|  |  |  |
| --- | --- | --- |
| **RSA Hardware accelerator** | | 22  LOGO |
| Group | Group22 | |
| Authors | Andreas Just, Mads Mølbach, Christian Heimvik | |
| Date | 20.11.24 | |

# 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 = Yk mod n | OK |
| **REQ\_FUNC\_02** | MUST | The design must be able to encrypt and decrypt message blocks using modular exponentiation:  Encryption: C = Me mod n, M < n, C < n, e < n  Decryption: M = Cd mod 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. |  |
| **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. |  |
| **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. |  |

# 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 2 | Sign up on Blackboard | Form term project groups | OK |
| Study algorithms and pick one | SEP 18 | Nothing to upload | Study algorithms and pick one | OK |
| High level model | SEP 25 | Upload code on Blackboard | Implement the algorithm in python or another high level language. | OK |
| Microarchitecture | OCT 2 | Upload diagram on Blackboard | Draw microarchitecture diagram for hardware design in this datasheet. | OK |
| Performance estimate | OCT 2 | Estimate performance. Upload to Blackboard. | Estimate the time needed to encrypt/decrypt a block, in this datasheet. | OK |
| Microarchitecture review/presentation | OCT 2 | Give presentation in class. | Staff and fellow students (peers) reviews the solutions proposed by each team and gives feedback. | OK |
| RTL Code (Alpha) | OCT 30 | Upload RTL code to Blackboard. | Write synthesizable register transfer level code. | OK |
| Testbench (Alpha) | NOV 6 | Upload Testbench to Blackboard. | Write testbenches for testing the design. | OK |
| Working on FPGA (Alpha) | NOV 13 | Upload PPA on Blackboard. | Design working on FPGA. |  |
| Hand in this document and all pieces of source code | NOV 22 | 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.

1. **Write design specification:**

* Describe the design you intend to make
* Draw microarchitecture diagrams
* Clearly define interfaces between modules in the design

1. **Design and verification:**

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

1. **Implement the design:**

* Synthesize the design
* Run Place & Route

1. **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.>**

|  |
| --- |
| \*PTHON FILE: HLM.py\*  import threading  import time  import csv  import queue  PIPELINE\_STAGES = 16 #Set dependent of PPA in final implementation  KEY\_LENGTH = 64 #Should be 256 in final implemntation  E = 8954    ## Encryption key  N = 25553    ## Modulus  messages = {}  pipelineLog = []  pipelineLogMtx = threading.Lock()  caseMtx = threading.Lock()  '''  pipelinentermediates holds:      [0] - Intermediate C after stage ID-1      [1] - Intermediate P after stage ID-1      [2] - Keeps the message ID  '''  pipelineIntermediates = [[queue.Queue() for \_ in range(3)] for \_ in range(PIPELINE\_STAGES+2)]  intermediatesLoaded = [threading.Semaphore(0) for \_ in range(PIPELINE\_STAGES+2)]  intermediatesPopped = [threading.Semaphore(1) for \_ in range(PIPELINE\_STAGES+2)]  intermediateMtx = [threading.Lock() for \_ in range(PIPELINE\_STAGES+2)]  requestNewCase = threading.Semaphore(0)  grantNewCase = threading.Semaphore(0)  pipelineFinished = threading.Semaphore(0)  ## Note that all semaphore signals correspond to interstage/intercontroller commnication in HW  def getCases(filename):      with open(filename, mode='r') as file:          reader = csv.DictReader(file)          casesQueue = queue.Queue()  # Initialize the queue            for case in reader:              M = int(case['M'])              messageID = int(case['ID'])              casesQueue.put([M, messageID])              messages[messageID] = [M]   ## For final reults      return casesQueue  def splitE(number, keyLength, pipelineStages):      binary\_str = bin(number)[2:]      total\_length = keyLength  # total length of the binary string      padded\_binary\_str = binary\_str.zfill(total\_length)      chunk\_size = total\_length // pipelineStages      chunks = [padded\_binary\_str[i:i + chunk\_size] for i in range(0, total\_length, chunk\_size)]      decimal\_numbers = [int(chunk, 2) for chunk in chunks]      return decimal\_numbers[::-1]  def reportResults():      print("\n--- Report results in hex ---\n")      mismatch\_found = False        # Print the header      print(f"{'Message ID':<15} {'Pipeline Result':<20} {'Expected Result':<20} {'Mismatch':<10}")      print("-" \* 65)  # Divider line      for messageID, data in messages.items():          M = data[0]  # Original message          pipelineResult = data[1]  # Result from the pipeline            expectedResult = pow(M, E, N)          mismatch = "Yes" if pipelineResult != expectedResult else "No"          print(f"{messageID:<15} {hex(pipelineResult):<20} {hex(expectedResult):<20} {mismatch:<10}")            if pipelineResult != expectedResult:              print(f"Mismatch found for message ID {messageID}!")              mismatch\_found = True        if not mismatch\_found:          print("\nAll results are correct!")      else:          print("\nThere were mismatches in the results.")  def reportProgress():      # Prepare a dictionary to store data for each messageID      progression = {}        for stageID, messageID, currentC, currentP in pipelineLog:          if messageID not in progression:              progression[messageID] = [''] \* (PIPELINE\_STAGES + 1)          progression[messageID][stageID] = f"C: {currentC}, P: {currentP}"        # Print the header      print("\n--- Pipeline Progression ---")      header = "Message ID | " + " ".join(f"Stage {i if i else ' ':<14}" for i in range(PIPELINE\_STAGES + 1))      print(header)      print("-" \* len(header))      # Print each messageID's progression      for messageID in sorted(progression.keys()):          row = f"{messageID:<11} | " + " | ".join(f"{value if value else ' ':<18}" for value in progression[messageID])          print(row)        print("-" \* len(header))  def getQueueElement(q):      if not q.empty():          return q.queue[0]  # Access the first element      else:          return ""  # Return 'Empty' if the queue is empty  def blakelyMulMod(a, b, n):      R = 0      a\_bin = bin(a)[2:][::-1]      for i in range(len(a\_bin)):          bit = int(a\_bin[(len(a\_bin)-1)-i])          shift = R<<1                    ## Parelell B          mul = bit \* b                   ## Paralell B          R = mul + shift          if R>n:              R = R - n          if R>n:              R = R - n      return R  def blakeleyPipelineStart(stageID):      while(True):          ## Request the pipeline controller a new case          requestNewCase.release()          grantNewCase.acquire()          ## Get the next case          caseMtx.acquire()          nextCase = cases.get()          caseMtx.release()            ## Wait for asynch signal from next stage that it has popped off the previous values in time          intermediatesPopped[stageID].acquire()          ## Push values to pipelineIntermediates          intermediateMtx[stageID].acquire()          pipelineIntermediates[stageID][0].put(1)          pipelineIntermediates[stageID][1].put(nextCase[0])          pipelineIntermediates[stageID][2].put(nextCase[1])          intermediateMtx[stageID].release()          ## Signal to next stage that data is ready          intermediatesLoaded[stageID].release()  def blakeleyPipelineStage(eSlice,n,stageID):      while(True):          ## Wait for asynch signal from previous stage that data is ready          intermediatesLoaded[stageID-1].acquire()            ## Get values from pipelineIntermediates          intermediateMtx[stageID-1].acquire()          currentC = pipelineIntermediates[stageID-1][0].get(0)          currentP = pipelineIntermediates[stageID-1][1].get(0)          currentID = pipelineIntermediates[stageID-1][2].get()          intermediateMtx[stageID-1].release()          ## Signal to previous stage it has popped, such that the previous stage can replace its values          intermediatesPopped[stageID-1].release()          pipelineLogMtx.acquire()          pipelineLog.append([stageID,currentID, currentC, currentP])          pipelineLogMtx.release()          ## Accumulate new values          if KEY\_LENGTH % PIPELINE\_STAGES != 0:              print(f"Error: KEY\_LENGTH / PIPELINE\_STAGES is not an integer")              raise ValueError          if len(bin(eSlice)[2:]) > (KEY\_LENGTH/PIPELINE\_STAGES):              print(f"Error: eSlice is {eSlice} and not KEY\_LENGTH/PIPELINE\_STAGES")              raise ValueError            mask = 0b1          for i in range(0, int(KEY\_LENGTH/PIPELINE\_STAGES)):              if eSlice & mask:                  currentC = blakelyMulMod(currentC, currentP, n)              currentP = blakelyMulMod(currentP, currentP, n)              mask = mask << 1          ## Wait for asynch signal from next stage that it has popped off the previous values in time          intermediatesPopped[stageID].acquire()          ## Push values to pipelineIntermediates          intermediateMtx[stageID].acquire()          pipelineIntermediates[stageID][0].put(currentC)          pipelineIntermediates[stageID][1].put(currentP)          pipelineIntermediates[stageID][2].put(currentID)          intermediateMtx[stageID].release()          ## Signal to next stage that data is ready          intermediatesLoaded[stageID].release()  def blakeleyPipelineEnd(stageID):      while(True):          ## Wait for asynch signal from previous stage that data is ready          intermediatesLoaded[stageID-1].acquire()          ## Push values to pipelineIntermediates          intermediateMtx[stageID-1].acquire()          endC = pipelineIntermediates[stageID-1][0].get(0)          endP = pipelineIntermediates[stageID-1][1].get(0)          messageID = pipelineIntermediates[stageID-1][2].get(0)          intermediateMtx[stageID-1].release()          messages[messageID].append(endC)    ## For final reults          ## Signal to previous stage it has popped, such that the previous stage can put if it lies ahead in time          intermediatesPopped[stageID-1].release()          if(messageID == (numCases-1)):              print("Last case out of the pipeline, signaled controller.")              pipelineFinished.release()  def blakeleyPipelineController():      while(True):          requestNewCase.acquire()          caseMtx.acquire()          if cases.qsize() == 0:              print("Finished, no more cases left. Awaiting signal from last pipeline stage.\n")              pipelineFinished.acquire()              print(f"Results of operation: C = M {hex(E)} mod {hex(N)} (E and N in hex)\n")              reportProgress()              reportResults()              ## Timing test does not make sence in software, as the gains from the pipelining is only existent in HW          caseMtx.release()          grantNewCase.release()  def paralellBinartExp():      ## Load all test cases that simulate the stream of M      global cases      cases = getCases("testCases.csv")      global numCases      numCases = cases.qsize()      eSlices = splitE(E, KEY\_LENGTH, PIPELINE\_STAGES)      ## Start all threads and asynchromus communication (semaphores)      threads = []      threads.append(threading.Thread(target=blakeleyPipelineController))      threads.append(threading.Thread(target=blakeleyPipelineStart, args=(0,)))      for i in range(1,PIPELINE\_STAGES+1):          threads.append(threading.Thread(target=blakeleyPipelineStage, args=(eSlices[i-1],N,i)))      threads.append(threading.Thread(target=blakeleyPipelineEnd, args=(PIPELINE\_STAGES+1,)))      for thread in threads:          thread.start()      for thread in threads:          thread.join()  def main():      paralellBinartExp()  if \_\_name\_\_ == "\_\_main\_\_":      main()  \*Extraction of some cases from CSV file: testCases.csv (in the same folder as HLM.py)\*  367,106  373,107  379,108  383,109  389,110  397,111  401,112  409,113  419,114  421,115  431,116  433,117  439,118  443,119  449,120  457,121  461,122  463,123  467,124  479,125  487,126  491,127  499,128  503,129  509,130  521,131  523,132  541,133  547,134  557,135  563,136  569,137  571,138  577,139  587,140  593,141  599,142  601,143  607,144  613,145  617,146  619,147  631,148  641,149 |

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

**Model description**

The high-level model is quite complex, but extremely accurate with respect to both the microarchitecture and the final realization of the system, as it also implements the multicore design of the system. The following assumes the reader is known with the contents of section “RSA CORE MICROARCHITECTURE”.

## Blakeley Module

At the lowest level, the model implements the module blakely\_module algorithm for doing modular multiplication. This is very straight forward in Python and looks like this.

def blakeley\_module(a, b, n):

    R = 0

    a\_bin = bin(a)[2:][::-1]

    for i in range(len(a\_bin)):

        bit = int(a\_bin[(len(a\_bin)-1)-i])

        shift = R<<1

        mul = bit \* b

        R = mul + shift

        if R>n:

            R = R - n

        if R>n:

            R = R - n

    return R

The way this works, is that it iterates downwards trough the bits in “a”, starting from the most significant, and whenever the current bit is ‘1’, the previous result, R, is multiplied by 2 (effectively a left shift) and added in with “b”. What this does is effectively a “textbook” multiplication, only starting at the MSB. To reduce complexity and space, intermediate reduction is desired. As R mod n<n and B<n, does this result in a at most 2 subtractions (maximum (n-1) + 2\*(n-1)). The reverse “textbook” multiplication with interleaving reductions is done when all bits in ‘a’ has been impacted whether ‘b’ is to be added with R<<1 and the result lies at ‘R’.

## Rsa Stage Module

One level up, can one find the rsa\_stage\_module. This is the module responsible for feeding two instances of blakeley\_stage\_module for doing modular exponentiation, using repeated squaring as given below.

        mask = 0b1

        for i in range(0, int(KEY\_LENGTH/PIPELINE\_STAGES)):

            if eSlice & mask:

                currentC = blakeley\_module(currentC, currentP, n)

            currentP = blakeley\_module(currentP, currentP, n)

            mask = mask << 1

The way this works, is that we split the exponent into bits and iterate upwards in the exponent. As the exponent can be written out as a sum of powers of two,

we can interpret the bit value at a given position as a boolean value, indicating whether this position contains the exponent corresponding power of 2. In other words,

, illustrating that it is only when the binary value of ‘b’ is one, the current 2’s exponent contributes to the result of the modular exponentiation. Starting at position i=0 in the exponent, one therefore needs to iteratively accumulate the two variables

* currentC = The current exponentiation result at bit i in the exponent. For each bit, currentC is potentially multiplied by currentP, which (as described above) represents . Seen from the next stage, curentC holds such that only an multiplication with is needed to currentC in the next stage.
* currentP = The variable being squared unconditionally, serving as the to me multiplied with seen from the next stage.

The representation with the mask variable is used to make the design closer to what is implemented in HW.

## RSA CORE

The model of both modules are with the described workings, able to do modular multiplication on their own. However, as described in section “RSA CORE MICROARCHITECTURE”, we implemented a pipelined multicore architecture, where asynchronous communications between the stages. A software implementation of this in python was implemented to investigate whether this pipelined architecture was

1. **viable** – it was crucial to detect whether it was possible to synchronize the transfer of information asynchronously between the stages with the control signals planned for. The designer of the multicore architecture needed to investigate the possibilities of dependencies between the stages as well as potential deadlocks,
2. **effective** – even though the high level model won’t be able to reflect the speed of the hardware implementation (as the OS still schedules the different threads with the same utilization of the CPU, i.e. no actual performance gain), one can investigate the any unforeseen delays between the pipeline stages and other dependencies that affect the overhead or the overall performance of the pipeline.

The complete high level model of the system uses ‘NUM\_PIPELINE\_STAGES’ threads which acts as hardware, all running the rsa\_stage\_module concurrently each operating on its slice of the exponent. To simulate the AXI interfaces both in and out (which also runs concurrently on hardware) two additional threads for these were created, as well as one for the control logic of the rsa\_core.

    threads = []

    threads.append(threading.Thread(target=rsa\_core\_control))

    threads.append(threading.Thread(target=axi\_in, args=(0,)))

    for i in range(1,NUM\_PIPELINE\_STAGES+1):

        threads.append(threading.Thread(target=rsa\_stage\_module, args=(eSlices[i-1],N,i)))

    threads.append(threading.Thread(target=axi\_out, args=(NUM\_PIPELINE\_STAGES+1,)))

    for thread in threads:

        thread.start()

    for thread in threads:

        thread.join()

To do the internal synchronization, real-time synchronization primitives are used. The inter-stage control signals IPI, ILI, IPO, ILO (read the michroarchitecture part for explanation), as well as some control signals to the control logic of the rsa\_core, are implemented by semaphores. The data propagates between the stages by using global variables and atomic mutexes. This makes all intermediates thread-safe.

Upon running the model of the rsa\_core, the controller fetches al testcases, and firstly allows the axi\_in module to fetch the next case from the file. The axi\_in module will through the very same ilx-ipx handshake mentioned in section “RSA CORE MICROARCHITECTURE”, now implemented with semaphores, request that stage 1 takes over it’s values. Once stage 1 has done it’s computations on the currentC and currentP, it will request the second stage to take over using the semaphore-based ilx-ipx handshake. This continues until the value has propagated the whole pipeline, and the correct result is ready on the axi\_out stage. This is an exact model of the system, even on a superscalar scale.

The model prints the progression and intermediate values all encryption messages has trough the pipeline. One example with a 8-stage pipeline is shown below.

A screen shot of a computer screen

Description automatically generated

In addition, the model also compares the results with what is expected and gives a report for all testcases as shown below.

A screenshot of a computer

Description automatically generated

Encryption sizes of small sizes were used in this demonstration to easier display the workings of the model.

# 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.

CPU

CPU

ZYNQ PROCESSING SYSTEM

RSA   
ACCELERATOR

DMA

AXI INTERCONNECT

AXI INTERCONNECT

RAM

MEM  
CTRL

ETH  
CTRL

**JUPITER NOTEBOOK  
(SW)**

LOAD BITFILE

SETUP DMA

SETUP RSA

ENCRYPT/DECRYPT

PLOT RESULTS

ZYNQ PROGRAMMABLE LOGIC

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).

valid

Sender  
(Master)

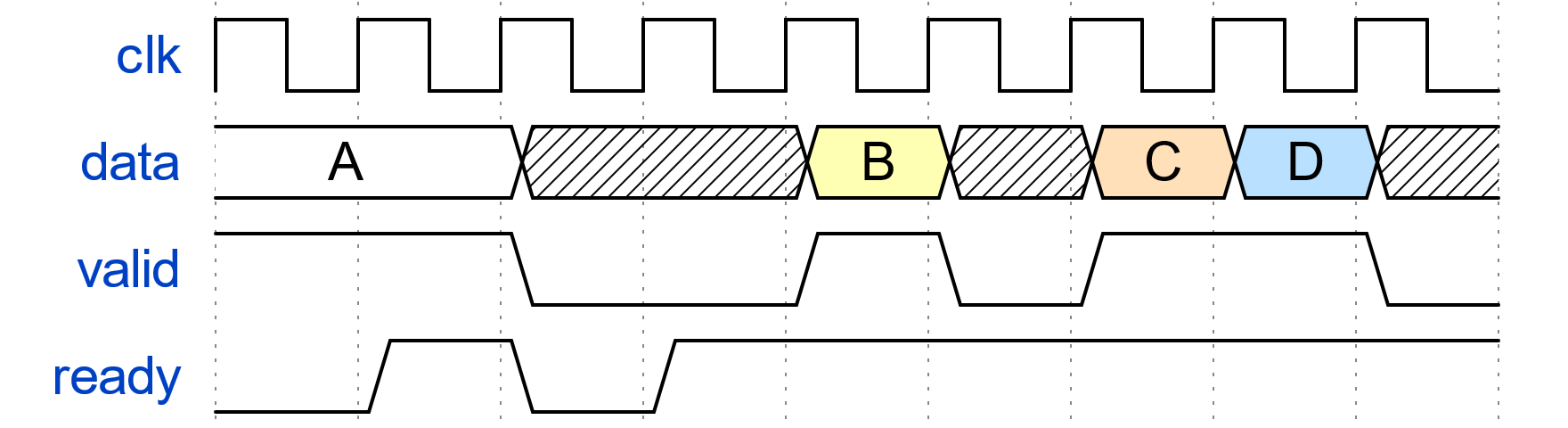
Receiver  
(Slave)

ready

data

valid

Figure 3. Sender and Receiver exchanging data.



**A**

**B**

**C**

**D**

**Time of**

**transfer**

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.

S00\_AXI

**rsa\_regio**

256

256

32

**key\_e\_d**

**key\_n**

**rsa\_status**

**rsa\_core**

256

msgin\_data

msgin\_valid

msgin\_ready

msgin\_last

256

msgout\_data

msgout\_valid

msgout\_ready

msgout\_last

**rsa\_  
msgin**

**rsa\_  
msgout**

S00\_AXIS

M00\_AXIS

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>**.

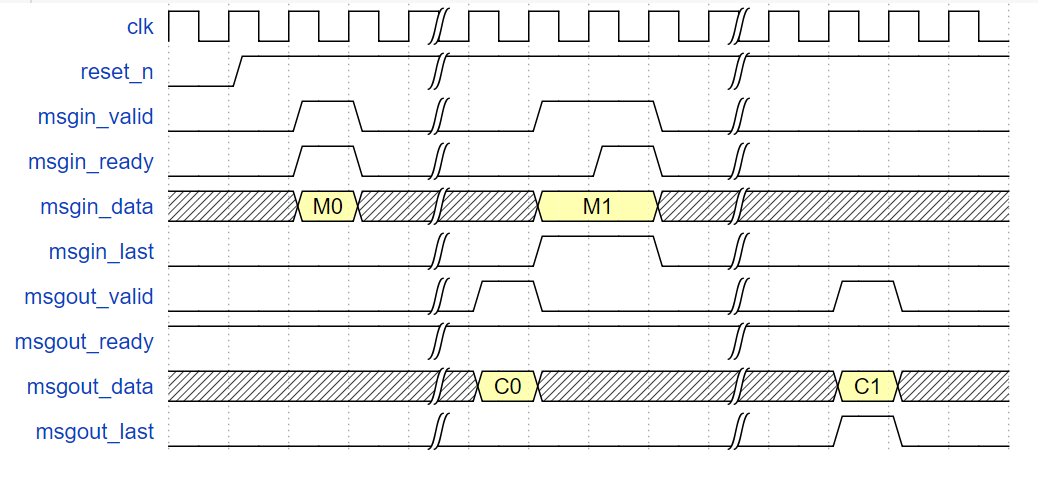


Figure 6. Message transport in and out of rsa\_core.

# RSA CORE MICROARCHITECTURE **(20 POINTS)**

## Blakeley module

This is the microarchitecture we designed for doing modular multiplication. This module works in the following way.

A computer screen shot of a diagram

Description automatically generated

It takes in a two unsigned operands, ‘A’ and ‘B’ and whenever these two values are valid. ABVAL must then be raised to signal to the module that its operands are ready. Once operands are ready, the following have been done in IDLE:

* REG1 which holds AINC (pointer to the current bit to check for 1 in the A operand) is reset.
* REG2 which holds the last sum of B and R<<1 is reset.

Whenever rsa\_stage\_module, the module over the blakeley\_module, raises the ABVAL signal at rising edge #1, both operands are ready. This also the blakeley\_module FSM into **RUN**. In this sate, the ainc(=0 initially) gets incremented each rising edge (ainc\_clk\_en = ‘1’) and a new intermediate add\_out is clocked in (add\_out\_clk\_en = ‘1’). Before we have rising edge #2, the current value of ainc gives the location in operand A that needs to be 1 of the whole B operand should be muxed trough the mux. This is the comparison done in CMP1. If A[(C\_BLOCK\_SIZE-1) – 1] = 1, the whole B then propagates down to the adder, and is added with the current result (now 0 as it was reset by the rsa\_stage\_module). On the upcoming rising edge, the FSM is in the state where it clocks in a new result in REG2 each clock cycle, leading to a clock in of the value B+0 at rising edge #2. The value ainc also gets incremented. The group is aware that this is the critical path of the design. One could have added a register at the input operands, but simulations show that the difference from this critical path, down to a lot of the other paths, were small. Thus, the potential increase in the clock frequency was not large enough to compensate for the decreased critical path. The input register was therefore not implemented in this design.

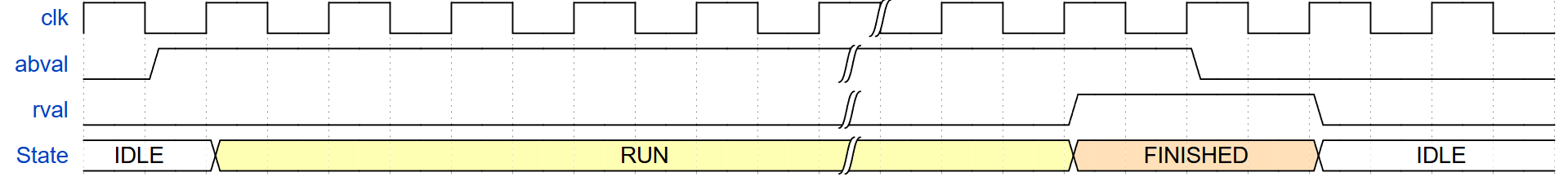
\*\*proof from sim\*\*

Once the intermediate result is clocked into the REG2 register, the intermediate value, called add\_out in blakeley\_module\_datapath.vhdl, propagates to the subtractors SUB1 and SUB2. Here the value add\_out is subtracting respectively nothing, 1x or 2x the modulus n, based of the comparison between add\_out and n happening in CMP2. The result is an intermediate R. Here also comes a second, but important, reason to why we have not included any additional registers in the blakeley\_module datapath. At rising edge #3, the value from MUX1 lies at the inputs of ADD2 and is the correct result of AINC=1. That is, it is the next correct value to add with this intermediate R. This enables us to have one single state, as stated before being **RUN**, simply enable the clock on reg1 and reg2 whenever in this state. The state diagram is shown below.

A diagram of a computer

Description automatically generated with medium confidence

The below diagram provides a how the interface signals abval and rval should be used in order to interface with the module.



## RSA stage module

This module has two main purposes

* Implementing a stackable, modularized interface to enable pipelining between multiple modules
* Running two instances of the blakeley\_module and controlling its datapath

We will first examine how the design implements each of these, starting by the first one.

It is crucial for the design to make it behave in such a way that several modules can be “stacked” beside each other without any modifications to ensure scalability. The rsa\_stage\_module implements a interface that make this possible by using 4 control signals:

* **Ili** (Intermediates Loaded In) = A input signal that goes to the previous stage’s ILO and is asserted whenever the previous stage intermediate values are ready to be popped off by the current stage.
* **ipo** (Intermediates Popped Out) = A output signal that goes to the previous stage’s IPI and is asserted by this stage whenever it has popped off the intermediate values from the previous stage.
* **ilo** (Intermediates Loaded Out) = A output signal that goes to the next stage’s ILI and is asserted whenever this stage’s outputs are ready to be popped by the next stage.
* **Ipi** (Intermediates Popped In) = A input signal that goes to the next stage’s IPO and is asserted buy the next stage whenever it as popped of this stage’s values.

To illustrate how this handshake is implemented, we introduce 2 stages, rsa\_stage\_module1 and rsa\_stage\_module2 as illustrated below, each with cascaded ipx and ilx signals.

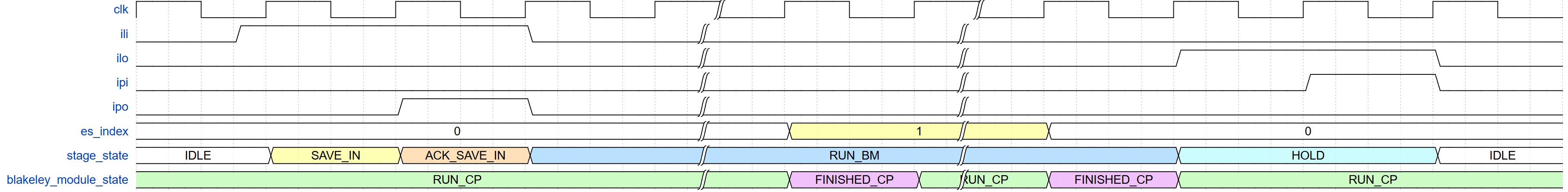
A diagram of a computer

Description automatically generated

When the time is right for rsa\_stage\_module1 to transfer its C and P to rsa\_stage\_module2, the following should happen:

1. rsa\_stage\_module1 asserts its ilo, signaling to rsa\_stage\_module2 that its intermediates are ready to be popped off
2. rsa\_stage\_module2 responds by clocking in the value lying on its DCI (Data C In) and DPI (Data P In), and then asserting ipo. This effectively signals rsa\_stage\_module2 has popped off the values from rsa\_stage\_module1.
3. Before rsa\_stage\_module2 can proceed, it needs to ensure itself that rsa\_stage\_module1, acknowledges that rsa\_stage\_module2 has taken over its values. This is ensure rsa\_stage\_module2 does not popp off the same value from rsa\_stage\_module1 twice, and is done by checking if ili is negated.
4. Both C and P have now successfully propagated from rsa\_stage\_module1 to rsa\_stage\_module2.

Below is a timing diagram, graphically showing how to use these signals in order to guarantee data consistency.



The implementation is done by a stage\_state FSM in rsa\_stage\_module\_control.vhdl. This FSM also controls the datapath of the rsa\_stage\_module and contains a inner blakeley\_module\_state FSM to run the 2 blakeley\_modules. The state diagram and the microarchitecture is shown below.

A diagram of a program

Description automatically generated

A diagram of a computer

Description automatically generated

In the following, the same modules as shown above rsa\_stage\_module1, rsa\_stage\_module2 and rsa\_stage\_module3 in cascade will be used to illustrate the workings of the rsa\_stage\_module.

When rsa\_stage\_module2 is in **IDLE**, both muxes are set to take inputs from DCI (Data C In) and DPI (Data P in). This is rsa\_stage\_module1’s DCO (Data C Out) and DPO (Data P Out). Upon getting an asserted ili from the rsa\_stage\_module1 stage, both REG3 and REG4 registers are reset, and on the upcoming rising edge, that is rising edge #1, the stage\_state FSM enter state **SAVE IN**. Here, both clocks for the registers are enabled, and the values from the previous stage will be clocked in at rising edge #2. At the same time, the stage\_state FSM will enter state **ACK\_SAVE\_IN**, tasked with acknowledging to the rsa\_stage\_module1 that it has taken over its values. This is done as described earlier by asserting the ipo signal, consecutively also setting rsa\_stage\_module1 in **IDLE**. When then rsa\_stage\_module1 enters **IDLE**, its ilo is set to ‘0’, hence making rsa\_stage\_module2 ili ‘0’ as well. This makes rsa\_stage\_module2’s stage\_state FSM to go into the state **RUN\_BM** where the muxes is shifted over to inputs M1.

Inside the sate **RUN\_BM** the blakeley\_module\_state FSM is used to control the blakeley\_modules. This FSM implements the handshake with the blakeley\_module, given in the timing diagram in figure XXX and checks if they are finished. In the state **RUN\_CP**, the intermediate R from each blakeley\_module is clocked in every clock cycle, when its respective \_clk\_en signal is set to ‘1’. From the high level model, we know that this must be done for every clock cycle for P, but only whenever bit es[es\_index] is 1 for C. The blakeley\_module\_state FSM remains in the state **RUN\_CP** until both c\_bm\_rval and p\_bm\_rval signals are asserted by the blakeley\_modue. The blakeley\_module\_state FSM then goes to state **FINISHED\_CP**, checking if its es\_index has had an overflow (i.e. es\_index = 0). If this is the case, the blakeley\_module has done modular multiplication on all bits in its e slice (es), and the result of both C and P is accumulated in REG3 and REG4 as the clocks are still enabled (c\_reg\_clk\_en <= e[es\_index] and p\_reg\_clk\_en <= '1').

When all bits in the encryption-key slice es is processed, the stage\_state FSM transitions to state **HOLD\_OUT**. Here the stage\_state FSM signal to rsa\_stage\_module3 that its values (C and P) are ready, by asserting ilo. Upon a response from rsa\_stage\_module3 with ipi=’1’, rsa\_stage\_module2 is aware that the next stage has taken its values, proceeding to enter **IDLE**. The transaction are now complete and rsa\_stage\_module2 are ready to pop the values from rsa\_stage\_module1.

## RSA core

As the microarchitecture now contains functionality for doing a complete modular exponentiation, either in a standalone configuration, or in a pipelined configuration, the design still needs an interface to the msg\_in, msg\_out and regio using respectively AXI and AXI light streams. For this reason, the following microarchitecture, with designated axi\_in and axi\_out modules was implemented to work as an interpreter between the internal ilx and ipx signals and the external msgx\_valid, msgx\_ready and msgx\_last signals.

A computer screen shot of a diagram

Description automatically generated

The input interpreter module axi\_in implements the axi\_out\_state FSM, walking trough the states , **GET\_FROM\_AXI** and **HOLD\_FOR\_PIPELINE** based off the signals from the axi interface and the pipeline.The output interpreter module axi\_out implements the axi\_out\_state FSM, walking trough the states **WAIT\_FOR\_PIPELINE**, **GIVE\_TO\_AXI**, **SIGNAL\_PIPELINE**, based off the signals from the axi interface and the pipeline. To keep track of what message is the last out, a FIFO buffer (message\_counter\_target) is with num\_pipeline\_stages+1 entries. This is a buffer that the axi\_in module fills at the position of the write pointer (message\_counter\_target\_wr\_ptr) and the axi\_out module reads at the position of the read pointer (message\_counter\_target\_rd\_ptr). It must have at least num\_pipeline\_stages+1 because if all stages are filled with only last-messages, the buffer will need to have space for the next entry, without overwriting some of the existing ones.

Lastly, a status register is implemented using a axi light stream to the rsa\_regio module. In the status register, one can find:

* STATUS[0-15]: Bit 7 from the internal status register from each rsa\_stage\_module, which is a bit indicating whether the module is in **RUN\_BM** state or not. This gives thus a good indication for all stages in the pipeline whether a stage is active and whether it does modular multiplication.
* STATUS[16-31]: A 16-bit performance counter, counting the number of clock cycles the last modular exponentiation took.

\*\*Final stages and remarks\*\*

# PERFORMANCE ESTIMATION **(8 POINTS)**

**<Estimate the number of clock cycles your system needs in order to encrypt/decrypt a message (worst case). Estimate the likely clock frequency of your design as well. >**

# 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>**

Cryptography in general already has a vital role in the

SDG9 - Cryptography underpins the security of modern digital infrastructures. By ensuring data integrity and confidentiality, it fosters a safe environment for technological innovation and industrial growth. RSA, one of the foundational public-key cryptographic algorithms, enables secure data exchange over insecure networks. Implementing RSA effectively enhances the robustness of communication systems, facilitating advancements in industries like finance, healthcare, and education. This secure foundation is essential for building resilient infrastructure and promoting sustainable industrialization.

SDG16 - Opressed people access to uncensored communication.

SDG7&13 - While cryptography contributes to societal goals, its implementations, particularly computationally intensive algorithms like RSA, have environmental implications. The processing power required for encryption and decryption consumes significant energy, potentially contributing to increased carbon emissions. By focusing on optimizing RSA implementations for energy efficiency—such as through hardware acceleration or algorithmic improvements—we can reduce energy consumption. This effort aligns with SDG 7 (Affordable and Clean Energy) by promoting energy efficiency and SDG 13 (Climate Action) by mitigating environmental impact.

# 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)