Performance Benchmarking for the Rivest-Shamir-Adleman Encryption Algorithm

Michael Becannon, Peter Chen, Logan Eldridge, Kyle Thomas Le,
Kyle A. Montesanti, Orlando Moya, Dr. Mohamed El-Hadedy
Contact: mabecannon@cpp.edu, pychen@cpp.edu, lgeldridge@cpp.edu, kyletle121@gmail.com,
anthonymontesanti@yahoo.com, orlandomoyarodriguez@gmail.com, mealy@cpp.edu

***Abstract* — Since modern technological advancement largely depends on performance evaluation feedback, the availability of performance metrics for the analysis of computerized hardware dependencies for the processing of software algorithms provides a mechanism by which improvements can be made to both hardware development and software design. Within the realm of computer encryption algorithms, after the security of a particular algorithm has been evaluated and confirmed, the primary decisive component in selecting the proper encryption and decryption algorithm to employ in general and commercial systems depends on the performance evaluation. Therefore, the primary framework beneath the following project design had largely been based on the analysis of the performance evaluation for performing the Rivest-Shamir-Adleman encryption algorithm on a variety of computerized hardware, including the comparison between common microcontroller units and generalized microprocessors and that of a Field-Programmable Gate Array. Since the underlying architecture for the involved microcontrollers and microprocessors is relatively static and dependent upon the design framework of the engineering staff that developed the product, the primary challenge of the project implementation had resulted from the development of optimized hardware cores for implementation on a Field-Programmable Gate Array. With the development of specialized hardware acceleration for the implementation of the Rivest-Shamir-Adleman encryption algorithm, it is thereby possible to provide for a generalized hardware system that is optimized for performing an encryption algorithm that accordingly provides for a better overall performance evaluation than that of generalized hardware units such as microcontrollers and microprocessors.**

***Index Terms*— Advanced eXtensible Interface (AXI), Average Clock Cycles Per Instruction (CPI) Field Programmable Gate-Arrays (FPGA), MicroBlaze, Rivest-Shamir-Adleman (RSA) Algorithm, Xilinx SDK, Xilinx Vivado Design Suite**

I. Introduction

It is frequently the case that an engineer would like to evaluate a computer system in terms of the performance evaluation. One method for achieving this evaluation is by using established performance metrics that all engineers who study computer systems can understand and quantify. A particularly intuitive performance metric is the execution time, that is, how long does it take the processing unit of a computer to execute a given program from start to finish. If the same program ran on two different computers, an engineer could compare the performance of the two machines by creating a ratio of the respective execution times, where a value greater than one, for example, two, indicates that the execution time for the computer in the numerator was twice as slow as the one in the denominator, or generically: $E_D = E_N / R$, where the ratio E_N / E_D is R, E_D is the denominator processor execution time, and, E_N is the numerator processor execution time. Another useful performance metric is the Average Clock Cycles Per Instruction (CPI), which is a measure of how many processor clock cycles are needed to execute each instruction on average. A CPI of one (or less) would indicate the best performance possible, while large numbers indicate poorer performance. Additional performance metrics include Millions of Floating-Point Operations Per Second (MFLOPS) and Millions of Instructions Per Second (MIPS). While these performance metrics can be helpful when working with very simple programs and hardware, they typically do not indicate execution time, and have very limited use cases, such as for comparing two machines running the same program at the same clock speed.

However, these performance metrics require context. It would not make sense to use a simple sorting algorithm on a microcontroller to estimate the performance of a complex weather forecasting system. In other words, the problem, the program that solves it, and the hardware it runs on provide meaning and context to the evaluation metrics. To begin providing some context for this particular project, suppose two people, Bob and Alice, want to share a secret message over a serial communication wire, but a third person, Eve, can also access this wire, and is always listening. The problem is how do Bob and Alice share their secret message without Eve being able to read it? A particularly clunky way to achieve this is for Bob and Alice to exchange their own private "keys" that can unlock a received "locked" message, which is unreadable otherwise. They could exchange locked messages, which Eve cannot read, and then once the message is received, unlock the messages and read them privately. However, as the number of people who would like to exchange messages grows, so do the number of keys required for secure communication, which can become unsecure if someone gives a key to Eve, and is also extremely redundant and inefficient. One solution for this problem was given by Ron Rivest, Adi Shamir, and, Len Adleman in 1977 and was named after them using their last names in the provided order, RSA, and is still one of the most widely used encryption algorithms in the world [1]. The insight was that instead of exchanging keys to unlock locked messages, each person would keep their key private, and instead, receivers of messages could exchange an unlocked lock with a sender. When the sender gets the unlocked lock, they can write their message, lock it, and send it back to the receiver, thus the

message is locked, and can be opened by the receiver's private key.

A. Rivest-Shamir-Adleman Algorithm

To achieve this, the RSA algorithm uses modular arithmetic and prime factorization. Let the numerical message bit-word be *M*, *N* be the product of two very large prime numbers not exceeding *k* bits (arbitrarily chosen), are the variables *p* and *q*, and *e* and *d* represent exponents, respectively, the encrypting exponent and decrypting exponent. Using a mathematical machine known as the ϕ function, which calculates the totient much like addition calculates the sum or multiplication and its product, the amount of numbers less than the input that are not factors or do not factor to factors of the input excluding one. For example, $\phi(4) = 2$, since 1 and 3 are not factors of 4, and $\phi(7) = 6$, since 7 is prime (1,2,3,4,5,6). For *p* and *q*, the following provides the arithmetic for all prime numbers *p* and *q*:

$$\phi(pq) = (p-1)(q-1)$$

Then, the values *e* and *d* can be chosen by selecting a value between one and the resulting equation provided above, namely:

$$ed = 1\ mod\big((p-1)(q-1)\big)$$

Let the following represent the ciphertext ("locked message"):

$$c = M^e mod(N)$$

Notice that if one were given the values for *c*, *e*, and, *N*, it would be rather challenging to solve for *M*, and could only be done through brute force methods, in other words, guessing what the message was, which is ideal for Bob and Alice. Thus, a lock has been established for the message, but the question of how to unlock the message remains. However, the message receiver can use the fact that:

$$M = c^d mod(N) = M^{ed} mod(N)$$

Thus, allowing for the original message to be recovered through decryption. There are many other features of the RSA that make it widely used that are outside the scope of this report but knowing the general procedure will allow for analyzing the algorithm to approximate the CPI [1].

To analyze this, one must first know the hardware description of each instruction. Multiplication instructions are assumed to be four clock cycles per instruction for this project, as it is highly dependent on architecture choice, and multiplication requires addition, load, and store operations, which require roughly one-two cycles in modern processing units due to pipelining implementation [2]. While other instructions such as jumps, stores, and loads are likely used, when considering the most dominant instruction type, it is the R type instructions, specifically multiplication, due to the exponentiation of very large prime numbers. If *p* and *q* were very small, thus *e* and *d* are very small, then this approximation would not apply. However, RSA demands very large prime numbers for security, thus the approximation is acceptable since multiplication will occur hundreds of thousands or millions of times, thus the CPI of the entire algorithm can be approximated to roughly the value of four. More specifically, for our consideration, the average clock cycles per instruction can be calculated as the following:

$$CPI = (4ed + c)/(ed + i)$$

where $ed >> c$ and $ed >> i$, where *c* and *i* are the cycles and instructions of other operations. Notice that it approximates to the value of four, as previously mentioned. If *e* and *d* are known, then the number of instructions can be approximated to be *ed*.

B. MicroBlaze Description

In terms of the implementation for the hardware acceleration of the RSA encryption/decryption algorithm employed on the Artix-7 based Nexys A7 Field-Programmable Gate Array, the implementation involves the utilization of the MicroBlaze microprocessor unit. This microprocessor unit is defined as a soft-microprocessor which is essentially a microprocessor unit virtualized through hardware implementation onboard the FPGA device technology. While traditional microprocessors are implemented through rigorous connection of control signals, logical binary hardware technology, and decoding operations to perform computation and decoding of control signals, the soft-microprocessor virtualization of the MicroBlaze microprocessor unit is implemented onboard the Nexys A7 utilizing a connection of lookup tables (LUTs) to be able to allow for proper connections of logic cells (LCs) and macrocells to perform the same operations of the standard microprocessor but in a different hardware implementation form. While this may not appear to be of practical importance to some, this allows for the development of an embedded system that can conform to the required hardware necessities of the designing engineers while within the maximum hardware constraints provided by the device itself. For example, when purchasing a standard microprocessor or microcontroller unit, the designing engineer is limited to the pin count, the processing power, the amount of signal converters, and other peripherals; however, when utilizing the soft-microprocessor, the designing engineer is capable of providing a tailor-made system to accommodate the necessities of the proposed design as long as the design framework does not exceed the maximum performance specifications of the soft-microprocessor virtualization itself.

One of the primary features of the MicroBlaze microprocessor virtualization is the implementation of the memory-mapped addressing system utilizing the Advanced eXtensible Interface (AXI) framework for bussing as provided by ARM based microcontrollers. This allows the designing engineer to be able to customize the addressing and bussing protocols of the microprocessor to provide for specific memory-mapped input/output designations for register control of specific operations, control signals, and input/output flow logic. Through this implementation, it is possible to design customized hardware acceleration core implementations to alleviate the performance stress placed on the microprocessor for specific operations and thereby allows for the hardware implementation of a specific core to provide for

encryption/decryption algorithms running in parallel to that of the standard control operations for the soft-microprocessor unit.

II. Related Works

The RSA algorithm is one of the most used methods of encryption since its introduction. [2] It remains the subject of academic research with interest in measuring and accelerating the algorithm's execution time. This paper compares the executive time of the RSA algorithm on various central processing units (CPUs) and devices (such as the Arduino or Raspberry PI 4), but it cannot account for all the different architectures that the RSA can be ran on, such as a graphics processing unit (GPU).

In one such paper that does use GPUs, Mahajan, and Singh, describe improving the execution time of RSA encryption by accelerating the most computationally demanding sections (modular multiplication and
modular exponentiation) of the algorithm on a GPU using CUDA and OpenCL technology [3]. In this case a CPU and GPU work in parallel, to encrypt messages. An analysis of the results found that CPUs assisted by CUDA or OpenCL had better execution times than those without.

Another Paper of similar topic, Fadhil and Younis compare the RSA algorithm performance of a multicore CPU (Intel Core I7-2670QM, 2.20 GHz) to that of a GPU (Nvidia GeForce GT630M) [4]. In this comparison three implementations of the RSA algorithm were tested (sequential CPU, multithreaded CPU, and GPU) at various key bit sizes. The results show that the average speedup of the GPU was 26 times over the sequential CPU and the multithread implementation of the CPU only had a 6 times speedup of the sequential CPU implementation.

In this paper we compare RSA performance of a field-programmable gate array (FPGA) to CPUs. Others have implemented the RSA encryption algorithm onto FPGAs as well. Iana, Anghelescu, and Serban, describe the implementation and performance [5]. This paper's implementation resulted in an FPGA capable of encrypting/decrypting a message in
212.99ms with a 50 MHz clock.

As show by the research above the topic of RSA implementation and performance is widely studied. This paper makes similar contributions by analyzing the execution time of the RSA on various CPUs and microcontrollers as well as describing implementations of RSA encryption.

III. Design

To implement the RSA algorithm onto the Nexys A7 FPGA, the algorithm needs to be broken down into its components so that it can be represented in a hardware description language. There are two main parts to the RSA algorithm, the generation of the key and the generation of the message signal. The code for these two parts had been separated into different modules, which are both encapsulated by a larger module that oversees assigning values and controlling when the appropriate module should be asserted. The following is a high-level schematic of how the RSA algorithm is established in our program:

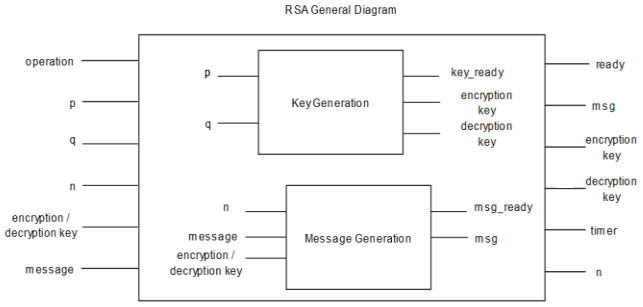

Fig. 1. General block diagram representing the logical control flow for each of the required signals for the hardware implementation of the Rivest-Shamir-Adleman encryption algorithm

Note, this is a very high-level diagram and formed as a basis for the hardware description language design. The actual code differs in structure and naming convention, but overall, has the same functionality. The following is the key: *p* and *q* are the prime numbers for creating the key, *n* is the modulus for generating the message, and the operation denotes how the RSA algorithm will function. There are two operations, specified by the user: generating keys and a message with those keys, or only generating a message signal. If the user wants to generate the keys and a message with those keys, they will need to input the two primes, *p* and *q*, and the message. The code will generate the keys using the key generation module, then use the key to encrypt or decrypt the message in the message generation module. Likewise, if the user wants to only generate a message since they have a key, they will need to input the key, message, and the modulus *n*. The code will skip the key generation module since the user will provide one and will generate a message based on the input key, message, and modulus operation.

A. Multiplication and Modulus Operations

As stated in the explanation on how RSA works, to generate the keys and messages involves modulus and multiplication arithmetic operations. Within the SystemVerilog hardware description language, multiplication and modulus are not as simple as adding and subtracting. Unless specified, these operations would need to be done in one clock cycle, which in our case is 10.00 ns. Since we are doing multiplication and modulus of very large numbers, the system will struggle to be able to do this in such a clock cycle constraint. This had been the initial issue with the code description that our system is based off, as it had originally failed to meet these specific timing requirements when attempting to perform very large multiplication operations. To fix this situation, pipelining was implemented, as well as a Xilinx multiplication hardware acceleration core.

Pipelining splits up the operation into multiple clock cycles, allowing calculations to be done over time and not being restricted to the signal clock cycle. The downside of pipelining is that our operation would take longer. Since we are not receiving a valid answer at one clock cycle but instead after many clock cycles, the overall execution time of our algorithm

will be longer. However, this trade-off allows the system to be implemented without decreasing the overall speed of our FPGA device, therefore meaning that other peripherals that are connected would not have to be impacted by the execution of the RSA algorithm.

Modulus functionality was written in the *mod_custom.sv* module. The functionality was implemented using the restorative division algorithm, where every clock cycle, the next procedure of the algorithm is performed. When the module is finished calculating, there is a flag that goes high to denote its status. This flag is used to move onto whatever step lies after the modulus. The following is the state machine on its operation:

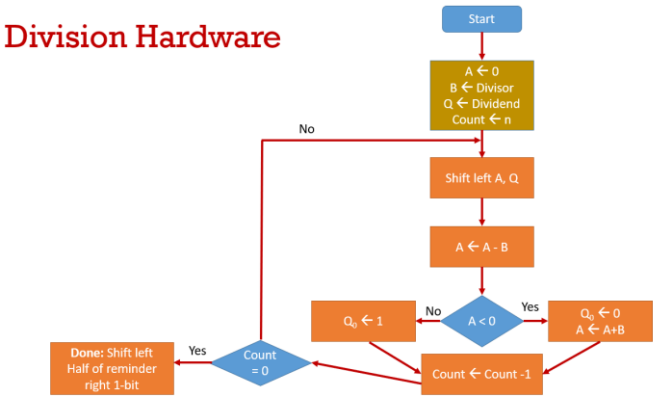

Fig. 2. Organizational flowchart representing the optimized implementation for performing the mathematical process of division in the hardware implementation for the Nexys A7 Field-Programmable Gate Array. Provided by ECE 4300 lecture notes [6]

Multiplication was performed using the Xilinx Complex Multiplier Intellectual Property (IP). Ideally, it would have been preferred to use a custom optimized multiplier but due to time constraints it was optimal to use a prefabricated module. The core allows for 32-bit signed multiplication and provides a flag when operations are finished. Furthermore, the IP core allows for customization such as choosing digital signal processing (DSP) slices or lookup tables (LUTs), allowing for the prioritization of performance or size on the chip, and much more. This thereby provided a simple instantiation module that worked for all practical purposes.

B. Key Generation

The key generation module needs to calculate both keys, encryption, *e*, and decryption, *d*, based on the prime numbers *p* and *q*. Using the Complex Multiplier IP, the totient is first found. Then, using the Euclidean algorithm and the modulus module, the first value that provides the greatest common denominator of one with the totient is used as the encryption key. To find the decryption key, the extended Euclidean algorithm needs to be implemented. However, since the standard Euclidean algorithm is already used to find the encryption key, calculations could be done during its calculations to calculate the decryption key simultaneously. The following is a general logic diagram on how the module behaves:

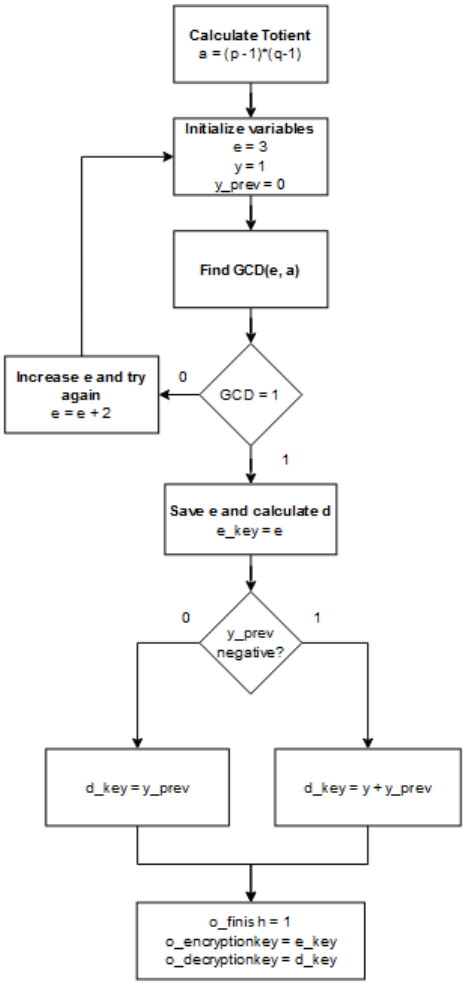

Fig. 3. Organizational flowchart representing the overall structure of implementation for key generation within the hardware implementation logic for the Nexys A7 Field-Programmable Gate Array

C. Message Generation

The message generation module uses one of the keys (if user chose encryption or decryption) as the power to raise the message by. Combined with the modulus *n*, the output message is generated. Depending on the encryption or decryption key, this operation could take a significant amount of time to a little amount of time, due to the fact that it is raising the message to a certain power, taking a significant amount of time to multiply. To help speed it up, the binary exponentiation algorithm was used. In this algorithm, the number of operations and memory size is greatly reduced to optimize the performance of the operation through hardware acceleration. The following state diagram shows how the binary exponentiation algorithm behaves:

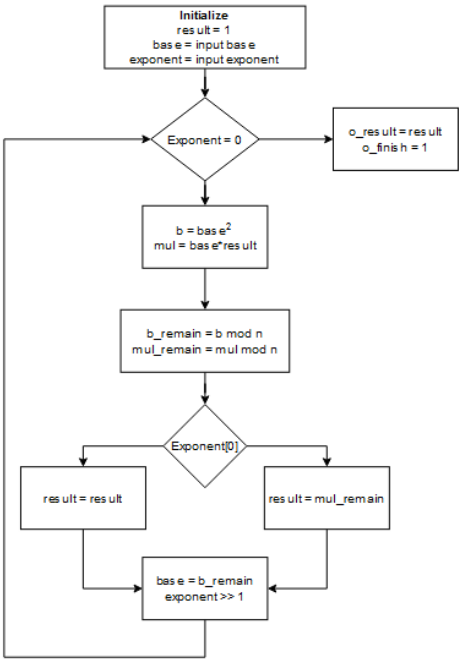

Fig. 4. Organizational flowchart representing the overall structure of implementation for message generation within the hardware implementation logic for the Nexys A7 Field-Programmable Gate Array

D. Top Module Design

The top module acts as a controller wrapping the key generation and message generation module together. It oversees reading the operation specified by the user and assigns the proper values to the modules. As stated in the overview, p, q, and specifying encryption or decryption are unused if the system is in message generation only mode, while the modulus and key are required. In contrast, in key generation with message generation mode, the modulus and key are unused but p, q, and specifying either encryption or decryption is necessary. This module ensures that an unwanted input does not get used. Also, the module calculates the modulus, n, if in key generation mode. In addition, the module handles counting how many clock cycles the operation takes, outputting it so that the user can read how long the module took. The following state diagram shows how inputs are mapped based on the mode:

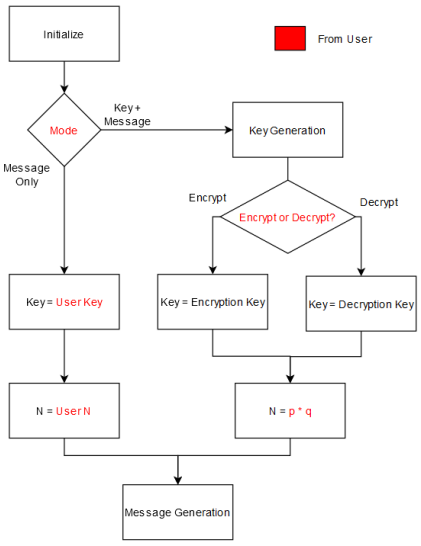

Fig. 5. Organizational flowchart representing the overall structure of implementation for the top module, which holds the key and message generation module. This implementation logic is for the Nexys A7 Field-Programmable Gate Array

IV. Implementation

With the RSA module designed, it must be implemented into the MicroBlaze system. Xilinx provides a set of tools that allow designers with the capability to program the FPGA for us to get our RSA design integrated into the MicroBlaze microprocessor unit. The tool used for programming the FPGA hardware with our MicroBlaze and RSA design had been Vivado 2019.1. The tool used for programming the MicroBlaze to interact with the RSA design had been Xilinx SDK 2019.1. Since programming the FPGA means that we are programming the FPGA directly, we need to program the board in order to have the MicroBlaze soft-processor, our RSA module, any other modules that we would need to function. Thankfully, Vivado provides an interactive way of building together a system using a graphical user interface (GUI) system called *Block Design*. With Vivado's *Block Design*, our group had been able to piece together Xilinx Intellectual Properties to produce a functioning MicroBlaze system.

A. Block Design

The minimum system for the MicroBlaze system consists of the MicroBlaze processor, a reset handler, clock handler, memory, and debug handler. *Block Design* automatically handles the creation of many these systems. With this, the user only inserts the MicroBlaze processor and based on the settings provided, *Block Design* will generate all the other systems required by the MicroBlaze [7].

Xilinx or custom intellectual properties can also be added, and using the AXI interface, will be connected to the processor. *Block Design* also handles automatic connections of IPs to AXI, and to the processor. Thus, the user only needs to add the IP with AXI support, and *Block Design* will automatically route the connections to the MicroBlaze processor. This allows for a

quick and efficient connection of systems for the group to create a bootable FPGA.

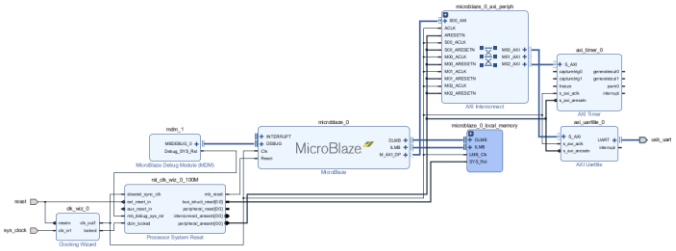

Fig. 6. Base block design hierarchy representing the design provided for the structure of the hardware implementation provided in Xilinx Vivado Suite

B. Custom Intellectual Property Core

To connect our custom RSA code to the AXI interface, we must package our own custom Intellectual Property. Vivado provides the tools to do so [8], as well as configuration options based on how we want to design our IP system. Since our RSA module does not require much memory size, the AXI4-Lite interface is more than enough for our needs. Based on our system, we needed eight registers to process and transmit and receive our data. Our module is also based on 32-bit inputs, with the potential for 64-bit outputs. However, the GUI only allows for a 32-bit wide input. These values would have to be modified in the IP to accommodate for our system. After the IP is created, we had to add our custom RSA module and make our changes to the data bus. In the file **IP name*_S00_AXI.v*, the code for interacting with the AXI bus is modifiable for adjusting. The parameters "C_S_AXI_DATA_WIDTH" had to be changed to allow for 64-bit data transfer over the AXI bus. In addition, the local parameter "OPT_MEM_ADDR_BITS" had to be changed to the value of two for the addressing from the processor to be decoded properly inside this module. To add our custom RSA module, we instantiated it in the portion of the code labeled "Add user logic here." Then, we had to make sure the inputs and outputs of our custom RSA module was accessible over AXI.

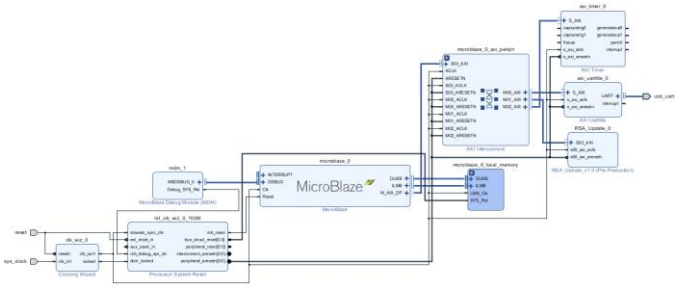

Fig. 7. Code segment representing the previously mentioned modifications for the Intellectual Property core implementations

In this section of the code lies the portion for how AXI decides what data the processor is given, based on the address it is trying to read. Since AXI writes and reads 8 bits at a time, the offset for each of our 64-bit wide data registers is eight. For example, accessing an offset of zero would read "slv_reg0", and accessing an offset of sixteen would read "slv_reg2". To have AXI read the data from our custom RSA module and not the "slv_reg", our group had connected the output registers of our module instead.

```
419         // Address decoding for reading registers
420         case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
421           4'h0   : reg_data_out <= data_msg;
422           4'h1   : reg_data_out <= data_encrypt;
423           4'h2   : reg_data_out <= data_decrypt;
424           4'h3   : reg_data_out <= n_output;
425           4'h4   : reg_data_out <= clk_count;
426           4'h5   : reg_data_out <= {31'b0, mod_exp_finish};
427           4'h6   : reg_data_out <= slv_reg6;
428           4'h7   : reg_data_out <= slv_reg7;
429           default : reg_data_out <= 0;
430         endcase
431     end
```

Fig. 8. Code segment representing our personal modifications to the Intellectual Property core implementation previously mentioned

Then, our group had to connect the inputs of our module to the "slv_reg" that AXI writes to. For this, we do not need to change the variables like what was done previously for the registers that AXI reads from. Instead, we can just call them directly into our module. For example, *p* can use the value of "slv_reg0" and *q* can use the value of "slv_reg1". This way, when we attempt to write to offset zero in the processor, we are actually writing to variable location *p* within the module. Likewise, if we were to write to offset eight in the processor, we are writing to variable location *q* within the module. With these values adjusted, the custom IP is complete and ready to be placed into our block diagram and connected to the processor.

```
419         // Address decoding for reading registers
420         case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
421           4'h0   : reg_data_out <= slv_reg0;
422           4'h1   : reg_data_out <= slv_reg1;
423           4'h2   : reg_data_out <= slv_reg2;
424           4'h3   : reg_data_out <= slv_reg3;
425           4'h4   : reg_data_out <= slv_reg4;
426           4'h5   : reg_data_out <= slv_reg5;
427           4'h6   : reg_data_out <= slv_reg6;
428           4'h7   : reg_data_out <= slv_reg7;
429           default : reg_data_out <= 0;
430         endcase
431     end
```

Fig. 9. Finalized block design including the custom RSA intellectual property combined with the base design provided in Xilinx Vivado Suite

C. SDK Interface

Our custom RSA module is now a custom RSA IP and can be communicated to by the processor over AXI. Xilinx SDK is used to create the code for interfacing that communication. Xilinx SDK allows for creating embedded applications on the MicroBlaze soft-processor, with driver and library support for their premade IPs [9]. This tool allows us to create a C++ application to interface with our custom RSA IP. For the AXI Uartlite and AXI Timers, Xilinx also provides C++ drivers and libraries for these. This allows us to connect to our board using the built-in terminal or PuTTY and print over a serial communications protocol. Two libraries are used extensively in our C++ project, the serial print methods in *xil_printf.h* and the register accessing methods in *xil_io.h*. With these methods, we can read and write to our custom RSA IP by writing to the AXI memory address as well as print the values read from them. The *xil_io.h* file lists the following methods which were used for

creating our own RSA drivers: Xil_Out64 and Xil_In64. These methods work by taking the base address of the AXI peripheral we are accessing and then the offset for the specific data register that we would like to select for operations. The following is a screenshot of the addressing map of our custom RSA IP, which was the result of modifying the AXI read registers and specifying which module inputs are connected to which "slv_reg" variable locations.

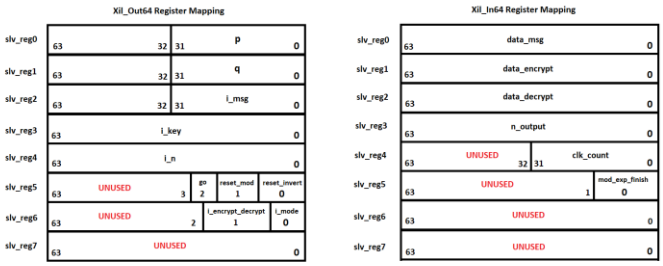

Fig. 10. Modified register map for the MicroBlaze implementation system involving our custom hardware acceleration core for the RSA algorithm

As it is shown, the Xil_Out64 method is for writing to the AXI peripheral while the Xil_In64 method is for reading from it. The base address of our custom RSA IP is in the *xparameters.h* file, which is then used with the *xil_io.h* methods to communicate with our IP. As mentioned in the custom IP portion, since we are dealing with 64-bit wide data busses, the offset between each register is eight. By creating our own custom methods to write and read to certain registers, we can create a simple driver for users to understand and use.

Note that the internal counter was used instead of the provided AXI timer for measuring the speed of our RSA module. This is because the AXI timer accounted for read and write times over the AXI bus and had a noticeable delay due to this. Since the goal is to measure execution time, it was decided to use the internal timer.

```
**Please enter the following for the corresponding mode:
    -> [0!] - Key generation with message generation.
    -> [1!] - Message generation only.

****You have chosen message generation only mode****

  Please enter your input message (digits only), followed by a ! when finished:
    -> You inputted 4782969

  Please enter your modulus, n, followed by a ! when finished:
    -> You inputted 800143763

  Lastly, please enter your key, followed by a ! when finished:
    -> You inputted 457173463


****  RUNNING RSA ALGORIHTM  ****
**** MESSAGE GENERATION ONLY MODE ****

 Finished! The following are your values:

    ->Output message: 9
    ->Operation took: 23.239 us!
```

Fig. 11. Example UART output of the MicroBlaze system on the Nexys A7 integration with Xilinx SDK. Displays values and allows user to interact with the system to run the RSA algorithm

V. EVALUATION

A. Resource Utilization

The optimization of the FGPA code can be determined by the amount of LUT and Flip-Flop (FF) resources were used. Since the AXI Complex Multiplier IP was used, there was an option to use dedicated digital signal processing (DSP) slices. These slices are typically dedicated for binary multipliers and accumulators [10] and are useful for arithmetic operations. These DSP slices are preferred since they are optimized for arithmetic and allow for less LUT and FF usage. The following table describes the LUT, FF, and DSP usage of only the RSA module, followed by the usage of the entire MicroBlaze system created in the *Block Design*. It must be noted that the MicroBlaze system has the usage of BRAMs, which are large memory pools. The RSA module itself does not use BRAMs, but it is important to highlight how much BRAM the system uses.

System	*LUT*	*FF*	*BRAM*	*DSP*
RSA Module	1370	3712	0	80
MicroBlaze System with RSA IP	5904	8895	34	83

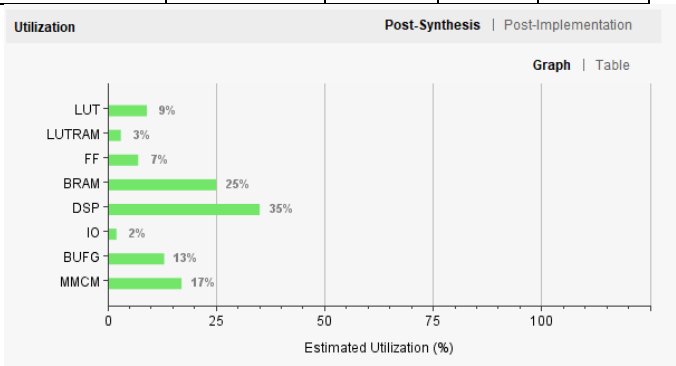

Fig. 12. Resource usage for MicroBlaze implementation system involving our custom hardware acceleration core for the RSA algorithm for the Nexys A7-100T

For the Nexys A7-100T, which is the higher performance version of the Nexys A7 line, it can be seen that the system does not use a significant amount of resources compared to what the system is capable of.

B. Performance Evaluation

From the internal counter inside of the RSA module, it is possible to measure the amount of system clock cycles it took for the module to run. Using the Nexys A7 clock speed of 100 MHz, this allows us to calculate the execution time of the code. This calculation is done on the MicroBlaze FPGA side since it is able to handle float and precision operations.

When using the RSA algorithm with MicroBlaze, we are able to gather a few different measurements to analyze the performance of the algorithm on an FPGA. As explained in detail at the beginning of this report, the encryption is done by evaluating the equation $m^e \pmod n$ where "m" is a number message, and (n,e) is the public key.

On the other hand, decryption with a private key is done by evaluating me*d (mod n).

However, before we can even begin encrypting a numeric message, we must first generate our keys. The public key will consist of the (mod n) and exponent "e". Then we are able to generate a secret key by use of the Euler φ function of "n" which is $\varphi(n) = (p - 1) \times (q - 1)$.

With this knowledge we can begin to break down the performance metric and data collected by implementing the RSA algorithm on an FPGA with the use of MicroBlaze.

To test the algorithm, we begin with the arbitrarily chosen prime numbers p = 9973 and q = 80231. Our exponent e is 7.

First, we begun by timing the overall time it takes to create the keys, encrypt the numeric message, and the finally decrypt the message. By running this test using the stated primes and exponent above, we find that it takes MicroBlaze, running on the FPGA, a total time of 30.690 μs.

We can then further break the execution time into the three components of key generation, encryption, and decryption times.

When testing only the time it takes to generate the keys, we find a time of 5.011 μs. Then proceeding to measure the encryption time we find a time of 2.440 μs elapsed. Finally, when measuring only the time it takes to decrypt the message, we find a time of 23.239 μs elapsed.

From this data, one of the first aspects to become prevalent is the ratio of each of this task to the total time. Key generation uses about 16.00% of the total execution time, as shown by the following:

$$5.0110\ \mu s / 30.690\ \mu s = 0.1600$$

Encrypting the message takes about 8.000% of the total execution time, as shown by the following:

$$2.4400\ \mu s / 30.690\ \mu s = 0.0800$$

Decrypting the message takes about 76.00% of the total execution time, as shown by the following:

$$23.239\ \mu s / 30.690\ \mu s = 0.7600$$

The decryption time truly takes the majority of the execution time. In each of these timing tests, we used the same prime numbers and exponent stated above to remain consistent. However, using different prime numbers or exponents will greatly change the time it takes to perform the RSA encryption algorithm.

The algorithm was then tested using prime numbers p = 11 and q = 13 with the same input message of 9. From this we find the key generation takes a total time of about 3.409 μs, encryption takes 2.440 μs, and decryption takes 5.639 μs.

With this data the percentage of execution time are as follows. Key generation uses about 29.67% of the total execution time, as shown by the following:

$$3.409\ \mu s / 11.488\ \mu s = 0.2967$$

Encrypting the message takes about 21.24% of the total execution time, as shown by the following:

$$2.440\ \mu s / 11.488\ \mu s = 0.2124$$

Decrypting the message takes about 49.08% of the total execution time, as shown by the following:

$$5.639\ \mu s / 11.488\ \mu s = 0.4908$$

When comparing the two datasets for the larger prime numbers and the smaller prime number we are able to see that the execution time is still mainly used for decryption time followed by key generation followed by the encryption time. This order of time usage by percentage remains the same across the board. However, we also notice that when using smaller prime numbers, the key generation time doesn't take as long, but it does take up a larger percentage of the total execution time similar to the encryption time. This is due in part to the decryption time being decorated as the algorithm for decryption does not need to loop as long for such a short encryption length with the smaller prime numbers.

As far as the total execution time goes, when testing with primes p = 9973 and q = 80231, the total execution time was measured to be 30.690 μs. In comparison when testing with primes p = 11 and q = 13, the total execution time was measured to be 11.488 μs. By increasing the primes from 4 bits to 17 bits we get an increase in execution time by about 37.43% as shown by the following:

$$11.488\ \mu s / 30.690\ \mu s = 0.3743$$

Based on this, it is observable how the bit length of the encryption can exponentially increase the time it takes the RSA algorithm to execute with the same message. The larger we make our primes the longer our algorithm takes to execute for generating keys and encrypting. Given that we can support up to a 32bit q and a 32bit p we can easily make this algorithm take longer to execute with such complexity. Also, the larger we make our exponent the longer it will take for the encryption and decryption process based on the encryption and decryption equation described briefly above. Another large impact on the speed performance of the RSA algorithm in all aspects is the size of our message. In our data, we used the numeric message "9" to test the algorithm. Given that this is a small message we are not making the algorithm perform as long on a long message which would require more time to encrypt and decrypt each number in the message.

C. Comparisons

Device:	*Execution Time (ms)*:
MicroBlaze Processor	0.00584
Raspberry Pi 4	0.00200
Arduino Uno Rev 3	60.0000
Esp8266 Microcontroller	331.000

The Rivest-Shamir-Adleman encryption algorithm was implemented on a multitude of different hardware devices to

compare the capabilities of their architecture. The performance evaluation was based mainly on the execution time of the algorithm. A clock timer was used in the program to measure the execution time of the algorithm beginning at the start and stopping at the end to print the output as total time taken. The function was run 1000 times on each device to obtain an average of the execution time for each device. For consistency, the prime values used for each device was the same.

Notably, the newer and more powerful FPGA and Raspberry Pi devices performed much better than the cheaper Esp8266 and Arduino boards. The Nexys A7 Artix-7 based FPGA is optimized for high performance logic with its MicroBlaze RISC Harvard architecture that features a rich instruction set. MicroBlaze is a soft-core processor that thrives in being dynamic and flexible. It can run a new instruction every cycle to achieve single-cycle throughput. Comparatively, the Raspberry Pi 4 rivals the performance of the FPGA with its 1.5 GHz quad-core ARM CPU that follows the ARM Cortex-A72 architecture. This is much faster than the 16MHz clock speed of the Arduino and even the 100 MHz clock speed of the FPGA. As expected, the basic, lower-spec devices could not keep up with the FPGA and Raspberry Pi 4.

Furthermore, the processor clock speed for the Raspberry Pi 4 is benchmarked at a maximum speed of 1.5 GHz while the maximum processing clock speed of the Nexys A7 MicroBlaze soft-core microprocessor is benchmarked at about 100 MHz. Therefore, while the microprocessor of the Raspberry Pi 4 is relatively fifteen times faster than that of the MicroBlaze microprocessor unit implemented on the Nexys A7, this performance evaluation does not scale accordingly to what would be expected for the execution time evaluation. In evaluating the performance of the Rivest-Shamir-Adleman algorithm implementation onboard the Raspberry Pi 4 and the MicroBlaze microprocessor unit, the following calculation represents the performance evaluation in terms of the execution time:

$$0.00584 \text{ ms} / 0.00200 \text{ ms} = 2.920$$

Therefore, while the processing unit of the Raspberry Pi 4 is relatively fifteen times faster than that of the MicroBlaze microprocessor unit implemented on the Nexys A7, the overall system performance in terms of the execution time for the Rivest-Shamir-Adleman algorithm shows that the Raspberry Pi 4 is only 2.92 times faster than that of the MicroBlaze microprocessor unit for the encryption algorithm under investigation. This is important to observe as the implementation of the Rivest-Shamir-Adleman encryption algorithm as a customized hardware acceleration core for the MicroBlaze microprocessor unit serves as means for minimizing the performance difference between the Raspberry Pi 4 and the MicroBlaze microprocessor unit, therefore displaying the importance of considering hardware acceleration cores for the implementation of rather mathematically involved algorithms.

VI. Conclusion

In conclusion, when evaluating the performance of a computerized system, there are several performance metrics that can be useful for comparing the relative performance of various machines. In order to utilize these metrics, architecture design, hardware implementation, the software involved in program execution, and the compiler used to compile a program into machine executable code should all be considered for the most complete system analysis. Note that in this report, the analysis for utilizing these metrics mostly revolves around the architecture and the software, which leaves a significant portion of analysis that could hypothetically be done for comparing compiler performance and specific hardware. It is possible that two machines have the same architecture, but realize the architecture using different models and makes of hardware. Ultimately, the major conclusion to be drawn from this experimentation results from the effects of hardware acceleration, which allows for the tremendous improvement of system performance as measured by the execution time in comparing the hardware acceleration core of the Rivest-Shamir-Adleman algorithm to that of the sequential execution of the same algorithm on microcontrollers and microprocessors. Therefore, for large server systems or database centers, there is potential improvement into the performance of their systems through inclusion of encryption-based hardware acceleration cores implemented within the system microprocessors in order to alleviate the workload from the main processing units and eliminate system overhead to provide for a more responsive system.

References

[1] D. Ireland, w., 2020. *RSA Algorithm*. [online] Di-mgt.com.au. Available at: <https://www.di-mgt.com.au/rsa_alg.html> [Accessed 29 November 2020].

[2] Valhalla.altium.com. 2020. [online] Available at: <http://valhalla.altium.com/Learning-Guides/Legacy/CR0163%20MicroBlaze%2032-bit%20RISC%20Processor.PDF> [Accessed 1 December 2020].

[3] R. L. Rivest, A. Shamir and L. Adleman “A method for obtaining digital signatures and public-key cryptosystems” Communications of the ACM, vol. 21, pp. 120-126, 1978

[4] Mahajan, Sonam, and Singh, Maninder. "Performance Analysis of Efficient RSA Text Encryption Using NVIDIA CUDA-C and OpenCL." *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing* (2014): 1-6. Web.

[5] Mohammed Fadhil, Heba, and Issam Younis, Mohammed. "Parallelizing RSA Algorithm on Multicore CPU and GPU." International Journal of Computer Applications 87.6 (2014): 15-22. Web.
Iana, G. V, Anghelescu, P, and Serban, G. "RSA Encryption Algorithm Implemented on FPGA." 2011 International Conference on Applied Electronics (2011): 1-4. Web.

[6] M. E. Chapter-3: Arithmetic for Computers. ECE 4300. California State Polytechnic University, Pomona. 8 October, 2020.

[7] Xilinx.com. 2018. Designing IP Subsystems Using IP Integrator, [online] Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug994-vivado-ip-subsystems.pdf

[8] Xilinx.com. 2018. Creating, Package Custom IP Tutorial, [online] Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1119-vivado-creating-packaging-ip-tutorial.pdf [Accessed 3 December 2020]

[9] Xilinx.com. 2019. Using Xilinx SDK, [online] Available at: https://www.xilinx.com/html_docs/xilinx2019_1/SDK_Doc/sdk_getting

_started/sdk_getting_started.html#sdk_getting_started [Accessed 3 December 2020].
[10] Xilinx.com. 2018. *7 Series DSP48E1 Slice User Guide*, [online] Available at: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf [Accessed 2 December 2020].

Michael Becannon (CPP'20) was born in Huntington Beach, California 1996. He will be graduating with a degree in Computer Engineering at California State Polytechnic University, Pomona. Michael would like to work wherever he can do the most good for others, either in software or hardware, with an end goal to teach at university or community college. Specializations include data analysis and structuring, circuit modeling and critical path analysis, imperative programming, and research.

Peter Chen (CPP'21) was born in San Francisco, California in the year of 1999 and is currently in their final year for their undergraduate degree in Computer Engineering at California State Polytechnic University, Pomona. With a focus on Software Engineering, Peter has gained experience working in backend and frontend throughout his undergraduate career. After graduation, he will begin working as a Software Engineer.

Logan G. Eldridge (CPP'21) was born in the city of Orange, California in the year 2000 and is currently pursuing an undergraduate degree in Electrical and Computer Engineering at California State Polytechnic University, Pomona.

At California State Polytechnic University, Pomona, Logan Eldridge is focused in power engineering and microelectronic design. Logan Eldridge specifically gears his focus toward DC-DC designing and printed circuit board design. He gained most of his experience through his year-round internship at Wavestream Corporation where he works on the electrical engineering team to design and develop power and auxiliary microcontroller PCB's for application in solid state amplifier products. After graduation he will most likely begin working full time at Wavestream Corporation should he continue to focus on his current interest.

Kyle Thomas D. Le (CPP'21) was born in Orange County, California in the year of 2000 and is currently in their final year for their undergraduate degree in Computer Engineering at California State Polytechnic University, Pomona.

At California State Polytechnic University, Pomona, Kyle Thomas focused on embedded systems and digital design. Specifically, Kyle Thomas focuses on the work of integrating systems using Field-Programmable Gate Arrays (FPGAs). Kyle Thomas has grown his experience through his internship at Jet Propulsion Laboratory, NASA, working on embedded systems, where he got his first experience in the professional workspace. After his graduation, he will begin working as an Electrical Engineer at Northrop Grumman Corporation in August 2021, with a focus on FPGA work, and aims to continue to explore the interesting work of the field.

Kyle A. Montesanti (CPP'20) was born in the city of Los Angeles, California in the year of 1996 and currently is finishing up their undergraduate degree in Electrical Engineering at the California State Polytechnic University, Pomona. Prior to studying Electrical Engineering, Kyle A. Montesanti had studied Biochemistry and Forestry and Ecological Services at Citrus Community College, where he worked alongside the United States Forest Service in providing restorative procedures to the local environmental communities and conducted biotechnical and botanical research with the Rancho Santa Ana Botanic Garden based in Claremont, California.

While at the California State Polytechnic University, Pomona, Kyle A. Montesanti has pursued an undergraduate degree in Electrical Engineering while specifically focusing on digital system design and software engineering. Notable interests include Field-Programmable Gate Arrays, Data Analytics and Statistical Analysis, Artificial Intelligence, and Computerized Systems Optimization. Kyle A. Montesanti will begin working as a Systems Engineer at RoviSys in January 2021 where he will work to provide the design of systems based on programmable logic controllers (PLCs) and will continue to provide an enthusiastic approach to computerized systems and software optimizations.

Orlando Moya (CPP'20) was born in Montclair, California in 1993. He will graduate from California State Polytechnic University, Pomona with an undergraduate degree in Computer Engineering. During his time at the California State Polytechnic University, Pomona, Orlando has served as the Requisitions Officer for the Cal Poly Pomona Robotics Club and wishes to purse a career in the field of Robotics. As a member of the Cal Poly Pomona Robotics Club, Orlando has provided work on various projects, where he has developed skills such as understanding how to employ Computer Vision onboard embedded systems and how to implement the Robot Operating System (ROS) with such systems.