RSA HARDWARE ACCELERATOR

Organisation	Norwegian University of Science and Technology (NTNU)
Authors	Giorgi Solomnishvili Besjan Tomja Mohamed Mahmoud Sayed Shelkamy Ali
Date	25/11/2022

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.	Ok
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.	
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 5	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 7	Upload diagram on Blackboard	Draw microarchitecture diagram for hardware design in this datasheet.	Ok
Performance estimate	OCT 7	Estimate performance. Upload to Blackboard.	Estimate the time needed to encrypt/decrypt a block, in this datasheet.	Ok
Microarchitecture review/presentation	OCT 7	Give presentation in class.	Staff and fellow students (peers) reviews the solutions proposed by each team and gives feedback.	Ok
RTL Code (Alpha)	NOV 4	Upload RTL code to Blackboard.	Write synthesizable register transfer level code.	Ok
Testbench (Alpha)	NOV 11	Upload Testbench to Blackboard.	Write testbenches for testing the design.	Ok
Working on FPGA (Alpha)	NOV 18	Upload PPA on Blackboard.	Design working on FPGA.	Ok
Hand in this document with and all pieces of source code	NOV 25	Upload this document together with all pieces of source code on Blackboard.	Hand in this document	ok

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.>

```
/*****************************************************************************

Publishers: Besjan Tomja, Mohhamed Mahmoud Sayed Shelkamy Ali, Giorgi Solomnishvili
Date: 20.09.2022

*****************************************************************************/

#include <stdio.h>
#include <stdint.h>

uint32_t squareMod(uint32_t a, uint32_t b, uint32_t n);
uint32_t encryption_dectyption(uint32_t M_C, uint32_t e_d, uint32_t n);


uint32_t encryption_dectyption(uint32_t M_C, uint32_t e_d, uint32_t n)
{
  uint32_t c = 1;
  uint32_t p = M_C;
  uint32_t i = 0;
  uint32_t k = 32;
  uint32_t j = 1;
  for(i = 0; i < k - 1; ++i){
    if(e_d & j)
    {
      c = squareMod(c,p,n); //c * p;

    }

    p = squareMod(p, p, n);
    j = j*2;
  }

  return c;
}

uint32_t squareMod(uint32_t a, uint32_t b, uint32_t n)
{
  uint32_t r = 0;
  uint32_t i = 0;
  uint32_t k = 32;
  uint32_t j = 1;
  j = j << 31;


  for(i = 0; i < k; ++i)
  {
    r = (a & j) ? 2*r + b : 2*r;

    if(r >= n) //
```

```
            r -= n;
            if(r >= n)
                r -= n;

        j = j/2;
    }

    return r;

}

int main()
{
    uint32_t m = 2; //19;
    uint32_t e = 7; //5;
    uint32_t n = 33; //119;

    uint32_t d = 3; //77;

    uint32_t c = encryption_dectyption(m, e, n);

    printf("Encrypted %u is %u\n",m, c);
    printf("Decrypted %u is %u\n",c, encryption_dectyption(c, d, n));

    return 0;
}
```

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

```
RL Binary Method
Input: $M, e, n$
Output: $C := M^e \bmod n$
1.    $C := 1$ ; $P := M$
2.    for $i = 0$ to $h - 1$
2a.       if $e_i = 1$ then $C := C \cdot P \pmod{n}$
2b.       $P := P \cdot P \pmod{n}$
3.    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 decided to use Blakely algorithm to compute modular multiplication and RL Binary method to calculate modular exponentiation. Blakely's algorithm uses bit by bit multiplication and performs a reduction at each step of the multiplication to make sure that result is less than n.

Our high-level module implements two functions: **Encryption_decryption** and **squareMod**.

Encryption_dectyption can perform RSA encryption and decryption algorithm. This function implements repeated squaring and calls **squareMod** to compute modular multiplications.

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.

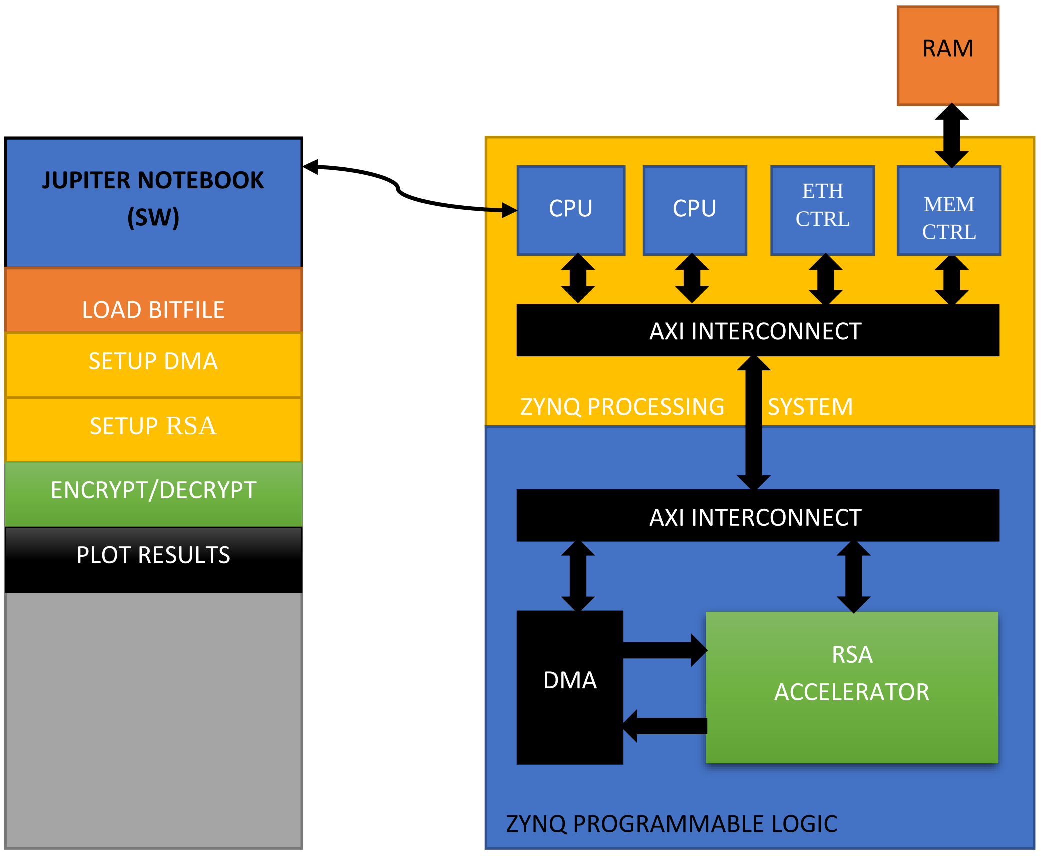


Figure 2. Software and hardware components of the RSA encryption platform.

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).

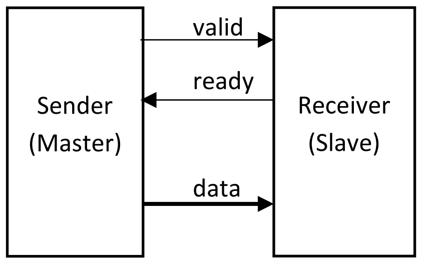


Figure 3. Sender and Receiver exchanging data.

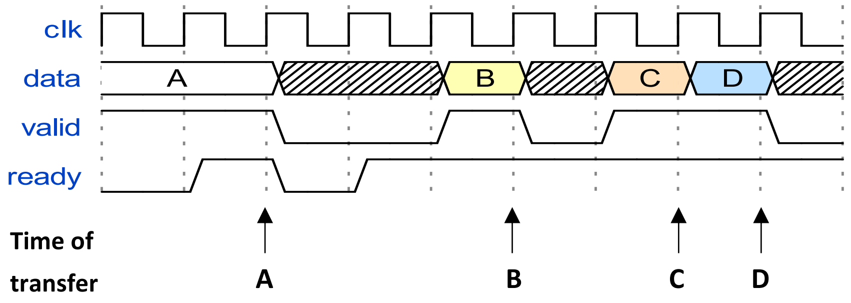


Figure 4. Valid - Ready handshaking. Timing diagram.

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.

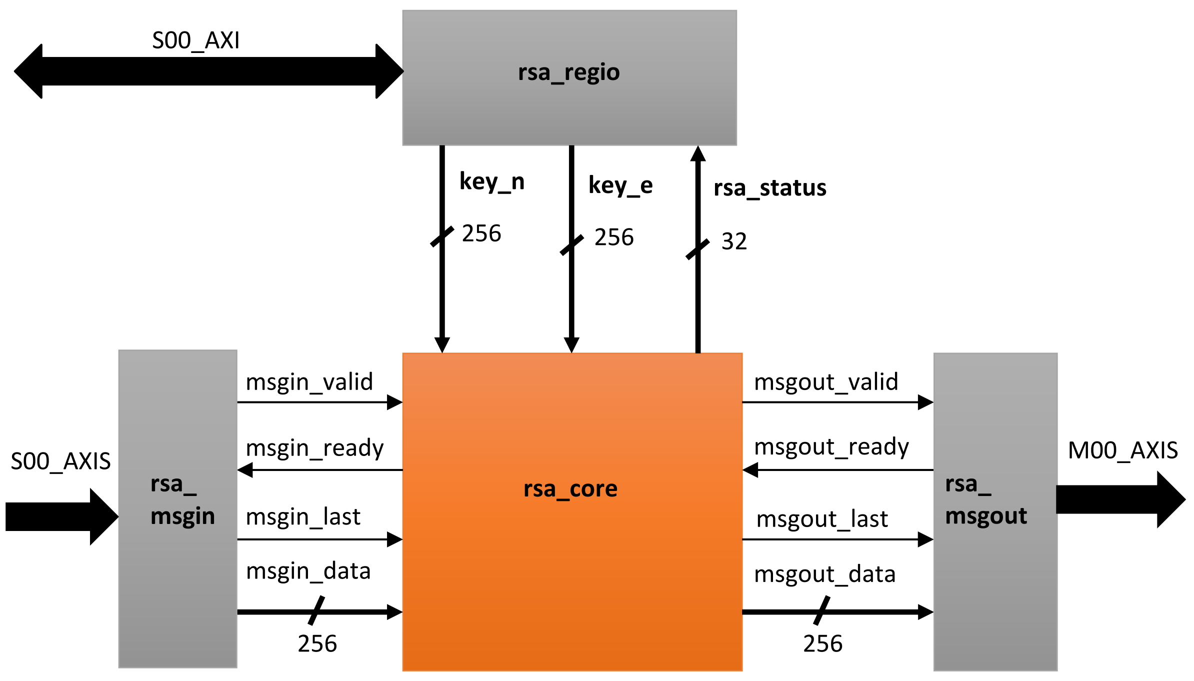


Figure 5. Main blocks within the RSA ACCELERATOR

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>**.

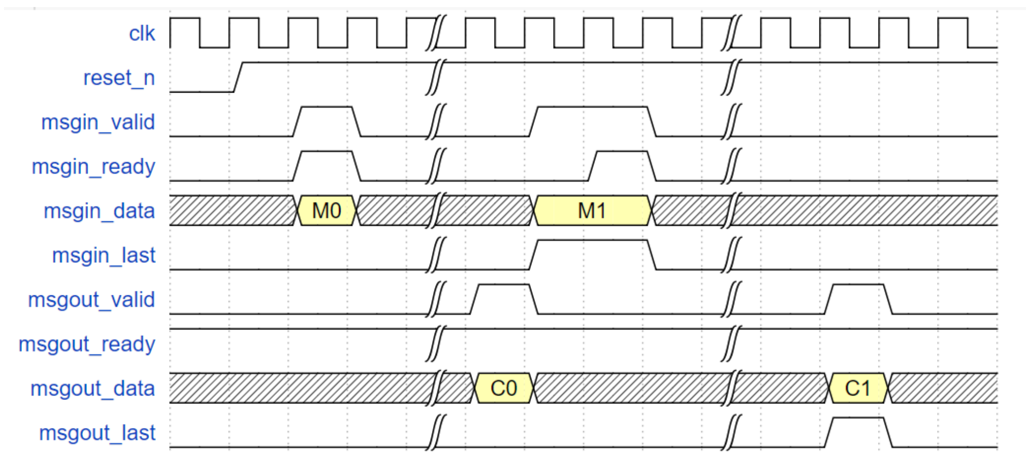


Figure 6. Message transport in and out of rsa_core.

RSA CORE MICROARCHITECTURE (20 POINTS)

<This chapter must contain one or more diagrams that illustrates the microarchitecture of the rsa_core. Also add a description of the design>

The following diagram shows microarchitecture for RSA_Core datapath.

On top half of the picture, the one can see microarchitecture that implements RL Binary Method and instantiates sqMod module to calculate modular multiplication.

The datapath has inputs from modules that surround RSA_Core and from controler that is part of RSA_Core. Inputs, such as key_n, key_e, M, are from surrounding modules. Input – start, is from controller.

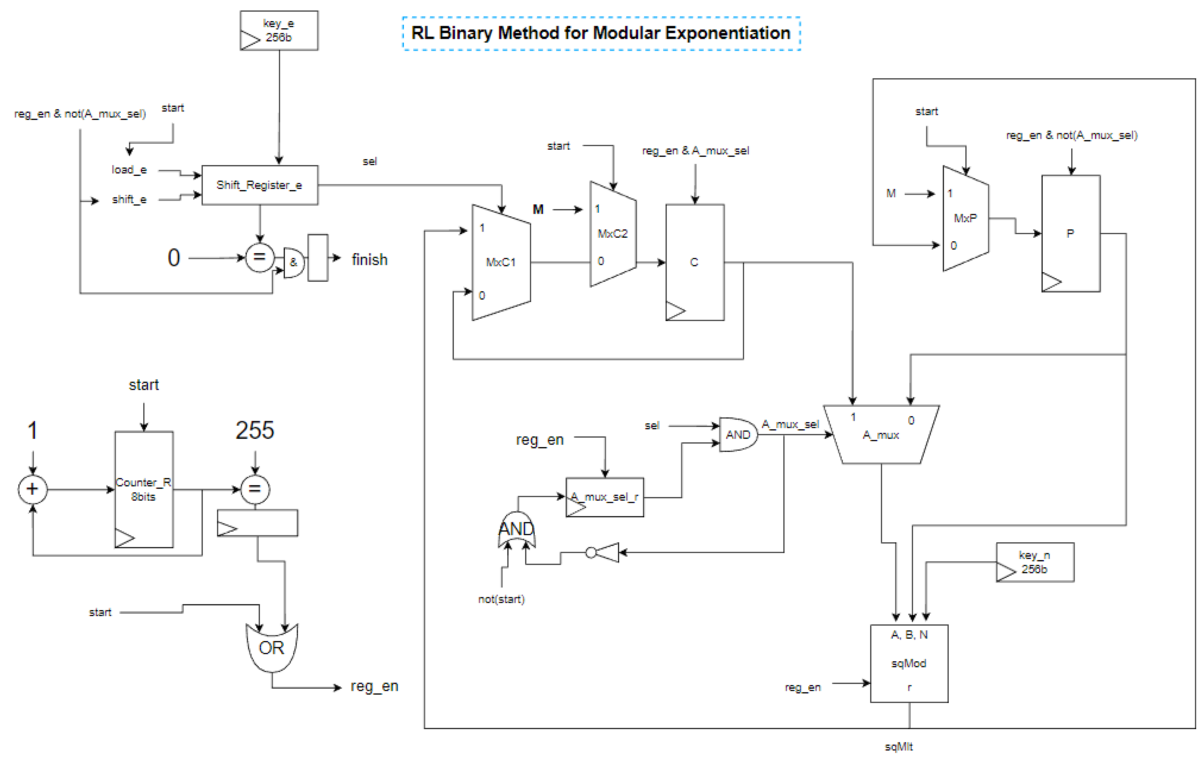
At first, when start is asserted, register C is initialized with M, P is initialized with M and R is initialized with 0. At the same time, Key_e is loaded into shift register. After that, shift register is shifted right. This way the system goes through each bit of e. If the bit of e is 1, C register is updated with value coming from sqMod, otherwise C register keeps its old value. When e becomes 0, signal finish is asserted and calculation terminates with register C containing encrypted or decrypted value.

The sqMod takes 256 clock cycles to finish the calculation. Therefore, we need to have 256 clock cycles between shifts in Shift_Register_e. This is achieved by implementing a counter.

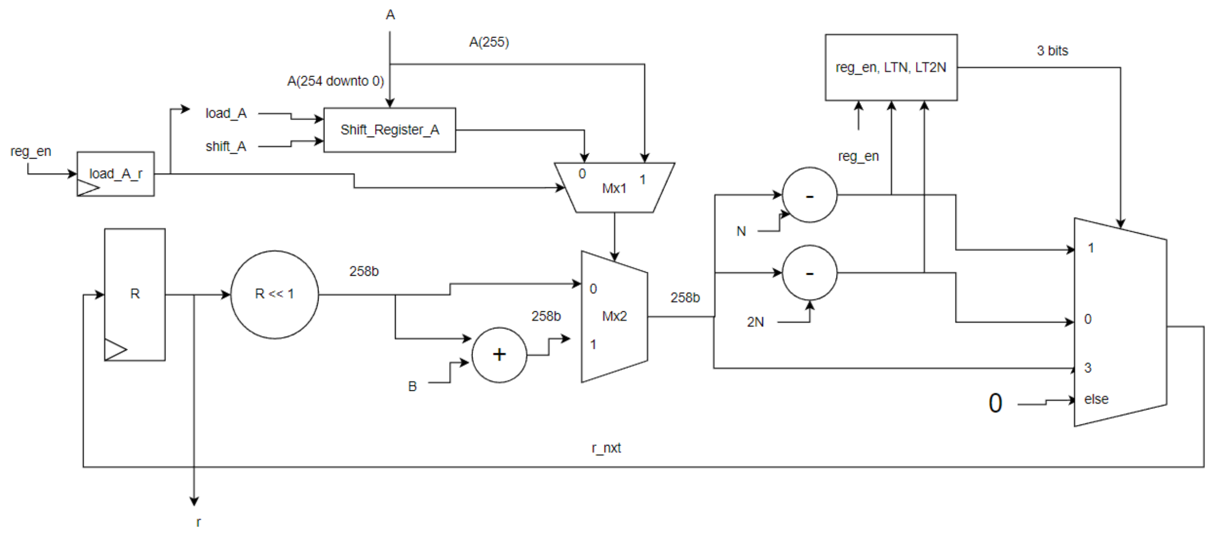
The bottom half of the picture depicts datapath for sqMod. sqMod returns A*B (mod n). A is loaded into a shift register. This way the submodule sqMod goes through each bit of A. If the bit of A is 1, partial sum equals 2R + B, otherwise partial sum equals 2R. Next, the partial sum is compared with N and 2N. If it is less than N, it is directly sent to register R. If it is between N and 2N, N is subtracted from partial sum and is sent to register R. If partial sum is more than 2N, 2N is subtracted from it and sent to register R.

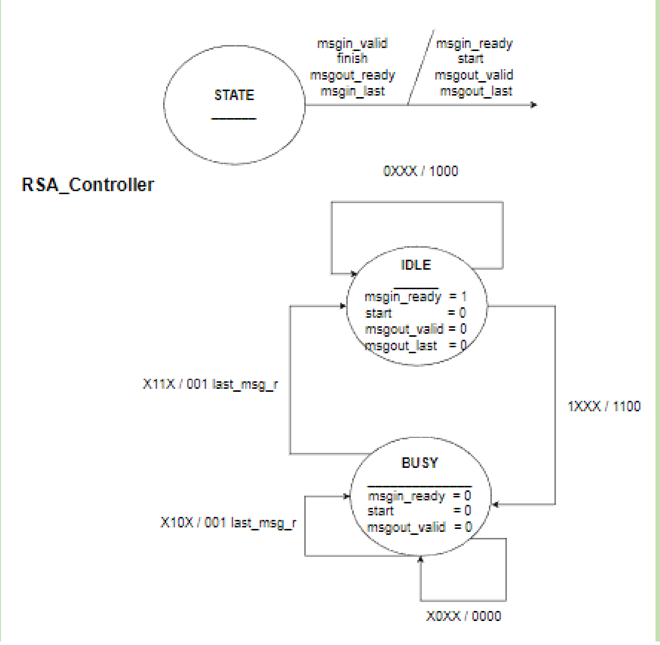
The top module instantiates only one instance of sqMod. Hence, register C and P have to take turns while using the module. We have A_multiplexer that selects between C and P based on A_mux_sel signal.

RL Binary Method for Modular Exponentiation



Blakely algorithm for modular multiplication





PERFORMANCE ESTIMATION (8 POINTS)

SqMod takes 256 clock cycles to calculate modular multiplication. At most we have to calculate 512 modular multiplications. This depends on the position of the leftmost 1 in KEY_E_D. If the position is 256, we have to perform 256 shifts, if it is 17 we perform 17 shifts, etc.

Estimated Number of Clock Cycles = 256 * 512 = 131 072 clock cycles.

VERIFICATION PLAN AND VERIFICATION SUMMARY (10 POINTS)

Our design consists of 5 modules: **counter**, **mod_mult**, **RSA_controller**, **RSA_datapath**, and **exponentiation**. We wrote testbenches for each of them that checked the correct functionality using assertions.

Counter_tb: The counter just counts from 0 to 255. We just checked the waveforms.

Mod_mult_tb: This module feeds input a, b, and n to mod_mult. Mod_mult calculates a*b mod n. Waits 256 clock cycles and checks if the output is correct. Then, we assign a, b and n with the maximum possible values and test it again.

RSA_controller_tb: This module instantiates RSA_controller. The controller consists of a Mealy FSM. We feed different control inputs and check the outputs and correct state transitions.

RSA_datapath_tb: This module implements the whole RL Binary algorithm. The microarchitecture can be seen above. This module returns encrypted/decrypted message. The testbench instantiates module, feeds it with keys, message, and control signals, which are supposed to be coming from controller, and tests if the calculated result is correct.

Exponentiation_tb: This module implements the whole RSA_core, simulates input and output interfaces and checks if the core works correctly.

Finally, we checked the whole accelerator with testbench provided to us.

SYNTHESIS AND IMPLEMENTATION RESULTS (20 POINTS)

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

SYNTHESIS RESULTS:

```
+-------------------------+-------+-------+-----------+-------+
|        Site Type        | Used  | Fixed | Available | Util% |
+-------------------------+-------+-------+-----------+-------+
| Slice LUTs*             | 45005 |     0 |     53200 | 84.60 |
|   LUT as Logic          | 45005 |     0 |     53200 | 84.60 |
|   LUT as Memory         |     0 |     0 |     17400 |  0.00 |
| Slice Registers         | 25386 |     0 |    106400 | 23.86 |
|   Register as Flip Flop | 25386 |     0 |    106400 | 23.86 |
|   Register as Latch     |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                |   197 |     0 |     26600 |  0.74 |
| F8 Muxes                |    66 |     0 |     13300 |  0.50 |
+-------------------------+-------+-------+-----------+-------+
```

Max Frequency: 57 MHz

Power Consumption: 0.71 W

IMPLEMENTATION RESULTS:

```
+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs                 | 50087 |     0 |     53200 | 94.15 |
|   LUT as Logic             | 49453 |     0 |     53200 | 92.96 |
|   LUT as Memory            |   634 |     0 |     17400 |  3.64 |
|     LUT as Distributed RAM |   442 |     0 |           |       |
|     LUT as Shift Register  |   192 |     0 |           |       |
| Slice Registers            | 31141 |     0 |    106400 | 29.27 |
|   Register as Flip Flop    | 31141 |     0 |    106400 | 29.27 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |   197 |     0 |     26600 |  0.74 |
| F8 Muxes                   |    66 |     0 |     13300 |  0.50 |
+----------------------------+-------+-------+-----------+-------+
```

Max Frequency: 55 MHz

Power Consumption: 0.88 W

We tested the design on FPGA and all tests were passed. If we use 55 MHz clock, worst negative slack will be positive. However, we uploaded design on the FPGA, we used 70 MHz and all tests were passed successfully.

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>	123900	296100	122500	412230	4811800	130900

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

Configuration	Frequency	T0	T1	T2	T3	T4	T5
SW	-	16.2	205.4	4.8	296	3933	84.84
<HW config 1>	70 MHz	1.88	4.3	1.8	5.97	68.7	1.75

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>.

The attached zip file contains the following folders:

- Model – contains source code for high level model.
- RSA/ tfe4141_rsa_integration_kit_2020/ tfe4141_rsa_integration_kit_2020
  - Bitfiles – Stores rsa_soc.hwh and rsa_soc.bit files.
  - Exponentiation:
    - Sources – counter.vhd, exponentiaitn.vhd, mod_mult.vhd, RSA_Controller.vhd, RSA_Datapath.vhd
    - Testbench – (module_under_test_name)_tb.vhd
  - RSA_accelerator:
    - Sources – rsa_accelerator, rsa_core.vhd, rsa_msgin.vhd, rsa_msgout.vhd, rsa_rgio.vhd, selector.vhd
    - Testbench – selector_tb.vhd, rsa_accelerator_tb.vhd
  - RSA_soc
  - Reports – synthesis and implementation reports
  - Master_constraints – constraints file

RTL coding rules:

- We tried to follow naming conventions
- At the beginning of each module, you find the description of module interface and functionality.
- The sequential part of the design and combinational parts are separated in different process.

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> In our design we tried to minimize the power consumption by
Our RSA implementation will have impact on the below UN goal:

- Affordable and clean energy & Climate Action
- In our design we considered power consumption in order to minimize it, which will help in reducing CO2 emissions.
- Decent Work and Economic growth & Peace, Justice, and strong institutions
  Nowadays a lot of companies are facing security issues, which create great damage to the company reputation which is reflected in financial damage. The RSA design will help in encrypting/decrypting sensitive information of the companies, creating thus a safe working environment and conflicts prevention.
- Quality Education
  During the process of implementation of the design for the project we also put effort into providing good documentation for the entire design. This will help in future projects and help students who can use this project in their research areas, because it reduces the time to understand the implementation, and it creates space for future advanced implementations.
- Responsible consumption and production
  This accelerator can be used in different filed for encryption/decryption, creating thus a broad area where this accelerator can be use, without creating a specific design for a specific application.

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)