UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO - CAMPUS FLORIANÓPOLIS
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Augusto Cezar Boldori Vassoler

Reconfigurable RSA accelerator based on efficient modular multipliers using high-performance modulos

Florianópolis
2023

Augusto Cezar Boldori Vassoler

Reconfigurable RSA accelerator based on efficient modular multipliers using high-performance modulos

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina para a obtenção do título de mestre em Engenharia Elétrica.
Supervisor:: Prof. Héctor Pettenghi Roldán, Dr.
Co-supervisor:: Prof. Eduardo Augusto Bezerra, Dr.

Florianópolis
2023

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Vassoler, Augusto Cezar Boldori
Reconfigurable RSA accelerator based on efficient modular multipliers using high-performance modulos / Augusto Cezar Boldori Vassoler ; orientador, Héctor Pettenghi Roldán, coorientador, Eduardo Augusto Bezerra, 2023.
94 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia Elétrica, Florianópolis, 2023.

Inclui referências.

1. Engenharia Elétrica. 2. Modular Exponentiation. 3. FPGA. 4. Criptography. 5. RISC-V. I. Roldán, Héctor Pettenghi. II. Bezerra, Eduardo Augusto. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

Augusto Cezar Boldori Vassoler

Reconfigurable RSA accelerator based on efficient modular multipliers using high-performance modulos

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Raimes Moraes, Dr.
Universidade Federal de Santa Catarina

Prof. Roberto de Matos, Dr.
Instituição Instituto Federal de Santa Catarina

Prof. Eduardo Luiz Ortiz Batista, Dr. (Suplente)
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Engenharia Elétrica.

Documento assinado digitalmente
Telles Brunelli Lazzarin
Data: 27/09/2024 16:32:26-0300
CPF: ***.400.719-**
Verifique as assinaturas em https://v.ufsc.br

Coordenação do Programa de
Pós-Graduação

Documento assinado digitalmente
Hector Pettenghi Roldan
Data: 03/09/2024 12:09:16-0300
CPF: ***.054.601-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Héctor Pettenghi Roldán, Dr.
Supervisor:

Documento assinado digitalmente
Eduardo Augusto Bezerra
Data: 03/09/2024 08:57:08-0300
CPF: ***.851.577-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Eduardo Augusto Bezerra, Dr.
Co-supervisor:
Universidade Federal de Santa
Catarina – UFSC

Florianópolis, 2023.

Este trabalho é dedicado à minha mãe, Salete, minha irmã, Gabriella, e ao meu falecido pai, Dolandy.

ACKNOWLEDGEMENTS

Agradeço primeiramente a Deus, pelo dom da vida e pela oportunidade de estar vencendo esta etapa.

À minha família, minha mãe, Salete, minha irmã, Gabriella, e meu falecido pai, Dolandy, por sempre estarem incondicionalmente ao meu lado me incentivando nos momentos mais difíceis, me dando forças e não me permitindo desistir jamais. Certamente nada disso seria possível sem vocês.

Aos meus orientadores Héctor Pettenghi Roldán e Eduardo Augusto Bezerra, por todo o conhecimento passado durante esse período, todo apoio, incentivo e compreensão nos momentos necessários, e pelo longo tempo dedicado à minha formação e caminhada acadêmica. Agradeço imensamente pelo grande exemplo de conduta pessoal e profissional.

Aos Meus queridos amigos Thomas, Pamela e Felipe, que sempre estiveram ao meu lado me apoiando e dando forças incondicionalmente para que esse objetivo fosse concluído.

A todos os colegas e amigos que fiz durante essa caminhada, em especial à equipe do Spacelab, por todo o auxílio técnico prestado, mas também pela parceria, trocas de experiência, risadas e bons momentos, os quais tornaram a jornada mais leve e encantadora. O crescimento pessoal e profissional que alcançamos, bem como as amizades que construímos, são certamente os retornos mais valiosos dessa caminhada.

Por fim, a todos os professores do Programa de Pós-Graduação em Engenharia Elétrica que contribuíram para minha formação com seus valorosos conhecimentos e experiência, bem como à Universidade Federal de Santa Catarina, por oferecer uma educação pública, gratuita e de qualidade.

“Great moments are born from great opportunity.”
(TESLA, N.)

ABSTRACT

O grande desenvolvimento das comunicações, alavancado pelo recente advento da internet das coisas (*internet of things*-IoT) e da indústria de *edge computing*, tem aumentado cada vez mais a troca de dados entre diferentes dispositivos. Este aumento do tráfego de informação leva inevitavelmente a um aumento da procura de segurança, que é garantida pela encriptação. Um exemplo é o Rivest-Shamir-Adleman (RSA), que é um dos primeiros sistemas criptográficos e continua a ser utilizado em diversas aplicações. Apesar disso, seu nível de segurança está diretamente ligado ao tamanho de sua chave, o que tende a reduzir o tempo total de cifragem/decifragem ao longo dos anos, contrastando com a imensa demanda por desempenho atual. A grande maioria dos aceleradores atuais faz uso do algoritmo de multiplicação modular Montgomery para realizar a exponenciação modular de blocos RSA. Apesar disso, esta abordagem é suscetível a um grande aumento no tempo total de execução com o aumento do tamanho das chaves criptográficas, devido à sua natureza iterativa. Porém, uma opção pouco explorada são os multiplicadores baseados em compressão, que fazem uso das árvores de compressão Dadda e Wallaca para realizar a multiplicação modular, eliminando a propagação de carry e extraindo o máximo de paralelismo da implementação, pois elimina iterações excessivas. Tais multiplicadores são suscetíveis ao tipo de módulo utilizado, de modo que $2^n \pm 1$ são chamados de módulos eficientes, enquanto módulos do formato $2^n \pm k$ apresentam pior desempenho. Esses módulos eficientes podem ser obtidos a partir dos módulos $2^n \pm k$ uma vez que são seus múltiplos, sendo chamados de psuedo-módulos. Pensando nisso, o presente trabalho propõe um acelerador criptográfico RSA baseado em pseudomódulos, no qual a exponenciação modular será realizada completamente no domínio do pseudomódulo, e apenas o seu resultado final será retornado ao domínio do módulo original, através da redução modular. Esta estratégia visa aproveitar ao máximo o menor tempo de execução dos módulos $2^n \pm 1$, evitando que o overhead de $2^n \pm k$ seja adicionado a todas as iterações da exponenciação modular. Esta abordagem foi comparada com duas outras: a multiplicação nativa e operação de módulo do FPGA, e a multiplicação modular de Montgomery. O sistema foi sintetizado para o FPGA Cyclone V GX 5CGXFC5C6F27C7N da Altera e integrado a um núcleo NEORV32 RISC-V. Os resultados mostraram que a abordagem baseada em compressão teve um tempo total de operação cerca de 10^7 vezes menor que o algoritmo de Montgomery, e apresentou uma frequência de operação cerca de três vezes maior que os demais casos, para cada comprimento de chave RSA até 1024, sendo o único sensível ao uso do pseudomódulo.

Palavras-chave: Exponenciação modular. RSA. Criptografia. FPGA. RISC-V. Acelerador de hardware. Multiplicadores baseados em compressão. Multiplicação modular de Montgomery.

ABSTRACT

The great development of communications, leveraged by the recent advent of the IoT and edge computing industry, has increasingly increased the exchange of data between different devices. This increase in information traffic inevitably leads to an increase in the demand for security, which is guaranteed by encryption. An example is RSA, which is one of the first cryptographic systems and continues to be used in several applications. Despite this, its security level is directly linked to the size of its key, which tends to reduce the total encryption/decryption time over the years, in contrast to the immense demand for performance today. The vast majority of current accelerators make use of the Montgomery modular multiplication algorithm to perform modular exponentiation of RSA blocks. Despite this, this approach is susceptible to a large increase in total execution time with increasing cryptographic key sizes, due to its iterative nature. However, a little explored option are compression-based multipliers, which make use of Dadda and Wallaca compression trees to perform modular multiplication, eliminating carry propagation and extracting maximum parallelism from the implementation, as it eliminates excessive iterations. Such multipliers are susceptible to the type of modulo used, so that $2^n \pm 1$ are called efficient modulos, while modulos of the format $2^n \pm k$ present worse performance. These efficient modulos can be obtained from the $2^n \pm k$ modulos once they are their multiple, being called psuedo-modulos. With this in mind, the present work proposes an RSA cryptographic accelerator based on pseudo-modulos, in which the modular exponentiation will be completely carried out in the pseudo-modulo domain, and only its final result will be returned to the original modulus domain, through reduction modular. This strategy aims to make the most of the shorter execution time of the $2^n \pm 1$ modules, preventing the overhead of $2^n \pm k$ from being added to all iterations of the modular exponentiation. This approach was compared with two others: the FPGA's native multiplication and modulo operation, and Montgomery's modular multiplication. The system was synthesized to the FPGA Cyclone V GX 5CGXFC5C6F27C7N from Altera, and integrated to a NEORV32 RISC-V core. The results showed that the compression-based approach had a total operation time about 10^7 times smaller than the Montgomery algorithm, and presented a frequency operation about three times higher than the other cases, to every RSA key-length untill 1024, being the only one sensitive to the pseudo-modulo usage.

Keywords: Modular exponentiation. RSA. Cryptography. FPGA. RISC-V. Hardware accelerator. Compression-based multipliers. Montgomery modular multiplication.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CLB	Configurable Logic Blocks
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
CSA	Carry Save Adder
CSR	Carry-Save Representation
DNN	Deep Neural Networks
DSP	Digital Signal Processing
EAC	End-Around-Carry
FA	Full-Adder
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSR	File Select Register
GCD	Great Common Divider
GPP	General Purpose Processor
HA	Half-Adder
IEAC	Inverted End-Around-Carry
IOB	Input/Output Blocks
IoT	Internet of Things
ISA	Instruction Set Architecture
LSB	Least Significant Bit
LSD	Least Significant Digit
LUT	Lookup Table
MMIO	Memory-mapped Input/Output
MSB	Most Significant Bit
MSD	Most Significant Digit
OAEP	Optimal asymmetric encryption padding
PKCS	Public-Key Cryptography Standards
PPA	Parallel Prefix Adder
RNS	Residue Number System
ROM	Read-Only Memory
RSA	Rivest-Shamir-Adleman
SoC	System-on-a-Chip
SSH	Secure Shell
SSL	Secure Sockets Layer

STA	Static Timing Analysis
TLS	Transport Layer Security
VLSI	Very Large Scale Integration
VPN	Virtual Private Network

CONTENTS

1 INTRODUCTION

The field of cryptography has witnessed rapid advancements in recent years, driven by the increasing need for secure communication and data protection in various domains, such as finance, healthcare, and national security. More than that, the exponential growth in the development and use of Internet of Things (IoT) technologies causes the amount of data sent between devices to increase at the same rate. For this reason, faster, smaller and more efficient encryption and decryption systems are required, allowing their implementation in embedded hardware as network nodes and increasing the data exchange speed, as occurs in edge computing applications (XIAO, Y. et al., 2019).

To ensure the necessary security for these scenarios, many cryptography systems have been developed over the last decades. One of the first standards to be published was the RSA, which is still widely used in many applications such as:

- Secure communication protocols as Transport Layer Security (TLS), Secure Sockets Layer (SSL), and Secure Shell (SSH) (SHOUP, 2000).
- Secure storing and exchange of information of various types through encryption of files and messages. Mostly used in e-mail and chat systems (LIU, Y.; GONG; FAN, 2018), password storing, and protecting digital content, such as eBooks, videos, and music, from unauthorized access and distribution (ÇALIŞKAN, 2011).
- Generation and verification of digital signatures, which are essential for ensuring the authenticity and integrity of electronic documents, software, and transactions (SHOUP, 2000).
- Secure key exchange between two parties without the need for a pre-existing shared secret, which is crucial in secure communication protocols and Virtual Private Networks (VPNs) (BHATTACHARJYA; ZHONG, X.; LI, X., 2019).
- Securing data storage, data transfer, and authentication mechanisms in cloud computing environments.

The RSA system is heavily based on modular multiplication and squaring operations, since it uses the modular exponentiation to perform encryption and decryption. Once these are the most costly operations in the processing, the overall performance directly depends on both. These operations are performed in the domain of modulos called public and private keys, which are obtained by the multiplications of two randomly generated large prime numbers.

In this way, the security of the RSA is based on the difficulty of factoring these large prime numbers, since the private key can be found through the factorization of the public key, thus enabling the decoding of the message. From the proposal of the algorithm until the present days, the computational power of processors has increased

significantly, and the only way to keep this standard safe is to increase the size of the generated prime numbers, consequently increasing the key's bit number. Nowadays, for an RSA encryption to be considered secure, it must be at least 1024-bit long. However, many communication systems are already using 2048-bit systems, and even 4096-bit encryption for more sensitive information. Unfortunately, this increase in security also causes a significant increase in time, complexity and consumption to perform the arithmetic operations involved. This makes system implementation more costly, slower and less efficient, especially in embedded hardware, where resources such as memory and processing power are limited.

Many devices of this kind use microcontrollers as the main processing unit, which usually do not have much processing power and can not implement protocols like RSA in a feasible time, needing dedicated co-processors to implement this kind of task (HAMEED et al., 2010). For this reason, the usage of Field Programmable Gate Arrays (FPGAs) have increased in the last years (TRIMBERGER, 2015), since they are versatile devices which allow easy hardware reconfiguration, counting on logical blocks, flip-flops/registers and block-RAMs, enabling a wide range of applications. One of this possibilities is the design of optimized hardware accelerators, which can be easily interfaced with soft-processors inside the FPGA or to external processors (SKLIAROVA; SKLYAROV, 2019).

In the last decade, different cryptographic hardware accelerators with FPGA implementation have been proposed (GOMES et al., 2022; COUSINS; ROHLOFF; SUMOROK, 2017; MATUTINO, P. M. et al., 2017; LOI; KO, 2018), many of them with standard architectures such as RISC-V (GOMES et al., 2022; NGUYEN-HOANG et al., 2022). Most of the proposed works make use of the Montgomery algorithm (MONTGOMERY, 1985) to perform modular multiplication and exponentiation. This algorithm is extensively explored in the literature, and several changes have been proposed to it over the years (NADJIA; MOHAMED, A.; MOHAMED, I., 2012; PU; ZHAO, 2009). However, these platforms enable other ways to perform modular operations, such as using the native multiplication and modulo functions of the hardware description languages, typically VHDL or Verilog. This kind of implementation tends to be much simpler in terms of code, but it also greatly restricts the possibility of optimization, so that the critical path will largely depend on the synthesis made by the tool used.

Another possible solution is to employ dedicated modular operators to perform the arithmetic operations. In these approaches, compression-based arithmetic circuits are commonly used, often making use of Dadda Trees (DADDA, 1965) and Booth encoding (FADAVI-ARDEKANI, 1993), mostly being intended for use in Residue Number System (RNS) applications (SOUSA; ANTAO; MARTINS, 2016). With these structures, modular operations using modulos in the format 2^n, $2^n - 1$ and $2^n + 1$, where n is the number of bits of the modulos, are known to have high efficiency, greatly simplifying

the operations and the conversions between residual and binary numbering systems. However, the keys used in RSA practical applications are outside these ranges, presenting the format $2^n \pm k$, where k is an integer number. This happens because this type of modulos are much more difficult to be factored when compared to the first ones mentioned, thus guaranteeing the necessary security for the system. Despite that, it is known that the multiplication architecture for this kind of modulos are less efficient than the three other ones, presenting bigger latency and area, due to the need to reprocess the bits of k (PATRONIK; PIESTRAK, 2017).

To circumvent these burdens, a promising strategy is the use of high-performance modulos. They consist of equivalent modulos of the format $2^n \pm 1$, which are obtained from the original $2^n \pm k$ by a constant multiplication, i.e., consisting of a multiple of the original number, but featuring a more efficient format (PATRONIK; PIESTRAK, 2017). Therefore, this property can be used to speed up modular arithmetic operations as multiplication by performing an initial conversion of the operands from the original to new one, which has a good format, then performing all the necessary operations in the high-performance domain and converting the final result back to the original modulos format. It can considerably reduce the operation time, specially for operations with a large number of bits and many cycles, as the modular exponentiation, since the gain in total operation time is cumulative.

Thus, the present work proposes modular multipliers based on compression as RSA cryptographic accelerators. Besides, it applies the little-visited technique of using high-performance modulos, to achieve a higher performance in the application.

1.1 GENERAL OBJECTIVES

This work has, as general objective, the development of a domain-specific reconfigurable accelerator for data encryption and decryption using RSA for application in FPGA soft-processors. The used arithmetic units are implemented similarly to a Wallace tree, using compression blocks and reinserting the carry-out bits to tree. In these structures, a first conversion will be carried out from the original modulos of format $2^n + k$ to its high-performance format $2^n - 1$, format in which all arithmetic operations will be performed, finally reconverting the result to the original modulos. Therefore, the specific objectives for carrying out such a task are listed below.

1.1.1 Specific Objectives

1. Compare the performance of three different methods for modular multiplication in FPGA technology: dedicated multipliers based on compression, module operation native to VHDL and Montgomery's algorithm.

2. Assess the benefits of highest performance modular exponentiation applied to RSA encryption/decryption.
3. Propose a portable architecture with a standard interface that can be easily integrated with state-of-the-art processors.
4. Elaborate examples with calculations for RSA encryption using the proposed architecture.

1.1.2 Contributions

The main contributions of the present dissertation are:

- First presentation of the high-performance modulos concept and validation of its applicability in the acceleration of modular multiplication, modular exponentiation and RSA cryptography operations.
- Validation of the compression-based multiplier structures generated by the software tool developed in (FERNANDES, 2021), and its implementation in an cryptography application.
- Evaluation of applicability of native VHDL modulo function for modular multiplication and exponentiation in the context of RSA cryptography in FPGA cores, being compared with state-of-the-art implementations (Montgomery modular multiplication and exponentiation).

1.1.3 Methodology

The dedicated multipliers used in this work are based on the implementation architecture proposed in (FERNANDES, 2021), in which a tool for automatic generation of arithmetic circuits in VHDL was developed. In this previous work, the generated circuits presented very efficient results in terms of area and delay, having been synthesized using the Standard Cells library 65 nm from the UMC (UMC, 2006). In the present work, they were synthesized using a FPGA Cyclone V GX 5CGXFC5C6F27C7N from Altera, in the Cyclone V GX Starter Kit board (INTEL, 2023).

2 BACKGROUND THEORY

This chapter presents the basic concepts around the RSA cryptography, as well as the existing modular exponentiation algorithms, the Montgomery modular multiplication, the proposal compression-based multiplication and the existing accelerator models.

2.1 RSA CRYPTOGRAPHY

The RSA cryptosystem was firstly proposed in (RIVEST; SHAMIR; ADLEMAN, 1978), being named by the initials of its authors surnames. It is considered an asymmetric cryptography system, since it uses different keys for encryption and decryption, and since the encryption key is given as public information, it is also called public-key cryptography system. The mathematical basis of RSA encryption is called modular exponentiation and can be summarized with Equation (1):

$$C = |M^e|_m \tag{1}$$

Here, M is the message to be encrypted, C is the encrypted message, e is the exponent and m is the public key, which is also the operation modulo. In this scenario, C, e and m are public, sent along with the encrypted message through the communication channel. The public key m is generated by choosing two large primes p and q, where $m = p \times q$. For a n-bit RSA encryption, these primes should contain $\frac{n}{2}$ bits each, so that m contains n bits, in order to make it difficult to factorize the key. The exponent e can be chosen arbitrary, but need to be pairwise prime to $p-1$. Common choices are 3, 5, 17, 257 and 65537, because of their low Hamming weight, i.e., their low number of ones in binary form, which reduces the number of multiplications needed for exponentiation.

On the other side of the communication channel occurs the decryption, which is performed as shown in Equation (2):

$$M = |C^d|_m \tag{2}$$

The encrypted message C and m are public and received through the channel, while the decrypted message M and decryption key d are private. Decryption key d is the modular multiplicative inverse of e with respect the Euler Totient function $\varphi = (p-1) \times (q-1)$ and is denoted by $d = |e^{-1}|_\varphi$, meaning that d follows from the relation $e \times d \equiv 1$, i.e., the Greatest Common Divider (GCD) between e and d must be 1. Therefore, the RSA key generation is performed by the following steps:

1. Select exponent e, a small prime with low Hamming weight.
2. Generate two large random primes q and p.
3. Calculate the modulo $m = p \times q$.

4. Calculate the Euler totient function $\varphi = (p-1) \times (q-1)$.
5. Verify that the $GCD(e, \varphi) = 1$, if not go back to step 2.
6. Calculate the decryption key $d = |e^{-1}|_{\varphi}$.
7. Publish e and m in a public domain.

Nevertheless, some precautions are recommended to increase the security of the system, such as adding extra requirements in the generation of p and q (WIENER, 1990), or even padding the message M with a proper mecanism, such as optimal asymmetric encryption padding (OAEP) (ZHONG, Y., 2022). As an example, suppose that a code message $M = 17111998$ needs to be sent through an unreliable medium, thus needing to be encrypted. Before the sender encrypts the message, the receptor needs to generate a pair of RSA keys. Following the steps described previously, the chosen prime numbers were $p = 62639$, $q = 53987$, and the exponent $e = 5$. From this, follow that $n = p \times q = 3381691693$ and $\varphi = (p-1) \times (q-1) = 3381575068$. Taking the modular multiplicative inverse of e with respect to φ results in decryption key $d = 2028945041$.

The public key pair (m,e) is then published, and the encryption is performed as follows in Equation (3):

$$C = |M^e|_m = |17111998^5|_{3381691693} = 407188056 \tag{3}$$

Now, the cipher C can be send over an unreliable channel, since it cannot be related to the original message M without the private key d, which is only available for the receptor. After receiving the encrypted text, the receptor can decrypt the message using the operation from Equation (4):

$$M = |C^d|_m = |407188056^{2028945041}|_{3381691693} = 17111998 \tag{4}$$

2.2 MODULAR OPERATIONS

Modular multiplication is the key operation of modular exponentiation, and can also be considered the kernel of modern cryptosystems, as RSA. It consists mainly in the calculation of $|a \times b|_m$ with a, b, $m \in \mathbb{N}_0$. In general, an intuitive approach would be to first perform the $a \times b$ multiplication, after that calculating the modular reduction to modulo m. However, this is an inefficient way to calculate multiplication, since the modular reduction tend to be a costly operation in hardware, especially when it uses division to be performed.

For this reason, many algorithms have been proposed in order to avoid these costly operations and improve the multiplication performance. Among them, one can mention the Barrett reduction algorithm (BARRETT, 1987), Karatsuba algorithm (KARATSUBA, 1995) and Montgomery modular multiplication algorithm (MONTGOMERY, 1985),

this last one being the most used of them. Besides that, there are also approaches that use dedicated hardware architectures, as in cases of RNS (ASIF; VESTERBACKA, 2017) or compression based modular multipliers (FERNANDES, 2021; PALUDO, 2020).

Therefore, this operation presents a wide range of applicable methods, but as said in chapter 1, this work will focus on 2 methods: the compression based multipliers and Montgomery multiplication. This choice is due to the fact that the use of compression operators generated by the software tool from (FERNANDES, 2021), and presented here in sections 2.2.1 and 2.2.2, are part of the main contribution of this work, while Montgomery's multiplication and modular exponentiation (section 2.2.3) are the state of the art for implementing modular exponentiation in hardware. The third method to be tested in this work is the direct multiplication in VHDL, followed by the native modulo operator. When implemented in ASIC, all the aforementioned approaches can be synthesized using exactly the circuit described in the hardware description language. On the other hand, when the implementation is done in FPGA, the synthesis is not done in this way, but rather using its internal resources, not necessarily using the same elements described. Despite this, the behavior of the circuits will still follow the expected pattern, as will be seen later.

2.2.1 Variable modular multipliers

Non-iterative approach:

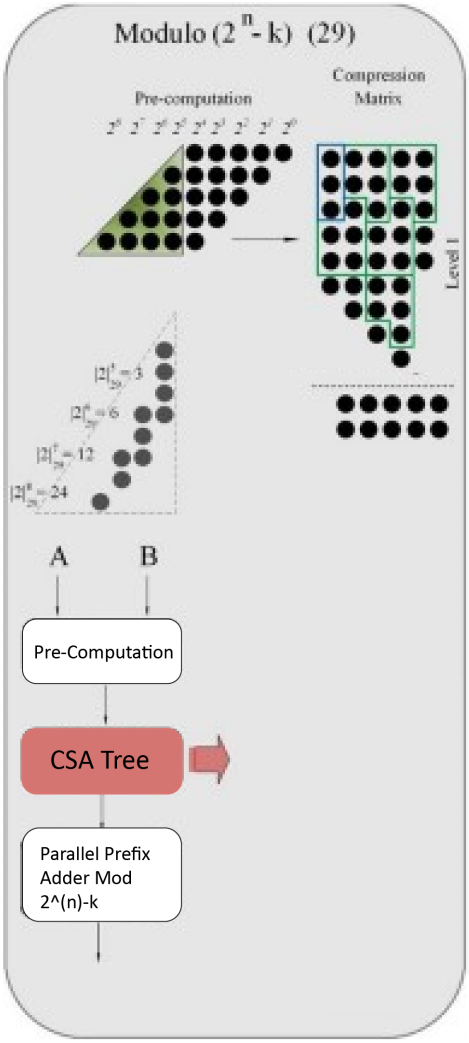
In general, the dedicated modular multipliers have different architectures, depending on the type of modulo used, which can be 2^n, 2^n-1, 2^n+1 and $2^n \pm k$, where k is a positive integer. An example of the standard implementation of a standard modular multiplication is presented in Figure 1, for a multiplication modulo $(2^n - k)$ with $n = 29$ and $k = 3$ (PARHAMI, 1996).

The pre-computation phase depends on the operation being performed and the type of modulo being used. On the other hand, the compression stage tend to be much more generic, presenting a more similar display in different situations, once it uses an carry save adder (CSA) tree for all cases. The purpose of such an element is to compress binary information, performing the sum of the vectors, which make up the information matrix, from the pre-computation process.

Iterative approach:

The circuits generated by the software tool from (FERNANDES, 2021) use an iterative approach to perform the compression. They are based on Dadda (DADDA, 1965) and Wallace (WALLACE, 1964) trees, using different levels of compression, in order to obtain 2 final vectors, which will be summed by a modular adder at the end of the operation. The used compression trees are made up by Half-Adders (HA), Full-Adders (FA) and 5:3 compressors (PATRONIK; PIESTRAK, 2017). Despite that, the compressors allocation still being a complex problem, and is a major topic of discussion

Figure 1 – Modular operation using CSA tree for n=29 and k=3



Source: The autor

in the literature (KIM, T.; JAO; TJIANG, 1998; STELLING et al., 1998; YU, Z.; YU, M.-L.; WILLSON, 2001).

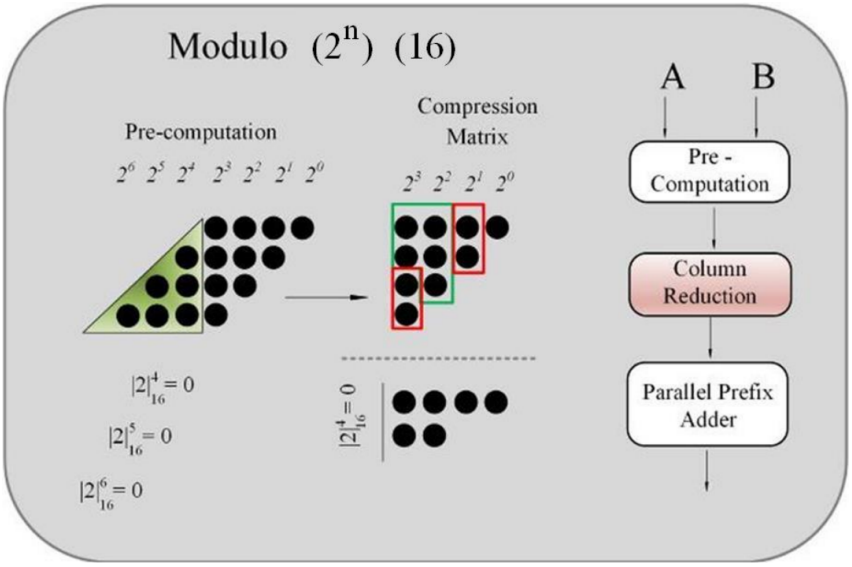
The used automated software tool gives compressors an order of priority based on their compression factors, meaning that the 5:3 compressor will come first, followed by FAs and HAs. After choosing the type of compressor from the priority list, they will be placed in order to reduce the information matrix until there is no possible fitting available. After that, the next compressor in the list will be chosen to satisfy any remaining possible fittings. The process will repeat itself until all compressors are covered. The attempt to place a compressor will occur from the column of the LSB to the column of the MSB by jumping one line at a time. Three compression solutions are generated. One uses only FAs, a second use 5:3 compressors and FAs, and a third uses all the stated compressors. The solution that provides the best delay per area will be chosen as the

final solution. The following examples make use of this allocation and can easily be extended to n bits.

2.2.1.1 2^n multipliers

In this case, the modular multiplication can be obtained by truncating the partial products with weight larger than 2^n due to $|2^n|_{2^n} = 0$, in the pre-computation process. The resulting A_j vectors are compressed by a CSA tree with truncate carry-outs. In the last stage, the two remaining are added using a Parallel Prefix Adder (PPA) modulo 2^n. Since the bits greater than 2^n, the adder will not need to use it carry-out signals. The structure for complete calculation can be observed in Figure 2, for a $n = 4$ example.

Figure 2 – 2^n modulo partial products for n=5



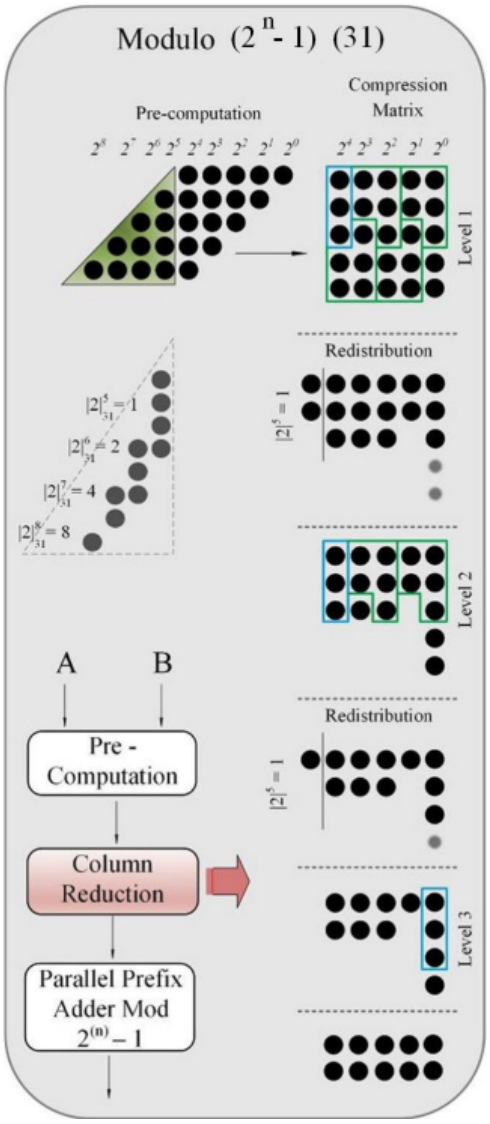
Source: The author

2.2.1.2 $2^n - 1$ multipliers

For modulo $2^n - 1$, the modular multiplication can be obtained by redistributing the positions of the partial products with weight greater than 2^n because $|2^{n+i}|_{2^n-1} = 2^i$ for $i \leq n$, to obtain the A_j arrays to add, as shown in Figure 3. It can be noticed that the weight 2^i, applied to the redistribution in the matrix, will always contain a single 1 in its binary representation, and thus is already conditioned to its best representation possible, without the need for any type of recoding. The reduction of the information matrix can be realized through the use of End-Around-Carry (EAC) in all the CSA tree levels, in order to reintroduce the carry bits to the matrix, since $|2^n|_{2^n-1} = 1$.

The maximum representation at the output of the compressor will be $A + B = 2 \times (2^n - 1)$, thus, for a modulo 31 as exemplified in Figure 3, the compression outputs

Figure 3 – $2^n - 1$ modulo partial products for n=5



Source: The autor

could take the number 30 as the highest value, and consequently, the highest value possible in the sum would be equal to 60. As this value is equal to twice the value of the modulo it is necessary to condition the output to a modular format. In this way, the modulo $2^n - 1$ adder is used, which performs the parallel calculation of additions $(A + B)$ and $(A + B - M)$, using the addition architecture presented in (FERNANDES, 2021). In the case of Figure 3 example, it is not possible to simplify the final sum hardware, since both vectors are complete. However, this is a specific solution. In general, the modular multipliers generated by the used software tool will automatically simplify the additions when the possibility is detected.

2.2.1.3 $2^n + 1$ multipliers

In case of modulo $2^n + 1$ multipliers, the modular multiplication can be obtained by redistributing the partial products with weight greater than 2^n since $|2^{n+i}|_{2^n+1} = -2^i$ for $i \leq n$. It is important to note that the weights reintroduced to the information matrix are negative, thus its necessary to introduce a corrector factor for the proper functioning of the calculation.

The addition of the corresponding A_j vectors in the compression tree can be carried out by applying an Inverted End-Around-Carry (IEAC) since $|2^n|_{2^n+1} = -1$, i.e., the weight is negative, so it needs to be inverted in its reinsertion in the information matrix. The addition of the compression phase output is performed by a modulo $2^n + 1$ PPA, which will be simplified when its possible. The pre-computation of partial products and the hardware architecture for a $n = 5$ example is shown in Figure 4.

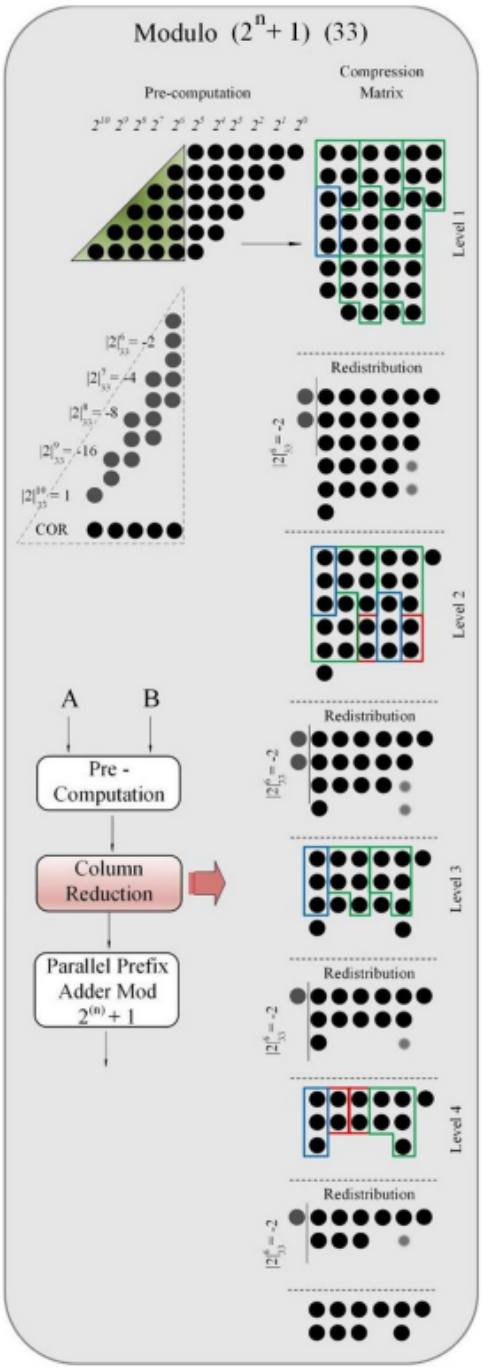
2.2.1.4 $2^n \pm k$ multipliers

In this case, the modular multiplication can be obtained by redistributing the positions with weight greater than 2^n because $|2^{n+i}|_{2^n-k} = \pm k \times 2^i$ for $i \leq n$, to obtain the A_j vectors to be added. Signed digits are used for the formation of these arrays and also for the bits that are reintroduced in the compressor tree during carry propagation. The choice of signed representation will depend on the number of ones of the specific weight. If both positive and negative representations have the same number of ones, the positive representation will have priority. Therefore, in this case, the need to include the correction factor will depend on the value of k. The A_j arrays can be added by applying EAC or IEAC in the CSA levels depending on the choice of representation. Finally, the outputs of the compression are added by a modular addition.

In Figure 5(b), the pre-computation of the partial product and the hardware architecture for $n = 5$ and $k = +11$. Note that the weight 2^6 can be represented as $|2^6|_{43}$ and is equal to 21 or –22. As both numbers have three ones in their possible representations, the positive number 21 is chosen to return as a partial residue. The rest of the weights $(2^7, 2^8, 2^9, 2^{10})$ are chosen as $-1, -2, -4, -8$ respectively, because they have only a single one in their binary representation. After the pre-computation, compression takes place, and afterward, modular addition concludes the calculation (MOHAN, 2012).

Note that in this example, the weight $|2^6|_{43} = 21$ will lead to a complexity increase, since each bit that leaves the matrix will be transformed into three bits when reintroduced at all compression levels. Smaller and simpler modules in the form of $2^n - k$ with k containing a reduced number of ones is preferable, but not always possible. For this reason, in this approach, the number of ones of k binary representation directly influences the multiplier delay, since the delay grows with the increasing num-

Figure 4 – $2^n + 1$ modulo partial products for n=5

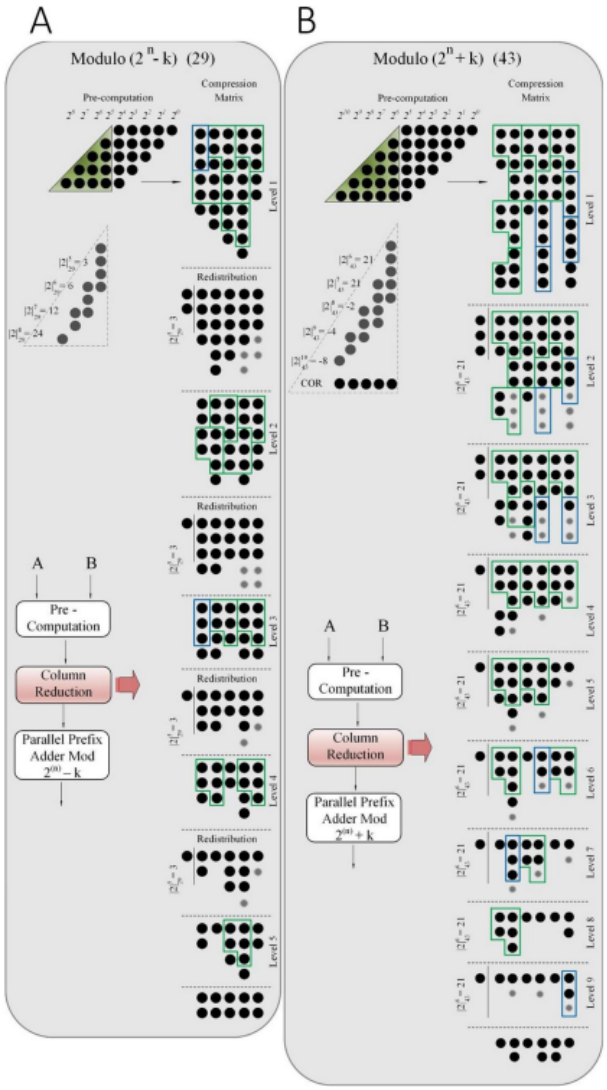


Source: The author

ber of ones. A different situation from some $2 \pm k$ standard architectures as those presented in (MATUTINO, P.; CHAVES; SOUSA, 2010), (MATUTINO, P. M. et al., 2012) and (VERGOS; EFSTATHIOU, 2007). In this implementations, the carry contribution of each vector are added and addressed to a Read-Only Memory (ROM), where they are stored to be added to the final two vectors from the CSA tree. This strategy eliminate the influence of the carry-out in to the total operation delay, which will be dictated by

the storing element.

Figure 5 – $2^n \pm k$ modulo partial products for n=5 and k=3 (A), and n=5 and k=11 (B)



Source: The author

2.2.2 Modular adders

As elucidated in subsection 2.2.1, the modular adders play an important role in the modular multiplication by adding the two final vectors, in an operation as $|x|_m = |A + B|_m$. Since A and B have a length of n bits, the maximum value for its sum is $A + B = (2^n - 1) + (2^n - 1)$. Depending on the modulo, this operation may exceed 3 times the value of the modulo itself, more specifically in the case of modulos $2^n + k$. Therefore, for performing the modular sum of these operands, the transformations presented Equation (5) are needed:

$$|X|_m = |A+B|_m = \begin{cases} A+B-3m, (\text{if } A+B \geq 3m) \\ A+B-2m, (\text{if } 2m \leq A+B < 3m) \\ A+B-m, (\textbf{if } m \leq A+B < 2m) \\ A+B, (\text{other cases}) \end{cases} \quad (5)$$

A widely used architecture for computing such modular sums is presented in Figure 6(a). The generic architecture, considering the case in which $A+B \leq m$, is made up of two adders that are connected in parallel. The first performs the sum $A+B$ while the second performs the sum $A+B-m$. As needed for performing subtractions by $2m$ or $3m$, adders can be connected in parallel and linked to a multiplexer for the correct selection of signals, as demonstrated in Figure 6b and Figure 6c. Note that for architecture that makes use of the sum $A+B-3m$, it is necessary to use the control logic indicated by Equation (6) and Equation (7), to reduce the number of inputs for mux connection from 4 to 1.

$$control(0) <= D_{Cout} \; or \; (not(C_{Cout} \; and \; B_{Cout})) \quad (6)$$

$$control(1) <= C_{Cout} \; or \; D_{Cout} \quad (7)$$

The indicated adder can be implemented by both CPAs and faster adders such as parallel prefix. In this work, Brent-Kung type parallel prefix adders were used (BRENT; KUNG, 1982).

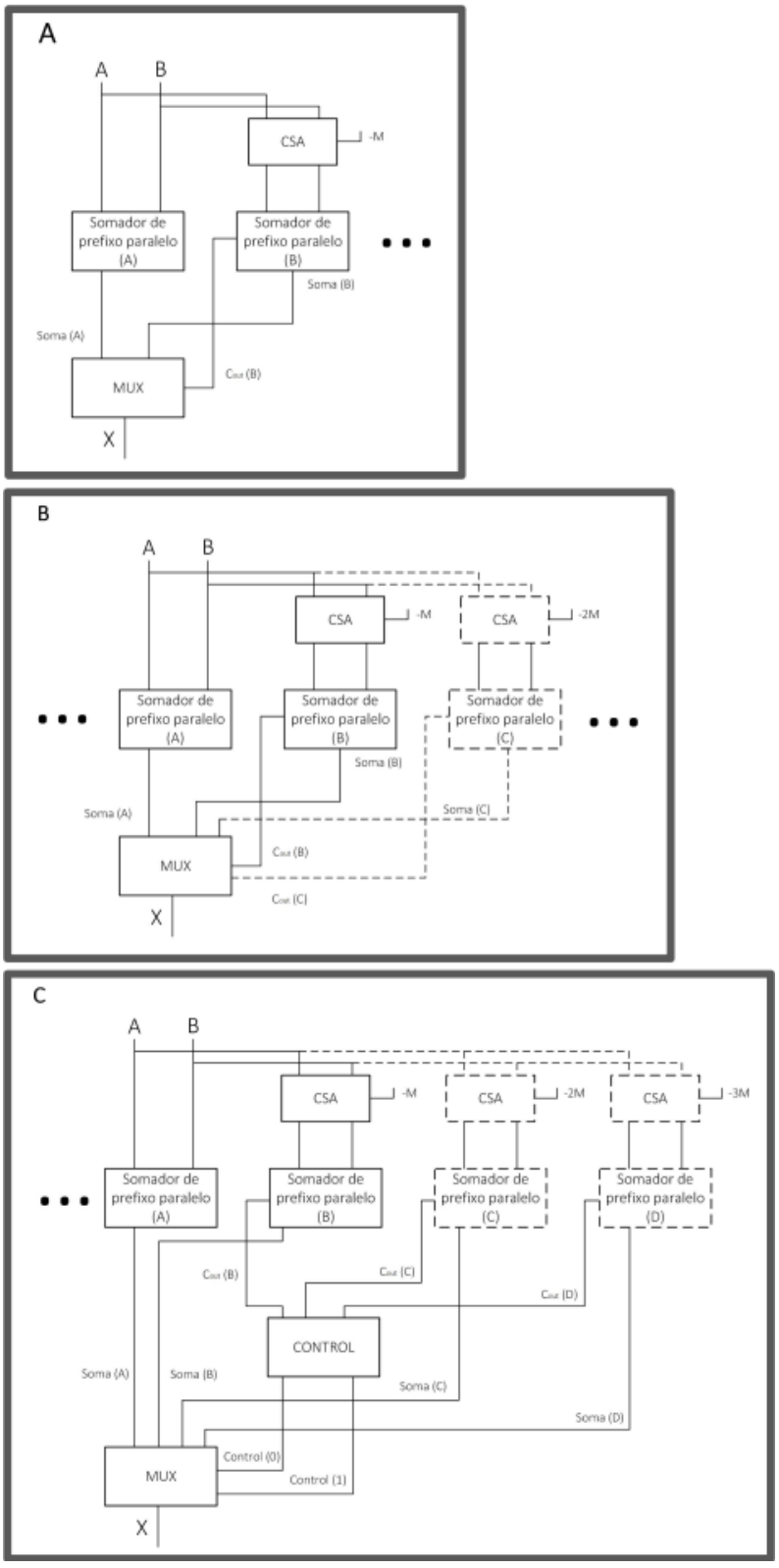
2.2.3 Montgomery Modular Multiplication

In 1985, Peter L. Montgomery proposed the currently known Montgomery modular multiplication (MONTGOMERY, 1985). It is an algorithm that improves the calculation of a modular multiplication $|a \times b|_m$ by replacing the modulo m with another number r. Being r a power of two, the division to perform the modulo reduction can be replaced by simple right shifts, but requiring some pre and post calculations in exchange. It happens because, before replacing m by r, the operands a and b need to be converted to it n-residue representation, also known as Montgomery form. The result of the Montgomery multiplication of a and b using modulo m and radix r is defined by Equation (8), where a, b and m are integers and r^{-1} is the inverse modular multiplicative of r with respect to m.

$$MontMult(a,b,m) = a \times b \times r^{-1} \; mod \; m \quad (8)$$

There are some restrictions for the choice of r; not all integers are eligible. r has to be greater than m, and both numbers must be coprimes. These constraints

Figure 6 – Architecture for modular addition



Source: The author

ensure that the inverse modular multiplicative of r with respect to m exists. Therefore, the conversion of a and b to their Montgomery forms $\overline{a}$ and $\overline{b}$ obtained through the modulo m multiplication with r, as in Equation (9).

$$\begin{aligned} \overline{a} &= a \times r \ mod \ m \\ \overline{b} &= b \times r \ mod \ m \end{aligned} \tag{9}$$

It also can be performed directly on the Montgomery algorithm, as in Equation (10).

$$\begin{aligned} \overline{a} &= MontMult(a, r^2 \ mod \ m, m) \\ \overline{b} &= MontMult(b, r^2 \ mod \ m, m) \end{aligned} \tag{10}$$

The reverse conversion is carried out by performing a Montgomery Multiplication of $\overline{a}$ and $\overline{b}$ with 1 being the second operand, as can be seen in Equation (11).

$$\begin{aligned} a &= MontMult(\overline{a}, 1, m) \\ b &= MontMult(\overline{b}, 1, m) \end{aligned} \tag{11}$$

Once a variable is converted to the Montgomery domain, one can perform as many operations as possible on it in an efficient manner, only the final result being reconverted. In this space, almost all arithmetic operations can be performed, such as addition, subtraction, multiplication, equality check and even the greatest common divider of a number with m.

The multiplication of two numbers in the Montgomery space requires an efficient computation of $|x \times r^{-1}|_m$. This operation is called the Montgomery reduction, and is also known as the algorithm REDC. Since r and m are coprimes, it is known that $m' < m$ such that $r \times r^{-1} - m \times m' = 1$, which is consequence of Bezout's identity. Reducing both sides of the equation modulo r, Equation (12) is obtained. In this way, m' is another integer required for the multiplication calculation.

$$m' = |-m^{-1}|_r \tag{12}$$

The Montgomery multiplication of two variable in m-residue representation is given by Algorithm 1. This modular multiplication can only be performed using operands converted to the Montgomery form, and after the calculation, the result need to be converted back to normal representation. The pre- and post-processing imposed by the method does not make it much faster than a conventional modular multiplication for a single operation. However, when repeated operations, it provides significant speed gain.

Algorithm 1 Montgomery modular multiplication

Input: a,b,m,$r(r > m, GCD(m,r) = 1)$
Output: $c = a \times b \times r^{-1} \ mod \ m$

1: $m' \leftarrow -m^{-1} \ mod \ r$
2: $t \leftarrow a \times b$
3: $q \leftarrow t \times m' \ mod \ r$
4: $u \leftarrow (t + q \times m)/r$
5: **if** $u \geq n$ **then**
6: $u \leftarrow u - m$
7: **end if**
8: **return** u

Many modifications have been proposed along the years for Algorithm 1 in order to improve different metrics such performance, power consumption and hardware resources usage. One of this modifications is the method proposed in (XIA; HU; YAN, 2009) and (PU; ZHAO, 2009), where the final subtraction is replaced by two iterations; thus, the size of the operand equal to (n+1) instead of (n–1). This approach is described in Algorithm 2:

Algorithm 2 Montgomery modular multiplication without final subtraction

Input: $A = a_{n+1}a_n...a_0, B = b_{n+1}b_n...b_0, M = M_{n-1}M_{n-2}...M_0$, with $A,B < 2M$, $a_{n+1} = b_{n+1} = 0$
Output: $S_{n+2} = A \times B \times 2^{-n-2} \bmod M$

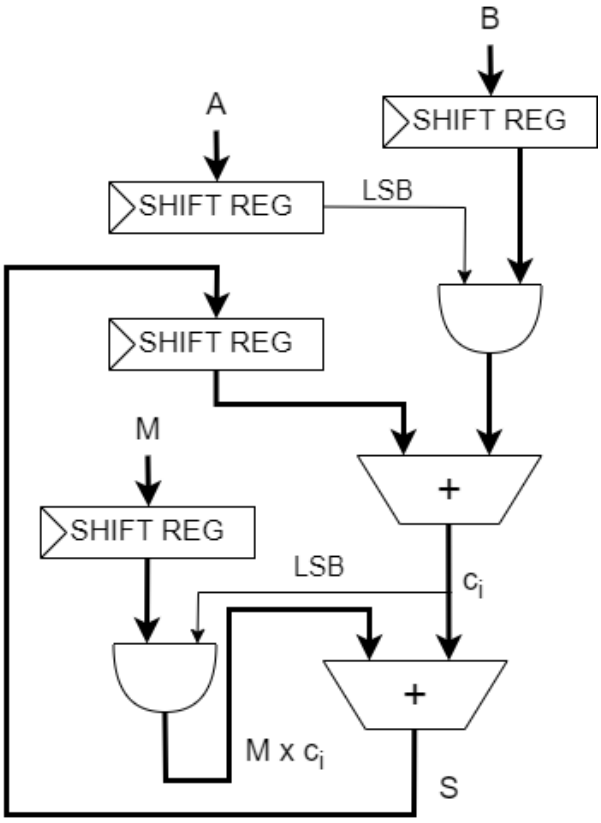
1: $S_0 \leftarrow 0$
2: **for** $i = 0$ **to** $n + 1$ **do**
3: $q_i = (S_i + a_i \times B) \bmod 2$
4: $S_{i+1} = (S_i + a_i \times B + 1_i \times M)/2$
5: **end for**
6: **return** S

This strategy allows reduction of the execution time and area. The multiplier will be implemented in this work as shown in Figure 7. Note that it has reduced complexity, using only two adders and a couple of and gates.

2.3 MODULAR EXPONENTIATION

The modular exponentiation presented in Equations (1) and (2) can be implemented in different ways. The so-called direct method (EL MAKKAOUI et al., 2022) firstly obtains $z = x^e$ and then, the modular reduction $|z|_m$, where x is the operand, e is the exponent and m is the modulus. This is clearly a slow and inefficient approach, especially for large operands and exponents, as RSA. For this reason, such an operation is performed using fast algorithms, the most common being next presented.

Figure 7 – Hardware implementation for Montgomery Modular Multiplication without final subtraction



Source: The autor

2.3.1 Repeated Multiplying

Keeping numbers small is one way to speed up exponentiation. Although it requires a greater number of modular reductions, multiplications will be faster, since they will have smaller operands, saving time and memory (M.T.GOODRICH, 2002). This is the purpose of repeated multiplying algorithm, which makes use of the identity property in modulo arithmetic, shown in Equation (13).

$$|a \times b|_m = ||a|_m \times |b|_m|_m \tag{13}$$

The formal algorithm is shown in Algorithm 3:

Algorithm 3 Repeated Multiplying exponentiation

Input: X,M,E
Output: $C = X^E \ mod \ M$

1: $C \leftarrow 1$
2: **for** $i = 0$ **to** $E - 1$ **do**
3: $\quad C \leftarrow (C \times X) \ mod \ M$
4: **end for**
5: **return** C

This method uses successive multiplications, and since it decrement the exponent in each iteration, it requires $O(e)$ multiplications for completion.

2.3.2 Binary Exponentiation

Binary exponentiation is probably the most widely known exponentiation algorithm. Also known as "square-and-multiply", it has an easy implementation in hardware, saving time, area and memory (ST DENIS; ROSE, 2006). It presents two different versions, which are called right-to-left and left-to-right binary methods. These names refer to the order in which the bits of the exponent are processed, starting from the Least Significant Bit (LSB) or from the Most Significant Bit (MSB).

The basic idea behind this method is to compute the exponentiation using the binary expression of the exponent e, thus breaking the operation into a series of squaring and multiplying operations. In this way, assuming l denotes the bit-length of the exponent e, it can be represented as $e = (e_{l-1},...,e_1,e_0)_2$, which can be sumarized as follows in Equation (14).

$$e = \sum_{0}^{e-1} e_i \times 2^i, \text{ where } e_i \in \{0,1\} \tag{14}$$

The left-to-right and right-to-left are formally presented in Algorithms 4 and 5 respectively.

Algorithm 4 Left-to-right binary exponentiation

Input: $X,M,E = (e_{n-1}...e_0)$
Output: $C = X^E \ mod \ M$

1: $R \leftarrow 1$
2: **for** $i = 0$ **to** $n-1$ **do**
3: $\quad R \leftarrow R \times R \ mod \ M$ ▷ Squaring
4: $\quad$ **if** $e_i = 1$ **then**
5: $\quad\quad R \leftarrow R \times X \ mod \ M$ ▷ Multiply
6: $\quad$ **end if**
7: **end for**
8: **return** R

Algorithm 5 Right-to-left binary exponentiation

Input: X,M,$E = (e_{n-1}...e_0)$
Output: $C = X^E \ mod \ M$

1: $R_0 \leftarrow 1$, $R_1 \leftarrow X$
2: **for** $i = 0$ **to** $n-1$ **do**
3: **if** $e_i = 1$ **then**
4: $R_0 \leftarrow R_0 \times R_1 \ mod \ M$ ▷ Multiply
5: **end if**
6: $R_1 \leftarrow R_1 \times R_1 \ mod \ M$ ▷ Squaring
7: **end for**
8: **return** R_0

The cost of naive exponentiation is e multiplications for any exponent e. For binary exponentiation, the minimum cost is given by $[log_2(e)]$ multiplications (GORDON, 1998), for when e is a power of two. An upper bound on the number of multiplications is given by $2[log_2(e)]$, for when e is a power of two minus one and consists of ones only.

As an example, to calculate the exponentiation $3^{23} \ mod \ 29$ by using the right-to-left method from 5, firstly the binary expansion of the exponent 23 need to be performed, which will result in Equation (15).

$$23 = 2^0 + 2^1 + 2^2 + 2^4 \tag{15}$$

Next, the terms 3^{2^0}, 3^{2^1}, 3^{2^2}, 3^{2^3}, $3^{2^4} \ mod \ 29$ need to be computed. It is interesting to note that each of these terms is the square of the previous one, hence the word square in the name "square-and-multiply". The Equation (16) shows all the squarings *modulo* 29, those marked with $*$ corresponding to the powers of 2 in the binary expansion of 23.

$$\begin{aligned} 3^{2^0} &\equiv 3* \\ 3^{2^1} &\equiv 9* \\ 3^{2^2} &\equiv 9^2 \equiv 23* \\ 3^{2^3} &\equiv 23^2 \equiv 7 \\ 3^{2^4} &\equiv 7^2 \equiv 20* \end{aligned} \tag{16}$$

After that, it is necessary to multiply all the $*$ marked numbers and perform the *modulo* 29 reduction, as shown in Equation (17).

$$\begin{aligned} 3^{23} \ mod \ 29 &\equiv (3 \times 9 \times 23 \times 20) \ mod \ 29 \\ 3^{23} \ mod \ 29 &\equiv (((27 \times 23) \ mod \ 29) \times 20) \ mod \ 29 \\ 3^{23} \ mod \ 29 &\equiv (12 \times 20) \ mod \ 29 \\ 3^{23} \ mod \ 29 &\equiv 8 \end{aligned} \tag{17}$$

These calculations are represented in a simpler way in Table 1, where the asterisks in the squaring column indicate that the results come from the powers of 2 in the binary expansion of the exponent, meaning that these will be the numbers to be multiplied in the third column.

Table 1 – 3^{23} mod 29 square and multiply operations

i	Squaring	Multiplying
0	3*	3
1	9*	27
2	23*	12
3	7	
4	20*	8

2.3.3 Montgomery modular exponentiation

In order to implement the modular exponentiation using the Montgomery modular multiplication, some modifications are needed. This work employs the implementations of the right-to-left method; thus, the adapted Algorithm 5 is presented in Algorithm 6:

Algorithm 6 Montgomery based right-to-left binary exponentiation

Input: X,M,$E = (e_{n-1}...e_0)$,r
Output: $C = X^E \ mod \ M$

1: $C = MontMult(1,r^2,M)$
2: $S = MontMult(X,r^2,M)$
3: **for** $i = 0$ to $n-1$ **do**
4: **if** $(e_i = 1)$ **then**
5: $C = MontMult(C,S,M)$ ▷ Multiply
6: **end if**
7: $S = MontMult(S,S,M)$ ▷ Squaring
8: **end for**
9: $C = MontMult(C,1,M)$
10: **return** C

First of all, it is necessary to perform two initial parallel modulo M Montgomery Multiplications, multiplying the factor r^2 by 1 for C and by the exponentiation basis X for S. C and S are the variables that holds the multiplying and squaring values, respectively; this process is necessary to initialize them, since it transform the operands to the so-called Montgomery domain that provides the output $C = |1 \times r|_M$ and $S = |X \times r|_M$. After that, the multiplication and squaring are performed according to the exponent bit scanning until its exhaustion, and the final result is converted back from the Montgomery domain by a final modulo M multiplication.

2.3.4 M-ary Exponentiation

The M-ary exponentiation is a generalization of the binary methods. While the last one consumes one bit per iteration, the *m*-ary algorithm is adapted to use more bits at same time, being *m* the radix. In the case of the binary exponentiation, it can be considered a 2-ary exponentiation.

Algorithm 7 presents the procedure for the m-ary modular exponentiation implementation. It is very similar to Algorithms 4 and 5, but instead of squaring, it computes r^m, and instead of multiplying by x, it can also multiply by x^2, x^3, ..., x^{m-1}, depending on the current radix-m digit of the exponent. Usually, these multiplicative values are pre-computed and stored (ARENAS-HOYOS; BERNAL-NOREÑA, 2017). Since the definition of MSB and LSB only exists for radix-2, it need to be replaced by the Most Significant Digit (MSD) and Least Significant Digit (LSD), respectively.

Algorithm 7 M-ary exponentiation

Input: $X, M, E = (e_{n-1}...e_0), m$
Output: $C = X^E \bmod M$

1: $R \leftarrow 1$
2: $<X^2, X^3, ..., X^{m-1}>$ ▷ Precompute m-1 powers of X
3: **for** $i = 0$ **to** $n-1$ **do**
4: $\quad R \leftarrow R^m \bmod M$ ▷ Always raise R by its radix
5: $\quad$ **if** $e_i \neq 0$ **then**
6: $\quad\quad R \leftarrow R \times X^{e_i} \bmod M$ ▷ Multiply R by a precomputed value
7: $\quad$ **end if**
8: **end for**
9: **return** R

For $m = 2^w$, the maximum number of multiplications is given by Equation (18).

$$2^w - 2 + (1 + \frac{1}{w}) \times log_2 e \tag{18}$$

From this, there are $2^w - 2$ of multiplications for pre-computation, $log_2 e$ squarings and at most $\frac{log_2 e}{w}$ multiplications (GORDON, 1998). For power of two exponents, only squares are needed, so by removing the part of multiplications, the new cost is given by Equation (19).

$$2^w - 2 + log_2 e \tag{19}$$

It uses considerably more memory when compared to binary exponentiation, as it needs to store the pre-computed values.

2.3.5 Sliding window exponentiation

Also known as 2^w-ary exponentiation, it is another generalization of the binary algorithm. This method can be considered as looking at the binary representation of

the exponent *e* through a fixed windows of width *w*, performing all operations related to that group of bits at once. In order to do that, the odd powers of *x* are pre-computed and stored, followed by cycles of squaring *w* times and one multiplication with a pre-computed value. In order to avoid zero-valued windows, the LSB is required to be one, and *w* is odd. The sliding window exponentiation implementation can be found in Algorithm 8.

Algorithm 8 Sliding window exponentiation

Input: X,M,$E = (e_{n-1}...e_0)$, w
Output: $C = X^E \ mod \ M$

1: $R \leftarrow 1$
2: $<X^3, X^5, ..., X^{2^w-1}>$ ▷ Precompute odd powers of X
3: **for** $i = 0$ **to** $n-1$ **do**
4: **if** $e_i \neq 0$ **then**
5: $R \leftarrow R^2 \ mod \ M$ ▷ Square result R
6: **else**
7: $s \leftarrow max(i-w,1)$ ▷ Ensure position s is not negative
8: **while** $e_s = 0$ **do**
9: $s \leftarrow s+1$ ▷ Decrease window position by 1
10: **end while**
11: **for** 0 **to** $i-s+1$ **do** ▷ Calculate $s-i+1$ times the square of R
12: $R \leftarrow R^2 \ mod \ M$
13: **end for**
14: $u \leftarrow decimal(e_i,e_{i-1},e_s)$ ▷ Gets $i-s+1$ bits from e starting at i
15: $R \leftarrow R \times X^u \ mod \ M$ ▷ Multiply R by a precomputed multiple of X
16: **end if**
17: **end for**
18: **return** R

One possible way to determine windows for a given exponent is to look at the binary representation of the exponent from left to right, starting a new window whenever a nonzero bit is encountered, choosing the maximum width up to *w* for this particular window such that the rightmost bit is also nonzero:

$$00\underline{1110}\underline{1}00\underline{11}00\underline{101}0$$

Another possibility is to look at the binary representation of the exponent from right to left, starting a new width-*w* window whenever a nonzero bit is encountered:

$$00\underline{111010}00\underline{11}00\underline{101}0$$

There is no reason to force the windows to be adjacent to each other. Neighbour zeros in the binary representation of *e* do not result in additional multiplications (only squares) and may be skipped. Its hard to define the total operation cost, since it depends on many factors, but the cost function is summarized in Equation (20) (KNUTH, 1997).

$$O((\frac{n}{w}) \times w + 2^w) \tag{20}$$

In the approximation of Equation (20), n is the number of bits in the exponent and w is the window size. The term $\frac{n}{w}$ represents the number of windows in the exponent, and the multiplicative w is related to the number of operations (squaring and multiplication) within each window. Finally, 2^w accounts for the pre-computation of values for each window.

Modifications have been introduced into the algorithm along the years, in order to improve performance and efficiency. Some examples are (BOS; COSTER, 1990), (MÖLLER, 2003) and (UENO; HOMMA, 2023). It is important to notice that these metrics depends directly on the windows sizing, which need to be optimally chosen. Also, it uses more memory, hardware resources and presents more complexity than the other two methods.

2.4 DOMAIN-SPECIFIC ACCELERATORS

This section introduces the domain-specific accelerator concept, and also the existing types of accelerators. Among them, one will be chosen to implement the project.

According to (DALLY; TURAKHIA; HAN, S., 2020), the most computing today, from an embedded processor in an appliance to large data centers, takes place on General Purpose Processors (GPP), also referred to as Central Processing Units (CPU). They are attractive since they are easily programmed and there is a large availability of code libraries for them. With the advent of Very Large Scale Integration (VLSI), nowadays there are CPUs with enough computational power to perform operations that a few decades ago were impractical for them, especially multiplication, division and exponentiation.

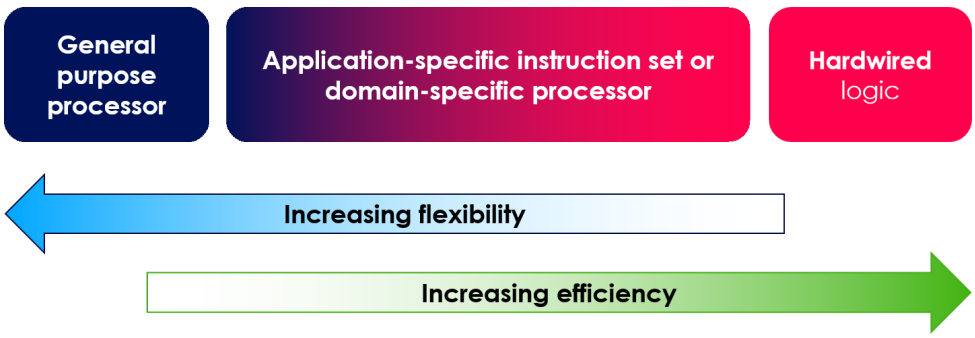
Despite this, CPUs are considered inefficient machines, as also pointed in (HAMEED et al., 2010). In modern RISC processors, the execution of a simple instruction as addition or branch, requires memory fetch, decodification and only after that the arithmetic operation is performed, followed by result storing. In this process, the amount of energy required to fetch and decode the instruction can be 10 to 4000 times more than that required to perform a simple arithmetic operation, also requiring extra clock cycles to conclude it. Thus, these instruction overheads use much more energy than arithmetic operations that perform useful work, which drastically reduces computing efficiency. It is known that complex instructions are capable of reducing the overhead, but even so the problem persists to CISC architectures.

In addition to that, the Moore's Law, which says that the number of transistors in a silicon chip doubles each two years, is coming to its end (WILLIAMS, 2017). As consequence, a significant improvement of the performance and efficiency for com-

putationally expensive as cryptography or artificial intelligence only can be achieved by the specialization of the processor. In this way, in order to keep escalating performance and efficiency, domain-specific accelerators emerge as a necessity for this tasks implementation (DALLY; TURAKHIA; HAN, S., 2020).

Domain-specific accelerators are processors or set of processors optimized for a given task, being adapted to meet the needs of the algorithms required for its application. For example, cryptographic accelerators usually contain dedicated hardware to implement modular multiplication and exponentiation, while accelerators for Digital Signal Processing (DSP) or Deep Neural Networks (DNN) commonly presents multiply and accumulation operations. Recently, they are being used not only to accelerate the computation time, but also to reduce the power consumption of certain operations. This becomes possible because even if the peak energy consumed by an accelerator is greater than that of a GPP, the reduction in the task execution time is so significant that the total energy required to complete it will be lower (KRISHNAKUMAR et al., 2023). As can be seen in Figure 8, the classification of the accelerator vary according to its specialization. Some of them are closer to GPP cores, allowing more programmability and presenting a bigger range of instruction, thus allowing more flexibility of implementation. On the other hand, other architectures are more similar to dedicated logic hardware, having a much more limited instruction set, but also presenting a better optimization for the relation silicon area and power consumption versus delay.

Figure 8 – Flexibility *versus* Efficiency comparison for domain specific accelerators



Source: (URQUHART, 2021)

According to (DALLY; TURAKHIA; HAN, S., 2020), in addition to eliminating overhead, modern accelerators use other techniques to ensure greater performance and energy efficiency. These techniques usually consist in the use of a specific type of data, higher parallelism and the use of local and optimized memory, which increase the speed of access to stored elements. As stated by (PATTERSON; HENNESSY, 2017), the new trend is for computers to have GPPs to run large, conventional programs, such as operating systems, together with specific-purpose accelerators that perform a small

number of tasks quickly and efficiently.

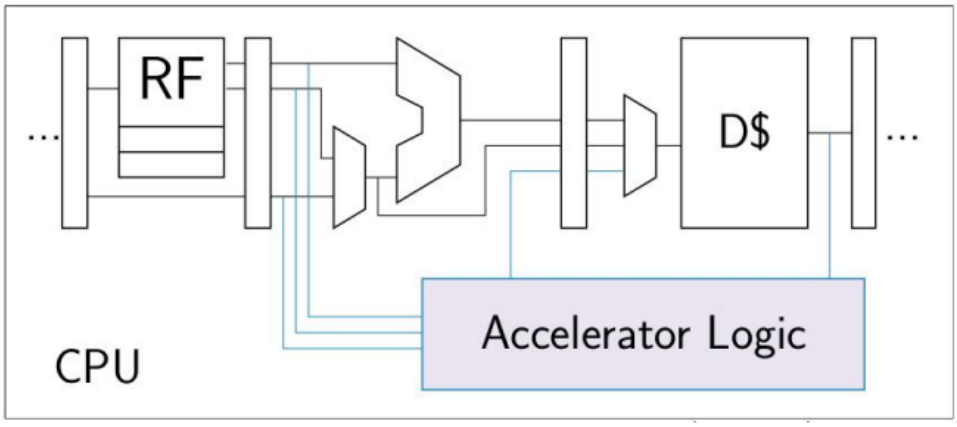
2.4.1 Models of Accelerators

In general, accelerators are broadly classified into loosely-coupled and tightly-coupled, depending on their degree of integration with the processor. Each strategy presents different implications in the design, and must be chosen based on project specifications (MANOR; GREENBERG, 2022).

2.4.1.1 Tightly-Coupled Architecture

Tightly-coupled consist of one or more hardware functional units which can accelerate critical portions of the application kernel, for example, the body of an inner loop for an algorithm or a sequence of trigonometric functions. This type of accelerator is located inside, or very close to, the processing core, being considered as an additional functional unit that is directly connected to the CPU data-path, as shown in 9. In this scenario, it is necessary to expand the Instruction Set Architecture ISA in order to add instructions to access and manage the accelerator, since it is an integral part of the pipeline. For this reason, it has a sequential execution, sharing core resources such as register banks and memory, therefore paralyzing the processor's execution until the accelerator task is completed (XIAO, C. et al., 2014) (MANOR; GREENBERG, 2022) (COTA et al., 2015).

Figure 9 – Tightly-coupled accelerator



Source: (COTA et al., 2015)

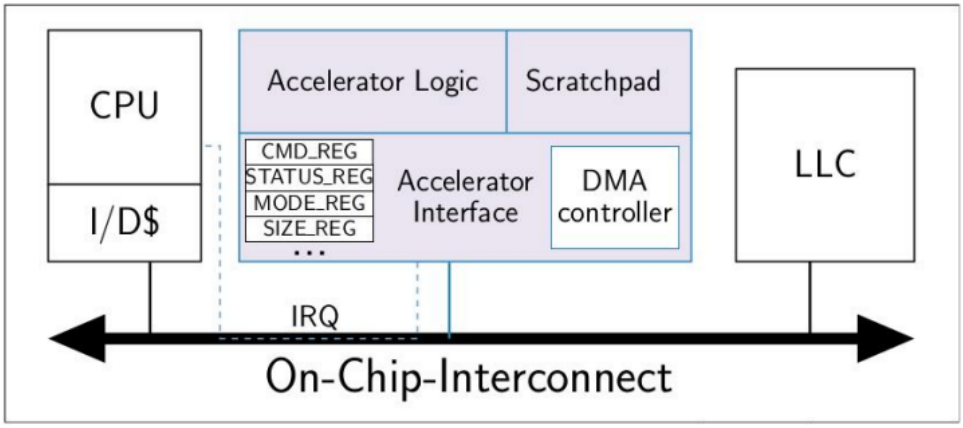
The main advantage of this approach is its very low or even zero communication and control overhead. On the other hand, it can increase considerably the complexity of the CPU design, imposing challenges such as synchronization, non-determinism and clock restrictions. Furthermore, its portability to different systems becomes very limited,

as it is often necessary to adapt the accelerator interface with the CPU, and even make control changes in the pipeline (COTA et al., 2015) (MANOR; GREENBERG, 2022).

2.4.1.2 Loosely-Coupled Accelerators

The loosely-coupled accelerators are located outside the CPU core and interact with the CPU through an on-chip interconnect, as shown in Figure 10. Being out-of-core, this type of accelerators a greater area than the tightly-coupled ones but they do not degrade the processor pipeline's performance or the cache access time. This allows for coarse-grained accelerator logic blocks with complex datapaths that implement and accelerate a complete application kernel, for instance a Fast-Fourier-Transform or a full image encoding algorithm. Furthermore, the out-of-core enables the implementation of private local memories, also known as scratchpads (BANAKAR et al., 2002), which store the input data to be processed, temporary results, and the output data to be written back to memory (COTA et al., 2015).

Figure 10 – Loosely-coupled accelerator



Source: (COTA et al., 2015)

This type of accelerator is advantageous for the easier implementation and largest portability when compared to the previous one, presenting a greater design freedom. Furthermore, it also does not interfere with the execution of the processor pipeline, as it runs parallel to it. The main disadvantage of this architecture can be the data overhead, which can be larger than the tightly-coupled approach due to the communication bus. However, researches as that presented in (COTA et al., 2015) and (MANOR; GREENBERG, 2022) shows that for workloads with non-trivial data sizes, loosely coupled accelerators perform better.

3 RELATED WORKS TO RSA HARDWARE ACCELERATORS AND MODULAR EXPONENTIATION STRATEGIES

Based on exploratory research, several works were identified that developed architectures to improve encryption. In general, it is clear that most implementations specifically of modular multiplication and exponentiation, mainly ECC and RSA, are based on Montgomery's modular multiplication algorithm, and commonly make use of binary algorithms left-to-right and right-to-left to perform the exponentiation. This chapter also presents details about FPGA devices and the RISC-V architecture, which make up the platform chosen for implementing the project.

3.1 ASIC IMPLEMENTATIONS OF RSA ACCELERATORS

When the first advances in this area emerged, the use of FPGAs was uncommon, so initial projects were synthesized using other technologies. As an example, we can mention the co-processor presented in (ROYO; MORAN; LOPEZ, 1997), which was synthesized using ASIC technology and proposes one of the first RSA co-processors, aiming at integration with a generic and making use of CSR for Montgomery multiplication, in an attempt to eliminate the carry propagation. Likewise, (ZHENG; LIU, Z.; PENG, 2008) presents the development of an integrated circuit for RSA acceleration for an 8051 CPU, with which it communicates via an file select register (FSR) protocol. Other examples of this are (YEH et al., 2006), (HISAKADO et al., 2006) and (CHEN; TSENG; CHANG, 2007), which are listed in Table 2.

Table 2 – Comparison between ASIC designs

Reference	Year	Technology	N. of bits	Area	Frequency	Power
Yeh	2006	UMC0.18	1024	5.76 mm^2	460 MHz	830 mW
Hisakadot	2006	TSMC0.18	2048	98500 *Gates*	60 MHz	61.5 mW
Chen	2007	UMC0.18	1024	175000 *Gates*	370 MHz	———
Zheng	2008	TSMC0.18	2048	61000 *Gates*	200 MHz	32.5 mW

Source: The author

It is notable that this type of application tends to prioritize consumption and area saving as the main metrics, unlike the present work, which aims to improve performance. This fact is probably due to resource constraints in ASIC applications, since excessively large circuits tend to increase the cost of the project and restrict the usability of the system.

3.2 IMPLEMENTATIONS OF RSA ACCELERATORS IN FPGAS

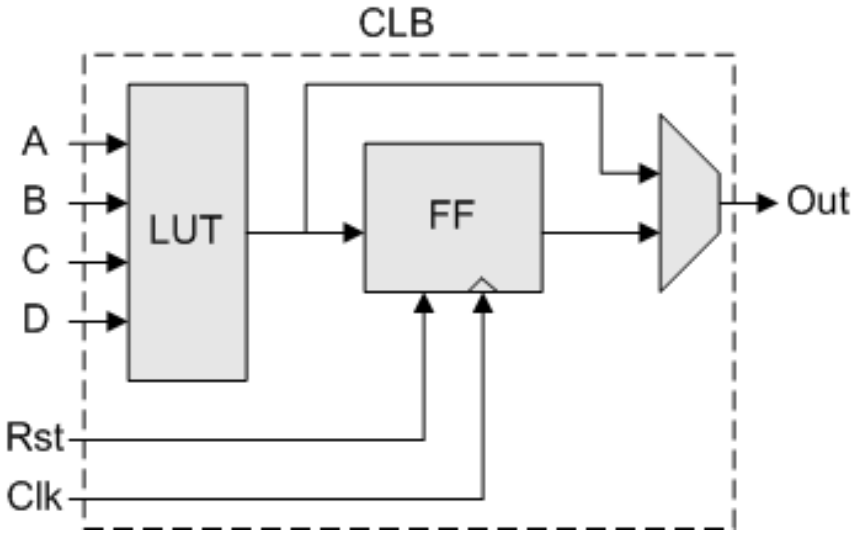
In this section, it will be presented a brief about the platforms being used to the development, more especifically FPGA and RISC-V architecture. After that, will be

presented the state-of-art for RSA implementation in FPGA, firstly showing the works that make use of Montgomery modular multiplication, and in sequence the ones that use other approaches.

3.2.1 FPGAs

Field-Programmable Gate Arrays are reconfigurable devices widely used in computational systems due to their flexibility, allowing a great portability, high throughput processing of data streams and relatively low development time and costs. The typical layout of modern FPGAs is an array of interconnected blocks, including interconnecting resources, clock-management resources, Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), and embedded blocks such as Digital Signal Processors, GPPs, high-speed IOBs, and memories (BRAMs). CLBs are used to perform simple combinational and sequential logic, typically consisting in of LUTs, multiplexers, flip-flops, and carry logic, being a fundamental building block of these reconfigurable devices. The CLB structure can vary depending on the FPGA model and manufacturer, but current industry-standard CLBs usually include multiple 6-LUTs. Figure 11 shows the internal organization of a very simple CLB.

Figure 11 – Basic CLB organization.

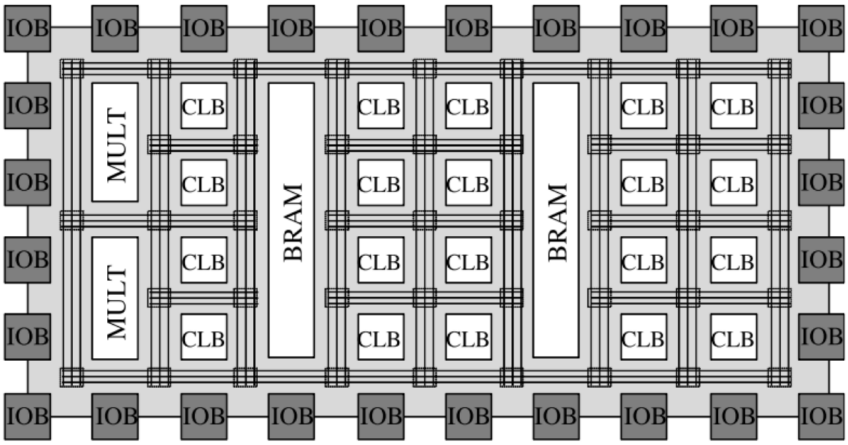


Source: The author

Aside from CLBs, the BRAM memory blocks are used to store larger amounts of data, dedicated multipliers to avoid excessive LUT usage and delay when implementing multiplications, and IOBs for connection with external signals. Programmable interconnect resources, such as routing switches, allow interconnecting all these elements, which are distributed over the FPGA area, along with interconnection switch-boxes, as shown in Figure 12. The logic and routing resources in an FPGA are configured by

the bits of a configuration memory, which may be based on anti-fuse, flash, or SRAM technology. More details about FPGAs architecture and usage can be found in (HAUCK; DEHON, 2008).

Figure 12 – FPGA architecture interconnection



Source: (NIEMIEC et al., 2020)

Taking into account all these characteristics, it can be concluded that FPGAs are are suitable platforms for implementing dedicated accelerators. Its possibility of reconfiguration during operation time guarantees great flexibility and versatility to the project, as it allows hardware changes. Also, the great amount of processing resources as the embedded adders, multipliers and fast carry chains (PARANDEH-AFSHAR; BRISK; IENNE, 2009) make it possible to implement a wide range of high-performance arithmetic operations at high clock frequencies, as cryptographic processing. Finally, the modern FPGAs count on counts on internal interfaces as AMBA bus (INTEL, 2023), which facilitates the implementation and integration of new peripherals with soft-processors.

3.2.2 RISC-V

The chosen architecture for the project implementation was the RISC-V, due to its facility of implementation and great support material. The RISC-V (WATERMAN et al., 2014) architecture was initially developed by Berkley University in 2010. It was created in order to offer a disruptive open-source alternative to proprietary architectures. For this reason, its main objective is to serve as a basis for companies, entities or any institution to develop not only compatible peripherals, but also their own cores, SoCs and accelerators.

Its design philosophy revolves around simplicity, modularity, and scalability. The architecture is based on a small set of well-defined instructions, facilitating easier implementation and verification. Its modular structure allows designers to select and

incorporate only the required components, resulting in efficient and tailored processor designs. The scalability of RISC-V enables its application across diverse computing platforms, from embedded systems to high-performance computing clusters.

3.2.3 Montgomery multiplication based implementations

After the popularization of the use of FPGAs, their use for implementing RSA accelerators and public key cryptography has also increased. Of these, the overwhelming majority are made up of architectures that use Montgomery's modular multiplication and exponentiation, due to its consolidation in the bibliography and versatility of implementation on FPGA platforms. An example of this are cores such as HARDRIOD (PICCOLBONI et al., 2021) and (NGUYEN-HOANG et al., 2022), which are generic cryptographic processor architectures, which have RSA accelerators within an SoC together with other schemes (such as ECC, AES...), and use the standard version of the Montgomery algorithm to perform modular exponentiation.

Despite this, the vast majority of architectures are specific to RSA, normally using some generalist version of Montgomery's modular multiplication, proposing small changes in the hardware implementation or in the algorithm itself in order to improve execution time, or even the number of logical elements used. In this sense, (HAN, J. et al., 2015), (VERMA; DUTTA; VIG, 2016), (REZAI; KESHAVARZI, 2015) and (MIYAMOTO et al., 2011) make use of CSA to eliminate the carry propagation. In (HAN, J. et al., 2015), a small compression tree is used to reduce the size of very long words before processing, thus reducing the number of iterations required. In the case of (VERMA; DUTTA; VIG, 2016) and (REZAI; KESHAVARZI, 2015), CSAs are used to perform the pre-computations necessary for the algorithm and avoid unnecessary sums every clock cycle, similar to (MIYAMOTO et al., 2011), which uses such structures to increase the speed and efficiency of its scalable radix-4 architecture.

Other strategies involve the use of different structures, such as (K et al., 2020), which makes use of LUT to store possible resulting values, thus reducing the required operation time. The same occurs for (KIM, D. W.; MAULANA; JUNG, 2022), where the operands and results are subjected to Barret reduction, without however using Montgomery multiplication. (THAMPI; JOSE, 2016) proposes some additional pre- and post-computation steps, in order to reduce the number of multiplication cycles and improve the total time. In the midst of this, there is also the format without final subtraction of the algorithm, implemented in (NADJIA; MOHAMED, A.; MOHAMED, I., 2012), which was designed specifically for the optimization of the area in hardware, costing two more iterations. Despite this, with one less subtractor, the critical path also decreases, making such an algorithm efficient. Finally, there are also cases like (SUZUKI; MATSUMOTO, 2011), where the FPGA signal processing structure was used to accelerate the implementation of the Montgomery algorithm, in order to use an existing block that has high

processing speed.

3.2.4 Implementations using unconventional algorithms

Despite the consolidation of the use of Montgomery's modular multiplication and its derivatives, there are also applications that make use of other approaches to perform modular multiplication. Among the alternative approaches, some have been applied to RNS, such as (ANTAO; SOUSA, 2014), (SOUSA; ANTAO; MARTINS, 2016) and (NOORDAM, 2019). The latter in particular is a RISC-V architecture for encryption and decryption of RSA-4096 bits for FPGA. In the implementation, several cores connected in a ring were used, in order to parallelize the operations, using an RNS-based modified version of Montgomery algorithm to perform the modular multiplications and exponentiation.

The project presented in (DING; LI, S., 2018) makes use of truncated multipliers with the addition of Barret reduction and Booth coding, with the aim of reducing the area and critical path in relation to the conventional truncated multiplier. A very similar strategy is used in (GROSSSCHÄDL, 2000), where partial product multipliers were used to perform modular multiplication during exponentiation, also together with Barret reduction and Booth coding. This recoding technique is still used in (CHO, K.-S.; RYU; CHO, J.-D., 2001), where an RSA processor is presented that uses the estimation technique presented in (HUNG, 1990) to calculate the product *AB mod M*.

3.3 IMPLEMENTATION CONSIDERATIONS

After analyzing the state of the art, it was realized that so far there are no specific applications for accelerating RSA encryption using efficient compression-based multipliers, as proposed in this work. Furthermore, among the large number of implementations using the Montgomery algorithm, the architecture presented in (NADJIA; MOHAMED, A.; MOHAMED, I., 2012) was chosen, as it is one of the most consolidated and also has great optimization for FPGA implementation. Therefore, the comparison with such an implementation will provide a good overview of the performance gains brought by the proposed implementation.

Furthermore, implementations using the RISC-V architecture in FPGAs are restricted to the works presented in (NGUYEN-HOANG et al., 2022), (NOORDAM, 2019) and (HAN, J. et al., 2015). All of these works makes use of Montgomery modular multiplier to perform the modular exponentiation, with or without modifications. All these approaches are significantly different from the proposal of this work: (NGUYEN-HOANG et al., 2022) presents the implementation of a cryptographic accelerator, which implements RSA, AES and SHA cryptography; (NOORDAM, 2019) presents an RNS version of Montgomery multiplication and modular exponentiation, communicating with various

RISC-V cores; (HAN, J. et al., 2015) presents a modified Montgomery multiplication algorithm implemented inside the processor pipeline, as a tightly-coupled accelerator. This fact, due to the lack of comparable information about area and performance, especially for (NGUYEN-HOANG et al., 2022), which does not focus on RSA only, it was difficult to make comparisons among them, opting to maintain as a state-of-the-art reference only the consolidated implementation of (NADJIA; MOHAMED, A.; MOHAMED, I., 2012).

4 PROJECT AND IMPLEMENTATION

The proposed methodology to design the accelerator consists primarily defining a strategy to take advantage of the modulos with high-performance format. After that, a modular exponentiation algorithm is chosen to be used, and a modular operator architecture is proposed to test it with three different modular multiplication approaches: direct multiplication using the VHDL modulo operator (**DM**), the compression-based multipliers generated by the tool from (FERNANDES, 2021) (**CM**), and the Montgomery modular multiplier (**MM**). The architectures will be designed in two versions: using only the original modulo, and using the proposed efficient modulo, which will demand a final correction block to convert the result back to the original modulo domain. Finally, the complete system is presented with all its components integrated: arithmetic unit, control unit and communication interface.

4.1 HIGH-PERFORMANCE MODULOS

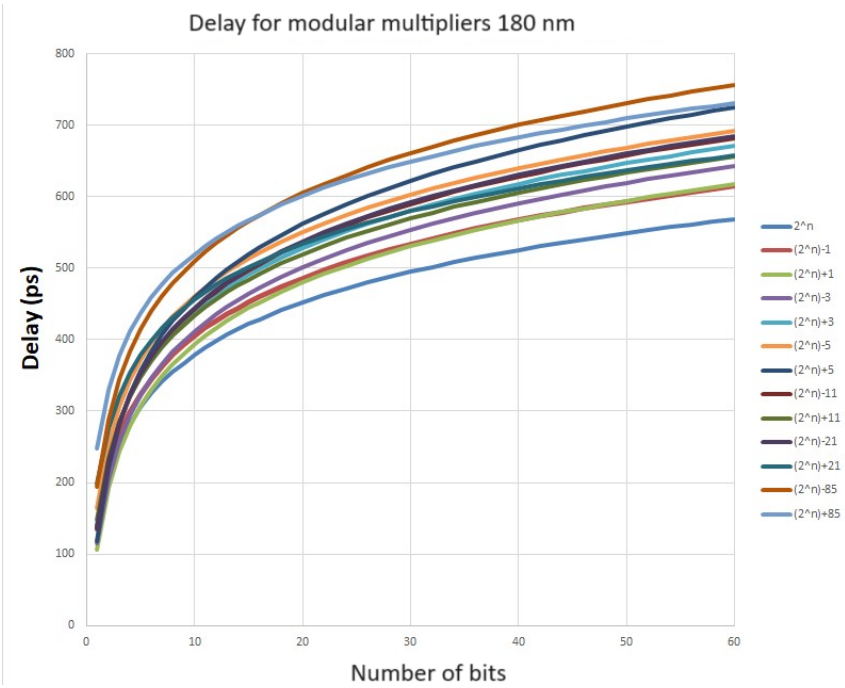
In hardware applications of modular arithmetic, the operation performance is strongly related to the modulo format, especially for multiplication and addition. In this way, modulos in format 2^n, $2^n - 1$, $2^n + 1$ are faster and more efficient than the ones in format $2^n \pm k$, where k is a positive integer, in many cases even when the former have a greater number of bits (PATRONIK; PIESTRAK, 2017). This becomes evident for the case of the compression-based architecture shown in Section 2.2.1, since the $2^n + 1$ and $2^n - 1$ multipliers require fewer compression levels than the $2^n \pm k$ cases, whose trees have even more layers for a high number of 1s in binary representation. To verify this, synthesis of modular multipliers generated by the tool were carried out for the set of modulos shown in Table 3, where they are presented together with the binary representation of their respective k's. The syntheses were performed using TSMC 180 nm technology, and the results for delay and area obtained are presented in Figures 13 and 14, respectively.

Table 3 – Modulos used for the synthesized multipliers

Modulo $2^n \pm k$	k **(Binary)**
2^n	$k = 0$
$2^n \pm 1$	$k = 1$
$2^n \pm 3$	$k = 11$
$2^n \pm 5$	$k = 101$
$2^n \pm 21$	$k = 10101$
$2^n \pm 85$	$k = 1010101$

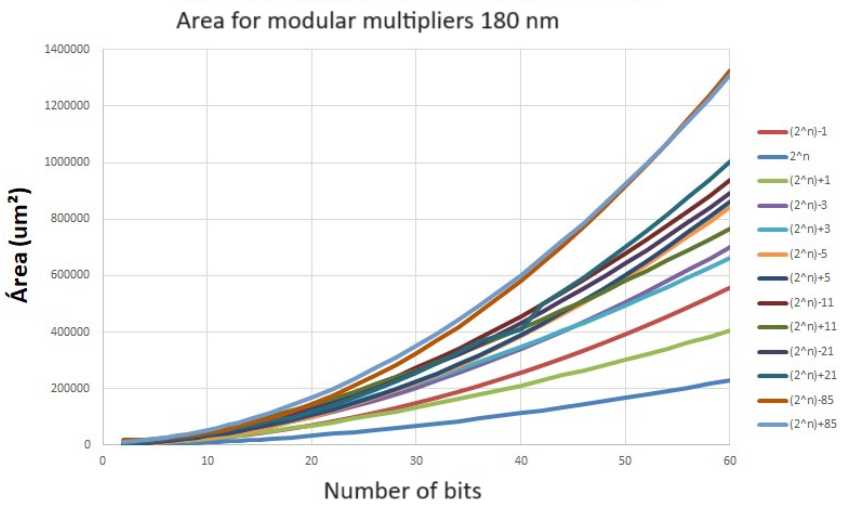
Source: The author

Figure 13 – Delay obtained for the synthesis of multipliers at 180 nm



Source: The author

Figure 14 – Area obtained for the synthesis of multipliers at 180 nm



Source: The author

Analyzing them, it is easy to see that the best result is 2^n, as it has the simplest compression structure among all cases. Excluding this, the best results for both *delay* and area were from the efficient modulos $2^n - 1$ and $2^n + 1$, respectively. Meanwhile, the circuits that presented the largest area and largest *delay* were those referring to the modulos $2^n + 85$ and $2^n - 85$, which have the k with the largest number of 1s, as

seen in Table 3. Looking at the intermediate results in these two cases, it is clear that the *delay* also increases along with the number of 1s of k, while the area decreases, demonstrating that there is really a gain in the efficient use of the modulos of the type $2^n \pm 1$ relative to $2^n \pm k$.

In fact, applications as cryptography usually require operations under the $2^n \pm k$ domain, since the other formats can be easily factorized, which makes its use as a cryptographic key unfeasible. Therefore, the use of $2^n \pm k$ modulos becomes inevitable, as they make operations slower and multipliers larger. However, the high-efficiency modulos can be obtained from those with lower performance, since these are multiples, using the well-known relation (HOLLMANN et al., 2018; DI CLAUDIO; ORLANDI; PIAZZA, 1990; PARHAMI, 1996):

$$||X|_{m_1}|_M = |X|_{m_2} \quad \forall \quad m_1 = c \times m_2, \tag{21}$$

where X is an arithmetic operation, m_2 is the original modulus, m_1 is the new efficient modulo, which here will be called pseudo-modulo, and c an integer constant. In this way, an operation can be performed in the domain of the efficient pseudo-modulo and the result can be converted back to the original inefficient modulo through a modulo reduction in the end. An example of this usage is depicted in Figure 15 for the modular multiplication $|A \times B|_{m_2}$. This strategy is especially useful for iterative operations like modular exponentiation, where the gain in execution time is accumulated every iteration, making a great difference in the total operation time at the end.

There are different moduli-sets that include modulos and their respective pseudo-modulos, where the pseudo-modulos are represented in $2^n \pm 1$ type, while the original modulos have a generic $2^n \pm k$ representation, usually varying the type of values for k. For use in this work, the chosen modulo and pseudo-modulo family present the format described by (22) and (23), respectively:

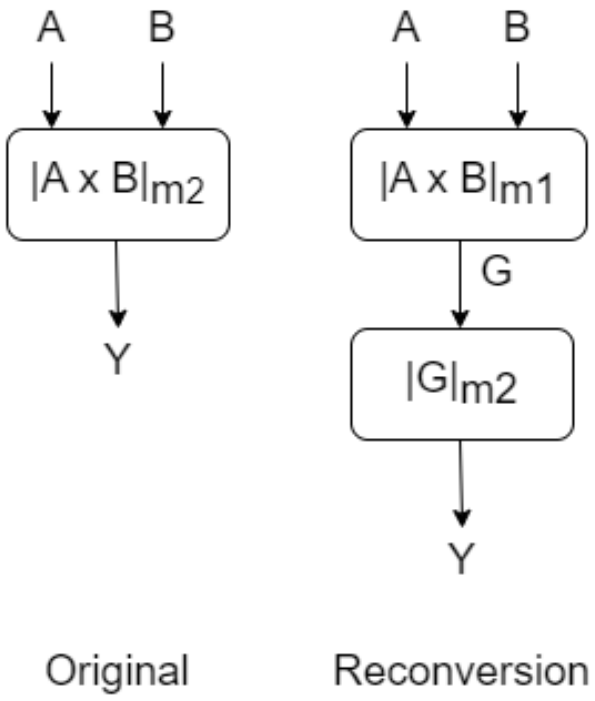
$$m_2 = 2^n + k, \quad \text{where} \quad k = \sum_{i=0}^{\frac{n}{2}} 2^{2i} \tag{22}$$

$$m_1 = 2^{n+2} - 1 = 3 \times m_2 \tag{23}$$

Here, n is the number of bits of the original modulo, and $b_{\frac{n}{2}}$ is the binary representation of k, which contains $n-1$ bits and presents $\frac{n}{2}$ ones, always interspersed with zeros. Some examples of modulos from $n = 2$ to $n = 8$ bits and its respective pseudo-modulos can be seen in Table 4.

Some modulos and pseudo-modulos from this family were tested on the same multipliers synthesized previously, with the results being presented in Table 5, where the multiplication of area and delay values are also presented. Analyzing them, it is clear that the pseudo-modulos present a gain in all cases, showing that there are benefits

Figure 15 – A and B modular multiplication on original modulo m_2 and on pseudo-modulo m_1 with reconversion



Source: The author

Table 4 – Correlation between modulos and pseudo-modulos with scalable *k*

Modulo $\mathbf{m_2 = 2^n + K}$	*k* **(Binary)**	**Pseudo-modulo** $\mathbf{m_1 = 2^{n+2} - 1}$
$5 = 2^2 + 1$	k = 1	$15 = 2^4 - 1$
$21 = 2^4 + 5$	k = 101	$63 = 2^6 - 1$
$85 = 2^6 + 21$	k = 10101	$255 = 2^8 - 1$
$341 = 2^8 + 85$	k = 1010101	$1023 = 2^{10} - 1$

Source: The author

to their use. Even in cases where the difference for a single operation is not extremely high, this time savings accumulates with each iteration, becoming significant at the end of the process.

Table 5 – Delay and area of modulos and pseudo-modulos in the presence of final adders for multipliers at 180 nm

Modulos	**Delay**	**Area**	**Area×Delay**
$2^4 + 5$	407 ps	15965 um	6.417×10^6
$\mathbf{2^6 - 1}$	**393 ps**	**15792 um**	$\mathbf{6.206 \times 10^6}$
$2^6 + 21$	463 ps	24000 um	11.112×10^6
$\mathbf{2^8 - 1}$	**457 ps**	**22000 um**	$\mathbf{10.054 \times 10^6}$
$2^8 + 85$	509 ps	32951 um	16.772×10^6
$\mathbf{2^{10} - 1}$	**480 ps**	**24458 um**	$\mathbf{11.739 \times 10^6}$
$2^{10} + 341$	525 ps	38500 um	20.212×10^6
$\mathbf{2^{12} - 1}$	**501 ps**	**32000 um**	$\mathbf{16.032 \times 10^6}$

Source: The author

4.2 PROPOSED MODULAR EXPONENTIATION OPERATOR

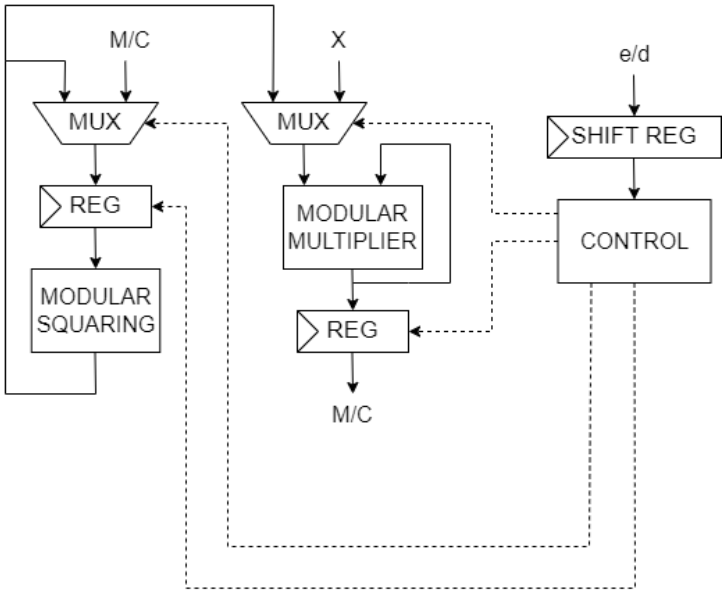
The architecture for modular exponentiation using the three different modular multiplications approaches are presented in this section. The CM and DM multipliers are implemented using the same circuit, since both multiplications are performed in one clock cycle, while the MM approach need more cycles to complete one multiplication, demanding a different architecture. In both cases, the chosen algorithm for modular exponentiation was the right-to-left binary method, which allows greater parallelization of the operation, as it has two registers, thus enabling greater optimizations in relation to the use of left-to-right, which has only one register.

4.2.1 Modular exponentiation for compression-based modular multipliers and direct modular multipliers

The proposed architecture for the modular exponentiation using CM and DM is shown in Figure 16. Here, the inputs for the basis can be the encrypted message *C* or the the decrypted message *M*, respectively described in Equations (1) and (2). For the exponent, the input is the chosen exponent *e* for encryption and the private key *d* for a decryption. Consequently, the output should be *C* when *M* is encrypted and *M* when *C* is decrypted.

As can be seen, the exponent need to be scanned bit by bit, which is done in the shift register, with the modular squaring being performed in every cycle, while the multiplication occurs only when the scanned exponent bit is 1. For the architecture of Figure 16, the modular multiplication block will be implemented in two different ways: using the compression-based multipliers generated by the software tool, and the regular multiplication in VHDL, in conjunction with the language's native modulo function. This last approach will also be tested in different ways: firstly performing the multiplication

Figure 16 – General architecture for the proposed modular exponentiation operator



Source: The author

of two variables A and B, and the modulo m reduction R in sequence, as shown in Equations (24) and (25):

$$C = A \times B \tag{24}$$

$$R = |C|_m \tag{25}$$

The second form is to perform the multiplication directly inside the modulo function argument, as depicted in Equation (26):

$$R = |A \times B|_m \tag{26}$$

The objective in doing so is to verify if the synthesis tool is capable of differentiating both operations, presenting some optimization for modular multiplication, using any dedicated structure or algorithm.

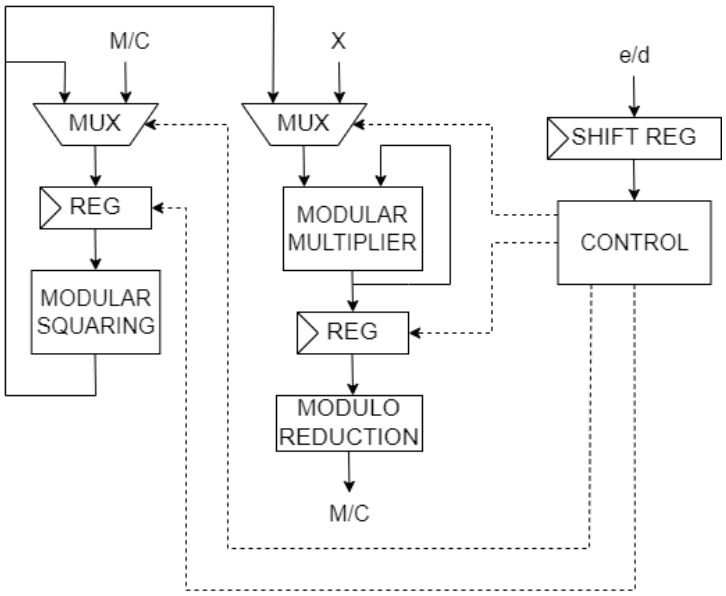
This implementations will also be applied to the modular squaring block, with the difference that it will be attempted to generate a compression-based multiplier optimized for squaring using the software tool, which should theoretically be faster and less expensive than a conventional two-variable multiplier (PARHAMI, 1996).

Modular exponentiation for compression-based modular multipliers and direct modular multipliers with correction:

The usage of pseudo-modulos in iterative operations consists in the transformation of the operands from the original modulo to the efficient pseudo-modulo domain, performing all the arithmetic calculation in it, and finally converting the resulting residue back to the original modulo format by using the modulo reduction. The greater the

number of iterations, the greater the total gain in execution time tends to be, which greatly benefits operations with several recursion steps, such as modular exponentiation, especially for exponents with large numbers of bits. In this way, to take advantage of the pseudo-modulos, the architecture from Figure 16 need to be modified as shown in Figure 17, adding a new block to perform the final correction after the operation is complete, converting the result back to the original modulo domain.

Figure 17 – General architecture for the proposed modular exponentiation operator



Source: The author

For this particular application, no initial conversion is needed, since all the iterations of the exponentiation can be performed in the pseudo-modulo format, in this case $2^n - 1$, only in the end converting the resulting value to the correct format through the modulo reduction by the original modulo $2^n + k$. This final correction demands an extra clock cycle, but since the gain in delay due to the use of the pseudo-modulo occurs in each cycle, it becomes cumulative. Thus, for a high number of bits, as is the case of the RSA-512 and RSA-1024 systems, which consequently have many iterations, the gain in total operating time is high in relation to the same process using only the original modulo, so that the delay generated by the extra cycle for correction now has an irrelevant contribution.

Since DM and CM approaches can be performed in a single clock cycle, in a first moment, the use of pseudo-modulos makes much more sense in this scenario, since the use of modulos which allows faster hardware is capable of reducing the critical path, thus increasing the maximum clock value and and decreasing the total operation time.

4.3 MONTGOMERY MODULAR EXPONENTIATION OPERATOR

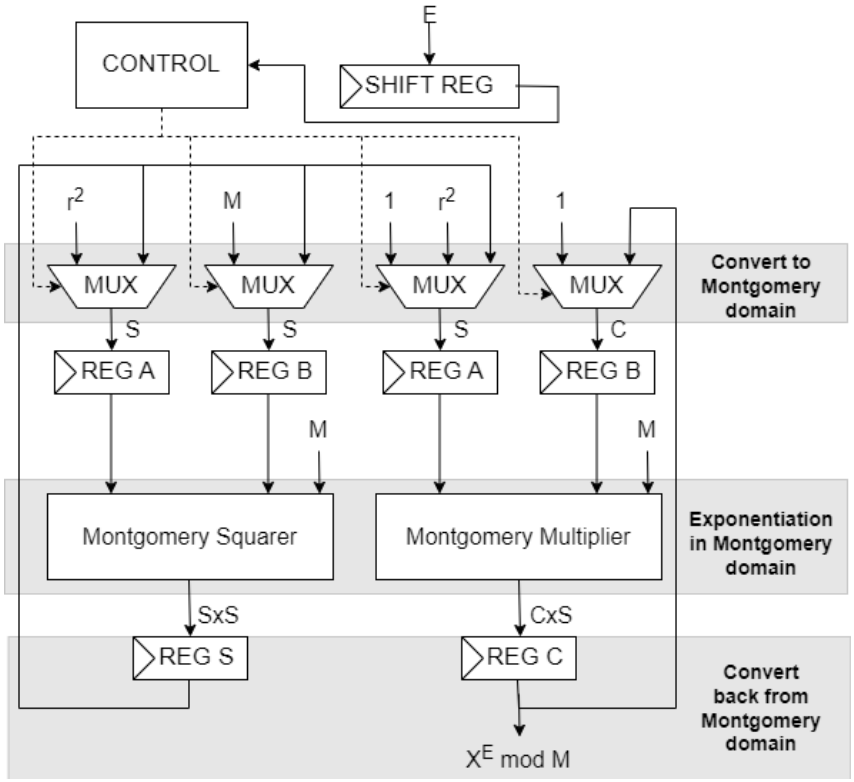
For the present work, the chosen Montgomery algorithm was the modular multiplication without final subtraction, which is described in Algorithm 2 (PU; ZHAO, 2009), since it presents a lower execution time when compared to the regular Algorithm 2. The hardware implementation for the multiplier can be seen in Figure 7.

The right-to-left binary modular exponentiation method adapted for the Montgomery modular multiplication application is presented in Algorithm 6, and the architecture developed to implement it is shown in Figure 18, which is based on the work presented in (NADJIA; MOHAMED, A.; MOHAMED, I., 2012). It consist of:

- One control block to generate the control signals and manage all the operation steps, analyzing the exponent bits and controlling square and multiply operations.
- A shift-register to shift the exponent bits.
- Two Montgomery modular multiplication blocks, one to perform the square operation and the other to the multiplication. Both of them are implemented by the circuit shown in Figure 7, since, unlike what occurs in Section 4.2.1, for the Montgomery algorithm there is no differentiation between the two operations. They are connected to S and C register to hold the intermediate values.
- Two registers A and B at the input of each multiplication block to store the inputs data (M, r^2) or the intermediate results.
- Two multiplexers at the input of each multiplication block to select the inputs data (M, r^2, 1) or the intermediate results.

In order to follow the Algorithm 6, firstly the multiplexer select signals are set to choose the inputs to perform the first parallel modular multiplications $C = MontMult(1,r^2,M)$ and $S = MontMult(X,r^2,M)$ using the Montgomery Multiplier and Montgomery Squarer respectively, giving the outputs $C = 1 \times r(mod \quad M)$ and $S = X \times r(mod \quad M)$, in order to enter into the Montgomery domain. In the Montgomery domain, multiplications and squares are performed in parallel by two Montgomery multipliers and successively until all bits of the exponent are shifted in the shift register. To do so, the multiplexers select the inputs ($X \times r \quad mod \quad M$) and ($1 \times r \quad mod \quad M$) or the outputs of the multipliers according to the logic control to give the result ($X^E \times r \quad mod \quad M$). Exiting from the Montgomery domain means performing a last Montgomery multiplication in order to eliminate the r factor from the result to finally obtain ($X^E \quad mod \quad M$). The multiplication blocks have, in addition to the two operator inputs, a third parameter referring to the modulo M. Since the multiplication is performed sequentially, one single step of the exponentiation takes the same number of clock cycles as the number of bits of the operands.

Figure 18 – Architecture of the Montgomery right-to-left binary exponentiation



Source: The author

Montgomery modular exponentiation operator with correction:

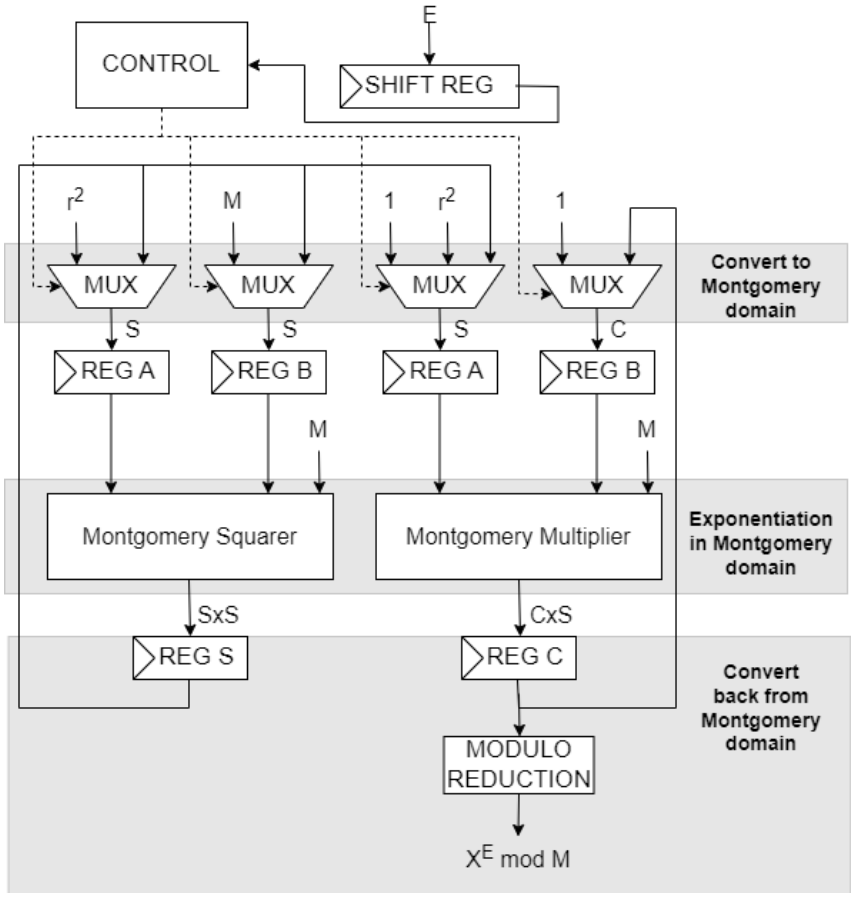
This architecture will also be tested using the pseudo-modulo strategy (PARHAMI, 1996), thus needing a new version with the final correction block to convert the result back to the original modulo domain. This new version is presented in Figure 19.

4.4 PADDING

In the context of cryptography, padding is the process to add extra data to the plaintext target message, before the encryption or after the decryption. For the RSA, the padding usage is focused in the improvement of security-related issues, as follows:

- **Prevent Deterministic Encryption:** Without padding, if the same plaintext is encrypted multiple times with the same key, the resulting ciphertext would be the same. This determinism can be exploited by attackers, leading to potential security vulnerabilities. Padding introduces randomness to the plaintext, making each encryption operation unique (BLEICHENBACHER, 2002).
- **Security Against Chosen Plaintext Attacks:** Padding helps in protecting against chosen plaintext attacks, where an attacker can carefully choose plaintexts to be encrypted and analyze the corresponding ciphertexts. Padding

Figure 19 – Architecture of the Montgomery right-to-left binary exponentiation with final correction



Source: The author

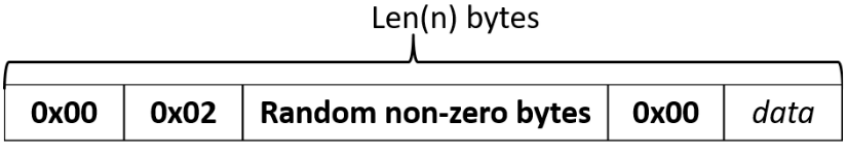
schemes add complexity and unpredictability to the encryption process, making it more resistant to such attacks (KATZ; LINDELL, 2007).

- **Handling Short Messages:** RSA encryption has a limitation in terms of the maximum length of the plaintext that can be encrypted. If the plaintext is too long, it may exceed the size of the RSA modulus, leading to practical difficulties. Padding ensures that even short messages reach the required length, and it helps in avoiding issues related to length limitations (BONEH et al., 2007).
- **Ensuring Consistent Message Format:** Padding ensures that all plaintext messages, regardless of their length, have a consistent format before encryption. This homogeneity simplifies the decryption process, as the recipient can reliably identify and remove the padding to obtain the original message (LABORATORIES, 2004).

Taking this into consideration, the need to use padding to obtain a safe system

becomes clear. For this reason, the guidelines for padding in RSA are specified in the Public-Key Cryptography Standards (PKCS), published by the RSA Laboratories (LABORATORIES, 2004). Among all the padding patterns contained in this family of standards, the one chosen to be used in this work was the PKCS#1 v1.5. It was selected because it is one of the most used schemes, in addition to being relatively easy and quick to implement. Its implementation follows the sequence shown in Figure 20, consisting of a leading byte (0x00) followed by a block type identifier ((0x02) for the most cases), additional random non-zero padding bytes, a final (0x00) byte to indicate the ending of the padding, and finally the data to be encrypted, the message effectively.

Figure 20 – PKCS#1 v1.5 padding structure



Source: The author

For PKCS#1 v1.5 to be effective, there must be at least 8 non-zero random padding bytes, which summed to the two (0x00) bytes and the identifier (0x02), leads to 11 bytes dedicated only to the scheme. Since they need to be added before the encryption and removed after decryption, this inevitably limits the size of the message that can be sent, which is given by Equation (27):

$$\textit{Maximum Message Size} = \frac{\textit{Key Size}}{8} - 11 \tag{27}$$

It can be seen that, along with the padding bytes, the message size is also limited to the key length, due to the modular exponentiation. In the same way, the number of random bits can also be calculated, as in Equation (28):

$$\textit{Padding Length} = \frac{\textit{Key Size}}{8} - \textit{Plaintext Length} - 3 \tag{28}$$

The minimum padding length to be obtained here is 8, in a way that Equations (27) and (28) are coherent. Evidently, for RSA keys with 32 or 64 bit long, the messages can not be padded, since their maximum length do not allow extra space for the necessary bytes.

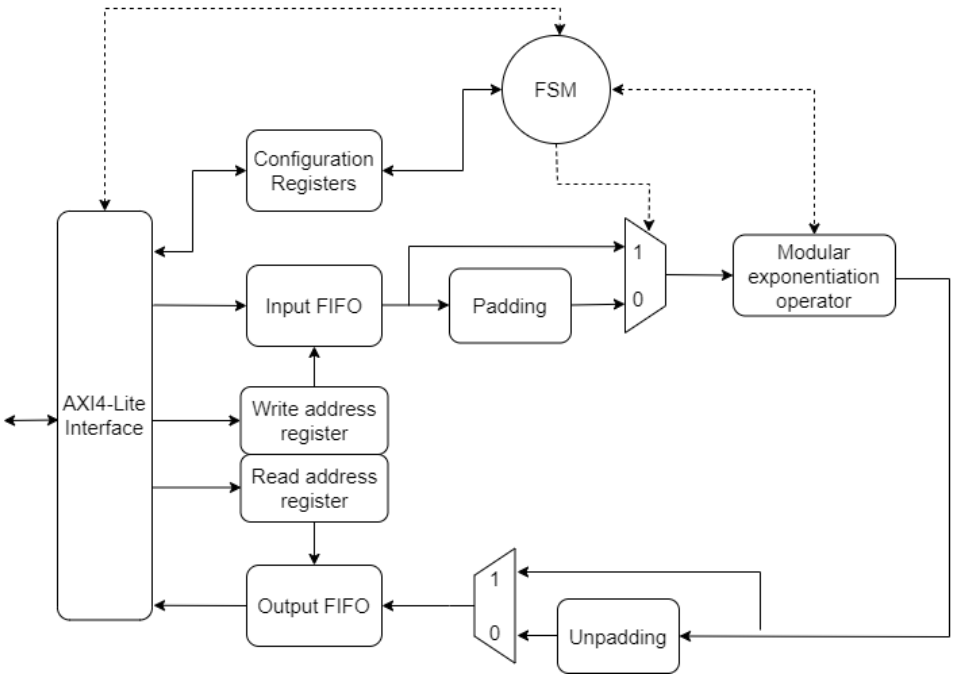
4.5 MICROARCHITECTURE

Taking into account the discussion presented in Section 2.4, about the types of accelerators, it was decided that the loosely-coupled model is the most suitable for carrying out the project. The main motivation for this choice is the portability and flexibility

of the design, since it will not be attached to one single processor or architecture. Also, the integration of the cryptography unit in the pipeline may lead to complex changes to the CPU control unit, even demanding to the core to stop the processing until encryption or decryption is complete. In this way, the usage of an external bus to interconnect the cryptography unit with the main core simplifies the design, allows the accelerator and the CPU to run in parallel and makes it possible for other peripherals to be more easily connected in the System-on-a-Chip (SoC), avoiding possible synchronization problems that could occur due to the control complexity of tightly-coupled architectures.

The accelerator was developed in VHDL language, and its architecture is shown in Figure 21. It has an AXI4-Lite interface for communicating with the nucleus, and is controlled by an FSM. A more detailed description of each block is provided in the following sections.

Figure 21 – Block diagram for the accelerator intern architecture



Source: The author

4.5.1 AXI4-Lite interface and configuration registers

The AXI interface is a standard SoC interface for communication between the core and peripherals or memory. The AXI4-Lite is a simplified implementation of this protocol, lighter and easier in terms of implementation, ideal to read and write operation in the registers of narrow band peripherals, as in this case. Since this is a MMIO, the read and write in the accelerator registers are performed by the reading and writing

of data in the memory space allocated for these registers (PATTERSON; HENNESSY, 2017).

The memory addresses available for each register are presented in Table 6. The "address" column corresponds to the base address offset, which is assigned by the data bus mapping. For example, in a system where the memory space "0x0004_0000" was assigned to this accelerator, to write to the MESSAGE register it is necessary to write to the memory space "0x0004_0008". A better description of every register is provided in sequence.

Table 6 – Registers addresses for the AXI4-Lite interface

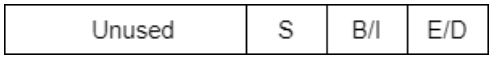
Name	**Address**	**Read(R) Write (W)**
STATUS	0x00	R/W
INPUT_LENGTH	0x04	W
OUTPUT_LENGTH	0x08	R
WRITE_ADDRESS	0x0C	W
READ_ADDRESS	0x10	W
INPUT_FIFO	0x14	W
OUTPUT_FIFO	0x18	R

Source: The author

4.5.1.1 STATUS register

The status register informs and also configure the current status of the accelerator, using two bits, as shown in Figure 22. Here, the bit 0, called E/D, which comes from encryption/decryption, is the write-only flag for the user to configure the operation to be performed by the accelerator, being set to 0 to perform an encryption, and set to 1 to decrypt a message.

Figure 22 – Status register bits



Source: The author

On the other hand, the bit 1, called by B/I, from busy/idle, is read-only and intends to inform the core if the accelerator is processing an encryption or decryption (B/I = 1), or if it is idle (B/I = 0).

The bit 2, tagged with S, is another write-only bit. When it is 1, the cryptography starts using the message and parameters available at the accelerator input. When it is

0, no processing is performed by the peripheral. The remaining bits will not be used in the first version of the system, but will be available for future extensions.

4.5.1.2 INPUT_LENGTH register

The INPUT_LENGTH register is used to inform the length of the message to be processed by the accelerator. The maximum length allowed is limited by the key size being used and by the padding, as described in Equation (27). For example, when using a 1024 bit key, the maximum message size is 117 bytes, which will also be the maximum value allowed to be written into the length register. If a value greater than this is written to the register, only the maximum value will be considered, in this case 117.

4.5.1.3 OUTPUT_LENGTH register

The OUTPUT_LENGTH register is used to inform to the core the size in bytes of the message available at the output, in order to know how many iterations it will be needed to read the entire message. The value will be available at the end of the processing. If the encryption/decryption is not finished yet, the value will be zero.

4.5.2 Input and output FIFOs

The input and output FIFO are register banks used to store the input message before the cryptography process, and the output after it is completed, respectively. To write into the input FIFO, the data need to be written to the INPUT_FIFO register, and then the address in which it will be written need to be put in the register WRITE_ADDRESS. As the structure is made of registers, every address refers to 32 bits words (4 bytes). Also, the data is stored from bottom to top, so a 64 bit message, for example, will have it least significant part stored in the address 0, while the most significant one will be in address 1. For this reason, the main core need to write the message to the input sequentially, addressing the least significant 4 bytes to the address 0.

The same process is required to reading the output data, firstly writing the address 0 in the READ_ADDRESS and reading it from OUTPUT_FIFO, increasing the address value after that, since the data will also be stored from the bottom to top.

The depth of the FIFOs will depend on the key size being used. Again using the example of RSA-1024, it was seen that the maximum message length considering the padding was 117 bytes. In this way, dividing this value for the 4 bytes respective to every register, the result will be 29,25. Thus, it turns out that the maximum FIFO depth for 1024 bit keys will 30 registers, meaning that they will have 30 addresses each.

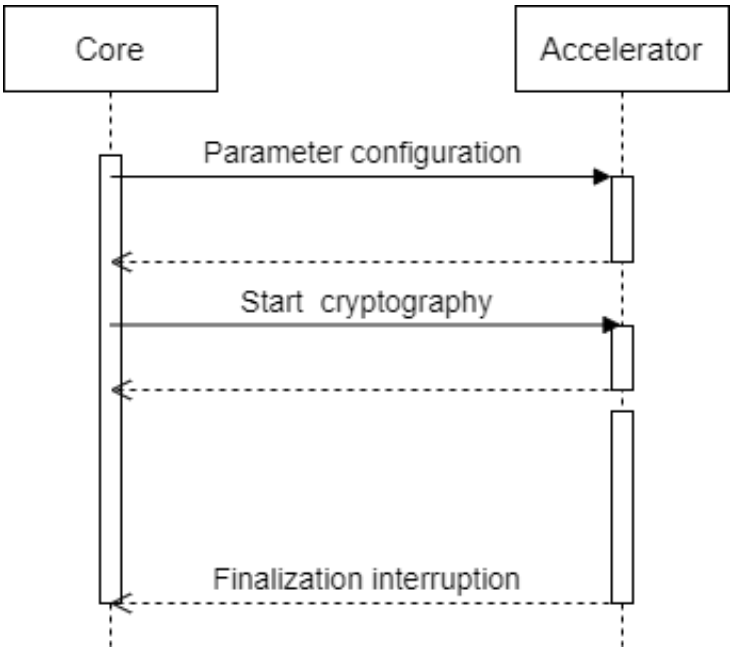
4.5.3 Padding and unpadding

The padding and unpadding blocks are the responsible for adding or removing the padding using the technique described in section 4.4. The padding is added for encryptions when the key is greater than 64 bits, as seen in section 4.4, and is bypassed during the decryptions. In the same way, when a decryption is performed, the padding is removed by the unpadding block, in order to only deliver the message, and this one is bypassed during the encryption.

4.5.4 Accelerator operation

Figure 23 shows the sequence diagram for using the accelerator. The parameters configuration stage refers to setting the E/D bit from STATUS register according to the desired operation, verifying the B/I bit to ensure that the accelerator is available, and finally to storing the message into the input FIFO. After that, the cryptography is started by setting the start bit S from STATUS register, and the operation ending is signalized by an interruption. After that, the data is available for reading in the output FIFO.

Figure 23 – Status register bits



Source: The author

5 TESTS AND RESULTS

The designed accelerator described in Chapter 4 was synthesized using the software Quartus II 2021 and implemented in the FPGA Cyclone V GX 5CGXFC5C6F27C7N. The tests were divided in different steps. Firstly, the performance of the arithmetic units that make up the exponentiation operators in Sections 4.2 and 4.3 are evaluated through Static Timing Analysis (STA), which allows obtaining their critical path. After that, the same is done for both complete operators, which allows checking their total operation delay. Such tests are carried out for each case: using VHDL's native modulo operator (**DM**), dedicated units based on compression (**CM**) and Montgomery multipliers (**MM**), in order to compare their results and perform possible optimizations. Finally, the operators are connected to the microprocessor, evaluating its operation time and clock restrictions.

5.1 MODULO REDUCTION SYNTHESIS

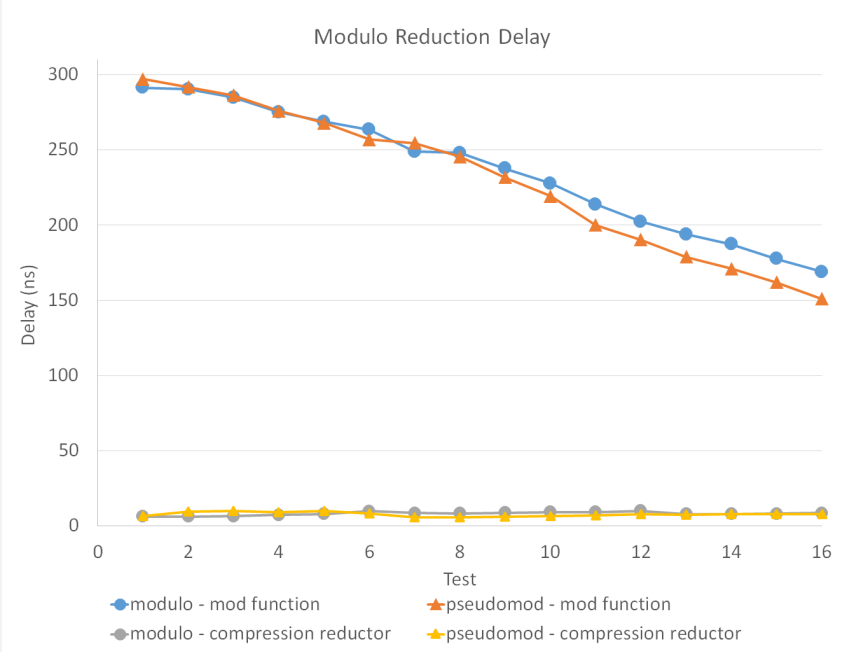
The modulo operation calculates the remainder after division of one number by another. In the VHDL language, it can be implemented by the function *mod* (Equation (29)) where A is the integer to be reduced, M is the modulo and R is the residue.

$$R <= |A|_M \tag{29}$$

In order to investigate the behaviour of this operator in the FPGA synthesis, it was tested using the moduli-set presented in Section 4.1, which contains modulos of $2^n + k$ type and its pseudo-modulos of $2^{n+2} - 1$ format. The same was done using the modulo operator generated by the software tool from (FERNANDES, 2021), for comparison. The obtained operation delay is shown in Figure 24. The x-axis of the graphic presents each set of modulo and it respective pseudo-modulo as "Tests". For example, following Equation (23) and Table 4, Test 2 corresponds to the values of the original modulo 2^2+1 and its pseudo-modulo 2^4-1, Test 4 holds the values of modulo 2^4+5 and the pseudo-modulo 2^6-1, and so on. As the *mod* function is limited to 32 bits inputs and outputs, the range of tests is 16, presenting as maximum values the modulo $2^{30} + 357913941$ and pseudo-modulo $2^{32}-1$.

Both *mod* function and compression-based modulo operators were tested using a fixed-length input of 32 bits. As can be seen, using the *mod* function, the delay for the modulos and pseudo-modulos are the same until 16 bits, but from 16 to 32 bits the pseudo-modulos are slightly faster. Also, it is easy to notice that the dedicated modulo operators have a considerably smaller delay than the presented by the VHDL function, and their behaviour are zoomed in Figure 25. By analyzing it, it is noticeable that for values above 12 bits (Test 6) the delay for the pseudo-modulo is smaller than for the

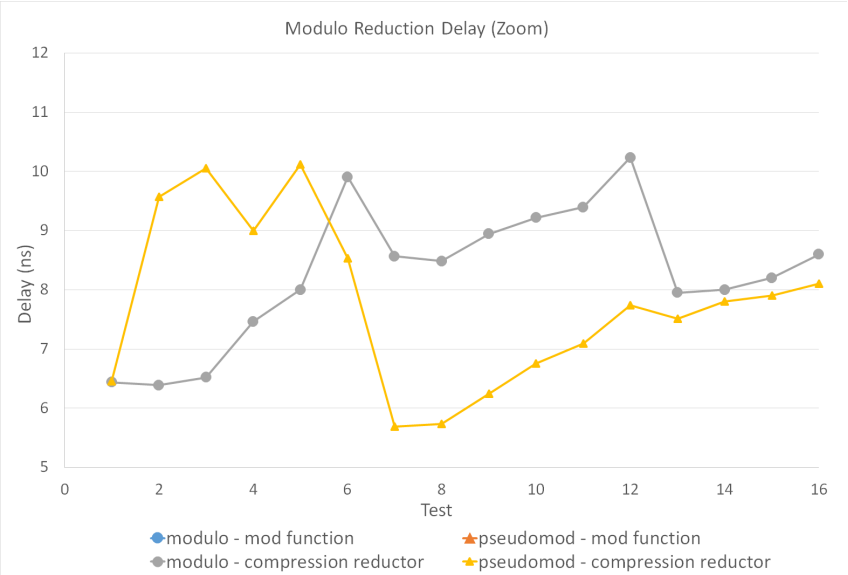
Figure 24 – Operation delay for modulo reduction of 32 bit input



Source: The author

modulo conversion. The difference is not large since it is a simple operation, but it means that the performance for the efficient format modulos is better, as expected.

Figure 25 – Zoom in operation delay for modulo reduction of 32 bit input



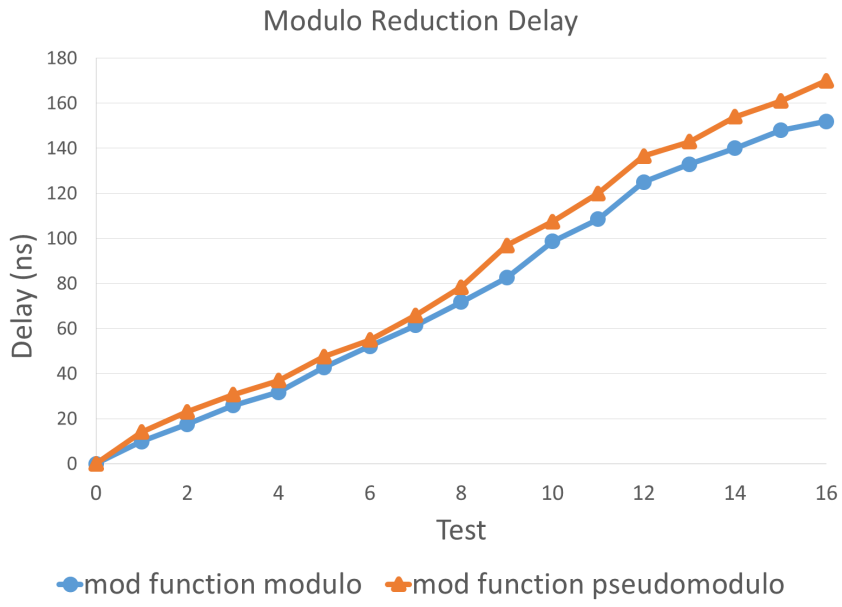
Source: The author

Despite the difference between the delay results of modulos and pseudo-modulos

for the *mod* function seems satisfactory in Figure 24, the curves present a peculiar progression, decreasing with the number of bits increase. It may indicate the use of Lookup Tables (LUTs) to calculate the results, as the shift, lookup and addition method proposed in (KAWAMURA; HIRANO, 1988). For this reason, as the modulo length grows, the number of possible residues for a fixed-length 32 bits variable reduces, causing the LUTs to be smaller, presenting smaller access times, which together with the extra arithmetic operations, results in the total time seen.

With the objective to test all the possible cases for the *mod* function, it was also evaluated using modulated inputs, i.e., inputs with $(n+1)$-bits, where n is the number of bit of the modulo. This is done to take into account the delay in processing values that have already undergone a previous modulation process, which is very common in iterative operations, especially modular exponentiation. The obtained results can be seen in Figure 26. It can be noted that the curves vary in a different way, growing together with the number of bits in the modulo and input. It happens because, as the number of bits of the input is increasing, the number of possible results also increases, making them execution time increase too, once the number of LUTs and its access time is growing. In this way, the pseudo-modulo of Test 16 $2^{32} - 1$ has the same value in Figures 24 and 26, since they have the same bit length, indicating that the delay of the operation is limited by the case of 32 bits input.

Figure 26 – Zoom in operation delay for modulo reduction of (n+1)-bit input



Source: The author

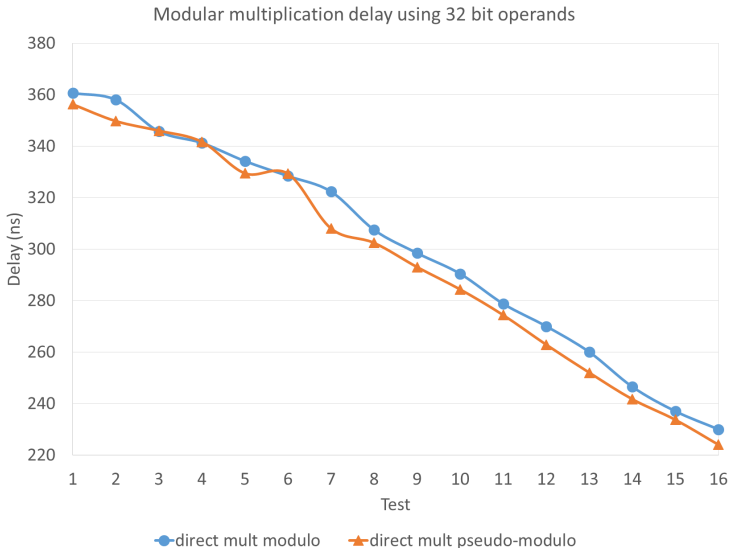
The compression-based operators were not tested for $(n+1)$-bit input because will not be used in any part of the design. However, by the results shown in Figure 24, it can be seen that they are much more optimized and faster performing the modulo of a

variable. Furthermore, although the delay for the pseudo-modulos is smaller than that of the modulos for the mod function in Figure 24, the exact opposite occurs in Figure 26, showing that the VHDL native operator synthesis tool is not sensitive to the type of used modulo, not performing the operation efficiently.

5.2 MODULAR MULTIPLICATION SYNTHESIS

The arithmetic structures to be used for the modular multiplication in each case are: the dedicated multipliers generate by the used software tool and described in Section 2.2.1, the architecture from Figure 7 for the Montgomery modular multiplication, and the implementation described in Equations (24), (25) and (26) for the *mod* function usage. Due to the results from Figure 24, it was initially evaluated the case for modular multiplication using the native function for a 32 bit input; their results are shown in Figure 27. As for the modulo operation, the modular multiplication presents the same descendent behaviour with the increase in the modulo bit length. It was expected, since the operation for this case is synthesized as a multiplier in series with the modulo operator, thus being limited by the *mod* variation, along with a delay overhead caused by the multiplication operation. With this, it was also possible to verify that there is no difference between the implementations of Equations (24) and (25) to the one of Equation (26), since both are synthesized in the same way, generating the exact same delay. It means that the synthesis tool is not capable to generate an efficient modular multiplier itself, demanding the usage of dedicated implementations to obtain satisfactory results.

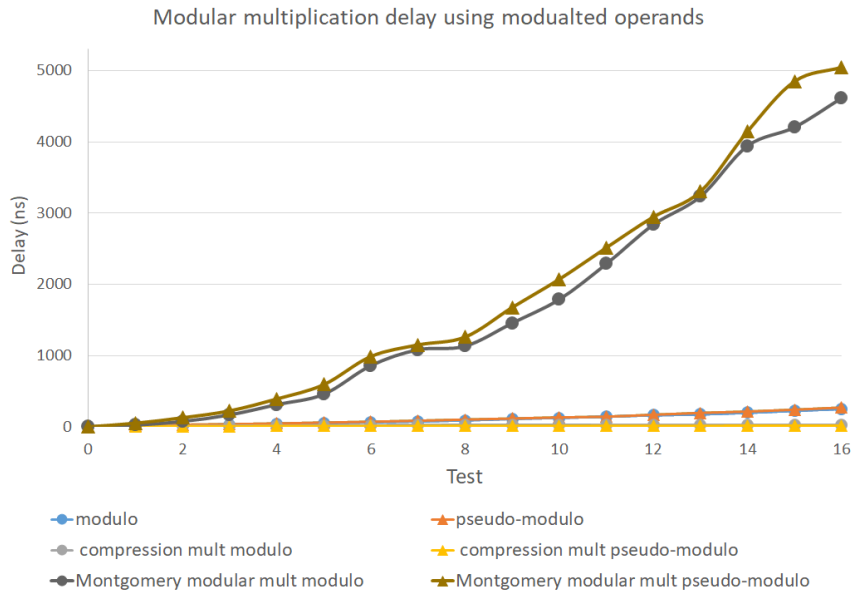
Figure 27 – Modular multiplication delay for 32 bit input using mod function



Source: The author

Despite that, in an application as modular exponentiation, where the first iteration has an 32 bit input, it can be performed a modulo reduction operation in the first iteration, in order to perform all the multiplications with modulated inputs. By doing this, the delay for operation will be as presented in Figure 28, where the *mod* function modular multiplication have $(n + 1)$-length input, being compared with the delay of the two other implementations. According to the obtained curves, the Montgomery multiplication presents a total delay much larger than the other two cases. In addition to that, the usage of pseudo-modulos does not bring any efficiency, presenting results even worse than those of the original modulos. As this multiplication is an iterative process, it processes bit by bit from the operand, and since the pseudo-modulos have a larger bit length than the original modulos, it will take more steps to complete the operation, naturally taking longer to do so.

Figure 28 – Comparison of operation delay between mod function, compression-based modular multipliers and Montgomery modular multipliers



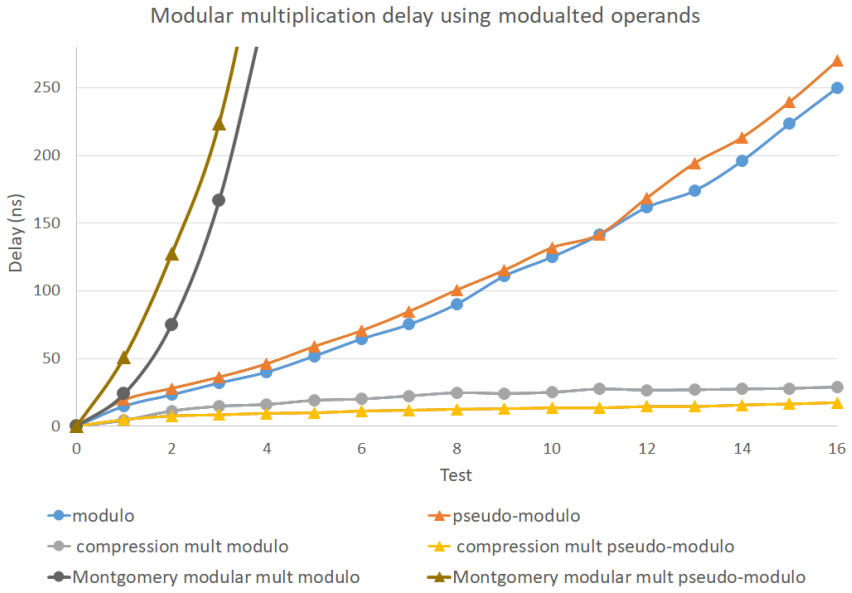
Source: The author

It is important to point that, due to the operation being performed in more than one step, the Montgomery modular multiplication needs many clock cycles to obtain the result, unlike the two other approaches, which are completed in a single cycle.

To evaluate the delay for the dedicated multipliers generated by the tool and the multiplication using *mod* function with $(n + 1)$-bit input, the graph from Figure 28 was zoomed in Figure 29. It can be observed that, for modulated inputs, the modular multiplication using the native VHDL operator is slower for pseudo-modulos than it is using the original modulos, again indicating that there is no sensitivity to the modulo type, for the same reason observed in Figure 26. The compression-based multipliers,

on the other hand, obtained a much higher performance than the two other approaches, with the pseudo-modulo multipliers being faster than the original modulo ones, showing that its usage is advantageous.

Figure 29 – Zoomed comparison of operation delay between mod function, compression-based modular multipliers and Montgomery modular multipliers



Source: The author

The squaring operation was also tested for the compression-based multipliers and the VHDL modulo. For the first one, new structures for the multiplication of two equal variables were generated using the software tool, however, the results obtained for execution time were the same from Figure 29. It shows that the tool is not capable to perform the necessary optimization for this kind of operation, treating it as a generic multiplication of two variables. In the case of *mod* function the same happened, obtaining equal results to those in Figure 28. This indicates that the synthesis of Quartus II do not optimize the multiplication when it is a squaring, making use of the generic multiplication hardware in FPGA. For the Montgomery algorithm, there is no difference between the multiplication and squaring operations, since both need to use the same hardware structure.

5.3 MODULAR EXPONENTIATION SYNTHESIS

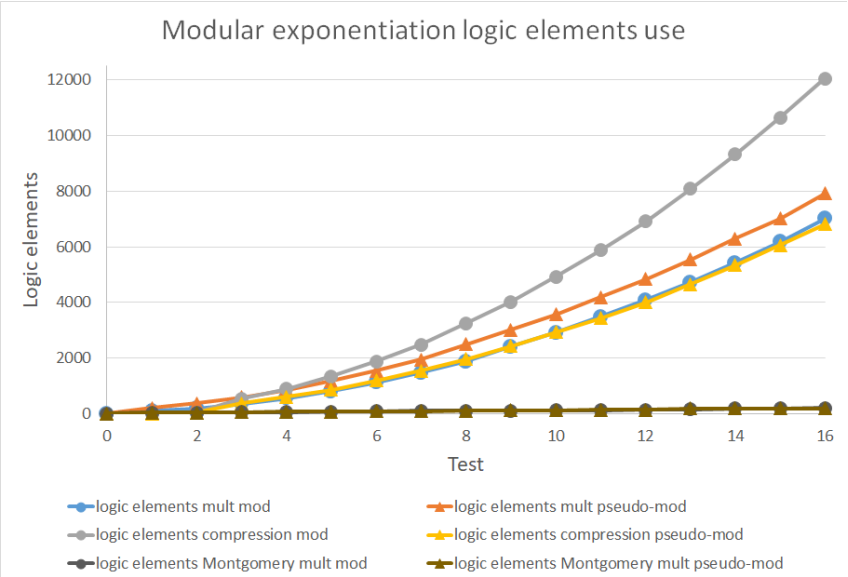
Finally, the complete modular exponentiation operators were evaluated, with all its blocks integrated, following the architecture of Figure 17 for the *mod* function and compression-based multipliers, and the architecture of Figure 7 for the Montgomery modular multiplication case. However, it is known that all the operands being tested

here have a bit length of 32 bits, due to the word size of the used processor. For this reason, the applications using the operator from Figure 17 for modulos with less than 32 bit need to perform a modulo operation in the first iteration, in order to pass to the modular multipliers inputs with the same length of the modulo.

This happens because the multipliers generated by the used software tool demands operands with the length of n+1 bits. To the *mod* function approach, this is a way to use only multipliers with modulated inputs, allowing them to work with the delay from Figure 29, avoiding the large critical path provided by the usage of the multipliers with 32 bits operands, shown in Figure 24. Evidently, this strategy is not needed for the pseudo-modulo $2^{32} - 1$ from test 16, which can use directly the circuit from Figure 17. This is also valid to other (32×2^n)-bit long inputs, as 64, 128 and so on.

The three modular exponentiation operators where synthesized, DM, CM and MM (NADJIA; MOHAMED, A.; MOHAMED, I., 2012), firstly being analyzed the usage of FPGA's logic elements for each one, which is depicted in Figure 30. The worst case obtained was using the compression-based multipliers for $2^n + k$ modulos, with its pseudo-modulos using considerably less resources. Regarding to the exponentiation using the native function, the original modulos showed a better result, being more advantageous in terms of area. However, the best chip occupation clearly belongs to the Montgomery modular exponentiation operators, using much fewer elements than the other applications. It is justified, due to the simplicity of the multiplication hardware used.

Figure 30 – Total FPGA logic elements used in modular exponentiation

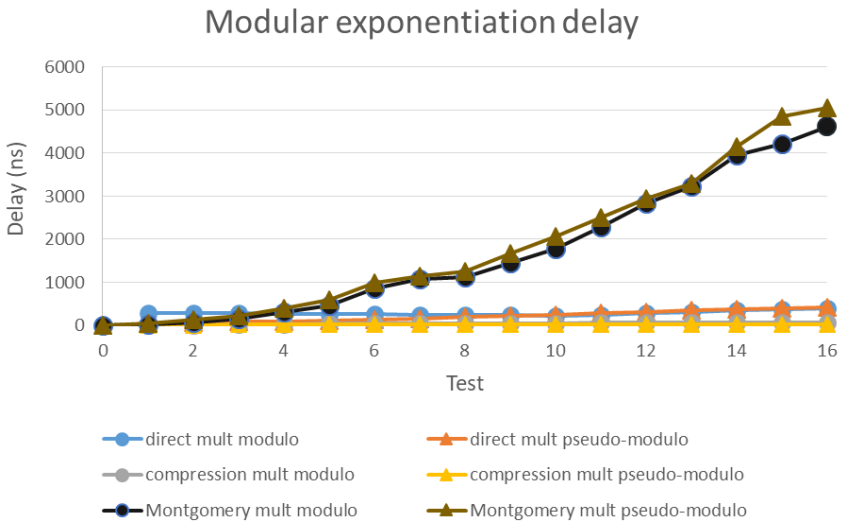


Source: The author

After that, the STA was performed and shown in Figure 31. It can be seen that,

despite using less chip area, the Montgomery modular exponentiation takes much more time to complete an exponentiation iteration, as happened for the multiplication in Figure 28.

Figure 31 – Delay for modular exponentiation

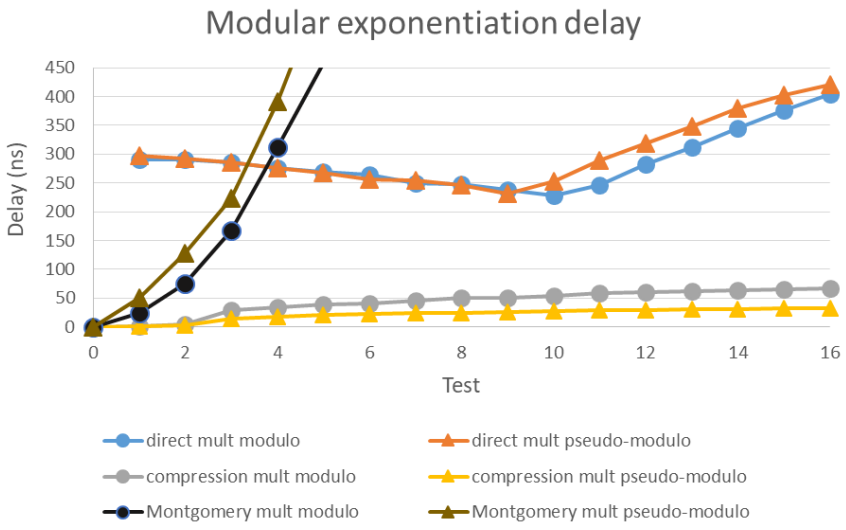


Source: The author

Aiming to enable the analysis of the curves for the other exponentiation cases, Figure 32 depicts a zoom in the results. Again, as for the multiplication, the fastest operation time was obtained using the compression-based operators, followed by its original modulos. In opposition to that, the delay for pseudo-modulos using *mod* function still worse than the original ones. In addition to that, the curves present a peculiar format, first decreasing and then increasing again. This behaviour can be explained by the need to process the 32 bits inputs in the first iteration, performing the modulo operation to pass the operands to the modulated $(n+1)$-bit format. By doing this, the initial variation will be the same as in Figure 24, since the delay of modulo reduction from *mod* function will dominate the operation. The increase from tests 10 for original modulos and 9 for pseudo-modulos means that the modular multiplication with the modulated inputs overrides the modular reduction.

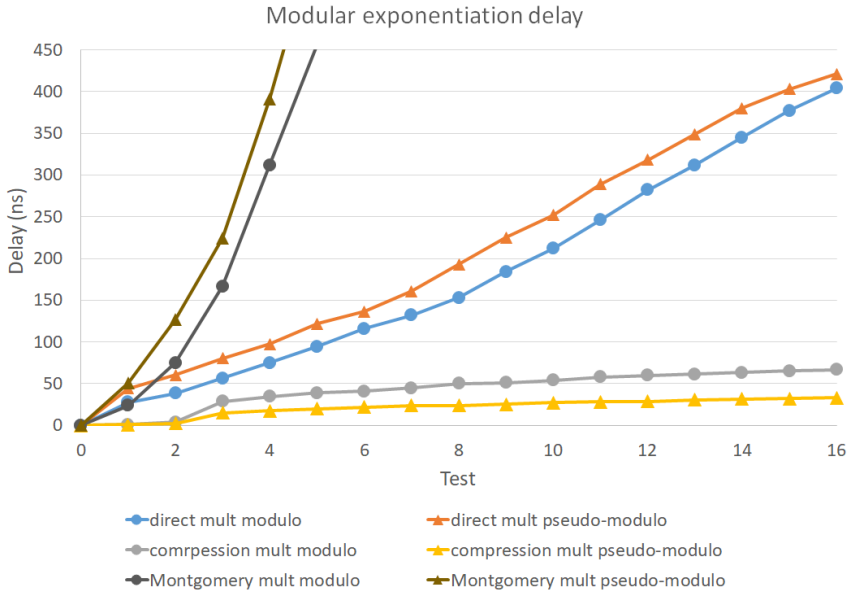
The modular exponentiation delay taking into account only modulated inputs for direct multiplication using *mod* function is presented in Figure 33, i.e., disregarding the first modulo reduction. Here, the delay variation is very similar to the multiplication of Figure 26. The delay from Figure 31 can be seen as superposition of the curves from Figures 33 and 24.

Figure 32 – Zoom in delay for modular exponentiation



Source: The author

Figure 33 – Zoom in delay for modular exponentiation with modulated inputs

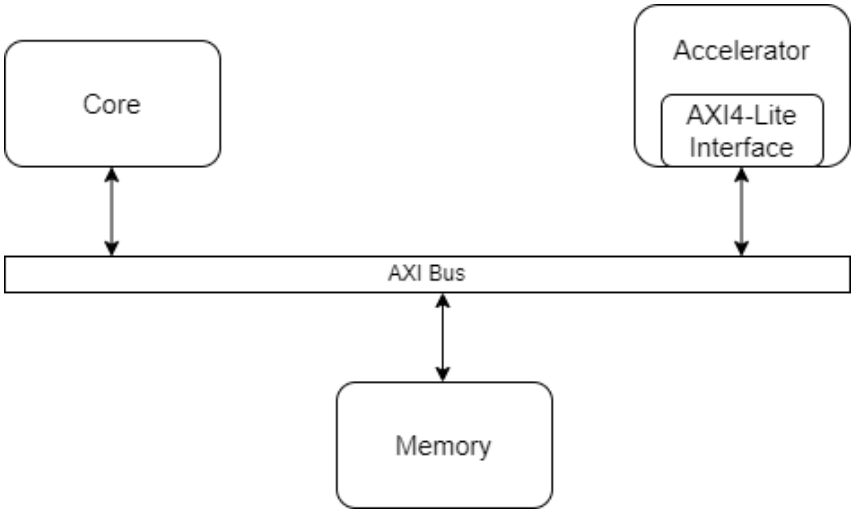


Source: The author

5.4 IMPLEMENTATION WITH THE INTEGRATED PROCESSOR

The integration between the processor and the accelerator was done by the AXI4-Lite bus, as can be seen in Figure 34, where is depicted the connection of the core with the co-processor and the main memory, where the accelerator registers values are set. The three blocks communicate via the AXI bus, which is interfaced to the proposed peripheral by the AXI4-Lite protocol.

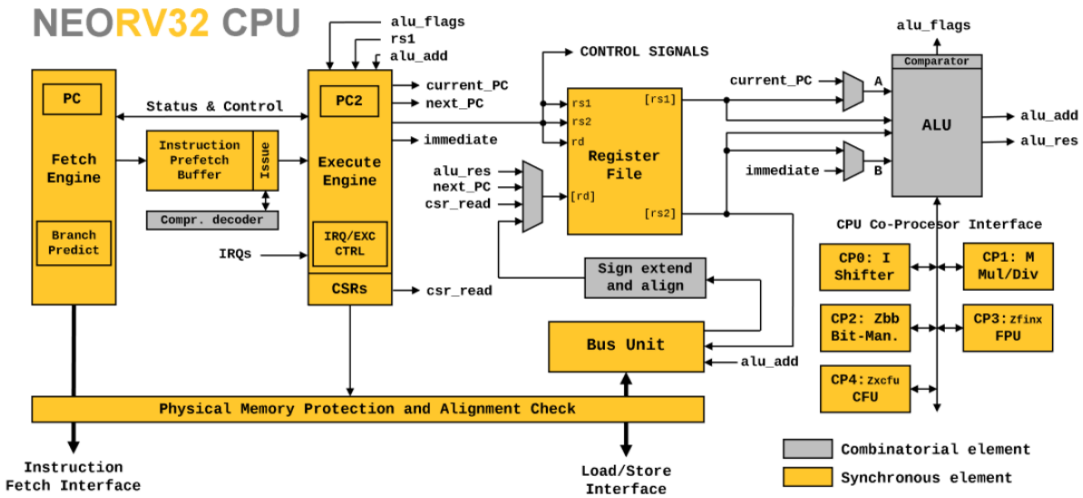
Figure 34 – Processor integrated with the accelerator



Source: The author

The chosen core was the NEORV32 (Figure 35) (NOLTING, n.d.), a RISC-V based SoC which intend to be a customizable CPU that support different interfaces, functions and allows the implementation of custom peripherals in an easy way. Its versatility and support for the protocol being used here were the main reasons for its selection.

Figure 35 – Block diagram of NEORV32 core

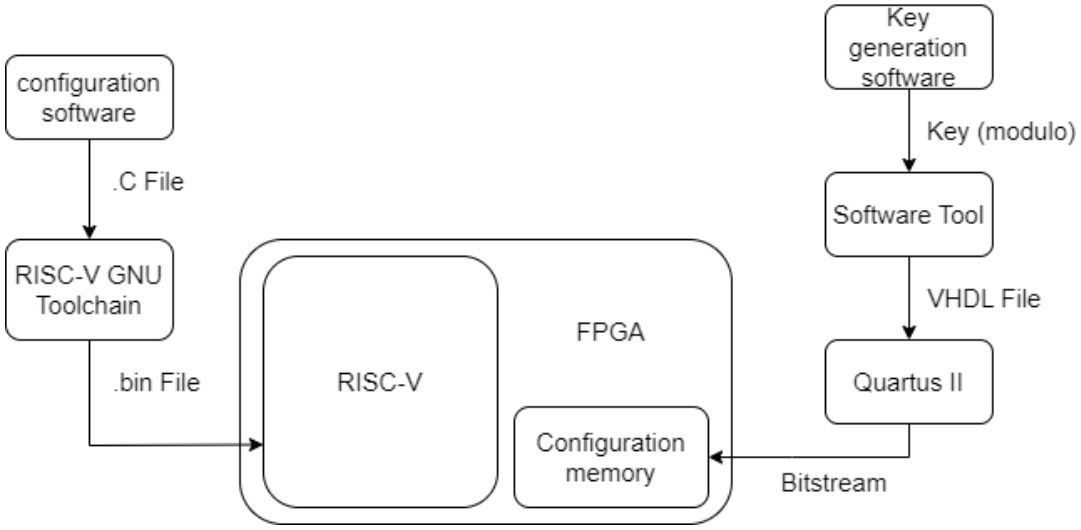


Source: (NOLTING, n.d.)

The tests were carried out as shows Figure 36. To the case using the compression-based multipliers, the arithmetic structures are generated externally by the software tool based on the public-key required, using the key generated by a Python software as

parameters (modulos). The new architecture is then compiled and the bitstream is sent to the FPGA configuration memory in order to reconfigure the system. In this context, the private key is also calculated by the external Python script, and is sent along with the new accelerator configuration. This strategy avoids the need for a block to perform this calculation internally in the accelerator, which would make it more expensive. Instead, it takes advantage of the fact that external software will already be needed anyway.

Figure 36 – Scheme of the structure used to test the processor



Source: The author

The other operators with Montgomery modular multiplier and direct multiplication by *mod* function do not need hardware changes, however, as they are using the same structure than the dedicated multipliers, the decryption key is also being calculated in the external software for simplicity, being updated when necessary. These two implementations also do not need the software tool, bypassing this block in Figure 36 and being implemented directly in Quartus II, but using the public and private keys generated by the Python script in the same way. Finally, the program required to configure and use the accelerator is sent to the processor via a JTAG interface. After that, the code to configure and control the accelerator is compiled by the RISC-V GNU toolchain and recorded to the microprocessor code memory, starting the program execution.

5.4.1 Multiple RSA-length Tests

Once it is known the behaviour of each approach of modular exponentiation with the increase of modulos and operand sizes, the operation delay for the encryption and decryption operation were evaluated with the accelerator integrated to the processor. To perform this tests, were used standard RSA keys, with lengths of 32, 64, 128, 256, 512 and 1024 bits. The obtained results for resources used and total operation time

Table 7 – Total logic elements used for encryption and decryption with RSA keys of 32, 64, 128, 256, 512 and 1024 bits

Test	DM (M)	DM (PM)	CM (M)	CM (PM)	MM (M)	MM (PM)
16	7019	7914	12040	6816	**400**	403
32	16890	17802	26215	14030	**774**	777
64	75264	73577	90691	58425	**1521**	1525
128	116540	119116	144143	110084	**3015**	3022
256	207202	206158	243042	193876	**6003**	6014
512	291971	294094	299828	236353	**11978**	12000

Source: The author

Table 8 – Exponentiation delay for RSA keys of 32, 64, 128, 256, 512 and 1024 bits

Test	DM (M)	DM (PM)	CM (M)	CM (PM)	MM (M)	MM (PM)
16	412 ns	435 ns	98 ns	**48 ns**	4.8 us	5.2 us
32	1.2 us	1055 ns	123 ns	**60 ns**	18.7 us	20.2 us
64	3.9 us	2883 ns	148 ns	**72 ns**	73.6 us	80 us
128	13.9 us	8887 ns	173 ns	**84 ns**	292.2 us	318 us
256	52.3 us	30294 ns	198 ns	**96 ns**	1164.6 us	1.268 ms
512	202.4 us	110700 ns	223 ns	**108 ns**	4649.5 us	5.07 ms

Source: The author

Table 9 – Exponentiation delay $\times$ logic used for RSA keys of 32, 64, 128, 256, 512 and 1024 bits

Test	DM (M)	DM (PM)	CM (M)	CM (PM)	MM (M)	MM (PM)
16	0.002891	0.00344	0.00118	**0.000327**	0.00192	0.00209
32	0.020268	0.0188	0.00322	**0.00084**	0.01447	0.01569
64	0.29353	0.212	0.0134	**0.0042**	0.11194	0.122
128	1.619	1.048	0.0249	**0.00924**	0.881	0.961
256	10.836	6.25	0.0481	**0.0186**	6.99	7.63
512	59.094	32.56	0.0668	**0.0255**	55.69	60.84

Source: The author

are presented in Tables 7 and 8, respectively. Also, in order to evaluate the correlation between thes two metrics, Table 9 presents the multiplicative factor between them. Here, the results for modulos are indicated with (M), while the ones for pseudo-modulos are (PM). In this scenario, the RSA-32 is labeled by Test 16, as in the previous evaluations. Following this, RSA-64 is represented by test 32 (64 bit operands and keys), RSA-128 by test 64 and so on.

Table 10 – Total execution time of encryption using RSA keys of 32, 64, 128, 256, 512 and 1024 bits

Test	CM (M)	CM (PM)	MM (M)	MM (PM)
16	1.6 us	**816 ns**	82 us	88 us
32	2 us	**1 us**	318 us	343 us
64	2.5 us	**1.2 us**	1.3 ms	1.4 ms
128	2.9 us	**1.4 us**	5 ms	5.4 ms
256	3.4 us	**1.6 us**	19.8 ms	21.6 ms
512	3.8 us	**1.8 us**	79 ms	86.2 ms

Source: The author

It is important to point that, the results for direct multiplication (DM) for word length larger than 32, i.e., after test 16, are extrapolations for comparison purposes only, since they can not be synthesized due to the 32 bit limit operands of *mod* function. As can be seen from Table 7, the Montgomery modular multiplication approach presents the smallest usage of resources, having smaller size in the original modulo case, since it has less bits. On the other hand, the largest resource consumption is given by the compression-based multipliers approach using $2^n + k$ modulos, especially for the 512 and 1024 bit keys. The accelerator using the same multipliers with pseudo-modulos used about the half of logic elements, even less than the direct multiplication cases.

Regarding to the delay, Table 8 shows that the maximum clock period for encryption and decryption using the dedicated multipliers with pseudo-modulos is, on average, 51.4 % faster than the same scheme with the original modulos, and 99.8% faster than DM. In opposition to that, the Montgomery multipliers presents the slowest results for the processing of one cycle of modular exponentiation, using both modulos and pseudo-modulos, being about 100% slower than CM in both cases. This is explained by the large number of iterations required by the operator, showing an unbalanced relationship between area and delay. Observing the results from Table 9, especially those referring to the 512 test (1024 bits), it is clear that the multiplier based on compression using pseudo-modulos has a much smaller value than the rest, indicating that the gain in operating time prevails the higher cost in area. In addition to that, the extrapolation shows that the multiplication using the native *mod* function would present the worst delay-area ratio for the original modulos, and for pseudo-modulos until test 256, when it is surpassed by MM.

In the sequence, the total operation time for encryption and decryption was verified, and they are presented in Tables 10 and 11, respectively. In this case, only the CM and MM approaches were evaluated, since DM cannot be synthesized for all the tests, precluding the results obtainment.

It is visible that the total encryption time is much smaller than the decryption time.

Table 11 – Total execution time of decryption using RSA keys of 32, 64, 128, 256, 512 and 1024 bits

Test	**CM (M)**	**CM (PM)**	**MM (M)**	**MM (PM)**
16	3.14 us	**1.58 us**	153.6 us	166.4 us
32	7.9 us	**3.9 us**	1.2 ms	1.3 ms
64	18.9 us	**9.3 us**	9.4 ms	10.3 ms
128	44.3 us	**21.6 us**	74.8 ms	162.8 ms
256	101.4 us	**49.2 us**	596.3 ms	649.2 ms
512	228.3 us	**110.7 us**	4.8 s	5.2 s

Source: The author

It happens because all the encryption were performed using the standard exponent of 65537, which has only 17 bits, meaning that it needs to perform only 17 iterations to finish the operation. The decryption, on the other hand, uses the decryption key *d* as exponent, which presents the same number of bits of the key for safety reasons, thus performing much more iterations and taking longer to finish the calculation.

Despite that, it can be seen that the acceleration resulting from the use of the pseudo-modulo compression multiplication is even more pronounced when it comes to the total time of the cryptographic operation for both encryption and decryption. This strategy is twice as fast compared to the same multiplier using the original modulo, even requiring one more cycle to carry out the modular correction. This is also much faster than the Montgomery multiplication approach, which proved to be much slower, specially for 1024 bit encryption, taking 4.8 seconds to finish the operation using original modulos, about to 21025 times more than CM, and 5.2 seconds using the pseudo-modulos, about to 46974 times the total operation time of CM. The pseudo-modulo use in this case was expected to present the worst performance, since it presents more bits, which means more iterations.

Finally, it was also verified the maximum clock frequency obtained for the whole integrated system, which is an important metric, since that a low frequency limitation caused by the accelerator will affect other functions and peripherals of the processor. The maximum frequency obtained for each case is presented in Table 12. Here is another advantage of compression-based pseudo-modulo exponentiation, which allowed the maximum processor frequency among all the tested schemes, more than twice higher than the other designs. It happens due to the combinational nature of the operator, having a smaller critical path, which is even shorter than the ones from Montgomery exponentiation, since they need to deal with the carry propagation of additions in every cycle.

Table 12 – Encryption/decryption maximum operation frequency for RSA keys of 32, 64, 128, 256, 512 and 1024 bits

Test	CM (M)	CM (PM)	MM (M)	MM (PM)
16	8 MHz	**19 MHz**	6.92 MHz	6.53 MHz
32	3.2 MHz	**8.3 MHz**	3.42 MHz	3.354 MHz
64	1.5 MHz	**3.6 MHz**	1.756 MHz	1.697 MHz
128	0.635 MHz	**1.502 MHz**	0.901 MHz	0.858 MHz
256	0.348 MHz	**0.842 MHz**	0.462 MHz	0.434 MHz
512	0.141 MHz	**0.426 MHz**	0.237 MHz	0.220 MHz

Source: The author

5.4.2 Optimization

Despite the results for total operation delay using the multipliers by compression can be considered satisfactory, the area and frequency can still be improved. Analyzing the architecture from Figure 17, it can be noticed that it has a long critical path due to the presence of squaring and multiplication operations in the same cycle. However, as seen in Section 5.2, the software tool is not capable to generate specific hardware for square operation, which is performed by the generic variable modular multiplier. For this reason, the block of square operation can be removed, leaving one unique multiplier to perform both operations. The new architecture is presented in Figure Figure 37, and its version with the final correction for pseudo-modulo usage is shown in Figure 38.

By doing so, the operator's critical path will be reduced by half, as will the number of logical elements used. In this way, when the scanned bit is 1, two clock cycles will be used to perform multiplication and squaring, but when it is 0, only squaring is necessary, using only one clock cycle. This action allows for great savings in total operating time, especially in encryption, where the highest exponents are used, consequently requiring more iterations. To verify that, the modifications in the exponentiation operator where implemented and the results compiled again, with the logic resources usage and operation delay for one clock cycle shown in Tables 13 and 14, respectively. The multiplicative factor between the two metrics is presented in Table 15.

As expected, the area occupation for the compression-based approach was reduced in approximately the half, as also happened with the operation delay, further improving the delay x area correlation (Table 15). Here, the MM approach presents the worse values for modulos and pseudo-modulos.

After that, the encryption and decryption total operation time were tested using this new architecture, and the obtained results are shown in Tables 16 and 17. By analyzing them, it can be noticed that an average acceleration of about 45 % for encryption and decryption using CM, comparing to results from Tables 10 and 11, for both modulos

Table 13 – Total logic elements used for encryption and decryption with RSA keys of 32, 64, 128, 256, 512 and 1024 bits using the new optimized architecture

Test	**DM (M)**	**DM (PM)**	**CM (M)**	**CM (PM)**	**MM (M)**	**MM (PM)**
16	3510	3957	6020	3408	**400**	403
32	8445	8901	13108	7015	**774**	777
64	37632	36789	45346	29213	**1521**	1525
128	58270	59558	72072	55042	**3015**	3022
256	103601	103079	121521	96938	**6003**	6014
512	145986	147025	149914	118177	**11978**	12000

Source: The author

Table 14 – Encryption and decryption operation delay for RSA keys of 32, 64, 128, 256, 512 and 1024 bits using the new optimized architecture

Test	**DM (M)**	**DM (PM)**	**CM (M)**	**CM (PM)**	**MM (M)**	**MM (PM)**
16	0.2 us	0.21 us	0.04 us	**0.02 us**	4.8 us	5.2 us
32	0.6 us	0.52 us	0.06 us	**0.03 us**	18.7 us	20.2 us
64	1.95 us	1.4 us	0.07 us	**0.036 us**	73.6 us	80 us
128	7 us	4.4 us	0.08 us	**0.042 us**	292.2 us	318 us
256	26.2 us	15.2 us	0.1 us	**0.048 us**	1164.6 us	1268 us
512	101.2 us	55.4 us	0.11 us	**0.05 us**	4649.5 us	5070 us

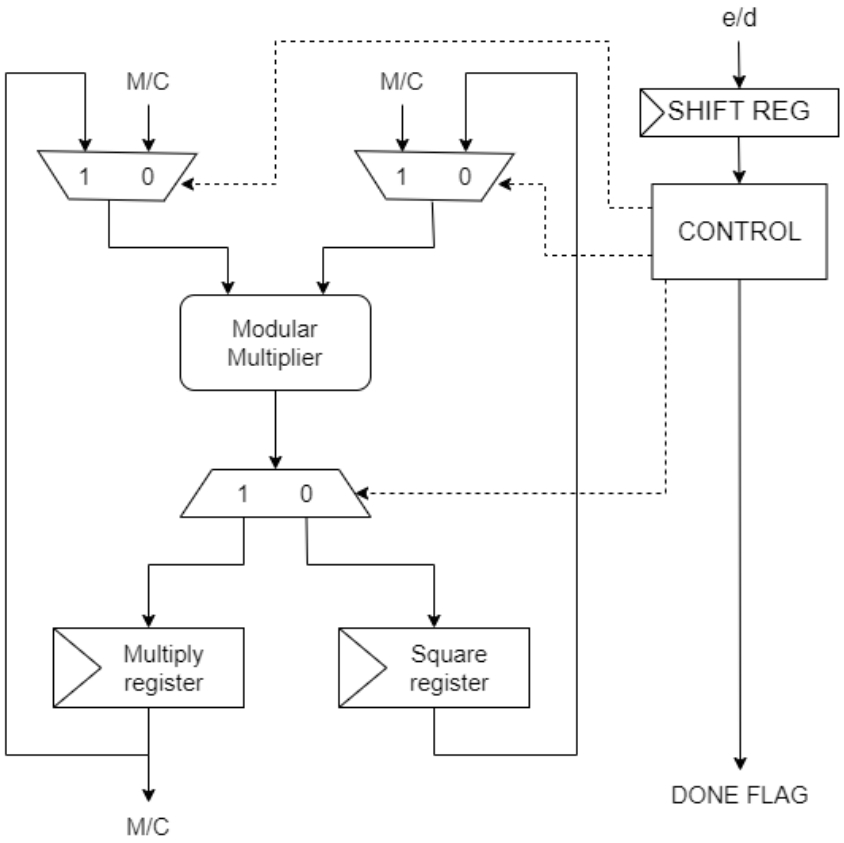
Source: The author

Table 15 – Exponentiation delay $\times$ logic used for RSA keys of 32, 64, 128, 256, 512 and 1024 bits for the new optimized architecture

Test	**DM (M)**	**DM (PM)**	**CM (M)**	**CM (PM)**	**MM (M)**	**MM (PM)**
16	0.00072	0.00086	0.0003	**0.00008**	0.00192	0.00209
32	0.005	0.0047	0.00079	**0.00021**	0.01447	0.01569
64	0.073	0.051	0.0033	**0.001**	0.11194	0.122
128	0.407	0.262	0.0061	**0.0023**	0.881	0.961
256	2.71	1.566	0.012	**0.0046**	6.99	7.63
512	14.77	8.145	0.016	**0.0064**	55.69	60.84

Source: The author

Figure 37 – New architecture proposed, using only one modular multiplier



Source: The author

Table 16 – Total execution time of encryption using RSA keys of 32, 64, 128, 256, 512 and 1024 bits using the proposed new architecture

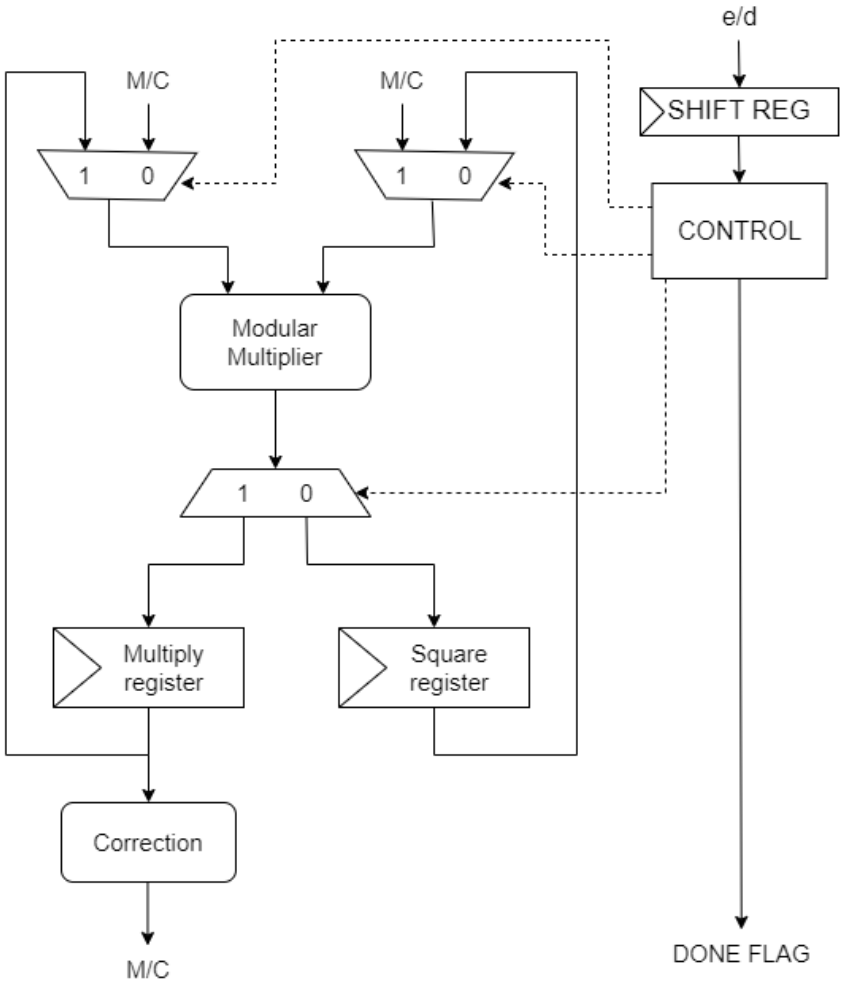
Test	**CM (M)**	**CM (PM)**	**MM (M)**	**MM (PM)**
16	0.93 us	**0.45 us**	82 us	88 us
32	1.1 us	**0.57 us**	318 us	343 us
64	1.4 us	**0.68 us**	1300 us	1400 us
128	1.6 us	**0.8 us**	5000 us	5400 us
256	1.8 us	**0.91 us**	19800 us	21600 us
512	2.1 us	**1 us**	79000 us	86200 us

Source: The author

and pseudo-modulos approaches.

Regarding to the maximum clock frequency, it is shown in Table 18, indicating that the maximum possible frequency also had an increase.

Figure 38 – New architecture proposed, using only one modular multiplier with final correction



Source: The author

Table 17 – Total execution time of decryption using RSA keys of 32, 64, 128, 256, 512 and 1024 bits using the proposed new architecture

Test	CM (M)	CM (PM)	MM (M)	MM (PM)
16	0.002 ms	**0.001 ms**	0.15 ms	166.4 us
32	0.006 ms	**0.003 ms**	1.2 ms	1.3 ms
64	0.014 ms	**0.007 ms**	9.4 ms	10.3 ms
128	0.033 ms	**0.016 ms**	74.8 ms	162.8 ms
256	0.076 ms	**0.036 ms**	596.3 ms	649.2 ms
512	0.17 ms	**0.083 ms**	4800 ms	5200 ms

Source: The author

Test	CM (M)	CM (PM)	MM (M)	MM (PM)
16	14 MHz	**35 MHz**	6.92 MHz	6.53 MHz
32	5.3 MHz	**15 MHz**	3.42 MHz	3.354 MHz
64	2.7 MHz	**7.3 MHz**	1.756 MHz	1.697 MHz
128	1.2 MHz	**2.5 MHz**	0.901 MHz	0.858 MHz
256	0.56 MHz	**1.6 MHz**	0.462 MHz	0.434 MHz
512	0.23 MHz	**0.75 MHz**	0.237 MHz	0.220 MHz

Table 18 – Encryption/decryption maximum operation frequency for RSA keys of 32, 64, 128, 256, 512 and 1024 bits using the new optimized architecture

6 CONCLUSION

In the present work, a proposal for implementing an RSA cryptographic accelerator based on the use of pseudo-modulos was developed. Since the most costly operation of modular exponentiation is multiplication, and the size of its operands tends to increase more and more, the operation becomes increasingly slower and inefficient. With this in mind, the project was carried out with the aim of improving the performance of modular multiplication and exponentiation as a whole, in order to avoid a drastic decrease in operating time.

To obtain performance gains, the strategy used was the use of pseudo-modulos, which consist of multiples of the original modulo, and which can present better results. For this work, we specifically focused on pseudo-modulos of the format 2^n-1, known to be efficient, which were used to replace their original modulos of the format 2^n+k, which have lower efficiency. In the proposal architecture, the entire modular exponentiation is performed using the pseudo-modulo, and the final result is converted back to the original modulo domain through a final modulo reduction.

The design methodology consists of evaluating and comparing three different architectures for modular multiplication in the exponentiation: direct multiplication using VHDL's native multiplication and modulo operations; the compression-based multipliers generated by the software tool; and Montgomery's modular multiplication, more specifically the optimal algorithm without final subtraction. The objective of this is to compare the three approaches and check which one has the best performance and the best trade-off between area and delay. All structures were synthesized for Altera's FPGA Cyclone V GX 5CGXFC5C6F27C7N, and had their delays evaluated using STA. The results showed that, although Montgomery's algorithm occupies fewer logical elements, compression-based multipliers had a total operating time 10^7 times greater, benefiting greatly from the use of pseudo-modulos, unlike Montgomery's case. Regarding the use of native operations, it was noted that there is a limitation of 32 bits of operands, not allowing multiplications greater than that. Despite this, it was also noted that there is no sensitivity in relation to the type of modulo used.

After that, the accelerator was integrated with the chosen RISC-V processor, in order to verify its performance given the system restrictions. Here the superiority of the operators generated by the software tool was also verified, which allowed the processor to work at a frequency approximately three times higher than the Montgomery operator and direct multiplication in all cases, again presenting the best result with the use of pseudo -modulos.

Thus, the results of this work show that there is a significant gain in the use of pseudo-modulos to accelerate multiplication and modular exponentiation for dedicated compression-based operators. More than that, its use brings a large gain in total oper-

ating time when compared to state-of-the-art architectures, such as the Montgomery algorithm, which tend to a much greater increase in operating time with increasing key sizes.

Given this, the main contributions of this work can be seen as:

- First formal presentation of the concept of pseudo-modulos and its application.
- First cryptogtaphic accelerator fully based in compression multipliers.
- First RISC-V RSA accelerator using compression-based multipliers.

6.1 FUTURE WORKS

- Investigate specialized algorithms for quadratic operation in order to obtain a more optimized result than a generic multiplication.
- Explore different exponentiation algorithm, as the window exponentiation, in order to optimize the exponentiation total time.
- Use more secure padding schemes, like OAEP, in order to improve the reliability.
- The use of paralelism of pseudo-modulos by dividing it in two terms: $(2^n-1) = (2^{\frac{n}{2}}-1) \times (2^{\frac{n}{2}}+1)$, and using a correction based on the chinese remainder theorem (CRT).

REFERENCES

ANTAO, Samuel; SOUSA, Leonel. A Flexible Architecture for Modular Arithmetic Hardware Accelerators based on RNS. **Journal of Signal Processing Systems**, v. 76, p. 249–259, Sept. 2014.

ARENAS-HOYOS, Sergio Andrés; BERNAL-NOREÑA, Álvaro. Performance evaluation of M-ary algorithm using reprogrammable hardware. **DYNA**, v. 84, n. 203, p. 75–79, Oct. 2017.

ASIF, Shahzad; VESTERBACKA, Mark. An RNS based modular multiplier with reduced complexity. In: 2017 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC). [S.l.: s.n.], 2017. P. 1–4.

BANAKAR, R.; STEINKE, S.; LEE, Bo-Sik; BALAKRISHNAN, M.; MARWEDEL, P. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: PROCEEDINGS of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627). [S.l.: s.n.], 2002. P. 73–78.

BARRETT, Paul. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In: ODLYZKO, Andrew M. (Ed.). **Advances in Cryptology — CRYPTO' 86**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987. P. 311–323.

BHATTACHARJYA, Aniruddha; ZHONG, Xiaofeng; LI, Xing. A lightweight and efficient secure hybrid RSA (SHRSA) messaging scheme with four-layered authentication stack. **IEEE Access**, IEEE, v. 7, p. 30487–30506, 2019.

BLEICHENBACHER, Daniel. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS1, Feb. 2002.

BONEH, Dan; CANETTI, Ran; HALEVI, Shai; KATZ, Jonathan. Chosen-ciphertext security from identity-based encryption. **SIAM Journal on Computing**, SIAM, v. 36, n. 5, p. 1301–1328, 2007.

BOS, Jurjen; COSTER, Matthijs. Addition Chain Heuristics. In: BRASSARD, Gilles (Ed.). **Advances in Cryptology — CRYPTO' 89 Proceedings**. New York, NY: Springer New York, 1990. P. 400–407.

BRENT; KUNG. A Regular Layout for Parallel Adders. **IEEE Transactions on Computers**, v. C-31, n. 3, p. 260–264, 1982.

ÇALIŞKAN, Dursun. An application of RSA in data transfer. In: 2011 5th International Conference on Application of Information and Communication Technologies (AICT). [S.l.: s.n.], 2011. P. 1–4.

CHEN, Yun-Lu; TSENG, Chih-Yeh; CHANG, Hsie-Chia. Design and Implementation of Reconfigurable RSA Cryptosystem. In: 2007 International Symposium on VLSI Design, Automation and Test (VLSI-DAT). [S.l.: s.n.], 2007. P. 1–4.

CHO, Koon-Shik; RYU, Je-Hyuk; CHO, Jun-Dong. High-speed modular multiplication algorithm for RSA cryptosystem. In: IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society (Cat. No.37243). [S.l.: s.n.], 2001. v. 1, 479–483 vol.1.

COTA, Emilio G.; MANTOVANI, Paolo; DI GUGLIELMO, Giuseppe; CARLONI, Luca P. An analysis of accelerator coupling in heterogeneous architectures. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). [S.l.: s.n.], 2015. P. 1–6.

COUSINS, David Bruce; ROHLOFF, Kurt; SUMOROK, Daniel. Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor. **IEEE Transactions on Emerging Topics in Computing**, v. 5, n. 2, p. 193–206, 2017.

DADDA, Luigi. Some schemes for parallel multipliers. **Alta frequenza**, v. 34, p. 349–356, 1965.

DALLY, William J.; TURAKHIA, Yatish; HAN, Song. Domain-Specific Hardware Accelerators. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 63, n. 7, p. 48–57, June 2020. ISSN 0001-0782.

DI CLAUDIO, E.D.; ORLANDI, G.; PIAZZA, F. Fast RNS DSP algorithms implemented with binary arithmetic. In: INTERNATIONAL Conference on Acoustics, Speech, and Signal Processing. [S.l.: s.n.], 1990. 1531–1534 vol.3.

DING, Jinnan; LI, Shuguo. A Modular Multiplier Implemented With Truncated Multiplication. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 65, n. 11, p. 1713–1717, 2018.

EL MAKKAOUI, Khalid; LAMRIJI, Youssef; OUAHBI, Ibrahim; NABIL, Omayma; BOUZAHRA, Anas; BENI-HSSANE, Abderrahim. Fast Modular Exponentiation Methods for Public-Key Cryptography. In: 2022 5th International Conference on Advanced Communication Technologies and Networking (CommNet). [S.l.: s.n.], 2022. P. 1–6.

FADAVI-ARDEKANI, J. M*N Booth encoded multiplier generator using optimized Wallace trees. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 1, n. 2, p. 120–125, 1993.

FERNANDES, Gabriel Bruno Monteiro. **Infraestrutura automática para aritmética computacional baseada em RNS**. Nov. 2021. MSc Thesis – Universidade Federal de Santa Catarina, Florianópolis, SC.

GOMES, Tiago; SOUSA, Pedro; SILVA, Miguel; EKPANYAPONG, Mongkol; PINTO, Sandro. FAC-V: An FPGA-Based AES Coprocessor for RISC-V. **Journal of Low Power Electronics and Applications**, v. 12, n. 4, 2022. ISSN 2079-9268.

GORDON, Daniel M. A Survey of Fast Exponentiation Methods. **Journal of Algorithms**, v. 27, n. 1, p. 129–146, 1998. ISSN 0196-6774.

GROSSSCHÄDL, Johann. High-Speed RSA Hardware Based on Barret's Modular Reduction Method. In: p. 191–203.

HAMEED, Rehan; QADEER, Wajahat; WACHS, Megan; AZIZI, Omid; SOLOMATNIKOV, Alex; LEE, Benjamin C.; RICHARDSON, Stephen; KOZYRAKIS, Christos; HOROWITZ, Mark. Understanding Sources of Inefficiency in General-Purpose Chips. In: PROCEEDINGS of the 37th Annual International Symposium on Computer Architecture. Saint-Malo, France: Association for Computing Machinery, 2010. (ISCA '10), p. 37–47.

HAN, Jun; DOU, Renfeng; ZENG, Lingyun; WANG, Shuai; YU, Zhiyi; ZENG, Xiaoyang. A Heterogeneous Multicore Crypto-Processor With Flexible Long-Word-Length Computation. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 62, n. 5, p. 1372–1381, 2015.

HAUCK, S.; DEHON, A. **Reconfigurable Computing: The Theory and Practice of FPGA-based Computation**. [S.l.]: Morgan Kaufmann, 2008. (Systems on Silicon Series). ISBN 9780123705228.

HISAKADO, Toru; KOBAYASHI, Nobuyuki; GOTO, Satoshi; IKENAGA, Takeshi; HIGASHI, Kunihiko; KITAO, Ichiro; TSUNOO, Yukiyasu. 61.5mW 2048-bit RSA Cryptographic Co-processor LSI based on N bit-wised Modular Multiplier. In: 2006 International Symposium on VLSI Design, Automation and Test. [S.l.: s.n.], 2006. P. 1–4.

HOLLMANN, Henk D.L.; RIETMAN, Ronald; HOOGH, Sebastiaan de; TOLHUIZEN, Ludo; GORISSEN, Paul. A Multi-layer Recursive Residue Number System. In: 2018 IEEE International Symposium on Information Theory (ISIT). [S.l.: s.n.], 2018. P. 1460–1464.

HUNG, C.Y. Carry-save adders for computing the product AB modulo N. English. **Electronics Letters**, Institution of Engineering and Technology, v. 26, 899–900(1), 13 June 1990. ISSN 0013-5194.

INTEL CORPORATION. **Cyclone V GX FPGA Configuration User Guide**. Santa Clara, CA, Mar. 2023.

K, Venkata Reddy; C, Simranjeet Singh; DESALPHINE, Vivian; SELVAKUMAR, David. A Low Latency Montgomery Modular Exponentiation. **Procedia Computer Science**, v. 171, p. 800–809, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19). ISSN 1877-0509.

KARATSUBA, Anatolii. The complexity of computations. **Proceedings of the Steklov Institute of Mathematics**, v. 211, p. 169–, Jan. 1995.

KATZ, Jonathan; LINDELL, Yehuda. **Introduction to modern cryptography: principles and protocols**. [S.l.]: Chapman and hall/CRC, 2007.

KAWAMURA, Shin-ichi; HIRANO, Kyoko. A fast modular arithmetic algorithm using a residue table. In: SPRINGER. WORKSHOP on the Theory and Application of of Cryptographic Techniques. [S.l.: s.n.], 1988. P. 245–250.

KIM, Da Won; MAULANA, Dalta Imam; JUNG, Wanyeong. Kyber Accelerator on FPGA Using Energy-Efficient LUT-Based Barrett Reduction. In: 2022 19th International SoC Design Conference (ISOCC). [S.l.: s.n.], 2022. P. 83–84.

KIM, Taewhan; JAO, W.; TJIANG, S. Circuit optimization using carry-save-adder cells. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 17, n. 10, p. 974–984, 1998.

KNUTH, Donald Ervin. **The art of computer programming**. [S.l.]: Pearson Education, 1997. v. 3.

KRISHNAKUMAR, Anish; OGRAS, Umit; MARCULESCU, Radu; KISHINEVSKY, Mike; MUDGE, Trevor. Domain-Specific Architectures: Research Problems and Promising Approaches. **ACM Trans. Embed. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 2, Jan. 2023. ISSN 1539-9087.

LABORATORIES, RSA. **PKCS v2.20: Cryptographic Token Interface Standard**. [S.l.: s.n.], 2004.

LIU, Ye; GONG, Wei; FAN, Wenqing. Application of AES and RSA Hybrid Algorithm in E-mail. In: 2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS). [S.l.: s.n.], 2018. P. 701–703.

LOI, K.C. Cinnati; KO, Seok-Bum. Flexible elliptic curve cryptography coprocessor using scalable finite field arithmetic blocks on FPGAs. **Microprocessors and Microsystems**, v. 63, p. 182–189, 2018. ISSN 0141-9331.

M.T.GOODRICH, R.Tamassia. **Algorithm Design: Foundations, Analysis, and Internet Examples**. New York, NY: JohnWiley Sons, Inc., 2002.

MANOR, Erez; GREENBERG, Shlomo. Using HW/SW Codesign for Deep Neural Network Hardware Accelerator Targeting Low-Resources Embedded Processors. **IEEE Access**, v. 10, p. 1–1, Jan. 2022.

MATUTINO, Pedro; CHAVES, Ricardo; SOUSA, Leonel. Arithmetic Units for RNS Moduli 2n-3 and 2n+3 Operations. In: p. 243–246.

MATUTINO, Pedro Miguens; ARAÚJO, Juvenal; SOUSA, Leonel; CHAVES, Ricardo. Pipelined FPGA coprocessor for elliptic curve cryptography based on residue number

system. In: 2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). [S.l.: s.n.], 2017. P. 261–268.

MATUTINO, Pedro Miguens; PETTENGHI, Hector; CHAVES, Ricardo; SOUSA, Leonel. RNS Arithmetic Units for Modulo $2^n + -k$. In: 2012 15th Euromicro Conference on Digital System Design. [S.l.: s.n.], 2012. P. 795–802.

MIYAMOTO, Atsushi; HOMMA, Naofumi; AOKI, Takafumi; SATOH, Akashi. Systematic design of RSA processors based on high-radix montgomery multipliers. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 19, n. 7, p. 1136–1146, 2011. Cited by: 71.

MOHAN, P.V.A. **Residue Number Systems: Algorithms and Architectures**. [S.l.]: Springer US, 2012. (The Springer International Series in Engineering and Computer Science). ISBN 9781461509974.

MÖLLER, Bodo. Improved Techniques for Fast Exponentiation. In: LEE, Pil Joong; LIM, Chae Hoon (Eds.). **Information Security and Cryptology — ICISC 2002**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. P. 298–312.

MONTGOMERY, Peter L. Modular multiplication without trial division. **Mathematics of Computation**, v. 44, p. 519–521, 1985.

NADJIA, Anane; MOHAMED, Anane; MOHAMED, Issad. Montgomery modular exponentiation on FPGA. In: 2012 24th International Conference on Microelectronics (ICM). [S.l.: s.n.], 2012. P. 1–4.

NGUYEN-HOANG, Duc-Thinh; MA, Khai-Minh; LE, Duy-Linh; THAI, Hong-Hai; CAO, Tran-Bao-Thuong; LE, Duc-Hung. Implementation of a 32-Bit RISC-V Processor with Cryptography Accelerators on FPGA and ASIC. In: 2022 IEEE Ninth International Conference on Communications and Electronics (ICCE). [S.l.: s.n.], 2022. P. 219–224.

NIEMIEC, Gabriel S.; BATISTA, Luis M. S.; SCHAEFFER-FILHO, Alberto E.; NAZAR, Gabriel L. A Survey on FPGA Support for the Feasible Execution of Virtualized Network Functions. **IEEE Communications Surveys Tutorials**, v. 22, n. 1, p. 504–525, 2020.

NOLTING, S. **The NEORV32 RISC-V Processor**. [S.l.: s.n.]. Available from: `https://github.com/stnolting/neorv32`.

NOORDAM, Leon. **VHDL Implementation of 4096-bit RNS Montgomery Modular Exponentiation for RSA Encryption**. June 2019. MSc Thesis – Delft University of Technology, Delft, Netherlands.

PALUDO, Rogério. **Implementações eficientes de conversores reversos e multiplicações por constante usando residue number systems**. Apr. 2020. PhD Thesis – Universidade Federal de Santa Catarina, Florianópolis, SC.

PARANDEH-AFSHAR, Hadi; BRISK, Philip; IENNE, Paolo. Exploiting fast carry-chains of FPGAs for designing compressor trees. In: 2009 International Conference on Field Programmable Logic and Applications. [S.l.: s.n.], 2009. P. 242–249.

PARHAMI, Behrooz. A note on digital filter implementation using hybrid RNS-binary arithmetic. **Signal Processing**, v. 51, n. 1, p. 65–67, 1996. ISSN 0165-1684.

PATRONIK, Piotr; PIESTRAK, Stanisław J. Design of residue generators with CLA/compressor trees and multi-bit EAC. In: 2017 IEEE 8th Latin American Symposium on Circuits Systems (LASCAS). [S.l.: s.n.], 2017. P. 1–4.

PATTERSON, David A.; HENNESSY, John L. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128122757.

PICCOLBONI, Luca; DI GUGLIELMO, Giuseppe; SETHUMADHAVAN, Simha; CARLONI, Luca P. HARDROID: Transparent Integration of Crypto Accelerators in Android. In: 2021 IEEE High Performance Extreme Computing Conference (HPEC). [S.l.: s.n.], 2021. P. 1–8.

PU, Qiong; ZHAO, Xiuying. Montgomery Exponentiation with No Final Comparisons: Improved Results. In: 2009 Pacific-Asia Conference on Circuits, Communications and Systems. [S.l.: s.n.], 2009. P. 614–616.

REZAI, Abdalhossein; KESHAVARZI, Parviz. High-Throughput Modular Multiplication and Exponentiation Algorithms Using Multibit-Scan-Multibit-Shift Technique. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 23, n. 9, p. 1710–1719, 2015. Cited by: 54.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 21, n. 2, p. 120–126, Feb. 1978. ISSN 0001-0782.

ROYO, A.; MORAN, J.; LOPEZ, J.C. Design and implementation of a coprocessor for cryptography applications. In: PROCEEDINGS European Design and Test Conference. ED TC 97. [S.l.: s.n.], 1997. P. 213–217.

SHOUP, Victor. Practical threshold signatures. In: SPRINGER. ADVANCES in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19. [S.l.: s.n.], 2000. P. 207–220.

SKLIAROVA, Iouliia; SKLYAROV, Valery. **FPGA-BASED hardware accelerators**. [S.l.]: Springer, 2019.

SOUSA, Leonel; ANTAO, Samuel; MARTINS, Paulo. Combining Residue Arithmetic to Design Efficient Cryptographic Circuits and Systems. **IEEE Circuits and Systems Magazine**, v. 16, n. 4, p. 6–32, 2016.

ST DENIS, Tom; ROSE, Greg. Chapter 5 - Multiplication and Squaring. In: ST DENIS, Tom; ROSE, Greg (Eds.). **BigNum Math**. Burlington: Syngress, 2006. P. 91–146. ISBN 978-1-59749-112-9.

STELLING, P.F.; MARTEL, C.U.; OKLOBDZIJA, V.G.; RAVI, R. Optimal circuits for parallel multipliers. **IEEE Transactions on Computers**, v. 47, n. 3, p. 273–285, 1998.

SUZUKI, Daisuke; MATSUMOTO, Tsutomu. How to Maximize the Potential of FPGA-Based DSPs for Modular Exponentiation. **IEICE Transactions**, 94-A, p. 211–222, Jan. 2011.

THAMPI, Nitha; JOSE, Meenu Elizabath. Montgomery Multiplier for Faster Cryptosystems. **Procedia Technology**, v. 25, p. 392–398, 2016. 1st Global Colloquium on Recent Advancements and Effectual Researches in Engineering, Science and Technology - RAEREST 2016 on April 22nd 23rd April 2016. ISSN 2212-0173.

TRIMBERGER, Stephen M. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. **Proceedings of the IEEE**, v. 103, n. 3, p. 318–331, 2015.

TSAI TERENCE, 1964-; CHENG BORSHIUAN, 1952-. **The silicon dragon : high-tech industry in Taiwan**. Northampton, MA: Edward Elgar Cheltenham, 2006.

UENO, Rei; HOMMA, Naofumi. How Secure is Exponent-blinded RSA–CRT with Sliding Window Exponentiation? **IACR Transactions on Cryptographic Hardware and Embedded Systems**, v. 2023, n. 2, p. 241–269, Mar. 2023.

URQUHART, Roddy. **What is an ASIP?** [S.l.: s.n.], Apr. 2021. Available from: `https://codasip.com/2021/04/16/what-is-an-asip/`.

VERGOS, Haridimos; EFSTATHIOU, C. Design of efficient modulo 2n + 1 multipliers. **Computers Digital Techniques, IET**, v. 1, p. 49–57, Feb. 2007.

VERMA, Rupali; DUTTA, Maitreyee; VIG, Renu. FPGA implementation of RSA based on carry save Montgomery modular multiplication. In: 2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT). [S.l.: s.n.], 2016. P. 107–112.

WALLACE, C. S. A Suggestion for a Fast Multiplier. **IEEE Transactions on Electronic Computers**, EC-13, n. 1, p. 14–17, 1964.

WATERMAN, Andrew; LEE, Yunsup; PATTERSON, David A.; ASANOVI, Krste. The RISC-V Instruction Set Manual. In.

WIENER, M.J. Cryptanalysis of short RSA secret exponents. **IEEE Transactions on Information Theory**, v. 36, n. 3, p. 553–558, 1990.

WILLIAMS, R. Stanley. What's Next? [The end of Moore's law]. **Computing in Science Engineering**, v. 19, n. 2, p. 7–13, 2017.

XIA, Hong; HU, Wenhao; YAN, Jiangyu. Design and Implementation of High-Performance Modular Exponentiation Arithmetic Unit. In: 2009 First International Conference on Information Science and Engineering. [S.l.: s.n.], 2009. P. 1691–1694.

XIAO, Chenglong; CASSEAU, Emmanuel; WANG, Shanshan; LIU, Wanjun. Automatic custom instruction identification for application-specific instruction set processors. **Microprocessors and Microsystems**, v. 38, 8, Part B, p. 1012–1024, 2014. ISSN 0141-9331.

XIAO, Yinhao; JIA, Yizhen; LIU, Chunchi; CHENG, Xiuzhen; YU, Jiguo; LV, Weifeng. Edge Computing Security: State of the Art and Challenges. **Proceedings of the IEEE**, v. 107, n. 8, p. 1608–1631, 2019.

YEH, Chingwei; HSU, En-Feng; CHENG, Kai-Wen; WANG, Jinn-Shyan; CHANG, Nai-Jen. An 830mW, 586kbps 1024-bit RSA chip design. In: PROCEEDINGS of the Design Automation Test in Europe Conference. [S.l.: s.n.], 2006. v. 2, 6 pp.-.

YU, Zhan; YU, Meng-Lin; WILLSON, A.N. Signal representation guided synthesis using carry-save adders for synchronous data-path circuits. In: PROCEEDINGS of the 38th Design Automation Conference (IEEE Cat. No.01CH37232). [S.l.: s.n.], 2001. P. 456–461.

ZHENG, Xinjian; LIU, Zexiang; PENG, Bo. Design and Implementation of an Ultra Low Power RSA Coprocessor. In: 2008 4th International Conference on Wireless Communications, Networking and Mobile Computing. [S.l.: s.n.], 2008. P. 1–5.

ZHONG, Yutong. An Overview of RSA and OAEP Padding. **Highlights in Science, Engineering and Technology**, v. 1, p. 82–86, June 2022.