has been started by assuming the message, $M = 7$. Lastly, by using equation 1, the Cipher text C has been calculated as $C = 49$.

B. Simulation Results of the RSA CORE Module

The timing simulation is performed by using the RSA parameters that have been developed earlier in this section. In this part the timing simulation of the RSA_CORE module is performed for 5 different key sizes. For each of the key size following input set is used:

Encryption : Message, $M = 7_{16}$
Public key, $E = 5_{16}$
Modulus, $N = 3F_{16}$
Decryption : Message, $M = 31_{16}$
Public key, $E = 1D_{16}$
Modulus, $N = 3F_{16}$

The simulation result of the Encryption and Decryption process with a key size of 512-bits is shown in Fig. 8.

Fig. 8(a) depicts the encryption process, while Fig. 8(b) shows the corresponding decryption process. The timing simulation is performed with 50ns simulation clock period (20 MHz).

Fig. 8 a) Encryption with 512-bits Key Size

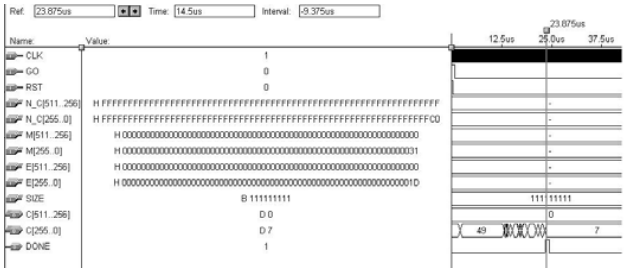

Fig. 8 b) Decryption with 512-bits Key Size

The parameters in Fig. 8 follow the RSA_CORE modules parameters. It can be noticed that all the inputs are in hexadecimal, while the output "C" is in decimal format. The input and output signals are divided into several 256-bits blocks because MaxPlus II only supports 256 bits in one single block. Thus to incorporate with MaxPlus II and to view the simulation result properly, the 512-bits signals are divided into groups. In the system, with the high "GO" signal, the calculation starts. The output "C" is giving the final result when the output "DONE" changed to '1' (high state). According to the mathematical calculations that is presented early in this section, the encryption of a message valued '7' with the encryption key '5' and modulus value '63' giving the result C= '49' . In Fig. 8(a), it can be seen that when the DONE signal goes high, the output signal "C" shows '49'. Thus it can be concluded that the encryption process works successfully. Fig. 8(b) shows the decryption process to get back the original message, '7'. Likewise part (a), this figure is also having all the input and output signals. Here, the input message block "M" is fed with the cipher text that is obtained from Fig. 8(a). The input " E" is changed to the private key, d = '29' ($1D_H$) to carry out the decryption. Original message that is obtained when the "DONE" = '1' (high state) is '7'. By comparing both the figures above, it can be seen that the decryption process is performed successfully as well. The same process is followed to verify the functionality of the system with the key size 32, 64, 128 and 1024-bits. For each key size, the output is verified with the theoretical output and in all instances both the obtained and theoretical output is successfully matched.

C. Simulation Result of the RSA COMBINE Module

The timing simulation of RSA_COMBINE module is performed with the same set of keys and message, which is used earlier. Fig. 9 shows the timing simulation of the encryption process with a 512-bits key size.

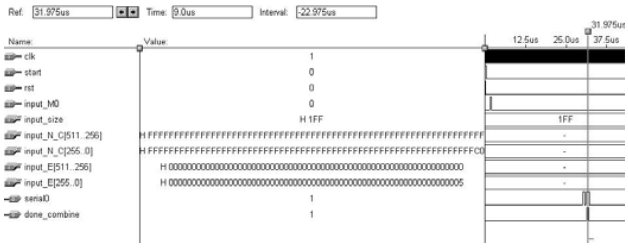

Fig. 9 Encryption Timing Simulation of COMBINE Module

The output obtained here is '49' which is same with calculated value. In binary, $49 = 0110001_2$, which is observed at the output pin *serial*. The pin *done_complete* indicated the encryption was completed. Note that any bits after *done_complete* = '1' is not a part of encrypted result. Thus the functionality of the design tested successfully.

D. Synthesis

The VHDL code is synthesized onto Altera STRATIX II device family. The synthesis tool has chosen the device EP2S30F484C3 for an efficient implementation of the flexible key RSA encryption engine. From the synthesis result, it is found that the device has utilized 13,779 units of Logic Elements (LEs) out of 33,880 units of total available which is about 41% utilization of the chosen device. By comparing this result with the synthesis result in the earlier ones, it is found that the total number of LEs have been decreased quite significantly. All different modules of RSA encryption engine is successfully tested and verified with a frequency of 20 MHz. The RTL view for the combine module is shown in Fig. 10.

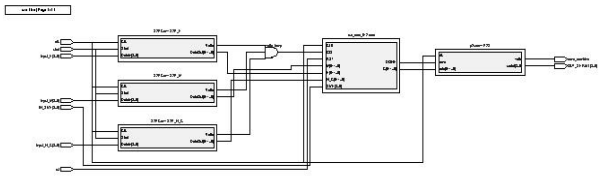

Fig. 10 RTL view of RSA Combine

V. Hardware Implementation

The project is successfully configured and downloaded to the STRATIX II EP2S30F484C3, tested and validated. The following steps are performed:

1. Prepare project for hardware design
2. Set-up
3. Apply power
4. Configure the STRATIX II device
5. Configure the Pattern Generator (PG) and Logic Analyzer (LA) devices
6. Test and verify results

In the set-up, the Altera DSP Development Board is connected to a laptop using an Altera byteblaster download cable. On the other hand, both the PG and LA mainframes are connected to the laptop through a USB cable. All the three devices are then connected together using signal connectors, ground lines, probes and jumper wires. Power is applied to the board by connecting the 5.0-V DC power supply adapter provided in the DSP kit. Once power is applied to the board, the POWER ON LED turns on. The STRATIX II device is directly configured using Quartus II software version 5.0 Programmer. Next, the Joint Test Action Group (JTAG), a type of protocol, configures the STRATIX II device through the download cable via the ByteBlasterMV cable. Among the

16 available channels in PG, 14 are utilized as inputs to the Altera FPGA.

After downloading the project into FPGA, it is not sufficient by only performing software simulations. Thus, PG is used to generate FPGA input label signals and feed them into the chip and capture signals by LA from these chip's output. This verification model reflects the real responses not only from virtual simulation by software, but it is also a real chip working result. Combining PG and LA provides an auto testing system or auto verification system. The ground lines of PG and LA are connected to the tested-circuit ground while the LA and PG grippers are connected according to the order of channel field number. The input wave patterns are then sent from the PG to the tested-circuit and the LA captures the outputs from the tested-circuit.

VI. Results and Discussion

This section presents the simulation and synthesis results of the combined encryption system. The results obtained from the simulation are verified manually to make sure that the components are functionally correct. Simulation is an important process that must be carried out in order to obtain a good design that meets the objective. It enables errors and imperfections to be detected early in the design cycle. After verification process is completed, synthesis is carried out on all the components. The project is the configured and downloaded to the Altera FPGA. The next section presents a comparison between the results obtained in software and hardware implementation. This provides a useful evaluation in terms of the effectiveness and feasibility of the proposed approach.

This RSA encryption engine has been tested and verified to perform 32-bit, 256-bit, and 1024-bit encryption operation in less than 41.585us, 531.515us and 790.61us respectively. This result has given the sufficient ground to claim this particular RSA encryption engine, a faster one than all other previous works [1]. The combine module is complied in the Timing Analyzer to obtain the critical frequency and timing delays. It is observed that the combine module of this Flexible key size RSA encryption engine has obtained a critical frequency of 17.77 MHz. Most importantly, from the synthesis result, it has clearly been observed that the RSA combine module has used 41% of total available LEs of the targeted ALTERA STRATIX II device family. This result resembles that the designed encryption engines takes a smaller space in the targeted FPGA and can be fitted in an FPGA with a smaller capacity. Moreover, the hardware implementation of the encryption has successfully tested and verified.

VII. Comparison between Software and Hardware Implementation

Table 2 presents the results comparison between the two approaches software and hardware implementation. Based on the results obtained, it is concluded that the results obtained from the hardware exactly matched the results obtained from software simulations. In both cases, the system worked as expected and successfully for both encryption and decryption process.

TABLE II
Results of Comparison

Data type	*Software Simulation*	*Hardware Output*
Encryption Key	041_H	041_H
Encryption Input	$2400C5F_H$	$2400C5F_H$
Encryption Result	$241FEBB_H$	$241FEBB_H$
Decryption Input	$241FEBB_H$	$241FEBB_H$
Decryption Key	$013A0C2_H$	$013A0C2_H$
Decryption Result	$2400C5F_H$	$2400C5F_H$

For the case of hardware implementation, it is noticed that for the both encryption and decryption, the process has taken almost the same amount of time as the software simulation. Table 3 below depicts the comparison of processing time take by both software simulation and the hardware implementation.

TABLE III
Comparison of Processing Time

Process	*Software Simulation*	*Hardware Implementation*
Encryption	19.385us	20.3us
Decryption	41.585us	41.6us

From the table above, it can be concluded that the hardware implementation has taken a little longer processing time for both encryption and decryption process. One possible reason of this can be the latency of the real world hardware.

VIII. Conclusion

The primary goal of this research project was to develop a flexible key RSA encryption engine which can be able to provide a significant level of security where it requires as well as can provide a faster processing time where necessary. The maximum bit length for both the public and private key is 1024-bit, which has made this particular RSA encryption engine possible to achieve a significant level of security. Beside the security issue, another major concern of this research project was to provide a faster processing time to the applications where speed gets a better preference than the security.

Overall, this project was successful. All design objectives were met, and the hardware implementation worked as anticipated based on software simulations. The VHDL implementation has shown that the language provides a useful tool of practicing the algorithms without drawings of large amounts of logic gates. And, it is yet another example of how FPGAs can be used with good results for real-world, computationally intensive problems such as RSA encryption algorithm. Although the current highest bit of this flexible key RSA encryption engine can provide a sufficient amount of security, a larger key size can always ensure a better security. But the result of using a larger key causes slower processing time. Thus, some improvements can be made to boost up the speed by including multiplication and division operations in the algorithm.

Acknowledgment

The authors would like to express sincere gratitude to the Research Centre, International Islamic University Malaysia

for providing fund for the research under IIUM Long-Term Research Grant (IIUM/504/RES/G/14/3/01/LT38).

REFERENCES

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

[2] Y.S. Kim, W.S. Kang, J.R. Choi, "Implementation of 1024-bit Modular Processor for RSA Cryptosystem", in Proceedings of Asia-Pasific Conference on ASIC, pp. 187-190, Cheju Island, Korea, 2000.

[3] M. Shand and J. Vuillemin, "Fast Implementation of RSA Cryptography", in Proceedings of 11th IEEE Symposium on Computer Arithmetic, pp. 252-259, Windsor, Ontario, 1993.

[4] C. Brueggen, J. Singh, D. Lord, B. Siever, D. Sullins, "A Hardware Approach to RSA Encryption", Department of Electrical and Computer Engineering University of Missouri-Rolla, pp. 1-14, Citing Internet Sources; URL: http://www.mentor.com/partners/hep/HDLcontest.htm

[5] Rivest, R., Shamir, A., and Adleman, L, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems", Communications of the ACM, 1978, vol. 21, no. 2, pp. 120-126.

[6] C. K. Koc., "RSA Hardware Implementation. Technical Report TR 801", RSA Laboratories, 1996, pp. 1-24.