AES Encryption Core with Microblaze on Xilinx Nexys A7: Performance and Benchmarking

Jeffrey Hymas, John Wilson, Raymond Banda, Kyle Gerfen, Freddy Estrada, and Dr. Mohamad El-Hadedy
jmhymas@cpp.edu, johnwilson@cpp.edu, rjbanda@cpp.edu, festrada@cpp.edu, ktgerfen@cpp.edu, mealy@cpp.edu

***Abstract*— This paper presents a performance evaluation of a 128-bit version Advanced Encryption Standard encryption core running on the Xilinx MicroBlaze 32-bit soft-core processor implemented on a Nexys A7-100T FPGA development board. Benchmarking was performed by writing a C++ software application to feed the AES core an encryption key, plain-text, and load signal. From here the encrypted cypher text can be retrieved and processed. Performance metrics used for the performance evaluation on the AES encryption core includes encryption execution time, I/O time, cycles per instruction, resource usage, and power usage.**

***Index Terms*— AES, AXI, Cypher, ECE, Encryption, Symmetric Key Algorithm, IP, FPGA, HDL, RISC, System Verilog, Verilog.**

I. Introduction

MicroBlaze is a RISC Harvard architecture soft-core processor by Xilinx designed for Xilinx FPGAs. It is implemented and configured entirely on the FPGA logic fabric with the ability to select a combination of peripherals, memory, and interfaces needed by the design. Xilinx software tools provide preset configurations for the MicroBlaze which can be configured as a microcontroller unit, real-time processor, or an application processor with a memory management unit to run a Linux operating system. This allows the MicroBlaze to be used for different designs and can be optimized for embedded applications. It also supports an optional floating-point unit, instruction and data cache, as well as support for an Advanced eXtensible Interface (AXI) communication protocol.

AXI is an advanced microcontroller bus architecture introduced by ARM in 1996. It is designed for on chip interconnect for the connection and management of a CPU in system-on-a-chip designs. It has grown and developed over the years with its second version, known as AXI4, released in 2010. There are three types of AXI4 interface, AXI4-Full, AXI4-Lite, and AXI4-Stream. AXI-Full is for memory mapped interfaces and can handle bursts of up to 256 data transfer cycles with a single address phase. AXI-Lite is a lightweight single transaction memory mapped interface that handles 32-bit transaction sizes. AXI-Stream allows unlimited data bursts; however, interfaces and transfers do not have address phases and are not considered memory-mapped. The AXI protocol allows an interface between a single AXI master which can represent the processing system and a single AXI slave representing an intellectual property (IP) core. Both the AXI4 and AXI4-Lite interfaces consist of five different channels, read address, write address, read data, write data, and write response.

The purpose of this paper is to present a performance evaluation on an Advanced Encryption Standard (AES) IP core. AES is one of the most popular encryption standards and was developed by two cryptographers: Joan Deamen and Vincent Rijmen. It is a symmetric block cipher which uses the same secret key for both encryption and decryption with three different block cipher variations based on the key length AES-128, AES-192, and AES-256. The AES cipher uses a sequence of mathematical operations starting from the secret key and the plain text known as “round keys”. Each round key begins with byte substitution, where the input bytes are substituted from a fixed lookup table known as an S-box table, which results in a 4x4 matrix. The rows from the matrix are then shifted using a ‘shift rows’ algorithm, followed by a ‘mix columns’ algorithm. Finally, the bytes of the matrix are XORed to the round key. This process is repeated until the final round which outputs the ciphertext. The number of rounds depend on the key length, AES-128 uses 10 rounds, 12 rounds are used for AES-192, and 14 rounds for AES-256.

In order to benchmark an AES encryption algorithm running on the MicroBlaze, an AES encryption algorithm had to be implemented on the FPGA logic fabric using a hardware description language (HDL). With the AES encryption algorithm written in Verilog, it was then tested, verified and packaged into an AXI4-Lite peripheral IP core for the purpose of interfacing with an AXI interconnect between the MicroBlaze processing system and the AES encryption IP core. The AES core was written in Verilog HDL and was modified from an existing source to work best with acquiring data for performance metrics. The benchmarking test includes performance metrics such as the I/O time to load the AES encryption key, plain-text, and execution time retrieving the final the cypher text.

II. Related work

An FPGA is a mesh of logical blocks that can be reconfigured to the user's needs. The FPGA platform is an attractive choice for data encryption due to its modest power consumption. Its configuration makes it well suited for large, non-complex data sets as opposed to CPUs/ASICs which may be better applied to smaller, more complex data sets. However, it is possible to synthesize a processor on an FPGA board. This processor synthesis is called a 'soft-core processor' and it provides certain advantages and disadvantages for encryption when compared to a purely hard processor or purely software. [1]

Considerable testing has been done in the area of cryptography with FPGAs. There are symmetric, asymmetric and hybrid encryption algorithms. Symmetric method uses a single key for encryption and decryption. Popular symmetric encryption methods include AES (Advanced Encryption Standard), 3DES (Triple Data Encryption Standard) and Twofish. Asymmetric method uses separate keys for encryption and decryption. Popular asymmetric encryption methods include RSA and ECC (Elliptic Curve Cryptography). The chosen method for this project is the Advanced Encryption Standard running on the MicroBlaze soft-core processor. Using the soft-core processor provides the advantage of flexibility, however it performs significantly slower than an ASIC platform. [2] Comparing the AES algorithm to other symmetric encryption methods, AES (running in ECB mode, used for testing with the MicroBlaze) performs faster than 3DES while Twofish is the fastest of the three symmetric algorithms compared here. [3] It should be noted that the AES algorithm in our tests was in ECB mode which is the simplest of the five available [4] modes for AES encryption which improves speed. Asymmetric algorithm performance can vary depending on the data set and the encryption end points, therefore it is difficult to compare these algorithms with AES in the scope this analysis.

III. Design

A. Design Overview

There are three primary components involved in this design: the soft-core processor, the communication protocol, and the encryption core. Since the project is deployed on a Xilinx Artix-7 chip the Microblaze soft-core processor was an obvious choice. It is developed and supported by Xilinx and is designed to run on a variety of their chips, including the Atrix-7. The AES core presented much more of a challenge.

Due to time constraints it was decided early in the project that an open source core would be used. The goal of this project was to benchmark an AES encryption protocol running on the FPGA, not design an encryption core from scratch. Several cores were considered during the research phase of the project, with each presenting its own challenges. Many cores were incompatible with the Artix-7 and would have required extensive modifications to make compatible. Other issues arose around the designs of the cores themselves, many of which were intended for simulation and test benching but not actual implementation. These cores would also require extension modifications to make compatible. After extensive research, an AES core developed by Rudolf Usselmann [5], and published on GitHub, was chosen. While still requiring modification, the design of the core was fundamentally compatible with the project and could be easily packaged into an IP core.

Since MicroBlaze is designed to be compatible with Xilinx's AXI communication protocols, the choice to use AXI was an easy one, but the type of AXI protocol still had to be selected. The core being utilized required data to be transferred to it in 128-bit blocks, so AXI-Stream was easily eliminated as an option. AXI-Full was an attractive choice, especially with its ability to send data in 256-bit bursts, but was ultimately eliminated due to its increased resource requirements and the fact that many of its capabilities weren't needed for a project of this scale. AXI-Lite allowed for simple, straightforward memory mapped communication, and can send 32-bits at a time, meaning that 128-bit inputs and outputs required by the AES core could be broken up into 4 chunks of 32-bits for easy communication.

B. AES CORE

As can be seen in appendix A figure 1, the AES core requires three inputs and has two outputs. The inputs are *id* (1-bit), *key* (128-bit), and *text_in* (128-bit). The outputs are *text_out* (128-bit), and *done* (1-bit). Since the core requires 128-bit input and output, and AXI-Lite only supports 32 bit transaction sizes, some considerations had to be made when packaging the IP into an AXI peripheral, but that will be discussed in a later section. The important signals for this section are *id* and *done.* The *id* signal is used to indicate when the text and key have been loaded, and to begin the encryption process. Simply put, when *id* is high the inputs have been loaded, and when it drops low the core starts the encryption process. The *done* signal is set when the encryption is complete. Encryption takes exactly 12 cycles to complete, as can be seen in the timing diagram in appendix A figure 2, and this presented a problem. As the timing diagram shows, the *done* signal only stays high for that single clock cycle, and the output is only right for that one clock cycle as well (after that clock cycle the output continues to get scrambled and becomes useless). Twelve clock cycles is faster than the AXI4-Lite communication protocol, which requires 17 clock cycles to read and 15 clock cycles to write. To resolve this issue the core was modified to hold its output signals until *id* is reasserted. The code with the changes can be found in the file *aes_cipher_top.v.*

C. Packaging the AES Core

As was mentioned earlier, the AES core requires 128-bit inputs and outputs, but the AXI4-Lite protocol uses 32-bit transaction sizes. To compensate for this the *text_in*, *text_out,* and *key* signals are mapped to four registers, each inside the AXI IP packager. In the case of *text_in* and *key,* the four registers associated with each variable are concatenated together to form a signal 128-bit signal that can be understood by the core. The *text_out* signal is broken into four registers, retrieved by the processor, and reassembled on the other side. The *id* and *done* signals are simply mapped through one register each.

D. Debugging, Timing, and External Communication

Besides the core components that make up the primary functionality, several other AXI peripherals were also utilized. The counter mode implemented in the AXI Timer core was used to determined total clock cycles and execution time of various functions of the AES core. The AXI Uartlite core was used for debugging/communication and the AXI GPIO Core was used to set LEDs on the FGPA board when the encryption process was complete (this was used primarily for gathering test data related to power consumption on the board).

E. Final Design

The final design of the system can be seen in appendix B. Vivado's IP integrator was used to route and tie all the cores together with the MicroBlaze via the AXI4-Lite communication protocol. The entire system runs on the 100Mhz clock native to the Atrix-7 chip.

IV. Results/Evaluation

In this next section, described is the performance evaluation and the results of the implementation of this AES design. Encryption execution time, I/O time, cycles per instruction, resource usage, and power usage were the key performance matrices used to determine the effectiveness and efficiency of the project.

To measure these matrices, several functions were written. First, *TimerCounter* was developed to count at a frequency corresponding to the number of clock cycles in a process. This counter is employed by *start_timer(),* initialized at 0 with its value returned via *time0*. Similarly, *stop_timer()* is used to stop the counter and return its value via *time1*. Once these values are placed into their corresponding variables, they are loaded into a *printf()* statement were the CPU time is computed employing the formula mentioned below[8]:

$$\text{CPU Time} = \frac{Number\ of\ Clock\ Cycles}{Frequency\ x\ 1{,}000{,}000}$$

Where,

Number of clock cycles = (time1 - time0)

With the method for timing explained, how execution time or CPU time is determined can be clearly described. Execution time is the time spent by a system executing a given task. [8] This design sought out the encryption execution time, which was initiated by calling the function *start_timer()*. Next, *input_key()* is called to load the four 32-bit register data blocks to be encrypted.

While still being measured by the timer and within a while-loop counting to 6,250,000, *input_plaintext()* is called to load the *INPUT_REG* registers with the data targeted for encryption. Initiated by calling the function *start_encryption()*, the loaded data is encrypted over a total of twelve clock cycles. The *set_load_low()* function then drops the load register to zero. Next, *read_output_cypher()* is called where the data is again placed into four 32-bit key registers and cyphered.

The execution time of this process of encrypting various sizes of data using this 128-bit input design is computed as described and determined as follows in Table 1:

Size of Encrypted Data (TB)	Execution Time (seconds)
0.0001	20.54
0.001	205.4
1	205400

Table 1

In addition to execution time, this project employs I/O time as another performance metric. I/O time is the input output operations per second. *Table 2* summarizes these findings for this AES algorithm.

Function	Size (bits)	Time (μs)	Number of Instructions
Input Key	128	1.78	1
Input Plain Text	128	1.36	1
Load Input Data	128	0.68	2
Recording Data/Receiving Cypher Text	128	1.12	1
Other Data	-	-	4

Table 2

The next metric used was cycles per instruction. Cycles per instruction or CPI is the average number of clock cycles each instruction takes to execute. [8] The *input_key* measured a total of 178 clock cycles; however, an additional 42 clock cycles are added due to the process's initialization. The function *input_plaintext()* totaled 136 clock cycles at 34 cycles per 32-bits while *load_input_data()* totaled 68 clock cycles, 34 clock cycles to set load low and 34 to set load high.

Also, recording data/receiving cyphered encrypted text totaled 112 clock cycles at 28 clock cycles per 28-bits. Other data included four instructions per loop. Finally, cycles per instruction were measured using the timer function, and *Table 3* summarizes this project's findings.

Function	Size (bits)	Number of Clock Cycles	Number of Instructions
Input Key	128	178	1
Input Plain Text	128	136	1
Load Input Data	128	68	2
Recording Data/Receiving Cypher Text	128	112	1
Other Data	-	-	4

Table 3

Another performance metric used in this project is resource usage. Resource utilization is the measure of how busy various resources of a system are during a process's execution. The AES resource utilization is summarized in *Image 1* below:

Resource	Utilization	Available	Utilization %
LUT	4359	63400	6.88
LUTRAM	139	19000	0.73
FF	4649	126800	3.67
BRAM	39	135	28.89
DSP	6	240	2.50
IO	20	210	9.52
BUFG	5	32	15.63
MMCM	1	6	16.67

Image 1

In addition to cycles per instruction, power usage was another metric used to determine performance and summarized in *Table 4*.

Data Size (MB)	Energy (Wh)	Power (Watts)	Charge (mAh)	Current(Amps)	Voltage (Volts)
100	0.01	1.24	1	0.23	5.24
1000	0.08	-	14	-	-

Table 4

V. Conclusion/Future Work

With the widespread adoption of AES as the standard for encrypting data, it is important to pursue new methods of implementing the encryption algorithm on low power devices. By bench marking the performance of this implementation and documenting the resources used, it can be determined whether it is a good candidate for certain applications. A key point of interest in the emerging IoT landscape is data encryption. IoT devices are a key target vector for malicious attacks and encrypting the data on these devices, many of which are battery powered with limited computing resources, is a growing concern for many industries. Having key performance metrics available for system integrators is a necessary part of developing computer hardware. Looking at current available benchmarks for comparable systems can help inform hardware developers about what they can do to optimize the efficiency of a system.

Moving forward, there are several design implementations the team is considering to advance the project. Several design changes have been considered to speed up the encryption process and allow for a stream of data to pass through the system. By integrating serial communication (I2C or RS-232), the system could take in a stream of data to be processed. To handle the incoming data, FIFO buffers on the input and output would be needed to parse the data into 128-bit blocks to be processed and returned without losing data while waiting for it to be processed. To process large amounts of data quickly, multiple soft-core processors on one device could be used to further streamline encryption and transmission of data.

Once data streaming and faster processing are handled, it would be possible to create a point-to-point system using multiple FPGA development boards to test and benchmark the results. Considerations regarding on-board resources would have to be made to implement these design changes. As it stands, the Artix-7 100t would not be an appropriate board for experimentation because of resource and memory restrictions.

VI. References

[1] U. Farooq and M. Faisal Aslam, "Comparative Analysis of Different AES Implementation Techniques for Efficient Resource Usage and Better Performance of an FPGA," 2017.

[2] V. J. V. Kanhiroth, "Embedded processors on FPGA: Hard-core vs Soft-core," Allendale, 2017.

[3] P. Townsend, "Townsend Security," 25 March 2019. [Online]. Available: https://info.townsendsecurity.com/rsa-vs-aes-encryption-a-primer. [Accessed 1 December 2020].

[4] P. B. Ghewari, J. K. Patil and A. B. Chougule, "Efficient Hardware Design and Implementation of AES Cryptosystem," 2010.

[5] R. Usselmann, "GitHub - Freecore - AES_core," 12 Novermber 2002. [Online]. Available: https://github.com/freecores/aes_core. [Accessed 02 Dec 2002].

[6] R. Usselmann, 12 November 2002. [Online]. Available: https://github.com/freecores/aes_core/blob/master/doc/aes.pdf. [Accessed 02 December 2020].

[7] A. Elbirt, W. Yip, B. Chetwynd and C. Paar, "An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," Worcester.

[8] K. Pal Singh and D. Shiwani, "An Efficient Hardware design and Implementation of Advanced Encryption Standard (AES) Algorithm," 2016.

[9] S. Devi, "AES Encryption and Decryption Standards," *Institute of Physics Journals,* vol. 1228, p. 10, 2019.

VII. APPENDIX

A. AES CORE

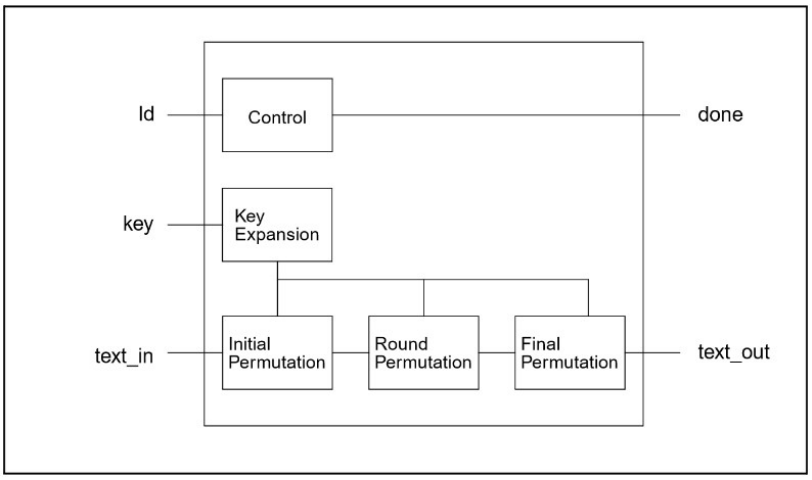


Figure VII:1 – AES Core Block Diagram from IP core documentation [6]

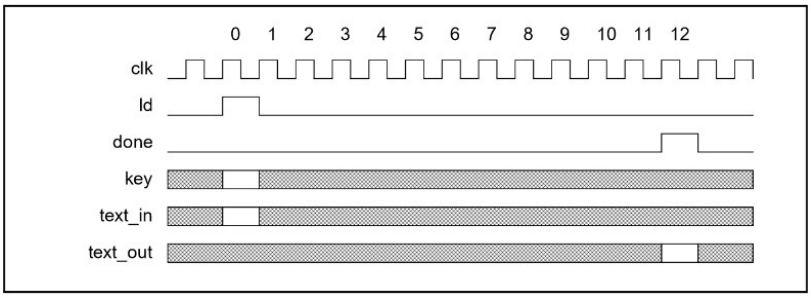
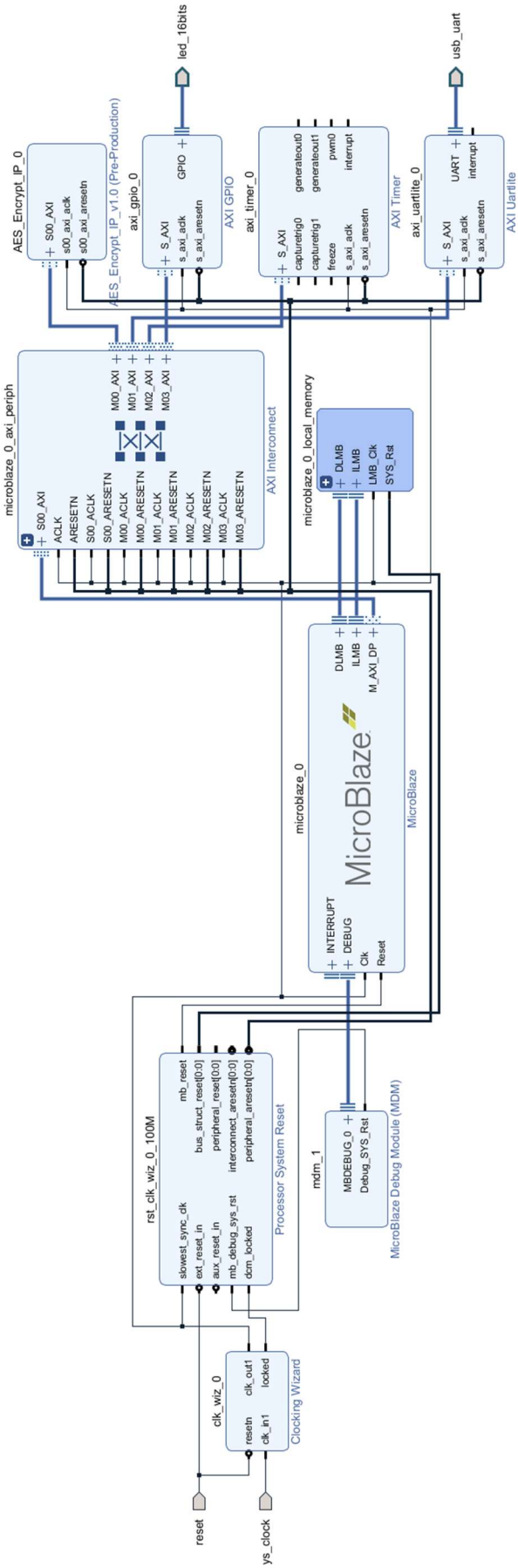


Figure VII:2 - AES core Timing Diagram from IP core documentation [6]

B. Complete Design



[6]