RSA HARDWARE ACCELERATOR

10 LOGO

Group	group10
Authors	Nicolas Abril Abdullah Shaaban Saad Allam Youssef Mohamad Youssef
Date	2023-10-06

INSTRUCTIONS

Fill out all parts of this document that are marked in green.

INTRODUCTION

This document contains the requirements, design specification and test plan for an RSA encryption circuit. The document also specifies key milestones, deliverables and the criteria used for evaluating the work of the group.

This document is written in such a way that it facilitates quick and efficient evaluation of the work done by each group and is not a template for how to write a typical project thesis or master thesis report.

CODE OF HONOR

We hereby declare that this design has been developed by us. This means that the high-level model, the microarchitecture, the RTL code and the testbench code has all been developed by the team.

Papers we have read that e.g. describes different ways of doing modular exponentiation are listed in the reference section.

We understand that attempts of plagiarism can result in the grade "F".

Signature of all team members

DESIGN REQUIREMENS

The design requirements are shown in Table 1. The requirements have been divided into functional (FUNC) requirements, requirements for performance, power and area (PPA), interface requirements (INT) and configuration requirements (CONF)

Priority is given for each requirement. The rightmost column contains a checkbox. Write **OK** in that if your design has met the corresponding requirement.

Table 1. RSA Hardware accelerator design requirements

Requirement ID	Priority	Description	Check
REQ_FUNC_01	MUST	The design must implement a function that can compute modular exponentiation $X = Y^k \bmod n$	OK
REQ_FUNC_02	MUST	The design must be able to encrypt and decrypt message blocks using modular exponentiation: Encryption: $C = M^e \bmod n$, $M < n$, $C < n$, $e < n$ Decryption: $M = C^d \bmod n$, $M < n$, $C < n$, $d < n$	OK
REQ_PPA_01	MUST	Encrypt/decrypt a message of length 256 bits as fast as possible.	OK
REQ_PPA_02	MUST	The design must fit inside the Zynq XC7Z020 FPGA on the Digilent Pynq-Z1 board.	
REQ_PPA_03	MUST	There is no requirement for the clock frequency of the programmable logic. The platform supports any clock frequency.	OK
REQ_PPA_04	SHOULD	The hardware accelerator should run testcase 4 faster than 400 ms.	
REQ_INT_01	MUST	The RSA design must be integrated as a hardware accelerator inside the Zynq SoC. It must be managed by the CPU and made accessible through the Juniper notebook interface.	
REQ_INT_02	SHOULD	The design should implement memory mapped status registers, performance counters and other mechanisms for debugging of features and performance at system level.	
REQ_INT_03	MUST	The design must have one AXI-Lite Slave interface to enable access of memory-mapped registers.	
REQ_INT_04	MUST	The design must have one AXI stream slave interface for input messages that shall be encrypted(decrypted) and one AXI stream master interface for output messages that have been encrypted(decrypted).	
REQ_CONF_01	SHOULD	The design should be optimized for 256 bit block/message/key size.	OK

DEVELOPMENT, DOCUMENTATION AND CODE REQUIREMENTS

This document has a lot of different sections the group must fill out. These sections are all marked in green. In addition to this document, the group shall also submit model code, RTL code for the design and code for the verification environments. These requirements are captured in Table 2

The rightmost column contains a checkbox. Write **OK** in that if your group has met the corresponding requirement.

Table 2. RSA Hardware accelerator documentation and code requirements

Requirement ID	Priority	Description	Check
REQ_DEV_01	MUST	The development is broken down into milestones. The group must deliver the milestones on time.	OK
REQ_DOC_01	MUST	All green parts of this document must be filled out.	OK
REQ_DOC_02	MUST	This document must contain information about algorithm used for computing modular multiplication.	OK
REQ_DOC_03	MUST	This document must contain description of the design including microarchitecture diagrams.	OK
REQ_DOC_04	MUST	This document must contain verification plan.	
REQ_DOC_05	MUST	This document must contain results from performance measurements.	
REQ_CODE_01	MUST	RTL code for the design must be attached the final delivery bundle.	
REQ_CODE_02	MUST	Code for the testbench(es) developed by the group must be attached the final delivery bundle.	
REQ_CODE_03	MUST	High level model code (Python, Matlab, C++) developed by the group must be attached the final delivery bundle.	OK

MILESTONES

A considerable amount of work and effort is needed in order to develop an RSA encryption circuit. The development is therefore split up into a set of milestones as listed in Table 3

The rightmost column contains a checkbox. Write **OK** in that if your group has met the corresponding milestone.

Table 3. Term project schedule and milestones

Milestone	Date	Delivery instructions	Description	Check
Form groups	SEP 4	Sign up on Blackboard	Form term project groups	OK
Study algorithms and pick one	SEP 20	Nothing to upload	Study algorithms and pick one	OK
High level model	SEP 27	Upload code on Blackboard	Implement the algorithm in python or another high level language.	OK
Microarchitecture	OCT 6	Upload diagram on Blackboard	Draw microarchitecture diagram for hardware design in this datasheet.	OK
Performance estimate	OCT 6	Estimate performance. Upload to Blackboard.	Estimate the time needed to encrypt/decrypt a block, in this datasheet.	OK
Microarchitecture review/presentation	OCT 6	Give presentation in class.	Staff and fellow students (peers) reviews the solutions proposed by each team and gives feedback.	OK
RTL Code (Alpha)	NOV 3	Upload RTL code to Blackboard.	Write synthesizable register transfer level code.	
Testbench (Alpha)	NOV 10	Upload Testbench to Blackboard.	Write testbenches for testing the design.	
Working on FPGA (Alpha)	NOV 17	Upload PPA on Blackboard.	Design working on FPGA.	
Hand in this document and all pieces of source code	NOV 24	Upload this document together with all pieces of source code on Blackboard.	Hand in this document	

DESIGN AND VERIFICATION PROCESS

When designing a hardware design, it is important to follow the following steps:

1) **Capture, understand and analyze all requirements.**
2) **Design exploration:**
   - Create a high level model that allow you to quickly and easily compute functionally correct output for a given set of inputs.
   - Come up with a way to efficiently search through the design space in order to find the design that satisfy the requirements.
   - Evaluate and improve the PPA of different alternative solutions.
3) **Write design specification:**
   - Describe the design you intend to make
   - Draw microarchitecture diagrams
4) **Design and verification:**

- Write RTL code according to the design specification
- Verify that the design is working using testbenches and other verification environments

5) **Implement the design:**
   - Synthesize the design
   - Run Place & Route
6) **Test on FPGA**
   - Run performance benchmarks on FPGA prototype platform

During the work with the design, verification and implementation of the RSA encryption circuit, you will go through all these phases.

HIGH LEVEL MODEL CODE (9 POINTS)

```
def mon_exp(msg, exponent, modulo):
    r2_mod = (1 << (2*k)) % modulo
    product = mon_pro(msg, r2_mod, modulo)
    result = mon_pro(1, r2_mod, modulo)
    for i in range(k):
        if get_bit(exponent, i):
            result = mon_pro(result, product, modulo)
        product = mon_pro(product, product, modulo)
    result = mon_pro(result, 1, modulo)
    return result

def mon_pro(A, B, n):
    bn = B + n
    u = 0
    for i in range(k):
        qi = (u & 1) ^ (get_bit(A, i) & (B & 1))
        ai = get_bit(A, i)
        if not qi and not ai:
            u = u
        elif not qi and ai:
            u = u + B
        elif qi and not ai:
            u = u + n
        elif qi and ai:
            u = u + bn
        u = u >> 1
    if u > n:
        u = u - n
    return u
```

Figure 1. High level model of modular multiplication and modular exponentiation.

We used the Montgomery product to implement the modular exponentiation. In both the product and in the exponentiation we read the bits of the numbers from right to left since that allows us to more easily have things in parallel.

The Montgomery product consists of the operation **P = A * B * 2^-k mod n**, where **A** and **B** are the numbers we're multiplying, **n** being the modulus and **k** the bitwidth of the modulus or **ceil(log2 n)**. This is implemented with an iterative addition approach, with one addition for each bit of the operands, followed by a halving of the number at each iteration. It can be thought as this operation: **P = sum with i from 0 to k-1 of (A * 2^i-k if B(i) = 1, 0 otherwise) mod n**.

The exponentiation is implemented with iterated multiplication using this Montgomery product module. At each step we multiply our result with the message, followed by a squaring of the message, with each multiplication accounting for 1 bit of the exponent.

To implement the exponentiation using the Montgomery product, we must first convert our numbers into the Montgomery domain. Since the Montgomery product shifts the result right by k bits, we must first shift the numbers left to get the correct result. By the properties of modular arithmetic, clipping the operand of the multiplication to mod n does not change the result, so we're able to work always in this range. This simplifies the hardware and allows us to reuse the Montgomery product to this initial shifting left and the shifting right at the end to bring the result back from the Montgomery domain.

SYSTEM OVERVIEW

The RSA encryption platform consists of a hardware design and a software driver stack that enables the user to interact with the hardware.

The hardware is implemented on a PYNQ-Z1 [1,2] development board. This board is equipped with a Xilinx ZYNQ-7020[3] system on chip. The ZYNQ contains a processing subsystem with two Arm CPUs and a programmable logic part. Our RSA accelerator is placed within the programmable logic. It is connected to the processing system through an AXI[4,5] interconnect as show in Figure 2.

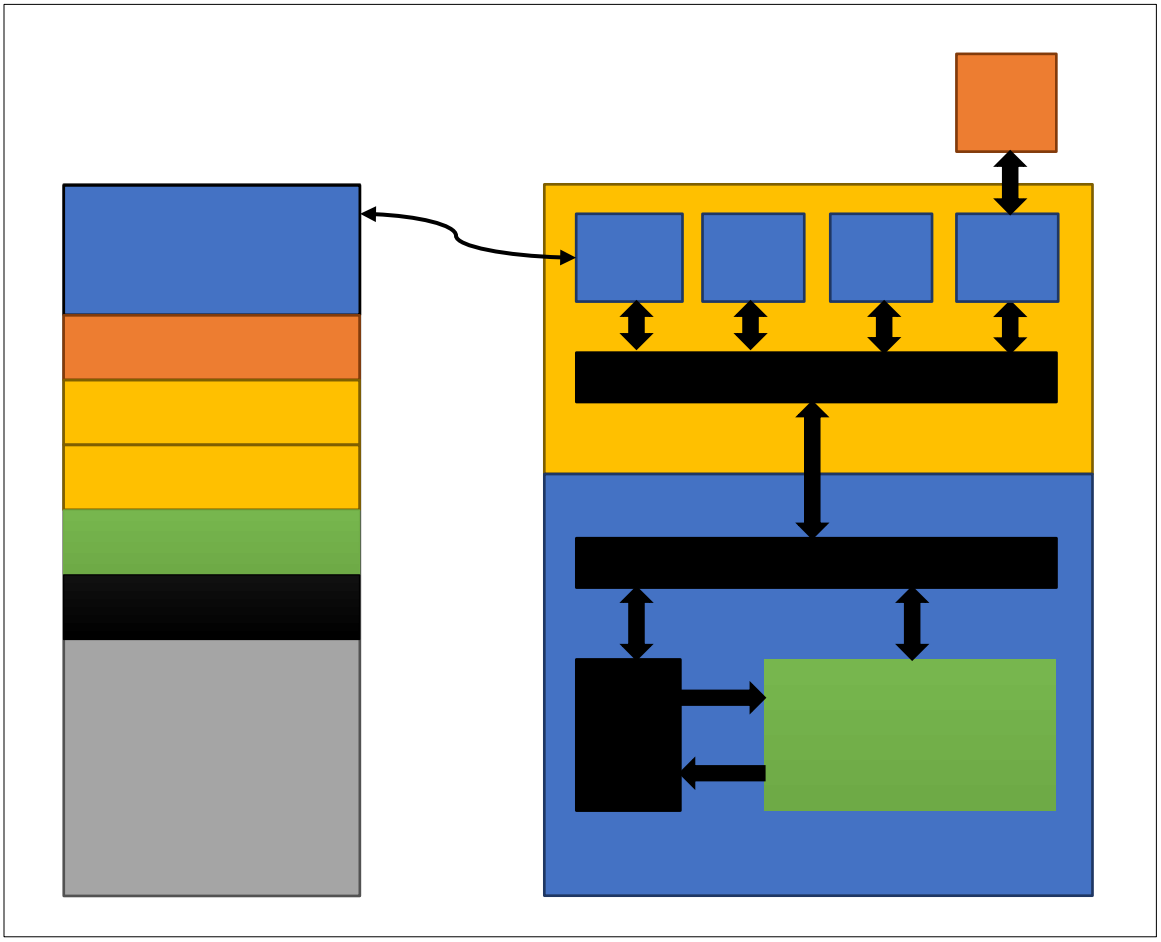


Figure 2. Software and hardware components of the RSA encryption platform.

FLOW CONTROL THROUGH VALID/READY HANDSHAKING

In a digital system, such as the one we are going to construct, data is transferred from block to block. It is important that data is transferred in such a way that none of the blocks gets ahead of other blocks and e.g. do not send data before the receiver is ready to accept new incoming data. It is necessary for some sort of flow control.

One very common flow control protocol is valid/ready handshaking. The protocol is illustrated in Figure 3 and Figure 4 (see also [6], page 480).

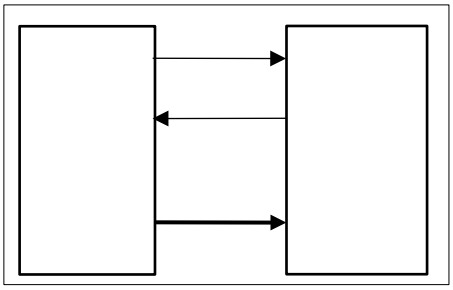


Figure 3. Sender and Receiver exchanging data.

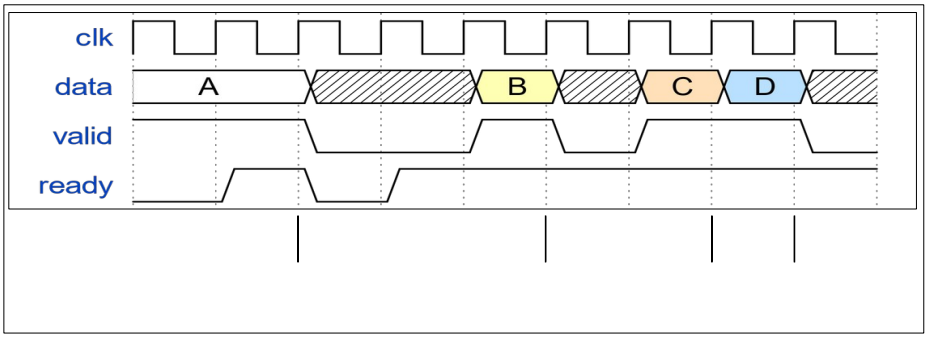


Figure 4. Valid - Ready handshaking. Timing diagram.

When a sender wants to send data to a receiver. It will signal that **data** is present and valid by asserting the **valid** signal. When the receiver can receive data, the receiver signals this by setting the **ready** signal high. The **data** will be successfully transferred from the sender to the receiver on the first positive edge of the clock where both the **valid** signal and the **ready** signal is high at the same time.

At the transfer of **A** in Figure 4 above, the sender had to wait for the **ready** signal of the receiver. When **B** and **C** were transferred the receiver was **ready** and waiting for the sender to send data. When both **ready** and **valid** remains high, a new datum is transferred in every cycle (this is the case with **D**).

If the valid signal is high and the ready signal is low, then none of the signals must change value until the ready signal has become high.

All the interfaces between modules within this project (that needs flow control) is based on valid-ready handshaking. It is also the protocol used for transferring data on AXI interfaces.

RSA CORE INTERFACE

The **RSA ACCELERATOR** from Figure 2 is shown in more detail in Figure 5. The **rsa_core** block in the middle is the block that does the modular exponentiation calculations. This is the module that you are going to implement as a part of the term project in TFE4141 Design of digital systems 1. The other blocks (rsa_regio, rsa_msgin and rsa_msgout) are already made.

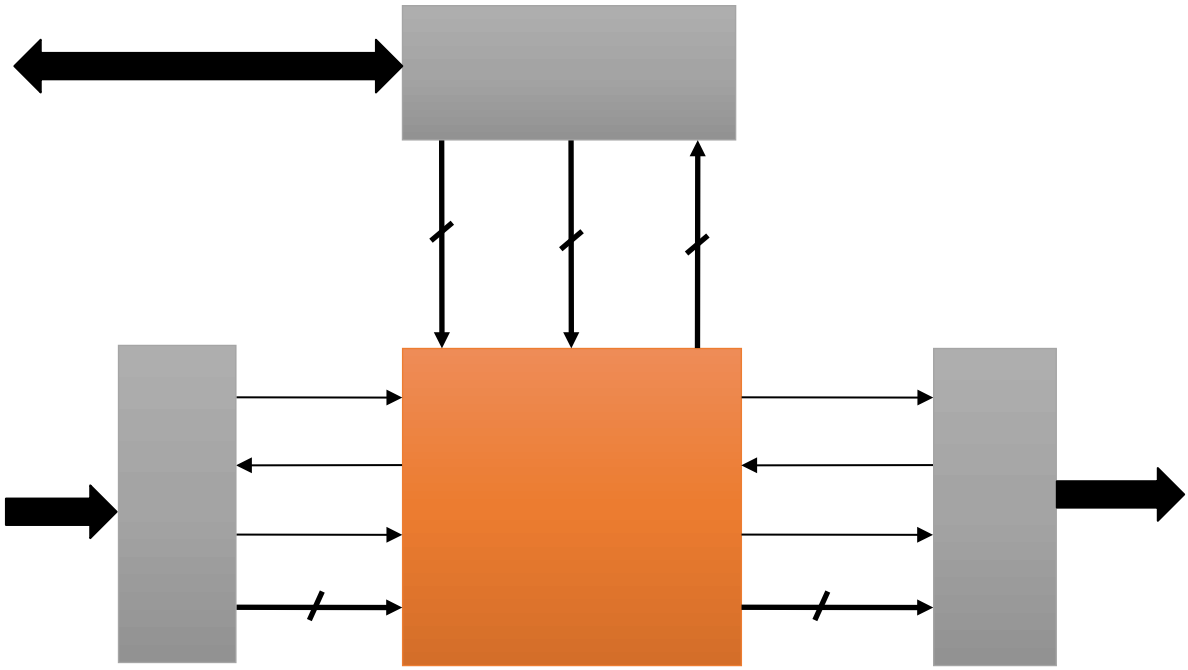


Figure 5. Main blocks within the RSA ACCELERATOR

The **rsa_regio** unit contains key registers. These registers can be written and read by a master in the system through the AXI master interface. The keys are sent out of the **rsa_regio** module to the **rsa_core** module where they are used during the encryption process. The **rsa_status** signal comes from the **rsa_core** and is written to one of the registers. This can be used by the CPU to retrieve information about the status of the rsa_accelerator. It is up to the group to decide what status information that could be interesting.

Messages that will be encrypted/decrypted are sent in to the **rsa_core** from the **rsa_msgin** block in a continuous stream (**msgin_***). The results are sent from the **rsa_core** to the **rsa_msgout** block through another stream (**msgout_*).** The diagram in Figure 6 shows how messages are sent in and out of rsa_core.

The message **M<n>** on **msgin_data** is transferred from the sender (rsa_msgin) to the receiver (rsa_core) on the first rising edge of **clk** when **msgin_valid** and **msgin_ready** are both high at the same time. The **msgin_last** signal indicates whether **M<n>** is the last message in the stream or not.

The message **C<n>** on **msgout_data** is transferred from the sender (rsa_core) to the receiver (rsa_msgout) on the first rising edge of **clk** when **msgout_valid** and **msgout_ready** are both high at the same time. The **msgout_last** signal indicates whether **C<n>** was the last message in the stream or not. It must therefore be identical to the value **msgin_last** had during the transfer of **M<n>**.

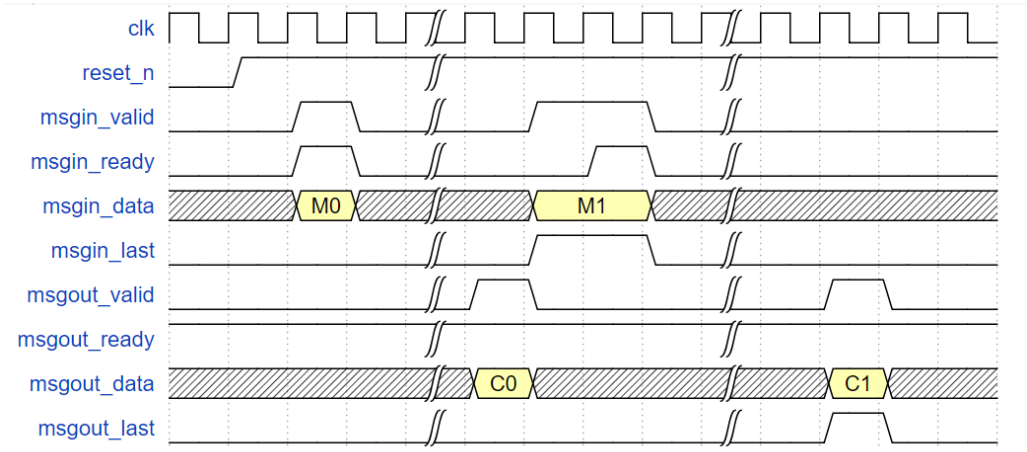
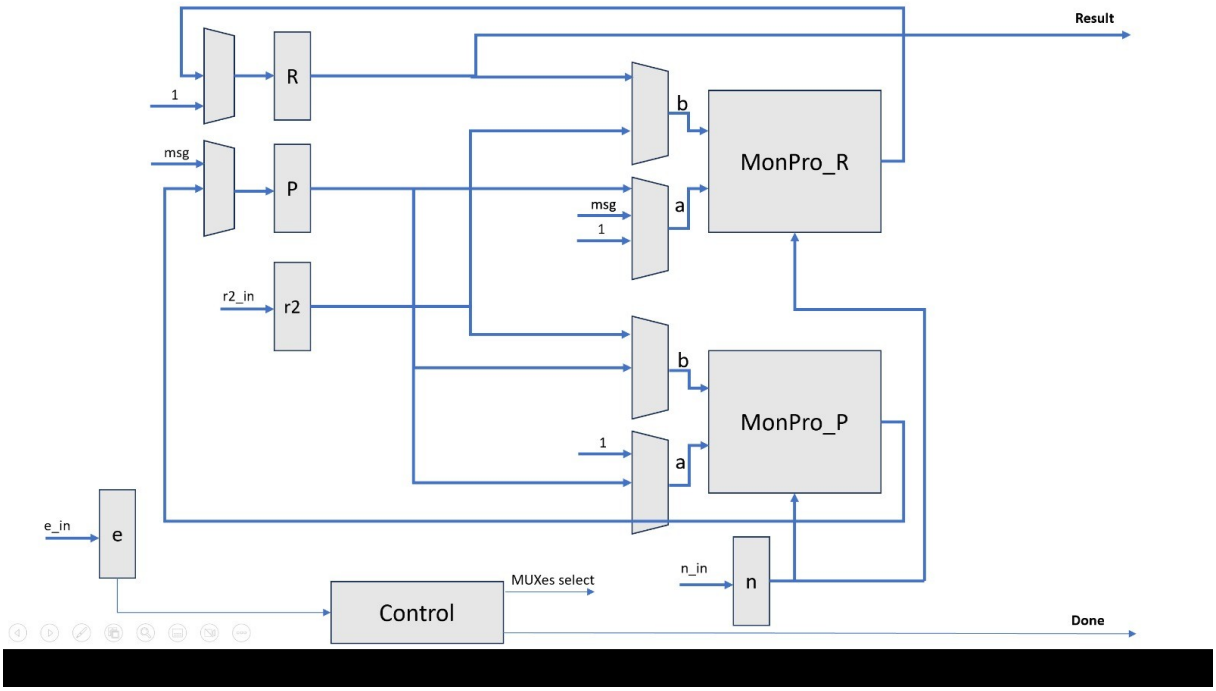


Figure 6. Message transport in and out of rsa_core.

RSA CORE MICROARCHITECTURE (20 POINTS)

The microarchitecture for the exponentiation is the following:

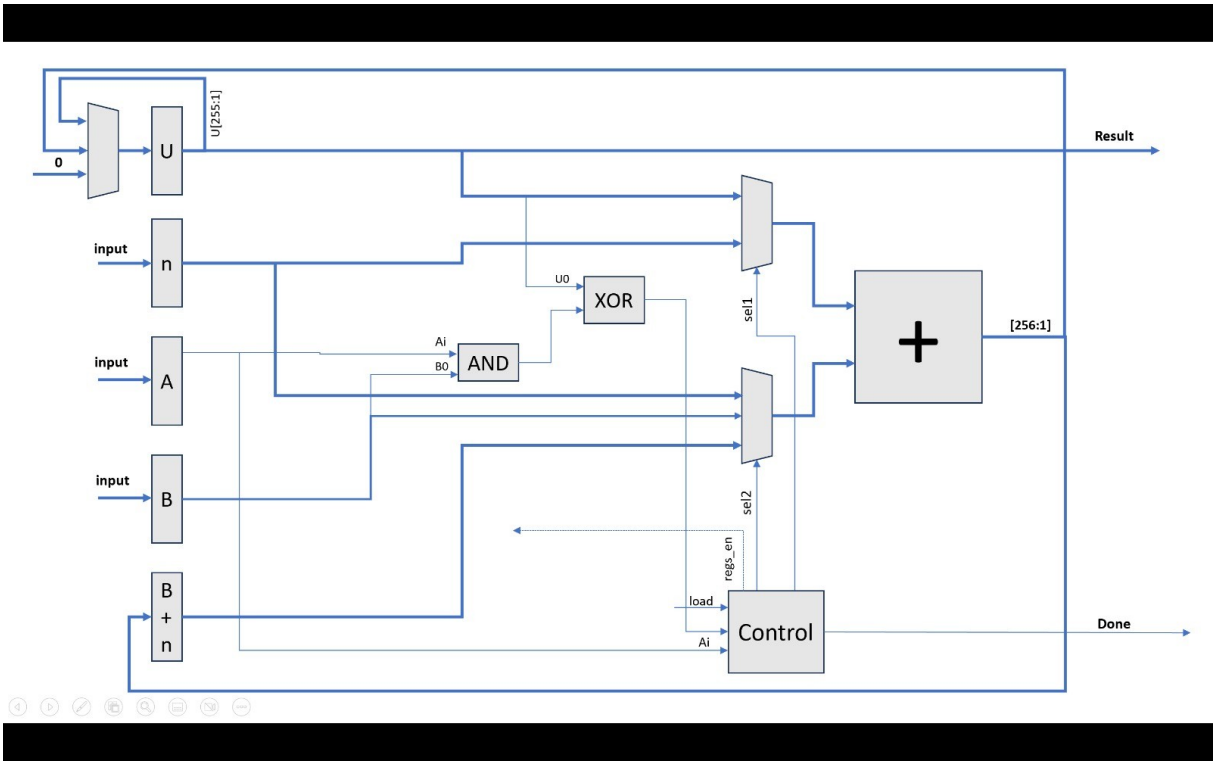


It is a direct translation of the algorithm presented above. The FSM used has 4 states: Idle, where wait for a trigger to read the inputs and start the calculation. Then one state where transform our two operands to the Montgomery domain, followed by a very long state where the exponentiation is calculated. Finally, a state where the result is brought back from the Montgomery domain. The system then goes to idle, keeping the results until the next trigger, marking the operation as done. The exponentiation state is implemented with many sub-states controlled by a counter that counts to **k**.

We use as few registers as possible to hold the values we operate with, all with **k** width (256): A register for holding the message and its squarings, one for holding the result, one for the exponent (the encryption/decryption key), one for the modulus and one for the shifting factor for bringing the operands into Montgomery domain (2^2k mod n). This is received as an input of the module, unlike in the functional model where it was calculated on the spot.

Two Montgomery product components were used. Since we need to do at most two products at each exponentiation iteration step (one for multiplying the result and one for squaring) and they are not dependent on each other withing a single step, we can do both in parallel and half the number of steps needed compared to using a single product component.

The microarchitecture for implementing the Montgomery product is the following:



It mainly consists of a 256 bit adder operating on some registers, plus some control logic to determine which registers to send the adder. It has 4 states, Idle, where we wait for the load pulse, Preprocess where we initialize some registers, then the main state where the addition loop is executed, with a counter to control how many cycles need to be done in this state and finally a state for adjusting the final result and outputting it.

By a careful arrangement and precalculation of control signals, we are able to limit the critical path to just register, a couple of muxes and the big adder. We were also able to do each iteration of the loop in a single clock cycle, needing only a single addition each time. The shifting operation is done by having the output of the adder be 1 bit larger and then discarding the LSB of the result.

PERFORMANCE ESTIMATION (8 POINTS)

Each execution of the Montgomery product takes 258 cycles, and we have to execute it 258 times to calculate the exponentiation of 256 bit numbers (because we use 2 MonPros, otherwise we'd need twice this amount). In total, each exponentiation takes around 66600 clock cycles.

In our initial implementations, we were able to get the module to run at around 120MHz.

Considering that we're not using a pipelined architecture and only have a single exponentiation unit, our rsa core module achieves an initial throughput of around 1800 encryptions/decryptions, not accounting for the transfer and communication overhead.

VERIFICATION PLAN AND VERIFICATION SUMMARY (10 POINTS)

<Describe the verification goals and the verification environments you put in place to meet these goals. Summarize the verification results. >

SYNTHESIS AND IMPLEMENTATION RESULTS (20 POINTS)

<Present area/utilization, max frequency, power consumption, for your design after synthesis>

<Present area/utilization, max frequency, power consumption, for your design after implementation>

<Describe to what extent the design is fully working on the FPGA. If not fully working, discuss why not>

<If the design works on the FPGA you will receive at least 15 point>

PERFORMANCE BENCHMARKING ON FPGA (15 POINTS)

<Present the performance benchmark results from FPGA runs. Include the performance graph from the juniper notebook and populate the tables>

<The faster the circuit is, the more points you will get. For instance, if you end up in the main part of the Hall of Fame, you get full score>

Table 4. Number of clock cycles spent while running the different testcases.

Testcase	T0	T1	T2	T3	T4	T5
Type	ENCR	ENCR	ENCR	DECR	DECR	DECR
Blocks	504	7056	144	504	7056	144
<HW config1>	<clock cycles>	<clock cycles>	<clock cycles>	<clock cycles>	<clock cycles>	<clock cycles>
<HW config 2>	<clock cycles>	<clock cycles>	<clock cycles>	<clock cycles>	<clock cycles>	<clock cycles>

Table 5. Runtime (in ms) for the different testcases.

Configuration	Frequency	T0	T1	T2	T3	T4	T5
SW	-	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>
<HW config 1>	<freq>	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>
<HW config 1>	<freq>	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>	<runtime in ms>

SOURCE CODE QUALITY (9 POINTS)

<Attach the model code, RTL code and testbench code as a part of the delivery bundle>
<Describe how the files in the zip file are organized (e.g. folder structure)>
<Define the RTL coding rules you have tried to follow while writing the RTL code>

DISCUSSION ON SUSTAINABILITY (9 POINTS)

<Discuss how cryptography in general and your RSA implementation in particular have impact on sustainability as defined in the UN goals>

EVALUATION CRITERIA

The evaluation of your term project will be based on this datasheet in addition to the attachments.

Model algorithm	9 points
Microarchitecture	20 points
Performance estimation	8 points
Verification plan and verification summary	10 points
Synthesis and implementation results	20 points
Performance benchmarking on FPGA	15 points
Source code quality	9 points
Discussion on sustainability	9 points
TOTAL	100 POINTS

REFERENCES

[1] PYNQ-Z1 board by Digilent,
https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/

[2] List of other compatible PYNQ boards,
http://www.pynq.io/board.html

[3] Xilinx ZYNQ-7000 SoC
https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[4] AMBA Specification
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html

[5] Vivado Design Suite, AXI Reference guide
https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[6] Dally, W. J., Curtis Harting, R. and Aamodt, T. M., *Digital design using VHDL: a systems approach*. (Cambridge: Cambridge University Press, 2016)