Performance Evaluation of Rivest-Shamir-Adleman Algorithm

MicroBlaze Soft Microprocessor

Group E
Logan Eldridge, Michael Becannon, Kyle Le, Kyle Montesanti, Peter Chen, Orlando Moya

Table of Contents

Introduction to Rivest-Shamir-Adleman

- In 1977, the RSA algorithm was publicly described as a replacement for symmetric key encryption algorithms
  - Specifically, the National Bureau of Standards Algorithm
- The asymmetric nature of the RSA algorithm means the algorithm is easy to compute in one direction but not in the other
  - Achieved via Modulus Arithmetic (Clock Arithmetic)
  - 100,000%17 = X is computable but what about $X^{60}\%=6$
    - There is a very wide solution space for the second equation compared to the first
    - With a clever arrangement, only one person (receiver) will be able to solve for X

Consider the following

1. Alice and Bob are sending information to each other, but Bob only wants Alice to be able to see the message.
2. Alice gives her public key to Bob and keeps her private key to herself.
3. Bob wants to send a message to Alice, and uses the public key and RSA to encrypt it.
4. Alice receives the message and can decode it using her private key and RSA.

Algorithm Fundamentals

- Consider a deciphering and encryption function D(X) and E(X) and a message M
  - It follows then that M = D(E(M))
  - Also notice that M = E(D(M))
    - You get the message back if you decipher the encrypted message or encrypt a deciphered message
    - D(X) and E(X) are easy to compute, but E(X) does NOT imply D(X)
    - In essence, you can’t figure out how to decipher a message from given encryption

RSA Fundamentals (Cont.)

- A simple analogy for understanding the RSA algorithm is Key and Lock
- A “locked” message is unreadable without a key
- A pair of people could exchange the key for their lock, so that they can send a locked message to the other person and they can unlock it
- With many people in the network, this is messy and inefficient
- RSA is like sending an unlocked lock to the message sender, and then locking the message with the receivers lock, E(M)
    - The sender does not have access to the receivers private key, the message is securely encrypted, and the message is able to be exchanged

RSA Mathematics

- As promised, the RSA employs modular arithmetic to achieve its goal
  - Let:
    - M = message bit-word (should be hard to solve for!)
    - e = encrypting exponent; E(X)
    - d = deciphering exponent; D(X)
    - C = ciphertext (encrypted message that everyone can see)
    - N = the product of two prime numbers, p and q
  - Then to make a message that is hard to solve for use:
    - $(M^e) \bmod (N) = C$ or $M^e \,\%\, N = C$
    - Note that solving for M requires trial and error, as there is no known solution otherwise
  - To get back the message:
    - Let $(C^d) \bmod (N) = M$
    - Then $(M^{ed}) \bmod (N) = M$
  - Using methods of prime factorization (Euler), values of e and d can be established such that e is easy to solve for but d would require years (prime factorization is NP)

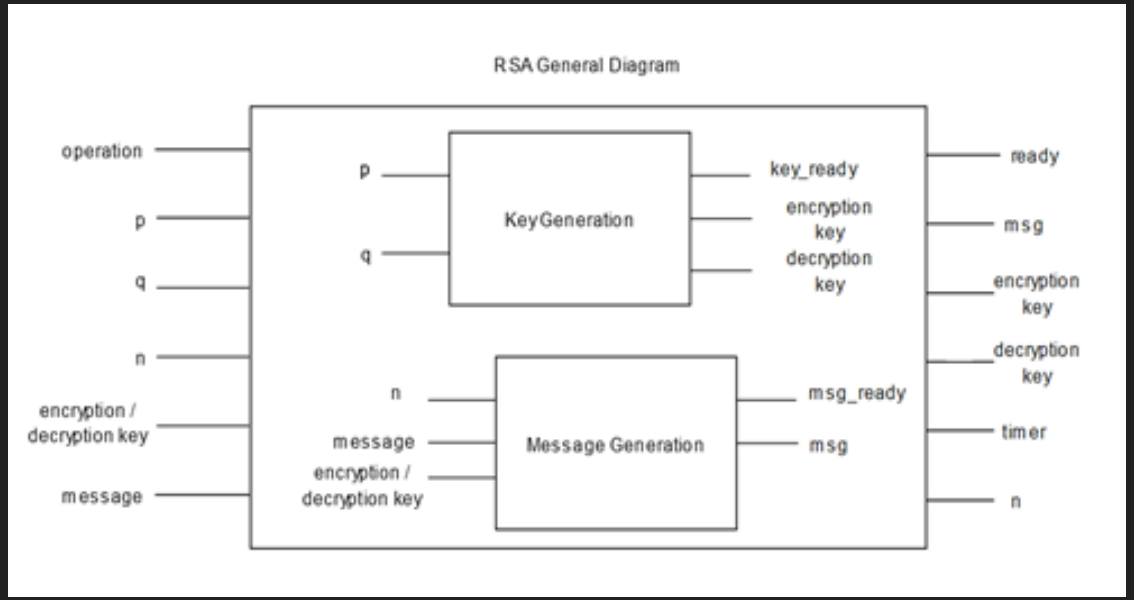
RSA Summarized

- Solving for the message is hard due to modular arithmetic and exponentiation
- N, the large product of prime numbers, are used to generate E and D
- If D is private and large, then solving for the original message is even harder
  - Let E = 7, N = 364,544,412,454,975,252,633, C = 277,975,615,976,533,914,999 then:
  - $M^{7*D} mod (364,544,412,454,975,252,633) = 277,975,615,976,533,914,999^{D} = M = ?$
  - You are given three pieces of information, N, C, and E, but M is hard to solve
  - Trying to solve for D using the Phi function means doing prime factorization + more
  - Prime factorization is NP

MicroBlaze Introduction

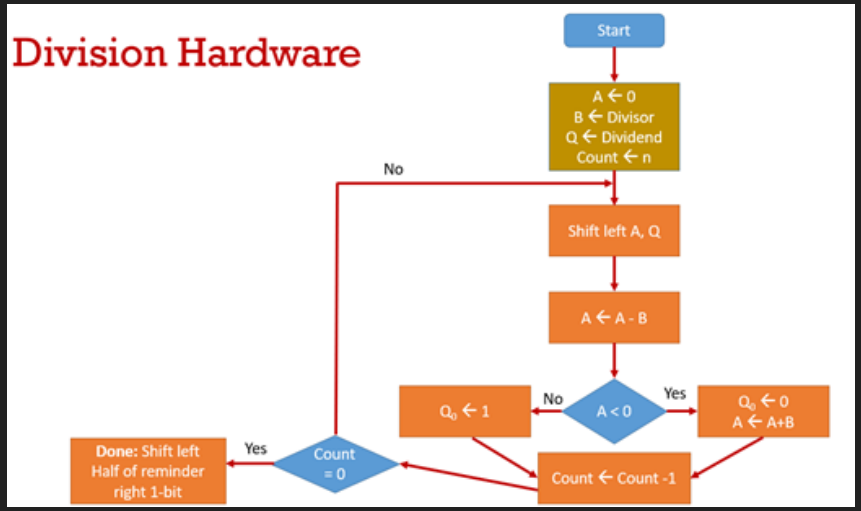
- soft-core microprocessor implemented on the Nexys A7 FPGA device
- allows the designer to develop a customized embedded system
- provides for custom hardware acceleration cores (such as an RSA core) to provide task execution
- implemented entirely in general-purpose memory and through logic cell (LC) and look-up table (LUT) connections
- implemented using the Advanced eXtensible Interface (AXI) bus

RSA Design - Overview

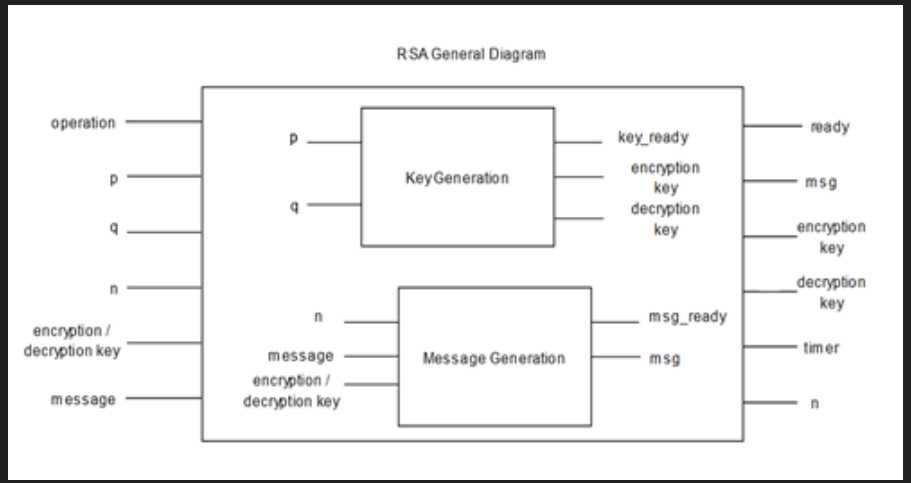


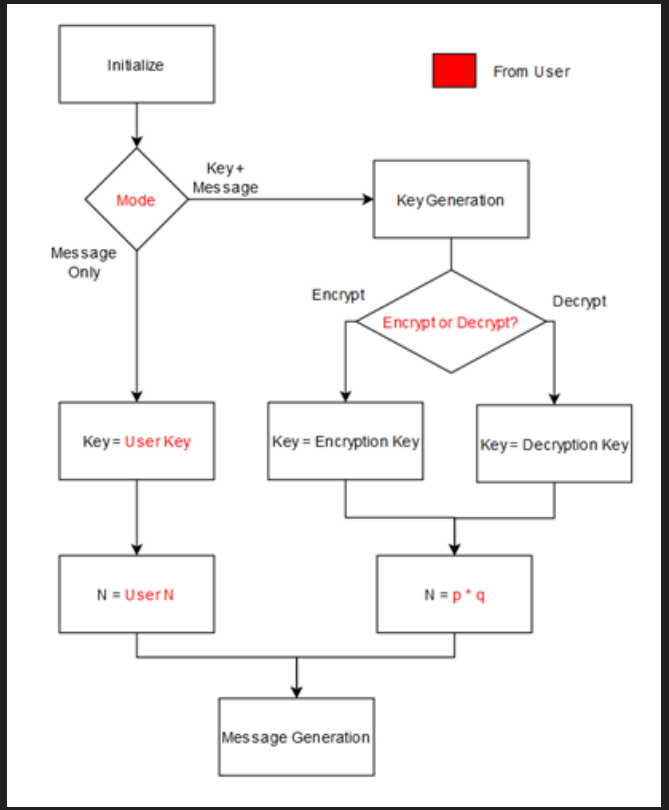
RSA Design - Core Functionalities

- Multiplication
  - Using Complex Multiplier IP provided by Xilinx
  - Maximum 32-bit signed multiplication
  - Used for calculating the totient (p*1) (q*1) and N (p * q)
  - Used for generating the message for (M^e)
- Modulus
  - Using Restorative Division Algorithm
  - Used for calculating e by ensuring the GCD(e, totient) = 1
  - Used for generating the message when (M^e)mod(N)



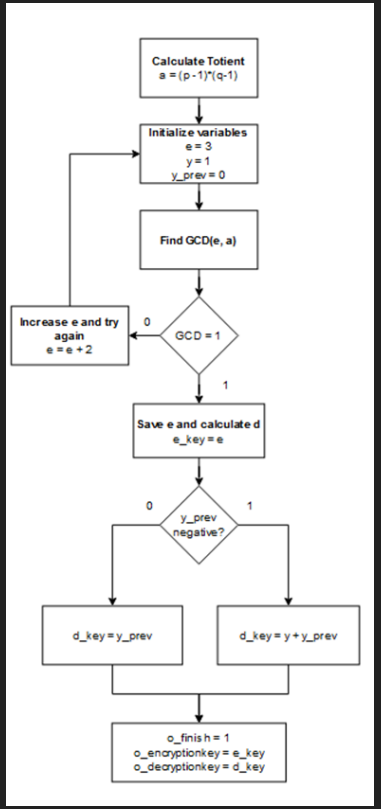
RSA Design - Top Module

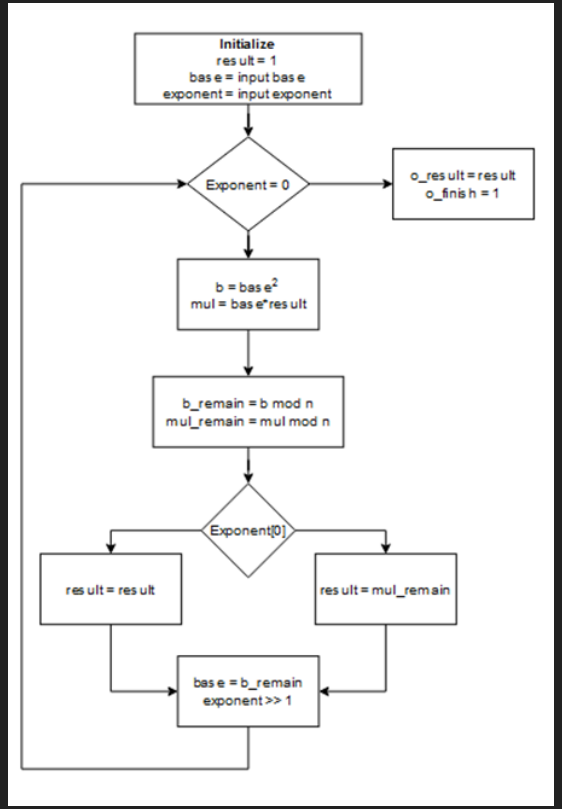




RSA Design - Sub-Modules

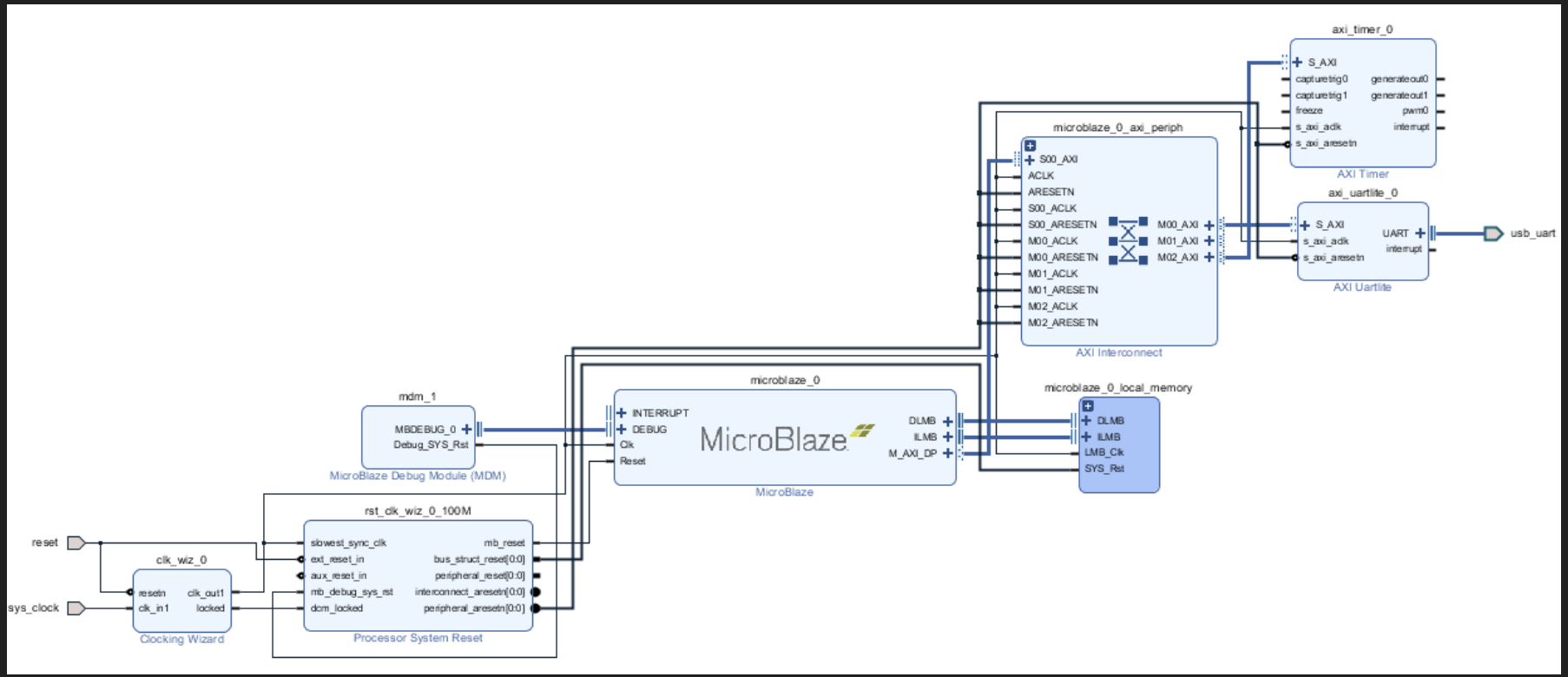
Key Generation Module





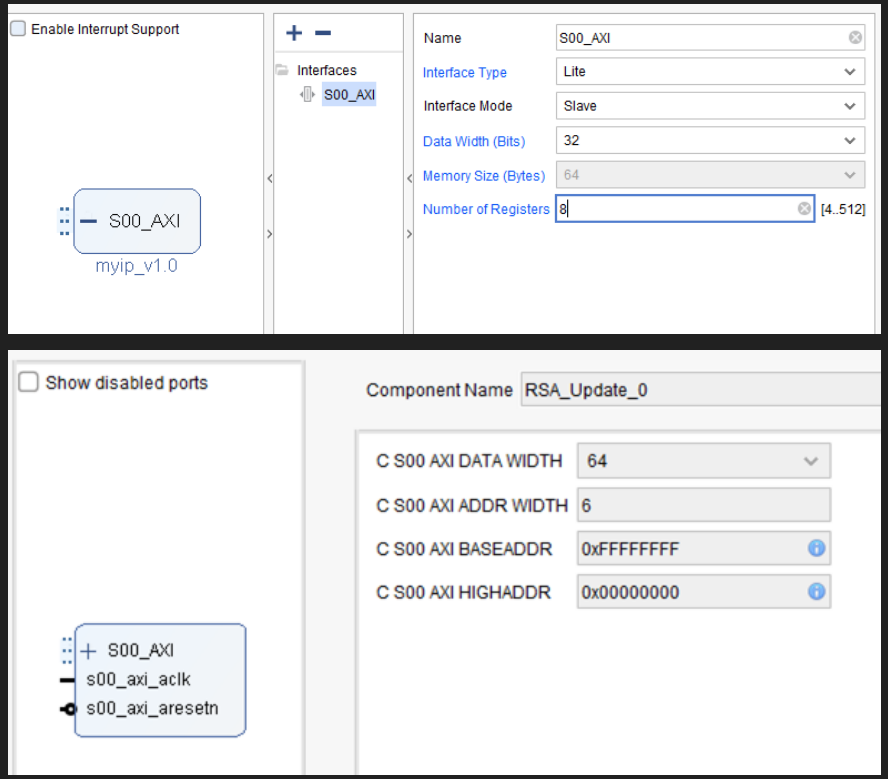
Message Generation Module

RSA Implementation - Block Design



RSA Implementation - Custom IP

- Creation done through Vivado
- Connects to AXI Bus
- Allows for customization
  - Data Width (32 bits default, more needs to be modified within code)
  - Number of Registers, each with X data width



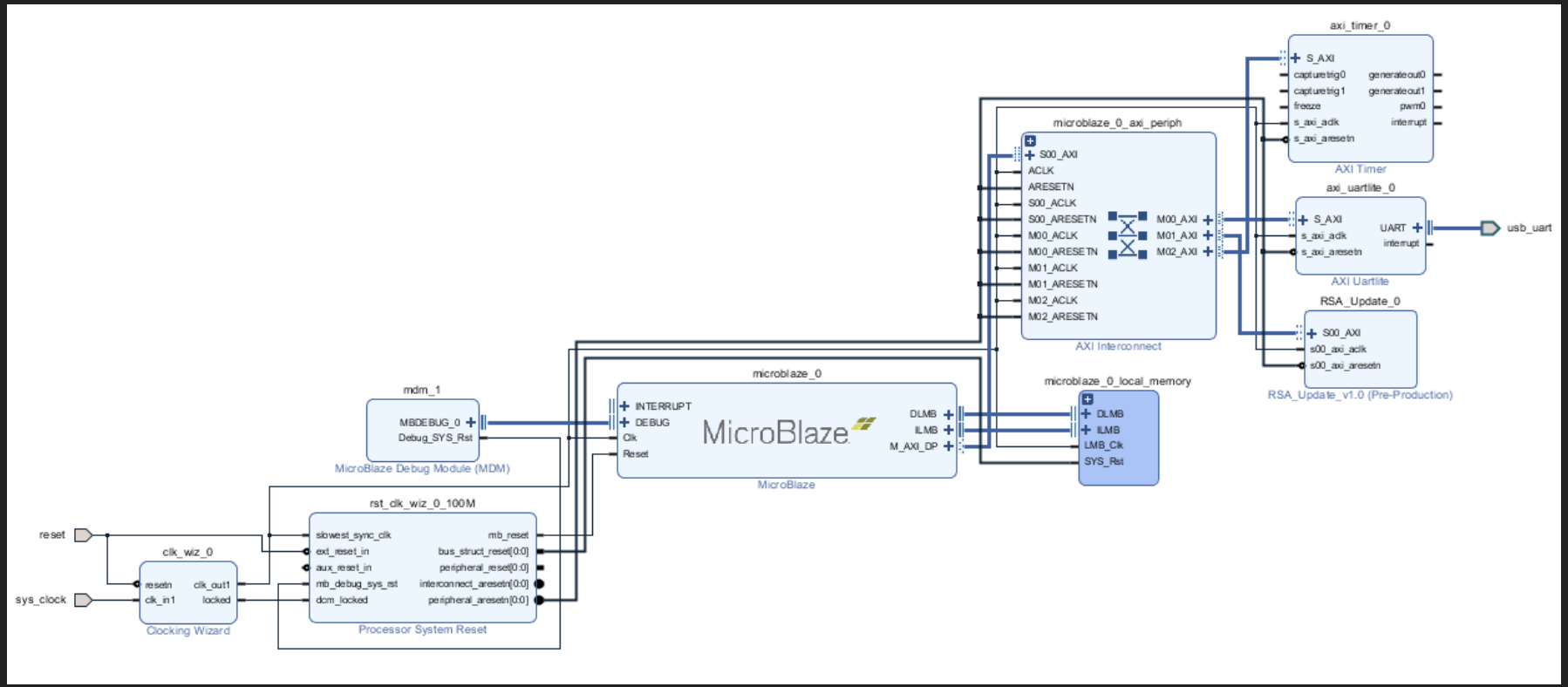
RSA Implementation - Custom IP Contd.

- Each register is implemented with each address holding 8-bits.
  - For example, an IP with 64-bit data width has every register consisting of 8 addresses
  - This offset is what is used when accessing the appropriate register. To access the second data use an offset of 8, third data is 16, etc.
- slv_reg[*] represents the corresponding write register
- reg_data_out assigns data according to read address assigned to AXI

```
always @(*)
begin
      // Address decoding for reading registers
      case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        4'h0   : reg_data_out <= slv_reg0;
        4'h1   : reg_data_out <= slv_reg1;
        4'h2   : reg_data_out <= slv_reg2;
        4'h3   : reg_data_out <= slv_reg3;
        4'h4   : reg_data_out <= slv_reg4;
        4'h5   : reg_data_out <= slv_reg5;
        4'h6   : reg_data_out <= slv_reg6;
        4'h7   : reg_data_out <= slv_reg7;
        default : reg_data_out <= 0;
      endcase
end
```

```
always @(*)
begin
      // Address decoding for reading registers
      case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        4'h0   : reg_data_out <= data_msg;
        4'h1   : reg_data_out <= data_encrypt;
        4'h2   : reg_data_out <= data_decrypt;
        4'h3   : reg_data_out <= n_output;
        4'h4   : reg_data_out <= clk_count;
        4'h5   : reg_data_out <= {31'b0, mod_exp_finish};
        4'h6   : reg_data_out <= slv_reg6;
        4'h7   : reg_data_out <= slv_reg7;
        default : reg_data_out <= 0;
      endcase
end
```

RSA Implementation - Implemented Block Design



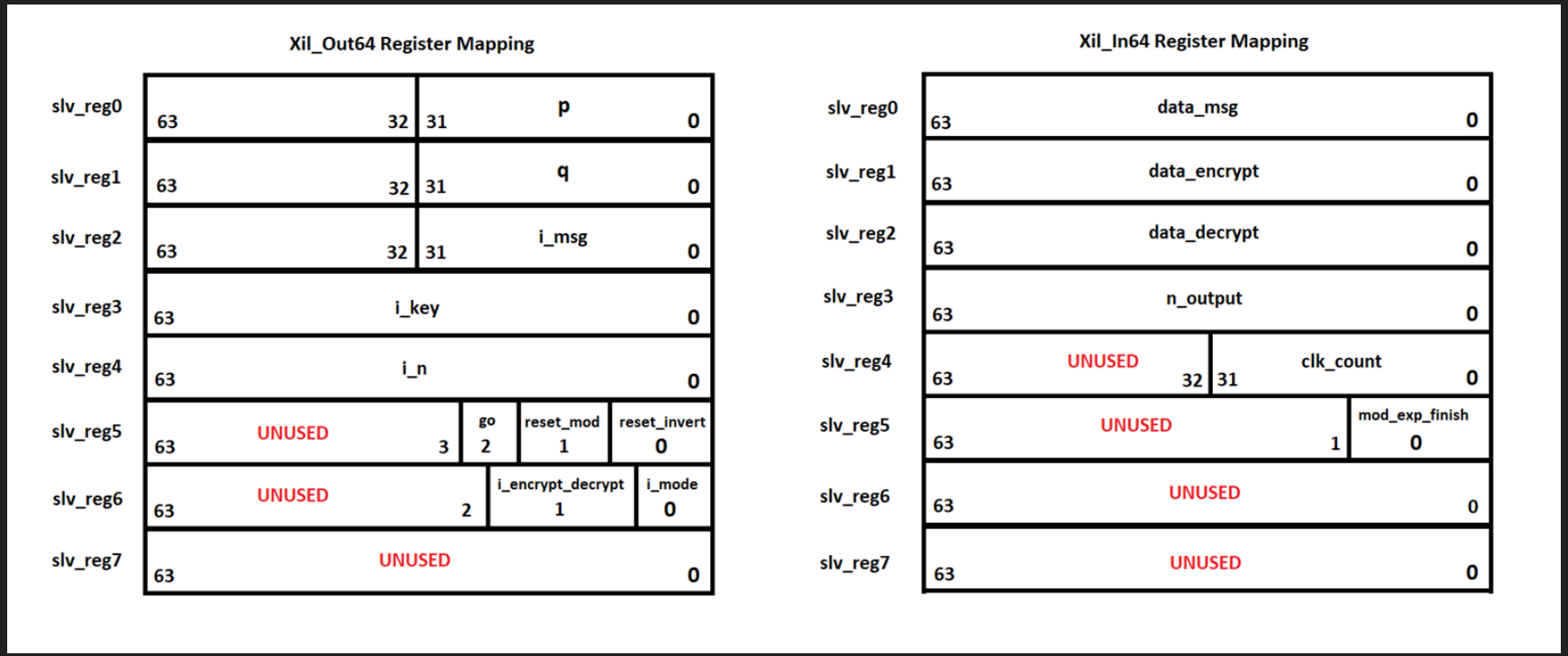
RSA Implementation - Xilinx SDK

- Xilinx SDK allows for exclusive driver to connect Xilinx IPs together and apply an embedded software layer on top of the hardware.
- Implements C or C++.
- Allows direct memory access to the modules inside of the hardware, including our custom IP.

RSA Implementation - Reading IP Registers

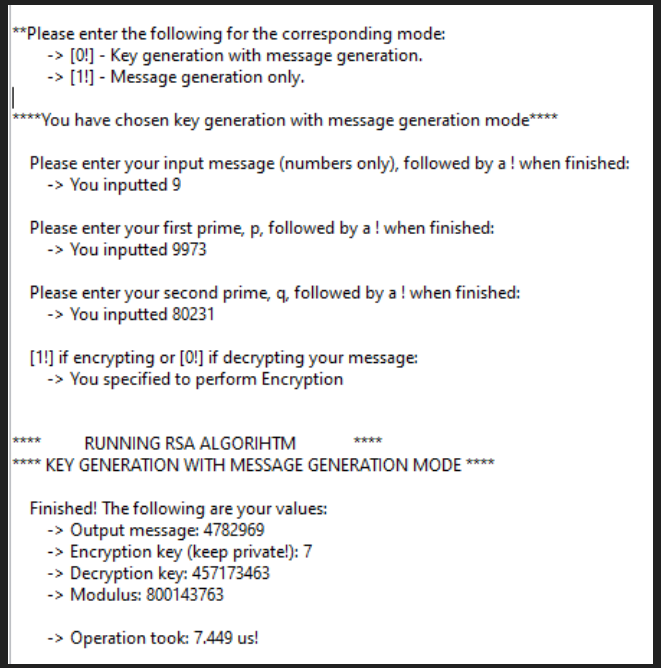
- Two core methods provide direct memory access to IPs.
- Xil_Out32/Xil_Out64
  - Write methods to the IPs.
  - Usage: Xil_Out64(IP_BASE_ADDR, ADDR_OFFSET, DATA)
- Xil_In32/Xil_In64
  - Read methods from the IPs.
  - Usage: Xil_In64(IP_BASE_ADDR, ADDR_OFFSET)

RSA Implementation - Address Map

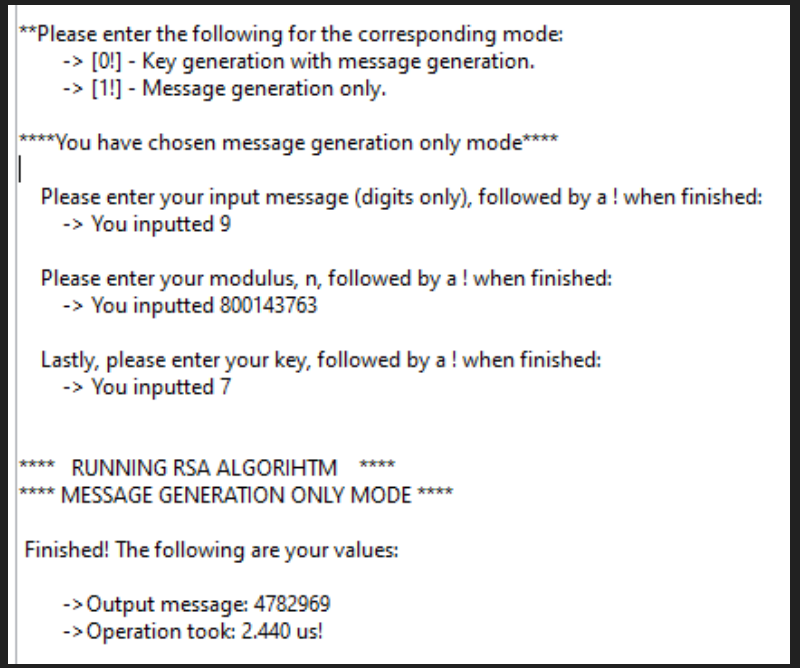


Example of User Input through UART

Key Generation + Message Generation (Encryption)

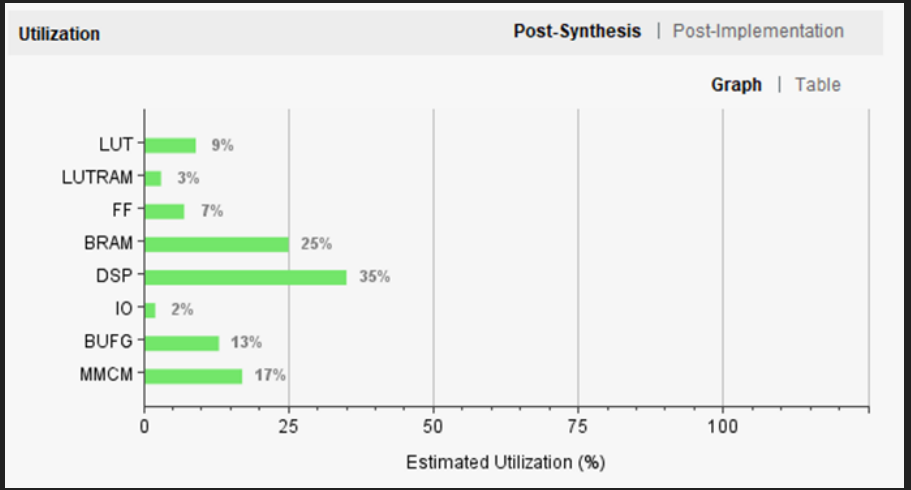


Message Generation (Encryption)



RSA MicroBlaze Resource Utilization

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



Performance Comparisons

- we ran the algorithm on the following devices: Nexys A7 FPGA (MicroBlaze), Arduino Uno Rev3, Raspberry Pi 4, ESP8266 Microcontroller
- thereby producing the following comparison table:

Device:	*Execution Time (ms)*:
Nexys A7 FPGA (MicroBlaze)	0.00584
Raspberry Pi 4	0.00200
Arduino Uno Rev3	60.0000
ESP8266 Microcontroller	331.000

Observations from Comparisons

Processor:	Clock Speed (GHz):	Execution Time (ms):
MicroBlaze Soft-Microprocessor	0.100	0.00584
Raspberry Pi 4	1.500	0.00200

- large clock speed difference but very minor execution time difference
- execution time is significantly improved utilizing hardware acceleration cores for RSA implementation as compared to sequential execution of algorithm

Performance Metrics

- Primes p = 9973 and q = 80231, Exponent e = 7, Message = 9
  - Total execution time: 30.690 μs
  - Key generation time: 5.0110 μs, 16.00%
  - Encryption time: 2.4400 μs, 8.00%
  - Decryption time: 23.239 μs, 76.00%
  - Decryption time is the majority of the total execution time
  - Key generation takes longer than encryption time
  - Decryption depends on e(encrypting exponent) and d(decryption exponent)

Performance Metrics

- Primes p = 11 and q = 13, Exponent e = 9, Message = 9
  - Total execution time: 11.488 µs
  - Key generation time: 3.490 µs, 29.67%
  - Encryption time: 2.440 µs, 21.24%
  - Decryption time: 5.639 µs, 49.08%
  - Key generation & encryption time percentages increase since decryption time decreases with shorter encryption length.
  - Going from 4 bit primes to 17 bit primes
    - Execution time increase by about 37%
    - Encryption time took the exact same amount of time (2.440 µs)

Conclusion

- CPI = {[(e)(d)]*4 + C}/[(e)(d)+i]
  - e*d >> C
  - e*d >> i
  - CPI = 4
- potential benefit for hardware acceleration embedded in systems that perform large amounts of encryption (ASICs or processing units for microprocessors)

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].