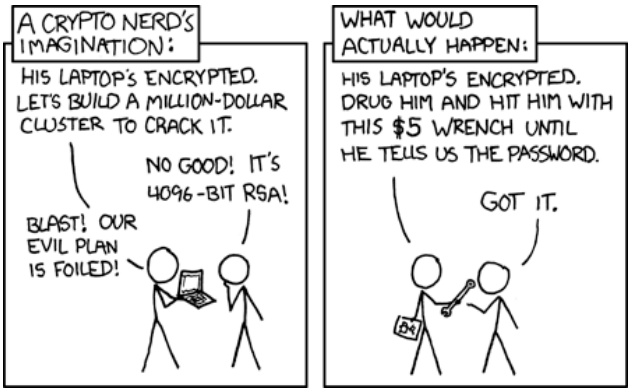
VHDL Implementation of 4096-bit RNS Montgomery Modular Exponentiation for RSA Encryption

Leon Martijn Noordam
Student MSc Computer Engineering
Technical University of Delft
Mekelweg 4, 2628CD Delft, Netherlands

Q&CE-CE-MS-2019-07

June 18, 2019



From: https://xkcd.com/538

TU Delft Delft University of Technology

Technolution

Abstract

Modular exponentiation is the basis needed to perform RSA encryption and decryption. Execution of 4096-bit modular exponentiation using an embedded system requires many arithmetic operations. This work aims to improve the performance of modular exponentiation for an existing FPGA platform containing a soft core RISC-V processor. The solution is to introduce a peripheral that performs Montgomery multiplication with 4096-bit operands. These operands are represented using the Residue Number System (RNS) and each residue is assigned to a RNS processor core. In total, the system consists of 121 RNS cores and each core is responsible for a 34-bit residue. RNS Montgomery multiplication requires a base extension algorithm, which will be represented by the Bajard and the Shenoy base extensions. Two designs are proposed: one using a tree-based reduction circuit and one with an iterative reduction circuit. The former focuses on performance overall, while the latter is more efficient in terms of performance per area. Synthesis for a `XCKU035` FPGA gives an area usage of 83484 LUTs for the tree-based design and 41857 LUTs for the iterative design. Both designs also require 132.5 BRAMs and 485 DSP blocks and are running at a clock frequency of 400 MHz. The tree-based design performs Montgomery multiplication using 4096-bit operands in 434 cycles (1.09 µs), while the iterative design does that in 577 cycles (1.44 µs). For performing modular exponentiation, the sliding window method is used with an optimal window size of seven. A modular exponentiation with 4096-bit operands takes on average 5.09 ms for the tree-based design, while the iterative version needs 6.78 ms. Few other implementations were found performing Montgomery multiplication using 4096-bit operands. However, compared to those the proposed design provides better performance.

Keywords: *4096 bits, RSA, RNS, FPGA, Montgomery multiplication, modular reduction, base extension, exponentiation, hardware accelerator, multi-core, parallel processing, heterogeneous architecture, encryption, decryption.*

Thesis committee

Title and name	Function	Affiliation
Dr. ir. S.D. Cotofana	Responsible professor	CE QCE EEMCS
Dr. ir. A.J. van Genderen	Daily supervisor	CE QCE EEMCS
Dr. J. Hoekstra	Committee member	EEE ME EEMCS
Ir. A. Wiersma	Daily supervisor	Technolution

Contents

1 Introduction

More than 40 years after the introduction of the RSA standard, it still is widely used for encryption and the signing of digital messages. The strength of RSA encryption depends, among other things, on the size of two randomly generated prime numbers. Since the original proposal of the algorithm, the computational power of processors has increased significantly. RSA encryption can be broken by factoring the modulus. The modulus is equal to the multiplication of the two randomly generated primes. So although there exist faster processors now, increasing the size of the primes makes factoring still difficult.

Nowadays 2048-bit RSA encryption is used for most communication channels, while for more sensitive information 4096-bit encryption is used. Decryption of a 4096-bit cipher requires a lot of arithmetic operations, but commonly used desktop computers are able to perform it quite fast. However, embedded platforms have difficulty decrypting this kind of cipher as they provide limited arithmetic processing capacity. Enabling such a platform with the ability to perform 4096-bit RSA encryption and decryption poses a challenge.

Many types of embedded platforms exist, but in this thesis we focus on a platform containing a Field-Programmable Gate Array (FPGA). An FPGA chip contains logical blocks that can be programmed to perform arithmetic operations like addition, multiplication or division. There is also storage available in the form of flip-flops/registers and small blocks of Random Access Memory (RAM). Using the building blocks provided by the FPGA, an implementation is made of a cryptographic accelerator that is able to perform 4096-bit RSA encryption.

1.1 Research goal

The PrimeLink from Technolution is a product that enables secure communication over the Internet between two or more private networks by creating a Virtual Private Network (VPN). It is compliant with the Dutch government's requirements for the securing of classified information and provides a high level of confidentiality. Internally, the PrimeLink contains a FPGA chip with a processor that runs the software for creating and maintaining VPN tunnels. This software is based on an implementation of OpenVPN-NL.

The device is designed to create a large number of secure tunnels at the same time. A certificate with a RSA signature is used in order to validate the authenticity of the endpoint for the connection. As the current design does not contain a dedicated accelerator for RSA encryption, validation of the certificate is a very slow operation. When setting up hundreds of tunnels this process takes a considerable amount of time to complete. The goal of this thesis is to design a logic circuit that accelerates 4096-bit RSA encryption.

For logic circuit design the two programming languages used most are: VHSIC Hardware Description Language (VHDL) and Verilog. As VHDL is the industry standard in Europe and it also is taught at the TU Delft, the RSA accelerator will be made using VHDL.

1.2 Contributions

The main contributions of this thesis are summarized as followings:

- Theoretical comparison of several existing exponentiation algorithms using addition chains and finding the optimal windows size for a 4096-bit exponent.
- Elaborate examples with calculations for RSA encryption, arithmetic in RNS and the Shenoy base extension, which is also applicable for the Bajard base extension.
- To show that using fixed-point precomputed constants for the Diophantine base extension is not suited for an implementation in VHDL, based on numerical analysis.
- First VHDL implementation of a fixed modulo reduction circuit using the iterative reduction approach consisting of table lookups, additions and shifting.
- Area- and performance-improved tree-based fixed modulo reduction circuit, in comparison to the fixed modulo reduction implementation generated by `Vivado 2018.2`.
- A very fast RSA encryption accelerator running at 400 MHz implemented in VHDL.

1.3 Thesis outline

The thesis contains five sections: background theory, design considerations, implementation overview, benchmark results and conclusion. Background theory is an overview of all theoretical knowledge needed to understand how the design works and can be found in Section 2. In Section 3, the theory to create a foundation for the implementation is discussed. Section 4 gives a top-down explanation on how the design for the RSA accelerator is implemented in hardware. In order to determine the performance of the accelerator, some benchmarks are executed and those results are presented in Section 5. Finally, Section 6 provides a summary of everything discussed before and gives some ideas for further optimizations and future work.

An important remark to make before reading further: in this thesis an additional notation for the modulo operator is used. In addition to the `mod` operator, the following notation applies:

$$x \bmod y = |x|_y$$

This notation is also used in most of the papers regarding the background theory.

2 Background theory

This section contains all theoretical knowledge needed to understand how the design works. It provides an introduction to RSA cryptography, the Residue Number System (RNS), exponentiation algorithms, reduction methods and Montgomery multiplication. Finally, an explanation is given about performing Montgomery multiplication in RNS.

2.1 RSA cryptography

In February 1978 the ACM magazine published an article written by R.L. Rivest, A. Shamir and L. Adleman about an implementation of a "public-key cryptosystem" [1]. The methods described are currently known under the acronym RSA encryption, which refers to the initials of the authors' surnames. It can be used to encrypt a message such that only receivers with knowledge of a certain decryption key are able to decipher it. RSA was a completely new algorithm, because it uses different keys for encryption and decryption. And because the encryption key is given as public information, it is called public-key encryption.

The color notation used for variables in this section denotes them either being public or secret. Sending just one of the orange-colored variables over a public domain enables anyone intercepting traffic to decrypt the secret message. The mathematical basis of RSA encryption is called modular exponentiation and can be summarized with Eq. (2.1).

$$c = m^e \bmod n \tag{2.1}$$

where m is the secret message, e is the exponent, n is the modulus and c is the resulting cipher. Modulus n is derived by multiplying two random large prime numbers q and p. For w-bit RSA encryption, primes q and p should contain $w/2$ bits each, so that n contains w bits. Exponent e can be chosen arbitrary, but should be pairwise prime to $p-1$. Common choices for e are 3, 5, 17, 257 and 65537, because of their low Hamming weight (number of ones in binary form). This reduces the number of multiplications needed for exponentiation.

Reversing the cipher-text can be done by calculating $m = c^d \bmod n$, where d is the decryption key. Decryption key d is the modular multiplicative inverse of e with respect to $\phi = (p-1) \cdot (q-1)$ and is denoted by $d = |e^{-1}|_\phi$. This notation means that d follows from the relation $e \cdot d \bmod \phi \equiv 1$, which can be solved for d using the Extended Euclidean algorithm from [2], referred to as `EGCD`. The steps for RSA key generation are as follows:

1. Select exponent e, a small prime with low Hamming weight.
2. Generate two large random primes q and p.
3. Calculate the modulus $n = p \cdot q$.
4. Calculate the Euler totient function $\phi = (p-1) \cdot (q-1)$.

5. Verify that the GCD$(e, \phi) = 1$, if not go back to step 2.
6. Calculate the decryption key $d = |e^{-1}|_{\phi}$ using `EGCD`.
7. Publish e and n in a public domain, so anyone can send you an encrypted message.

RSA has survived many years of analysis, but it can be broken when the random number generator is not suited for cryptographic usage [3]. To ensure correct usage, extra requirements should be placed on the generation of prime values q and p [4]. Also, the sender should make sure that message m is properly padded with a mechanism such as OAEP [5]. The following scenario is given to illustrate the encryption and decryption process.

Alice wants to send Bob a secret PIN-code $m = 12021993$ to open a locker. She does not trust the medium over which they are communicating and therefore she wants to encrypt the code. Before Alice can perform encryption, Bob needs to generate a pair of RSA keys. Following the steps described previously, he chooses primes $p = 62639$, $q = 53987$ and exponent $e = 5$. From this decision follows that $n = p \cdot q = 3381691693$ and $\phi = (p-1) \cdot (q-1) = 3381575068$. Taking the modular multiplicative inverse of e with respect to ϕ results in decryption key $d = 2028945041$. These steps are summarized in Table 2.1.

Generated			Calculated			
e	$=$	5	n	$=$	$p \cdot q$	$=$ 3381691693
p	$=$	62639 $\rightarrow$	ϕ	$=$	$(p-1) \cdot (q-1)$	$=$ 3381575068
q	$=$	53987	d	$=$	$\lvert e^{-1} \rvert_{\phi}$	$=$ 2028945041

Table 2.1: Example of variables needed to perform 32-bit RSA encryption and decryption.

The public key pair $\{n, e\}$ is given to Alice. To encrypt the PIN-code, she computes:

$$c = m^e \bmod n = 12021993^5 \bmod 3381691693 = 1227753926$$

Now she can send the encrypted code c over an untrusted medium. Cipher c cannot be related to the PIN-code without having decryption key d, which is only available to Bob. Finally, Bob can decrypt the message by computing:

$$m = c^d \bmod n = 1227753926^{2028945041} \bmod 3381691693 = 12021993$$

Having obtained the secret PIN-code m, Bob is now able to open the locker. The security of RSA encryption is related to the factorization problem. Decryption key d can be calculated from p, q and e. Because n is available to anyone, being able to reverse the operation $n = p \cdot q$ will break encryption. For the 32-bit modulo used in the example, it is trivial to do with any modern computer. So for practical applications, larger prime numbers should be generated. The largest n currently reported to be successfully factorized has a length of 768 bits [6].

2.2 Residue number system

In RNS an integer is represented by taking the modulo of that number with respect to a set of pairwise primes called the moduli. A set of k moduli is called a RNS base and is denoted by $\mathcal{B} = \{m_1, m_2, ..., m_k\}$. Pairwise prime means that the Greatest Common Divisor (GCD) for each pair of moduli in $\mathcal{B}$ is equal to 1. The range of numbers M that can be represented by RNS base $\mathcal{B}$ is equal to the product of all moduli m_i. Conversion of integer $X < M$ to RNS base $\mathcal{B}$ containing k moduli, results in residues $X_{RNS} = \{x_1, x_2, ..., x_k\}$.

The advantage of RNS is that numbers can be split into parts and those can be processed in parallel. This approach improves the speed of addition, subtraction and multiplication compared to standard representation. Divisions that have an exact answer are calculated by multiplying them with the multiplicative inverse of the dividend. However, divisions with a non-exact answer cannot be computed, as the system only represents integers.

Another disadvantage is that the magnitude of a number cannot be determined while in RNS. To perform these operations, the number should be converted to another format that allows it, like regular binary arithmetic. Conversions from and to the RNS domain are computationally expensive, which will be shown in the next section. Executing a single calculation in RNS will not make that calculation any faster. However, performing many consecutive calculations while keeping the number in the RNS domain, will increase performance.

For example, converting $X = 1976$ to RNS with base $\mathcal{B} = \{5, 7, 13, 17\}$ will result in residues $X_{RNS} = \{1, 2, 0, 4\}$, as derived from Eq. (2.2).

$$\left.\begin{array}{rrrcl} X & \text{mod} & 5 & = & 1 \\ X & \text{mod} & 7 & = & 2 \\ X & \text{mod} & 13 & = & 0 \\ X & \text{mod} & 17 & = & 4 \end{array}\right\} \rightarrow X_{RNS} = \{1, 2, 0, 4\} \tag{2.2}$$

Conversion from the RNS domain back to the positional system requires more calculations. It can be computed using the Chinese Remainder Theorem (CRT) [7]. Let $M_i = M/m_i$ be the product of all moduli but one. X can then be derived from Eq. (2.3).

$$X = \sum_{i=1}^{k} x_i \cdot M_i \left|M_i^{-1}\right|_{m_i} \bmod M = \left|\sum_{i=1}^{k} x_i \cdot M_i \left|M_i^{-1}\right|_{m_i}\right|_M \tag{2.3}$$

To find X, each residue x_i has to be multiplied with a weight factor $W_i = M_i \left|M_i^{-1}\right|_{m_i}$. Note that $|M_i^{-1}|_{m_i}$ is the modular multiplicative inverse of M_i with respect to m_i. Following up on the previous example, dynamic range $M = 5 \cdot 7 \cdot 13 \cdot 17 = 7735$. The weight factors for RNS base $\mathcal{B} = \{5, 7, 13, 17\}$ can now be calculated as shown by the equations in Table 2.2.

Calculate M_i				Calculate weight factor W_i						
M_1	$=$	$7735/5$	$=$ 1547	W_1	$=$	$1547 \cdot \lvert 1547^{-1} \rvert_5$	$=$	$1547 \cdot \lvert 2^{-1} \rvert_5$	$=$	4641
M_2	$=$	$7735/7$	$=$ 1105	W_2	$=$	$1105 \cdot \lvert 1105^{-1} \rvert_7$	$=$	$1105 \cdot \lvert 6^{-1} \rvert_7$	$=$	6630
M_3	$=$	$7735/13$	$=$ 595	W_3	$=$	$595 \cdot \lvert 595^{-1} \rvert_{13}$	$=$	$595 \cdot \lvert 10^{-1} \rvert_{13}$	$=$	2380
M_4	$=$	$7735/17$	$=$ 455	W_4	$=$	$455 \cdot \lvert 455^{-1} \rvert_{17}$	$=$	$455 \cdot \lvert 13^{-1} \rvert_{17}$	$=$	1820

Table 2.2: Calculation of weight factor W_i for each modulo m_i in RNS base $\mathcal{B}$.

Note that this process only depends on the set of moduli in $\mathcal{B}$ and can be precomputed when $\mathcal{B}$ is constant. Continuing with Eq. (2.3), the next step is to multiply each W_i with the corresponding residue x_i, sum all intermediate results and reduce them to modulo M. The conversion of $X_{RNS} = \{1, 2, 0, 4\}$ to its original representation is shown by Eq. (2.4).

$$\begin{aligned} X &= \left| \sum_{i=1}^{4} x_i \cdot W_i \right|_M \\ &= (1 \cdot 4641 + 2 \cdot 6630 + 0 \cdot 2380 + 4 \cdot 1820) \bmod 7735 \\ &= 25181 \bmod 7735 = 1976 \end{aligned} \tag{2.4}$$

Addition, subtraction and multiplication of two RNS numbers (which have equal base $\mathcal{B}$) can be performed on the individual residues. Operations on the residues of two RNS numbers are always reduced to modulo m_i, to keep results within dynamic range M. For example, you can calculate $X^2_{RNS} = X_{RNS} \cdot X_{RNS}$ using Eq. (2.5).

$$\left.\begin{array}{rcrcrl} x_1 & = & 1 \cdot 1 & \bmod & 5 & = 1 \\ x_2 & = & 2 \cdot 2 & \bmod & 7 & = 4 \\ x_3 & = & 0 \cdot 0 & \bmod & 13 & = 0 \\ x_4 & = & 4 \cdot 4 & \bmod & 17 & = 16 \end{array}\right\} \rightarrow X^2_{RNS} = \{1, 4, 0, 16\} \tag{2.5}$$

The expected result of this operation is $X^2 \bmod M = 1976^2 \bmod 7735 = 6136$. Converting $X^2_{RNS} = \{1, 4, 0, 16\}$ using the CRT from Eq. (2.3) results in Eq. (2.6).

$$\begin{aligned} X^2 &= \left| \sum_{i=1}^{4} x_i^2 \cdot W_i \right|_M \\ &= (1 \cdot 4641 + 4 \cdot 6630 + 0 \cdot 2380 + 16 \cdot 1820) \bmod 7735 \\ &= 60281 \bmod 7735 = 6136 \end{aligned} \tag{2.6}$$

It can be observed that the result matches the expected outcome.

2.3 Exponentiation

Exponentiation is a mathematical operation to calculate x^e. A straightforward implementation would be to initialize a variable to 1 and multiply it e times with x. However, this approach is slow for large values of e. It can be improved by reusing intermediate values and multiplying those with each other. But which order should the intermediate values be multiplied to result in the least number of multiplications? The following sections will discuss methods for finding a reasonable number of multiplications to perform exponentiation.

2.3.1 Addition chains

Since exponents are additive, the problem of finding an optimal chain of multiplications can be reduced to finding a chain of additions [8]. An addition chain is a sequence of integers starting at 1, with the property that the next number is calculated by adding any of the current values to the most recent one. The goal is to find the shortest list of additions that result into the number of interest e. For example, a possible chain for calculating $e = 8190$ consists of the following 16 addition steps:

```
1 2 3 6 7 14 28 56 63 126 252 504 1008 2016 4032 4095 8190
```

It can be seen that when appending one integer to the chain, the number of possible paths is multiplied by the current length of the chain. For an addition chain of length l elements (omitting start number 1), there are $l!$ paths that can be created. The problem grows exponentially, although there are multiple paths leading to the same result. These duplicates can be ignored, because we are interested in the most efficient (i.e. shortest) path only.

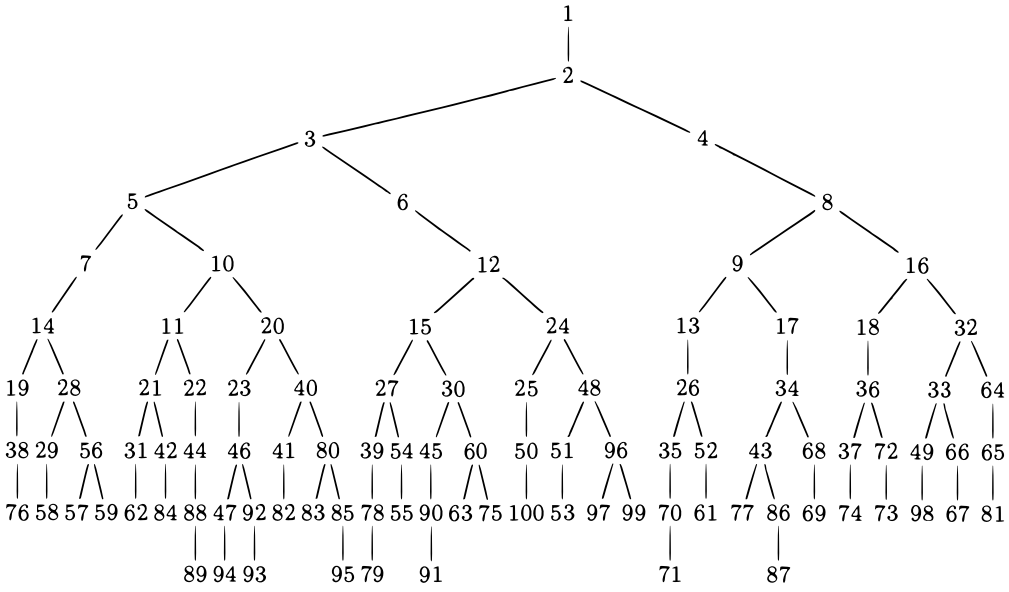


Figure 2.1: A tree that minimizes the number of multiplications, for $e \leq 100$ [8].

The tree depicted in Fig. 2.1 shows the shortest addition chains for $e \leq 100$. Determining the optimal addition chain for any exponent has been proven NP-complete [9]. In practice, this means that for large values of e approximation algorithms are used to find close-to-optimal chains. The next sections will review some of these algorithms.

2.3.2 Binary exponentiation

The most widely known exponentiation algorithm is binary exponentiation. It is easy to implement in hardware and has low memory requirements. There are two versions of binary exponentiation: right-to-left and left-to-right. These refer to the order in which the bits of the exponent are processed, least significant bit (LSB) or most significant bit (MSB) first.

In terms of addition chains, binary exponentiation is a more restricted version of the conditions set on the generic addition chain. To calculate the next integer in the sequence, only two options are available: double the most recent value in the sequence (square) or add 1 (multiply). Therefore binary exponentiation is also known as the square-and-multiply algorithm. For example, calculation of x^{11} can be written as the following addition chain:

```
1 2 4 5 10 11
```

From this sequence follows that x^{11} can be decomposed as $x(x(x^2)^2)^2$. This sequence consists of 5 multiplications and is an optimal solution, as follows from Fig. 2.1. However, due to the restriction of only using multiply and square operations, finding an optimal chain almost never occurs. Using binary exponentiation to calculate x^{8190} gives a chain of 23 multiplications, which is more than the solution from the previous section.

Although not being optimal, the algorithm runs in polynomial time and does not require traversing multiple paths of a tree structure. These two properties make it an excellent strategy for practical implementations. The formal algorithm is shown in Algorithm 1. The bits in exponent e are indexed from right to left, starting at position zero. For example, when taking an exponent of one byte $e = 00001101_2$, the function $\texttt{MSB}(e)$ will return 3.

Algorithm 1 Left-to-right binary exponentiation

```
1: procedure BinExp(x, e)                          ▷ Calculate x^e
2:     r ← 1                                       ▷ Initialize result r with 1
3:     i ← MSB(e)                                  ▷ Find the most significant bit
4:     while i ≥ 0 do                              ▷ Loop until i is negative
5:         r ← r · r                               ▷ Always square result r
6:         if n_i = 1 then                         ▷ If bit i in exponent e is set
7:             r ← r · x                           ▷ Also multiply result r by x
8:         i ← i − 1
9:     return r
```

In Algorithm 1, the first step is setting the result variable $r = 1$. Next, the most significant bit in e is located and its value is stored in i. Starting at the MSB, for each bit position in e, a square operation is applied to r. If the bit is asserted, squaring of r is followed by a multiplication with x. The steps of the algorithm are shown in Table 2.3.

Bit number	Bit value	Operation	Result r
3	1	square	1
		multiply	x
2	0	square	x^2
1	1	square	$(x^2)^2$
		multiply	$x(x^2)^2$
0	1	square	$(x(x^2)^2)^2$
		multiply	$x(x(x^2)^2)^2$

Table 2.3: Calculation of $r = x^{11}$ using left-to-right binary exponentiation.

The cost of naive exponentiation is e multiplications for any exponent e. For binary exponentiation, the minimum cost is given by $\lfloor \log_2 e \rfloor$ multiplications [10], for when e is a power of two. An upper bound on the number of multiplications is given by $2\lfloor \log_2 e \rfloor$, for when e is a power of two minus one and consists of ones only.

2.3.3 M-ary exponentiation

The binary exponentiation algorithm consumes one bit of the exponent in each iteration. Instead of consuming only one bit, the algorithm could be adapted to use more bits at the same time. A generalized version of the `BinExp` algorithm is called m-ary exponentiation, where m is used to denote the radix, which is the number of digits used to represent numbers in a positional numeral system. Thus Binary exponentiation is equal to 2-ary exponentiation.

Taking $m = 2^w$ for $w > 1$ gives interesting results, because raising x to the power of m requires only w squarings. For a practical implementation, it is required to precompute values $x^2, x^3, ..., x^{m-1}$. The algorithm is related to Algorithm 1, but instead of squaring, it computes r^m and instead of multiplying with x only, it can also multiply with one of the precomputed values, depending on the current radix-m digit of the exponent. The definition of the MSB does exist only in radix two, therefore this needs to be replaced by the most significant digit (MSD). The routine for m-ary exponentiation can be found in Algorithm 2.

When using $m = 2^w$, the number of multiplications is at most $2^w - 2 + (1 + 1/w)\lfloor \log_2 e \rfloor$. This upper bound can be broken down in: $2^w - 2$ multiplies for the precomputation, $\lfloor \log_2 e \rfloor$ squarings and at most $\lfloor \log_2 e \rfloor / w$ multiplies [10]. The lower bound is satisfied for an exponent that is a power of two, only squares are needed for its computation. So by removing the part for multiplications in the upper bound, the lower bound is formed: $2^w - 2 + \lfloor \log_2 e \rfloor$. Additional memory is needed to store the precomputed values, compared to `BinExp`.

Algorithm 2 Left-to-right m-ary exponentiation

```
1:  procedure M-ARYEXP(x, e, m)                ▷ Calculate x^e in radix m
2:      r ← 1                                  ▷ Initialize result r with 1
3:      i ← MSD(e)                             ▷ Find the most significant digit
4:      ⟨x^2, x^3, ..., x^(m−1)⟩               ▷ Precompute some powers of x
5:      while i ≥ 0 do                         ▷ Loop until i is negative
6:          r ← r^m                            ▷ Always raise result r by its radix
7:          if e_i ≠ 0 then                    ▷ If digit i in exponent e is not zero
8:              r ← r · x^(e_i)                ▷ Multiply r by a precomputed value
9:          i ← i − 1
10:     return r
```

In Algorithm 2, note that the conditional statement on line 7 can be removed. Substitution of $e_i = 0$ on line 8 then results in $r \leftarrow r \cdot x^{e_i} = r \cdot x^0 = r$, so the value of r is not changed. The conditional statement is used to emphasize that a multiplication by one can be avoided.

Having an upper bound defined for 2^w-ary exponentiation is useful, but is not clear which values of w are a good choice for a given value of e. In the case of 4096-bit RSA, exponent e contains 4096 bits and therefore $\lfloor \log_2 e \rfloor = 4096$. Substitution of this value in the previously defined upper and lower bound equations results in Fig. 2.2.

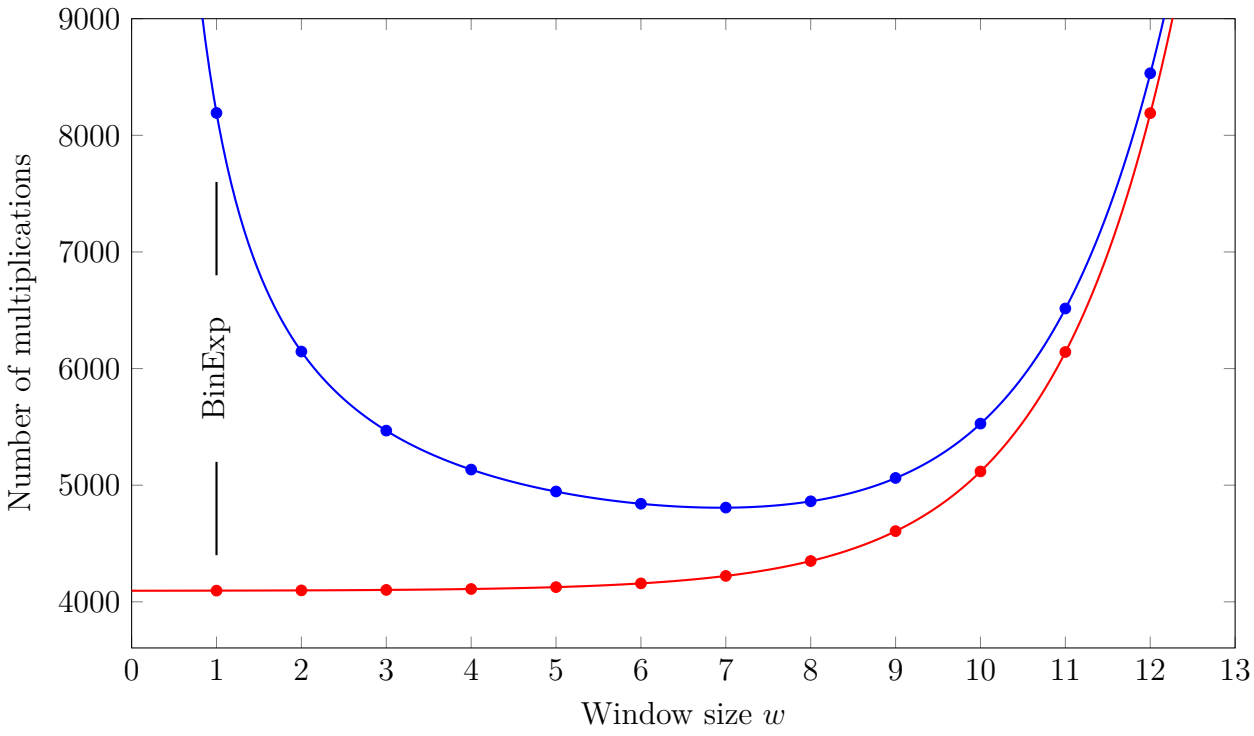


Figure 2.2: Number of multiplications in 2^w-ary exponentiation for $\lfloor \log_2 e \rfloor = 4096$.

From Fig. 2.2 it can be derived that $w = 7$ seems to be the optimal choice for the least number of multiplications. The average number of multiplications can be computed by taking the upper bound and subtracting the probability of finding a window containing only zeros [11]. For exponent e the average number of multiplications is therefore given by:

$$2^w - 2 + \lfloor \log_2 e \rfloor + \frac{\lfloor \log_2 e \rfloor}{w}(1 - 2^{-w}) \tag{2.7}$$

Eq. (2.7) gets closer to the upper bound for a larger windows size as the probability of finding a window of consecutive zeros is smaller. An overview of the maximum, average and minimum of these numbers for an exponent e of 4096 bits is given by Table 2.4. Non-integer results were rounded to their closest integer value.

Window size w	Minimum	Average	Maximum
4	4110	5070	5134
5	4126	4920	4946
6	4158	4830	4841
7	4222	**4803**	**4808**
8	4350	4860	4862
9	4606	5060	5062

Table 2.4: Expected number of multiplications for an exponent of 4096 bits.

The lowest upper bound and average number of multiplications are given by $w = 7$. Note that Eq. (2.7) is not equal to equation 1 in [11]. The difference is that Eq. (2.7) is related to Algorithm 2. In equation 1 from [11] an additional optimization step is introduced, namely setting initial result r to the first applicable precomputed value instead of one. This saves w squares and 1 multiplication, increasing the efficiency of the algorithm.

2.3.4 Sliding window exponentiation

The 2^w-ary exponentiation method can be seen as selecting a w-bit window in the binary representation of e, precomputing the powers in the window itself, followed by cycles of squaring w times and one multiplication with a precomputed value. However, there is no reason to force the windows to be next to each other. Adjacent zeros in the binary representation of e do not result in additional multiplications (only squares) and may be skipped. For example, take exponent $e = 26235947428953663183191$. Its binary representation is:

```
1011000111001000000111010010100111010100000010111100000111110011001010101011
```

The optimal choice for the 2^w-ary method for this 75-bit number is $w = 3$ and it consists of 102 multiplications in total [10]. Note that this number also follows from computing its upper bound and subtracting three multiplications for the windows containing only zeros.

Using the window method with a window size up to 4, the number of multiplications is only 93: 8 multiplies to compute the odd powers up to 15, 71 squarings and 14 multiplies for the intermediate values. The selected windows are underlined with their decimal value below.

```
1011000111001000000111010010100111010100000010111100000111110011001010101111
  11      7  1        7    9    9  13 1        11 3       15   9   9   5   7
```

The addition chain used to precompute the odd multiples of x is as follows:

$$\underline{1}\ 2\ \underline{3}\ \underline{5}\ \underline{7}\ \underline{9}\ \underline{11}\ \underline{13}\ \underline{15}$$

If the window contains one of the underlined values, a precomputed multiple of x is used to multiply with. For an equal window size, sliding window exponentiation is faster than 2^w-ary exponentiation, as only odd multiples of x are precomputed. The algorithm for sliding window exponentiation can be found in Algorithm 3, which is a modified version from [12].

Algorithm 3 Left-to-right sliding window exponentiation

```
1:  procedure SlidingWindowExp(x, e, w)                ▷ Calculate x^e
2:      r ← 1                                          ▷ Initialize result r with 1
3:      i ← MSB(e)                                     ▷ Find the most significant bit
4:      ⟨x^3, x^5, ..., x^(2^w − 1)⟩                   ▷ Precompute odd powers of x
5:      while i ≥ 0 do                                 ▷ Loop until i is negative
6:          if e_i = 0 then                            ▷ If bit i in exponent e is zero
7:              r ← r^2                                ▷ Square result r
8:              i ← i − 1
9:          else
10:             s ← max(i − w + 1, 0)                  ▷ Ensure position s is not negative
11:             while e_s = 0 do                       ▷ Find largest possible window size
12:                 s ← s + 1                          ▷ Decrease window position by 1
13:             for 0 to i − s + 1 do                  ▷ Calculate s − i + 1 times the square of r
14:                 r ← r^2
15:             u ← decimal(e_i, e_(i−1), ..., e_s)    ▷ Get i − s + 1 bits from e starting at i
16:             r ← r · x^u                            ▷ Multiply r by a precomputed multiple of x
17:             i ← s − 1
18:     return r
```

An improvement for exponent $e = 26235947428953663183191$ is suggested in [13]. This method uses a larger window and creates an improved addition chain of the necessary intermediate values. The example exponent e is decomposed as follows:

```
10110001110010000001110100101001110101000000101111000001111100110010101011111
        5689              993      117       47          499          343
```

And with the following addition sequence, the precomputed values can be calculated:

1 2 4 8 10 11 18 36 47 55 91 109 117 226
343 434 489 499 933 1422 2844 5688 5689

This further reduces the required number of multiplications to 89: 62 squarings, 5 multiplications of intermediate values and 22 multiplications for the precomputation. There is a trade-off between finding the optimal addition chain and the amount of computation time it takes. When the exponent is not constant, it might not be worth the extra time.

2.4 Modulo operation

The modulo operation calculates the remainder after division of one number by another. Given two positive numbers, dividend x and divisor n, x modulo n is the remainder of Euclidean division of x by n. Computation of the remainder is written as "$x \bmod n$". This definition does not say anything about how the modulo operator can be used in hardware or software. The next sections will discuss implementation strategies for modular reduction.

2.4.1 Reduction by division

As $x \bmod n$ is equal to the remainder of x/n, integer division can be used to calculate the modulo value. When division, multiplication and addition operators are available, the remainder can be calculated by:

$$x \bmod n = x - (\texttt{int}(x/n) \cdot n) \tag{2.8}$$

An advantage is that in many hardware architectures and software platforms these operations are supported. However, calculating it this way has poor performance, as division is a slow operation compared to addition and multiplication. The reason is that division algorithms have dependencies between loop iterations and therefore are not efficiently parallelized.

On many hardware platforms, computing $x \bmod n$ is done at the same time when computing the division of x by n [14][15]. This results in modular reduction being equally expensive as division. However, this is only applicable when hardware division is available. For platforms where hardware division is not available, other methods exist to compute the modulo value.

When modulo n is a constant, it is also possible to precompute the reciprocal $1/n$ and store that in a lookup-table or register. Then the division of Eq. (2.8) can be replaced by a multiplication, which should be faster than executing the division. Note that multiplication by a precomputed reciprocal is an approximation, because $1/n$ could be an irrational number. The minimum number of bits for $1/n$ required to ensure a correct result should be taken into consideration.

2.4.2 Reduction by subtraction

Division can be implemented in logic using iterations of a shift, comparison and conditional subtraction [16]. When the result is found, the remainder is also available, because it is a number smaller than the last possible subtraction. The basic algorithm for division using subtractions can be found in Algorithm 4.

Algorithm 4 Basic division algorithm that calculates the remainder

```
1:  procedure CalcRemainder(x, n)          ▷ Calculate the remainder of x/n
2:      d ← x                              ▷ Load dividend x into variable d
3:      i ← MSB(x) − MSB(n)                ▷ Calculate number of iterations i
4:      s ← n << i                         ▷ Shift n to MSB position of x
5:      while i ≥ 0 do                     ▷ Loop until iteration i is negative
6:          if d ≥ s then                  ▷ If d is greater than or equal to s
7:              d ← d − s                  ▷ Subtract the divisor from the dividend
8:          s ← s >> 1                     ▷ Shift the divisor right; divide by 2
9:          i ← i − 1
10:     return d
```

For example, in Table 2.5 is shown how to find to find the remainder of $x = 1023$ divided by $n = 76$ using Algorithm 4. Calculating the position of the MSB for both numbers gives $\texttt{MSB}(x) = 9$ and $\texttt{MSB}(n) = 6$. From this follows the initial value of $i = 3$. Next, divisor n is shifted left three times to get the initial value for $s = 76 << 3 = 608$. Now the reduction loop can start and in each iteration it will subtract s from d if and only if $d \geq s$. The result is $d = 35$ in the last row of Table 2.5, as expected of computing "1023 mod 76".

Iteration i	Divisor s	Dividend d
3	608	1023
2	304	415
1	152	111
0	76	111
-1	38	35

Table 2.5: Computing x mod n by iterative subtraction.

The advantage of this method is that it is easy to implement in hardware. However, carry propagation of the subtractions scales linearly with the number of bits in x. So for reduction of large values of x, this method is less suitable than those that will be discussed next.

2.4.3 Reduction by lookup and addition

Another approach for modular reduction is described in [17] and variants are proposed in [18], [19] and [20]. It works best for a fixed modulo value n so that the precomputed values can be stored as constants in a Read-Only Memory (ROM) or Look-Up Table (LUT).

The idea is to split x into parts and use precomputations to reduce those parts instead of the whole number at once. By summing the reduced parts, a new number is constructed that is smaller than x but gives the same result after reduction to modulo n. Assuming n is constant, the amount of storage needed for the lookup table is dependent on the partitioning of x. Larger partitions result in faster computation, but require more storage.

For example, $x = 592$ and $n = 23$ are taken and used to compute $x \bmod n$. Converting x to its binary representation gives $x = 1001010000_2 = 2^9 + 2^6 + 2^4$. Next, computation of $x \bmod n$ can be written as $(2^9 + 2^6 + 2^4) \bmod n$. By moving the modulo operator inside the brackets, a summation of numbers related to original reduction appears. However, the result of this summation could be larger than n, so a new modulo operator has to be appended. The calculation steps for the example numbers are denoted in Eq. (2.9).

$$592 \bmod n = (2^9 + 2^6 + 2^4) \bmod n = (2^9 \bmod n + 2^6 \bmod n + 2^4 \bmod n) \bmod n \tag{2.9}$$

The advantage of this method is not evident, as there are three new modulo operations introduced in Eq. (2.9). However, for fixed n the powers of two reduced modulo n can be precomputed and the last modulo reduction can be replaced by a few subtractions. The maximum number of subtractions needed depends on the values of x and n and their size in bits. This process is demonstrated in Eq. (2.10).

$$592 \bmod 23 = (6 + 18 + 16) \bmod 23 = 40 \bmod 23 = 17 \tag{2.10}$$

Several optimizations can be made to this basic method of reduction by addition. Note that a certain number of the lower bits in x cannot be reduced to modulo n, as they are already smaller than n. So precomputing powers of two and summing those is not needed for the lower bits of x. In the example, $2^4 \bmod 23$ does not have to be computed as $16 < 23$.

Another improvement would be to create a higher radix version by reducing multiple bits of x in the same lookup. This requires larger precomputation tables, but increases the performance of the circuit as less iterations are needed. For example, when processing 5 bits per lookup (radix-32) the operation 592 mod 23 can be performed in two additions.

First, lookup table $\mathbb{T}$ is created for the values $a \cdot 2^5 \bmod 23$ with $0 \leq a < 32$. The left shift by five is needed to match the bit positions that will be reduced from x. Lookup table $\mathbb{T}$ is created as follows:

$$
\begin{aligned}
\mathbb{T}[0] &= (0 \cdot 2^5) \bmod 23 \\
\mathbb{T}[1] &= (1 \cdot 2^5) \bmod 23 \\
\mathbb{T}[2] &= (2 \cdot 2^5) \bmod 23 \\
&\ldots \\
\mathbb{T}[29] &= (29 \cdot 2^5) \bmod 23 \\
\mathbb{T}[30] &= (30 \cdot 2^5) \bmod 23 \\
\mathbb{T}[31] &= (31 \cdot 2^5) \bmod 23
\end{aligned}
$$

Now by using this table, $x = 1001010000_2$ can be reduced modulo 23 by splitting x into two parts $x = x_1 \cdot 2^5 + x_0$, looking up x_1 from the table and adding the result to x_0. Splitting x into two parts results into $x_1 = 10010_2$ and $x_0 = 10000_2$. Looking up $\mathbb{T}[x_1] = \mathbb{T}[18] = 1$ and summation of the parts gives Eq. (2.11).

$$
592 \bmod 23 = (\mathbb{T}[18] + x_0) \bmod 23 = (1 + 16) \bmod 23 = 17 \tag{2.11}
$$

As discussed earlier in this section, summation of the lookups from $\mathbb{T}$ and x_0 might be larger than n. When the result is larger than n, a small number of subtractions are needed to get the expected result. How many subtractions are necessary depends on the number of additions in the summation. For a practical implementation, the maximum number of subtractions should be calculated to ensure the result is correct.

In hardware, there are two approaches to making an implementation of this method. The first method is the tree-based solution, which has a large area footprint in terms of logic gates, but is also fast. It works by using multiple tables, so that all parts of x (except x_0) can be reduced in one lookup executed in parallel. Next, all these results are summed together with x_0 using one large multi-operand adder tree.

Now based on the number of additions, the maximum value of the summation can be calculated. Using this upper bound, the number of subtractions is determined and a circuit can be appended for the final reduction step. For large values of x compared to n, the number of subtractions might still be significant. Instead, a second pass of lookup tables and additions can be used to reduce the value within a small distance of n.

One pass of using lookup tables and summation is referred to as one stage. Each stage results into a number y smaller than the previous stage, such that $y \bmod n$ is still congruent to $x \bmod n$. The final stage is always a few conditional subtractions, how many depends on where modulo n is located between the closest powers of two. Values of modulo n close to the upper power of two are preferred as that results into one conditional subtraction only.

The second approach for a hardware implementation is by making an iterative solution. Each iteration one lookup and addition is performed, reducing the amount of hardware

needed. However, the performance is less than the tree-based solution, because the lookups are executed sequentially instead of in parallel. For a design that focuses on low area usage it still is a good option.

2.4.4 Reduction by shift, lookup and addition

A further improved version of the previous method is proposed in [21]. Instead of making a lookup table for each of the bits in x larger than the MSB of n, this method uses one lookup table only. The number of entries in the lookup table depends on the available storage, but the minimum number is one entry. To make up for the missing constants, the summation register shifts left once in each iteration. If shifting result into an overflow, the carry bit is dropped and the precalculated constant in the lookup table is added to the summation register. This process is repeated until it does not overflow anymore.

Continuing on from the example in the previous section, taking $x = 592$ and $n = 23$ and calculating $x \bmod n$ is performed as follows. Since x contains ten bits and n has five bits, x is split into two segments $x_1 = 10010_2$ and $x_0 = 10000_2$. The summation register is defined as $\mathbb{S}$. The lookup table consists of one entry related to n: $\mathbb{T}[1] = 2^5 \bmod n = 9 = 01001_2$.

initialize	$\mathbb{S} = x_1 = 10010_2$
iteration 0	$\mathbb{S} = \mathbb{S} << 1 = (1)00100_2$
	$\mathbb{S} = 00100_2 + \mathbb{T}[1] = 01101_2$
iteration 1	$\mathbb{S} = \mathbb{S} << 1 = 11010_2$
iteration 2	$\mathbb{S} = \mathbb{S} << 1 = (1)10100_2$
	$\mathbb{S} = 10100_2 + \mathbb{T}[1] = 11101_2$
iteration 3	$\mathbb{S} = \mathbb{S} << 1 = (1)11010_2$
	$\mathbb{S} = 11010_2 + \mathbb{T}[1] = (1)00011_2$
	$\mathbb{S} = 00011_2 + \mathbb{T}[1] = 01100_2$
iteration 4	$\mathbb{S} = \mathbb{S} << 1 = 11000_2$
add x_0	$\mathbb{S} = 11000_2 + x_0 = (1)01000_2$
	$\mathbb{S} = 01000_2 + \mathbb{T}[1] = 10001_2$

This example shows that the method proposed in [21] works, as $592 \bmod 23 = 17 = 10001_2$. Note that in the example no final conditional subtraction was needed to get the correct result. However, for other values of x this subtraction might be needed. The algorithm reduces until $\mathbb{S}$ is less than six bits, so at the start of each iteration the maximum value of $\mathbb{S}$ is 31. For modulus $n = 23$ and segments of five bits, at most one conditional subtraction from $\mathbb{S}$ is needed to get a result less than n. This does not change when using more segments.

Instead of processing one bit at a time, multiple bits can be processed in parallel. This results into a higher radix implementation of the algorithm. The advantage is that less additions and shifts are needed to complete the computation. However, a larger lookup

table is required as it is made up of more entries. For example, reducing an input operand modulo $n = 23$ in radix-4 results into the following lookup table.

$$\begin{aligned}
\mathbb{T}[0] &= (0 \cdot 2^5) \bmod 23 = 00000_2 \\
\mathbb{T}[1] &= (1 \cdot 2^5) \bmod 23 = 01001_2 \\
\mathbb{T}[2] &= (2 \cdot 2^5) \bmod 23 = 10010_2 \\
\mathbb{T}[3] &= (3 \cdot 2^5) \bmod 23 = 00100_2
\end{aligned}$$

And the computation of $x \bmod n = 592 \bmod 23 = 10001_2$ using this table is shown below.

initialize	$\mathbb{S} = x_1 = 10010_2$
iteration 0,1	$\mathbb{S} = \mathbb{S} << 2 = (10)01000_2$
	$\mathbb{S} = 01000_2 + \mathbb{T}[2] = 11010_2$
iteration 2,3	$\mathbb{S} = \mathbb{S} << 2 = (11)01000_2$
	$\mathbb{S} = 01000_2 + \mathbb{T}[3] = 01100_2$
iteration 4	$\mathbb{S} = \mathbb{S} << 1 = 11000_2$
add x_0	$\mathbb{S} = 11000_2 + x_0 = (1)01000_2$
	$\mathbb{S} = 01000_2 + \mathbb{T}[1] = 10001_2$

Note that in each iteration the summation register $\mathbb{S}$ is shifted left by two positions, except for the last iteration. Iteration four is different, because the n has a length of five bits. Five is not a multiple of two, so the summation register shifts left one bit to get into the correct position for loading a new segment. Different configurations can be designed using the following parameters: size of x, size of n and the number of bits processed per iteration.

2.5 Montgomery multiplication

In 1985 Peter L. Montgomery proposed what currently is known as Montgomery multiplication [22]. It is a method for performing fast repeated modular multiplication. The following sections will discuss the algorithm itself and how it can be applied to numbers in RNS.

2.5.1 Modular multiplication

Modular multiplication is the operation of calculating $a \cdot b \bmod n$ with $a, b, n \in \mathbb{N}_0$. Division is a relatively expensive operation for processors and so is modular reduction. Montgomery multiplication improves the computation speed by replacing n with another number r. When r is a power of two, division can be replaced by shifting and calculation of the remainder is simplified to applying a bit mask. The catch is however, that before being able to replace n by r, numbers a and b need to be converted to Montgomery form (n-residue representation).

The result of Montgomery multiplication for $a \cdot b \bmod n$ using radix r is defined as

$$\texttt{MontMult}(a, b, n) = a \cdot b \cdot r^{-1} \bmod n \tag{2.12}$$

where a, b and n are integers and r^{-1} is the multiplicative modular inverse of r with respect to n. For the choice of r some restrictions apply, not all possible integers are allowed. Two requirements are: r needs to be larger than n and r and n are coprime. These constraints ensure that the multiplicative modular inverse of n with respect to r exists. Converting a and b to their Montgomery forms $\bar{a}$ and $\bar{b}$ is done using Eq. (2.13).

$$\bar{a} = \texttt{MontMult}(a, r^2 \bmod n, n) \quad \text{and} \quad \bar{b} = \texttt{MontMult}(b, r^2 \bmod n, n) \tag{2.13}$$

The reverse conversion is performed using `MontMult` with one of the operands set to one. Reverting $\bar{a}$ and $\bar{b}$ back to standard representation can be done by using Eq. (2.14).

$$a = \texttt{MontMult}(\bar{a}, 1, n) \quad \text{and} \quad b = \texttt{MontMult}(\bar{b}, 1, n) \tag{2.14}$$

Montgomery multiplication requires an extra integer n' such that $r \cdot r^{-1} - n \cdot n' = 1$, which is a consequence of Bézout's identity. Reducing both sides of the equation modulo r gives

$$n' = -n^{-1} \bmod r = \left|-n^{-1}\right|_r \tag{2.15}$$

The `MontMult` algorithm for multiplication of two variables in n-residue representation is given by Algorithm 5. Performing modular multiplication using this algorithm is only possible if the operands are converted to Montgomery form. After the calculation the operands need to be converted back to normal representation. Due to these pre- and post-calculation steps, Montgomery multiplication is not faster than regular modular multiplication. However, repeated usage of `MontMult` can result into a significant increase in speed.

Algorithm 5 Montgomery multiplication algorithm with reduction based on `REDC` [22]

Input: $r > n$, $\texttt{GCD}(n, r) = 1$, $n' = -n^{-1} \bmod r$
Output: $a \cdot b \cdot r^{-1} \bmod n$

```
1: procedure MontMult(a, b, n)
2:     t ← a · b
3:     q ← t · n' mod r
4:     u ← (t + q · n)/r
5:     if u ≥ n then
6:         u ← u − n
7:     return u
```

For example, let us calculate $7 \cdot 13 \bmod 23$ using Montgomery multiplication with $r = 32$. The first step is to calculate n' using Eq. (2.15).

$$n' = -n^{-1} \bmod r = -23^{-1} \bmod 32 = 9^{-1} \bmod 32 = 25$$

Next, operands a and b are converted to their n-residue representation using Eq. (2.13).

$$\begin{aligned} \bar{a} &= \texttt{MontMult}(a, \left|r^2\right|_n, n) = \texttt{MontMult}(7, 12, 23) &= 17 \\ \bar{b} &= \texttt{MontMult}(b, \left|r^2\right|_n, n) = \texttt{MontMult}(13, 12, 23) &= 2 \end{aligned}$$

Multiplication of $\bar{a}$ and $\bar{b}$ using Algorithm 5 gives answer $\bar{c}$.

$$\bar{c} = \texttt{MontMult}(\bar{a}, \bar{b}, n) = \texttt{MontMult}(17, 2, 23) = 14$$

Eq. (2.14) is used to convert $\bar{c}$ from Montgomery domain to default representation.

$$c = \texttt{MontMult}(\bar{c}, 1, n) = \texttt{MontMult}(14, 1, 23) = 22$$

The solution of $7 \cdot 13 \bmod 23 = 22$ is equal to c, which shows that the example calculation is correct. Note that for a system using a constant value for modulo n and radix r, the values for n' and "$r^2 \bmod n$" can be precomputed and stored into a lookup table.

2.5.2 Exponentiation

Exponentiation is basically repeated multiplication, which also holds for exponentiation in n-residue form. All methods from Section 2.3 can be used for Montgomery exponentiation.

Algorithm 6 Montgomery exponentiation is a combination of Algorithms 1 and 5.

```
1:  procedure MontExp(m, e, n)                    ▷ Calculate m^e mod n
2:      i ← MSB(e)                                ▷ Find the MSB position
3:      x̄ ← MontMult(1, |r^2|_n, n)               ▷ Initialize result x̄ with 1
4:      m̄ ← MontMult(m, |r^2|_n, n)               ▷ Convert m to n-residue form
5:      while i ≥ 0 do
6:          x̄ ← MontMult(x̄, x̄, n)                 ▷ Always square result x̄
7:          if e_i = 1 then                       ▷ If bit i in exponent e is set
8:              x̄ ← MontMult(m̄, x̄, n)             ▷ Also multiply result x̄ by m̄
9:          i ← i − 1
10:     x ← MontMult(x̄, 1, n)                     ▷ Convert x̄ out of n-residue form
11:     return x
```

The real advantage of Montgomery multiplication can be observed during modular exponentiation, as the cost of reduction is smaller than using standard division. Conversion to and from n-residue representation is done before and after exponentiation only. Using binary exponentiation combined with Montgomery multiplication results in Algorithm 6.

2.5.3 RNS modular reduction

Montgomery multiplication solves the problem of expensive division and reduction. But for large numbers having over a thousand bits, carry propagation becomes the bottleneck of the computational speed. RNS can be used to split large numbers into multiple smaller parts, reducing the impact of carry propagation. Posch et al. [23] show that Montgomery multiplication can be adapted to work with numbers in the RNS domain. Their approach enables Montgomery multiplication of large numbers.

Given the RNS base $\mathcal{B}$ with range M, Algorithm 5 requires some modifications before usage with RNS numbers is possible. The residue number system easily handles addition, subtraction and multiplication, but division, reduction and comparison are difficult. In weighted binary number systems, Montgomery multiplication uses a power of two as choice for radix r. For the RNS version it is better to choose $r = M$, as this makes reduction modulo r a costless operation. Division is also possible, because $u \leftarrow (t + vn)/r$ is an exact division that can be computed by multiplying with the multiplicative inverse M^{-1} when $r = M$.

However, base $\mathcal{B}$ represents numbers in the range $[0, M)$ only, so the value for M and thus for M^{-1} does not exist. The problem is solved by introducing a second RNS base $\widetilde{\mathcal{B}}$ with range $\widetilde{M}$ and representing M^{-1} in that domain instead. All moduli in $\widetilde{\mathcal{B}}$ should be pairwise prime with the moduli in $\mathcal{B}$. Calculation of ABN' mod M is done in base $\mathcal{B}$, but the result should be available in $\widetilde{\mathcal{B}}$. Conversion of a number from one RNS base to another is called a base extension, denoted by the function `bex`. There are different algorithms capable of performing a base extension and these will be discussed in Section 2.6. Adaption of Algorithm 5 for usage with RNS numbers results into the pseudocode shown in Algorithm 7.

Algorithm 7 General approach for performing RNS Montgomery multiplication

1: **procedure** RnsMontMult(A, B, N) ▷ Calculate ABM^{-1} mod N
2: $Q \leftarrow ABN'$ mod M ▷ Compute in RNS base $\mathcal{B}$
3: $\hat{Q} \leftarrow \texttt{bex}(Q)$ ▷ Base extension $\mathcal{B} \rightarrow \widetilde{\mathcal{B}}$
4: $\hat{R} \leftarrow (AB + \hat{Q}N)|M^{-1}|_{\widetilde{M}}$ ▷ Compute in RNS base $\widetilde{\mathcal{B}}$
5: $R \leftarrow \texttt{bex}(\hat{R})$ ▷ Base extension $\widetilde{\mathcal{B}} \rightarrow \mathcal{B}$
6: **return** R

The range of R is $[0, 2N)$, but repeated usage of the algorithm will remove multiples of N. This offset needs to be removed only in the last iteration, using a conditional subtraction. Also, the base extension methods suggested here provides an exact result R. Usage of `bex` algorithms that give an approximated result for R will have a different output range.

2.6 RNS base extension

Base extension is the process of converting one RNS representation to another. An easy implementation of this method would be to convert the RNS number of base $\mathcal{B}$ to binary representation and then convert that number to RNS base $\widetilde{\mathcal{B}}$. However, faster methods have been proposed without the need of converting to binary form. This section will discuss some of these methods.

2.6.1 Shenoy base extension

The Shenoy base extension is proposed in [24] and summarized by [25]. It is based on using the CRT to perform a base extension, in contrast to mixed radix conversion algorithms like the Szabo-Tanaka method [26]. The latter one will not be elaborated on, because it requires more computations than newer methods.

We consider $X = \{x_1, x_2, ..., x_k\}$ in RNS base $\mathcal{B} = \{m_1, ..., m_k\}$ with $X \in [0, M)$. The goal is to convert X to RNS base $\widetilde{\mathcal{B}} = \{m_{k+1}, ..., m_{2k}\}$ with residues $\{x_{k+1}, ..., x_{2k}\}$ and range $\widetilde{M}$. All moduli in $\widetilde{\mathcal{B}}$ should be pairwise prime to M ensuring the multiplicative inverses exist. Recall the CRT formula from Eq. (2.3), which can be rewritten for X as in Eq. (2.16).

$$\sum_{i=1}^{k} \left|x_i |M_i^{-1}|_{m_i}\right|_{m_i} M_i = X + \alpha M \tag{2.16}$$

Where $M_i = M/m_i$ and $0 \leq \alpha < k$. Now a redundant modulus m_r is introduced with properties $m_r \geq k$ and $\texttt{GCD}(m_r, M) = 1$. Moving X to the left side of the equation, reducing both sides modulo m_r and multiplying by the inverse of M with respect to m_r gives

$$\alpha = |\alpha|_{m_r} = \left| |M^{-1}|_{m_r} \left(\sum_{i=1}^{k} \left| \left|x_i |M_i^{-1}|_{m_i}\right|_{m_i} M_i \right|_{m_r} - |X|_{m_r} \right) \right|_{m_r} \tag{2.17}$$

The value of α is equal to $|\alpha|_{m_r}$, because m_r was chosen to be larger than or equal to k. Eq. (2.17) allows us to compute α, given that value of residue $|X|_{m_r}$ is known. Now it is possible to compute the residues x_j of RNS base $\widetilde{\mathcal{B}}$ for $j = k + 1...2k$.

$$x_j = |X|_{m_j} = \left| \sum_{i=1}^{k} \left| \left|x_i |M_i^{-1}|_{m_i}\right|_{m_i} M_i \right|_{m_j} - |\alpha M|_{m_j} \right|_{m_j} \tag{2.18}$$

Note that the constants $|M_i^{-1}|_{m_i}$, $|M_i|_{m_r}$, $|M^{-1}|_{m_r}$, $|M|_{m_j}$ and $|M_i|_{m_j}$ can be precomputed, as their values only depend on the moduli choices for $\mathcal{B}$, $\widetilde{\mathcal{B}}$ and redundant modulus m_r.

For clarification, the theory is illustrated with an example. The following is given:

$$X = \{x_1, x_2, x_3\} = \{2, 5, 10\}, \quad \mathcal{B} = \{m_1, m_2, m_3\} = \{5, 7, 13\}, \quad M = 455.$$

Here X represents the residues of the number 257 in RNS base $\mathcal{B}$ with range M. Also given is redundant modulus of $m_r = 4$ with residue $|X|_{m_r} = 1$. The goal is to transform X into RNS base $\widetilde{\mathcal{B}} = \{m_4, m_5, m_6\} = \{3, 11, 17\}$ without converting to binary representation.

The first step is to precompute the required constants for the calculation.

$$\begin{aligned}
|M_1^{-1}|_{m_1} &= |(455/5)^{-1}|_5 &&= |91^{-1}|_5 &&= 1\\
|M_2^{-1}|_{m_2} &= |(455/7)^{-1}|_7 &&= |65^{-1}|_7 &&= 4\\
|M_3^{-1}|_{m_3} &= |(455/13)^{-1}|_{13} &&= |35^{-1}|_{13} &&= 3
\end{aligned}$$

$$\begin{aligned}
|M_1|_{m_4} &= |455/5|_3 &&= |91|_3 &&= 1\\
|M_2|_{m_4} &= |455/7|_3 &&= |65|_3 &&= 2\\
|M_3|_{m_4} &= |455/13|_3 &&= |35|_3 &&= 2
\end{aligned}$$

$$\begin{aligned}
|M_1|_{m_5} &= |455/5|_{11} &&= |91|_{11} &&= 3\\
|M_2|_{m_5} &= |455/7|_{11} &&= |65|_{11} &&= 10\\
|M_3|_{m_5} &= |455/13|_{11} &&= |35|_{11} &&= 2
\end{aligned}$$

$$\begin{aligned}
|M_1|_{m_6} &= |455/5|_{17} &&= |91|_{17} &&= 6\\
|M_2|_{m_6} &= |455/7|_{17} &&= |65|_{17} &&= 14\\
|M_3|_{m_6} &= |455/13|_{17} &&= |35|_{17} &&= 1
\end{aligned}$$

$$\begin{aligned}
|M_1|_{m_r} &= |455/5|_4 &&= |91|_4 &&= 3\\
|M_2|_{m_r} &= |455/7|_4 &&= |65|_4 &&= 1\\
|M_3|_{m_r} &= |455/13|_4 &&= |35|_4 &&= 3
\end{aligned}$$

$$\begin{aligned}
|M|_{m_4} &= |455|_3 &&&&= 2\\
|M|_{m_5} &= |455|_{11} &&&&= 4\\
|M|_{m_6} &= |455|_{17} &&&&= 13
\end{aligned}$$

$$|M^{-1}|_{m_r} = |455^{-1}|_4 \qquad = 3$$

The value of α can now be computed using Eq. (2.17).

$$\begin{aligned}
\alpha &= \left| |M^{-1}|_{m_r} \left(\sum_{i=1}^{k} \left| |x_i|M_i^{-1}|_{m_i} \right|_{m_i} M_i \right|_{m_r} - |X|_{m_r} \right) \Bigg|_{m_r} \\
&= \left| 3 \left(\left| M_1 \left| x_1 |M_1^{-1}|_{m_1} \right|_{m_1} + M_2 \left| x_2 |M_2^{-1}|_{m_2} \right|_{m_2} + M_3 \left| x_3 |M_3^{-1}|_{m_3} \right|_{m_3} \right|_{m_r} - 1 \right) \right|_{m_r} \\
&= \left| 3 \left(\left| 3|2 \cdot 1|_5 + 1|5 \cdot 4|_7 + 3|10 \cdot 3|_{13} \right|_4 - 1 \right) \right|_4 \\
&= \left| 3 \left(\left| 6 + 6 + 12 \right|_4 - 1 \right) \right|_4 = 1
\end{aligned} \tag{2.19}$$

Next, substitution of $\alpha = 1$ in Eq. (2.18) allows to calculate x_j for $\widetilde{\mathcal{B}}$ using

$$\begin{aligned}
x_j &= \left| \sum_{i=1}^{k} \left| |x_i|M_i^{-1}|_{m_i} \right|_{m_i} M_i \right|_{m_j} - |M|_{m_j} \Bigg|_{m_j} \\
x_j &= \left| \left| M_1 \left| x_1 |M_1^{-1}|_{m_1} \right|_{m_1} + M_2 \left| x_2 |M_2^{-1}|_{m_2} \right|_{m_2} + M_3 \left| x_3 |M_3^{-1}|_{m_3} \right|_{m_3} \right|_{m_j} - |M|_{m_j} \right|_{m_j}
\end{aligned} \tag{2.20}$$

Substitution of $j = 4, 5, 6$ gives the new residues

$$\begin{aligned}
x_4 &= \left| \left| 1|2 \cdot 1|_5 + 2|5 \cdot 4|_7 + 2|10 \cdot 3|_{13} \right|_3 - 2 \right|_3 &&= \left| \left| 2 + 12 + 8 \right|_3 - 2 \right|_3 &&= 2 \\
x_5 &= \left| \left| 3|2 \cdot 1|_5 + 10|5 \cdot 4|_7 + 2|10 \cdot 3|_{13} \right|_{11} - 4 \right|_{11} &&= \left| \left| 6 + 60 + 8 \right|_{11} - 4 \right|_{11} &&= 4 \\
x_6 &= \left| \left| 6|2 \cdot 1|_5 + 14|5 \cdot 4|_7 + 1|10 \cdot 3|_{13} \right|_{17} - 13 \right|_{17} &&= \left| \left| 12 + 84 + 4 \right|_{17} - 13 \right|_{17} &&= 2
\end{aligned} \tag{2.21}$$

So the residues for X in RNS base $\widetilde{\mathcal{B}}$ are $X = \{2, 4, 2\}$.

2.6.2 Bajard base extension

The Bajard base extension [25, 27, 28] is also based on the CRT from Eq. (2.3), but allows an offset α in the resulting residue of RNS base $\widetilde{\mathcal{B}}$. For an exact result, this offset needs to be removed. However, when doing repeated Montgomery multiplication this is not necessary, given that the intermediate results are smaller than the range of $\widetilde{\mathcal{B}}$.

Suppose the Bajard algorithm is used as first base extension in the generic RNS Montgomery multiplication algorithm discussed in Section 2.5.3. The first base extension is used to transform Q from $\mathcal{B}$ to $\widetilde{\mathcal{B}}$. Allowing an offset in the result, gives a value $\hat{Q}$ related to Q.

$$\hat{Q} = \sum_{i=1}^{n} \left| q_i |M_i|_{m_i}^{-1} \right|_{m_i} M_i = Q + \alpha M \tag{2.22}$$

In RNS Montgomery multiplication variable R is calculated by $R = (T + QN)M^{-1}$. Let $\hat{R}$ denote the same formula, but now allow the extra offset from Eq. (2.22) to propagate here as well. This results in Eq. (2.23).

$$\hat{R} = (T + \hat{Q}N)M^{-1} = (T + QN + \alpha MN)M^{-1} = (T + QN)M^{-1} + \alpha N \tag{2.23}$$

It can be observed that the offset translates from a multiple of M to a multiple of N. Because of this translation, the relation $\hat{R} \equiv R \equiv ABM^{-1} \bmod N$ still holds. The Bajard base extension cannot be used as the second base extension in RNS Montgomery multiplication, because the introduced offset is a multiple of $\widetilde{M}$ and does not have a relation with N.

Residues resulting from the Bajard base extension are calculated similar to the Shenoy method, but the correction factor α is omitted. The residues can be calculated by computing

$$q_j = \left| \sum_{i=1}^{n} \left| q_i |M_i|_{m_i}^{-1} \right|_{m_i} M_i \right|_{m_j} \quad \text{for } i = 1...k \text{ and } j = k+1...2k \tag{2.24}$$

Note that the sum in the equation hints towards the use of an accumulator register in hardware. An implementation could start by initializing a register $t = 0$, adding a factor of $\sigma_i |M_i|_{m_j}$ and reducing it to modulo m_j each iteration. See Algorithm 8 for a summary of the Bajard base extension used to convert Q from $\mathcal{B}$ to $\widetilde{\mathcal{B}}$.

Algorithm 8 Bajard base extension (based on algorithm 2 from [27])

procedure BajardBex($q_1, ..., q_i$)
  $\sigma_i = q_i |M_i^{-1}|_{m_i} \bmod m_i$ ▷ In parallel for $i = 1...k$
  $t_0 = 0$ ▷ Initialize accumulator to zero
  for $i = 1...k$ **do** ▷ Accumulate values in t and reduce
    $t_i = (t_{i-1} + \sigma_i |M_i|_{m_j}) \bmod m_j$ ▷ In parallel for $j = k+1...2k$ and $j = r$
  $\hat{q}_j = t_k$ ▷ In parallel for $j = k+1...2k, r$

RNS Montgomery multiplication with the Bajard base extension first and the Shenoy base extension second is an interesting combination. The Bajard base extension can be used to generate the extra modulus m_r required for the Shenoy base extension. Repeated use of Montgomery multiplication with equal modulus N, reduces the offset of αN introduced by the Bajard extension. When using this algorithm for exponentiation, care should be taken that in the final step these extra multiples of N are removed.

2.6.3 Diophantine base extension

An alternative to a combination of the Bajard and Shenoy base extension is the Diophantine base extension [29]. This base extension is built on solving a linear Diophantine equation, which is a polynomial equation containing two unknowns and has integer solutions.

For a non-negative integer $X < M$ represented in RNS, having dynamic range $M = \prod_{i=1}^{k} m_i$, k prime moduli $\{m_1, m_2, ..., m_k\}$ and residues $\{x_1, x_2, ..., x_k\}$, it can be determined that

$$\sum_{i=1}^{k} \left(x_i \frac{m_{k+1}}{m_i} \left| (M_i m_{k+1})^{-1} \right|_{m_i} \right) = a + \frac{X}{M} \tag{2.25}$$

where a is a non-negative integer and $\frac{X}{M}$ is the leftover fraction in the range $[0, 1)$. Because a is an integer, its value is found by truncating both sides of Eq. (2.25), resulting in

$$a = \left\lfloor \sum_{i=1}^{k} \left(x_i \frac{m_{k+1}}{m_i} \left| (M_i m_{k+1})^{-1} \right|_{m_i} \right) \right\rfloor = \left\lfloor \sum_{i=1}^{k} x_i C_i \right\rfloor \tag{2.26}$$

In this equation, the constants $C_i = \frac{m_{k+1}}{m_i} \left| (M_i m_{k+1})^{-1} \right|_{m_i}$ are precalculated and stored as fixed-point radix-2 numbers into a lookup table. However, because the fixed-point representation has a finite precision, Eq. (2.26) should be modified to

$$\hat{a} = \left\lfloor \sum_{i=1}^{k} x_i \lfloor C_i \cdot 2^b \rfloor 2^{-b} \right\rfloor \tag{2.27}$$

where $b \in \mathbb{N}_0$ is the number of fractional bits in the vector for the constants. It can be said that $\hat{a}$ is an estimator of a. However, Eq. (2.27) requires a certain precision for the constants in order to give an equal result as a. The error introduced with the truncation scales with residues x_i as well. Performing truncation after an infinite number of fractional bits does not truncate anything, therefore it holds that

$$\lim_{b \to \infty} \hat{a} = \lim_{b \to \infty} \left\lfloor \sum_{i=1}^{k} x_i \lfloor C_i \cdot 2^b \rfloor 2^{-b} \right\rfloor = \left\lfloor \sum_{i=1}^{k} x_i C_i \right\rfloor = a \tag{2.28}$$

Eq. (2.28) shows that $\hat{a}$ approaches a for $b \to \infty$. For $b = 0$, Eq. (2.27) simplifies to

$$\hat{a}\Big|_{b=0} = \left\lfloor \sum_{i=1}^{k} x_i \lfloor C_i \rfloor \right\rfloor \tag{2.29}$$

It should be obvious that Eq. (2.29) is smaller than or equal to a, because of the extra truncation inside the sum. Combining this with Eq. (2.28) creates an upper bound on $\hat{a}$ which is equal to $\hat{a} \leq a$. This means that $\hat{a}$ approaches a from below for larger values of b.

In order to perform a correct base extension a and $\hat{a}$ must give an equal result. This can be achieved by taking enough fractional bits b, but what is the minimum number of bits required? Unfortunately, both equations contain the truncation operator and that makes finding a solution using algebra impossible. Instead, an iterative approach is used.

First, all constants C_i are computed using a very high precision setting. These values are used to determine the expected value for a using $X = 1$. Next, the constants are truncated to zero fractional bits ($b = 0$) and a is computed again, now denoted by $\hat{a}$. When a and $\hat{a}$ are equal, the minimum number of fractional bits b is found. If those are different, the number of fractional bits is increased by one and the process is repeated.

For example, let us take an RNS base B with moduli $m_i = \{7, 13, 19, 29\}$. The goal is to perform a base extension of $X = 1$ to modulo $m_{k+1} = 8$, the expected result is $|X|_{m_{k+1}} = 1$. High-precision calculation of the constants C_i gives the results in Eq. (2.30).

$$\begin{aligned}
C_1 &= 8/7 \cdot 4 &&= 4.571428571428571428571428571428571428571428571428..\\
C_2 &= 8/13 \cdot 2 &&= 1.230769230769230769230769230769230769230769230769..\\
C_3 &= 8/19 \cdot 13 &&= 5.473684210526315789473684210526315789473684210526..\\
C_4 &= 8/29 \cdot 28 &&= 7.724137931034482758620689655172413793103448275862..
\end{aligned} \tag{2.30}$$

Given $X = 1$, all four residues are $x_i = 1$ and calculation of a is a summation of all C_i.

$$a = \left\lfloor \sum_{i=1}^{4} x_i C_i \right\rfloor = \lfloor C_1 + C_2 + C_3 + C_4 \rfloor = \lfloor 19.0000199437586.. \rfloor = 19 \tag{2.31}$$

Next, an implementation of the Diophantine base-extension is made in `C++` with the GNU Multiple Precision Arithmetic Library (GMP). The program first computes the constants C_i using a very high precision to calculate the ground truth of a. Then the minimum number of fractional bits b required for $\hat{a}$ to correctly perform the base extension is determined. The version numbers of software used to create the program are given in Table 2.6.

Software	Type	Version	License
GCC	Compiler	7.3.0	Modified GPLv3
GMP	Library	6.1.2	LGPLv3 and GPLv2

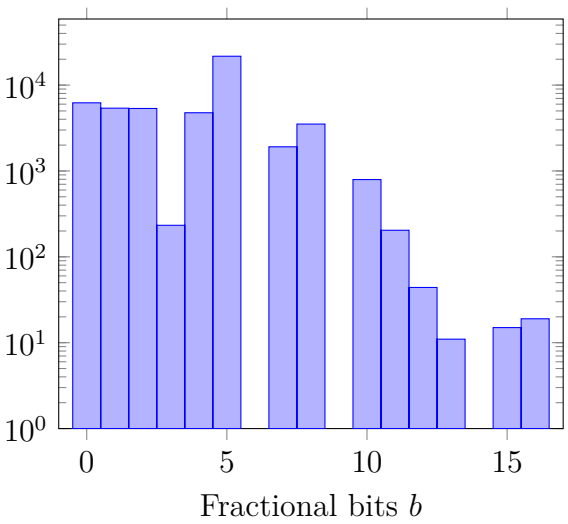
Table 2.6: Version numbers of the software used to compile the program.

Using an iterative approach for increasing the number of fractional bits b in the constants C_i, the results shown in Table 2.7 are obtained. It shows that to perform correct base extensions for $X = 1$ from RNS base B to $m_{k+1} = 8$, a minimum of 16 fractional bits is required.

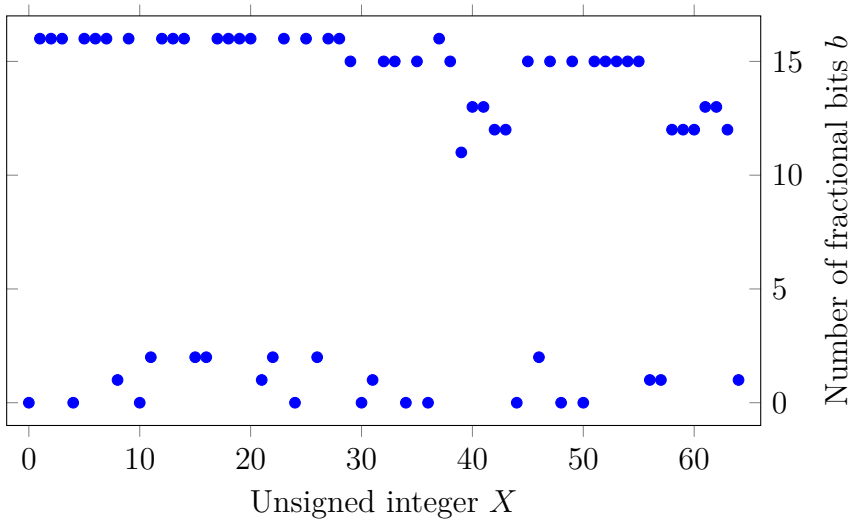
b	x_1C_1	x_2C_2	x_3C_3	x_4C_4	Sum
0	4	1	5	7	17
1	4.5	1	5	7.5	18.0
2	4.5	1	5.25	7.5	18.25
3	4.5	1.125	5.375	7.625	18.625
4	4.5625	1.1875	5.4375	7.6875	18.8750
5	4.5625	1.21875	5.46875	7.71875	18.96875
6	4.5625	1.21875	5.46875	7.71875	18.96875
7	4.5703125	1.2265625	5.46875	7.71875	18.9843750
8	4.5703125	1.23046875	5.47265625	7.72265625	18.99609375
9	4.5703125	1.23046875	5.47265625	7.72265625	18.99609375
10	4.5712890625	1.23046875	5.4736328125	7.7236328125	18.9990234375
11	4.5712890625	1.23046875	5.4736328125	7.72412109375	18.99951171875
12	4.5712890625	1.230712890625	5.4736328125	7.72412109375	18.999755859375
13	4.5714111328125	1.230712890625	5.4736328125	7.72412109375	18.9998779296875
14	4.5714111328125	1.230712890625	5.4736328125	7.72412109375	18.9998779296875
15	4.5714111328125	1.230743408203125	5.473663330078125	7.72412109375	18.999938964843750
16	4.5714263916015625	1.2307586669921875	5.4736785888671875	7.7241363525390625	19.0000000000000000

Table 2.7: Each iteration step the number of fractional bits b is increased until $\hat{a} = 19$ is reached.

Table 2.7 only shows the results for the conversion of $X = 1$ from RNS base B to modulo 8. However, dynamic range $M = 7 \cdot 13 \cdot 19 \cdot 29 = 50141$, so there are more possible values of X that need to be computed to find the actual minimum number of fractional bits. The program is modified so that it is able to find the required number of fractional bits for all possible values of X. An overview of the results is given by Fig. 2.3a and a more detailed close-up of the range $0 \leq X \leq 64$ is shown by Fig. 2.3b.



(a) Histogram of the minimum number of fractional bits b for all values $X < M$.



(b) Number of fractional bits b for correct base extension to modulo 8 for $X \leq 64$ showing all occurrences of $b = 16$.

Figure 2.3: Overview of running the `C++` program for a base extension from $B = \{7, 13, 19, 29\}$ to modulo 8.

Fig. 2.3a shows that using 16 fractional bits is sufficient for all values of X. In the histogram there exist no numbers that require 6, 9 or 14 fractional bits. This makes sense, as from Table 2.7 it follows that increasing the number of bits to these numbers results in equal values for C_i. Looking at the outline of Fig. 2.3b, it seems that for larger values of X less fractional bits are needed to ensure a correct base extension.

In the previous example, the required number of fractional bits b for all X is equal to the number of bits required for converting $X = 1$. From Fig. 2.3b it follows that the largest requirements for b are needed for extending the smallest values of X to the new modulus. It would make sense to assume that computing b for $X = 1$ also gives enough fractional bits ensuring an error-free base extension for all values of X, but that has never been proven.

Usability of the Diophantine base extension for implementation of 4096-bit RSA depends on the number of bits required for the values of C_i. If these are too large, it will slow down the computation instead of giving better performance. The example RNS bases to perform benchmarks on are taken from [29] and shown in Table 2.8 for further analysis.

4 moduli		**5 moduli**		**6 moduli**	
Base B	Base B'	Base B	Base B'	Base B	Base B'
$2^{512} - 2^{10} - 1$	$2^{512} - 2^{22} - 1$	$2^{512} - 2^{10} - 1$	$2^{512} - 2^{8} + 1$	$2^{512} - 2^{10} - 1$	$2^{512} - 2^{5} - 1$
$2^{512} - 2^{19} - 1$	$2^{512} - 2^{22} + 1$	$2^{512} - 2^{19} - 1$	$2^{512} - 2^{16} - 1$	$2^{512} - 2^{16} - 1$	$2^{512} - 2^{17} - 1$
$2^{512} - 2^{28} - 1$	$2^{512} - 2^{23} - 1$	$2^{512} - 2^{20} - 1$	$2^{512} - 2^{17} - 1$	$2^{512} - 2^{19} - 1$	$2^{512} - 2^{18} + 1$
$2^{512} - 1$	2^{512}	$2^{512} - 2^{28} - 1$	$2^{512} - 2^{17} + 1$	$2^{512} - 2^{20} - 1$	$2^{512} - 2^{25} - 1$
		2^{512}	$2^{512} - 2^{22} - 1$	$2^{512} - 2^{28} - 1$	$2^{512} - 2^{26} - 1$
				2^{512}	$2^{512} - 1$

Table 2.8: Three sets of moduli in table 5.1 from [29] revised with values of table 1 from [30].

The moduli sets from Table 2.8 are used as input for the `C++` program that calculates the minimum number of fractional bits. However, as the dynamic range M is a number of more than 2000 bits, the required number of bits cannot be computed for all possible values of X. So based on the assumption that computation of $X = 1$ gives an indication of how many bits are required, that value is used instead. These results are shown in Table 2.9. In short, the number of fractional bits is too large for an implementation based on fixed-point constants.

Moduli set	**Base extension**	**Fractional bits**
4 moduli	$B \to B'$	2050
4 moduli	$B' \to B$	2051
5 moduli	$B \to B'$	2563
5 moduli	$B' \to B$	2561
6 moduli	$B \to B'$	3074
6 moduli	$B' \to B$	3073

Table 2.9: Minimum number of fractional bits b to perform an error-free base extension of $X = 1$.

3 Design considerations

This section will discuss the design choices made for the implementation of the cryptographic processor. The goal of the design is to create a high-performance implementation of RSA encryption and decryption, but also to perform optimizations for reducing the area usage on the FPGA. The overall result will be a balanced design having good performance.

3.1 Multiplication hardware

Section 2.5 discusses RNS Montgomery multiplication and Section 2.6 is about RNS base extensions. For the implementation, these topics have to be merged into one algorithm which can be used for one round of RNS Montgomery multiplication. Combining the generic Montgomery multiplication algorithm with the Bajard and Shenoy base extensions results in Algorithm 9. The algorithm is derived by merging algorithms 1, 2 and 3 from [27].

Algorithm 9 RNS Montgomery Multiplication using Bajard and Shenoy base extensions

Input: Two RNS bases $\mathcal{B} = \{m_1, ..., m_k\}$, $\widetilde{\mathcal{B}} = \{m_{k+1}, ..., m_{2k}\}$ with range $M = \prod_{i=1}^{k} m_i$, $\widetilde{M} = \prod_{j=1}^{k} m_{k+j}$. Redundant modulus $m_r \geq k$ with $\texttt{GCD}(M, \widetilde{M}, m_r) = 1$. Multiplicands $A = \{a_1, ..., a_{2k}, a_r\}$, $B = \{b_1, ..., b_{2k}, b_r\}$ and modulo $N = \{n_1, ..., n_{2k}, n_r\}$ in $\{\mathcal{B}, \widetilde{\mathcal{B}}, m_r\}$.

Output: $\hat{R} = ABM^{-1} \bmod N = R + (2 + \alpha)N$ with $\alpha < k$.

1: **procedure** RnsMontMult(A, B, N)
2: $q_i = a_i b_i |-n_i^{-1}|_{m_i}$ ▷ In parallel for $i = 1...k$
3: $\sigma_i = q_i |M_i^{-1}|_{m_i} \bmod m_i$ ▷ In parallel for $i = 1...k$
4: $t = 0$ ▷ Initialize accumulator to zero
5: **for** $i = 1...k$ **do** ▷ Accumulate in t and reduce
6: $t = (t + \sigma_i |M_i|_{m_j}) \bmod m_j$ ▷ In parallel for $j = k + 1...2k, r$
7: $\hat{r}_j = (a_j b_j + t n_j) |M^{-1}|_{m_j} \bmod m_j$ ▷ In parallel for $j = k + 1...2k, r$
8: $\xi_j = \hat{r}_j |\widetilde{M}_j^{-1}|_{m_j} \bmod m_j$ ▷ In parallel for $j = k + 1...2k$
9: $t = 0$ ▷ Set accumulator to zero
10: **for** $j = 1...k$ **do** ▷ Accumulate in t and reduce
11: $t = (t + \xi_{k+j} |\widetilde{M}_j|_{m_r}) \bmod m_r$
12: $\beta = |\widetilde{M}^{-1}|_{m_r} (t - |\hat{r}|_{m_r}) \bmod m_r$ ▷ Calculate correction factor β
13: $t = 0$ ▷ Set accumulator to zero
14: **for** $j = 1...k$ **do** ▷ Accumulate in t and reduce
15: $t = (t + \xi_{k+j} |\widetilde{M}_j|_{m_i}) \bmod m_i$ ▷ In parallel for $i = 1...k$
16: $\hat{r}_i = (t - |\beta \widetilde{M}|_{m_i}) \bmod m_i$ ▷ In parallel for $i = 1...k$
17: **return** $\hat{R}$ ▷ Return residues $\{\mathcal{B}, \widetilde{\mathcal{B}}\}$

Exponentiation essentially is repeated multiplication, so to accelerate RSA encryption and decryption the multiplication algorithm should be implemented in hardware. Analysis of Algorithm 9 gives some hints useful for mapping the calculations to a FPGA platform.

It should be noted that most of the statements are computed in parallel for one of the moduli m_i with $i = \{1, ..., k\}$ or m_j with $j = \{k+1, ..., 2k\}$. Usage of RNS in hardware suggests the design of k independent cores equal to the number of moduli in the RNS base. Each of the RNS cores is then responsible for one residue and all its arithmetic operations are followed by reduction with a modulo constant m_i. After performing the base extension $\mathcal{B} \rightarrow \widetilde{\mathcal{B}}$, the core is still responsible for one residue, but now has to reduce modulo m_j. So by enabling each core to perform arithmetic operations followed by reduction, using either m_i or m_j, allows execution of Algorithm 9.

Also interesting is that the base extension algorithms require a lot of multiply-accumulate (MAC) operations. These are found in the `for` loops on lines 6, 11 and 15. Modern FPGAs contain Digital Signal Processing (DSP) blocks that are able to perform MAC operations at high frequencies. Creating a design that focuses on usage of the available DSP blocks should increase the performance significantly. By using the accumulator register inside the DSP block to store the intermediate values t, the reduction can be postponed until accumulation has completed. Downside of this approach is that the size of the accumulate register now places an upper bound on the bit width of the moduli.

For the computation of redundant residue $|r|_{m_r}$ an extra core can be added that only is active between the two base extensions. The Bajard base extension produces the redundant residue and the Shenoy base extension consumes it. Correction constant β is produced by the redundant RNS core and should be broadcasted to all other RNS cores as they need it to complete the last step of the algorithm. Assigning the redundant residue to its own core, also helps to increase the amount of parallelism. The loops on lines 11 and 15 of Algorithm 9 are now computed in parallel as they each use different moduli constants.

3.2 Datapath dimensions

Each RNS core is a simple processor that can perform addition, subtraction and multiplication. But how many cores are needed to perform 4096-bit RSA encryption depends on the number of moduli in the RNS base. As each RNS core is responsible for one modulus, the main datapaths can be equally sized as the size of that modulus in bits. The mapping on DSP blocks is important for the performance and these blocks have fixed operand sizes. One Xilinx DSP48E2 block can perform a multiplication with two 17-bit unsigned numbers [31]. Using a datapath containing a multiple of seventeen bits is preferred, so the DSP block multiplier ports are completely utilized.

Usage of RNS cores with a 17-bit wide datapath requires a minimum of $\lceil 4096/17 \rceil = 241$ cores in the system. Algorithm 9 shows that there are a lot of multiplications to perform. Performing a base extension means iterating over the number of cores in the system, values

σ_i and ξ_j are needed at all cores once per Montgomery multiplication. Reducing the number of cores could result in a faster base extension, however resource usage per core will increase.

Increasing the datapath width to 34-bit will require a minimum of $\lceil 4096/34 \rceil = 121$ cores. As the DSP blocks have operands of 17-bit each, now four of these blocks are needed to perform one multiplication. The number of cores is halved, but the number of DSP blocks needed per core is four times as high. So for a design utilizing 34-bit datapaths $121 \cdot 4 = 484$ DSP blocks are needed. Whether that is a significant amount depends on the available FPGA platform. This number can be reduced by reusing one multiplier to perform the multiplication in four parts. However, that would result in a multiplication taking multiple clock cycles to complete. As multiplication is used very often, this is not a good trade-off.

Further increasing the datapath to 51 bits would require a design with $\lceil 4096/51 \rceil = 81$ cores. A multiplication of two 51-bit operands using 17-bit multipliers requires $(51/17)^2 = 9$ DSP blocks. Which results in a total multiplier usage of $81 \cdot 9 = 729$ DSP blocks. Increasing the datapath width any more would need so many DSP blocks that it is not feasible. In short, there are three datapaths widths that are feasible for further analysis and they are summarized in Table 3.1. Note that the estimated resource usage for the redundant core processing the residue $|\hat{r}|_{m_r}$ is not included here.

Datapath width (bits)	Minimum cores	Total DSP blocks
17	241	241
34	121	484
51	81	729

Table 3.1: Overview of feasible datapath widths for a RNS core.

3.3 RNS moduli selection

An important choice is the selection of moduli m_i and m_j for RNS bases $\mathcal{B}$ and $\widetilde{\mathcal{B}}$. 4096-bit RSA encryption means that the plain-text message, modulus and exponent have a size of up to 4096 bits. The RNS representation should be able to fit a number of at least an equal size. So the product of the moduli m_i giving range M should have at least a value of 2^{4096}.

The maximum number of RNS cores is limited by how many primes are available for bases $\mathcal{B}$ and $\widetilde{\mathcal{B}}$. To prevent creating unbalanced logic, the selected prime number for moduli m_i and m_j should contain an equal number of bits as the datapath of the RNS core. The number of primes less then or equal to a number x is given by the prime counting function $\pi(x)$. Note that the number 2^x has a width of $x + 1$ bits, so to set an upper bound $2^x - 1$ is needed. The number of primes between two powers of two therefore is given by Eq. (3.1).

$$\rho_{available} = \pi(2^{x+1} - 1) - \pi(2^x) \tag{3.1}$$

where x is the number of bits in the RNS moduli as well as the datapath of the RNS cores. Now this can be compared to the number of primes needed to create RNS bases $\mathcal{B}$ and $\widetilde{\mathcal{B}}$ for 4096-bit RSA. In the previous paragraph, it was stated that M should be at least 4096 bits. However, in Algorithm 9 intermediate results need a representation with a larger number of bits. Also important is that the algorithm will be used for exponentiation, so result value $\hat{R}$ will be used as inputs A or B in following iterations. The range of M is bounded by

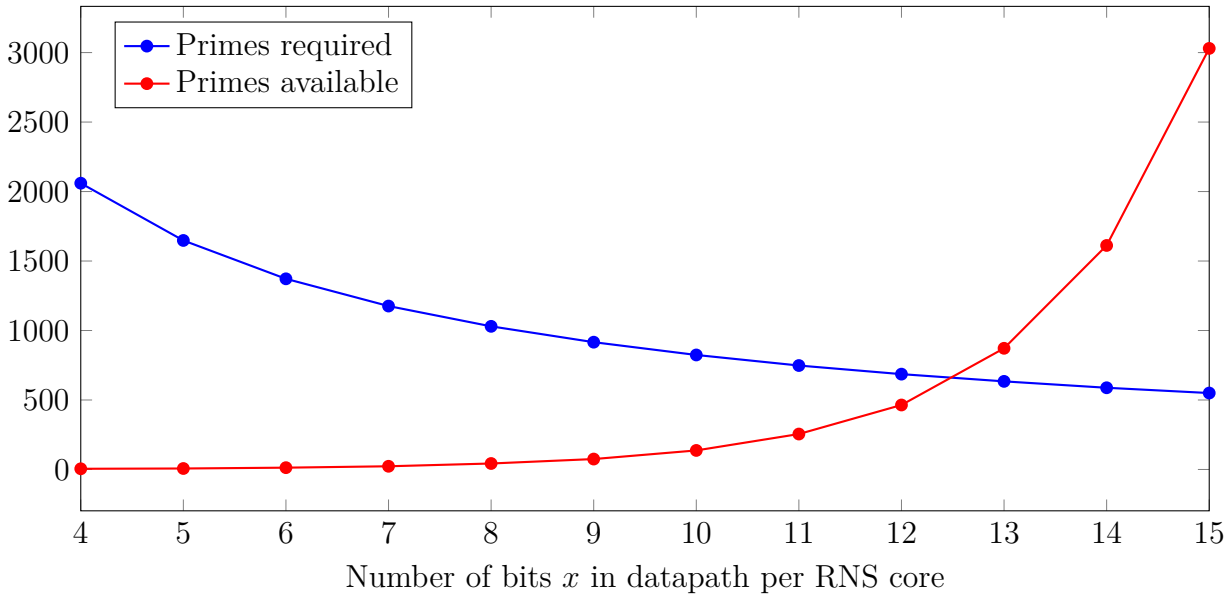
$$(k+2)^2 N < M \tag{3.2}$$

where k is the number of moduli in base $\mathcal{B}$ and N is the maximum value of the RSA modulus [27]. As N has a maximum size of 4096 bits, Eq. (3.2) shows that the boundary condition can be satisfied by adding $\lceil \log_2((k+2)^2 \rceil$ bits to M. The number of moduli k can be calculated by dividing the size of modulus N by the number of bits per modulo x, which gives $k = \lceil 4096/x \rceil$. Each RNS core processes two moduli and that doubles the number of primes needed, so the constraint on the minimum number of primes is now given by

$$\rho_{required} = 2\lceil (4096 + \lceil \log_2((\lceil 4096/x \rceil + 2)^2) \rceil)/x \rceil \tag{3.3}$$

A comparison of Eqs. (3.1) and (3.3) for different datapath widths x results in Fig. 3.1. The figure shows that the smallest possible datapath width is equal to 13 bits. For example if the datapath width would be 12 bits, there exist not enough primes of 12 bits to construct two RNS bases large enough to perform 4096-bit RSA encryption.

Figure 3.1: Available and required number of primes for 4096-bit RSA.



The previous section discussed three possible datapath widths: 17, 34 and 51 bits, which are multiples of the Xilinx DSP block input operands. All these options are larger than 13 bits, so they can be used to design a RSA processor based on RNS Montgomery multiplication. Each of these three options poses a trade-off in area usage and number of cycles to complete the computation. A small datapath results in a lot of RNS cores, so additional execution time is spent on performing base extensions. Using a wide datapath adds more overhead in the computations that are not the base extension, because the carry propagation path becomes longer. Therefore the choice is made to create an implementation using moduli of 34 bits, as this should give a good trade-off between both situations.

Previous research suggests using a small set of optimal RNS bases [30]. However, this only gives RNS bases consisting of at most six primes, which would require large datapaths for 4096 bits encryption. Another option is using pseudo Meresenne primes, which are close to a power of two and make reduction easy to implement in logic. Still the same problem arises, that not enough of these primes exist to create an implementation of $\lceil 4096/34 \rceil = 121$ cores.

From Eq. (3.1) it follows that for $x = 34$ there are $\rho = 717267168$ primes available. Taking the largest primes from this list should make is easier to satisfy the bound in M from Eq. (3.2). As the prime numbers chosen are close to each other and have equal bit widths, both RNS bases can be considered balanced. The sets of 34 bits prime numbers for $\mathcal{B}$ and $\widetilde{\mathcal{B}}$ used for implementation can be found in Appendices A.1 and A.2. Reduction by a prime number is an expensive operation. To reduce the impact, lazy reduction of the DSP accumulator is performed. This means that reduction is started after completion of the accumulation phase. As long as the DSP accumulation register is wide enough to support the required number of iterations during the base extension, it will be possible.

3.4 RNS processor core

Analysis of Algorithm 9 shows that each RNS core should be able to perform: addition, subtraction and multiplication. Of those operations the result should be reduced using one of two moduli, one from RNS base $\mathcal{B}$ and one from RNS base $\widetilde{\mathcal{B}}$. So an additional input bit is needed to select which reduction should be applied. Reduction of the multiplication result needs more computations, because the result is larger than that of addition or subtraction.

It is assumed that the operands on which arithmetic is performed, are already within the bounds of the selected moduli set. So reduction for addition and subtraction is just one compare and corrective subtraction or addition. For the reduction of the multiplication result an extra block of logic is needed. Design of this block is inspired by the theory discussed in Section 2.4. The operand size for this block is determined by the size of the accumulation register in the multiplication stage. One multiplication result contains twice the number of bits as the 34-bit operands and thus has a width of 68 bits. For $k = 121$ cores, one base extension loop executes an equal number of multiply-accumulate operations. So the size of the accumulation register should be at least $\lceil \log_2(121) \rceil + 68 = 75$ bits.

Each core should be able to use precalculated constants as they are needed to perform the base extension. Also some storage is needed to save intermediate results while performing the Montgomery multiplication algorithm. The solution is to use a large register file with a read-only part to store the constants. For example, this could be implemented using Block Random Access Memory (BRAM) logic found in Xilinx FGPAs [32].

Not all operations found in Algorithm 9 can be executed in parallel on multiple RNS cores. During the base extension methods, the values for σ_i and ξ_i are needed on every RNS core. Most of the processing time is consumed in the loops for the base extension, so it seems that this could cause performance issues. However, inspired by the architecture from [33], a solution is found. Although all σ_i and ξ_j are needed at every core, they are not needed in exactly the same sequence. The solution is to connect the RNS cores in a circle, so each core has one neighbor to send data to and one neighbor to consume data from.

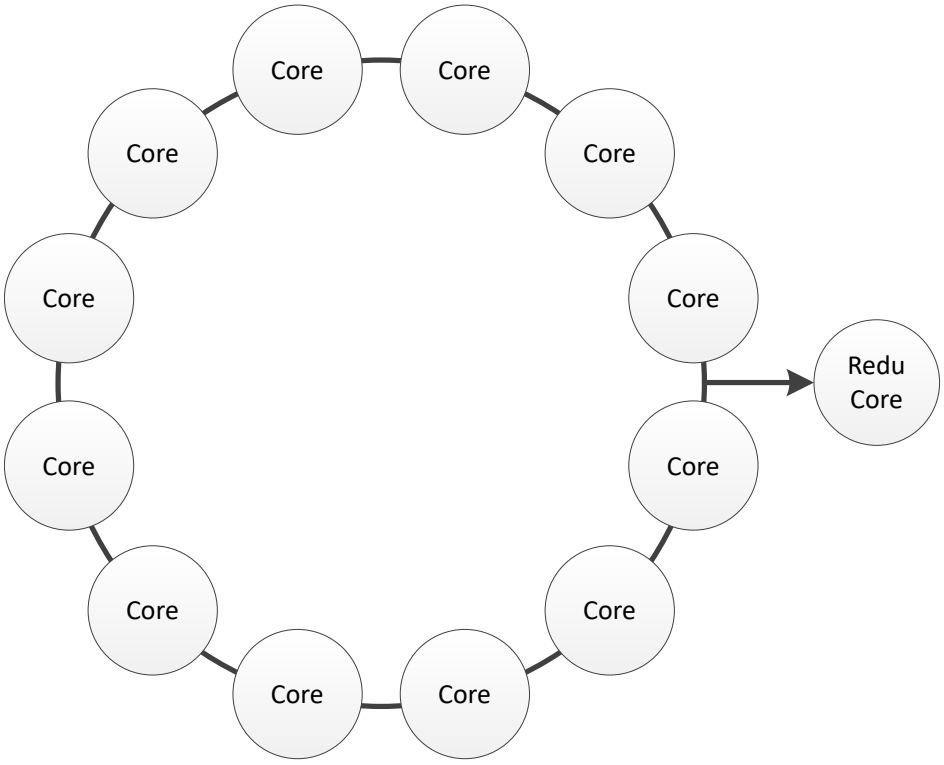


Figure 3.2: Each RNS core is connected to a neighbor in clockwise direction.

By sending these values from one neighbor to the other, nobody has to wait for data and computation of t can be done in parallel for both base extensions. The redundant core also needs σ_i and ξ_i for the computation of correction factor β. By adding the redundant core as leaf connection, it can now grab these values from the ring. The value for β can than be calculated in parallel while all other cores are busy doing their computation of t. The connections between the RNS cores and the redundant core are shown in Fig. 3.2.

For example, suppose we have a RNS processor with four cores. Efficient use of the ring structure is done by scheduling the RNS Montgomery multiplication as shown in Table 3.2.

#	**Core 1**	**Core 2**	**Core 3**	**Core 4**	**Redundant core**
1	$c = \|a_1 b_1\|_{m_1}$	$c = \|a_2 b_2\|_{m_2}$	$c = \|a_3 b_3\|_{m_3}$	$\|a_4 b_4\|_{m_4}$	
2	$q_1 = \|c\|{-}n_1^{-1}\|_{m_1}\|_{m_1}$	$q_2 = \|c\|{-}n_2^{-1}\|_{m_2}\|_{m_2}$	$q_3 = \|c\|{-}n_3^{-1}\|_{m_3}\|_{m_3}$	$q_4 = \|c\|{-}n_4^{-1}\|_{m_4}\|_{m_4}$	
3	$\sigma_1 = \|q_1 \|M_1^{-1}\|_{m_1}\|_{m_1}$	$\sigma_2 = \|q_2 \|M_2^{-1}\|_{m_2}\|_{m_2}$	$\sigma_3 = \|q_3 \|M_3^{-1}\|_{m_3}\|_{m_3}$	$\sigma_4 = \|q_4 \|M_4^{-1}\|_{m_4}\|_{m_4}$	
4	$t = 0$	$t = 0$	$t = 0$	$t = 0$	$t = 0$
5	$t = t + \sigma_1 \|M_1\|_{m_5}$	$t = t + \sigma_2 \|M_2\|_{m_6}$	$t = t + \sigma_3 \|M_3\|_{m_7}$	$t = t + \sigma_4 \|M_4\|_{m_8}$	$t = t + \sigma_1 \|M_1\|_{m_r}$
6	$t = t + \sigma_2 \|M_2\|_{m_5}$	$t = t + \sigma_3 \|M_3\|_{m_6}$	$t = t + \sigma_4 \|M_4\|_{m_7}$	$t = t + \sigma_1 \|M_1\|_{m_8}$	$t = t + \sigma_2 \|M_2\|_{m_r}$
7	$t = t + \sigma_3 \|M_3\|_{m_5}$	$t = t + \sigma_4 \|M_4\|_{m_6}$	$t = t + \sigma_1 \|M_1\|_{m_7}$	$t = t + \sigma_2 \|M_2\|_{m_8}$	$t = t + \sigma_3 \|M_3\|_{m_r}$
8	$t = t + \sigma_4 \|M_4\|_{m_5}$	$t = t + \sigma_1 \|M_1\|_{m_6}$	$t = t + \sigma_2 \|M_2\|_{m_7}$	$t = t + \sigma_3 \|M_3\|_{m_8}$	$t = t + \sigma_4 \|M_4\|_{m_r}$
9	$\hat{q}_5 = \|t\|_{m_5}$	$\hat{q}_6 = \|t\|_{m_6}$	$\hat{q}_7 = \|t\|_{m_7}$	$\hat{q}_8 = \|t\|_{m_8}$	$\hat{q}_r = \|t\|_{m_r}$
10	$d = \|a_5 b_5\|_{m_5}$	$d = \|a_6 b_6\|_{m_6}$	$d = \|a_7 b_7\|_{m_7}$	$d = \|a_8 b_8\|_{m_8}$	$d = \|a_r b_r\|_{m_r}$
11	$e = \|\hat{q}_5 n_5\|_{m_5}$	$e = \|\hat{q}_6 n_6\|_{m_6}$	$e = \|\hat{q}_7 n_7\|_{m_7}$	$e = \|\hat{q}_8 n_8\|_{m_8}$	$e = \|\hat{q}_r n_r\|_{m_r}$
12	$f = \|d + e\|_{m_5}$	$f = \|d + e\|_{m_6}$	$f = \|d + e\|_{m_7}$	$f = \|d + e\|_{m_8}$	$f = \|d + e\|_{m_r}$
13	$\hat{r}_5 = \|f \|M^{-1}\|_{m_5}\|_{m_5}$	$\hat{r}_6 = \|f \|M^{-1}\|_{m_6}\|_{m_6}$	$\hat{r}_7 = \|f \|M^{-1}\|_{m_7}\|_{m_7}$	$\hat{r}_8 = \|f \|M^{-1}\|_{m_8}\|_{m_8}$	$\hat{r}_r = \|f \|M^{-1}\|_{m_r}\|_{m_r}$
14	$\xi_5 = \|\hat{r}_5 \|\widetilde{M}_5^{-1}\|_{m_5}\|_{m_5}$	$\xi_6 = \|\hat{r}_6 \|\widetilde{M}_6^{-1}\|_{m_6}\|_{m_6}$	$\xi_7 = \|\hat{r}_7 \|\widetilde{M}_7^{-1}\|_{m_7}\|_{m_7}$	$\xi_8 = \|\hat{r}_8 \|\widetilde{M}_8^{-1}\|_{m_8}\|_{m_8}$	
15	$t = 0$	$t = 0$	$t = 0$	$t = 0$	$t = 0$
16	$t = t + \xi_5 \|\widetilde{M}_5\|_{m_1}$	$t = t + \xi_6 \|\widetilde{M}_6\|_{m_2}$	$t = t + \xi_7 \|\widetilde{M}_7\|_{m_3}$	$t = t + \xi_8 \|\widetilde{M}_8\|_{m_4}$	$t = t + \xi_5 \|\widetilde{M}_5\|_{m_r}$
17	$t = t + \xi_6 \|\widetilde{M}_6\|_{m_1}$	$t = t + \xi_7 \|\widetilde{M}_7\|_{m_2}$	$t = t + \xi_8 \|\widetilde{M}_8\|_{m_3}$	$t = t + \xi_5 \|\widetilde{M}_5\|_{m_4}$	$t = t + \xi_6 \|\widetilde{M}_6\|_{m_r}$
18	$t = t + \xi_7 \|\widetilde{M}_7\|_{m_1}$	$t = t + \xi_8 \|\widetilde{M}_8\|_{m_2}$	$t = t + \xi_5 \|\widetilde{M}_5\|_{m_3}$	$t = t + \xi_6 \|\widetilde{M}_6\|_{m_4}$	$t = t + \xi_7 \|\widetilde{M}_7\|_{m_r}$
19	$t = t + \xi_8 \|\widetilde{M}_8\|_{m_1}$	$t = t + \xi_5 \|\widetilde{M}_5\|_{m_2}$	$t = t + \xi_6 \|\widetilde{M}_6\|_{m_3}$	$t = t + \xi_7 \|\widetilde{M}_7\|_{m_4}$	$t = t + \xi_8 \|\widetilde{M}_8\|_{m_r}$
20	$g = \|t\|_{m_1}$	$g = \|t\|_{m_2}$	$g = \|t\|_{m_3}$	$g = \|t\|_{m_4}$	$g = \|t\|_{m_r}$
21					$h = \|g - \hat{r}_r\|_{m_r}$
22					$\beta = \|h \|\widetilde{M}^{-1}\|_{m_r}\|_{m_r}$
23	$h = \|\beta \|\widetilde{M}\|_{m_1}\|_{m_1}$	$h = \|\beta \|\widetilde{M}\|_{m_2}\|_{m_2}$	$h = \|\beta \|\widetilde{M}\|_{m_3}\|_{m_3}$	$h = \|\beta \|\widetilde{M}\|_{m_4}\|_{m_4}$	
24	$\hat{r}_1 = \|g - h\|_{m_1}$	$\hat{r}_2 = \|g - h\|_{m_2}$	$\hat{r}_3 = \|g - h\|_{m_3}$	$\hat{r}_4 = \|g - h\|_{m_4}$	

Table 3.2: Scheduling of RNS Montgomery multiplication on four RNS cores.

Each RNS core is equipped with an additional register called the `propagate` register. It is not located in the register file, but used as input register to buffer data from its neighbor RNS core. A multiplexer is used to connect either one port of the register file or the `propagate` register to the multiplier. The `propagate` register is used only during the base extension process to propagate the values for σ_i and ξ_j. An additional multiplexer is used to select which source is connected to next neighbor: an output port from the register file or the `propagate` register. It is now possible to pipeline the multiply-accumulate operation.

Table 3.2 shows that near the end of the algorithm the RNS cores are waiting on the calculation of β. To speed up this computation, redundant modulus m_r can be a power of two which makes reduction equal to dropping bits. For $k = 121$ cores, a redundant modulus of $m_r = 128$ is chosen to satisfy the boundary condition $m_r \geq k$ in Algorithm 9.

3.5 Memory requirements

Each RNS core has its own register file, as it needs to work independently of the other cores. The number of registers required to perform RNS Montgomery multiplication depends on:

1. The number of temporary variables in the working set
2. The number of precomputed constants used in Algorithm 9
3. The number of precalculated multiples of x for exponentiation

The number of temporary variables is found by mapping the variables used in Table 3.2 to registers. How many precomputed constants are needed depends mostly on the number of cores, as for each iteration in the base extensions an extra constant is needed. Extra registers available for exponentiation depend on which of the algorithms is chosen from Section 2.3. Mapping of the variables to registers can be done without use of a compiler, as the number of variables is relatively low. To improve readability, $|-n_1^{-1}|_{m_1}$ is denoted by n'_1. Looking back at the example from Table 3.2, the variables used in the first RNS core are:

$$a_1, b_1, n'_1, a_5, b_5, n_5, c, q_1, \sigma_1, \hat{q}_5, d, e, f, \hat{r}_5, \xi_5, g, \beta, h, \hat{r}_1$$

where a_1, b_1, n'_1, a_5, b_5 and n_5 are input operands for the Montgomery multiplication. The result is stored in $\hat{r}_5$ and $\hat{r}_1$, which should be kept until execution finishes. The remaining variables are used as temporary storage and can be overwritten after usage. Note that residues n'_1 and n_5 are derived from N and are reused in multiple rounds of the Montgomery multiplication. So those are assigned to their own registers and do not need scheduling. Also b_1 and b_5 are kept persistent, as those could be used to store precomputed multiples and should not be modified. The lifetime of the non-persistent variables is shown in Table 3.3.

Cycle	**Lifetime of variables**										**Total**
0	a_1	a_5									2
1		a_5	c								2
2		a_5		q_1							2
3		a_5			σ_1						2
9		a_5				$\hat{q}_5$					2
10						$\hat{q}_5$	d				2
11							d	e			2
12								f			1
13									$\hat{r}_5$		1
14									$\hat{r}_5$	ξ_5	2
20									$\hat{r}_5$	g	2
22									$\hat{r}_5$	g	β	3
23									$\hat{r}_5$	g	h	3
24									$\hat{r}_5$		$\hat{r}_1$	2

Table 3.3: Register usage for RNS Montgomery multiplication on the first core.

Note that register t was not included in the list, because that register will not be located in the register file. Instead, it is placed in the DSP blocks to enable performing the multiply-accumulate operation in 1 cycle. This is important, as that operation makes up for a large part of the total execution time of the algorithm. Also the values for σ_i and ξ_j from the other cores are not included as they will be available from the `propagate` register. Scheduling the operations from Table 3.3 using seven registers results in Table 3.4.

Register binding

Register 1	Register 2	Register 3
a_1	a_5	
c	a_5	
q_1	a_5	
σ_1	a_5	
$\hat{q}_5$	a_5	
$\hat{q}_5$	d	
e	d	
f		
	$\hat{r}_5$	
ξ_5	$\hat{r}_5$	
g	$\hat{r}_5$	
g	$\hat{r}_5$	β
g	$\hat{r}_5$	h
$\hat{r}_1$	$\hat{r}_5$	

Table 3.4: Register binding for RNS Montgomery multiplication on the first core.

All operations involving the RNS Montgomery multiplication can be scheduled using three registers only. Also two additional registers are needed for storage of n'_1 and n_5. Storage of b_1 and b_5 will be considered according to the choice of the exponentiation algorithm. The precalculated constants needed for the first RNS core in the example from Table 3.2 are:

$$|M_1^{-1}|_{m_1}, |M_1|_{m_5}, |M_2|_{m_5}, |M_3|_{m_5}, |M_4|_{m_5}, |M^{-1}|_{m_5},$$

$$|\widetilde{M}_5^{-1}|_{m_5}, |\widetilde{M}_5|_{m_1}, |\widetilde{M}_6|_{m_1}, |\widetilde{M}_7|_{m_1}, |\widetilde{M}_8|_{m_1}, |\widetilde{M}|_{m_1}$$

These constants are unique for each RNS core. The number of constants depends mostly on the number of cores in the system, as both base extension algorithms need an extra constant for each extra core. For the remaining part of algorithm there are four constants needed, which is independent from the number of cores. So for a system with $k = 121$ cores, there should be storage for $2k + 4 = 246$ constants per core. The redundant core requires two constants less than a RNS core. Multiplication by k and addition of the required number of constants for the redundant core gives a total number of constants needed equal to $2k^2+6k+2$.

Analysis of Section 2.3 shows that the methods for performing exponentiation have different memory requirements. For computation of x^e using binary exponentiation, it is necessary to store a copy of x to use in the algorithm. After conversion of x to RNS bases $\mathcal{B}$ and $\widetilde{\mathcal{B}}$, this value can be stored in residue registers b_i with $i \in \{1, ..., 2k, r\}$. But for usage of 2^w-ary and sliding window exponentiation more storage is needed, which depends on the window size.

Sliding window exponentiation has half the memory requirements of 2^w-ary exponentiation, because only odd powers of x are stored in the first case. From Algorithm 2 follows that for a window size w, there need to be $2^w - 1$ precomputed multiples of x. Also, the discussion in Section 2.3.3 gives an optimal window size of $w = 7$ for 4096-bit RSA encryption. So there are $2^7 - 1 = 127$ precomputed multiples of x that need to be stored in the RNS cores. These values need to be available in both RNS bases, so each precomputed constant results in two residues. This doubles the number of registers needed in the RNS core to $2 \cdot 127 = 254$.

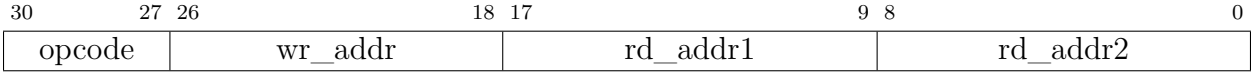
Source	**Number of registers**
$\lvert -n_1^{-1} \rvert_{m_1}$	1
$\lvert n_{k+1} \rvert_{m_{k+1}}$	1
Temporary variables	3
Montgomery constants	246
2^7-ary exponentiation	254
Total	505

Table 3.5: Memory requirements for 4096-bit RSA encryption on the first RNS core.

A summary of the register requirements for one RNS core is given in Table 3.5. Rounding up to a power of two results into a memory block with 512 entries of 34 bits. Using an address line of nine bits wide should enable access to all needed locations in memory. To support execution of two-port operands, the register file should have two read ports. Also one write port is needed for storing the result of the operation. So using three address lines of nine bits wide each should be enough to run RNS Montgomery exponentiation on the RNS cores.

3.6 RNS core instructions

The calculations in Table 3.2 can be transformed into a basic set of arithmetic operations with one output and two input operands. A combination of an arithmetic operation with read and write registers forms an instruction. Each of the RNS cores accepts a 31-bit instruction consisting of an operation code (opcode), one write address and two read addresses. As discussed in the previous section, using address lines of nine bits each should be sufficient.



Using an opcode of four bits is sufficient to encode all operations needed to perform RNS Montgomery multiplication. Most operations perform arithmetic and reduce the result in either RNS base $\mathcal{B}$ or $\widetilde{\mathcal{B}}$. Other instructions are used for writing data to the register file or performing partial single-cycle computations. An overview of the opcodes and a short description of the operations they perform can be found in Table 3.6.

Instruction	Opcode	Description	Reduction
`addu_m0`	`0000`	Addition of two unsigned numbers	$\mathcal{B}$
`addu_m1`	`0001`	Addition of two unsigned numbers	$\widetilde{\mathcal{B}}$
`subu_m0`	`0010`	Subtraction of two unsigned numbers	$\mathcal{B}$
`subu_m1`	`0011`	Subtraction of two unsigned numbers	$\widetilde{\mathcal{B}}$
`mult_m0`	`0100`	Multiplication of two unsigned numbers	$\mathcal{B}$
`mult_m1`	`0101`	Multiplication of two unsigned numbers	$\widetilde{\mathcal{B}}$
`mult_read_m0`	`0110`	Get the multiplication result	$\mathcal{B}$
`mult_read_m1`	`0111`	Get the multiplication result	$\widetilde{\mathcal{B}}$
`modu_read`	`1000`	Get result from the last reduction process	-
`prop_read`	`1001`	Get the value from `propagate` register	-
`bcas_read`	`1010`	Get the value from `broadcast` bus	-
`no_operation`	`1011`	Perform no operation; idle	-
`multiply`	`1100`	Multiplication of two unsigned numbers	-
`mult_accum_p`	`1101`	MAC an unsigned and the `propagate` register	-
`modu_bcas_m0`	`1110`	Do partial reduction on `broadcast` bus	$\mathcal{B}$
`modu_bcas_m1`	`1111`	Do partial reduction on `broadcast` bus	$\widetilde{\mathcal{B}}$

Table 3.6: Available RNS core instructions and their opcodes.

The first six instructions `addu_m`, `subu_m` and `mult_m` perform addition, subtraction and multiplication of two operands respectively. The result is reduced based on the moduli set chosen (`m0` or `m1`) and written to the register located at `wr_addr`. Each of these instructions takes multiple cycles to complete and does not use any form of pipelining.

The five `read` instructions are used to read data from intermediate registers in the RNS core and write them to the register file. All `read` instructions ignore the `rd_addr1` and `rd_addr2` fields, as nothing is read from the register file. The two `mult_read` instructions also perform reduction using one of the two RNS bases and are used to transfer data from the multiplier.

Instructions `no_operation`, `multiply` and `mult_accum_p` are executed in one cycle. The `no_operation` is mostly used internally as it is inserted when none of the other operations is scheduled. It is also useful for halting the RNS cores when the redundant core is busy doing the computation and broadcast of correction factor β. The `multiply` and `mult_accum_p`

instructions are provided to accelerate the base extension loop and are fully pipelined. Where `multiply` uses two registers as input operands, is `mult_accum_p` connected to one register and the `propagate` register. The latter also accumulates values instead of overwriting them in multiplication result register t.

Finally, the two `modu_bcas_m` instructions are used to convert 4096-bit binary numbers to RNS representation. Using a form of iterative reduction the operand is streamed in chunks of 16 bits and reduced sequentially from MSB to LSB. When reduction completes, the instruction `modu_read` is used to move the result to the register file.

3.7 Exponent calculation

The design of the RNS cores is used to perform Montgomery multiplication. However, repeated use of that algorithm is needed to get exponentiation. So additional hardware is needed to process the RSA exponent e and control the RNS cores. Algorithms 1 to 3 show that for each bit in e a square is always performed. Depending on which bits are set in the current window this could be followed by a multiplication with a precomputed value.

For computation of $x^e \bmod n$ by repeatedly calculating $ab \bmod n$, this means that operand a is the working variable with initial value x. Operand b then either is equal to a (for squaring) or a precomputed multiple of x. Choosing which value is used for b in hardware means implementing branching. This feature was deliberately kept out of the RNS cores to reduce complexity and area usage. So processing of exponent e has to be done using some logic outside of the RNS core complex.

In practice the RSA processor design will be used as a peripheral, so it can work together with the Central Processing Unit (CPU) of the main system. General purpose CPUs always contain some form of branching, so the decision making required for performing exponentiation can be moved to the CPU. Now the RSA processor does not need information about the exponent, it just needs to know which registers should be used for the multiplication.

Assuming that a general purpose CPU is slower than the system of optimized RNS cores, the instructions used to execute Algorithm 9 are embedded into a read-only memory program. Using an extra circuit to replace certain registers from this program with others, makes it possible to switch the values for operand b. Usage of an embedded program results in a much higher frequency of the system, as instructions can be generated at the same speed as the RNS cores operate on. The slower CPU has time to carefully inspect exponent e and decides which multiplication has to be done in the next step, while the RSA processor is busy performing one round of Montgomery multiplication.

The block that is responsible for generating the instructions needed to perform RNS Montgomery multiplication is called `MontMult Program`. An overview of the architecture is given in Fig. 3.3. It shows the main building blocks of the system and the connection between the CPU and RNS cores. Also the location of the multiplication program is shown.

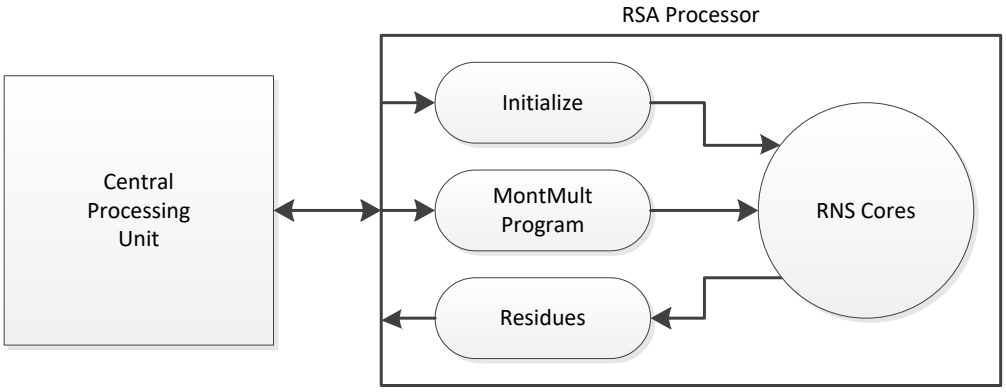


Figure 3.3: The RSA processor is connected as peripheral to a CPU.

As a peripheral, the RSA processor has become quite generic. It might be possible that other encryption algorithms are also able to use this architecture. Finding those algorithms is out of the scope of this thesis, but it is something that can be done in future research. The RNS cores have their own instruction set, but an additional pseudo-instruction set is created for usage of the `MontMult Program` block. This layer of instructions has the following goals:

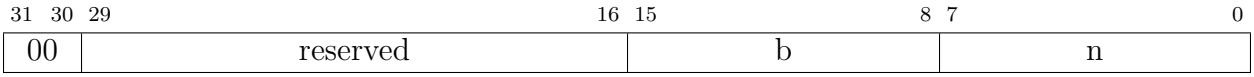
1. Perform one round of RNS Montgomery multiplication with specified registers
2. Bypass the program and execute an instruction directly on the RNS cores
3. Bypass the program and execute an instruction directly on the redundant core
4. Load an immediate value to a specified register location on all RNS cores

These four goals can be encoded using two bits and can be called from the CPU. The bus protocol used for communication between the processor and the RSA peripheral will be AXI4-Lite, which is commonly used in FPGA designs. To support usage with 32-bit processor architectures, the data lines of the bus will have a width of 32 bits. The instruction width used for the four features is also set to 32 bits, to ensure it fits in one data packet.

For one round of Montgomery multiplication (1), there are six registers needed as inputs. One pair of registers is needed for the operands a, b and n to represent these values in RNS base $\mathcal{B}$ and $\widetilde{\mathcal{B}}$. However, $32 - 2 = 30$ bits divided by six gives a maximum of five bits per register address. As the registers are used for accessing the precomputed multiples of x for the exponentiation, they should be able to use at least 256 memory locations. That would require eight bits per register address and does not fit six times into the available vector.

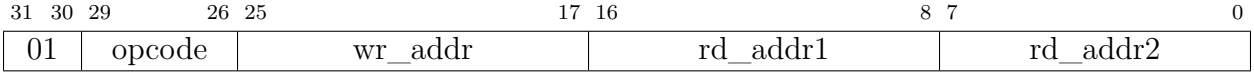
The solution is to set a constant location offset between the two registers in a pair. This allows to double the number of address bits per register, as they only are specified for the registers in RNS base $\mathcal{B}$. Location of the values for $\widetilde{\mathcal{B}}$ are computed by adding a constant value to the address. Another optimization is to assign the registers for a to a fixed location. When performing exponentiation, there always are two registers that hold the intermediate values. This prevents the need of copying the values for a to a location that won't be overwritten after execution of one Montgomery multiplication.

Using two bits `00` to denote feature (1), the format for RNS Montgomery multiplication is:

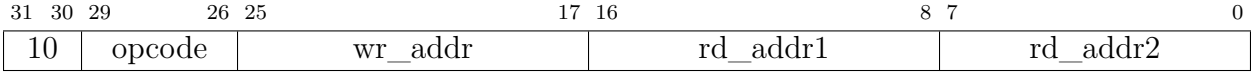


Instruction format for `mont_mult`.

Features (2) and (3) are actually a wrapper for the instruction format defined in Section 3.6. However, that instruction format contains 31 bits while here only 30 bits are available. This can be solved by hardwiring the MSB from the second read address line to zero. Now the instruction format contains one bit less and therefore fits into 30 bits. Using bit values `01` and `10` to select between the RNS cores and the redundant core, the two encapsulated pseudo-instructions are:

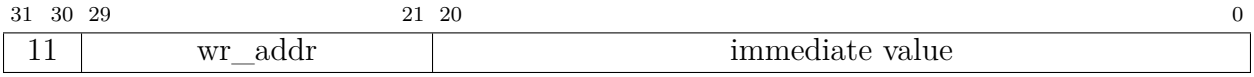


Instruction format for `core_instr`.



Instruction format for `redu_instr`.

Lastly, feature (4) requires its own instruction format to allow setting immediate values in the RNS cores. The redundant core is included when executing this instruction, as the result can always be zeroed by invoking an extra call to `redu_instr` if needed. From the 30 bits available, nine bits are used for the write address line. The remaining bits are used to set the immediate value and the format is:



Instruction format for `load_imme`.

During execution of the RSA encryption/decryption, the `mont_mult` instruction is used to perform Montgomery multiplications, instructions `core_instr` and `redu_instr` are used to copy precomputed values to the correct memory locations and `load_imme` is used to write values of one to certain registers. Writing values of one to registers is needed for initialization and to convert the residues from Montgomery form back to standard representation.

3.8 RSA encryption/decryption

This section gives an overview of how one RSA operation is performed. The bullet point icons show whether the statement is processed by the processor, RSA peripheral or both.

- • Executed by the processor.
- ★ Executed by the RSA peripheral.
- ■ Control flow generated by processor, instruction executed by RSA peripheral.

One RSA operation, encryption or decryption of message/cipher X, is executed by as follows:

- • Compute two 2048-bit random primes p and q.
- • Compute RSA parameters modulus N and decryption key d from p and q.
- • Precompute $N' = |-N^{-1}|_M$ using `EGCD`.
- • Precompute conversion factor $C = M^2 \bmod N$.
- • Write X, N, N' and C to the RSA peripheral.
- • Decide on exponentiation method (sliding window or 2^w-ary) and window size w.
- ■ Convert X to the Montgomery domain by multiplying with C.
- ■ Precompute multiples of X for the chosen window size w.
- ■ Initialize working registers 1 and 2 to a value of one on all cores.
- ■ Convert working registers 1 and 2 to the Montgomery domain by multiplying with C.
- • Iterate over the exponent/decryption key and generate `mont_mult` instructions.
- ■ Initialize two used registers to a value of one on all cores.
- ■ Convert working registers 1 and 2 to standard representation by multiplying with one.
- ■ Signal the RSA peripheral that it should start its computation.
- ★ The RSA peripheral converts the RSA parameters to RNS and initialize the cores.
- ★ The RSA peripheral processes all instructions from the instruction FIFO.
- • Wait until the RSA peripheral finishes its operations.
- • Transfer the RNS residues of the result back to the CPU.
- • Convert the residues from RNS to binary using CRT with precomputed weights.
- • Result $\hat{R}$ has range $0 \leq \hat{R} < (k+2)N$, so perform one final reduction by modulo N.

4 Implementation overview

This section will discuss the implementation of the design in logic using a top-down approach. The top-level entity is the RSA peripheral, which consists of the RNS processor, RSA controller, RSA loader, instruction generator, residue reader and some storage elements. Next, we will inspect the RNS cores, redundant core and ring structure that connects them. And finally, the implementation of an individual RNS core will be explained.

4.1 RSA peripheral

The top-level entity of the implementation is the RSA peripheral. Its detailed structure is shown in Fig. 4.1. The peripheral provides an interface between the processor and the RNS cores. The processor is expected to run at a different speed than the RSA peripheral. Data transfers between clock domains can be tricky, so the clock domain transition is placed at the storage elements. This simplifies synchronization, as FPGAs contain primitives for FIFOs and BRAMs that support read and write ports in unrelated clock domains. Timing constraints for these primitives are inserted automatically by the synthesis tools.

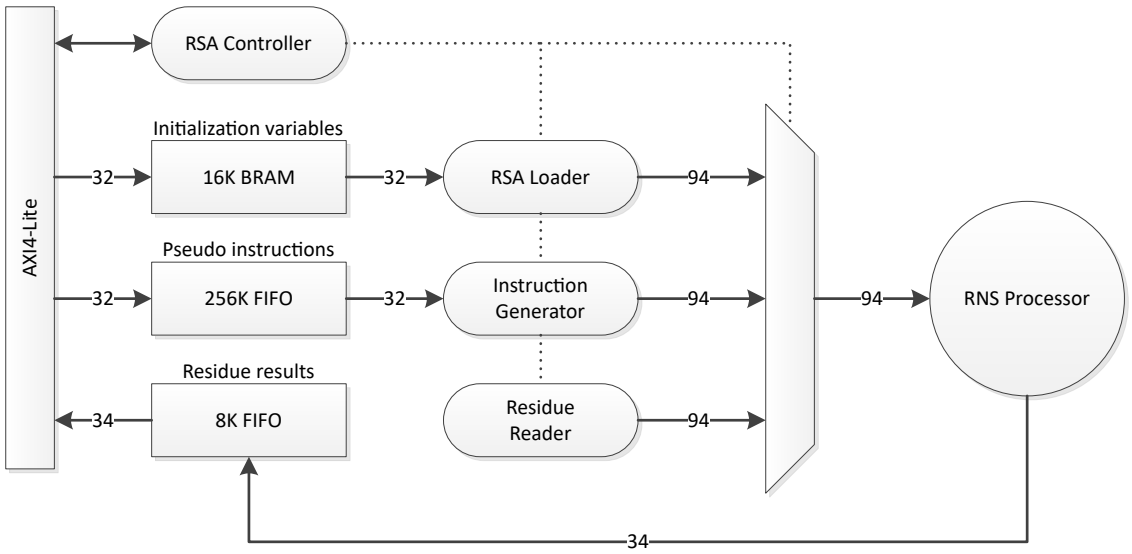


Figure 4.1: Overview of the RSA accelerator AXI4-Lite peripheral.

The RSA controller is a Finite-State Machine (FSM) that controls the sequence in which the RSA loader, instruction generator and residue reader are enabled. It also contains status registers to provide information about the state of the RSA peripheral. Fig. 4.1 shows three selectable 94-bit data streams to the RNS cores, which are a concatenation of the 32-bit broadcast line and two 31-bit instruction streams. Depending on the state of the RSA controller, it selects one of these streams to be processed by the system of RNS cores.

4.1.1 RSA loader

The implementation of the RSA loader is shown in Fig. 4.2. Note that the BRAM is also shown in Fig. 4.1, but in the design it is considered to be part of the RSA loader. The goal of the RSA loader is to initialize the registers of the RNS cores with the proper values before Montgomery multiplication can start. It is also used for the conversion of 4096-bit numbers to the residue number system. The BRAM holds the initialization values and is used as clock domain crossing between the AXI4-Lite bus and the RSA peripheral.

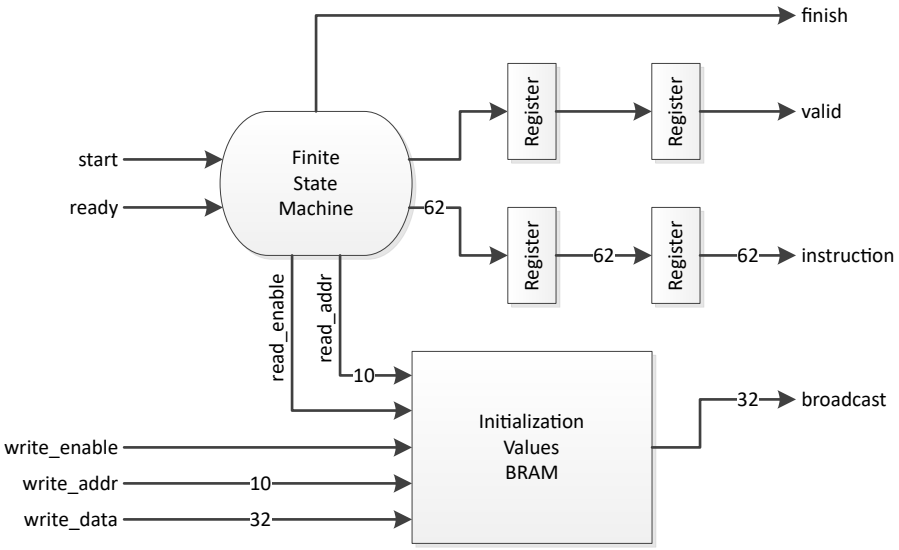


Figure 4.2: Overview of the RSA loader component.

There are four precomputed values needed by the RNS cores before starting the multiplications. For calculation of $x^e \bmod n$, values x and n are needed in the RNS cores. Exponent e is not needed in the RSA peripheral, as it is processed by the CPU. The other two precomputations are derived from n, because it does not change during one RSA encryption round. From Algorithm 9 follows that n is only needed in $\widetilde{\mathcal{B}}$ and $n' = |-n^{-1}|_M$ can be precomputed in $\mathcal{B}$. The latter is larger than 4096 bits, because of its relation to range M. The last initialization variable follows from conversion to Montgomery n-residue representation, for which "$M^2 \bmod n$" is needed. An overview of the initialization values is given in Table 4.1.

Value	**Size (bits)**	**RNS Base**
x	4096	$\mathcal{B}$ and $\widetilde{\mathcal{B}}$
n	4096	$\mathcal{B}$
$M^2 \bmod n$	4096	$\mathcal{B}$ and $\widetilde{\mathcal{B}}$
$n' = \lvert -n^{-1} \rvert_M$	4114	$\widetilde{\mathcal{B}}$

Table 4.1: Initialization values for the RSA peripheral.

From Table 4.1, it follows that the initialization values have a memory requirement of at least 513 storage entries of 32 bits. Conversion from binary to RNS is done by the reduction circuit available in each RNS core. By streaming the initialization values from the BRAM to the RNS cores over the 32-bit `broadcast` line and invoking `modu_bcas_m0` or `modu_bcas_m1` instructions, the large precomputed values are sequentially reduced to 34-bit numbers. However, as the reduction circuit can only reduce using one modulo value, two iterations are needed for values that need to be available in base $\mathcal{B}$ as well as in base $\widetilde{\mathcal{B}}$. So by iterating twice over values x and "$M^2 \bmod n$", a total number of six binary numbers is converted to RNS in the initialization phase of one RSA operation.

4.1.2 Instruction generator

The instruction generator is used to consume instructions from the CPU and generates new instructions such that the RNS cores execute Algorithm 9. Its implementation is shown in Fig. 4.3. Note that the instruction FIFO on the left side is the same as the FIFO depicted in Fig. 4.1. The instruction stream needed to perform one Montgomery multiplication is stored into a read-only memory (ROM), which allows for fast execution. The FSM uses a program counter that keeps track of which instruction should be executed next.

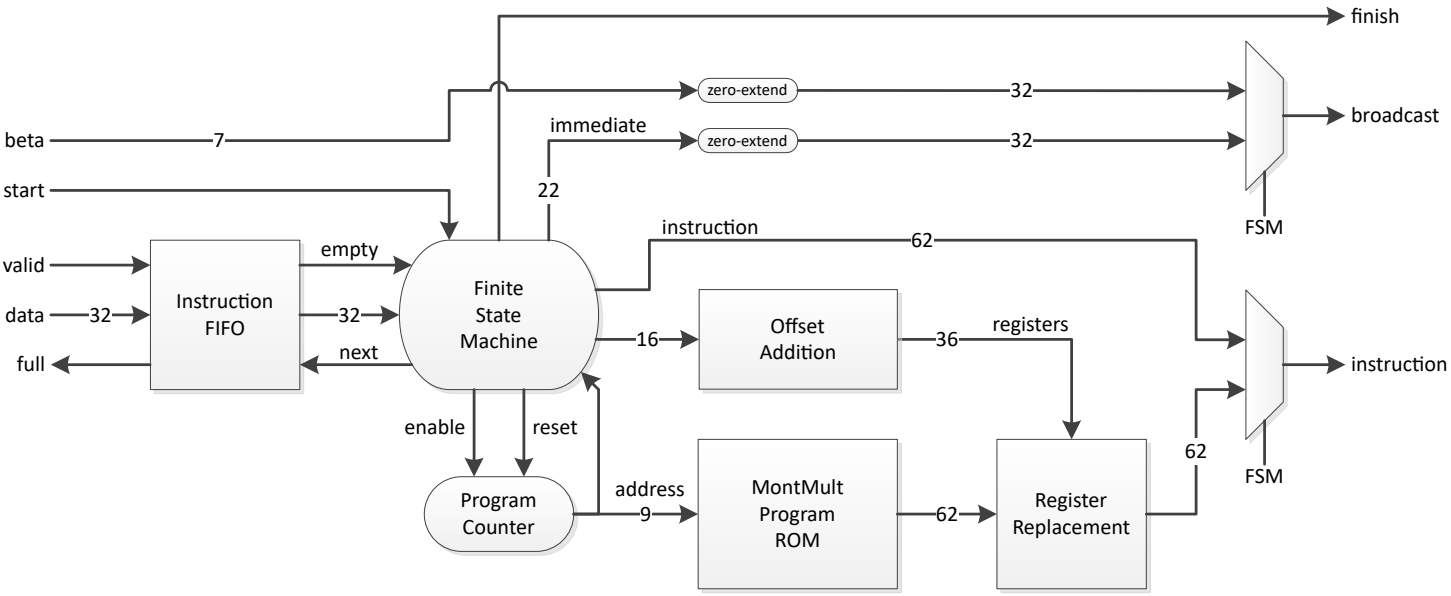


Figure 4.3: Overview of the instruction generator for RNS Montgomery Multiplication.

However, storing a compiled version of Algorithm 9 in ROM introduces a new problem. Operands b and n of the `mont_mult` instruction contain different register addresses depending on processing of the RSA exponent by the CPU. As those register addresses are stored statically in the ROM, it means their value cannot be changed. A solution is found by introducing the `Register Replacement` stage that replaces specific registers during runtime.

Recall the four instruction formats defined in Section 3.7. The instruction FIFO contains instructions using one of these formats, which can be distinguished by inspection of two MSBs. Instruction decoding of this format is done by the FSM and it selects another state based on the instruction. For Montgomery multiplication, 16 bits in the instruction are output from the FSM to the `Offset Addition`. This stage decodes the input vector into four address lines of nine bits each, according to the format of the `mont_mult` instruction. Once the actual registers are known, they can be replaced in the program code. The RNS Montgomery Multiplication program code can be found in Appendix A.3.

The other instructions that can be processed by this component are `core_instr`, `redu_instr` and `load_imme`. These three instructions can be mapped directly to the instruction format used by the RNS cores. So those instructions are processed by the FSM, using the bypass paths for the `broadcast` and `instruction` lines. The bypass paths are delayed using three register stages, to match the delay introduced by the `MontMult Program` block.

4.1.3 Residue reader

The goal of the residue reader is to transport the residues from the RNS cores back to the main processor of the system. This is done after the RSA encryption/decryption step has completed. An overview of its implementation is given in Fig. 4.4.

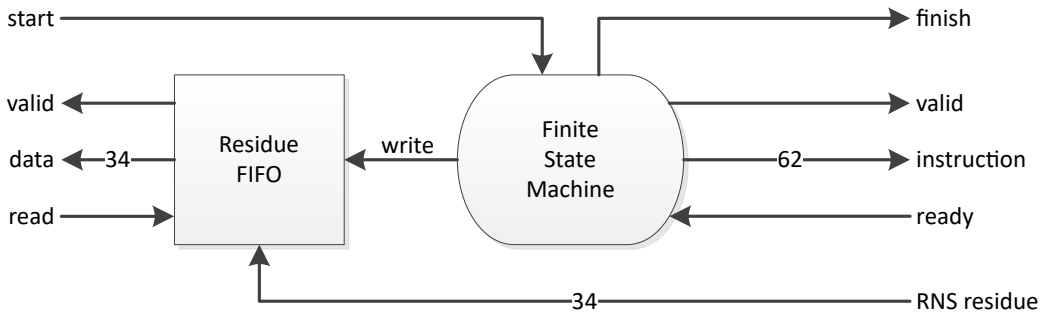


Figure 4.4: The RSA reader is used to transfer RNS residues to the AXI4-Lite bus.

The FSM is responsible for generating instructions that let the RNS cores propagate their residues around the ring structure to their neighbors. RNS core zero has its `propagate` register connected to a neighbor, but also to the `RNS residue` port of the residue reader component. After a certain number of cycles, depending on how deep the instruction pipeline is, the requested residues appear at the `RNS residue` input. By timing this exact moment with generation of the FIFO `write` flag, the residues can be stored in the output FIFO.

Fig. 4.4 shows that the residue reader FIFO has an output port of 34 bits wide. However, this cannot be connected to the AXI4-Lite bus directly, as its size was previously determined to be 32 bits. By assigning the lower 32 bits from the residue to one AXI address and the upper 2 bits to a seconds address this is solved. Introducing an extra flag register `read`,

which is high for one cycle only when asserting it, the next value will be made available. The downside of this approach is that for reading one residue, two reads and one write are needed from the host CPU. However, this is not a problem as this process has to be completed once per residue at the end of a modular exponentiation operation only.

4.1.4 Memory mapping

Connection of the RSA peripheral to the AXI4-Lite bus follows the addressing scheme given in Table 4.2. These addresses are needed to give the CPU access to the registers of the peripheral. Registers `message`, `montgomr`, `modulus` and `mininvn1` are located in the BRAM from the RSA loader component. The AXI4-Lite bus uses a data width of 32 bits per transfer, larger regions can be written to by incrementing the address in steps of four bytes.

Address	Size (bits)	Register name	Description
0x0000	4096	`message`	RSA message or cipher
0x0200	4096	`montgomr`	Initialization constant $C = M^2 \bmod N$
0x0400	4096	`modulus`	RSA modulus N
0x0600	4128	`mininvn1`	RSA minus inverted modulus $N' = \lvert -N^{-1} \rvert_M$
0x1000	1	`start`	Assert to start RSA encrypt/decrypt
0x1004	1	`finish`	Asserts when RSA operation is finished
0x1008	32	`instruction`	Montgomery instruction stream
0x100C	32	`residue_low`	32 lower bits of last residue
0x1010	2	`residue_high`	2 upper bits of last residue
0x1014	32	`residue_count`	Residues available for reading
0x1018	1	`residue_next`	Get next residue and purge the last

Table 4.2: AXI4-Lite bus interface of the RSA peripheral.

Next, the `start` and `finish` registers are connected to the RSA controller. These are flags used to start the encryption or decryption process and to signal back when it has finished. The `instruction` register is write-only for the CPU and connected to the FIFO from the `MontMult Program` block. It can be filled with instructions of the format from Section 3.7.

Finally, the registers `residue_low`, `residue_high`, `residue_count` and `residue_next` are located in the residue reader component. As the AXI4-Lite bus is used for data transfers of 32 bits, the residue values of 34 bits are split up and assigned using two register addresses. The lower part of the residue can be read from `residue_low` and the upper two MSBs of the residue are available from `residue_high`. Also, the numbers of residues stored in the output FIFO can be found in the `residue_count` register.

4.2 RNS processor

The RNS processor is located on the right side of Fig. 4.1 and its implementation overview is given by Fig. 4.5. It contains the ring of RNS cores, one redundant core, two instruction decoders and one stream synchronization component called `join`. The purpose of `join` is to provide synchronization between the RNS cores and the redundant core, when they are executing two different instructions that do not take an equal number of clock cycles. This results in two instruction streams behaving as one and that reduces the overall complexity.

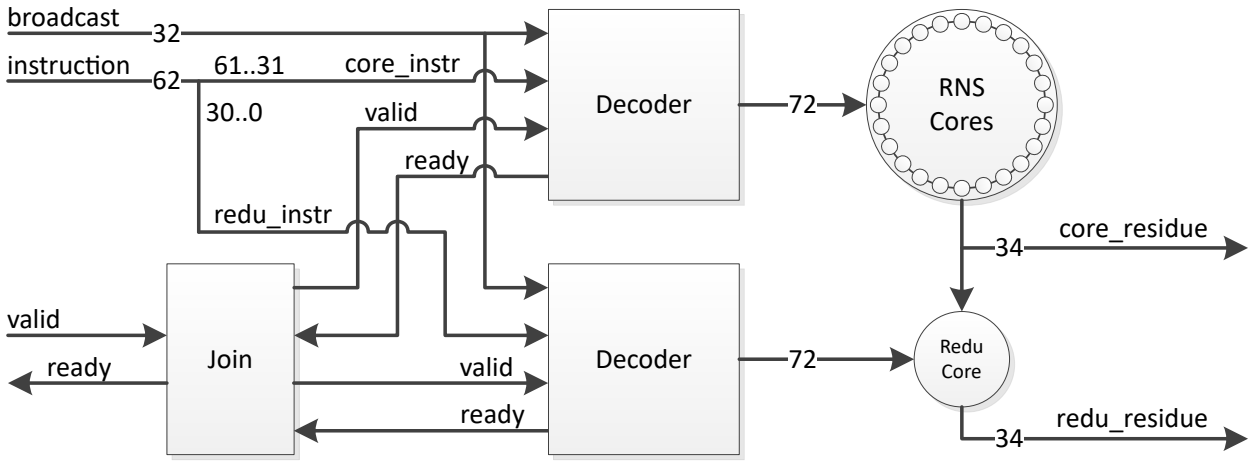


Figure 4.5: Overview of the RNS processor with two instruction decoders.

The decoder stage is an abstraction level for the timings associated with the instruction that is executed. It contains definitions about when enable signals for arithmetic blocks or registers inside the RNS core need to be enabled. All RNS cores in the ring execute the same instruction, therefore one decoder is enough to control the whole ring. The redundant core needs to execute different instructions than the RNS cores, so it has its own decoder stage.

4.2.1 Instruction decoder

The decoder consumes the instruction stream and generates the control signals needed for the RNS cores to perform arithmetic functions. It is implemented using a finite-state machine with a different state for each of the sixteen possible opcodes. Recall the instruction format defined in Section 3.6. Based on the `opcode` field, the number of computation cycles associated with that opcode is loaded into a counter. Each clock cycle this counter is decremented until it reaches zero and the next instruction can be processed. A circuit of combinatorial logic determines which control signals should be raised, based on the current state and counter value. These control signals consist of clock enables and multiplexer selection bits. An overview of the signals can be found in Table 4.3.

Signal name	Bits	Description
`add_nsub`	1	Select addition or subtraction
`read_enable`	1	Enable reading from register file
`mult_enable`	1	Enable multiplier operation
`accum_enable`	1	Enable multiply-accumulate operation
`adder_enable`	1	Start adder operation
`write_enable`	1	Enable writing to the register file
`modulo_enable`	1	Start modulo reduction
`next_select`	1	Select register connected to next core
`mult_select`	1	Select multiplier operand
`modulo_select`	1	Select RNS base to perform reduction in
`reduce_select`	1	Select modulo reduction operand
`write_select`	2	Select what is written to the register file

Table 4.3: Controls signals generated by the decoder stage.

Combination of the signals in Table 4.3 results into a 13-bit control vector. The signal vector from the decoder to the RNS cores in Fig. 4.5 contains 72 bits and the control vector is a part of it. The other 59 bits are the 32-bit `broadcast` signal and three 9-bit address lines for the register file. Fig. 4.6 gives an overview of all signals connected to the decoder.

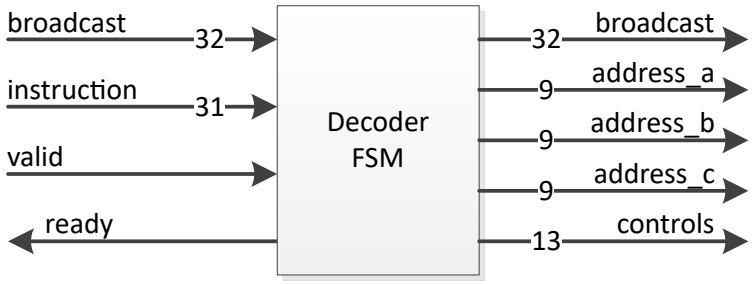


Figure 4.6: Schematic view of the inputs and outputs from the decoder FSM.

Routing a 72-bit bus to 121 cores poses a design challenge, as all signals need to arrive at all RNS cores within the same clock cycle. To ease placement of these paths, several register stages are inserted to enable long travel distances across the FPGA chip. This gives more degrees of freedom for the placement and routing of the RNS cores. In the design of the RSA peripheral, three clock cycles are needed for transporting the 72-bit signal to a core.

Not all control signals from Table 4.3 are relevant for all types of instructions. Each instructions has its own timings at which certain signals are set or reset. An overview of how the control signals are mapped to an instruction can be found in Appendix A.4.

4.3 RNS core

The RNS core is required to perform at least the following functions: addition, subtraction, multiplication and reduction. These operations are mapped to hardware and combined with a register file for storage of the variables and precomputed constants. The schematic overview of the RNS core is given in Fig. 4.7. Multiplexers are used to select which value is written to the register file, is propagated to the next RNS core or is input to the reduction stage.

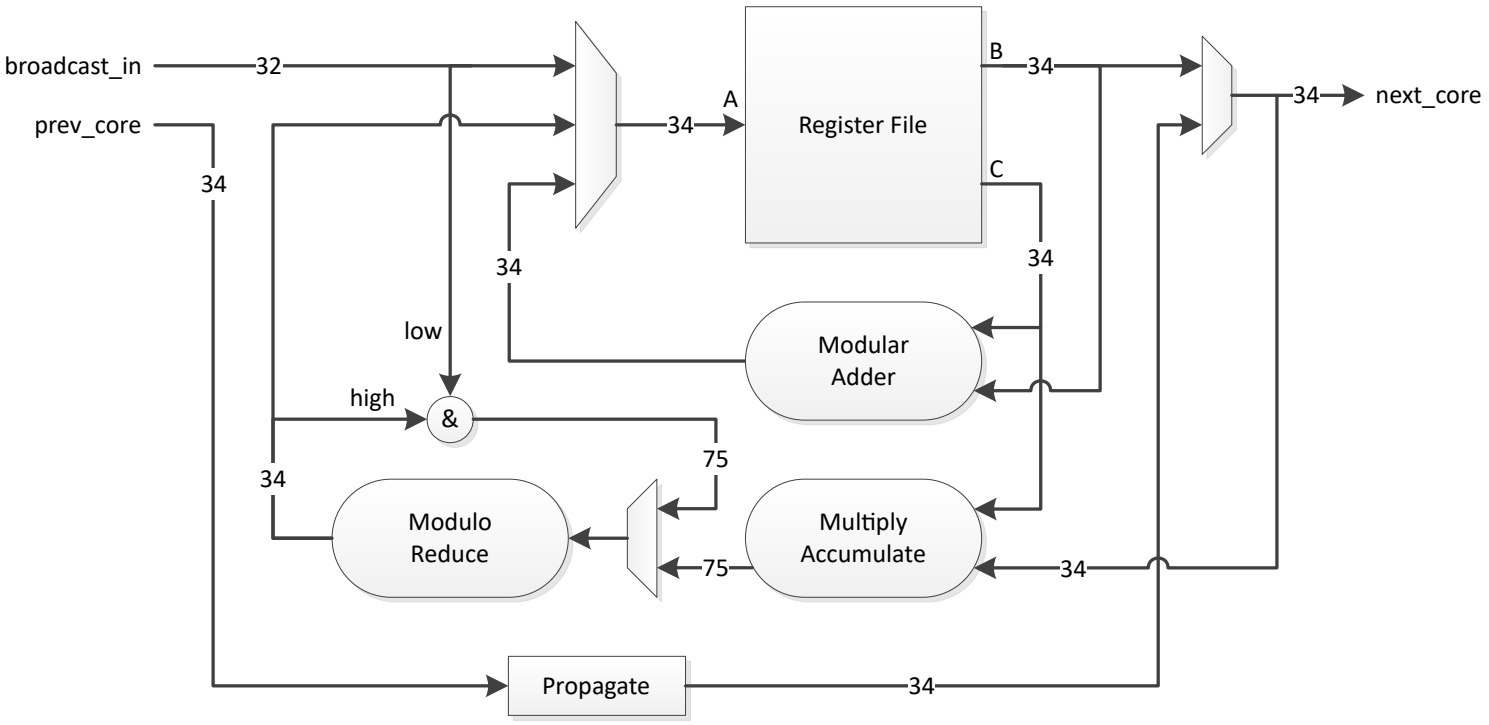


Figure 4.7: Implementation overview of one RNS core.

Each of the arithmetic functional blocks also has an enable input port. This is not shown in the schematic, as only datapaths are visible and not the control signals. The register file has one write port and two read ports, so that instructions with a maximum of three operands can be processed. As the RNS cores are connected to each other in a ring, each core has two neighbors. Each core has its `next_core` output connected to another core's `prev_core` input. The `propagate` register ensures that the number of logic elements in the ring is kept low and allows to run on higher frequencies. This path is used for transporting the values that are shared between the cores for performing the RNS base extensions.

The multiplexer between the multiply-accumulate and the modulo reduce blocks is used to select which value is reduced: the result of the multiplier or the loop-back path from the reduction unit combined with the `broadcast_in` signal. The latter is used for conversion of a large binary number split in chunks to the RNS residue needed for that core.

4.3.1 Register file

The register file is used for storing precomputed constants and working variables. When using a FPGA from the Xilinx Ultrascale series, the register file is mapped to one `RAMB36E` primitive. An overview of the inputs and outputs from the register file is given in Fig. 4.8. Note that the clock input is not drawn.

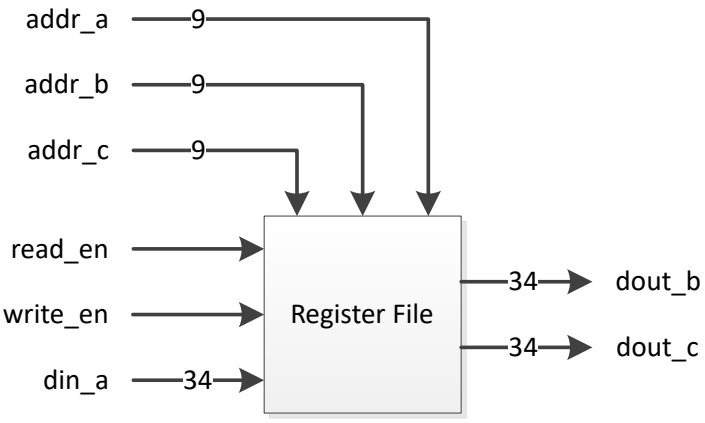


Figure 4.8: Input and outputs of the register file.

The register file has an addressable depth of 512 entries containing 34 bits each. Most locations are reserved for storing precomputed values, which is shown in the lower half of Table 4.4. The upper half of the memory is used for storing multiples of the input x in the computation of $x^e \bmod n$. Remaining locations are reserved for execution of the Montgomery multiplication algorithm.

Which multiples of x are stored in the register file, depends on the exponentiation algorithm chosen. Table 4.4 shows the assignments for use with 2^w-ary exponentiation. For sliding window exponentiation, even powers of x are not stored so either the amount of memory locations is halved or the window size w can be increased by 1 bit.

Location	Description
0	Hardwired to zero
1	Temp register t_0
2	x_1
3	$(x_1)^2$
...	...
127	$(x_1)^{126}$
128	$(x_1)^{127}$
129	$\lvert -n_1^{-1} \rvert_{m_1}$
130	$\lvert M^2 \bmod N \rvert_{m_1}$
131	Free
132	Free
133	Temp register t_2
134	Temp register t_1
135	x_{122}
136	$(x_{122})^2$
...	...
260	$(x_{122})^{126}$
261	$(x_{122})^{127}$
262	n_{122}
263	$\lvert M^2 \bmod N \rvert_{m_{122}}$
264	Free
265	Free
266	$\lvert M_1^{-1} \rvert_{m_1}$
267	$\lvert M_1 \rvert_{m_{122}}$
268	$\lvert M_2 \rvert_{m_{122}}$
...	...
386	$\lvert M_{120} \rvert_{m_{122}}$
387	$\lvert M_{121} \rvert_{m_{122}}$
388	$\lvert \widetilde{M}_{122}^{-1} \rvert_{m_{122}}$
389	$\lvert \widetilde{M}_{122} \rvert_{m_1}$
390	$\lvert \widetilde{M}_{123} \rvert_{m_1}$
...	...
508	$\lvert \widetilde{M}_{241} \rvert_{m_1}$
509	$\lvert \widetilde{M}_{242} \rvert_{m_1}$
510	$\lvert \widetilde{M} \rvert_{m_1}$
511	$\lvert M^{-1} \rvert_{m_{122}}$

Table 4.4: Register binding of the first RNS core for 121 cores in total.

4.3.2 Modular addition

Modular addition and modular subtraction are performed by the `Modulo Addition` module. Its schematic overview is given in Fig. 4.9. The `mod_sel` signal is used to select which of the two modulo values is used for performing the reduction. Each RNS core is associated with two residues, so modular reduction is performed with two possible values as the modulo.

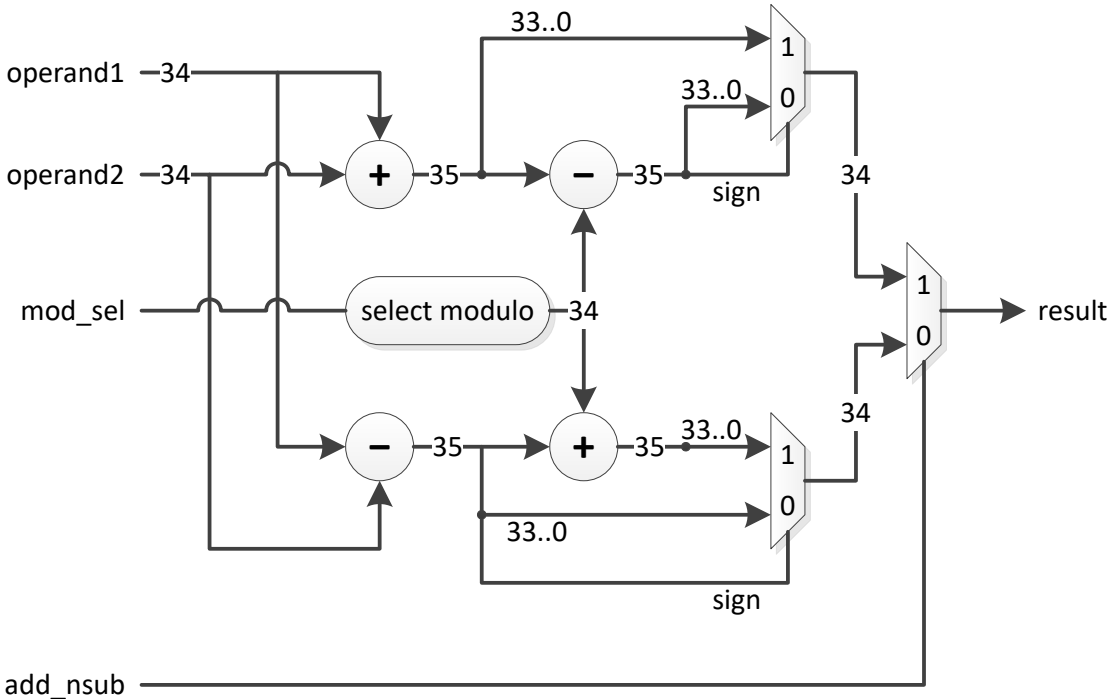


Figure 4.9: Implementation overview of the modular addition/subtraction unit.

Input operands `operand1` and `operand2` are connected to the register file. Values written to the register are always reduced by the currently selected modulo, so the input operands are already reduced. This means that for modulo n, addition of two values gives a results smaller than $2n$. To ensure this result is smaller than n a trial subtraction is performed. If the trial subtraction gives a negative result, it was not needed and the result from the addition is selected as the output value. Otherwise, the trial subtraction was needed and that result is selected by the multiplexer. To determine if the value is negative, the sign bit is used and connected to the select line of the multiplexer.

The subtraction circuit works similarly. First step is a signed subtraction of the inputs. Based on the sign bit of that result, addition is performed to ensure a positive value. Finally, a multiplexer selects the result that was requested based on the `add_nsub` signal. The inputs and outputs of Fig. 4.9 are registered to ensure timing is met. Two cycles are needed for propagation over the critical path, which is configured using a multi-cycle-path constraint.

4.3.3 Multiply-accumulate

In order to perform multiplication four Xilinx `DSP48E2` blocks are utilized. Manual instantiation of these blocks allows to also function as multiply-accumulate stage. Multiplication of two 34-bit unsigned values `A` and `B` is done by splitting these values into parts of 17 bits and then multiplying these parts with each other. This is needed, as the maximum symmetric input operand size for multiplication with one DSP block is 17 bits. Fig. 4.10 shows the schematic overview of how the DSP blocks in the 34-bit multiplier are connected.

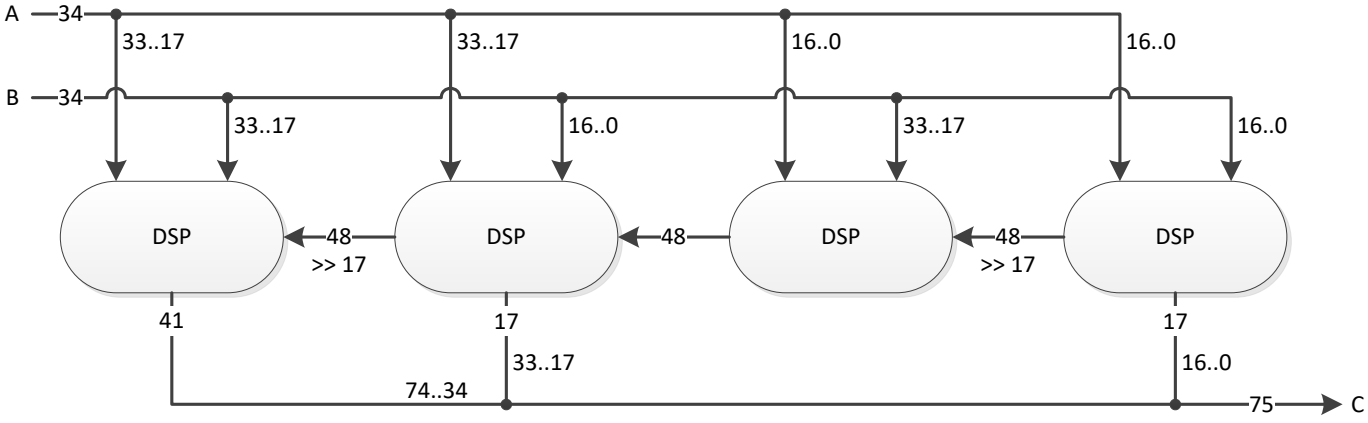


Figure 4.10: Implementation overview of the multiplication unit.

Each DSP block uses two register stages to perform the multiplication and one register stage for summation of the partial products and cascade input. Carry propagation from the right to left takes four cycles and for this the cascade outputs of the DSP blocks are used to save routing resources. These cascade inputs are shift right 17 bits where appropriate. The total propagation time for performing one multiplication of two 34-bit operands is seven cycles.

The setup of Fig. 4.10 also enables for a multiply-accumulate function, as parts of the DSP blocks can be reconfigured during runtime. An extra input signal `enable_accum` is added to allow accumulation of the results. Assertion of this signal disconnects the cascade inputs and reconnects the DSP block output register to its internal addition chain. Now new multiplication results are added to the current output of the DSP block and the multiply-accumulate functionality is enabled. Carry propagation is reactivated automatically again after no new multiplications are input for two cycles.

Output port `C` of the multiplier is larger than twice the size of the operands to allow for writing the accumulated value. The multiply-accumulate operation is only used during the two base extension phases of the algorithm. As the RNS processor ring contains $k = 121$ cores, the maximum number of consecutive accumulate operations also is equal to 121. The maximum output size therefore is $\lceil 68 \cdot \log_2(121) \rceil = 75$ bits, as discussed in Section 3.4.

4.3.4 Modular reduction

The theory from Sections 2.4.3 and 2.4.4 provides two approaches for designing a circuit that performs modular reduction. A 75-bit input operand needs to be reduced by one of the two possible moduli from the RNS base and that results into one 34-bit output. The first design is a combinatorial circuit that uses lookup tables and an addition tree for summation of the partial reduced values. Implementation of tree-based reduction is shown in Fig. 4.11.

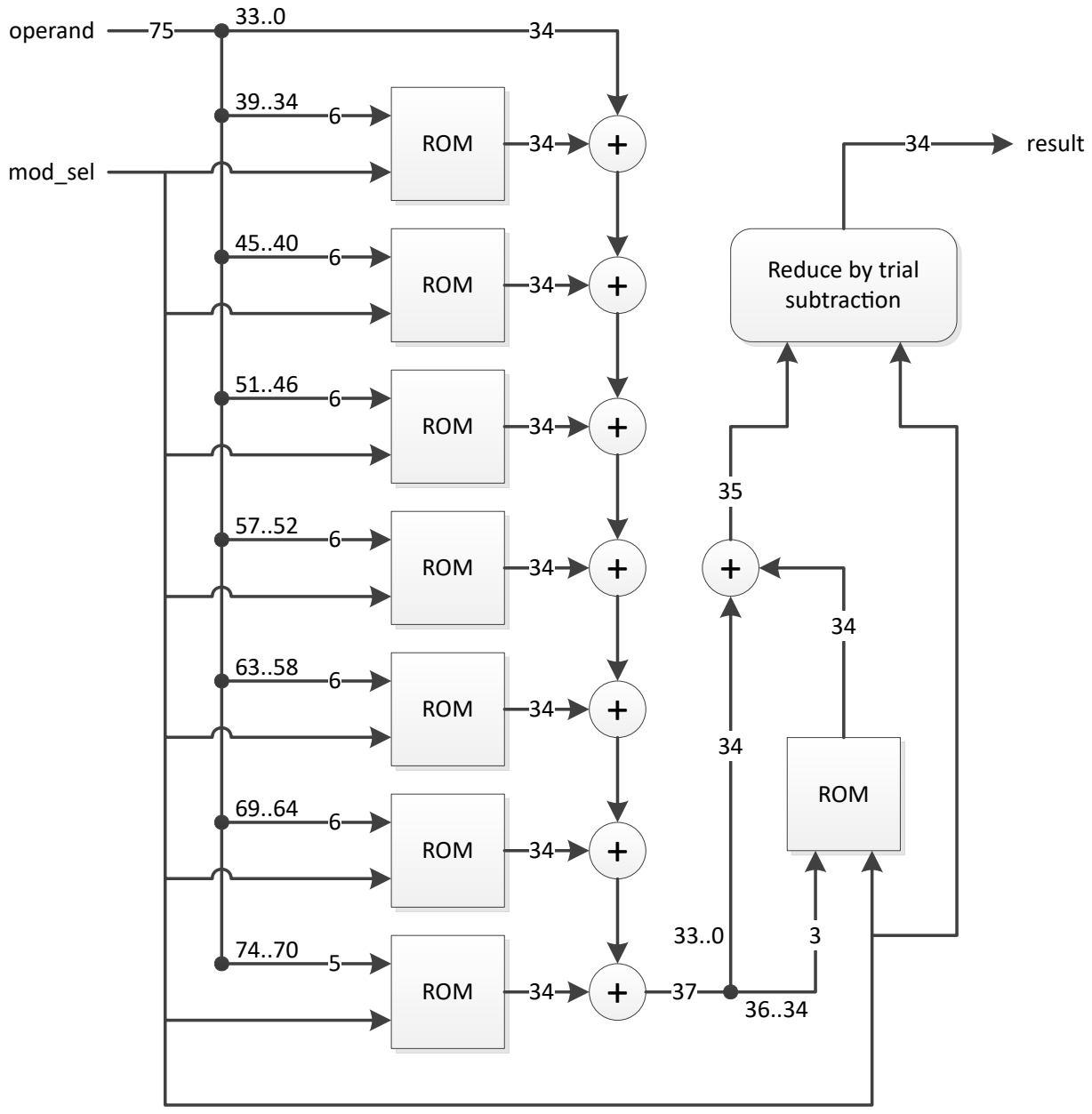


Figure 4.11: Implementation overview of the tree reduction unit.

According to the theory from Section 2.4.3, the input operand is split into multiple segments. The segment size should not be too large, as that results in very big lookup tables. But it should not be too small either, because that increases the critical path delay because of the additional adders needed. A good trade-off is given by a segment size of six bits, which results into the requirement of $\lceil (75-34)/6 \rceil = 7$ lookup tables and eight adders.

To make the critical path shorter, the addition chain can be converted into an addition tree. The need of using eight adders is optimal as it is a power of two and that maps nicely to a binary tree. Disadvantage of this method is that it increases the required amount of hardware resources. How the addition chain is replaced by a tree is shown in Fig. 4.12.

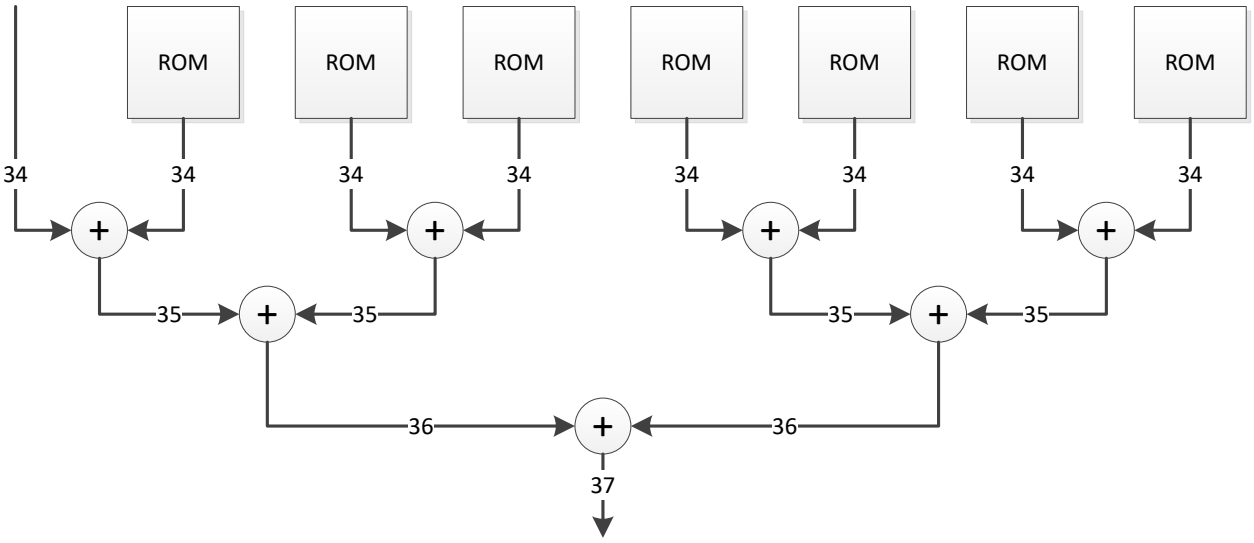


Figure 4.12: Addition tree implementation that shortens the carry propagation path.

The circuit in Fig. 4.12 is called the first reduction stage, as that reduces the input operand from 75 bits down to 37 bits. But the goal is a reduction to 34 bits, so a second reduction stage is needed. Fig. 4.11 also shows the second stage, which is just a look-up table containing 16 entires of 34 bits. Note that the `mod_sel` is also input as MSB in the address line of the lookup tables, which is used to select between two possible modulo values. The second reduction stage reduces an input of 37 bits down to 35 bits, as that is the result of one addition with two 34-bit values.

Finally, after the second stage, a trial subtraction is used for the reduction of bit 35. This step cannot be done using a lookup table and addition, as this addition could again result into a 35-bit value. Also, the result can be smaller than 35 bits, but larger than the modulo of 34 bits we are reducing to. In this case a lookup cannot be done, as there is no overflow bit. An extra check is included in the simulation model to ensure one subtraction of the modulo from the 35-bit value results into a modulo n reduced result.

Following the method discussed in Section 2.4.4 results into the second design for a reduction circuit, but now it is an iterative solution. Instead of using lookup tables for all bits above bit 34, only one table is used and the partial reduced result is shifted to the left. An FSM with sixteen states is used to keep track of where in the reduction process the circuit is, which value should be loaded into the adder and how many bits the `sum` register should be shifted. The schematic overview of the iterative reduction circuit is shown in Fig. 4.13.

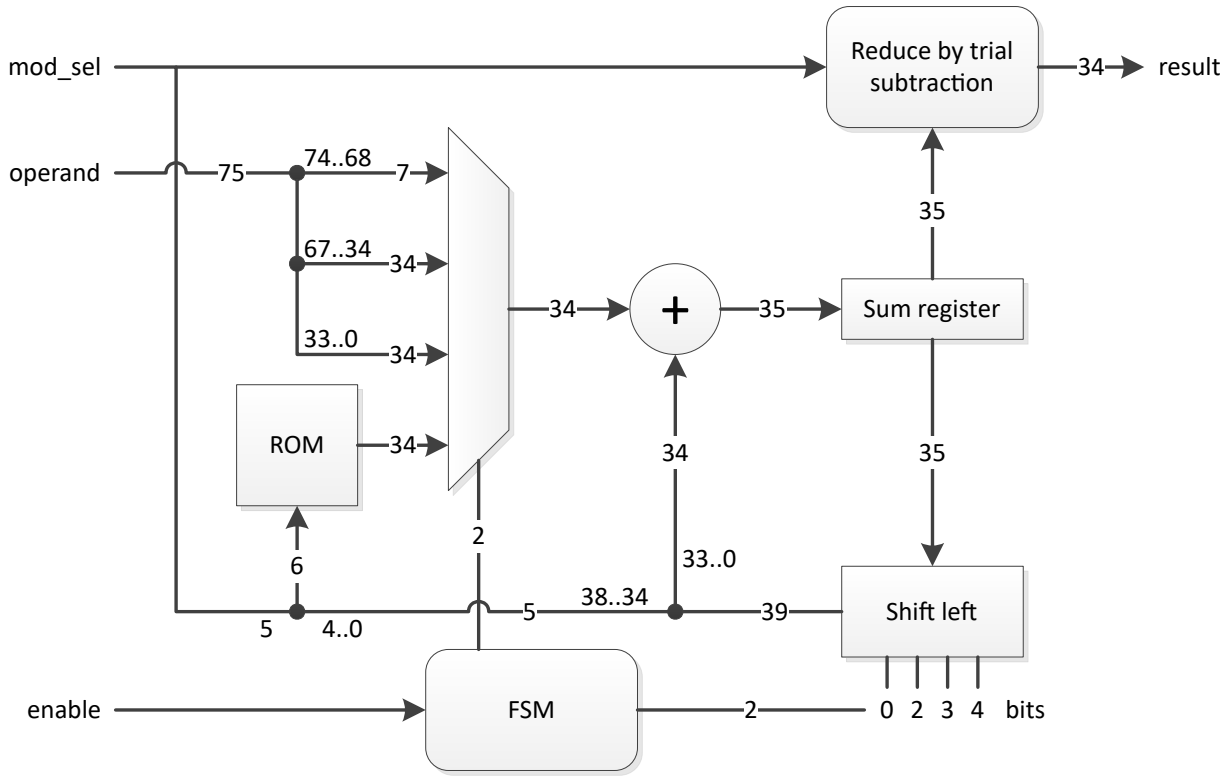


Figure 4.13: Implementation overview of the iterative reduction unit.

The block "Reduce by trial subtraction" is responsible for reduction of 35-bit value from the `sum` register to the 34-bit final result. Its performs a signed subtraction with the selected 34-bit modulo m and checks if the result has its sign bit set. When the sign bit is set, it means that the subtraction was not needed and the value prior to subtraction is propagated to the output. If the sign bit is not set, the subtracted value is propagated to the output. However, subtraction of m from `sum` number does not ensure that the result is smaller than m. In order to find out whether this is true, an upper bound on `sum` should be known.

The value written to the `sum` register is the result of the addition of a 34-bit number and a value from the lookup table. So the maximum value stored in `sum` is equal to $2^{34} - 1$ plus the maximum value stored in the lookup table. From Section 2.4.4 follows that this lookup table contains precomputed values generated by the equation $\mathbb{T}[i, m] = i \cdot 2^{34} \bmod m$.

The ROM shown in Fig. 4.13 uses five bits from the shifters output, so the table contains 32 entries for each modulo value and $0 \leq i \leq 31$. Next, a list of moduli is compiled from Appendices A.1 and A.2 and that enables computation of $\texttt{max}(\mathbb{T})$. Using a Python script with the list of the moduli, the value for $\texttt{max}(\mathbb{T}) = 161603$ was found by calculating:

$$\texttt{max}(i \cdot 2^{34} \bmod m) \quad \text{for} \quad 0 \leq i \leq 31 \quad \text{and} \quad m \in (\mathcal{B}, \widetilde{\mathcal{B}})$$

In order to reduce the 35-bit value of `sum` to a 34-bit number smaller than m using one subtraction, the `sum` register should contain a value smaller than $2m$. The maximum value stored in the `sum` register is equal to $\texttt{max}(\texttt{sum}) = \texttt{max}(\mathbb{T}) + 2^{34} - 1 = 17180030786$. From the list of moduli follows that the smallest value is $\texttt{min}(m) = \texttt{min}(\mathcal{B}, \widetilde{\mathcal{B}}) = 17179863971$. As equation $\texttt{max}(\texttt{sum}) < 2 \cdot \texttt{min}(m)$ holds, it proves that for all possible m also is given that $\texttt{max}(\texttt{sum}) < 2m$. This property proves that the `sum` register can be reduced modulo m using one subtraction only, after execution of a table lookup.

The FSM of the iterative reduction circuit is responsible for its operation. An overview of the possible states is given in Table 4.5. The state machine is executed sequentially, which means that the table is traversed from top to bottom. Each state is executed in one cycle, except for the `s_idle` state, which is used to wait for the reduction enable signal from the instruction `decoder`. After power-up the state machine also starts in the `s_idle` state.

State	Multiplexer	Shift	Description
`s_idle`	ROM lookup	0	Wait for enable signal, then sample inputs
`s_step0`	`(74 downto 68)`	0	Add bits `74..68` from the operand
`s_step1`	ROM lookup	4	Reduce bits `74..70`
`s_step2`	ROM lookup	3	Reduce bits `70..67`
`s_step3`	`(67 downto 34)`	0	Add bits `67..34` from the operand
`s_step4`	ROM lookup	4	Reduce bits `67..63`
`s_step5`	ROM lookup	4	Reduce bits `63..59`
`s_step6`	ROM lookup	4	Reduce bits `59..55`
`s_step7`	ROM lookup	4	Reduce bits `55..51`
`s_step8`	ROM lookup	4	Reduce bits `51..47`
`s_step9`	ROM lookup	4	Reduce bits `47..43`
`s_step10`	ROM lookup	4	Reduce bits `43..39`
`s_step11`	ROM lookup	4	Reduce bits `39..35`
`s_step12`	ROM lookup	2	Reduce bits `35..33`
`s_step13`	ROM lookup	0	Extra reduction step to ensure $\texttt{sum} < 2^{34}$
`s_step14`	`(33 downto 0)`	0	Add bits `67..34` from the operand

Table 4.5: FSM states of the iterative reduction circuit.

5 Benchmark results

The design described in Sections 3 and 4 is synthesized using `Xilinx Vivado 2018.2` with a clock frequency of 400 MHz. Interfacing with the design is done using an already existing UART component with an AXI4-Lite master output port. This allows for performing benchmarks with the RSA peripheral without needing a dedicated processor in the FPGA. As a development platform the `HTG-K816` board from `HiTech Global` is used. It contains the `XCKU035` FPGA from `Xilinx` with speed grade `-2`. The board is shown in Fig. 5.1.

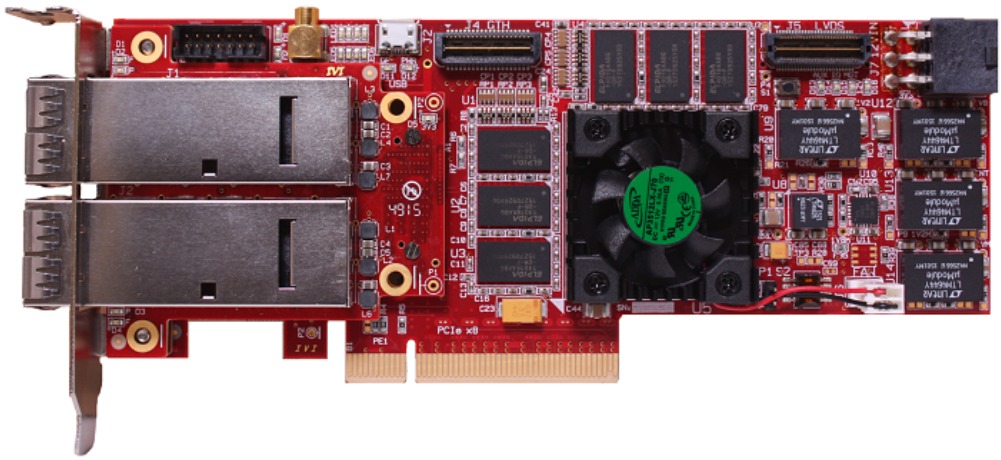


Figure 5.1: The `HTG-K816` prototyping board from `HiTech Global`.

5.1 Area usage

Two designs are synthesized: one using the iterative reduction circuit from Fig. 4.13 and the other using the tree reduction circuit from Fig. 4.11. The reason two designs are used for an implementation is that an interesting trade-off between performance and area usage is expected. Both designs are placed and routed without timing violations. Synthesis of the design results in the area usage given by Table 5.1. In terms of LUTs, the `tree` design is twice as large as the `iter` design. Whether this is acceptable depends on the maximum area requirements and the target FPGA platform, but for the `XCKU035` it is not an issue.

Design	LUT	FF	BRAMs	DSP
`iter`	41857	62459	132.5	485
`tree`	83484	57782	132.5	485

Table 5.1: Area usage for the designs using the tree or iterative reduction circuit.

5.2 Performance

Due to time constraints it was not possible to connect the RSA peripheral to an existing CPU core, so the benchmarks reflect the computational time of the RSA peripheral only. This is not equal to a complete RSA encryption/decryption. However, as most of the operations are executed on the RSA peripheral it still provides a good performance estimate. Another difficulty is that the duration of decryption depends on the value of the decryption key. Besides the fact that this allows for side channel analysis possibly leaking the private key, it makes obtaining a realistic benchmark not equal to performing only one RSA decryption.

So for the benchmark a script has been written that generates 100 random RSA private keys and using those the decryption is performed. For each of those private keys, the number of cycles that the RSA peripheral needs to complete its computation is measured. This is done for sliding window and 2^w-ary exponentiation using all possible window sizes. The average number of cycles for all benchmarks is calculated and the results are shown in Fig. 5.2.

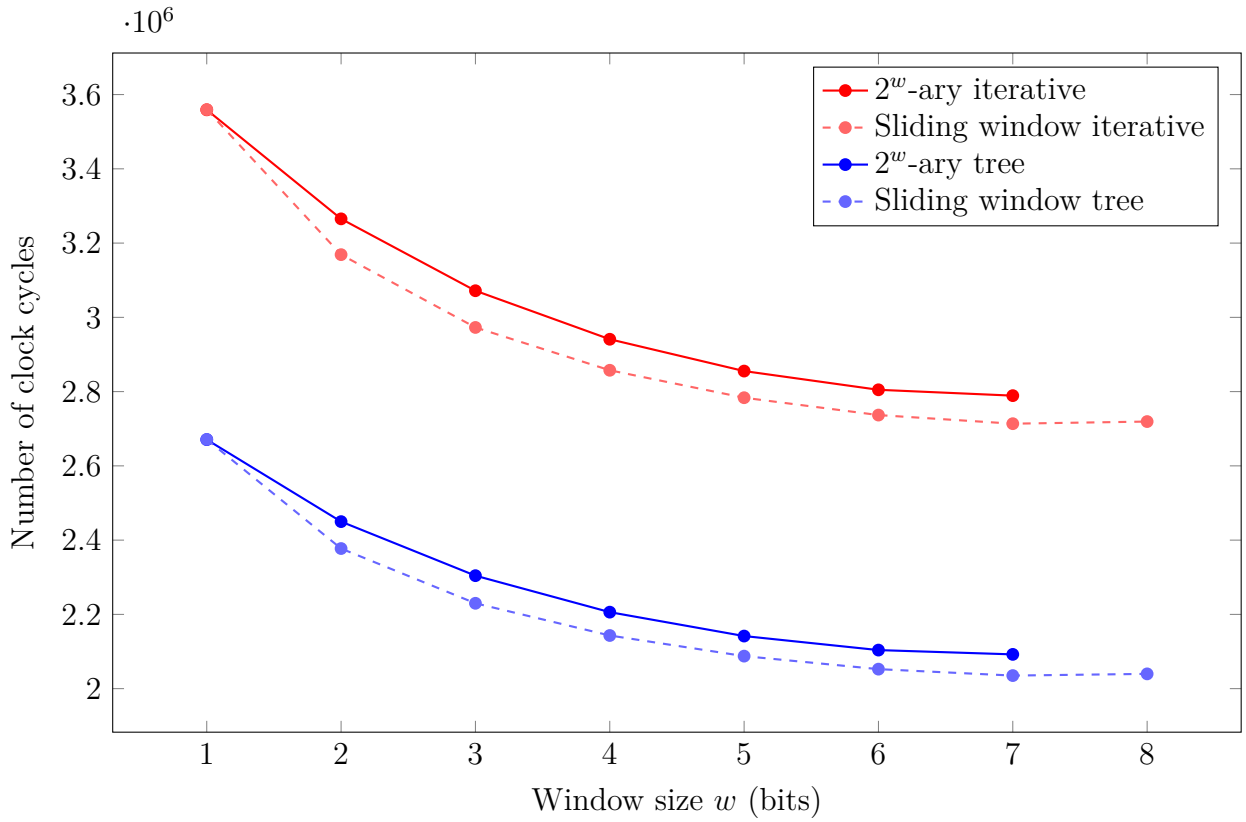


Figure 5.2: Benchmark of modular exponentiation for 100 randomly generated 4096-bit decryption keys.

Using a 99%-confidence interval, the largest margin of error is less than 5000 cycles. Since the margin of error is almost three orders of magnitude smaller the number of cycles, it means that the computed averages are fairly accurate. Taking more samples into the average would

not change the results significantly. The margin of error is the largest for a window size of $w = 1$, which is equal to using binary exponentiation. That makes sense, as for larger window sizes the probability of finding a window containing only zeros gets smaller. This results into a smaller standard deviation in the number of cycles for larger window sizes.

To enable comparison with other designs, it is easier to use the amount of time instead of the number of cycles. Taking the results from Fig. 5.2 and converting those into a time duration value gives Table 5.2. The windows size for the exponentiation methods is limited by the available memory space in the register file of the RNS cores.

Window size w (bits)	Execution time (ms)			
	Sliding window tree	Sliding window iter	2^w-ary tree	2^w-ary iter
1	6.68	8.90	6.68	8.90
2	5.94	7.92	6.12	8.16
3	5.57	7.43	5.76	7.68
4	5.36	7.14	5.52	7.35
5	5.22	6.96	5.35	7.14
6	5.13	6.84	5.26	7.01
7	5.09	6.78	5.23	6.97
8	5.10	6.80	-	-

Table 5.2: Average duration of modular exponentiation for 100 random 4096-bit decryption keys.

Section 3.5 explains that the largest window possible size for 2^w-ary exponentiation is $w = 7$. This gives a maximum window size for sliding window exponentiation of $w = 8$, as that requires half the number of precomputations. However, from Table 5.2 it follows that taking a window this large is actually slower than using $w = 7$. This behavior has already been predicted in Section 2.3.3, but it is good that the benchmark results confirm it.

Verification of the functional behavior is proven by running regression tests with the design loaded into the FPGA. However, in the design phase the functionality of separate parts was verified by simulation. To speed-up development, testbenches are written in `Python` and executed using `cocotb`, which is a library that interfaces with the VHDL simulator. This approach enables co-simulation of a software implementation and the VHDL design. Simulation results give the execution time for a Montgomery Multiplication in RNS with 4096-bit operands, which is shown in Table 5.3.

Design	**Cycles**	**Duration (µs)**
`iter`	577	1.4425
`tree`	434	1.0850

Table 5.3: Execution of one Montgomery Multiplication in RNS on the RSA peripheral.

5.3 Remarks

The time duration during the benchmark does not include some of the computations that the CPU needs to perform. Including these tasks would give a better overview of the total time needed to run one RSA decryption. The first two steps are only for the key generation process and can be omitted if keys are available. The CPU tasks blocking the RSA peripheral from starting are:

- Computation of two 2048-bit random primes p and q.
- Computation of modulus N and decryption key d.
- Computation of $N' = |-N^{-1}|_M$ using `EGCD`.
- Computation of conversion factor $C = M^2 \bmod N$.
- Transfer X, N, N' and C to the RSA peripheral over the AXI4-Lite bus.

Processing of the decryption key and generation of `mont_mult` instructions by the CPU can be done in parallel with the RSA peripheral executing those `mont_mult` instructions. Also, after the RSA peripheral finishes operation some final processing by the CPU is needed:

- Transfer the RNS residues of the result back to the CPU over the AXI4-Lite bus.
- Convert the residues from RNS to binary using CRT with precomputed weights.
- Result $\hat{R}$ has range $[0, kN)$ so one final reduction by modulo N is performed.

As the duration of one RSA decryption cannot be determined without the processor, comparisons with other designs focuses on execution of a single Montgomery Multiplication.

5.4 Comparison

In existing literature it is difficult to find existing implementations of 4096-bit Montgomery Multiplication. Most papers discuss the duration of one multiplication using 1024 bits operands or less. But the proposed design scales better with larger operand sizes, so it would be unfair to compare it with solutions processing less than 4096 bits. However, some papers were found that discuss a 4096-bit operation. An overview is given in Table 5.4.

Design	Platform	Frequency (MHz)	Cycles	Duration (µs)
[34] Architecture 1	`XC2V6000`	116.4	4352	37.397
[35] Compact arch.	`Virtex-7`	210.08	17507*	83.333*
[36] Radix-8 Booth	`EP4SGX230`	337.72	5465	16.18
[this work] `iter`	`XCKU035`	400.0	577	1.4425
[this work] `tree`	`XCKU035`	400.0	434	1.0850

Table 5.4: Comparison of Montgomery Multiplication using 4096-bit operands.

Comparison with other implementations is not always fair, because FPGA technology still improves every year, which allows for higher frequencies and therefore improved performance. It would be interesting so see the benchmarks results of the other implementations using the same technology, but unfortunately that is not possible since the sources are not available.

Another paper discussing the implementation of 4096-bit Montgomery multiplication is [37]. However, in table 2 the duration for 4096 bits is not mentioned, so it cannot be included in Table 5.4. If the number of multiplications is known the duration of a multiplication can be computed from the results for exponentiation. But since many different FPGA platforms and design frequencies are used, the results in tables 2, 3 and 4 cannot be compared with each other. Additionally, sliding window exponentiation is used, but the window size is not mentioned, which makes it impossible to compare the results from this paper to the implementation discussed in this thesis.

*Derived assuming 4096 iterations of two parallel Montgomery multiplications per exponentiation.

6 Conclusion

Acceleration of modular exponentiation using a FPGA platform is a valid strategy for achieving increased performance on an embedded platform. The main goal of this thesis was to design an accelerator that offers great performance, while still keeping the area usage relatively low. Usage of RNS Montgomery multiplication combined with the Bajard and Shenoy base extension algorithms are key building blocks for the design. Two designs are presented: `tree` contains a tree-based reduction circuit and `iter` has an iterative reduction circuit. The first design performs a Montgomery multiplication in 434 cycles and the second in 577 cycles. However, the `tree` design requires twice as many LUTs in comparison the `iter` design.

If these extra area requirements are acceptable depends on whether the extra performance is needed. From the results it follows that area usage is quite large, but considering that the `iter` design contains 121 identical RNS cores, this means that each core uses less than 340 LUTs. A RNS core contains a 34-bit modular adder/subtracter and a modulo circuit providing reduction from 75-bit to a 34-bit number. Combined with a number of multiplexers that also consume LUTs, this result is still an achievement.

6.1 Future work

Although the proposed design works properly, there are still many ways to improve the architecture. These improvements will increase the performance or reduce the area usage, with the latter being the main disadvantage of the design. The following sections propose some ideas for refining the design.

6.1.1 Benchmark complete RSA operation

The current design is able to perform a full exponentiation using multiple iterations of Montgomery multiplication. However, the benchmarks only include the computational time spend by the RSA peripheral. Part of the tasks required to perform one RSA encryption or decryption are performed by a central processing unit that is expected to be available in the system. To allow for comparison with other solutions for a RSA operation, a system should be synthesized that includes the CPU. Once this system is available, new benchmark scores can be measured that should give a better overview of the practical performance.

6.1.2 Update redundant core decoder

The instruction decoder used for the redundant core is currently equal to the decoder for the RNS cores. This makes sense, as it should be able to execute the same instructions. However, as the redundant core performs reduction by a power of two, it does not contain

either the iterative or tree reduction circuit. Performing a right-shift operation is enough to perform reduction. By introducing a decoder specifically for the redundant core, the cycles where it is currently waiting for the reduction to finish could be removed.

6.1.3 Addition of second finish flag

The current design requires that the processor finishes processing the exponent or decryption key before the RSA peripheral can start working. Although the start flag of the peripheral can be raised before completing the generation of `mont_mult` instructions, there is a chance that the FIFO containing the instructions for the RSA peripheral gets empty before the processor finishes execution. When the FIFO is empty, the state machine in the peripheral activates the `Residue reader` and stops processing `mont_mult` instructions. The processing of the exponent is fast, but when the program is paused due to a context switch and the RSA peripheral continues working, the FIFO could get empty. As a safety precaution, the start flag is raised after processing the exponent.

By introducing an extra flag register in the RSA peripheral, the CPU could signal that it is done writing `mont_mult` instructions. The `Instruction generator` now has to wait on both the FIFO being empty and the processor finish flag to be set before signaling it is done to the `RSA controller`. This way, even though the process on the CPU could be paused anytime, the RSA peripheral will wait properly for new incoming instructions. Then the start flag of the RSA peripheral can be raised right after writing the initialization constants, increasing the total performance.

6.1.4 Introduce second redundant core

Recall the RNS processor overview from Fig. 3.2 and the scheduled example program for Montgomery multiplication from Table 3.2. The system uses a redundant core `ReduCore1` for two purposes: generation of intermediate result $\hat{q}_r$ and calculation of correction factor β. Both values are needed to properly execute Shenoy base extension. During computation and transfer of β, all RNS cores are running idle as they require the correction factor before they can can continue. This is also shown by the `MontMult` program from Table A.1 as there exist a lot of `no_operation` instructions in the left column. Evidently, idle time is not useful.

Table 3.2 shows that during both base extensions, the redundant core accumulates the values being passed around on the ring. Introducing a second redundant core `ReduCore2` and attaching that to the opposite side of the ring could speed up this process. Now both redundant cores only have to accumulate half the number of values from the ring. Next, `ReduCore2` writes its accumulated result to the `next_core` output. By connecting this port to the input of the register file in `ReduCore1`, the value can be transferred. Only one modular addition of the received value and the accumulated value in `ReduCore1` is needed to get result $\hat{q}_r$. This can be done before the Bajard base extension completes on the RNS cores.

The same process can be performed during the Shenoy base extension, to speed up the calculation of g (line 20 of Table 3.2) on the redundant core. Introduction of a second redundant core makes that the value for g is available in about $\lceil k/2 \rceil = 61$ cycles for a system with $k = 121$ cores. From this it follows that `ReduCore1` now has $121-61 = 60$ cycles available for computation and broadcasting of β, which should be realistic as that currently takes 43 cycles. Using an extra redundant core `ReduCore2` will eliminate all `no_operation` instructions performed by the RNS cores, although some extra area is needed for that. This change is expected to save around 20 cycles per Montgomery multiplication.

6.1.5 Compute modular inverse in RNS

From Section 5.3 it follows that some operations for performing a full RSA operation are executed on the processor. Out of this list of operations, the one for which the most arithmetic operations is likely needed is the computation of $N' = |-N^{-1}|_M$. In order to solve N', the processor needs to perform modular inversion of two numbers a few bits larger than 4096 bits. One method of performing modular inversion is using the Extended Greatest Common Denominator `EGCD` algorithm. Assuming that the processor runs at a lower frequency than the RSA peripheral, this may even take longer than a 4096-bit modular exponentiation.

Therefore it would be good to move this computation to the peripheral, but some modifications are required before it is able to perform modular inversion in RNS. An iterative approach like using the `ECGD` does not map to the RSA peripheral, as it does not contain instructions for performing division. Also, the RSA peripheral containing 121 cores with different moduli values, modular inversion using `ECGD` does not take an equal amount of cycles per core. As all RNS cores are connected the same instruction line, so it is not possible to let the cores execute different instructions.

Luckily there is a paper [38], which provides inspiration for a solution. Instead of using `ECGD`, Fermat's Little Theorem (FLT) could be used. FLT says that the modular inversion of a with respect to p with p prime can be computed by calculating $|a^{p-2}|_p$. So instead of division, we perform modular exponentiation and that exactly is what the RSA peripheral is made for. Since all moduli in $\mathcal{B}$ are prime FLT can be used to compute the modular inversion with respect to M. Conversion of $-N$ to RNS base $\mathcal{B}$ gives resides $|-n_i|_{m_i}$. Substitution of these residues in FLT gives a formula for the modular inversion that can be solved in parallel:

$$\texttt{modinv}(-n_i, m_i) = |-n_i^{-1}|_{m_i} = |-n_i^{m_i-2}|_{m_i} \quad \text{for} \quad i \in \{1, ..., k = 121\} \quad \text{cores} \tag{6.1}$$

However, there is a problem with this method, because the values $p = m_i$ are different for each core. Calculation of the exponent $m_i - 2$, using binary exponentiation, would mean that some cores should perform a multiplication in a certain cycle and others not, depending on the value of their modulus m_i. As all moduli in $\mathcal{B}$ minus two are values of 34 bits, the maximum possible number of cycles is equal though. So a small modification to the RNS

cores is proposed, which would allow the cancellation of the multiplication step in binary exponentiation for when the current bit in the exponent is zero.

Each of the RNS cores is given a circular shift register containing the value $m_i - 2$. A new control signal `mask_write_en` is added to the set from Table 4.3 and connected to the clock enable of the circular shift register. This signal is also combined with the `write_enable` of the register file using combinatorial logic. When `mask_write_en` asserts, the `write_enable` is routed through an `and` gate with the MSB of the shift register. It gives the ability to cancel the write-back of an operation to the register, based on a bit from exponent $m_i - 2$.

Next, an additional instruction is needed to flag multiplications with this new control signal. Recall the instruction set in Table 3.6, from this set the instruction `read_prop` is never used. By replacing this instruction with a regular multiplication that performs reduction by m_i, the new functionality can be enabled. This new instruction is called `mult_m0_mask`, which performs exactly the same sequence as `mult_m0`, but has the `mask_write_en` flag asserted. Now issuing a sequence of exactly 34 normal multiplications for squaring and 34 multiplications that can be canceled, results in the modular inversion of a register.

6.1.6 Reducing block RAM usage

From Table 5.1 it follows that the number of BRAMs used in the design is quite large. Most BRAMs are used as register file for the RNS cores. The design contains $k = 121$ cores and each of them uses one `RAMB36E2` primitive. Each of these primitive contains 1024 entries for storing data when using a word size of 34 bits [32]. However, from Section 3.5 follows that the maximum number of storage locations used is 512. This means that only half of the available memory is used and potentially the number of BRAMs can be halved.

However, the utilized synthesis tools only use a `RAMB36E2` primitive instead of a `RAMB18E2`. The reason is that the register files use two independent read ports of 34 bits width and that is not supported by the `RAMB18E2` primitive. A solution for reducing the memory footprint is to replace the true dual port register with a single port one. Then a smaller BRAM primitive can be used for each register file, halving the required number of memory blocks. Of course this has impact on the performance, as many operations require two operands. But this can also be implemented using an extra register stage to store the first memory lookup and then reading the second operand in the next cycle.

From Appendix A.3, it follows that not all computations require two operands, many use zero for `read_addr1`. This property can be taken advantage of by adding a reset signal to the control set from Table 4.3, which allows the lookup register to reset to zero when the address also equals zero. It makes it so that only instructions using two operands of which the first has a non-zero address incur an extra memory lookup cycle. Mapping this to the `MontMult` program shows that only for 12 instruction this extra cycle is needed. So the usage can be improved by replacing the register file with a single port variant and adding 12 cycles per Montgomery multiplication.

6.1.7 8K and 16K bits modular exponentiation

The design represents the intermediate values for the Montgomery multiplication using small RNS residues of 34 bits each. This approach makes it easy to scale up the design for usage with 8192 or even 16384-bit operands. It would be interesting to see what the performance of using those operands will be. The only limit is the amount of resources available on the FPGA platform. Scaling up can be done by adding more cores and that will not influence the frequency as those cores are only connected to their neighbors.

The number of cycles to perform one Montgomery multiplication using larger operands can be estimated, as adding more cores only increases the duration of the two base extensions. From Table 5.3 it follows that one Montgomery multiplication with the `tree` design takes 434 cycles. As the system contains $k = 121$ cores, there are $2 \cdot 121 = 242$ cycles needed for the two base extensions. This leaves $434 - 242 = 192$ cycles for the other computations, which will not change as the number of cores increases. For 8192-bit exponentiation using 34-bit moduli, a total of $\lceil 8192/34 \rceil = 241$ cores are needed. One Montgomery multiplication in this system would then take $2 \cdot 241 + 192 = 674$ cycles. Correspondingly, 16384-bit exponentiation will need 482 cores and a Montgomery multiplication takes 1156 cycles.

Next, Eq. (2.7) is used to find the optimal window size $w = 8$ and the expected average number of multiplications. Using these values combined with the expected number of cycles in one Montgomery multiplication, an estimation of the performance is given in Table 6.1.

Operand size	Multiplications	Total cycles	Duration (ms)
8192	9466	6380084	15.95
16384	18678	21591768	53.98

Table 6.1: Estimated performance of RNS Montgomery exponentiation for 8K and 16K bits operands.

It should be noted that additional memory is needed for storing all the required precomputed constants used in the two base extensions. The optimal window size $w = 8$ is one larger than for 4096-bit exponentiation, which doubles the memory requirement for these precomputations. Although it is always an option to reduce the window size if the register file cannot be made larger. In addition, the area usage will be a major problem, especially when using the tree design.

Acronyms

BRAM	Block RAM
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DSP	Digital Signal Processing
EGCD	Extended GCD
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GCD	Greatest Common Denominator
LSB	Least Significant Bit
LUT	Lookup Table
MAC	Multiply-Accumulate
MM	Montgomery Multiplication
MSB	Most Significant Bit
MSD	Most Significant Digit
RAM	Random Access Memory
RNS	Residue Number System
ROM	Read-Only Memory
RSA	Rivest Shamir Adleman
VPN	Virtual Private Network

Appendix

A Design parameters

A.1 RNS primes first base

For $k = 121$, the list of prime moduli $m_i = \{m_1, ..., m_k\}$ in $\mathcal{B}$ is given by

```
17179869107 17179869053 17179869019 17179868977 17179868903 17179868887 17179868873
17179868861 17179868833 17179868807 17179868759 17179868711 17179868681 17179868549
17179868521 17179868479 17179868437 17179868383 17179868357 17179868351 17179868317
17179868297 17179868249 17179868183 17179868087 17179867997 17179867937 17179867829
17179867807 17179867759 17179867721 17179867709 17179867571 17179867547 17179867493
17179867487 17179867423 17179867363 17179867333 17179867307 17179867301 17179867291
17179867219 17179867193 17179867141 17179867129 17179867057 17179866971 17179866931
17179866917 17179866883 17179866857 17179866833 17179866773 17179866727 17179866653
17179866569 17179866527 17179866421 17179866397 17179866359 17179866287 17179866241
17179866173 17179866163 17179866127 17179866071 17179866023 17179865993 17179865929
17179865887 17179865827 17179865819 17179865767 17179865753 17179865743 17179865701
17179865671 17179865609 17179865543 17179865503 17179865467 17179865453 17179865411
17179865401 17179865371 17179865317 17179865299 17179865257 17179865239 17179865119
17179865081 17179865063 17179865051 17179865029 17179864993 17179864979 17179864931
17179864841 17179864787 17179864763 17179864729 17179864717 17179864693 17179864609
17179864571 17179864529 17179864483 17179864447 17179864417 17179864379 17179864343
17179864327 17179864217 17179864211 17179864181 17179864147 17179864073 17179864043
17179864021 17179863971
```

The product of these moduli is range $M = \prod_{j=1}^{k} m_i$, which is given by

```
27377503024592940501810791217667313160033564808083247837747224295359531880235250061
81286003640993113496268702802728580245484913472570193701239931207739558631935155568
21211218111714656503311630079787301435753475957018586518480717295367357721781709032
09838040212212484716191951061604961883156272765053690086404111644830582188676028993
90148893078614671759599247798047505127794945448878163807095872730048441982728520875
79027393633522378691519007312725377951511403314747352208904232517542805187509269850
82817638580265850002381252587583805691315906333935164229801178588795296224184569153
72225319102817177116606085962230122388177965959700176478616021757567722658008270072
41153075856880675252073659233624372102587486676432610714760794305066382949414048870
80391120152611520484885216222547918472306769605178671202574475271558598637748898046
77265163479468169090638626678119742530653334085962136854832283009418335809400035088
53019153609642215657303497343155299792228312446899801637118860960649426403646795165
81884985254015225249550786408043840916115395532295754578943752434430184504668161469
25161720879009105139042994715168091220570379195502465005772101791939299264365958697
33845834203686379717782490228894995882540012394295541694200396804111471114091
```

For 4096-bit RSA encryption, range M should be able to contain numbers of

$$4096 + \lceil \log_2((k+2)^2) \rceil = 4096 + \lceil \log_2(15129) \rceil = 4096 + 14 = 4110 \text{ bits}$$

Range M has a length of 4114 bits, so the required lower bound is satisfied.

A.2 RNS primes second base

For $k = 121$, the list of prime moduli $m_j = \{m_{k+1}, ..., m_{2k}\}$ in $\widetilde{\mathcal{B}}$ is given by

```
17179869143 17179869071 17179869041 17179868999 17179868957 17179868899 17179868879
17179868869 17179868843 17179868809 17179868777 17179868729 17179868683 17179868597
17179868543 17179868513 17179868443 17179868429 17179868369 17179868353 17179868333
17179868309 17179868287 17179868243 17179868141 17179868081 17179867951 17179867909
17179867819 17179867781 17179867753 17179867717 17179867673 17179867567 17179867501
17179867489 17179867433 17179867421 17179867349 17179867321 17179867303 17179867297
17179867223 17179867217 17179867163 17179867139 17179867093 17179867049 17179866959
17179866923 17179866889 17179866881 17179866853 17179866787 17179866761 17179866667
17179866637 17179866553 17179866499 17179866401 17179866383 17179866323 17179866251
17179866217 17179866167 17179866157 17179866091 17179866049 17179866019 17179865977
17179865921 17179865849 17179865821 17179865777 17179865761 17179865749 17179865713
17179865693 17179865621 17179865581 17179865519 17179865477 17179865459 17179865431
17179865407 17179865399 17179865347 17179865309 17179865273 17179865251 17179865221
17179865093 17179865077 17179865057 17179865039 17179864997 17179864987 17179864951
17179864861 17179864837 17179864783 17179864759 17179864721 17179864703 17179864661
17179864579 17179864547 17179864517 17179864463 17179864439 17179864399 17179864349
17179864333 17179864279 17179864213 17179864183 17179864159 17179864123 17179864069
17179864033 17179863977
```

The product of these moduli is range $\widetilde{M} = \prod_{j=1}^{k} m_{k+j}$, which is given by

```
27377507512116423392208906289419363485245399529533780754564292275539014900553282904
44636049730437878728699867817912033015330721284296394915568234936205959531269995080
96089049055620262751890380065744372110376434660389206836690419391323130619875582322
72804191911425397174371565811324401369179601948902515809314940535367468029713518352
66710519142346565181938237529036440281513558769737675328196838358585827201624672488
87338327488155546357595010994543853977689855611324598307300420112322340764723217882
78003506287135933633902176262836101082267594108994488323845365427337754186183529611
86026885948538046804682730961362681293137671607042772194076159546905143964412342271
18624561805954954724309350856520544158999818239940461489220075073101332310959491753
92082334361538681756629921050792620395573575192381168664157955624642381820382934997
36006392568584809512591594766178051753810202437731460070722199941640407865611892974
86809584663559554451696684495915334365449694398485701966452429075175374042465562151
37860162611910704678747752530531277595705228882975061063928644153565689933622722250
37527080347441180837970496840239943399242498208447074787211445470067840856949637618
54469355621981456880270268414466676606347956347310724566012968606394298445211
```

For 4096-bit RSA encryption, range $\widetilde{M}$ should be able to contain numbers of

$$4096 + \lceil \log_2(k+2) \rceil = 4096 + \lceil \log_2(123) \rceil = 4096 + 7 = 4103 \text{ bits}$$

Range $\widetilde{M}$ has a length of 4114 bits, so the required lower bound is satisfied.

A.3 MontMult program

The microcode located in the RSA peripheral, stored in read-only memory and used for performing RNS Montgomery Multiplication on the RNS cores is shown in Table A.1. Registers 4, 5, 6 and 7 are replaced based on the values of b and n in the `mont_mult` instruction.

Core instruction	wr	rd1	rd2	Redu instruction	wr	rd1	rd2
mult_m0	1	1	4	no_operation	0	0	0
mult_m0	1	1	5	no_operation	0	0	0
mult_m0	1	1	266	no_operation	0	0	0
multiply	0	1	267	multiply	0	0	0
mult_accum_p	0	0	268	mult_accum_p	0	0	267
mult_accum_p	0	0	269	mult_accum_p	0	0	268
mult_accum_p	0	0	...	mult_accum_p	0	0	...
mult_accum_p	0	0	386	mult_accum_p	0	0	385
mult_accum_p	0	0	387	mult_accum_p	0	0	386
no_operation	0	0	0	mult_accum_p	0	0	387
mult_read_m1	1	0	0	mult_read_m1	1	0	0
mult_m1	134	134	6	mult_m1	134	134	6
mult_m1	1	1	7	mult_m1	1	1	7
addu_m1	1	1	134	addu_m1	1	1	134
mult_m1	134	1	511	mult_m1	134	1	511
mult_m1	1	134	388	no_operation	0	0	0
multiply	0	1	389	multiply	0	0	0
mult_accum_p	0	0	390	mult_accum_p	0	0	389
mult_accum_p	0	0	391	mult_accum_p	0	0	390
mult_accum_p	0	0	...	mult_accum_p	0	0	...
mult_accum_p	0	0	508	mult_accum_p	0	0	507
mult_accum_p	0	0	509	mult_accum_p	0	0	508
no_operation	0	0	0	mult_accum_p	0	0	509
mult_read_m0	1	0	0	mult_read_m0	1	0	0
no_operation	0	0	0	subu_m0	133	1	134
no_operation	0	0	0	mult_m0	133	133	510
no_operation	0	0	0	multiply	0	133	0
no_operation	0	0	0	no_operation	0	0	0
no_operation	0	0	0	no_operation	0	0	0
no_operation	0	0	0	no_operation	0	0	0
no_operation	0	0	0	no_operation	0	0	0
no_operation	0	0	0	no_operation	0	0	0
no_operation	0	0	0	no_operation	0	0	0
no_operation	0	0	0	no_operation	0	0	0
bcas_read	133	0	0	no_operation	0	0	0
mult_m0	133	133	510	no_operation	0	0	0
subu_m0	1	1	133	no_operation	0	0	0

Table A.1: RNS Montgomery multiplication program for $k = 121$ RNS cores.

A.4 Instruction timing

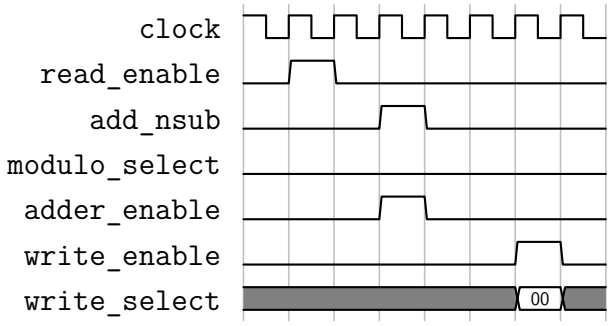


Figure A.1: Control signals for `addu_m0`.

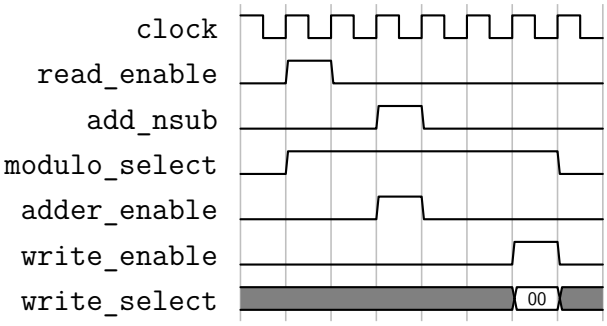


Figure A.2: Control signals for `addu_m1`.

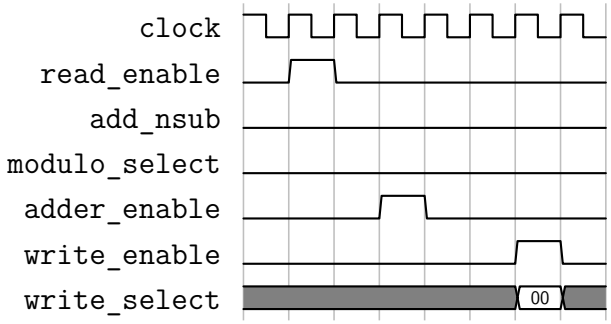


Figure A.3: Control signals for `subu_m0`.

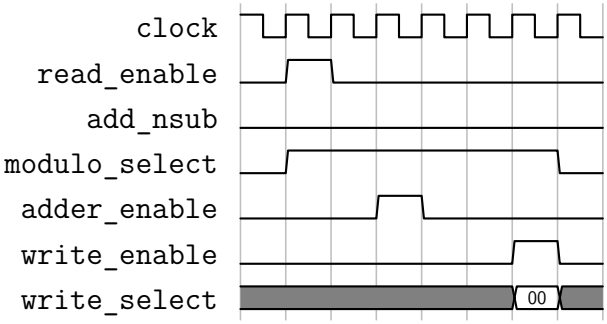


Figure A.4: Control signals for `subu_m1`.

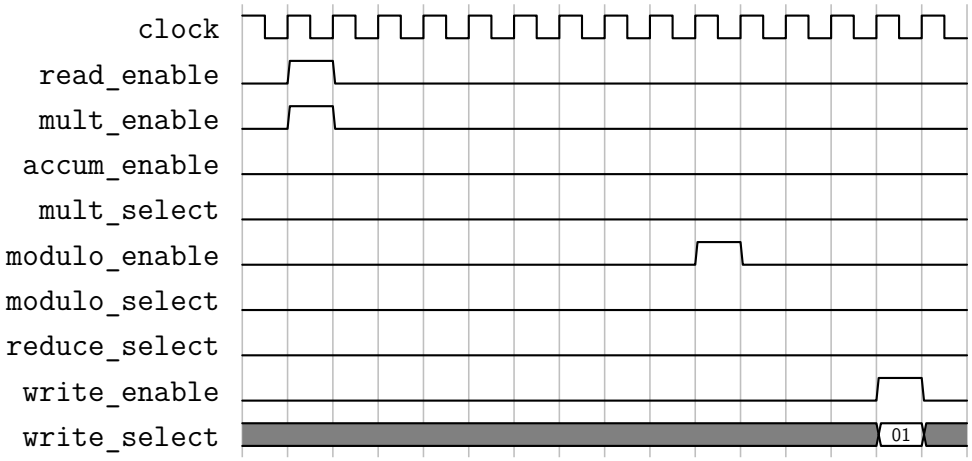


Figure A.5: Control signals for `mult_m0`.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: https://people.csail.mit.edu/rivest/Rsapaper.pdf

[2] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, pp. 342–345.

[3] D. Boneh, "Twenty years of attacks on the rsa cryptosystem," *Notices of the American Mathematical Society (AMS)*, vol. 46, no. 2, pp. 203–213, 1999. [Online]. Available: https://www.ams.org/journals/notices/199902/boneh.pdf

[4] M. J. Wiener, "Cryptanalysis of short rsa secret exponents," *IEEE Transactions on Information theory*, vol. 36, no. 3, pp. 553–558, 1990.

[5] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," in *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 1994, pp. 92–111.

[6] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik *et al.*, "Factorization of a 768-bit rsa modulus," in *Annual Cryptology Conference*. Springer, 2010, pp. 333–350.

[7] P. A. Mohan, *Residue Number Systems: Theory and Applications*. Birkhäuser Basel, 2016, pp. 95–97. [Online]. Available: https://doi.org/10.1007/978-3-319-41385-3

[8] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, pp. 461–481.

[9] P. Downey, B. Leong, and R. Sethi, "Computing sequences with addition chains," *SIAM Journal on Computing*, vol. 10, no. 3, pp. 638–646, 1981.

[10] D. M. Gordon *et al.*, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, no. 1, pp. 129–146, 1998.

[11] C. K. Koç, "Analysis of sliding window techniques for exponentiation," *Computers & Mathematics with Applications*, vol. 30, no. 10, pp. 17–24, 1995. [Online]. Available: https://doi.org/10.1016/0898-1221(95)00153-P

[12] Wikipedia contributors, "Exponentiation by squaring — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?oldid=868883860

[13] J. Bos and M. Coster, "Addition chain heuristics," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 400–407.

[14] Intel, "Intel® 64 and ia-32 architectures software developer's manual," 2016. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[15] Advanced Micro Devices, "Amd64 architecture programmer's manual volume 3: General-purpose and system instructions," 2018, revision 3.26. [Online]. Available: https://www.amd.com/system/files/TechDocs/24594.pdf

[16] B. Parhami, *Algorithms and Design Methods for Digital Computer Arithmetic*. Oxford University Press, 2012.

[17] S. Kawamura and K. Hirano, "A fast modular arithmetic algorithm using a residue table," in *Lecture Notes in Computer Science on Advances in Cryptology-EUROCRYPT'88*. Springer-Verlag, 1988, pp. 245–250. [Online]. Available: https://doi.org/10.1007/3-540-45961-8_21

[18] B. Parhami, "Analysis of tabular methods for modular reduction," in *Proceedings of 1994 28th Asilomar Conference on Signals, Systems and Computers*. IEEE, 1994, pp. 526–530.

[19] Z. Cao, R. Wei, and X. Lin, "A fast modular reduction method." *IACR Cryptology ePrint Archive*, vol. 2014, p. 40, 2014.

[20] C. H. Lim, H. S. Hwang, and P. J. Lee, "Fast modular reduction with precomputation," in *Proceedings of Korea-Japan Joint Workshop on Information Security and Cryptology (JWISC'97)*, 1997, pp. 65–79.

[21] M. A. Will and R. K. L. Ko, "Computing mod with a variable lookup table," in *Security in Computing and Communications*, P. Mueller, S. M. Thampi, M. Z. Alam Bhuiyan, R. Ko, R. Doss, and J. M. Alcaraz Calero, Eds. Springer Singapore, 2016, pp. 3–17. [Online]. Available: https://doi.org/10.1007/978-981-10-2738-3_1

[22] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[23] K. C. Posch and R. Posch, "Modulo reduction in residue number systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 5, pp. 449–454, 1995.

[24] A. Shenoy and R. Kumaresan, "Fast base extension using a redundant modulus in rns," *IEEE Transactions on Computers*, vol. 38, no. 2, pp. 292–297, 1989.

[25] J.-C. Bajard, L.-S. Didier, and P. Kornerup, "Modular multiplication and base extensions in residue number systems," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001.* IEEE, 2001, pp. 59–65.

[26] N. S. Szabo and R. I. Tanaka, *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.

[27] J.-C. Bajard and L. Imbert, "A full rns implementation of rsa," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 769–774, 2004.

[28] J.-C. Bajard, J. Eynard, and N. Merkiche, "Montgomery reduction within the context of residue number system arithmetic," *Journal of Cryptographic Engineering*, vol. 8, no. 3, pp. 189–200, 2018.

[29] N. Cucu-Laurenciu, "Rns support for rsa cryptography," Master's thesis, Delft University of Technology, 2010.

[30] J.-C. Bajard, M. Kaihara, and T. Plantard, "Selected rns bases for modular multiplication," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*. IEEE, 2009, pp. 25–32.

[31] Xilinx, *UG579 UltraScale Architecture DSP48E2 Slice*, v1.7. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

[32] ——, *UG573 UltraScale Architecture Memory Resources*, v1.10. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

[33] S. Kawamura, M. Koike, F. Sano, and A. Shimbo, "Cox-rower architecture for fast parallel montgomery multiplication," in *Advances in Cryptology — EUROCRYPT 2000*, B. Preneel, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 523–538.

[34] M. Huang, K. Gaj, and T. El-Ghazawi, "New hardware architectures for montgomery modular multiplication algorithm," *IEEE Transactions on computers*, vol. 60, no. 7, pp. 923–936, 2010.

[35] L. Rodriguez-Flores, M. Morales-Sandoval, R. Cumplido, C. Feregrino-Uribe, and I. Algredo-Badillo, "A compact fpga-based microcoded coprocessor for exponentiation in asymmetric encryption," in *2017 IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE, 2017, pp. 1–4.

[36] C. P. Rentería-Mejía, V. Trujillo-Olaya, and J. Velasco-Medina, "8912-bit montgomery multipliers using radix-8 booth encoding and coded-digit," in *2013 IEEE 4th Latin American Symposium on Circuits and Systems (LASCAS)*. IEEE, 2013, pp. 1–4.

[37] T. Wu, S. Li, and L. Liu, "Fast rsa decryption through high-radix scalable montgomery modular multipliers," *Science China Information Sciences*, vol. 58, no. 6, pp. 1–16, Jun 2015. [Online]. Available: https://doi.org/10.1007/s11432-014-5215-4

[38] K. Bigou and A. Tisserand, "Improving modular inversion in rns using the plus-minus method," in *Cryptographic Hardware and Embedded Systems - CHES 2013*, G. Bertoni and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 233–249. [Online]. Available: https://eprint.iacr.org/2015/193.pdf