|  |  |  |
| --- | --- | --- |
| **RSA Hardware accelerator** | | NO SLEEP |
| Group | Group18 | |
| Authors | Knut Vågnes Eriksen  Lars Gjardar Musæus | |
| Date | 26.10.2020 | |

# 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 |  |
| **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 |  |
| **REQ\_PPA\_01** | MUST | Encrypt/decrypt a message of length 256 bits as fast as possible. |  |
| **REQ\_PPA\_02** | MUST | The design must fit inside the Zynq XC7Z020 FPGA on the Digilent Pynq-Z1 board. |  |
| **REQ\_PPA\_03** | MUST | There is no requirement for the clock frequency of the programmable logic. The platform supports any clock frequency. |  |
| **REQ\_PPA\_04** | SHOULD | The hardware accelerator should run testcase 4 faster than 400 ms. |  |
| **REQ\_INT\_01** | MUST | The RSA design must be integrated as a hardware accelerator inside the Zynq SoC. It must be managed by the CPU and made accessible through the Juniper notebook interface. |  |
| **REQ\_INT\_02** | SHOULD | The design should implement memory mapped status registers, performance counters and other mechanisms for debugging of features and performance at system level. |  |
| **REQ\_INT\_03** | MUST | The design must have one AXI-Lite Slave interface to enable access of memory-mapped registers. |  |
| **REQ\_INT\_04** | MUST | The design must have one AXI stream slave interface for input messages that shall be encrypted(decrypted) and one AXI stream master interface for output messages that have been encrypted(decrypted). |  |
| **REQ\_CONF\_01** | SHOULD | The design should be optimized for 256 bit block/message/key size. |  |

# 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. |  |
| **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. |  |
| **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 11 | Sign up on Blackboard | Form term project groups | X |
| Study algorithms and pick one | SEP 18 | Nothing to upload | Study algorithms and pick one | X |
| High level model | SEP 25 | Upload code on Blackboard | Implement the algorithm in python or another high level language. | X |
| Microarchitecture | OCT 05 | Upload diagram on Blackboard | Draw microarchitecture diagram for hardware design | X |
| Performance estimate | OCT 05 | Estimate performance. Upload to Blackboard. | Estimate the time needed to encrypt/decrypt a block. | X |
| Microarchitecture review/presentation | OCT 05 | Give presentation in class | Staff reviews the solutions proposed by each team and gives feedback. | X |
| RTL Code (Alpha) | OCT 26 | Upload RTL code to Blackboard. | Write synthesizable register transfer level code. |  |
| Testbench (Alpha) | NOV 2 | Upload Testbench to Blackboard. | Write testbenches for testing the design. |  |
| Working on FPGA (Alpha) | NOV 9 | Upload PPA on Blackboard. | Design working on FPGA. |  |
| Hand in this document with and all pieces of source code | NOV 19 | 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

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 **(10 points)**

|  |
| --- |
| import unittest  import random  k = 256  def modular\_product(a, b, n):  """  Input: a, b, n  Output: r = a \* b mod(n)  Note: n > a,b  """  r = 0  a\_bin = "{0:0256b}".format(a)  for i in range(len(a\_bin)):  r = 2\*r + int(a\_bin[i]) \* b  # r = r % n :  if(r >= n):  r = r-n  if(r >= n):  r = r-n  return r  def modular\_exp(m, e, n):  x = 1  p = m  e\_bin = "{0:0256b}".format(e)  for i in range(k-1, 0, -1):  if int(e\_bin[i]):  x = modular\_product(x, p, n)  p = modular\_product(p, p, n)  if int(e\_bin[0]):  x = modular\_product(x, p, n)  return x  def gen\_rand():  base = random.randint(1,1000)  exp = random.randint(1,1000)  modulo = random.randrange(base, base+1000, 2)  return base, exp, modulo  class tests(unittest.TestCase):  def test\_blakley\_exponential(self):  for i in range(100):  random.seed(i)  base, exp, modulo = gen\_rand()  expected = (pow(base,exp))%modulo  actual = modular\_exp(base, exp, modulo)  try:  self.assertEqual(expected, actual)  except:  print(f"\n{base}^{exp}(mod {modulo}) = (actual: {actual}, expected: {expected})")    def test\_blakley\_product(self):  for i in range(100):  random.seed(i)  a, b, n = gen\_rand()  n = a+b \* 2  expected = (a\*b)%n  actual = modular\_product(a, b, n)  self.assertEqual(expected,actual)  def test\_encrypt(self):  message = "0x0000000011111111222222223333333344444444555555556666666677777777"  m = int(message, 0)    power = "0x0000000000000000000000000000000000000000000000000000000000010001"  e = int(power, 0)    n = "0x99925173ad65686715385ea800cd28120288fc70a9bc98dd4c90d676f8ff768d"  n = int(n, 0)  expected = int("0x23026c469918f5ea097f843dc5d5259192f9d3510415841ce834324f4c237ac7",0)  actual = modular\_exp(m, e, n)  self.assertEqual(expected, actual)  print("Encryption successful\n")    def test\_decrypt(self):  message = "0x23026c469918f5ea097f843dc5d5259192f9d3510415841ce834324f4c237ac7"  c = int(message, 0)  power = "0x0cea1651ef44be1f1f1476b7539bed10d73e3aac782bd9999a1e5a790932bfe9"  d = int(power, 0)  n = "0x99925173ad65686715385ea800cd28120288fc70a9bc98dd4c90d676f8ff768d"  n = int(n, 0)  expected = int("0x0000000011111111222222223333333344444444555555556666666677777777", 0)  actual = modular\_exp(c, d, n)  self.assertEqual(expected, actual)  print("Decryption successful\n")  if \_\_name\_\_ == '\_\_main\_\_':  unittest.main(argv=['first-arg-is-ignored'], exit=False) |

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

The high level model uses the Blakley Interleaving Algorithm to compute the modular product in combination with the RL-Binary Method to compute the modular exponentiation. Moreover we have written unittests that randomly generates messages to encrypt and decrypt.

# 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)

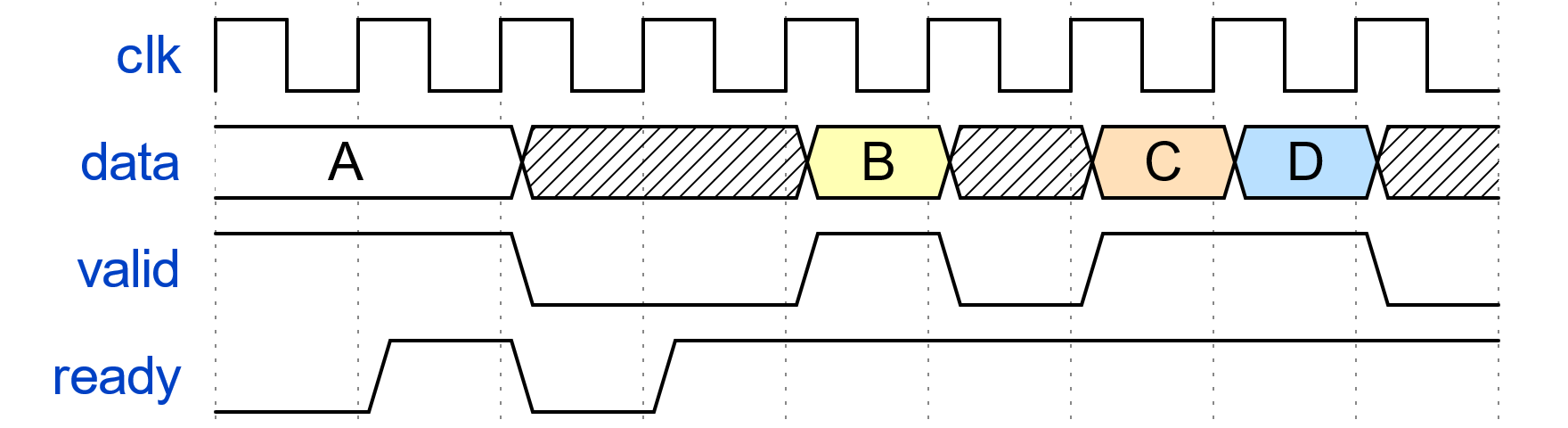
Receiver  
(Slave)

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>**.

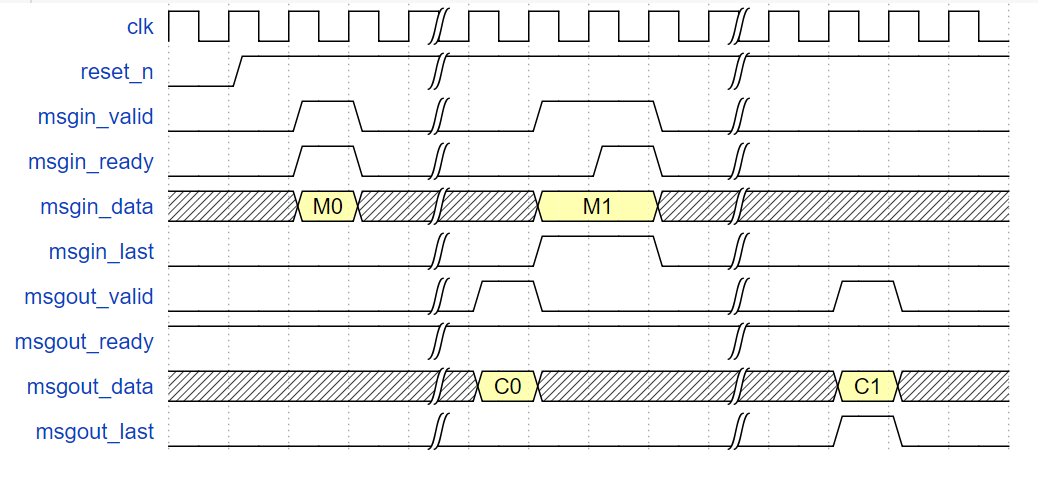
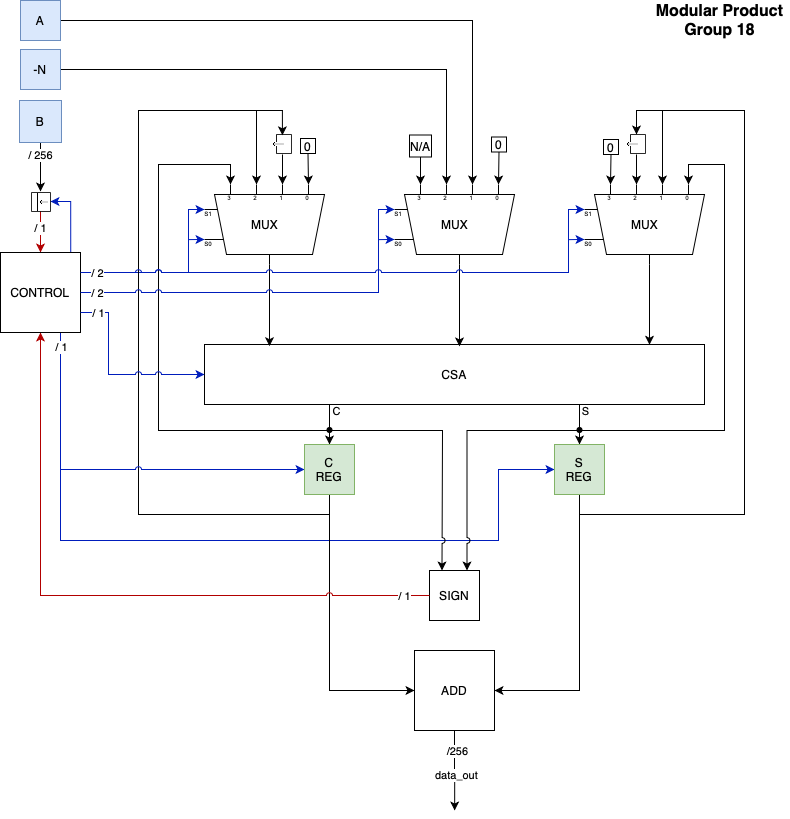


Figure 6. Message transport in and out of rsa\_core.

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



# PERFORMANCE ESTIMATION **(10 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. >**

256\*256 = 65536

# 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 **(10 POINTS)**

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

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

# PERFORMANCE BENCHMARKING ON FPGA **(30 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>**

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 **(10 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>**

Don't read from the signals to which you write. Have a correct sensitivity list (all signals that you read should be in the sensitivity list) Make sure that all signals to which your write are assigned in every path. (for example: in each branch of an if-else-statement) For processes which use variable, make sure every variable is initialized a default value before reading it (in another variable or signal ).

**HVOR ER DETTE HENTET FRA?^**

# EVALUATION CRITERIA

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

|  |  |
| --- | --- |
| **Model algorithm** | 10 points |
| **Microarchitecture** | 20 points |
| **Performance estimation** | 10 points |
| **Verification plan and verification summary** | 10 points |
| **Synthesis and implementation results** | 10 points |
| **Performance benchmarking on FPGA** | 30 points |
| **Source code quality** | 10 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)