RSA HARDWARE ACCELERATOR

Group 16

Group	Group 16
Authors	Sindre Danielsen Riccardo Cali Herman Dyre Hansen
Date	24/11/2023

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".

Sindre Danielsen
Riccardo Cali
Herman Dyre Hansen

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.	OK
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.	OK
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.	OK
REQ_INT_03	MUST	The design must have one AXI-Lite Slave interface to enable access of memory-mapped registers.	OK
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).	OK
REQ_CONF_01	SHOULD	The design should be optimized for 256 bit block/message/key size.	OK

DEVELOPMENT, DOCUMENTATION AND CODE REQUIREMENS

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.	OK
REQ_DOC_05	MUST	This document must contain results from performance measurements.	OK
REQ_CODE_01	MUST	RTL code for the design must be attached the final delivery bundle.	OK
REQ_CODE_02	MUST	Code for the testbench(es) developed by the group must be attached the final delivery bundle.	OK
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.	OK
Testbench (Alpha)	NOV 10	Upload Testbench to Blackboard.	Write testbenches for testing the design.	OK
Working on FPGA (Alpha)	NOV 17	Upload PPA on Blackboard.	Design working on FPGA.	OK
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)

<Create a high level model of the algorithm(s) you used for modular multiplication and modular exponentiation.>

```
import math

# Solving R = (a*b) mod n, using Blakley's algorithm.
def blakley(a, b, n):
    k=256)
    R = 0
    counter = 0
    for bit_i in range(k):
        counter+=1
        R = 2 * R  # Double R
        if (a >> (k - 1 - bit_i)) & 1:
            R = R + b
        if R >= n:
            R = R - n
        if R >= n:
            R = R - n
    return R


# Modular multiplication using the left-to-right (LR) Binary Method.
def binarymethod(M, e, n):
    C = 1
    for bit_i in range(e.bit_length() - 1, -1, -1):
        C = blakley(C, C, n)
        if (e >> bit_i) & 1:
            C = blakley(C, M, n)

    return C
```

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

We chose to implement the Blakley algorithm for modular multiplication as it had a fairly straightforward structure that would transfer well into hardware. The high-level model consists of two functions: Blakley() for modular multiplication, and binarymethod() for modular exponentiation. The Blakley function calculates the value R = (a*b)%n, while the binarymethod function calculates (M^e)%n. The logic for both functions is taken from the learning material “From algorithm to HW”.

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.

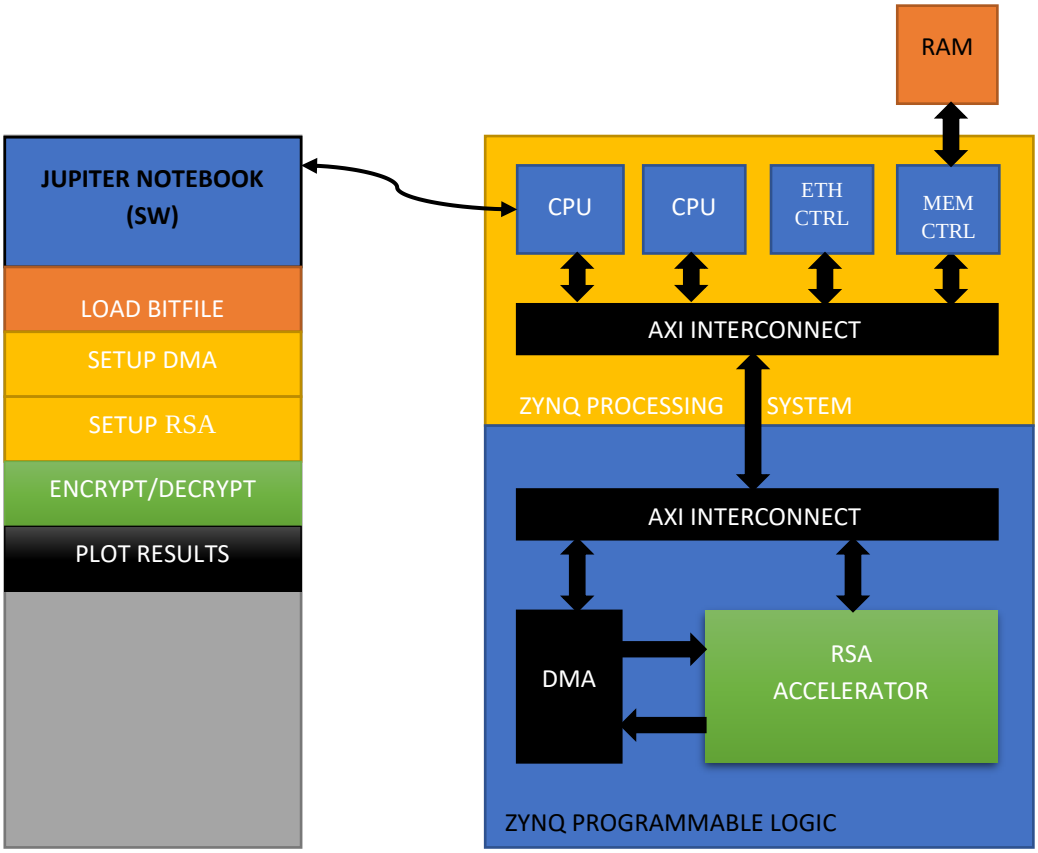


Figure 2. Software and hardware components of the RSA encryption platform.

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).

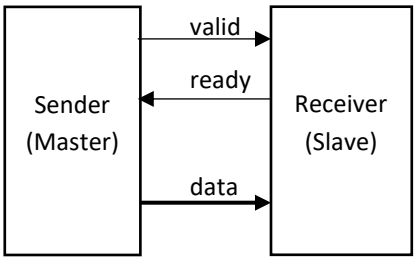


Figure 3. Sender and Receiver exchanging data.

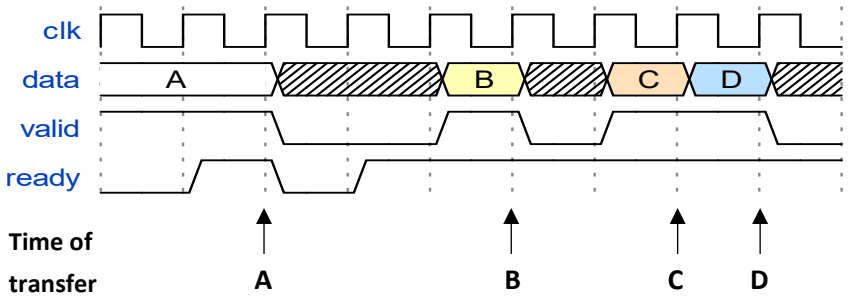


Figure 4. Valid - Ready handshaking. Timing diagram.

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.

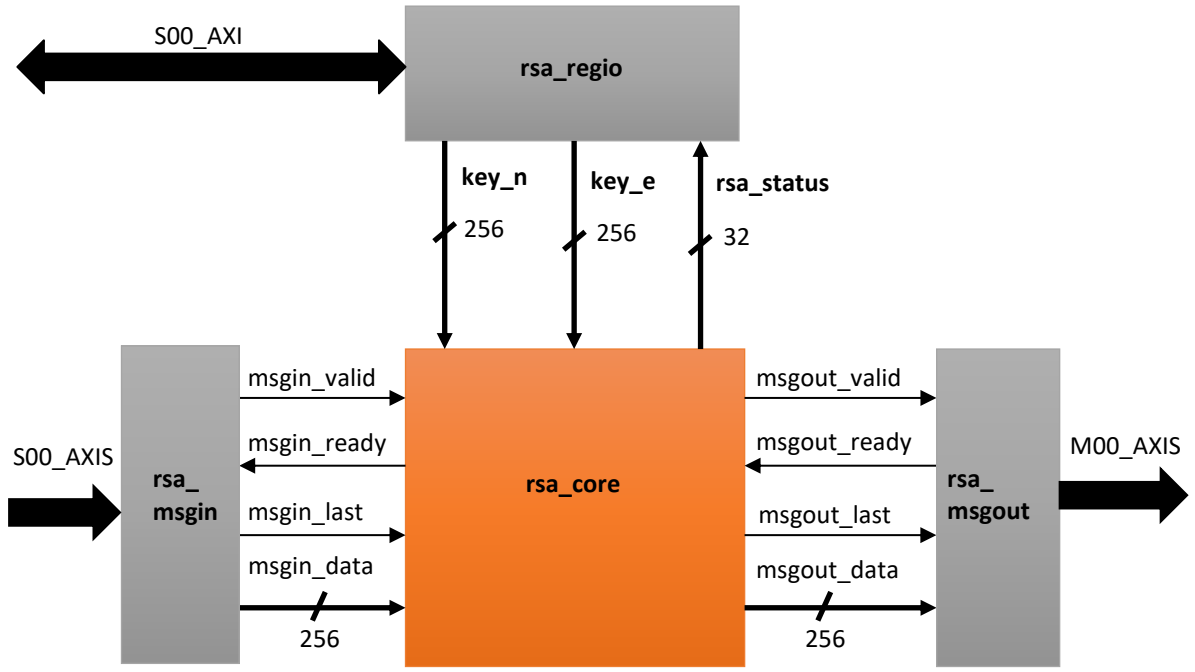


Figure 5. Main blocks within the RSA ACCELERATOR

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>**.

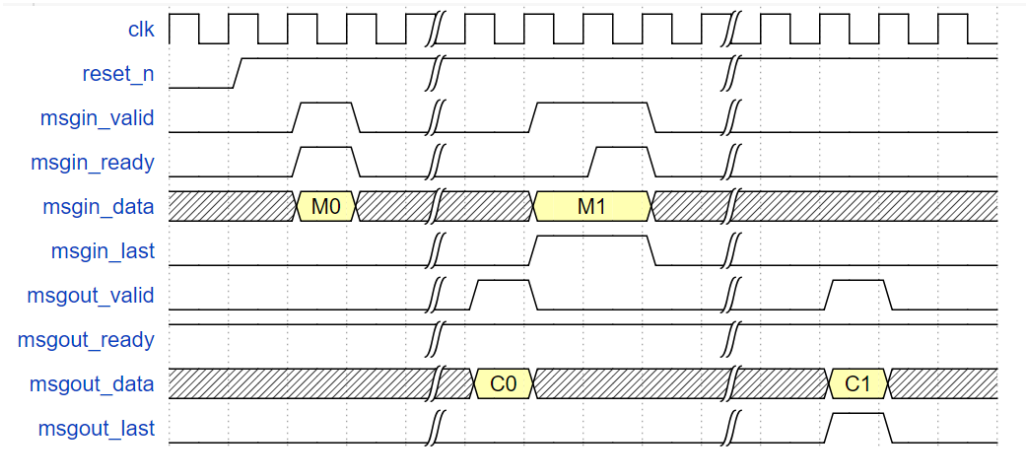


Figure 6. Message transport in and out of rsa_core.

RSA CORE MICROARCHITECTURE (20 POINTS)

We chose the Blakley algorithm for modular multiplication, as it is a simplistic, yet proven from previous years to be a quick algorithm. The Montgomery algorithm seems more complicated, especially when going from algorithm to hardware, hence our choice. For the exponentiation module we chose to use the right-to-left binary method due to the parallelism of two Blakley functions. The algorithms can be seen in the figure below.

RL Binary Method

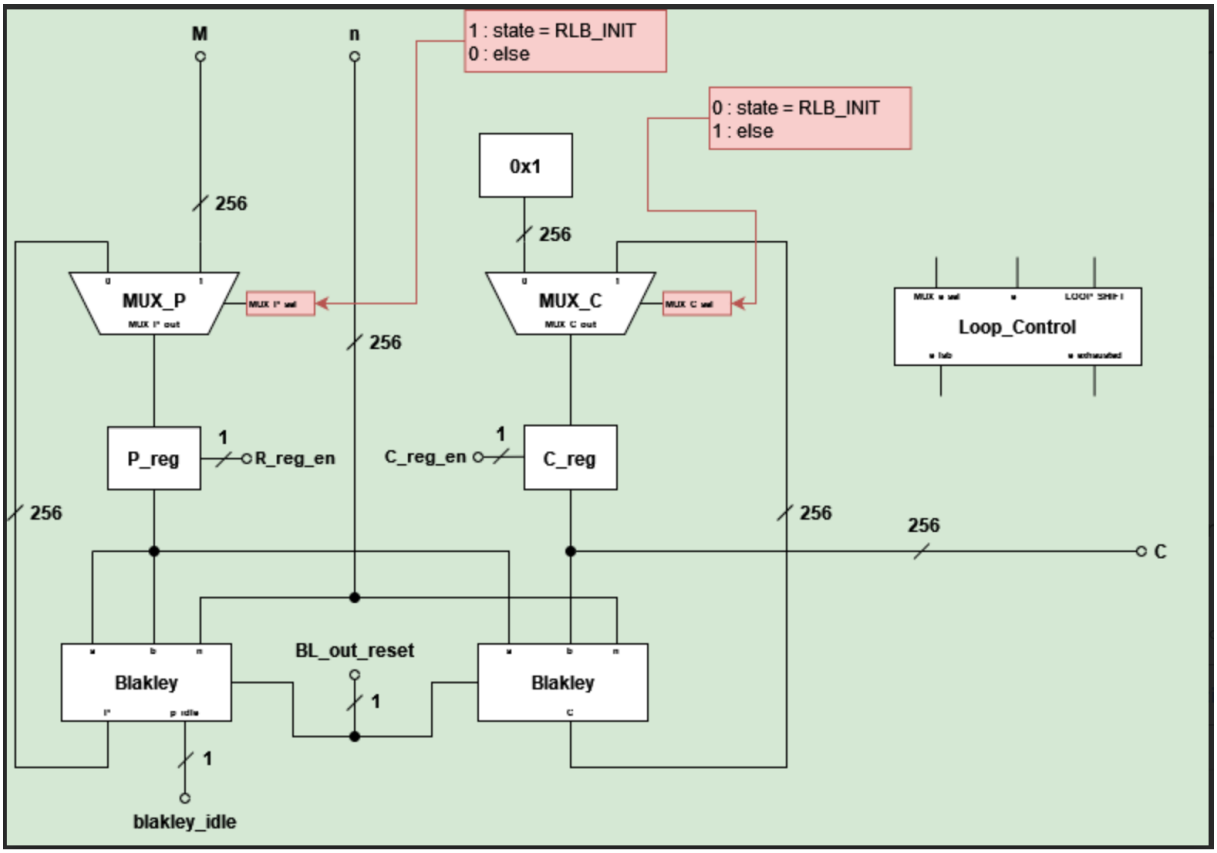
Input: M, e, n
Output: $C := M^e \bmod n$
1. $C := 1$; $P := M$
2. **for** $i = 0$ **to** $h - 2$
2a. **if** $e_i = 1$ **then** $C := C \cdot P \pmod n$
2b. $P := P \cdot P \pmod n$
3. **if** $e_{h-1} = 1$ **then** $C := C \cdot P \pmod n$
4. **return** C

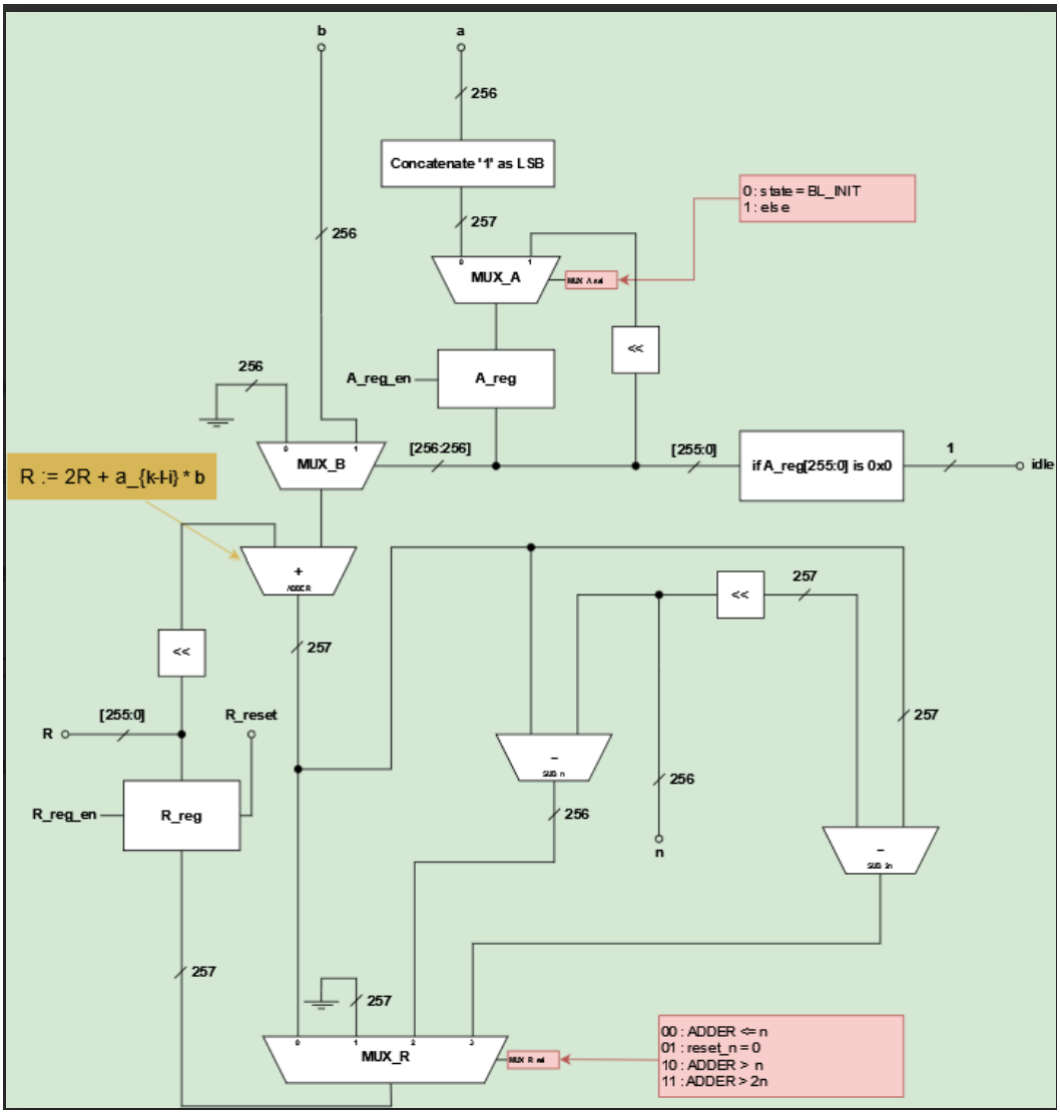
The Blakley Algorithm

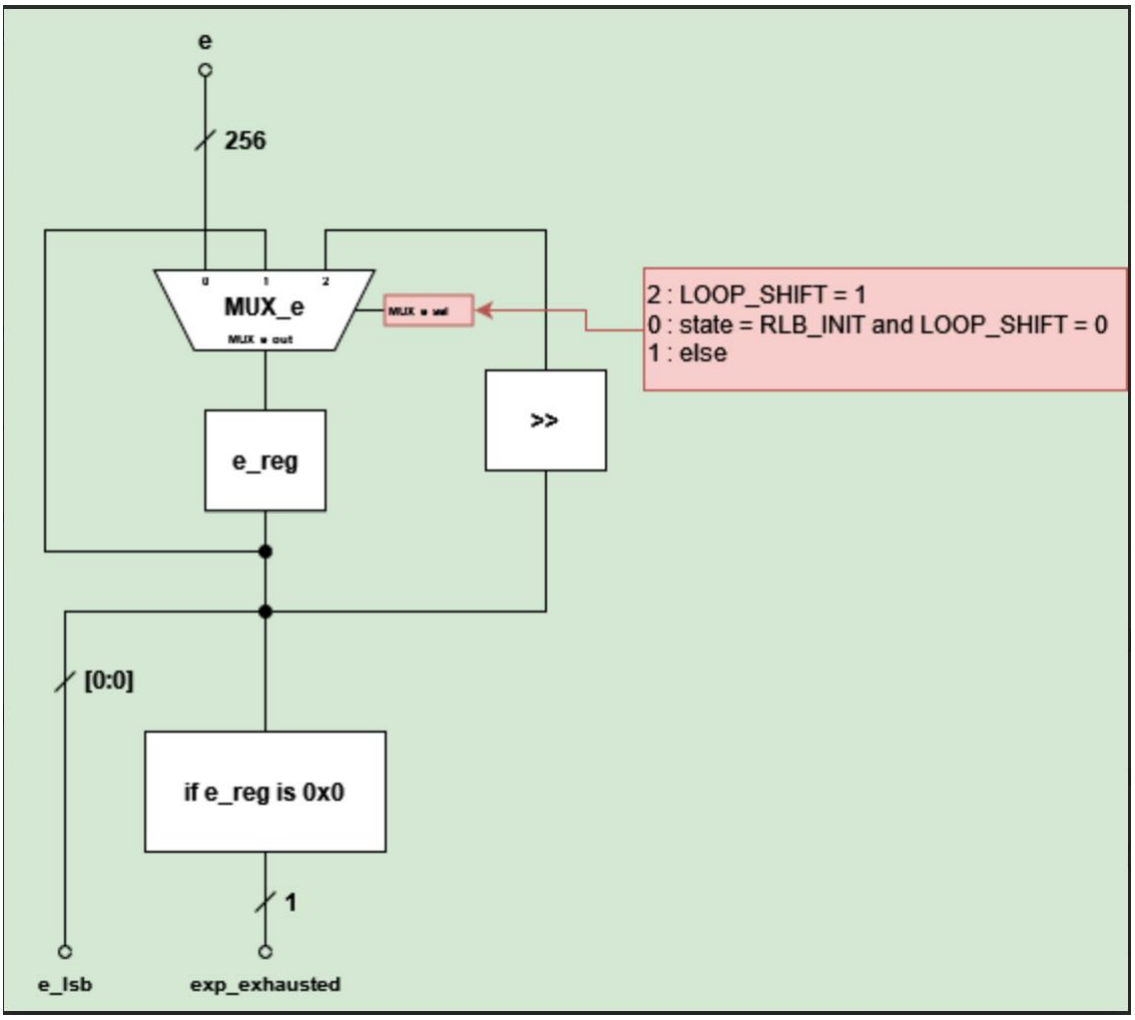
Input: a, b, n
Output: $R = a \cdot b \bmod n$
1. $R := 0$
2. **for** $i = 0$ **to** $k - 1$
3. $R := 2R + a_{k-1-i} \cdot b$
4. $R := R \bmod n$
5. **return** R

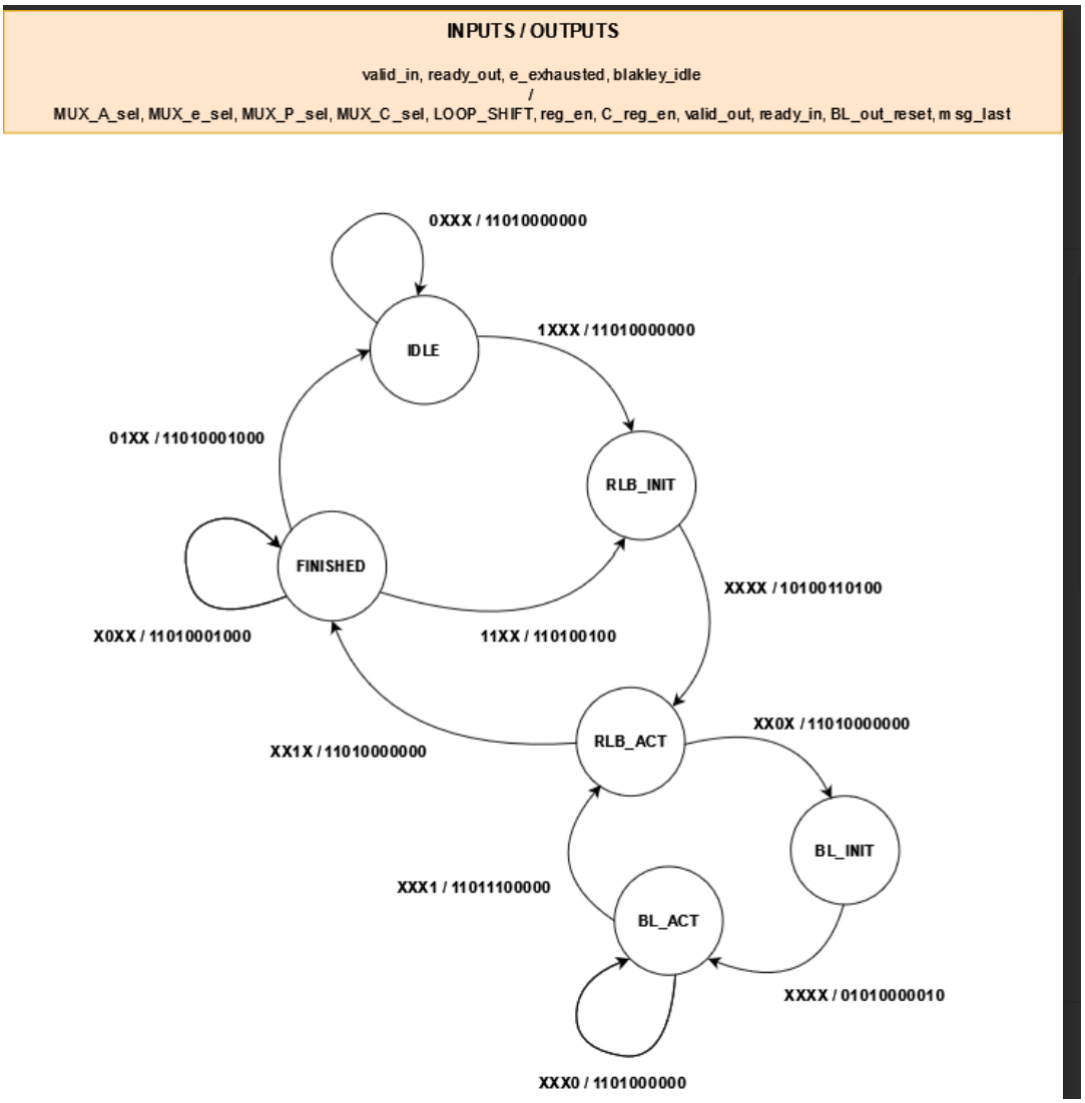
We can also achieve parallelism with “2R” and “a_(k-1-i)”. It is possible to avoid multiplication component, due to “a_(k-1-i)” being either 1-bit of value “0” or “1”, so we can just use a mux with “0” or “b”. This is faster than multiplication. We will also implement a Loop Control module, which will essentially iterate through each bit of the exponent e and determine when to activate the equivalent of the for-loop in the RL Binary Method. Where a variable can have multiple values depending on where in the algorithm it is, or if the value is based on an if-condition, we use multiplexers.

The microarchitecture of the different components we made so far can be seen in the figures below.




b
a
256
Concatenate '1' as LSB
257
256
0 : state = BL_INIT
1 : else
MUX_A
<<
256
A_reg_en
A_reg
MUX_B
[256:256]
[255:0]
if A_reg[255:0] is 0x0
1
idle
R := 2R + a_{k-l-i} * b
+
257
<<
257
<<
[255:0]
R
R_reset
257
R_reg_en
R_reg
256
n
257
257
257
MUX_R
00 : ADDER <= n
01 : reset_n = 0
10 : ADDER > n
11 : ADDER > 2n


e
256
0
1
2
MUX_e
MUX e out
MUX e sel
2 : LOOP_SHIFT = 1
0 : state = RLB_INIT and LOOP_SHIFT = 0
1 : else
e_reg
>>
[0:0]
if e_reg is 0x0
1
e_lsb
exp_exhausted



PERFORMANCE ESTIMATION (8 POINTS)

This estimation was made before we implemented the design.

We have not looked into most critical path or the time adders, multiplexers, shift operations and subtractions take, but assuming it is less than 1 clock, we use registers, which in turn is clocked at 1 cycle. This means:

1. Blakley: “2R”- and “a_(k-1-i)*b”-operation is parallel to each other and same clock cycles = 1 clock cycle.
2. Blakley: Done in k-bit times (in this case 256) → 1 * 256 clock cycles.

3. RL Binary: Done in k-bit times (in this case 256) → k^2 clock cycles = (256 * 256) clock cycles = 65 536 clock cycles.
4. We might in the future add a register after the adder, if all the (shift + adder + 2 subtractions + shift)-operations makes too long of a path. This would imply 2k^2 clock cycles = 131 072 clock cycles.

VERIFICATION PLAN AND VERIFICATION SUMMARY (10 POINTS)

Our verification goals were centered around the Design Requirements as given in this document. The priority was to implement a function for modular multiplication and modular exponentiation. To do this, we created modules that performed different operations that together would perform the required task. For each module and submodule we created testbenches. To verify the functionality of these modules, used relatively small input values and manually checked the waveform output and compared it with our expected results.

As a final test for the completed module, the testbench provided in the file handout was used as verification. The test bench takes in a lot of different inputs and automatically compares the outputs with known correct values. In the final verification simulation, all tests passed successfully.

In order to test the design, we ran the provided code in the Jupyter Notebook to run the design on hardware. The results from this can be seen in the performance section below.

As for pass rate, it is difficult to provide an exact number. However, given our design passed all given tests at many different input sizes, and the fact that the overall design is fairly simple in comparison to many advanced RSA implementations, it would not be unreasonable to estimate the usage rate to be close to 100%.

SYNTHESIS AND IMPLEMENTATION RESULTS (20 POINTS)

Synthesis results:

Utilization after synthesis:

Resource	Utilization	Available	Utilization %
LUT	10500	53200	19.74
FF	9453	106400	8.88
BRAM	2	140	1.43

Max frequency: 60 MHz

Power consumption dynamic: 1.369 W

Power consumption static: 0.137 W

Utilization results:

Utilization after implementation:

Resource	Utilization	Available	Utilization %
LUT	10121	53200	19.02
LUTRAM	645	17400	3.71
FF	9220	106400	8.67
BRAM	2	140	1.43

Max frequency: 60 MHz

Power consumption dynamic: 1.370 W

Power consumption static: 0.137 W

The design appears to work perfectly on the FPGA, and passed all tests in the notebook.

PERFORMANCE BENCHMARKING ON FPGA **(15 POINTS)**

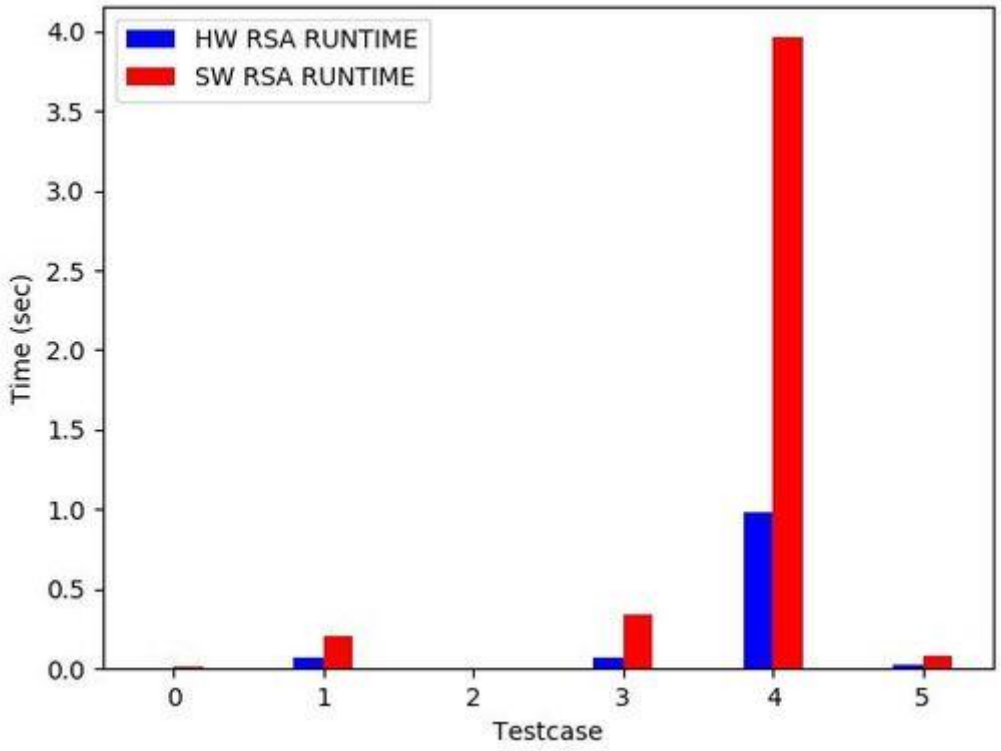


Table 4. Number of clock cycles spent while running the different testcases.

Testcase	T0	T1	T2	T3	T4	T5
Type	ENCR	ENCR	ENCR	DECR	DECR	DECR
Blocks	504	7056	144	504	7056	144
Single Core Config 1	352921	4021124	124239	4213199	58740305	1218667

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

Configuration	Frequency	T0	T1	T2	T3	T4	T5
SW	-	16.49 ms	204.95 ms	4.40 ms	334.37 ms	3960.42 ms	85.20 ms
Single Core Config 1	60 MHz	5.88 ms	67.02 ms	2.07 ms	70.22 ms	979.00 ms	20.31 ms

SOURCE CODE QUALITY (9 POINTS)

The model code, RTL code and testbench code are included in the zip-file. We have divided the code into 3 folders: sources, testbenches and high level model.

In order to achieve the proper functionality as well as good readability, we tried to follow most RTL coding conventions. We commented parts of our code, especially parts that may be harder to intuitively understand. We also made sure to give unique and specific names to all modules and values.

We defined values for all registers, and set them to 0 on reset. We also only updates registers on a rising clock edge.

As for our state machine, we made sure that it initializes to a known state at every reset. We also programmed it with Case statements, which is easy to read. All states are defined, and values are set in each case.

DISCUSSION ON SUSTAINABILITY (9 POINTS)

While cryptography in general could be applied to many, if not all, of the UN Sustainable Development Goals, we decided to focus on the following: 9. Industry, Innovation, and Infrastructure, 13. Climate Action and 16. Peace, Justice, and Strong Institutions.

For Industry, Innovation and Infrastructure, cryptography, and more specifically advanced techniques like RSA, is vital for preserving digital infrastructure. They allow for safe communication and secure services online, which is vital for innovation and industries where privacy is vital such as digital healthcare services.

As for Climate Action, while cryptography might not seem directly relevant, a focus in our project was to minimize the power consumption of our system. On a larger scale, developing efficient RSA implementations will contribute to the climate cause by reducing the energy consumption in data centers around the world.

Finally, we believe that cryptography and better RSA implementations could be important for Peace, Justice and Strong Institutions. In a time where our society is constantly digitalizing, the demand for trust in digital systems is greater than ever. This means that more advanced cryptography methods will be more in demand and more needed than ever going forward. With more digitalization comes more possibilities for cyber-attacks, which again means that more advanced cryptography will be able to protect against these and maintain peace and justice.

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)