RSA Cryptography Acceleration for Embedded System

Rolando Duarte Chen Liu Xinwei Niu

Computer Architecture and Microprocessor Engineering Lab (CAMEL)

Department of Electrical and Computer Engineering

Florida International University

Miami, FL 33174, U.S.A.

{rduar002, cliu, xniu001}@fiu.edu

***Abstract*—Cryptography plays an important role for data security and integrity and is widely adopted, especially in embedded systems. On one hand, we want to reduce the computation overhead of cryptography algorithms; on the other hand, we also want to reduce the energy consumption associated with this computation overhead. In this paper, we explore techniques to improve the overall throughput and energy consumption of RSA (Rivest, Shamir and Adleman) public-key cryptography. Instead of implementing the entire algorithm into hardware format, we carefully implemented a custom coprocessor design to accelerate a single hotspot function of RSA algorithm on a Virtex5 FPGA platform. Then, we compare the effectiveness of the coprocessor design against the software implementation of RSA. The hardware accelerates the execution time by 10% thus minimizing the energy by 9%, achieving our goal.**

Keywords-Coprocessor; cryptography; RSA; hardware accelerator.

I. INTRODUCTION

The Internet has evolved so fast that it not only provides information but also permits to communicate and make electronic transactions around the world. Hence, we want our personal data to be secured, reliable, and efficient over the Internet. Many cryptography algorithms have been implemented to prevent intruders from stealing information during an electronic transaction. They are widely used in applications such as ATM cards, computer password, e-mails and even within the world of electronic commerce. As secure communication bandwidth demands continue to grow, it requires faster cryptographic processing. This serves as the motivation for our hardware acceleration approach. Hardware acceleration hastens some specific operations, allowing the overall system, including the general purpose processor and the coprocessor, to execute concurrently in order to achieve performance improvement. The processor assigns specific function to the coprocessor while the processor executes its own instructions. For example, Irwansyah et al. [2] and Hodjat et al. [3] integrated the AES cryptography [4] as a complete system and interfaced it with the microprocessor. This technique is very efficient to accelerate as most as you can to finish executing the process as fast as possible. However, we need to take into consideration that adding more hardware to the system implies that it will consume more power. Clearly, there is a trade-off between performance improvement and power consumption. Also, what if the system requires some other hardware accelerator? Then it can be a critical part since this will incur even more power consumption.

Our design is based on FPGA platform, which allows us to customize our hardware implementation without going through the process of realizing the hardware into a physical chip. However, the resources on the FPGA are limited. If the system is complex enough with multiple accelerators needed and has occupied most of the FPGA resources, we might run out of space and probably will end up using several FPGAs to accommodate the customized hardware. Thus, we want a system that not only executes faster but also consumes less power and takes less space or resources in the FPGA platform. Nuan et al. [11], Hani et al. [12] and Zutter et al. [13] focused on speeding up the RSA [1] cryptography core by enhancing the modular exponentiation of square and multiplication. However, they implemented the whole RSA algorithm as a coprocessor. We followed a similar path, but instead of implementing the RSA cryptography core in its entirety, we selectively implemented only a single hardware accelerator targeted at a single hotspot function of the algorithm. We employed hardware in the form of a customized IP that accelerates the computation, so that we can observe a performance improvement when we execute the software code with the new integrated hardware.

The rest of the paper is organized as following: Section II will cover the system architecture of our design, which describes all the components that are used during the implementation of the hardware accelerator. This section also provides information why we chose to accelerate specific function as a custom IP and how it interfaces with the microprocessor. Section III presents the experiment results we obtained when we added the new peripheral to our system. Finally, the conclusion is drawn in Section IV.

II. SYSTEM ARCHITECTURE

FPGA are widely used because its reconfigurability and reprogrammability can meet the different needs of the user. Thus, we chose FPGA because of its flexibility to add a hardware component into the device and its ease to reprogram the device. We could implement the same design using a simulator-based approach. However, we believe FPGA

platform could generate more accurate performance and energy consumption readings. We used the Virtex5 FPGA board [7] to interface our IP with Microblaze processor [8]. The overall system architecture is shown in Figure 1. Our system contains the following microprocessor, peripherals, and buses for intercommunication: Microblaze, BRAM, Local Memory Bus (LMB), Processor Local Bus (PLB), RS232, Timer, Interrupt and our customized accelerator. Microblaze is a soft-core microprocessor and it is of RISC architecture optimized for Xilinx FPGA boards. Microblaze is the only soft-core processor Virtex5 supports and it runs at 125 MHz. This processor is responsible for the execution of all instructions and communication among peripherals. BRAM is a Block RAM memory system with 64 KB memory space and the main purpose of this peripheral is to hold all instructions and data to be executed during the process. Microblaze access either instruction through ILMB or data through DLMB. These two buses are 32-bit wide. The LMBs are only connected to the Microblaze because it is the only component responsible for executing instruction and its data. The PLB provides communication between Microblaze and all the peripherals. If any peripheral needs to access another peripheral, PLB is the one accountable for this communication as well. RS232 is in charge of receiving and sending data to the user via HyperTerminal. Thus, we use RS232 to verify the results of RSA encryption/decryption between with acceleration and without acceleration approaches. We use a timer to gather information about how many clock cycles certain process takes. The interrupt peripheral is needed since we want our process to be interruptible because in an event that Microblaze needs to execute an instruction with higher priority, then any other peripheral can be interrupted. Finally, we have *Power_HW(2, n)* IP, which is our customized accelerator. It requires 12 slices and 133 LUTs. Thus adding the custom hardware, as shown in Figure 1, increased the usage of overall FPGA resource by 6% with respect to the hardware platform without acceleration. Microblaze will determine when our customized peripheral will be used and will be responsible for collecting the data provided by the customized hardware and sending the data to other peripherals connected through the same PLB bus.

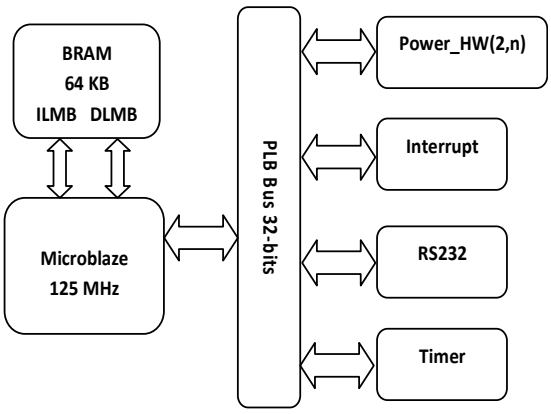

Figure 1. Power_HW IP Interfacing Microblaze

Since FPGA is reconfigurable, Microblaze can be specified to run up to 125 MHz, its maximum speed. Moreover, any peripheral connected through PLB has dual-port communication, meaning that each peripheral can send data to and receive data from the PLB. Then, if any other peripheral or another customized hardware should be connected, it would be listed at the right side of the bus like the other peripherals. It also should be kept in mind that the PLB runs at the same speed as Microblaze to keep reliability consistent. Hence, our customized IP will also run at 125 MHz since it is also connected to the PLB.

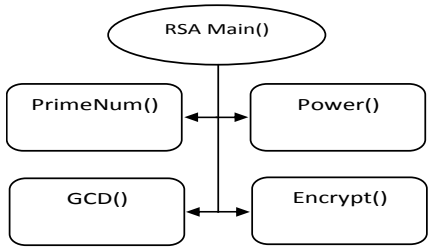

Figure 2. RSA Functions

The key point of our design is to identify which function executes most of the time and how long the entire process takes. The RSA software code in this specific design has four functions which are *PrimeNum()*, *GCD()*, *Encrypt()* and *Power()*, as shown in Figure 2. *PrimeNum()* provides a prime number every time it is called. RSA uses the multiplication of two prime numbers to decrypt and encrypt the data. The *GCD()* function determines the greatest common divisor since we know that two numbers are coprime if their greatest common divisor equals 1. This function is used in the process of generating the public key and private key. The *Encrypt()* method takes on the core mathematical operation of RSA algorithm, which is to calculate X to the power of Y modulo N. It performs either the encryption of the original data to convert it into a cipher data or the decryption of the cipher data to produce back the original data, depending on the parameter. *Power()* calculates 2 to the power of n, where n = 0, 1,…, 31. Based on the specific software implementation we use in this design, *Power()* actually is called by *Encrypt()* and its main functionality is to prepare the data into a specific format in order to calculate X to the power of Y modulo N. In [9], Chang et al. profiled the RSA algorithm using Intel® VTune™ Performance Analyzer [5] to gather information about which part of the software code takes most of the execution time, defined as hotspot function. They indicated *Power()* function as the hotspot function for RSA. Thus, we implemented a hardware accelerator that performs the same operation as software function *Power()* but the only difference is that our customized hardware will complete its task in fewer clock cycles. Consequently, it requires less execution time as we will see in the next section. Realizing all the RSA functions in hardware implies that the overall execution time will be faster. However, this will increase the usage of system resource, thus increasing the total power consumption. Our focus is to achieve the best speedup for the

overall system while minimizing the power and energy consumption, which naturally leads to the hotspot function acceleration.

III. Experimental Results

The experiment consists of encrypting and decrypting 32-bit (4-byte) data. At first, the pure software code is executed without the existence of the customized hardware, and we observe that it takes approximately 39100 clock cycles to finish the process of encrypting and decrypting a 4-byte data, as shown in Table I. Since, Microblaze runs at 125 MHz, it takes about 0.31 ms to execute the entire process. Then, we followed the same process of encrypting/decrypting data, but each time the data size is incremented by multiple of ten to be consistent. Thus, the test set consists of data size from 4, 40, and all the way to 40000 bytes. As the data size increases, the number of clock cycles hence the execution time increases accordingly. They keep almost a constant relationship since the number of cycles for each data encryption/decryption is the same. The data-flow of the RSA software code is shown in Figure 3.

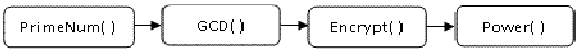

Figure 3. RSA Flow _ without Acceleration

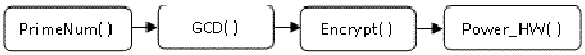

Figure 4. RSA Flow _ with Acceleration

Next, the Microblaze interfaces with our customized IP which is embedded into the FPGA. Partial of the RSA software code is now replaced by the corresponding hardware part. Here, the native software function *Power()* that computes 2 to the N*th* power is replaced by a coprocessor *Power_HW()*. The FPGA system is a memory-mapped system, thus each peripheral or embedded IP [6] is accessed by providing its corresponding memory address location. Hence, the *Power_HW()* is called upon by providing its memory location and the integer N. We execute the modified RSA software code to call the *Power_HW( )* instead, as shown in Figure 4. Now, the same data is loaded to *Power_HW()* as the one we used to call native software function *Power()*. The observation is that encrypting/decrypting 4 bytes of data takes 35000 clock cycles as shown in Table II. This leads to an execution time of 0.28 ms. The overhead of accessing the hardware accelerator through the PLB bus is 7 clock cycles plus 2 to 3 cycles to perform a load or store operation. The *Power_HW()* takes about 58 clock cycles to perform the operation of two to the power of N. RSA pure software function *Power()* takes about 118 clock cycle to finish the same execution and return the result. Hence, the custom IP performs 2 times faster than its pure software counterpart. If we compare the execution time of pure software approach with the hardware accelerator approach, we can examine that the overall speedup of our customize hardware is more than 10%, and as we increase the input size data, the speedup we achieves converges to a constant reading of 10.58%, as illustrated in Figure 5. We can apply Amdahl's Law [14] to verify the speedup we obtained while adding the custom hardware. Based on our observation, 22% of total execution time of RSA code is spent on *Power()* function, which is converted into hardware and this conversion acquires a speedup of 2, then we can see that the overall speedup = $\frac{1}{(1-p)+\,p/s}$, where p = 0.2 and s = 2. Then, the overall speedup is 1.1235, which also means the ideal speedup that can be obtained is 12.35%. We obtained an overall speedup of 10.58%. But we need to take into consideration that the timer is also connected to the PLB bus; thus it takes about 7 cycles to access the timer over the PLB bus. Therefore, this affects the overall execution time of the system and the overall speedup we achieve.

We utilized XPower Analyzer [10] to get the power consumption of the system. Xilinx claims that XPower Analyzer provides accurate power analysis after design implementation. The power consumption for the hardware system without the customized IP is 1. 2519 Watts and with the added IP is 1.2653 Watts, a 1% increase. Table I and Table II show the number of clock cycles and execution time of the two different implementations. Since we know the power consumed, we can calculate the energy for the RSA without customized IP approach and with the customized IP approach respectively. From these two tables, we can observe that our design executed the RSA algorithm effectively and the energy consumption is reduced after adding the embedded peripheral. Figure 6 shows the energy reduction of our hardware design over software. Moreover, the energy reduction converges to a constant reading of 9.62% with increased input data size. Hence, our hardware acceleration design not only gained speedup but also reduced the energy consumption. Figure 7 illustrates the normalized energy consumed per byte over the 4-byte input case, based on the data from Table II. We examined that the energy consumption per byte decreased as the data size increased.

TABLE I. RSA WITHOUT ACCELERATION

# of Bytes	Clk Cycles (10^6)	Exec Time (ms)	Energy (mJoules)	uJoules / Byte
4	0.0391	0.3127	0.3915	97.8795
40	0.3905	3.1241	3.9110	97.7756
400	3.8764	31.0109	38.8219	97.0547
4000	38.8182	310.5456	388.7658	97.1915
40000	388.2355	3105.8842	3888.1944	97.2049

TABLE II. RSA WITH ACCELERATION

# of Bytes	Clk Cycles (10^6)	Exec Time (ms)	Energy (mJoules)	uJoules / Byte
4	0.0350	0.2799	0.3541	88.5358
40	0.3494	2.7955	3.5372	88.4308
400	3.4656	27.7245	35.0809	87.7022
4000	34.7102	277.6816	351.3616	87.8404
40000	347.1555	2777.2442	3514.1582	87.8540

However, the energy consumption converges if data size is over 1 Kbytes. This is due to the factor that our system speedup also converges once the data size is above 1 Kbytes. Therefore, we can estimate how long it can take to encrypt and decrypt a larger data size of 400 Kbytes, 4 Mbytes, or even more. Basically we would expect the same speedup and energy reduction as shown in Figure 5 and Figure 6 respectively. If input data set is greater than BRAM capacity, it is required to put it into off-chip memory. In this case we employ a 256MB DDR memory. It takes 1074225 and 916194 clock cycles to encrypt/decrypt 4-byte data with and without acceleration respectively. If we compare with the BRAM case, BRAM-based design is more than 26 times faster than DDR-based design in both scenarios.

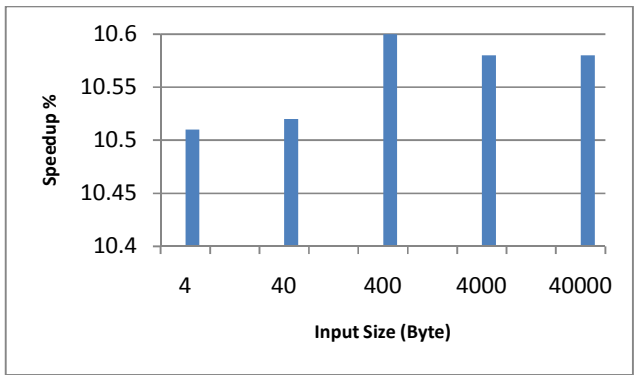

Figure 5. Speedup of with-Acceleration over without-Acceleration

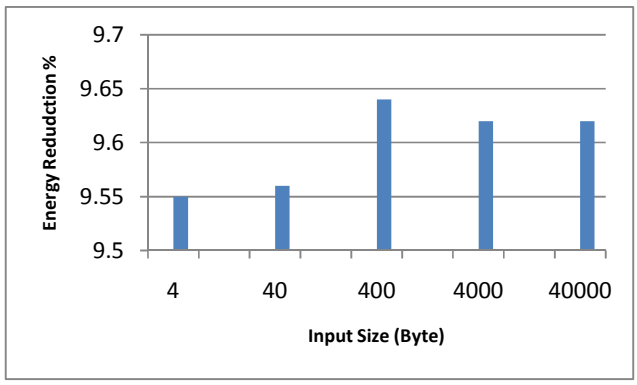

Figure 6. Energy Reduction of with-Acceleration over without-Acceleration

IV. Conclusion

The implementation of an entire software algorithm into hardware is not always the best choice since we lose the reconfigurability and reprogrammability. We do not just want to accelerate our process but also want to reduce the energy consumption. In this paper, we explore the technique of identifying the hotspot function of a program and then realizing it as a hardware accelerator using RSA cryptography as an example design. The coprocessor helped the system to execute the specific function while the main processor executed the remaining of the code. We achieve an overall speedup of more than 10% and reduced its energy consumption by more than 9%. This technique can be implemented in other systems to explore ways of minimizing the hardware overhead and energy consumption as to maximizing its overall throughput. For future work, we are going to explore the hotspot functions for AES, Blowfish, MD5, 3DES and IDEA cryptography and implement them as customized hardware so that we can compare their execution time and energy reduction the same way we accomplished for the RSA cryptography.

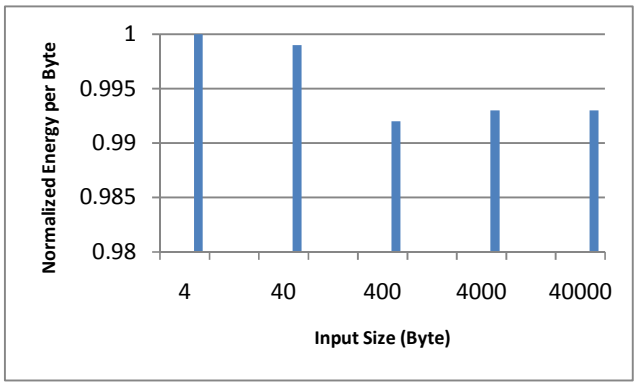

Figure 7. Normalized Energy per Byte Consumption with-Acceleration

References

[1] Introduction of RSA for public-key cryptography. [Online]. Available: http://en.wikipedia.org/wiki/RSA

[2] Arif Irwansyah, Vishnu P. Nambiar, and Mohamed Khalil-Hani, "An AES Tightly Coupled Hardware Accelerator in an FPGA-based Embedded Processor Core," *Proc. ICCET'09*, vol. 02, p. 521-525, 2009.

[3] SAlireza Hodjat, and Ingrid Verbauwhede, "Interfacing a High Speed Crypto Accelerator to an Embedded CPU," *Asilomar SSC'04,* vol. 1, p. 488-492, Nov. 2004.

[4] Announcing the Advanced Encryption Standard (AES). [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[5] Intel® VTune™ Performance Analyzer. [Online]. Available: http://software.intel.com/en-us/intel-vtune/

[6] Paolo Giusto and Grant Martin, "Reliable Estimation of Execution Time of Embedded Software," *Proc. DATE'01,* p. 580-588, Mar. 2001.

[7] MicroBlaze Processor Reference Guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[8] Virtex-5 FPGA User Guide. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[9] Jed Chang, Chen Liu, Shaoshan Liu and Jean-Luc Gaudiot, "The Performance Analysis and Hardware Acceleration of Crypto-Computations for Enhanced Security," *PRDC'2010,* (accepted).

[10] Xilinx Xpower Analyzer [Online]. Available: http://www.xilinx.com/products/design_tools/logic_design/verification/xpower_an.htm.

[11] Wen Nuan, Dai Zi Bin, and Shang Yong Fu, "FPGA Implementation of Alterable Parameters RSA Public-Key Cryptographic Coprocessor," ASIC'05, vol. 2, p. 769-773, 2005.

[12] Mohamed Khalil Hani, Tan Siang Lin, and Nasir Shaikh-Husin, "FPGA Implementation of RSA Public-Key Cryptographic Coprocessor," TENCON'00, vol.3, p. 6-11, 2000.

[13] Jan Zutter, Max Thalmaier, Martin Klein, and Karsten-Olaf Laux, "Acceleration of RSA Cryptographic Operations using FPGA Technology," DEXA'09, vol. 1, p. 20-25, 2009.

[14] The description and explanation of Amdahl's Law. [Online]. Available: http://en.wikipedia.org/wiki/Amdahl's_law