Hardware Acceleration of RSA Encryption and Decryption

Alexandre Seidy Ioshisaqui
Guilherme Kairalla Kolotelo

Prof. André Ricardo Fioravanti

Introduction

RSA encryption is an asymmetric key encryption algorithm developed by Ron Rivest, Adi Shamir and Leonard Adleman in the late 1970's with it's most notable use, along with other methods of asymmetric encryption, being on the SSL/TLS system used in web communications.

Asymmetric encryption systems work by providing a publicly available key, ranging from 32 to, more recently, 4096 bits, with which a message will be encrypted. This encrypted message, called a cypher, can now only be decrypted through the use of a private key, which should only be available to the entities responsible for decrypting the content.

As it is the case for many other core subsystems used in web applications, fast server-side algorithms are essential, since fast running algorithms result in greater energy efficiency and better client-side user experience. Through the use of hardware acceleration, several server tasks can be optimized in order to free the server's resources to other processes. One of these algorithms is the RSA asymmetric encryption algorithm, which may be implemented as a standalone device responsible for encrypting and decrypting SSL/TLS handshakes as requested by the server.

Algorithm description

A basic RSA algorithm can be described mathematically as such:

$$C = M^e mod\ P$$

Where C is the encrypted message, M is the message being encrypted, e is the public exponential, and P is the public key. Both the public exponential and public key can be used by any entity in order to encrypt the message, however only the private key holder can decrypt the cyphered message using the same algorithm:

$$M = C^e mod\ R$$

Where C is the encrypted message, M is the decrypted message, e is the public exponential, and R is the private key.

Circuit implementation

In this project, the RSA algorithm is being implemented in hardware through the use of a modular exponentiation algorithm. The modular exponentiation algorithm works by multiplying a number, and taking its modulus multiple times, until reaching the desired power, as such:

```
$Input:\ M,\ e,\ n$
$Output:\ C = M^e mod\ n$
1.  $C\ :=\ M$
2.  $for\ i\ =\ 1\ up\ to\ e-1$
    2a.  $C\ :=\ (C \times M)\ mod\ n$
3.  $return\ C$
```

That is called the ***Naïve Method***. Though there are many other methods to implement and optimize this sort of operation, one of the simplest is the ***Binary Method*** for modular exponentiation, which skips a few steps of the algorithm presented by, instead of multiplying by M and applying the $(mod\ n)$ every iteration, storing the modular powers of two $\{[M^k(mod\ n)] \times M^k\}(mod\ n)$ and then updating the final result by multiplying to it, according to the Binary Method's algorithm. A brief explanation on how these methods works is presented below:

Naïve Method

- Let us call, for simplicity, $P_k = M^k\ (mod\ n)$.
- To achieve a desired P_N, we iterate by calculating:

$$P_{k+1} = \{P_k \times [P_1(mod\ n)]\}\ (mod\ n)$$

- That stands true for the definition of P_k, noting that $P_0 = 1$ and $[M(mod\ n)]\ (mod\ n) = M(mod\ n)$.
- Then, we find the result P_N following the progression below:

$$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_6 \rightarrow P_7 \rightarrow \dots \rightarrow P_N$$

Binary Method

- Using the same definition for P_k, we change our approach by taking advantage that we already know the binary expansion of $e = (e_1, \dots, e_K)$ such that:

$$e = e_1 2^0 + e_2 2^1 + \dots + e_K 2^{K-1} = \sum_{i=1}^{K} e_i 2^{i-1}$$

- Introducing $S_i = M^{2i-1}(mod\ n)$, we now aim to calculate:

$$P_N = M^N (mod\ n) = (\prod_{i=1}^{K} S_i^{e_i})\ (mod\ n)$$

- For that, we iterate through $C_{i+1} = (C_i \times S_i^{e_i})\ (mod\ n)$ until we find $C_K = P_N$.

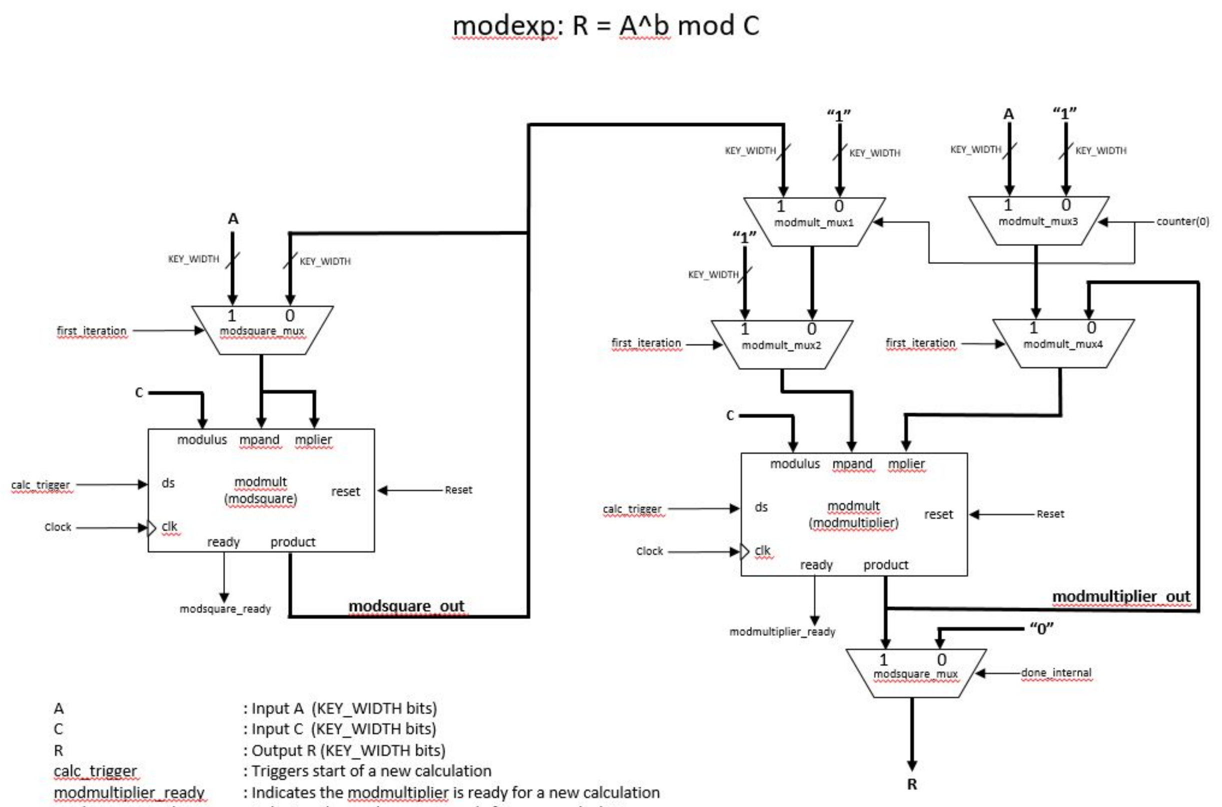
For example, if we want to calculate $C = M^{575}(mod\ n)$, we have:

- $e = (575)_{10} = (1000111111)_2$ (where e_i is the *i-th* binary digit from right to left)

i	1	2	3	4	5	6	7	8	9	10
S_i	P_1	P_2	P_4	P_8	P_{16}	P_{32}	P_{64}	P_{128}	P_{256}	P_{512}
e_i	1	1	1	1	1	1	0	0	0	1
C	P_1	P_3	P_7	P_{15}	P_{31}	P_{63}	P_{63}	P_{63}	P_{63}	P_{575}

To implement this modular exponentiation method as a circuit, two components that performs modular multiplications are used, one for updating the values of S_i and the other is used to update the value of the result $C_k = C_{k-1} \times S_k^{e_k}$. One difference in the actual implementation though, is that instead of multiplying by one when $e_i = 0$, we simply skip to the next iteration.

The circuit implementing the binary modular exponentiation method can be seen in Figure 1.:



A	: Input A (KEY_WIDTH bits)
C	: Input C (KEY_WIDTH bits)
R	: Output R (KEY_WIDTH bits)
calc_trigger	: Triggers start of a new calculation
modmultiplier_ready	: Indicates the modmultiplier is ready for a new calculation
modsquare_ready	: Indicates the modsquare is ready for a new calculation
modsquare_out	: Output of modsqaure (KEY_WIDTH bits)
modmultiplier_out	: Output of modmultiplier (KEY_WIDTH bits)
counter	: Binary exponentiation arbiter
done_internal	: Internal done signal, indicates when computation is complete

Figure 1. Circuit implementation of the binary modular exponentiation method. The circuits generating the control signals for this implementation are shown in Appendix I.

A modular multiplication circuit must be used in order to execute the $M\ (mod\ n)$ operation iteratively. This circuit has many possible implementations, the simplest being multiply and reduce, where the multiplication is executed first, followed by the modulus operation. However more efficient methods exist, such as the Montgomery method, which makes use of computationally efficient power of 2 operations for increased speeds.

Interfacing with a host device

In order for this standalone implementation of the RSA algorithm to communicate with a host device, such as a web server or a workstation, a communication interface between these devices must be used. Many methods of communication between peripherals exist, such as USB, RS232, CAN and Ethernet. For this implementation a RS232 serial communication will be used as a proof of concept for other faster and more secure communication protocols. In the case o a 1024 bit message, and a 115200 bit per second transfer speed, such a transfer is expected to take about 8.8ms.

In this implementation a character indicating the operation to be performed, either encryption or decryption, is prepended to the message. The first byte of the this modified data will then be used to select the operation and set the devices to execute it on the remaining bytes of the message, for example:

Taking 12 byte string "`Hello World!`" as the message to be encrypted, the 2 byte command '`e`' will be prepended, indicating an encryption operation, and the message will be zero-padded in order to fill all 1024 bits of data. The final ASCII string will be equivalent to "`e\x00\x00...\x00Hello World!`", with the 116 bytes between the operation byte and the message padded with zeros.

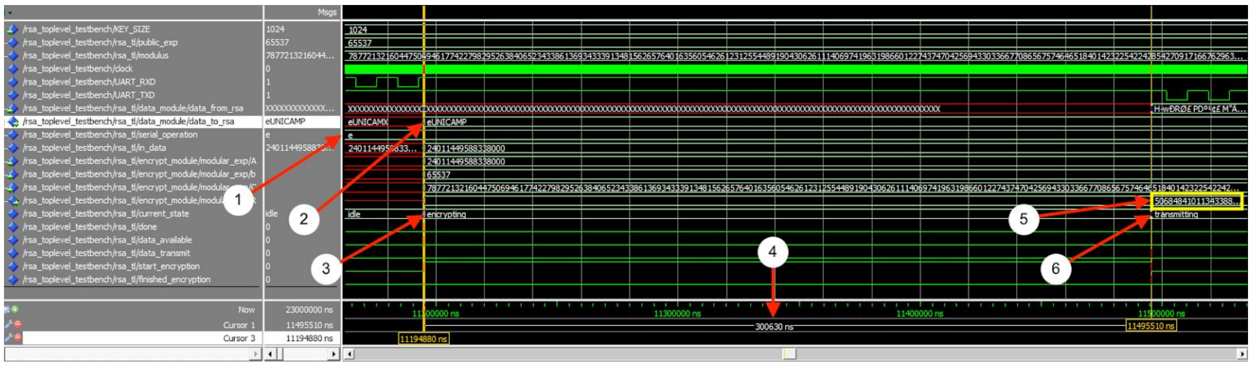
After the operation has completed, and a response is available, a transmit operation is requested and the response is transmitted, in a similar fashion, back to the host device.

Simulation

The basic behavior of this implementation can be understood as a simple FSM of 6 states: ***reset***, ***idle***, ***receiving***, ***encrypting***, ***decrypting*** and ***transmitting***.

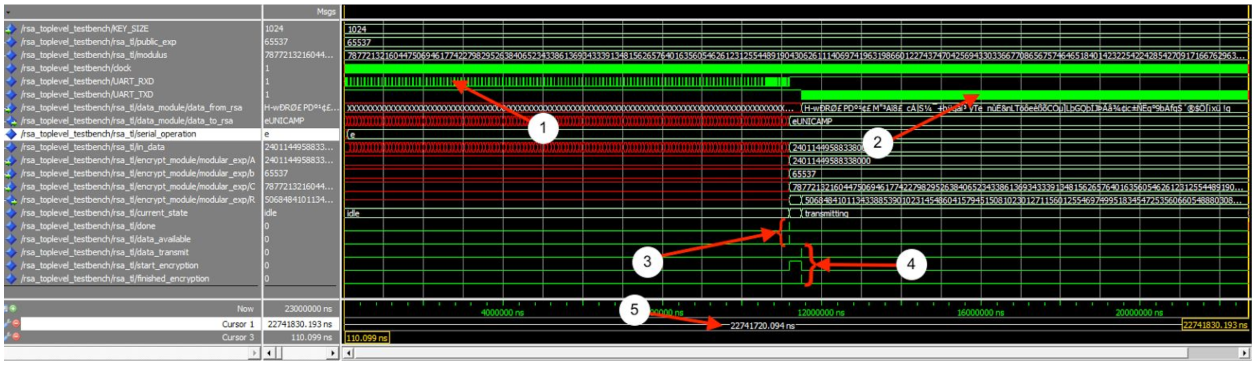
- **Reset**: A basic beginning state, initializing control signals related to serial communication and RSA calculations (modular exponentiation circuit).
- **Idle**: A standard state where the device is neither performing RSA calculations (encrypting or decrypting) nor serial communications (receiving or transmitting).
- **Receiving**: The device changes to this state when the transfer of data of a given predefined size (1032-bits at the simulation, 40-bits at FPGA) is finished (Figure 3. (3) *data_available*). Then, an operation (encrypting or decrypting) is decoded based on the first 8 bits of data (Figure 2. (1) *serial_operation*), determining the next state. The remaining bits constitutes the actual data to be processed (original message or encrypted message).

- **Encrypting**: The encryption component is a modular exponentiation circuit that takes as its input signals: original message, public exponent and modulus. These are fixed in this implementation, but could be set (given that the keys and modulus were valid). At the end of the encryption process, the result of the operation becomes available (Figure 2. (5)) and it is indicated that the operation has finished (Figure 3. (4) *finished_encryption*).
- **Decrypting**: Just as the encryption component, the decryption is executed by a modular exponentiation circuit taking input signals: encrypted message, private exponent and modulus.
- **Transmitting**: After the encryption/decryption procedure is finished, the device then transmits back the corresponding processed data through the serial communication through control signals (Figure 3. (4) *data_transmit*)



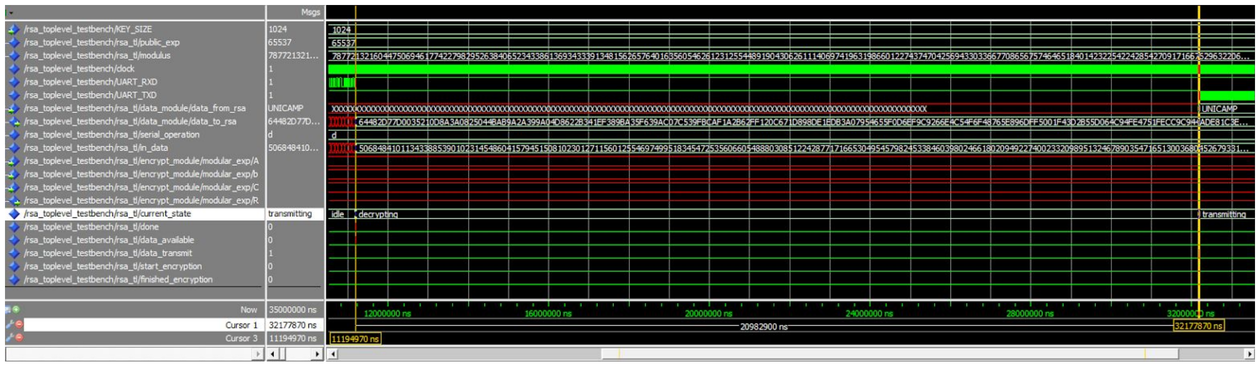
/rsa_toplevel_testbench/KEY_SIZE
/rsa_toplevel_testbench/rsa_tl/public_exp
/rsa_toplevel_testbench/rsa_tl/modulus
/rsa_toplevel_testbench/clock
/rsa_toplevel_testbench/UART_RXD
/rsa_toplevel_testbench/UART_TXD
/rsa_toplevel_testbench/rsa_tl/data_module/data_from_rsa
/rsa_toplevel_testbench/rsa_tl/data_module/data_to_rsa
/rsa_toplevel_testbench/rsa_tl/serial_operation
/rsa_toplevel_testbench/rsa_tl/in_data
/rsa_toplevel_testbench/rsa_tl/encrypt_module/modular_exp/A
/rsa_toplevel_testbench/rsa_tl/encrypt_module/modular_exp/b
/rsa_toplevel_testbench/rsa_tl/current_state
/rsa_toplevel_testbench/rsa_tl/done
/rsa_toplevel_testbench/rsa_tl/data_available
/rsa_toplevel_testbench/rsa_tl/data_transmit
/rsa_toplevel_testbench/rsa_tl/start_encryption
/rsa_toplevel_testbench/rsa_tl/finished_encryption
1
2
3
4
5
6
idle
encrypting
transmitting
eUNICAMP
300630 ns
11194880 ns
11495510 ns

Figure 2. Zoomed-in view of encryption process.
(1) *serial_operation* = *'e'* (encryption); (2) *data_to_rsa*: operation('*e*')+message(*'UNICAMP'*); (3) *current_state*: changing from '*idle*' to '*encrypting*'; (4) Time spent: 300630 ns; (5) Encrypted message (same as *data_from_rsa*); (6) *current_state*: changing from '*encrypting*' to '*transmitting*'.



/rsa_toplevel_testbench/UART_RXD
/rsa_toplevel_testbench/UART_TXD
/rsa_toplevel_testbench/rsa_tl/current_state
/rsa_toplevel_testbench/rsa_tl/done
/rsa_toplevel_testbench/rsa_tl/data_available
/rsa_toplevel_testbench/rsa_tl/data_transmit
/rsa_toplevel_testbench/rsa_tl/start_encryption
/rsa_toplevel_testbench/rsa_tl/finished_encryption
1
2
3
4
5
eUNICAMP
transmitting
22741720.094 ns

Figure 3. Full view of encryption process.
(1) UART_RXD: receiving data through serial communication; (2) UART_TXD: transmitting data through serial communication; (3) control signals (*done* and *data_available*) indicating a data has been completely received; (4) control signals (*data_transmit*, *start_encryption* and *finished_encryption*) indicating the beginning and ending of the encryption procedure and transition to the transmission state; (5) Time spent: 22741720.094 ns (the encryption state corresponds to 1.32% of the total time).



	Msgs
/rsa_toplevel_testbench/KEY_SIZE	1024
/rsa_toplevel_testbench/rsa_tl/public_exp	65537
/rsa_toplevel_testbench/rsa_tl/modulus	787721321...
/rsa_toplevel_testbench/clock	1
/rsa_toplevel_testbench/UART_RXD	1
/rsa_toplevel_testbench/UART_TXD	1
/rsa_toplevel_testbench/rsa_tl/data_module/data_from_rsa	UNICAMP
/rsa_toplevel_testbench/rsa_tl/data_module/data_to_rsa	64482D77D...
/rsa_toplevel_testbench/rsa_tl/serial_operation	d
/rsa_toplevel_testbench/rsa_tl/in_data	506848410...
/rsa_toplevel_testbench/rsa_tl/encrypt_module/modular_exp/A	
/rsa_toplevel_testbench/rsa_tl/encrypt_module/modular_exp/b	
/rsa_toplevel_testbench/rsa_tl/encrypt_module/modular_exp/C	
/rsa_toplevel_testbench/rsa_tl/encrypt_module/modular_exp/R	
/rsa_toplevel_testbench/rsa_tl/current_state	transmitting
/rsa_toplevel_testbench/rsa_tl/done	0
/rsa_toplevel_testbench/rsa_tl/data_available	0
/rsa_toplevel_testbench/rsa_tl/data_transmit	1
/rsa_toplevel_testbench/rsa_tl/start_encryption	0
/rsa_toplevel_testbench/rsa_tl/finished_encryption	0
Now	35000000 ns
Cursor 1	32177870 ns
Cursor 3	11194970 ns

Figure 4. Full view of decryption process.
Now, sending at the *in_data* signal, the previously encrypted message with the *serial_operation* = *d* (*decryption*), it can be seen that the decrypted message (*data_from_rsa*) shows us back the same original message (‘*UNICAMP*’).

We can observe from the simulations above that, during encryption of a small message, most of the time is spent receiving and transmitting data through the serial interface. For this example, with the message `"UNICAMP"` the following results are obtained:

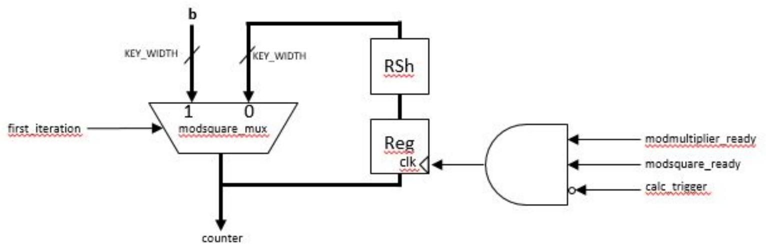
Operation	Time spent [ms]
Full encryption operation	22.741
Full decryption operation	43.425
Receiving or transmitting @ 115200bps	11.221
Encrypting	0.3
Decrypting	20.983

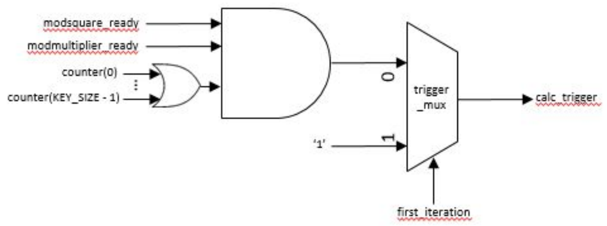
After the encryption process has completed, it can be seen on Figure 2. (5) that the encrypted data produced was then fed to the decryption circuit through the *in_data* signal as shown on Figure 4. For smaller messages, the encryption time can be very small, however decryption time, due to the usually larger operand, takes considerably longer.

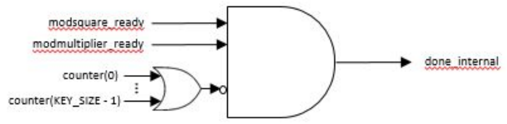
References

- https://certsimple.com/blog/measuring-ssl-rsa-keys
- Koç, Çetin Kaya, 1994. High-Speed RSA Implementation. RSA Laboratories.

Appendix I


b
KEY_WIDTH
KEY_WIDTH
1
0
modsquare_mux
first_iteration
RSh
Reg
clk
modmultiplier_ready
modsquare_ready
calc_trigger
counter


modsquare_ready
modmultiplier_ready
counter(0)
counter(KEY_SIZE - 1)
0
1
trigger _mux
'1'
calc_trigger
first_iteration


modsquare_ready
modmultiplier_ready
counter(0)
counter(KEY_SIZE - 1)
done_internal