POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Implementation of the comparison in Residue Numeral System

Supervisor

Prof. Guido MASERA

Candidate

Maria Francesca CASCONE

July 2022

Summary

Residue Number Systems, RNSs, have been considered a potential tool to parallelize the arithmetic elements breaking the long carry-propagation chain by bounding the RNS inside smaller modulo channels that work in parallel with each other. This parallelism is quite profitable for addition and multiplication and has made possible the usage of RNS in a range of application from embedded and digital signal processing systems to cryptography [1].

However, some operations are difficult to perform using RNS: its non-positional representation makes hard to implement the comparison operation, indeed the existing comparison methods compare the RNS numbers by converting them in positional numeral system.

The *Positional Attribute Non-positional Code*, PANC, method is a mathematical algorithm that allows the comparison in RNS without converting the numbers under consideration. It associates the RNS number to an index that identifies the interval in which the number falls and then performs the comparison between the two indexes associated to both inputs [2].

After a brief introduction to the RNS considering its advantages, disadvantages and applications in Chapter 1, it is fully described in Chapter 2. The different comparison methods along with the PANC one are reported in Chapter 3. Chapter 4 aims to portray the implemented architectures in detail, but their results in terms of simulation in **ModelSim** and synthesis with **Synopsys Design Compiler**, using a 65 nm technology, are both reported in the first part of Chapter 5, while in the second one those new architectures are compared with the state of the art. Finally, the future implementations together with the conclusion are in Chapter 6.

For openers, it must be pointed out that RNS is a mathematical non-positional method based on the Chinese Remainder Theorem, CRT. This last one takes credit to the reduction of the carry-propagation chain speeding up the arithmetic comparison- and division- free operations. This reduction is achieved since the RNS operands are tuples of residues associated to a proper modulus in the moduli set, and therefore, given that the operations are performed on the tuples' components and not on the whole tuple itself, those are going to have a reduced bit-width.

The addition, multiplication and subtraction operations are carried out only on residues associated with the same modulus, generating results that could be negative or higher than the modulus itself. As a consequence, since the residues are part of the tuple and that they are associated with each modulus, these two conditions can't occur, but if this should happen, the resultant residues will be reconverted. This has led to seek better solutions in the implementation of those combinatorial elements.

Moreover, it is necessary to convert and reconvert numbers from a positional numeral system to RNS, for this reason several methods of comparison have been studied in order to obtain the most convenient in terms of performance.

Even if the reconversion methods have been used also for the comparison operation in RNS, there are no implemented algorithms that performs the comparison in the RNS' domain. The PANC method is one of those algorithms.

However, nowadays this mathematical method has not been implemented yet, indeed this work aims to create a digital design to transpose this method and analyse its performances. The considered moduli set is $[2^n - 1,\ 2^n,\ 2^n + 1]$ since it is the most used set in RNS' datapaths [3], while the n values considered are $n = 5$ and $n = 8$, together with $n = 3$ even if it is not usually adopted for RNS datapaths. Several solutions have been evaluated in order to implement the digital design, nevertheless only the most interesting ones have been selected. In particular, these implemented structures are:

- The *Golden model* is the architecture that parallelize all the operations, thus it is very attractive because it is the fastest one. In fact, the latency of the operation doesn't depend on the value n or on the moduli set and yet, those last values affect the area occupied by the structure that exponentially increases with n.

- The *Resource sharing* design is the opposite of the previously described golden model as the resources used are fixed and do not change either as n or the moduli set vary, but the latency depends on those values giving us a very slow architecture.

- Finally, the *Unfolded resource sharing* wants to combine the positive sides of both previous architectures applying the technique of the *unfolding* to the resource sharing design. In this way, the resources are increased by 2^k according to the unfolding level, hence slightly complicating the structure but proportionally reducing the delay.

Those three designs have been analysed in terms of delay, area occupied and power consumption considering a 65 nm technology.

Comparing the performances of these three implementations with the state of the art of the actual comparators in RNS, it is clear that our comparators are slower and wider than the existent implementations, although the total power dissipation is lower than the those last ones by at least one order of magnitude.

This is due not only to the lower operating frequency of our implementations but also to the management of the different combinatorial elements that are totally switched-off when not in use. Indeed, even if the increased area affects the power increase, the strong reduction of the switching activity allows to obtain good results, offsetting both area and delay drawback.

Finally, this thesis shows how it is possible, starting by this algorithm, to implement a comparison system not leaving the residue numbering system. As well as the low-power advantages that allows to apply this comparator in energy-saving systems so as IoT, the lack of a reconvertion system allows to use the same comparator in datapath RNS taking advantage of the same components again.

Acknowledgements

Se insisti e resisti,
raggiungi e conquisti

Table of Contents

List of Tables

List of Figures

Acronyms

RNS
Residue Number System

CRT
Chinese Reminder Theorem

MRC
Mixed Radix Conversion

VLSI
Very Large Scale Integration

FPGA
Field-Programmable Gate Array

ASIC
Application-Specific Integrated Circuit

TPP
Totally Parallel Prefix

RPP
Regular Parallel Prefix

Chapter 1

Introduction

1.1 Motivation

The *Residue Number System* is a mathematical method invented in 1959 based on the *Chinese Remainder Theorem*, CRT. The CRT has detected some interesting number theoretic properties and features that can speed up some operations by reducing the the carry-propagation chain which is the main bottleneck of fast arithmetic operations [4]. The RNS operands have reduced bit-width with respect of the binary ones since the operations are performed on the residues associated to the division of each modulus in a specific moduli set. In this way the actual bit-width depends on the modulus value which cannot be exceeded so that all the operations conducted in each modulus channel are independent and carry-free. From 1961, the RNS features have been used in design digital systems division and comparison free, finding out a great speed advantage and motivating researchers to use RNS to improve the speed for digital systems that performs difficult operations.

Since the RNSs lead to increasing performances and reducing hardware cost [5], those systems are widely applied in digital signal processing, medical imaging [6] and artificial neural networks [7]. The RNS' modularity offers advantages for all the division and comparison free algorithms because of the progresses in the architectures of addition [8], subtraction and multiplication [9].

However, the operations of division and comparison are still sore points in the RNSs' structures. About the comparisons methods, those operations are supported with converting structures in order to convert the non-positional RNS' numbers in positional ones, but nowadays it has not been implemented a digital system that performs the comparison in RNS without converting the values. Nonetheless there are algorithms that allows to perform the RNS comparison without converting the values. One of those algorithm has been exposed in [2], but never implemented with digital systems.

This work aims to create a possible digital design to compare two RNSs numbers using the proposed mathematical algorithm and evaluate its performances in terms of delay, power consumption and occupied area.

1.2 Work flow

To implement the algorithm proposed in [2], it is first important to understand the RNS operations and generations also referring to open-source tools as seen in Chapter 2. In this way it becomes easier to figure out the mathematical algorithm and the necessary elements to perform the comparison with this new method as reported in 3.2. After the stabilization of the algorithm using **Python** and the selection of the moduli set, all the possible structures have been designed on paper in order to choose the best designs to implement.

Three structures have been selected and described in **VHDL**: the first and the second ones are respectively the fastest but widest and the slightest but slowest, so the third architecture is a trade-off between those two designs applying the *unfolding method* to the second design. All the structures have been analysed in Chapter 4, while in Chapter 5 are reported the related results in **ModelSim** together with the synthesized ones with **Synopsys Design Compiler** and the comparison between those new implementations with respect of the existing ones, exploited in 3.1.

The other possible choices and the ideas for future implementations are reported in Chapter 6, together with the final conclusions.

Chapter 2

Residue Number System method

An RNS is characterized by a set of N relatively coprime numbers known as moduli m_i with $i = 1,2, \ldots, N$ to represent all the numbers in the dynamic range M equal to the multiplication between all the used moduli m_i, as reported in Equation (2.1) [4].

$$M = \prod_{i=1}^{N} m_i \tag{2.1}$$

Considering a number X, the non-negative reminders x_i associated to the division $\frac{X}{m_i}$ represent the RNS value as $X_{RNS} = [x_1, x_2, \ldots, x_N]$ with the moduli set $[m_1, m_2, \ldots, m_N]$.

If X is an unsigned number, then it is represented in the range $[0, M)$, while if it is a signed number it falls in the range $[0, \frac{M}{2})$ if X is positive, or in $[\frac{M}{2}, M)$ if it is negative [1]. To better understand their representation, we may refer to Figure 2.1.

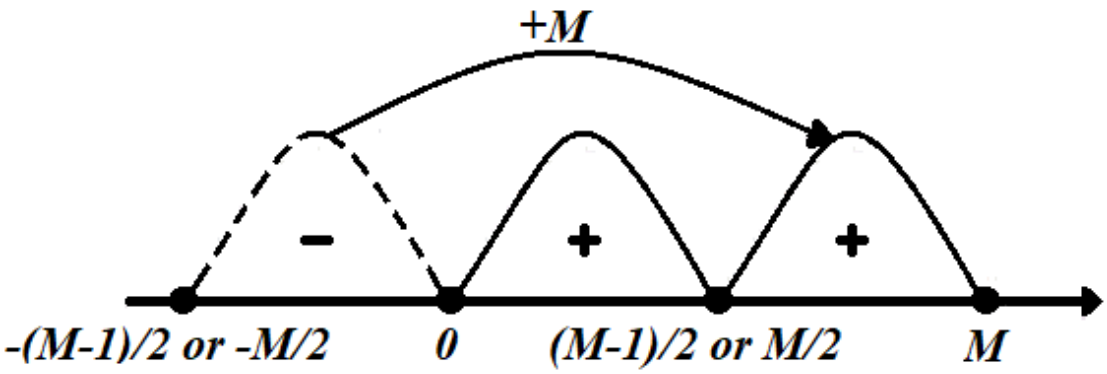

Figure 2.1: Representation of the signed RNS numbers in range M.

It is evident that all the residues are non-negative and cannot exceed the associated modulus value, but rather if those two conditions do not occur those values must be reconverted, as exploited in detail in 2.2.

The RNS is a non positional numeral system in which there is not a weight associated with any digit, however its tuple representations as $X_{RNS} = [x_1, x_2, \ ... \ , x_N]$ allows to perform each operations in parallel across smaller modulus channels.

Indeed, in Figure 2.2 is reported an overview of an RNS based application: at first the positional values are converted in RNS with a *Forward Converter*, then in a *RNS Processing Unit* the residues associated to each modulus are menaged and finally are reconverted thanks to the *Reverse Conversion* [4].

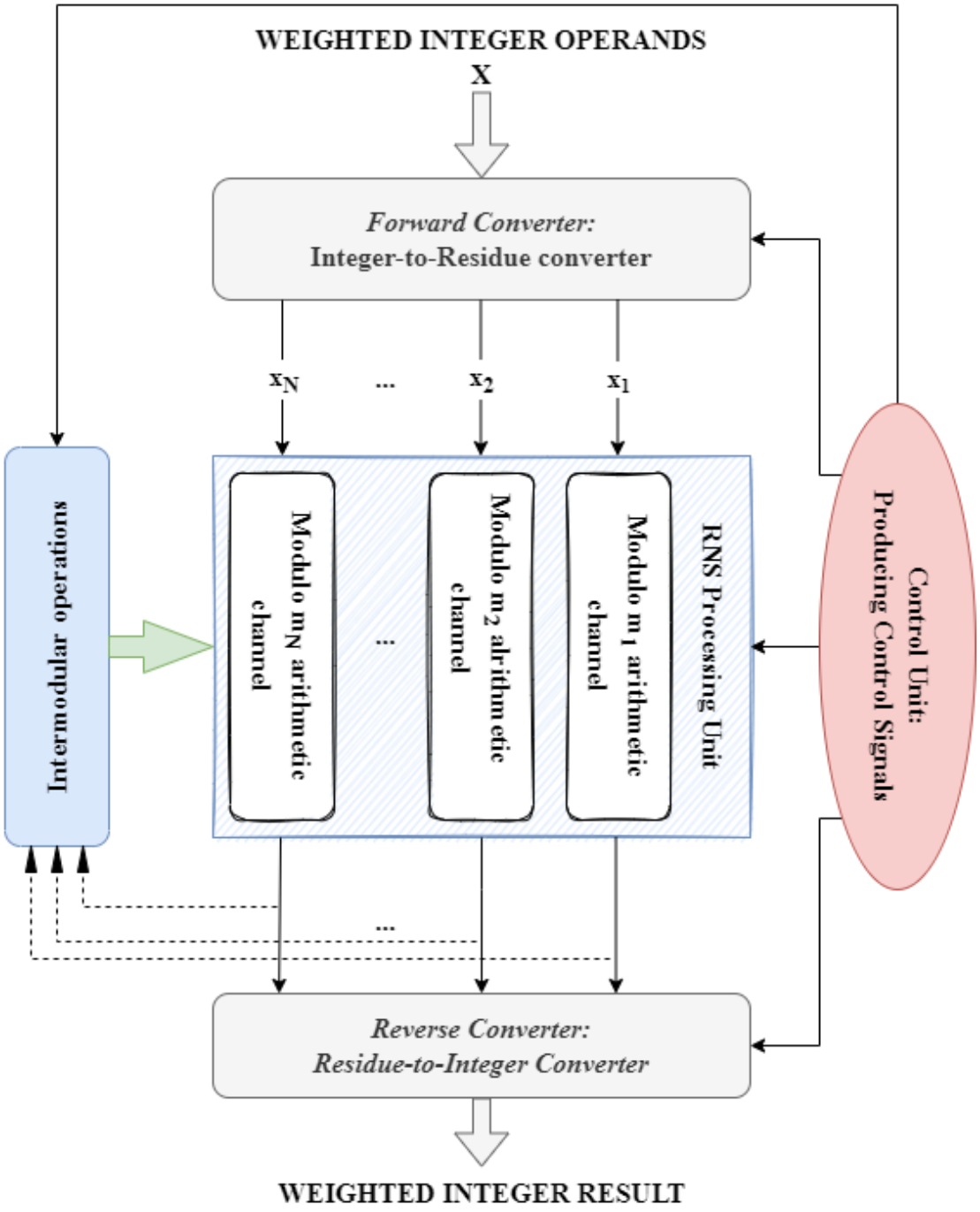

Figure 2.2: Overview of an RNS based application.

In this work the traditional 3-moduli set $[2^n - 1,\ 2^n,\ 2^n + 1]$ is used since it is the most common one in RNS datapaths [3], even if it is not the best one in terms

of dynamic range: this is the reason together with its reduced parallelism that has led to investigate on larger and different moduli set [1].

2.1 Conversion and Reconversion

The *Residue Numeral System* is based on the *Chinese Reminder Theorem*, CRT, a method whose theory ensures us the RNS application. Thanks to the CRT it is possible to say that if the moduli set has been chosen appropriately, then each number in the considered dynamic range will have a unique representation in RNS so that it can be converted and reconverted [10]. The CRT reconversion is also used as a method to implement the comparison by compare the numbers in a positional numeral system after the proper reconversion.

Indeed, the CRT methods states that considering a moduli set $[m_1, m_2, \ ... \ , m_N]$ of positive pairwise relatively prime integers, any number X in the M range can be written in RNS as reported in Equation (2.2):

$$X_{RNS} = [|X|_{m_1}, |X|_{m_2}, \ ..., \ |X|_{m_N}] = [x_1, x_2, \ ... \ , x_N] \tag{2.2}$$

where $|X|_{m_i}$ is the mathematical representation of the reminder of $\frac{X}{m_i}$.

Any RNS number can be reconverted thanks to the CRT as in Equation (2.3):

$$X = \left| \sum_{i=1}^{N} M_i \times \left| k_i \times x_i \right|_{m_i} \right|_M \tag{2.3}$$

where $M_i = \frac{M}{m_i}$ and $k_i = \left| M_i^{-1} \right|_{m_i}$ which represents the multiplicative inverse of $\left| M_i \right|_{m_i}$ so that $\left| M_i \times M_i^{-1} \right|_{m_i} = 1$ [4].

As it is possible to see, the CRT method requires a binary inner product operation and a large modulo M operation that makes the VLSI realization slow and complex. Therefore, other reconversion's methods may be used to improve VLSI reconversion's realization such as the *Mixed Radix Conversion*, MRC, algorithm.

The MRC method is sequential and cannot be parallelized, so that it is not suitable for high-speed design [11]. The reconversion takes place through Equation (2.4).

$$X = \sum_{i=1}^{n} v_i a_i \tag{2.4}$$

where $n > 1$, $v_i = \prod_{j=1}^{i-1} m_j$ for $2 \leq i \leq n$ considering $v_1 = 1$ while $a_i = |Y_i|_{m_i}$, by imposing $Y_1 = X$ and $Y_i = (Y_{i-1} - a_{i-1})|m_{i-1}^{-1}|_{m_i}$ [11].

The combination of CRT and MRC methods generates another reconversion technique [11] reported in Equation (2.5).

$$X = \sum_{j=1}^{n-2} \left[\alpha_{j+1} \prod_{i=1}^{j+1} m_i \right] + \alpha_1 m_1 + \alpha_0 \tag{2.5}$$

where $\gamma_i = M|M_i^{-1}|_{m_i}/m_1 m_i$, $\alpha_{j+1} = \left| \left\lfloor \sum_{i=1}^{j+2} \gamma_i x_i / \prod_{i=2}^{j+1} m_i \right\rfloor \right|_{m_{j+2}}$.

The *modified CRT* algorithm reduces the modulo base of m_i leading to an efficient and independent of the moduli sets' size converter design for small-sized moduli set [11]. It uses the Equation (2.6).

$$X = x_1 + m_1 \left| \sum_{i=1}^{n} w_i x_i' \right|_{m_n \ldots m_2} \tag{2.6}$$

where $n > 1$, $w_1 = \frac{(M_1|M_1^{-1}|_{m_1} - 1}{m_1}$, $w_i = \frac{M_i}{m_1}$, $x_1' = x_1$ and $x_i' = |M_i^{-1}|_{m_i} x_i$ considering $i \in [2, n]$.

In order to get a faster reconversion, other techniques and algorithms are investigated. One of the faster is reported in [12] which is based on the CRT scheme, while partial methods such as the *partial CRT* or *partial MRC* are preferred for the better performances [3].

2.2 Operations in RNS

The operations carried successfully with RNS are addition, subtraction and multiplication. Those operations are performed on each residue in a modular way. To better understand their formulation, considering two numbers X and Y and the moduli set $[m_1, m_2, \ \ldots\ , m_N]$ of positive pairwise relatively prime integers, the operations of addition, subtraction and multiplication on the two RNS values $X_{RNS} = [x_1, x_2, \ \ldots\ , x_N]$ and $Y_{RNS} = [y_1, y_2, \ \ldots\ , y_N]$ are reported respectively in Equations (2.7), (2.8) and (2.9).

$$X_{RNS} + Y_{RNS} = [|x_1 + y_1|_{m_1}, |x_2 + y_2|_{m_2}, \ \ldots,\ |x_N + y_N|_{m_N}] \tag{2.7}$$

$$X_{RNS} - Y_{RNS} = [|x_1 - y_1|_{m_1}, |x_2 - y_2|_{m_2}, \ \ldots,\ |x_N - y_N|_{m_N}] \tag{2.8}$$

$$X_{RNS} \times Y_{RNS} = [|x_1 \times y_1|_{m_1}, |x_2 \times y_2|_{m_2}, \ \ldots,\ |x_N \times y_N|_{m_N}] \tag{2.9}$$

An example is reported in Figure 2.3 in which two values of X and Y are converted with the moduli set [3,4,5] as in Equation (2.2), then the three operations

of addition, subtraction and multiplication are applied and finally those two values are reconverted as in Equation (2.3).

As it is possible to see, the RNS operations' results are always positive reminders that do not exceed the associated modulus value.

Those consideration help us to understand the algorithms used in open source software solutions: if the result of a modular addition or multiplication is higher than the modulus, then the reminder associated to the modulus is recomputed; instead, if the result of a modular subtraction is negative, then the reminder associated to the modulus is recomputed by adding the modulus value itself.

The algorithms of addition, subtraction and multiplication are reported respectively in Algorithm 1, 2 and 3

Algorithm 1 Addition in RNS

```
for i=1:1:length(moduli_vector) do
    add(i)=a(i)+b(i);
    if add(i)>moduli_vector(i) then
        add(i)%=moduli_vector(i);
    end if
end for
return add
```

Algorithm 2 Subtraction in RNS

```
for i=1:1:length(moduli_vector) do
    sub(i)=a(i)-b(i);
    if sub(i)<0 then
        sub(i)+=moduli_vector(i);
    end if
end for
return sub
```

Algorithm 3 Multiplication in RNS

```
for i=1:1:length(moduli_vector) do
    mult(i)=a(i)*b(i);
    if mult(i)>moduli_vector(i) then
        mult(i)%=moduli_vector(i);
    end if
end for
return mult
```

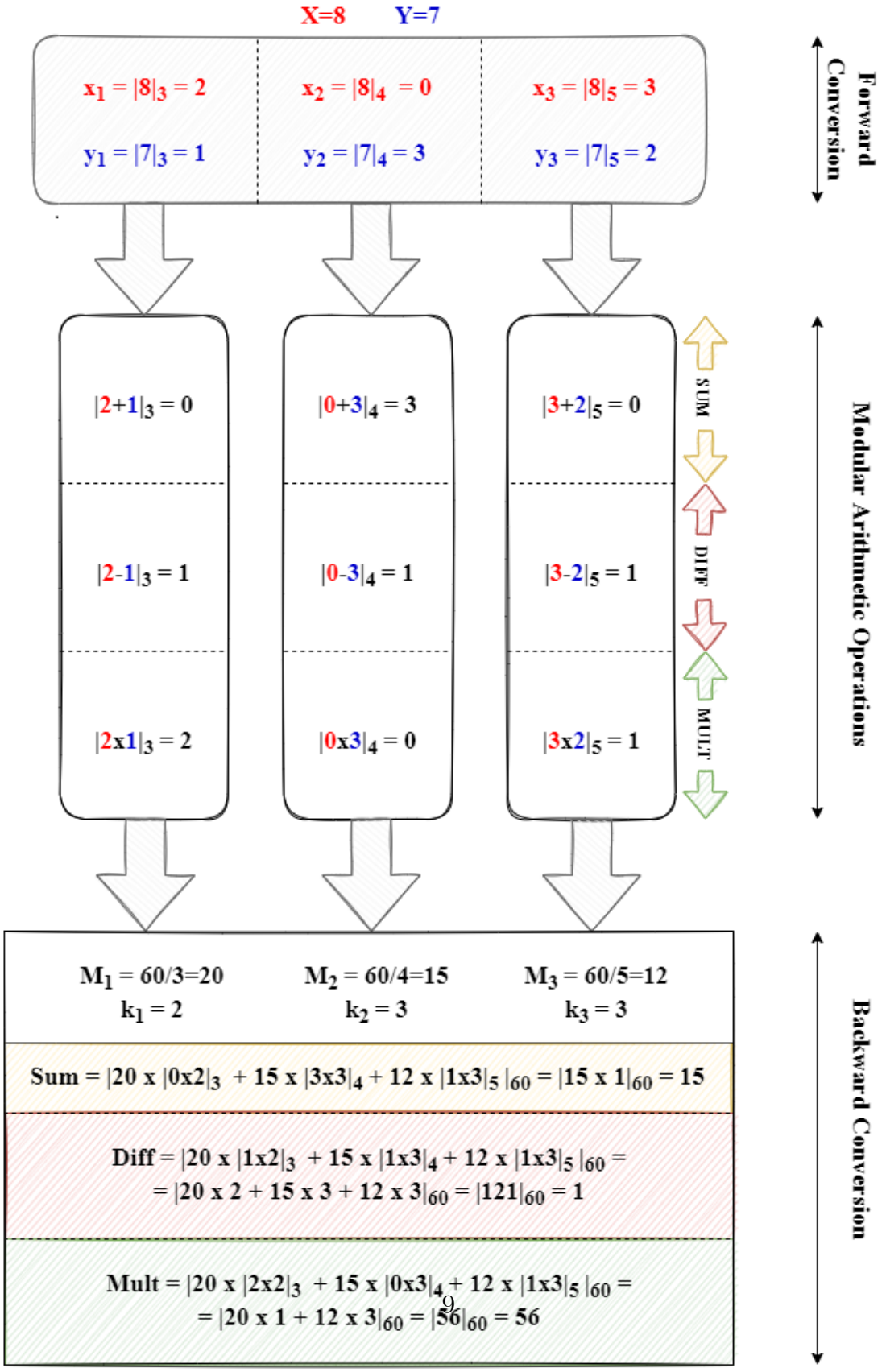

Figure 2.3: Operations of addition subtraction and multiplication considering the moduli set [3,4,5] and the input values $X = 8$ and $Y = 7$.

Chapter 3

Comparison with Residue Number System

3.1 Current comparison methods

Nowadays to implement the comparison in *Residue Number System* different solutions are proposed. The most used and exploited methods are the following [3].

1. *Conversion-Based Schemes:* This first method is the easiest one that solves the problem at the source comparing the two values in a positional system after their reconversion. However, the drawback is choosing the best reconversion system to get the best performances.

2. *Subtraction-Based Methods:* This method is about checking the sign of the difference between the two operands to understand their relation. The sign check operation is not easy to implement in RNS, so the RNS subtraction result can be both reconverted in binary or its sign is recognized using the proper techniques.

3. *Parity Checking Schemes:* Studying the comparison results, it is possible to see that if all the modulus are odd the comparison is a result of the operands' parity and their difference. This method avoids the conversion but needs some specifications in the moduli set that are not so used since the modulus $m_i = 2^n$ is one of the most efficient arithmetic channel.

4. *Diagonal Mapping:* Since the numbers in the dynamic range of a k moduli RNS can be arranged in a $k-$dimension space, it is possible to group all the RNS numbers in *diagonal lines* if $k = 2$, *diagonal surfaces* if $k = 3$ and so on. The diagonals are associated to integers so that it is possible to compare the

numbers associated to each diagonal to evaluate the comparison result, if the inputs are on the same diagonal the comparison is performed between the two residues.

The method exploited in this work is similar to the Diagonal mapping one: as in this latter method, every number is associated to indexes that are compared. The diagonal evaluation uses the Equation (3.1) [13].

$$D(x) = \left| \sum_{i=1}^{n} k_i x_i \right|_{SQ} \tag{3.1}$$

where $SQ = \sum_{i=1}^{n} M_i$, $k_i = \left| -\frac{1}{m_i} \right|_{SQ}$ and x_i represents the residue associated to the modulus m_i [13].

The diagonal mapping has been investigated to speed up the most difficult and costly operation which is the modulo-SQ addition. In addition, there are also different methods that modifies the diagonal function evaluation that represents the RNS numbers [14].

The parity checking method examines the parities of both the operands and their difference and then effectuates the final decision using a simple algorithm. However, the parity detection method is quite complex to implement and requires that all the moduli are odd.

The conversion-based scheme is widespread and encourages to research improved conversion methods: in [15] the comparison is achieved after converting the RNS values in MRC digits, [16] and [3] split the range in subranges via partitioning functions achieved thanks to a *partial CRT* reconversion, [11] uses Equation (2.5) to reconvert the inputs and then compare them in binary.

The conversion method is also needed for the subtraction-based schemes since after the RNS-subtraction the result can be fully [17] or partially [18] reconverted to check its sign ([19], [20]).

3.2 PANC method

The method exploited in this work is the *Positional Attribute of Non-positional Code*, also known as PANC. The numbers represented in RNS cannot be directly compared due to the non-positional representation, however the PANC method aims to carry out the comparison without converting the numbers and therefore using the Residue Number System representation.

The PANC method is not based on the comparison between the two RNS values in question, but it associates them to a specific range in order to compare those two latter numbers.

Even if this method does not resort to any conversion of the numbers, it features some disadvantages such as the computational complexity of forming PANC and the difficulty of directly use the method during the implementation of data comparing operations.

The aim of this work is to create architectures that can perform the mathematical method which uses the PANC, as done in [2], for quickly and accurately comparing two numbers in RNS.

The procedure to perform the comparison with the mathematical method which uses the PANC is reported in this section to better understand the architectural choices.

Before the description it is necessary to define some operands:

1. m_n is the highest modulus considered in the moduli set.
2. N_{m_i} is the product between all the moduli except the highest one.
3. $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$ represent the *zeroing constants* of the two operands A_{RNS} and B_{RNS}.
4. The vectors $K_{N_{m_n}}^{(n_A)}$ and $K_{N_{m_n}}^{(n_A)}$ contain all the difference values respectively of $Z_i^{A_{RNS}}$ and $Z_i^{B_{RNS}}$.
5. a_n and b_n are the residues associated to the last modulus m_i.
6. n_A and n_B represent the range associated respectively A_{RNS} and B_{RNS} and they are the *positional attribute of the non-positional code.*

The inputs A_{RNS} and B_{RNS} should be in the range $(0, M]$ with $M = \prod_{i=1}^{n} m_i$, the aim of the exposed mathematical method is identify and compare the two values n_A and n_B by dividing the whole range M in N_{m_i} intervals of m_n magnitude as represented in Figure (3.1).

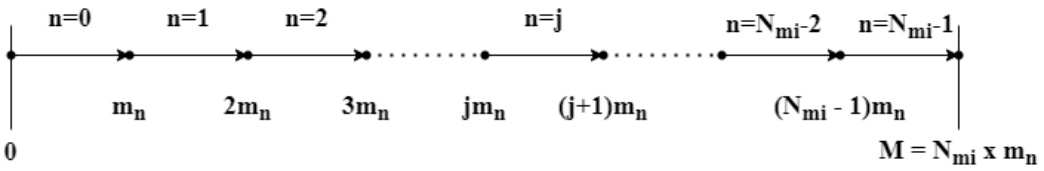

Figure 3.1: Whole range M divided in N_{m_i} intervals of m_n magnitude.

The accuracy by which the comparison is evaluated depends on the size of the N_{m_i} intervals as $W_{m_i} = \frac{1}{m_i}$. Since m_i is the largest modulus, the accuracy W_{m_i} is the lowest possible. However the number of intervals N_{m_i} is also the lowest and

this is a huge benefit given that, as will be described in the algorithm, on this value depends the number of operations used to obtain the vectors $K_{N_{m_i}}^{(n_A)}$ and $K_{N_{m_i}}^{(n_A)}$ whose magnitude is, as a matter of fact, N_{m_i}.

In order to obtain an higher accuracy, it is checked the relationship between the residues associated to the highest modulus: a_n and b_n. If the two numbers under observation A_{RNS} and B_{RNS} fall in the same interval $j = n_A = n_B$, indeed, then the values a_n and b_n are going to establish exactly the magnitude of each operands leading to the result of the comparison.

The algorithm can be described in six steps, taking into account that both the numbers under observation A and B are already in their RNS form, respectively A_{RNS} and B_{RNS}:

1. The evaluation of N_{m_i};
2. The formation of zeroing constants $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$;
3. The evaluation of A_{m_n} and B_{m_n}, computed as in Equation (3.3). Those latter values are multiples of modulo m_n of RNS;
4. The definition of each single component $Z_i^{A_{RNS}}$ and $Z_i^{B_{RNS}}$ in order to create the vectors $K_{N_{m_n}}^{(n_A)}$ and $K_{N_{m_n}}^{(n_A)}$, computed as in (3.4);
5. The formation of the quantitative values n_A and n_B for which $Z_{n_A}^{(A)} = 0$ and $Z_{n_B}^{(B)} = 0$;
6. The comparison's implementation for A_{RNS} and B_{RNS} as reported in (3.5).

As already said, the N_{m_i} is the product of all the moduli except the last one. Since this value is not generally used when operating in RNS, we must evaluate this as in Equation (3.2):

$$N_{m_i} = \prod_{i=1}^{n-1} m_i \tag{3.2}$$

where n is the maximum modulus' index.

Each zeroing constant represents the RNS value of the number in the range $[0, m_n)$ that has the same input's m_n-th residue [21]. To better understand its significance, knowing that any value X in RNS can be written as

$$X_{RNS} = (x_1||x_2||...||x_{i-1}||x_i||x_{i+1}||...||x_n)$$

its related zeroing constant $KN_{m_n}^{(X_{RNS})}$, which is associated to the m_n modulus, is going to have all different residues with respect of the ones of the input X_{RNS} except the n-th one:

$$KN_{m_n}^{(X_{RNS})} = (x_1'||x_2'||...||x_{i-1}'||x_i'||x_{i+1}'||...||x_n)$$

If a number is multiple of a modulus, then its associated residue is going to be 0. This latter consideration let us understand that A_{m_n} and B_{m_n}, defined as *multiples of the modulus* m_n, can be extrapolated as the difference between the inputs and their proper zeroing constant as reported in Equation (3.3).

$$\begin{gathered} A_{m_i} = A_{RNS} - KN_{m_n}^{(A)} = \\ = (a_1||a_2||...||a_{i-1}||a_i||a_{i+1}||...||a_n) - (a'_1||a'_2||...||a'_{i-1}||a'_i||a'_{i+1}||...||a_n) = \\ = (a_1^{(1)}||a_2^{(1)}||...||a_{i-1}^{(1)}||a_i^{(1)}||a_{i+1}^{(1)}||...||0) \\ B_{m_i} = B_{RNS} - KN_{m_n}^{(B)} = \\ = (b_1||b_2||...||b_{i-1}||b_i||b_{i+1}||...||b_n) - (b'_1||b'_2||...||b'_{i-1}||b'_i||b'_{i+1}||...||b_n) = \\ = (b_1^{(1)}||b_2^{(1)}||...||b_{i-1}^{(1)}||b_i^{(1)}||b_{i+1}^{(1)}||...||0) \end{gathered} \tag{3.3}$$

During this third step, both the inputs A_{RNS} and B_{RNS} have been reported in one of the N_{m_i} intervals, even if to get their exact position it is necessary to execute the next operation: the computation of each single $Z_i^{A_{RNS}}$ and $Z_i^{B_{RNS}}$ to form the vectors $K_{N_{m_n}}^{(n_A)}$ and $K_{N_{m_n}}^{(n_A)}$.

Both $Z_i^{A_{RNS}}$ and $Z_i^{B_{RNS}}$ are evaluated by subtracting respectively from A_{m_i} and B_{m_i} all the multiples of m_n in the whole range $[0, M)$, evaluated as the multiplication between m_n and all the numbers in the interval $[0, N_{m_i})$. Their computation is reported in Equation (3.4).

$$\begin{cases} A_{m_i} - 0 \cdot m_n = Z_0^{(A_{RNS})}, \\ A_{m_i} - 1 \cdot m_n = Z_1^{(A_{RNS})}, \\ A_{m_i} - 2 \cdot m_n = Z_2^{(A_{RNS})}, \\ ... \\ A_{m_i} - (N-2) \cdot m_n = Z_{N-2}^{(A_{RNS})}, \\ A_{m_i} - (N-1) \cdot m_n = Z_{N-1}^{(A_{RNS})}; \end{cases} \quad \begin{cases} B_{m_i} - 0 \cdot m_n = Z_0^{(B_{RNS})}, \\ B_{m_i} - 1 \cdot m_n = Z_1^{(B_{RNS})}, \\ B_{m_i} - 2 \cdot m_n = Z_2^{(B_{RNS})}, \\ ... \\ B_{m_i} - (N-2) \cdot m_n = Z_{N-2}^{(B_{RNS})}, \\ B_{m_i} - (N-1) \cdot m_n = Z_{N-1}^{(B_{RNS})}; \end{cases} \tag{3.4}$$

The result of the Equation (3.4) is a binary sequence of ones and only one zero. This only one zero is representing the exact interval position of the considered input:

$$A_{m_i} - n_A \cdot m_n = 0 \rightarrow A_{m_i} = n_A \cdot m_n$$

$$B_{m_i} - n_B \cdot m_n = 0 \rightarrow B_{m_i} = n_B \cdot m_n$$

Now that the exact interval positions of both the intervals are known, the algorithm can be finally implemented as in Equation (3.5):

$$\begin{aligned} &A_{RNS} = B_{RNS}, if \quad [(n_A = n_B)\ and\ (a_n = b_n)] \\ &A_{RNS} > B_{RNS}, if \quad (n_A > n_B)\ or\ [(n_A = n_B)\ and\ (a_n > b_n)] \\ &A_{RNS} < B_{RNS}, if \quad (n_A < n_B)\ or\ [(n_A = n_B)\ and\ (a_n < b_n)] \end{aligned} \tag{3.5}$$

In Equation (3.5), according to the accuracy's considerations, also the residues associated to the last modulus a_n and b_n are taken into account.

Chapter 4

Design to implement the PANC method comparison

In this chapter are reported all the architectural choices made in order to create a valuable PANC comparator. Once the proper moduli set has been chosen, the best architectural designs in terms of delay, area and power consumption have been selected. The next three models are described in the paragraphs 4.1, 4.2 and 4.3, while the results associated to each design are discussed in the next chapter, section 5.1.

The chosen moduli set is $[2^n - 1,\ 2^n,\ 2^n + 1]$, since it is the most popular one in Residue Number System's datapaths [22]. The n values most commonly used are $n = 5$ and $n = 8$: all the considered designs operate with both the two moduli set [31, 32, 33] and [255, 256, 257]. Another moduli set for which tests are done is the one with $n = 3$, [7, 8, 9], since its simplicity makes it easier to build and useful to find out the system's problems.

With the moduli set proposed, the comparator has to work with values represented on $3n + 1$ bits $((n) + (n) + (n + 1))$ and for each modulus m_i considered the maximum number represented is $m_i - 1$ [17]. Therefore, for $n = 5$ the numbers are represented with 16 bits and the maximum RNS number representable is [30, 31, 32] equal to 32735 in decimal, while with $n = 8$, 25 bits are needed to operate on all the numbers within 0 and 16776959, whose value in RNS is [254, 255, 256].

To be aligned with the others architectures found, the number's representation for each design is composed by the first $n + 1$ most significant bits associated to the modulus $2^n + 1$, then the last $2n$ bits are related respectively to the moduli 2^n and $2^n - 1$.

The first proposed solution in the paragraph 4.1 is the *Golden Model* in which we assume that all the operations are performed in parallel with the highest resources

possible. This architectural design is expecting to be the faster but also the wider.

The second architectural design in paragraph 4.2 is the opposite of the Golden Model because the resources used are reduced to the minimum and shared: indeed, it is the *Resource sharing* one. Since this latter design is the reverse of the first one, we anticipate the smaller but slower performances because all the operations are done sequentially. Furthermore, as it is possible to see from the algorithm's description in 3.2, all the steps are sequential except the comparison between the residues associated to the larger modulus m_n. Besides, it is possible to divide all the sequential steps depending on the values of m_n and N_{m_i}: the zeroing constants evaluation rely on m_n, while all the next operations, the generation of the vectors $K_{N_{m_n}}^{(n_A)}$ and $K_{N_{m_n}}^{(n_A)}$ and the indexes identification n_A and n_B, depend on N_{m_i}. Only the determination of $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$ is not depending on any of those values. To clearly illustrate the values of m_n and N_{m_i} for the different n considered, it is possible to refer to Table 4.1.

n	N_{m_i}	m_n
3	56	9
5	992	33
8	65280	257

Table 4.1

From those consideration, the Golden Model is expected to take the same clock cycles to perform the algorithm regardless of the n considered, however to get the performance's parallelization, the number of architectural elements is depending on both the values m_n and N_{m_n}, and, therefore, on n.

Conversely, the resource sharing model is expected to be formed by the same number of logical elements independently on n and to take about N_{m_i} clock cycles, since the zeroing constant evaluation is performed thanks to a *decoder*.

Both those two solutions are not ideal: the first one will occupy a lot of area, the second one will be too slow. To reach a trade off between those two solutions it has been applied the *unfolding* technique to the resource sharing design as reported in the section 4.3. Considering the N_{m_i}'s magnitude, only the algorithmic steps related to N_{m_i} have been unfolded.

In order to evaluate the best unfolding level, in Figure 4.1 is shown a graph in which are reported the clock cycles taken for each unfolding level 2^k with increasing k. The clock cycles are computed as $2 + \frac{N_{m_i}}{2^k}$ since we surely need 2 clock cycles to generate the zeroing constants $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$ and the multiples A_{m_n} and B_{m_n}, but we are replicating each unit that accomplishes the algorithm' steps depending on N_{m_n} as 2^k.

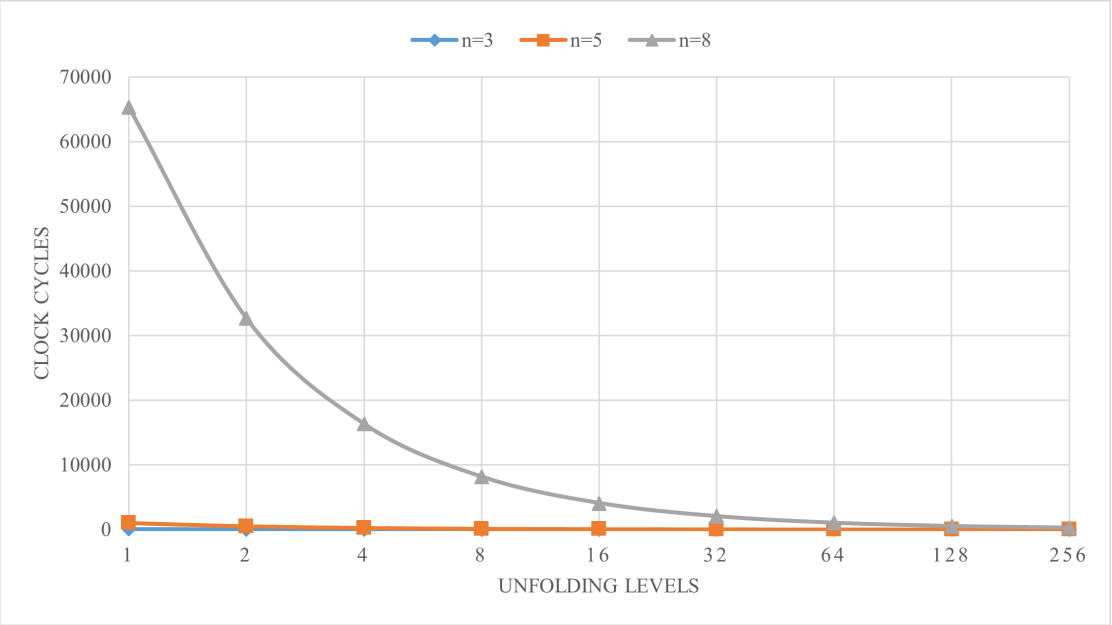

Figure 4.1: Graphical representation of the decreasing clock cycles for increasing unfolding level computed as 2^k.

The most interesting cases are the ones with $k = 2,3,4$: considering $k = 2$ the clock cycles are reduced by almost 4 times, but in the $n = 8$ case the computation should still need more than 10^4 cycles (16322 more specifically), also the results for $n = 5$ are not optimal, while for $n = 3$ the reduction is significant; this is why we moved to the next unfolding level $k = 3$ for both $n = 5$ and $n = 8$, we have not considered the case for $n = 3$ since we calculated that clock cycles' gain would have been lower with respect of the structure's complexity. Those considerations also apply for $k = 16$ in the $n = 5$ case, which is where we stopped also in case $n = 8$ to be in line with the results.

In Figure 4.2, 4.3 and 4.4 are reported in detail the decreasing clock cycles for increasing unfolding level 2^k respectively for $n = 3$, $n = 5$ and $n = 8$ in order to better understand their behavior.

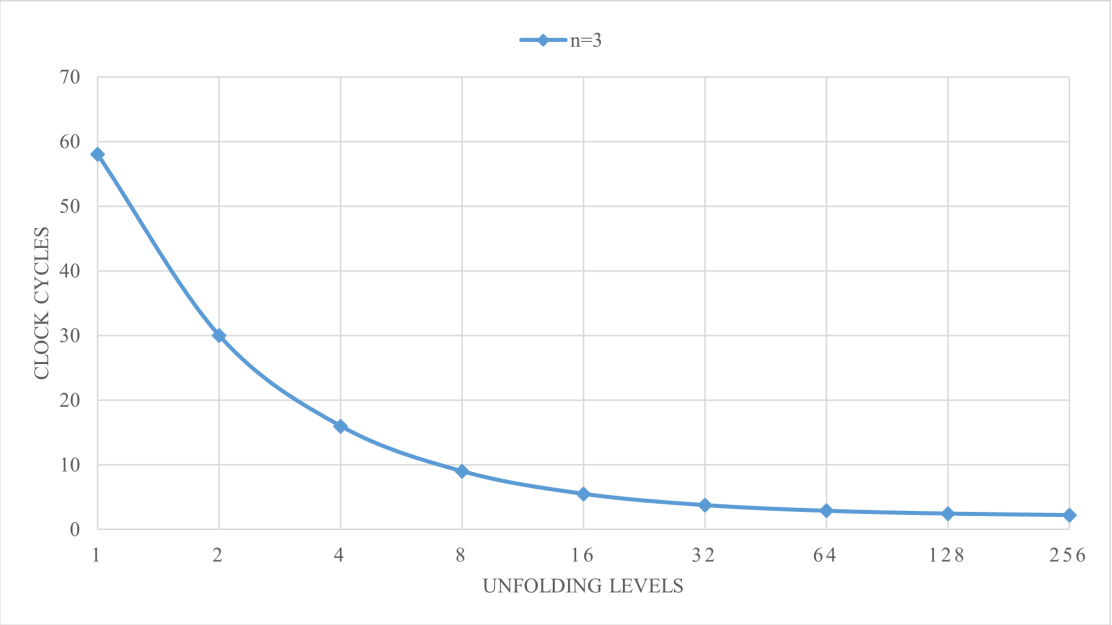

Figure 4.2: Graphical representation of the decreasing clock cycles for increasing unfolding level computed as 2^k considering $n = 3$.

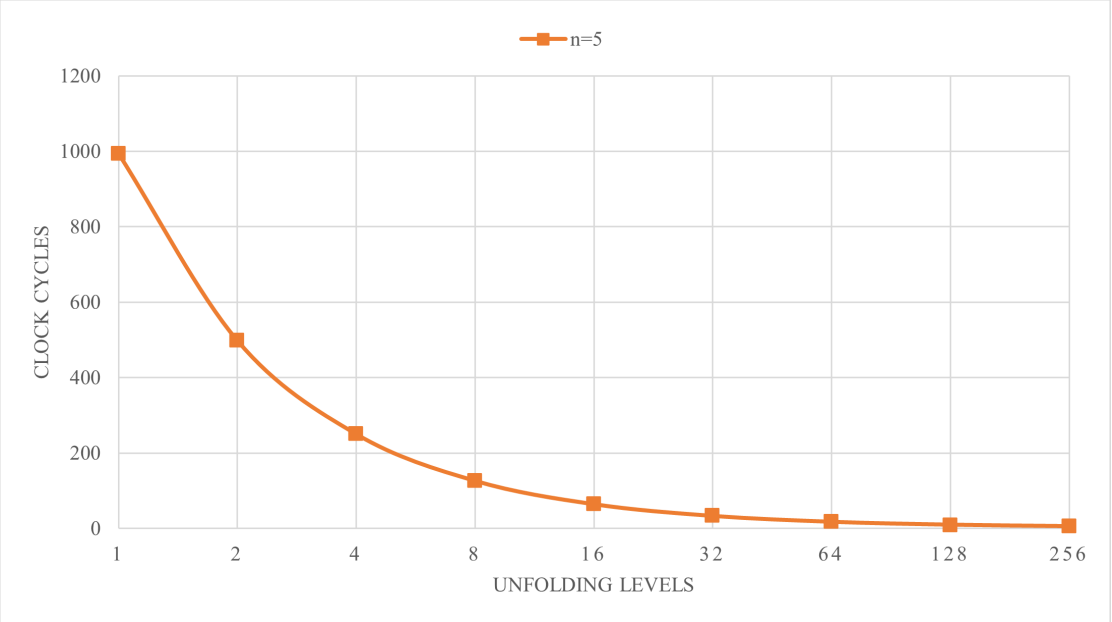

Figure 4.3: Graphical representation of the decreasing clock cycles for increasing unfolding level computed as 2^k considering $n = 5$.

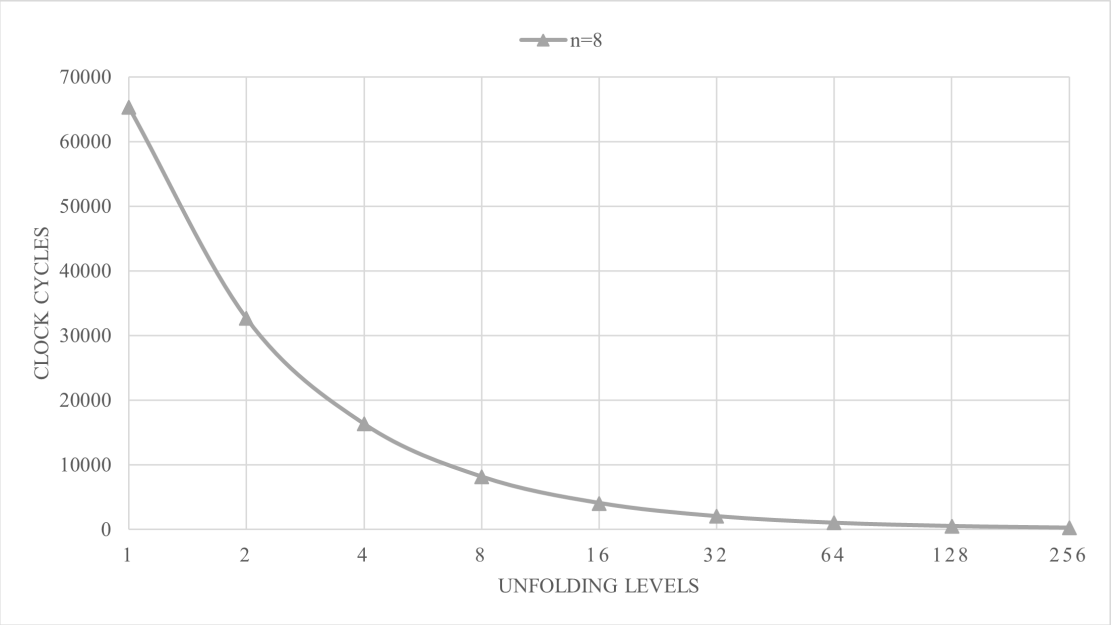

Figure 4.4: Graphical representation of the decreasing clock cycles for increasing unfolding level computed as 2^k considering $n = 8$.

In section 4.3 those designs are reported, analyzed and commented and those consideration are better exploited.

4.1 Golden Model

The first structure analyzed is the *Golden Model*. As already anticipated, this architecture is the fastest and largest, but it is also the simplest that can be imagined and the starting point for the definition of other designs.

At first it is described in details the *Datapath* associated to this design, then it is explained how its *Control Unit* works.

4.1.1 Datapath

The Golden Model's design is a direct implementation of the algorithm, this is why in its description it is divided in three different parts that will be described in detail in this section.

The *first part* includes the definition of the *zeroing constants* and evaluates the result of the *comparison between* a_n *and* b_n. Both those two instructions can be performed by simply using the inputs' $n + 1$ most significant bits since the latter instruction can be easily performed by a simpler $n + 1$ **bits comparator**, while

the zeroing constants $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$ are evaluated using **two decoders** with $n+1$ input bits and $3n+1$ output bits. Each decoder associates the m_n's residue to its number in RNS as reported in the truth table in Table 4.2 referred to the simplest case of $n = 3$ where each output has been lined with its associated modulus.

INPUT	m_9	m_8	m_7
0000	0000	000	000
0001	0001	001	001
0010	0010	010	010
0011	0011	011	011
0100	0100	100	100
0101	0101	101	101
0110	0110	110	110
0111	0111	111	000
1000	1000	000	001

Table 4.2

To clarify the functioning of the first part in Figure 4.5 is reported its RTL scheme for $n = 3$. The decoder's *EN* input signal is generated by the control unit and used to make the decoder working only when necessary, so during this first clock cycle.

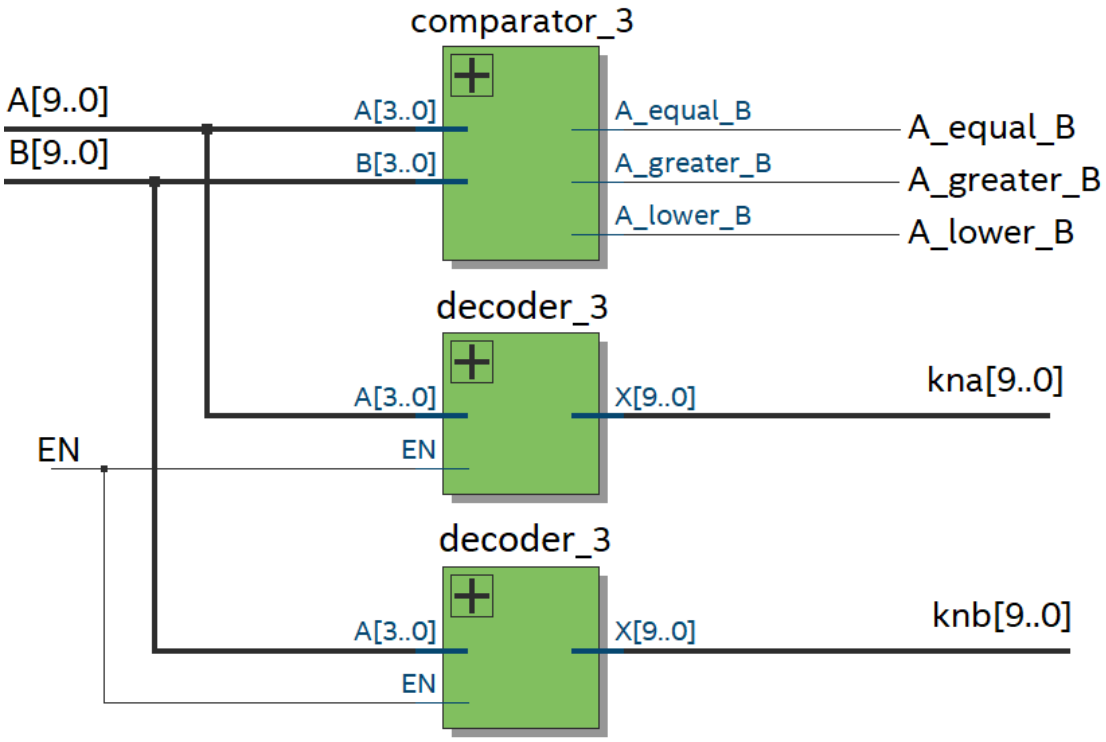

Figure 4.5: The first part's RTL structure for $n = 3$ in which the inputs A and B's $n + 1$ bits are used to both generate the proper zeroing constant with the decoders and the result of the comparison between its m_n-th residue.

This first part's outputs are $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$, the two zeroing constants on $3n + 1$ bits, and the outcome of the comparison represented as 3 single bit signals representing the possible outcome of the operation: $a_n > b_n$, $a_n < b_n$ and $a_n = b_n$. All the results are written in their proper registers, ready to be used in the following steps.

The *second part* gives as results the *inputs'* m_n *multiples* A_{m_n} and B_{m_n} and *each single component* $Z_i^{A_{RNS}}$ and $Z_i^{B_{RNS}}$, both computed as in Equation (3.3) and (3.4) respectively. Those two RNS operations are both modular and are respectively a **subtraction** and a **multiplication for a constant value**. It is, indeed, performed the subtraction between the inputs and their proper zeroing constants, computed in the previous part and stored in their proper registers, and the multiplication of all the numbers in the range $[0, N_{m_i})$ by the constant m_n value in RNS equal to $[2,\ 1,\ 0]$.

The subtraction cannot give any negative residue result since the negative numbers in RNS are identified considering the dynamic range [5]. In order to avoid the negative numbers, the subtractor has been built following the Algorithm 2 as

reported in all the open-source software solutions.

Each modular subtractor is formed by 4 elements:

1. **Sign extension** by which is added '0' at the start of the residue knowing that all the values are for sure positive;

2. **Subtractor** to perform the subtraction between signed numbers;

3. **Sign detection** to detect the most significant bit and add the modulus value in case the MSB is equal to '1', so the number is negative, otherwise the number is given to the last element without any change;

4. To keep the number on its proper bitwidth, the last component eliminates the most significant bit, that is for sure '0', and it is named as **redundant**.

With the chosen moduli set it is necessary to add only one bit to the residues associated to both the highest and lowest moduli. The 2^n modulus is written on $n+1$ bits, so its signed subtraction must be performed on $n+2$ bits and this is the reason why the *sign extention* component adds two zeroes "00" and the last component truncates the two most significant bits.

In Figure 4.6 is reported the modular subtractor structure for $n = 3$ where each component is marked with respect of the associated modulus ([7, 8, 9]).

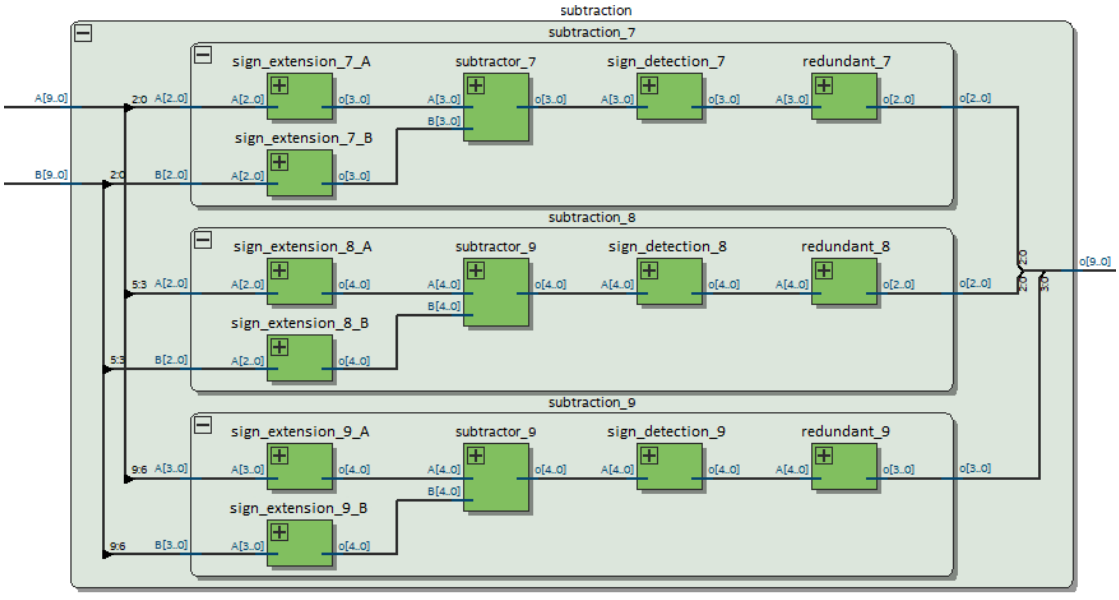

Figure 4.6: Modular subtractor in case $n = 3$: each component is considered with respect of the associated modulus ([7, 8, 9]).

As it is possible to see from the algorithm, each number in the interval $[0, N_{m_i})$ must be multiplied by the constant m_n value in RNS form. Also, all those inputs must be available all at the same moment, that is why they are all stored in their proper $3n+1$ bits registers. Considering our moduli set, this RNS constant is equal to $[2,\ 1,\ 0]$. It is not worth to use a proper RNS multiplier, but, looking at the numbers, it is possible to set all the $3n+1$ bits as follows:

- The most significant $n+1$ bits are forced to 0;
- The following n bits are equal to the input ones;
- The least n bits are multiplied by 2. Observing the software' solutions Open-Source, if after the multiplication the resultant residue is higher than the modulus' value, the result's residue must be ri-evaluated, as reported in the Algorithm 3. Although, it is possible to see that each multiplication result is its *circular shifted* value, making possible to only use a **circular shift** to perform the multiplication.

This is the reason why the multiplier has been named as Fake-multiplier. Its structure considering the design with $n = 3$ is reported in Figure 4.7.

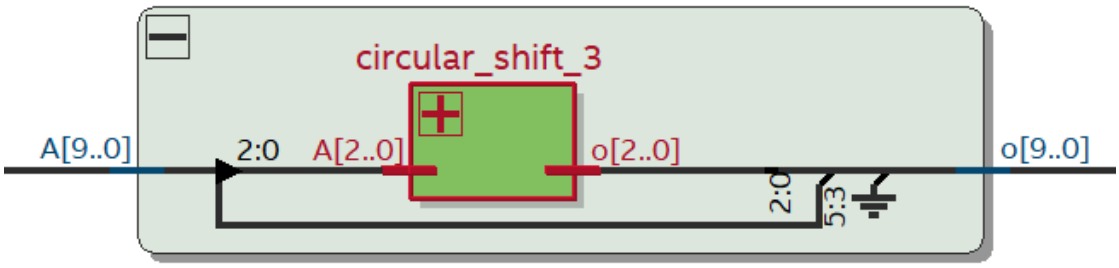

Figure 4.7: Multiplier used in the designs for $n = 3$: the first $n+1$ bits are forced to 0, the following n bits are given as output while the least n bits are modified by the circular shift.

To accomplish this second algorithmic part are necessary 2 subtractors, N_{m_i} circular shifts and $N_{m_i} + 2$ registers of $3n + 1$-bits to memorize the results and parallelize the operation. Also it is necessary to remember the needs of N_{m_i} registers to get the multiplication inputs.

The *third part* completes the comparison. The operations managed in this part are the *evaluation of the vectors* $K_{N_{mn}}^{(n_A)}$ *and* $K_{N_{mn}}^{(n_A)}$, computed as in (3.4), *the formation of the quantitative values* n_A *and* n_B for which $Z_{n_A}^{(A)} = 0$ and $Z_{n_B}^{(B)} = 0$ and finally the comparison between A_{RNS} and B_{RNS} as reported in (3.5). The

algorithmic formation of the vectors $K_{N_{m_n}}^{(n_A)}$ and $K_{N_{m_n}}^{(n_A)}$ is implemented by subtracting from the m_n's multiples, A_{m_n} and B_{m_n}, the results of the previous multiplication and by inserting a $'1'$ whenever the subtraction gives a result different from 0, otherwise is put a $'0'$. Those two vectors can be created by using only subtractors and an **equality comparator**, but there has been made a slightly modification by inserting a '1' if the subtraction result gives a 0 and '0' otherwise. In this way has been exploited an **encoder** to achieve the indexes n_A and n_B's detection. In Table 4.3, is reported the encoder's truth table considering $n = 3$ (even if is not reported in its enterety).

INPUT	**OUT**	**Decimal value**
00000000000000000000000000000000000000000000000000000000	000000	0
00000000000000000000000000000000000000000000000000000001	000001	1
00000000000000000000000000000000000000000000000000000010	000010	2
00000000000000000000000000000000000000000000000000000100	000011	3
00000000000000000000000000000000000000000000000000001000	000100	4
00000000000000000000000000000000000000000000000000010000	000101	5
00000000000000000000000000000000000000000000000000100000	000110	6
00000000000000000000000000000000000000000000000001000000	000111	7
...	...	...
00000000000000000000100000000000000000000000000000000000	100011	35
00000000000000000001000000000000000000000000000000000000	100100	36
00000000000000000010000000000000000000000000000000000000	100101	37
...	...	...
00100000000000000000000000000000000000000000000000000000	110101	53
01000000000000000000000000000000000000000000000000000000	110110	54
10000000000000000000000000000000000000000000000000000000	110111	55

Table 4.3

Now that both the indexes are ready to be compared, a **comparator** on $log_2(N_{m_i})$ bits decides which is the relationship between the indexes n_A and n_B.

This third part uses $2 \cdot N_{m_i}$ subtractors and equality comparator, 2 encoders $N_{m_i} - to - log_2(N_{m_i})$ and one comparator on $log_2(N_{m_i})$ bits. At the end of this path the results, the 3 bits that identifies respectively $n_A > n_B$, $n_A < n_B$ and $n_A = n_B$, are stored in 3 final registers of 1 bit.

Since the last step is the selection of the result, it is used a multiplexer $2 - to - 1$ which takes as inputs the 3 bits associated to the last residue and indexes comparison, but the selection signal is due to the control unit and depends on the $n_A = n_B$ result. This operation needs to delay the output of one clock cycle thanks to 3 register of 1 bit in which the 3 output signals that decides if $A > B$, $A < B$ or $A = B$ are stored.

In Table 4.4 are reported all the units used in this datapath, considering the modular operators.

Algorithm part	**Number of elements**	**Element**	**Number of Bits**
First part	2	Decoder	$n+1\ -to-\ 3n+1$
	1	Comparator	$n+1$
Second part	2	Subtractor	$3n+1$
	N_{m_i}	Fake-multiplier	$3n+1$
Third part	$2 \cdot N_{m_i}$	Subtractor	$3n+1$
	$2 \cdot N_{m_i}$	Zero check	$3n+1$
	2	Encoder	$N_{m_i}\ -to-\ log_2(N_{m_i})$
	1	Comparator	$log_2(N_{m_i})$
	1	Multiplexer $2\ -to-\ 1$	3
Total registers	$4+2 \cdot N_{m_i}$	Register	$3n+1$
	9	Register	1

Table 4.4

Since the components are too wide, it was not possible to generate the Golden model with $n = 8$.

4.1.2 Control Unit

The *control unit* is formed by 8 states:

1. **RESET** if the general *reset* is asserted;

2. **IDLE** when the inputs are awaited;

3. **OP_1** to enable the registers and the operations related to the *first part* (enabling the two decoders and the $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$ registers and the output of the comparison between a_n and b_n ones). The system gets in this state if all the inputs are ready to be processed;

4. **OP_2** to enable the registers related to the *second part* (the A_{m_n} and B_{m_n} ones and the multiplication results). The system gets in this state if nothing in the previous step has changed;

5. **OP_3** to enable the registers related to the *third part* (the registers of the comparison between the indexes n_A and n_B). The system gets in this state if nothing in the previous step has changed;

6. **ab** to correctly select the outputs related to the first comparison. The system gets in this state if the last comparison is over and if the signal related to $n_A = n_B$ is asserted;

7. **n** to correctly select the outputs related to the last comparison. The system gets in this state if the last comparison is over and if the signal related to $n_A = n_B$ is negated;

8. **Result** the output is available and are enabled the 3 registers related to the result (the ones associated to the relationship between the inputs). The system gets to this state when one of the last output signals is asserted and thanks to a proper signal announces the end of operation.

The control unit scheme is reported in Figure 4.8.

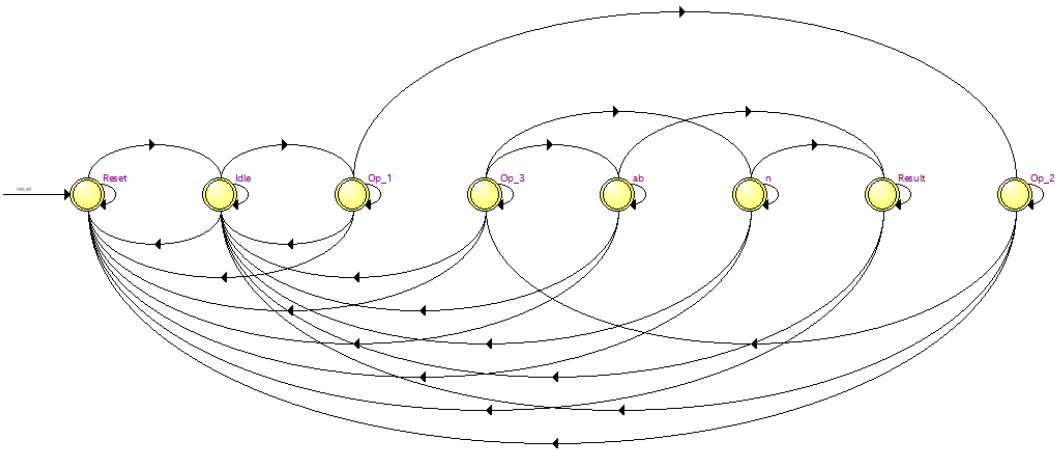

Figure 4.8: Golden Model's Finite State Machine.

In Table 4.5 are reported both the signals in input and output and is described their function and their computation.

Signal	**Input/Output**	**Significance**
rst	Input	The general **Reset** is asserted: the system is resetted
ready	Input	The inputs signal are ready to be analyzed.
nanb_ready	Input	When the both indexes have been analyzed and their relation has been evaluated, this signal is asserted
done	Input	The operation is ended: the result has been computed
sel	Input	This signal is asserted only if the two indexes n_A and n_B are equal and allows to step in one of the states **ab** or **n**.
enable_1	Output	The registers related to the first part are enabled.
enable_2	Output	The registers related to the second part are enabled.
enable_3	Output	The registers related to the third part are enabled.
sel_dp	Output	It is the multiplexer' selection signal deciding if the output depends on the indexes' comparison or the last residue.
enable_last	Output	It enables the last registers associated to the result and it is asserted only in the **Result**' state.

Table 4.5

4.2 Resource sharing

The *Resource sharing* design is expected to be the smallest and slowest. In this chapter its architecture is described and analyzed. This paragraph is divided as the Golden model one, but the main components are not described since they are the same used in the previous design and so they have been already explained.

4.2.1 Datapath

The *first part* is related to the *generation of the zeroing constants* and the *comparison between the two residues associated to the greatest modulus.*

Since the usage of decoders is helping us in the zeroing constants' evaluation, this first part is unchanged with respect of the Golden Model design. Its RTL structure is the same reported in Figure 4.5 and to achieve those first results are needed only **two decoders** of $n+1$ $-to-$ $3n+1$ bits and a $n+1$ bits **comparator** to evaluate the relation between the inputs' m_n-th residues. Also, two registers on $3n+1$ bits and three single bit registers are needed.

The *second part* is designated to *compute the multiples* A_{m_n} and B_{m_n} and the *first result of the multiplication* by the RNS constant [0, 1, 2]. Reusing the resources implies that one multiplication at time is performed, so only the very first one is performed in this cycle obtaining that only **two subtractors** and **one circular shift** are used together with another component that generates all the values included in the interval $[0, N_{m_i})$. This latter component is a modular **counter** because it is formed by 3 counters that works on $n+1$, n and n bits respectively simultaneously to generate the $3n+1$ output. Observing the sequence of the RNS numbers, indeed, it is possible to write each number by always adding 1 and ensuring that each counter reaches its associated maximum value which is $m_i - 1$. This feature is obtained by generating an internal reset to ensure that when the counter gets to $m_i - 1$ it is restarted. In Table 4.6 are reported the maximum values reached by each modulus for all the n considered in order to make clear that the internal reset is applied only to the $2^n - 1$ and $2^n + 1$ moduli, since the 2^n one has its internal reset generated by its overflow.

	2^n-1		2^n		2^n+1	
$n=3$	6	110	7	111	8	1000
$n=5$	30	11110	31	11111	32	100000
$n=8$	254	11111110	255	11111111	256	100000000

Table 4.6

In Figure 4.9 is reported the structure of this counter component in case $n = 3$ in order to clarify its RTL architecture.

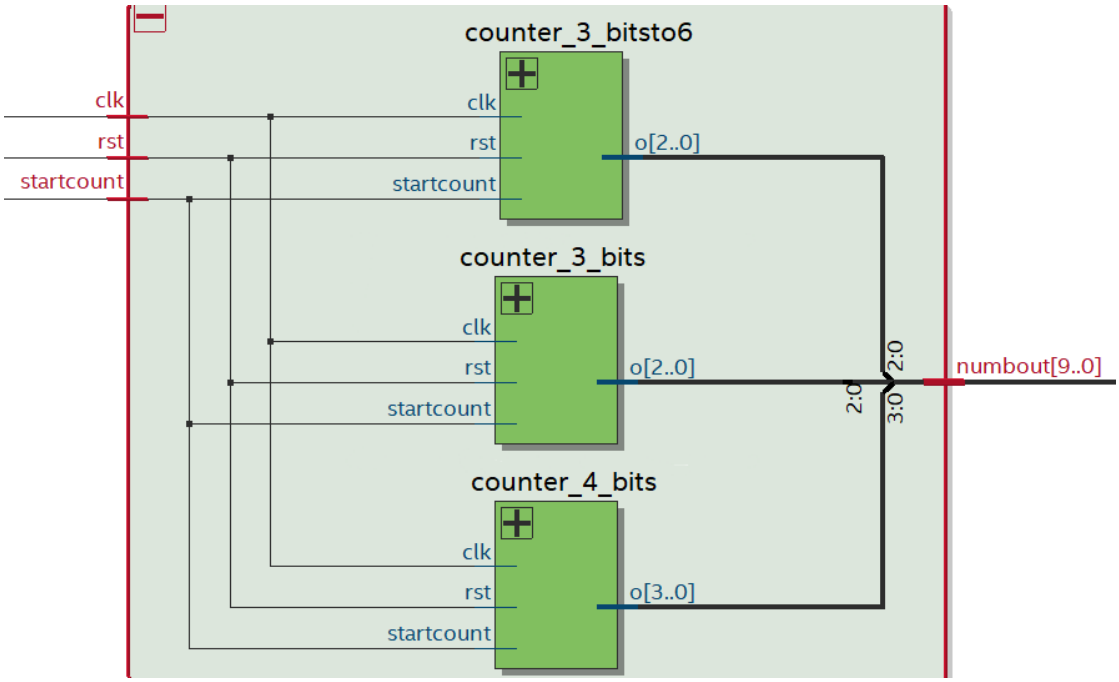

Figure 4.9: Total counter's structure for $n = 3$. When the control signal *startcount* is asserted, the counters start computing all the following values adding always +1. The counters *counter_3_bitsto6* and *counter_4_bits* have an internal reset when they reach the maximum value respectively of 6 and 8.

To evaluate the multiples A_{m_n} and B_{m_n}, as in Equation (3.3), the two subtractors must have as inputs both the A and B values and the zeroing constants previously computed: both their two inputs are selected thanks to **two multiplexers** $2 - to - 1$ of $3n + 1$ bits whose selecting signal is given from the control unit.

In Figure 4.10 is reported the architecture of the first and second part (even if the comparator on $n + 1$ bits is not present) for $n = 3$. It is possible to see, considering the multiplexers mux_1, mux_2, mux_3 and mux_4, that their function is switching between the second and the third part: initially the subtraction is performed between the zeroing constants $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$ and the inputs A and B, then between the multiples A_{m_n} and B_{m_n} and the multiplication output (O_Mult). This is also better underlined in Figure 4.11, while the generation of the multiplication output (O_Mult) is reported in Figure 4.12.

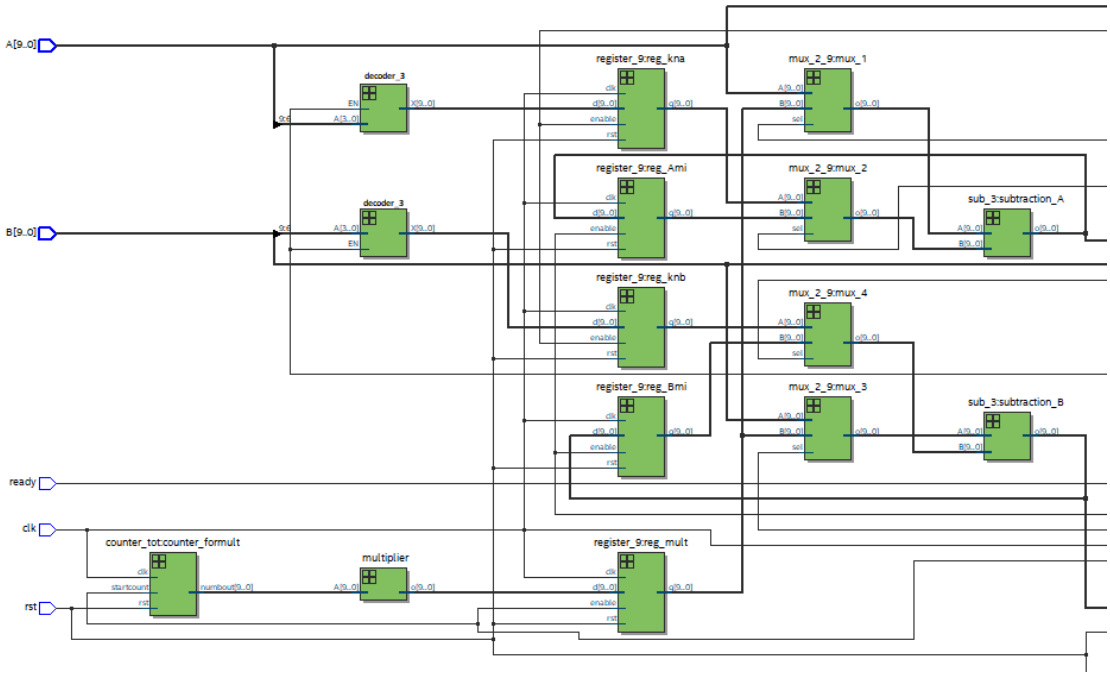

Figure 4.10: Resource sharing's first and second part with the two decoders used to generate the zeroing constants, the registers allocated for the zeroing constants and the multiples, the four multiplexers to select the subtractors inputs and the counters together with the fake-multiplier and the multiplication's output register.

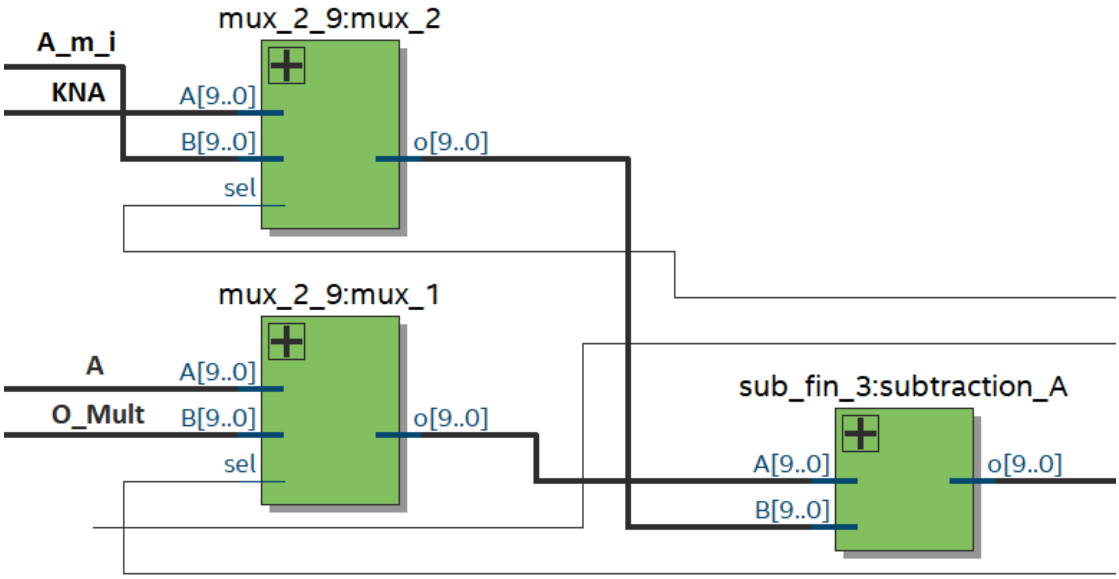

Figure 4.11: The multiplexers and their inputs together with the subtractor used in the Resource sharing's design.

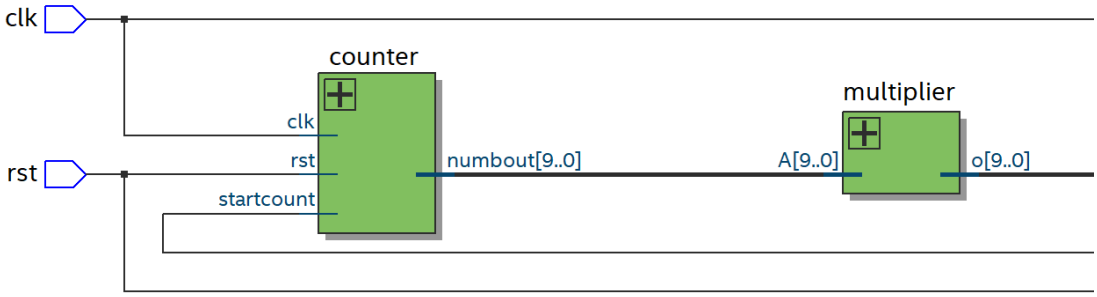

Figure 4.12: The connection between the counter and the fake-multiplier in the Resource sharing's design.

It is possible to highlight that for this architecture the second part is only the preliminary step before the slowest part begins: the *third part*. The second part' s required resources are shared with the third one thanks to the two multiplexers (as underlined in Figure 4.10) since in this third part the inputs of the two subtractors are properly switched taking as new inputs the multiplies A_{m_n} and B_{m_n} and the result of the multiplication correctly stored in a register at each clock cycle.

As for the Golden Model, thanks to the third part the indexes n_A and n_B are evaluated. This is why everytime the subtractors give us a result, they are

checked with an **equality comparison**: if their value is 0, then the index has been identified. To generate the indexes, however it is not possible to use an encoder since we generate a result at each clock cycle. This is the reason why during this latter part another **counter** starts working: it counts all the numbers in the interval $[0, N_{m_i})$, so it is on $log_2(N_{m_i})$ bits, and is synchronized with the multiplication results (ready only after the second part) to take trace of the interval under consideration. The comparison between the two indexes is performed only when all the values in the range $[0, N_{m_n})$ have been analyzed thanks to a signal generated by the binary counter when it ends the counting, called rst_55, making this phase the slowest.

The resources added to perform this step are only two equality comparison for each output and the 2 $log_2(N_{m_i})$-bits registers to store the n_A and n_B values and finally their proper comparator whose outcomes are stored in 3 1-bit registers and eventually selected with another multiplexers driven by a control unit signal, as for the Golden Model.

Since the selection of the multiplexer is given by the control unit, we need to delay the computation by storing the resultant signal $A > B$, $A < B$ and $A = B$ in their proper registers.

In Figure 4.13 and 4.14 are reported those last steps: the generation of the indexes and the selection of the result respectively.

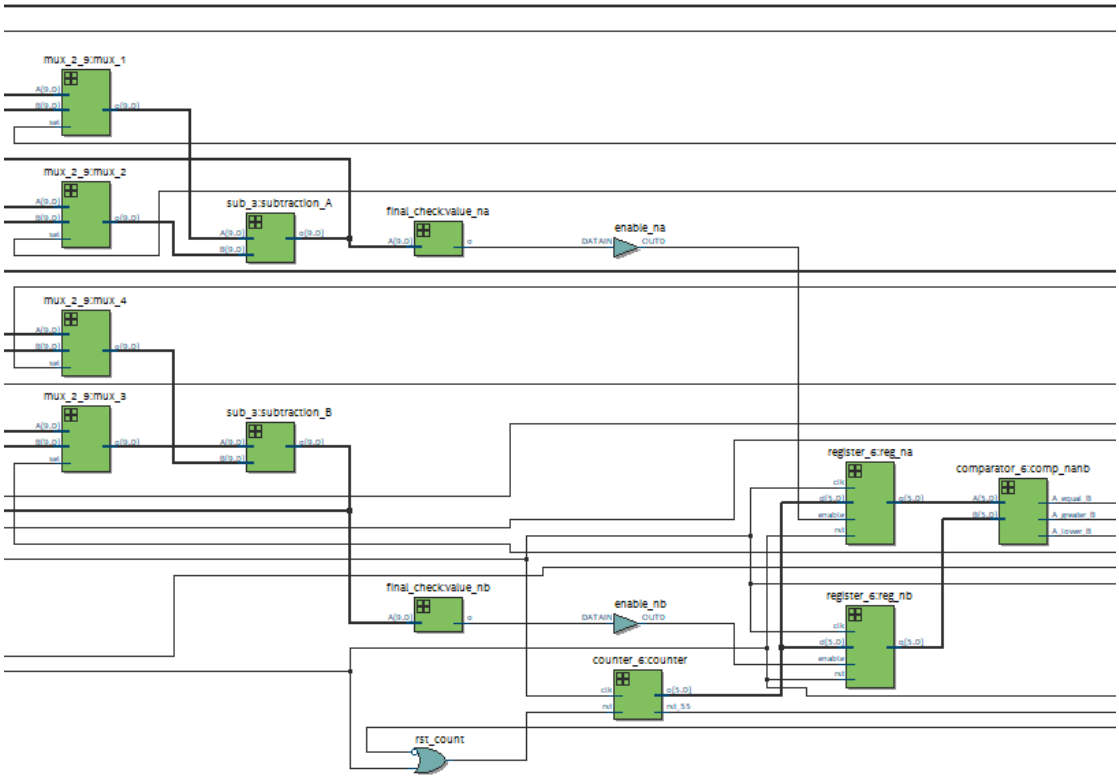

Figure 4.13: The indexes n_A and n_B generation in the third part: the subtraction results are checked thanks to an equality comparator and enables the indexes register if their outcome is asserted. To generate the indexes, it is used a binary counter. After the indexes memorization, those values are compared in a comparator.

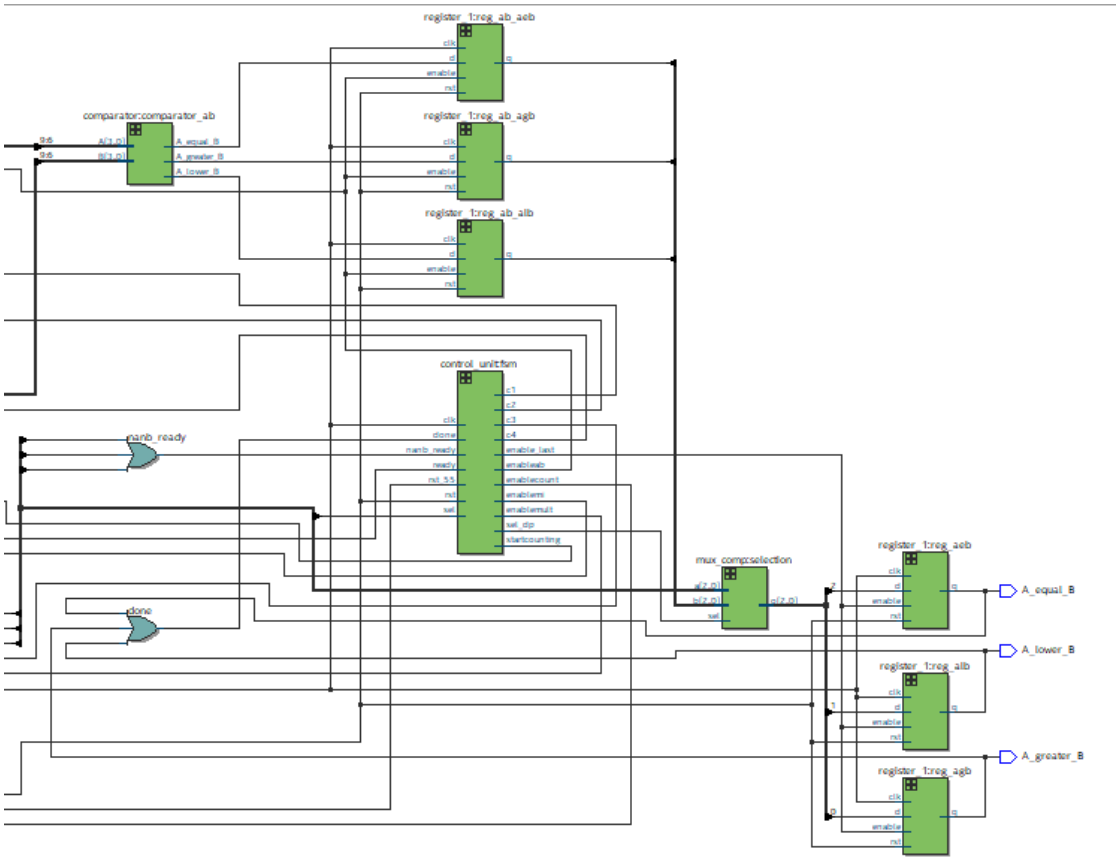

Figure 4.14: The last part of the Resource sharing design: the comparator related to the first part and the registers associated to its outcomes in the upper part, the control unit and the generation of the control signals $nanb\_ready = nA\_greater\_nB\ OR\ nA\_lower\_nB\ OR\ nA\_equal\_nB$ and the $done = A\_greater\_B\ OR\ A\_lower\_B\ OR\ A\_equal\_B$ signal, finally the multiplexer that select the final output which is stored in the registers.

In the Table 4.7 are reported all the used elements, considering the difference between the counter used to generate the multiplication signals and the binary counter to generate the indexes. Moreover, in this case it is not useful to divide the resources in various algorithmic parts because they are reused, but we must keep in mind that both subtractors and "fake-multipliers" are modular.

Circuital part	Number of elements	Element	Number of Bits
Combinatorial	1	Counter with proper reset	$3n+1$
	1	Comparator	$n+1$
	2	Decoders	$n+1\ -to-\ 3n+1$
	2	Subtractors	$3n+1$
	1	Fake-multiplier	$3n+1$
	2	Equality check	$3n+1$
	1	Binary Counter	$log_2(N_{m_i})$
	4	Multiplexers $2\ -to-\ 1$	$3n+1$
	1	Multiplexer $2\ -to-\ 1$	3
Sequential	5	Register	$3n+1$
	2	Register	$log_2(N_{m_i})$
	9	Register	1

Table 4.7

4.2.2 Control Unit

The *control unit* associated to the Resource sharing design is composed almost in the same way of the Golden Model. Its states are 8:

1. **RESET** if the general *reset* is asserted;
2. **IDLE** when the inputs are awaited;
3. **OP_MN** to enable the computation of the zeroing constants and the comparison between the residues a_n and b_n;
4. **OP_SUB** to properly select the inputs of the two subtractors to compute the multiples A_{m_n} and B_{m_n};
5. **OP_SUB_MULT** to compute the multiplication results and properly set the subtractor inputs;
6. **ab** to correctly select the outputs related to the first comparison. The system gets to this state if the last comparison is over and if the signal related to $n_A = n_B$ is asserted;
7. **n** to correctly select the outputs related to the last comparison. The system gets in this state if the last comparison is over and if the signal related to $n_A = n_B$ is negated;

8. **RESULT** when the output is computed and the operation is over. The system evolves in this state if one of the last output signals is asserted and thanks to the proper signal announces the end of operation.

The control unit scheme is reported in Figure 4.15.

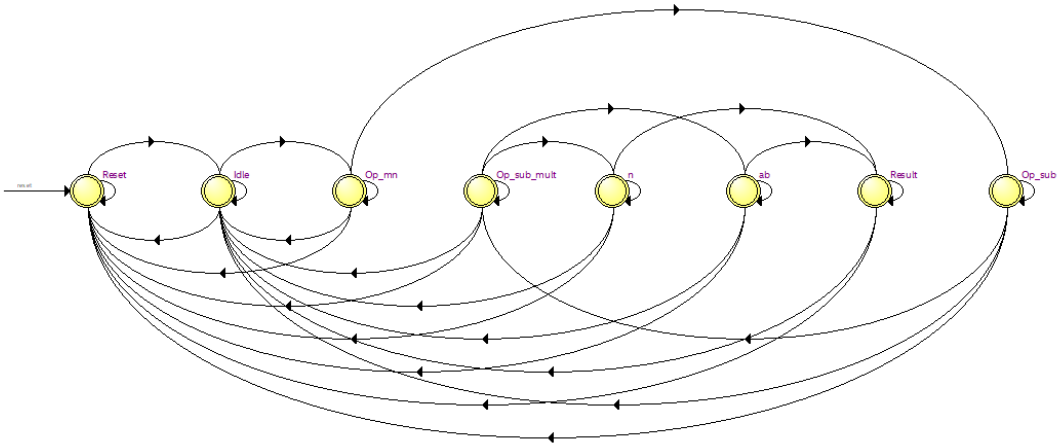

Figure 4.15: Resource sharing's Finite State Machine.

Finally, in Table 4.8 are reported the signals in input while in Table 4.9 the output ones and is described their function and their computation.

Signal	Significance
rst	The general Reset is asserted: the system is resetted.
ready	The inputs signals are ready to be analyzed.
rst_55	This signal is asserted only if the binary counter reaches the N_{m_i} value.
nanb_ready	When both the indexes have been analyzed and their relation has been evaluated, this signal is asserted.
sel	This signal is asserted only if the two indexes n_A and n_B are equal and allows to step in one of the states ab or n.
done	The operation is ended: the result has been computed

Table 4.8

Signal	Significance
c1, c3	Those two signals are selecting in both the two multiplexers the input values A and B when asserted, the result of the multiplication when negated.
c2, c4	Those two signals are selecting in both the two multiplexers the inputs' zeroing contants if asserted, the inputs' multiples when negated.
enablemi	This signal enables the registers associated to the second step's results (the multiples and the first multiplication result).
enablemult	This signal enables the register associated to the third step's results (the multiplication ones) and starts counting the counter associated to the multiplication.
enableab	This signal enables the registers associated to the first step's results (the zeroing constants' registers and the first comparison's results).
enable_last	This signal enables the last registers, the ones associated to the result memorization.
enable_dec	This signal enables the decoders that decide the $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$ values.
enablecount	This signal has been introduced since the generation of the indexes n_A and n_B is delayed with respect of the counting of the values to be multiplied in RNS. Indeed, this signal starts counting the binary counter.
sel_dp	It is the last multiplexer' selection signal deciding if the output depends on the indexes' comparison or the last residue's one.

Table 4.9

4.3 Unfolding of the resource sharing

The most worrying aspect of the Resource sharing design is the required time to end the operation. In fact, it is equal to about N_{m_i} clock cycles (as also described in Figure 4.1), where the different N_{m_i} values are reported in table 4.1. As it is possible to see, the more the n value is increased, also N_{m_i} rises but exponentially getting that the operation execution takes too long. This is why it has been implemented the *Unfolding of Resource sharing* design in which the last operation's execution that takes about N_{m_i} clock cycles is split between more units working in parallel. The unfolding technique, indeed, is used to provide a parallel versions of

the system and allows to achieve delay improvements even at the cost of a larger area. As reported in Equation (3.2), to get an higher performance's improvement and to get a lower control's complexity, the added levels are equal to 2^k with increasing k, so that the expected time reduction is equal to 2^k.

As the previous sections, this architecture is described in the same way with the same divisions.

4.3.1 Datapath

To ensure continuity with the previous designs, the division in three parts has been kept. That is why the division in three parts is still used although the operations done are differently parted.

Again, the *first part*, by which the zeroing constants are computed and the two residues a_n and b_n are compared, is kept unchanged with respect of the resource sharing design. This means that both the two **decoders** of $n+1\ -to-\ 3n+1$ bits and the **comparator** of $n+1$ bits are again used. Also, this first part needs two registers to store the $KN_{m_n}^{A_{RNS}}$ and $KN_{m_n}^{B_{RNS}}$ values and the three 1-bit registers for the comparison's results.

The *second* and *third parts* are modified. During the second one the multiples A_{m_n} and B_{m_n} are computed exactly as in the resource sharing design and, thanks to the unfolding technique, are also evaluated the first 2^k results multiplied by the RNS constant [2, 1, 0]. This is the first main difference with respect of the Resource sharing design. To accomplish this latter operation is necessary to consider all the values included in the interval [0,2^k) and add to each of them 2^k. This adding operation is iterated until is reached the N_{m_i} value, so the best solution is using counters properly started with the internal value. Following the Resource sharing design, all the counters' outputs should be given to the so-called "fake-multiplier" in order to evaluate the multiplication's results. However observing the "fake-multiplier"'s outputs it is possible to identify a sequential trend that could be recreated by modifying the adding constant. To better understand this process, in Table 4.10 is reported the sequential trend considering the unfolding level $2^2 = 4$, in which the starting numbers in the range [0,4) are already multiplied by the RNS constant [2, 1, 0].

[0, 0, 0]	[2, 1 0]	[4, 2, 0]	[6, 3, 0]
[8, 4, 0]	[10, 5 ; 0]	[12, 6, 0]	[14, 7, 0]
[16, 8, 0]	[18, 9, 0]	[20, 10, 0]	[22, 11, 0]

...

Table 4.10

The counters are, indeed, initialized to the first numbers included in the range [0,2^k) and the added value is 2^k in RNS form. In the represented case the first numbers in RNS form are in the range is [0,4) and the adding number is 4. Because of the multiplicative value in RNS [0, 1, 2], the adding value becomes [0, 4, 8].

As reported in the open source software solutions, also in the addition case the residue should be recomputed if its result is higher than the associated modulus as reported in Algorithm 1.

This latter consideration is the reason why the counters must be modified: the residues' counters associated to the 2^n modulus do not need any variation since anytime the result exceed the modulus the most significant bit associated to the overflow is truncated; however the counter associated to the $2^n - 1$ modulus, instead, must be properly set: it cannot count its latter value, but must stop the count before. This goal is achieved by studying the sequential trend and observing that when reached a certain threshold number the adding value is not equal to $2 \cdot 2^k$, as happened for all the previous values, but $2 \cdot 2^k + 1$.

This is why, considering the case $n = 3$ and $k = 2$, the represented sequence becomes the one reported in Table 4.11.

[0, 0, 0]	[2, 1 0]	[4, 2, 0]	[6, 3, 0]
[1, 4, 0]	[3, 5 ; 0]	[5, 6, 0]	[0, 7, 0]
[2, 0, 0]	[4, 1, 0]	[6, 2, 0]	[1, 3, 0]

...

Table 4.11

As it is possible to see, the adding value is [0, 4 1] for all the counters used. In addition, this is considered the limiting value for the $n = 3$ case since for the $k = 3$ case the sequential trend gets harder to replicate.

In the case with $n = 5$ and $k = 2$, instead the first counted values are reported in Table 4.12.

[0, 0, 0]	[2, 1 0]	[4, 2, 0]	[6, 3, 0]
[8, 4, 0]	[10, 5 ; 0]	[12, 6, 0]	[14, 7, 0]
[16, 8, 0]	[18, 9,]	[20, 10, 0]	[22, 11, 0]
[20, 12, 0]	[26, 13 0]	[28, 14, 0]	[30, 15, 0]
[1, 16, 0]	[3, 17 ; 0]	[5, 18, 0]	[7, 19, 0]

...

Table 4.12

The adding value is always [8, 4, 0], except that the counter associated to the modulus $2^n - 1$ counts $2^k + 1 = 9$ when the counting value reaches a certain number

threshold that in this case is equal to 12

This second part needs only 2^k counters properly initialized and built with $2 \cdot 2^k$ subtractors to create the multiples A_{m_n} and B_{m_n}, as in Equation (3.3). As experienced in the resource sharing architecture, this second part also needs two multiplexers for each input in order to select the proper subtractors inputs in this phase: the two inputs and the zeroing constants previously computed. Another difference with respect of the resource sharing design is the registers allocated in this part which are $2 + 2^k$ of $3n + 1$ bits in order to store not only the multiples value but also the multiplication's results outputs.

In the third part, indeed, those $2 + 2^k$ latter results are the new subtraction's inputs, this is why are needed $2 \cdot 2^k$ subtractors to evaluate the difference between both the multiples A_{m_n} and B_{m_n} and the 2^k multiplication's results. The actual subtractors used are 2^k by taking advantage of the multiplexers to switch between the inputs as exploited in the resource sharing design.

In addition, the subtractions' results are check in order to find the indexes value thanks to 2^k equality comparison for each output. As performed in the previous design, when the subtractions' result is equal to zero, then is generated an asserted signal that enables the proper index' register. However, in the Unfolding case 2^k signals are generated by the associated equality check. So the equality comparison' s outputs are given to an encoder $2^k \ -to- \ k$ in order to select the proper index to store in the $log_2(N_{m_i})$ bits register by using a multiplexer $2^k \ -to- \ 1$. As in the resource sharing design, the indexes are generated thanks to 2^k binary counters on $log_2(N_{m_i})$ bits initialized with all the values included in [0,2^k) and that adds the binary 2^k value. Those elements start counting in this third phase to ensure the indexes fairness, just as happened in the previous design.

The *third part* shares two subtractors with the second part and the counters, but it also needs $2^k - 1$ more subtractors and 2^k equality comparators both for each input together with the 2^k external binary counter on $log_2(N_{m_i})$ bits, the two encoders $2^k \ -to- \ k$ and two multiplexers $2^k \ -to- \ 1$ of $3n + 1$ to generate the indexes n_A and n_B. The register used are the 2^k ones shared with the second phase to store the counters' outputs and the two indexes registers on $log_2(N_{m_i})$ bits. Finally the indexes are compared in a $log_2(N_{m_i})$ comparator and its results are given to the last multiplexer $2 \ -to- \ 1$ whose selection signal depends on the control unit. The result's storage needs to be delayed thanks to another register that stores the comparison results, as seen for both the previous comparisons.

In Figure 4.16 is reported the whole Unfolded resource sharing design considering $n = 3$, while in Figure 4.17 we have focused on the counters used to generate the RNS values and the binary ones unfolded.

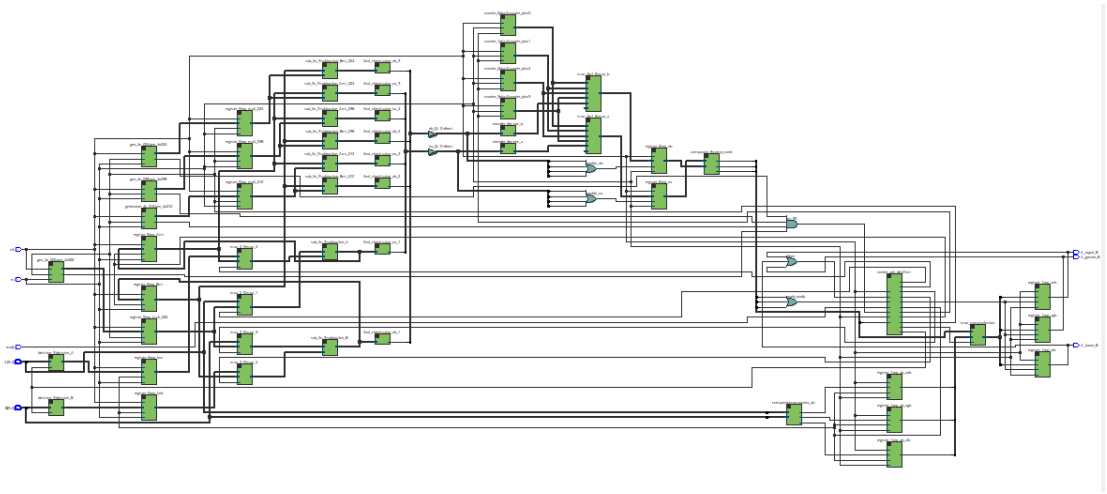

Figure 4.16: Unfolded Resource sharing design for $n = 3$.

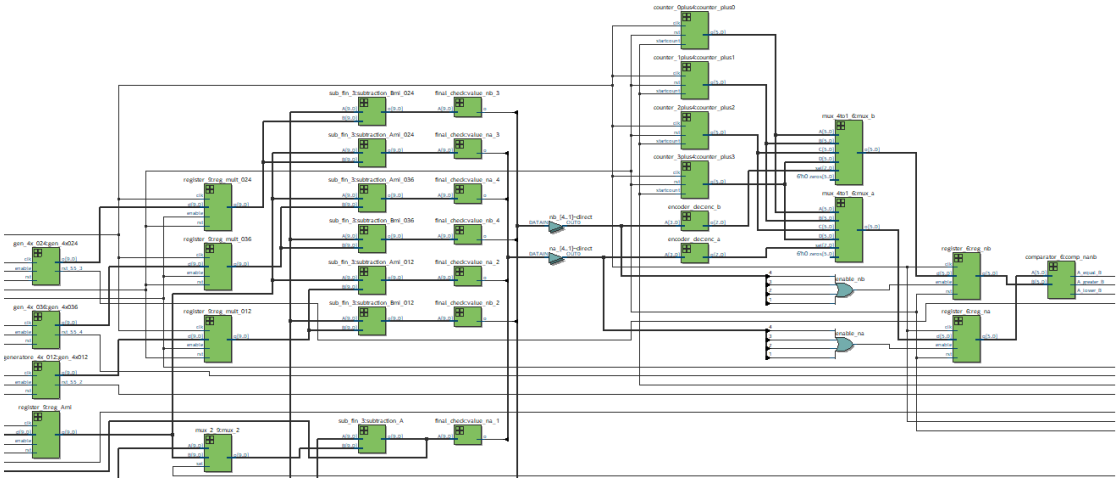

Figure 4.17: Unfolded Resource sharing design for $n = 3$: the counter generates all the multiplied results that are stored in their proper registers, then they are subtracted from the multiplies values and this latter output is detected from the equality comparator. All the outcomes of the 4 equality comparator are stored used from the encoder to select the 4 binary counters' output to be stored in the index' register. Finally those indexes values are compared.

In Table 4.13 are summarized all the elements used.

Circuital part	Number of elements	Element	Number of Bits
Combinatorial	2^k	Counter that generates the values already multiplied	$3n+1$
	2	Decoder	$n+1\ -to-\ 3n+1$
	2	Comparator	$n+1$
	$2 \cdot 2^k$	Subtractors	$3n+1$
	$2 \cdot 2^k$	Equality check	$3n+1$
	2^k	Binary Counter	$log_2(N_{m_i})$
	4	Multiplexers $2\ -to-\ 1$	$3n+1$
	2	Encoders	$2^k\ -to-\ k$
	2	Multiplexers $2^k\ -to-\ 1$	$log_2(N_{m_i})$
	1	Multiplexer $2\ -to-\ 1$	3
Sequential	$4+2^k$	Register	$3n+1$
	2	Register	$log_2(N_{m_i})$
	9	Register	1

Table 4.13

4.3.2 Control Unit

The Unfolding Resource sharing's *control unit* is exactly the same of the Resource sharing one. We can refer to section 4.2.2 without reporting or explaining in here how it works.

The states and control signals are the same and are generated in the same way. The only difference is the rst_55 signal which is generated when all the counters used to generate the multiplication's results have finished the count. This choice depends especially on the need to verify the multiplication's results' fairness since this signal is asserted only when all the counters reaches the final result thanks to an AND gate.

Chapter 5

Results

All the designs' results are discussed in this chapter: in 5.1 all the designs' performances with different n and k are reported, then those results are commented and compared with the existing architectures' performances in 5.2.

5.1 Results

The aim of this section is describing all the architectures and commenting their performances in terms of delay, area and power consumption.

All the designs structures have been described in **VHDL** and simulated in **ModelSim**, while the synthesis are performed with **Synopsys Design Compiler** using the technology library *"uk65lscllmvbbr_090c125_wc"* considering the standard cell *BUFM14R*.

5.1.1 Golden model n=3

The first design analyzed is the Golden model for $n = 3$. It is the fastest design as expected and reported in its simulation in Figure 5.1.

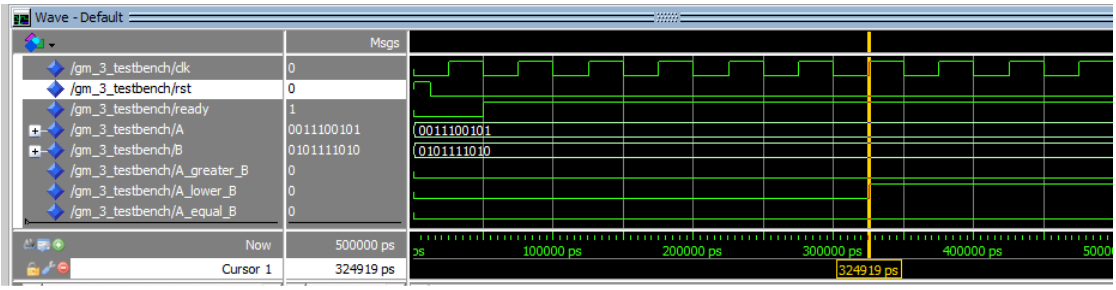

Figure 5.1: Golden model design simulation for $n = 3$ with inputs $A = 12$ and $B = 23$: the result is $A < B$.

Even if this design is the fastest and needs only 5 clock cycles to end the computation, its structure is the widest one. Its performances are reported in Table 5.1.

Golden model design's results for $n = 3$	
f_{MAX}	120.9 MHz
Total Area	27149.7 μm^2
Total Power	458.4 μW

Table 5.1

Thanks to this analysis, this system will give the result after 41.35 ns.

5.1.2 Resource sharing n=3

The Resource sharing design with $n = 3$ is expected to take at least 56 clock cycles. In Figure 5.2 is demonstrated that to conclude the operation are required 61 clock cycles.

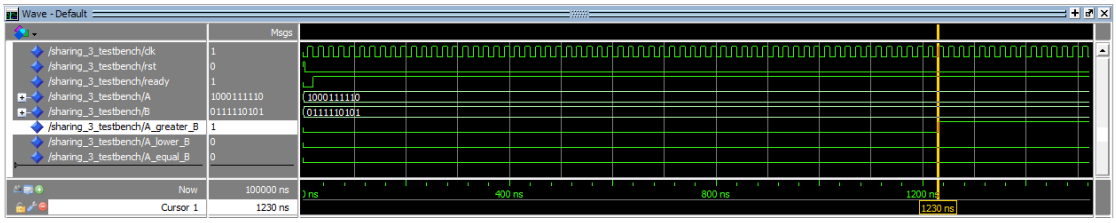

Figure 5.2: Resource sharing design simulation for $n = 3$ with inputs $A = 503$ and $B = 502$: the result is $A > B$.

However, its performances are improved with respect of the Golden model design as confirmed in Table 5.2.

Resource sharing design's results for $n = 3$	
f_{MAX}	221.7 MHz
Total Area	2447.3 μm^2
Total Power	125.4 μW

Table 5.2

As expected, the area reduction with respect of the Golden Model is reduced of 90.9%, the power consumption decreases as 72.6% and finally the maximum frequency is increased of about 45.5%. Even if the performances are improved, the result is ready only after 275.11 ns.

5.1.3 Unfolded resource sharing n=3 with k=2

The only Unfolding level applied for $n = 3$ is the one with $k = 2$ since increasing the unfolding level would have meant an higher algorithmic complexity. The counter associated to the $2^n - 1$ modulus, instead, must be properly set: it cannot count its latter value, but must stop the count before. This goal is achieved by studying the sequential trend and observing that when reached a certain threshold number the adding value is not equal to $2 \cdot 2^k$, as happened for all the previous values, but $2 \cdot 2^k + 1$.

In this first case $2^k = 4$, so the number to be added in RNS is [8, 4, 0]: the counter associated to the modulus 8 adds everytime +4, however the one associated to 7 should add +1, the residue value of 8 with respect to 7. This is why the counter associated to the modulus 7 is always counting +1 but is resetted to 0 after counting the value 6.

In this case the clock cycles expected are almost 14, while the actual ones are 18 as highlighted in Figure 5.3.

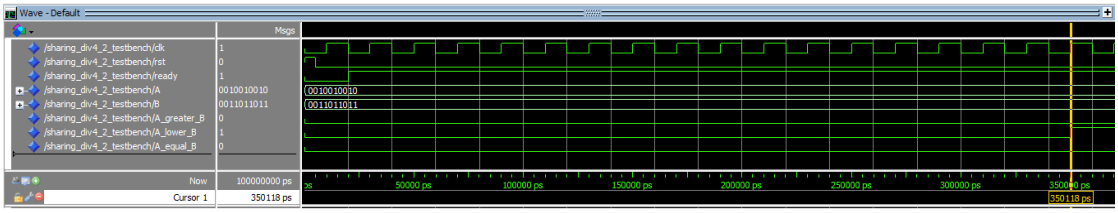

Figure 5.3: Unfolded resource sharing $n = 3$ with $k = 2$ design simulation with inputs $A = 2$ and $B = 3$: the result is $A < B$.

Even if the needed clock cycles are reduced of about 70% with respect of the Resource sharing one, its performances got worst with respect of the Resource sharing design as reported in Table 5.3

Unfolded Resource sharing design's results for $n = 3$ with $k = 2$	
f_{MAX}	165.1 MHz
Total Area	5045.1 μm^2
Total Power	151.7 μW

Table 5.3

The occupied area is increased of about 51.5% with respect of the Resource sharing model, but still reduced with respect to the Golden model (81.4%). Also its power consumption gets 17.3% worse with respect of the Resource sharing design, but 66.9% better than the Golden model. Its maximum frequency is instead reduced with respect of the Resource sharing architecture of 25.5%, while it is 26.8% increased with respect of the Golden model design.

In this case the result is ready only after 109.08 ns, which is improved of 60.3% with respect of the Resource sharing model.

5.1.4 Golden model n=5

In Figure 5.4 is reported the ModelSim simulation of the Golden Model design.

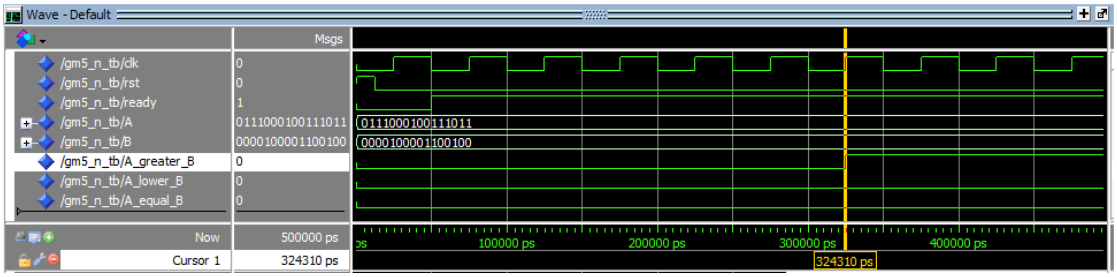

Figure 5.4: Golden model design for $n = 5$ with inputs $A = 2569$ and $B = 35$: the result is $A > B$.

In Table 5.4 are reported the Golden model with $n = 5$'s performances.

Golden model design's results for $n = 5$	
f_{MAX}	83.6 MHz
Total Area	717799.3 μm^2
Total Power	8377.0 μW

Table 5.4

The result is generated from this architecture after 71.7 ns

5.1.5 Resource sharing n=5

The case of the Resource Sharing design for $n = 5$ is expected to need almost 992 clock cycles to end the computation. In Figure 5.5 is reported that it needs instead 998 clock cycles.

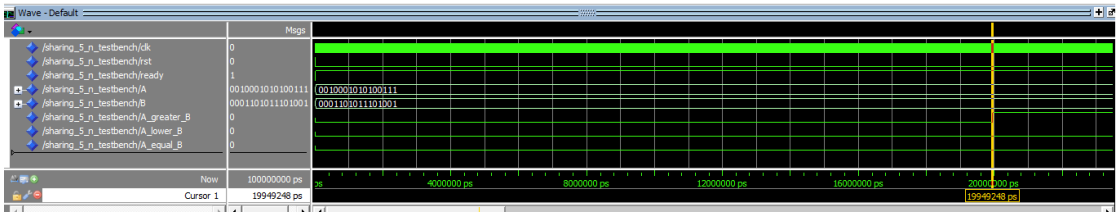

Figure 5.5: Resource sharing's Finite State Machine.

As expected, it takes more clock cycles with respect of the $n = 3$ case, since the N_{m_i} value is greater. Its performances are reported in Table 5.5.

Resource sharing design's results for $n = 5$	
f_{MAX}	189.7 MHz
Total Area	3866.4 μm^2
Total Power	165.1 μW

Table 5.5

As expected it is 98% slower than the golden model design, however it requires the 99.5% less area and dissipates 98% less power. Also the resource sharing's maximum frequency is 56% higher than the Golden model's one.

In this case the result is obtained only after 5.26 μs.

5.1.6 Unfolded resource sharing n=5 with k=2

The first Unfolding case considered is the one with $k = 2$, which means that the adding value is [8, 4, 0]. This value is fully representable in the $n = 5$ case, except that there is a limiting value in the representation associated to the modulus 31. Indeed, when the counted value is higher than 12, the adding value becomes 9 rather than 8. In Figure 5.6 is reported that 252 clock cycles are needed to evaluate the result.

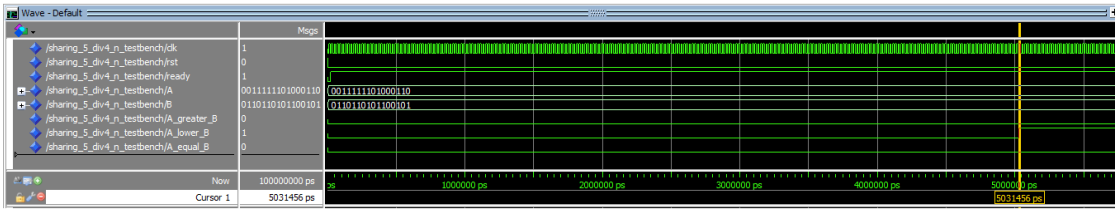

Figure 5.6: Unfolded resource sharing $n = 5$ with $k = 2$ design simulation with inputs $A = 378$ and $B = 4779$: the result is $A < B$.

As reported in Table 5.6, this architecture occupies an area which is 54% higher than the Resource sharing design, dissipates the 25.5% more, but its maximum frequency is 20.9% lower. However, it requires an area 98.8% lower and dissipates 97% less power than the Golden model design, while its maximum frequency is 44.3% higher than the Golden model design.

Unfolded resource sharing design's results for $n = 5$ with $k = 2$	
f_{MAX}	150.1 MHz
Total Area	8406.3 μm^2
Total Power	221.7 μW

Table 5.6

Finally this architecture needs 1.68 μs to generate an output, 68% lower than the Resource sharing design but 95.7% higher than the Golden model architecture.

5.1.7 Unfolded resource sharing n=5 with k=3

When the unfolding level gets equal to $2^3 = 8$, the counter's adding RNS value becomes [16, 8, 0]. This implies that the counter associated to the modulus 32 has to sum always 8, the modulus 31's counter has to sum 16 to all the values lower than 15, and 17 to all the higher ones. In Figure 5.7 is reported the Unfolded resource sharing' simulation in ModelSim in which the clock cycles needed to get the output are equal to 128, in line to the expectation.

From the results in Table 5.7, it is possible to see that the higher unfolding level has brought an higher area of 38% with respect of the Unfolding with $k = 2$, and 71.4% with respect of the Resource sharing design, but the area is reduced of 98% with respect of the Golden model architecture. The power consumption is also higher with respect of both the previous designs (20% with respect of the unfolding level $k = 2$ and 40.5% with respect of the resource sharing design) and 96% lower than the Golden model design. The maximum frequency is instead reduced of 16.3% with respect of the unfolding with $k = 2$ and of 33.8% with respect of the

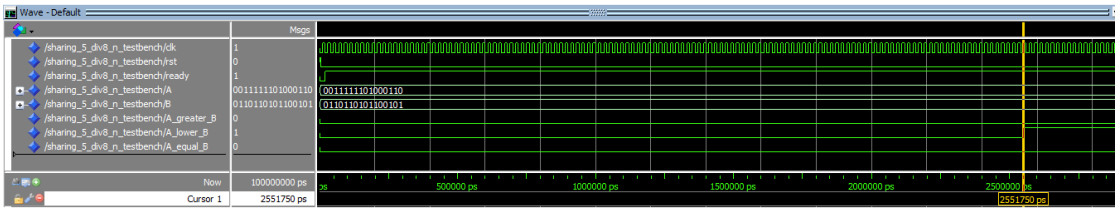

Figure 5.7: Unfolded resource sharing $n = 5$ with $k = 3$ design simulation with inputs $A = 378$ and $B = 4779$: the result is $A < B$.

resource sharing one, but it is increased of 44.3% with respect of the Golden model design.

Unfolded resource sharing design's results for $n = 5$ with $k = 3$	
f_{MAX}	125.6 MHz
Total Area	13537.8 μm^2
Total Power	277.7 μW

Table 5.7

To get the result are needed 1.02 μs which is a better result with respect of both the other designs (39.3% with respect of the unfolding level $k = 2$ and 80% with respect of the resource sharing design). However it takes 91.8% more than the Golden model to generate the result.

5.1.8 Unfolded resource sharing n=5 with k=4

This Unfolding level is the limiting one for the $n = 5$ design. Indeed, for $k = 5$ the counters are going to be more complicated to build. As happened in the design for $n = 3$ with unfolding level $k = 2$, considering $k = 4$, the adding RNS value becomes $[32,\ 16,\ 0]$, so that the counter associated to the modulus 32 has not to be modified, the modulus 31's counter has to count +1, which is equal to the residue of 32 with respect to the value 31. Naturally, this latter counter cannot count the last two values, so it is resetted when it reaches the 30 value.

In this case, reported in Figure 5.8 the required clock cycles are 66, in line with the expected result.

In Table 5.8 are reported the Unfolded resource sharing for $n = 5$ and $k = 4$ performances.

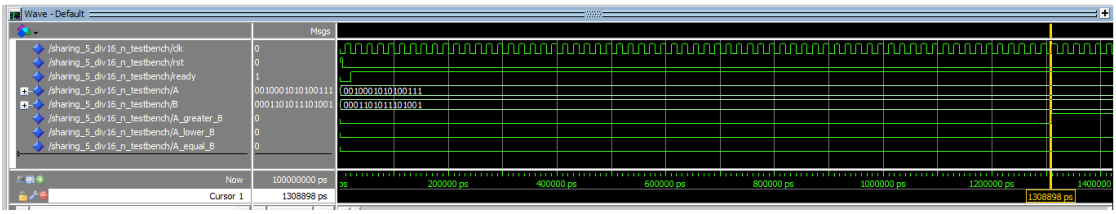

Figure 5.8: Unfolded resource sharing $n = 5$ with $k = 4$ design simulation with inputs $A = 2549$ and $B = 567$: the result is $A > B$.

Unfolded resource sharing design's results for $n = 5$ with $k = 4$	
f_{MAX}	117.5 MHz
Total Area	26598.2 μm^2
Total Power	430.3 μW

Table 5.8

This design is for sure the widest in terms of area since it requires 85%, 68% and 49% more than respectively the resource sharing design and both the unfolded ones for $k = 2$ and $k = 3$. In terms of power it also dissipates more than the previous architectures (61.6% than the resource sharing, 48.5% than the unfolded with $k = 2$ and 35.5% than the $k = 3$ unfolded). Its maximum frequency is instead reduced than the previous designs respectively of 38% than the resource sharing, 21.7% than the unfolded with $k = 2$, and 6.5% than the unfolded with $k = 3$. However, the area and power consumption are still lower than the Golden model respectively of 93% and 94.8%, while the maximum frequency is 28.8%.

To generate the result it requires 561.6 ns, which is highly reduced with respect of all the previous architectures (89.3% than the resource sharing, 66.6% than the unfolding with $k = 2$ and 44.9% for the unfolding with $k = 3$). This design requires also 87% more time to generate the output with respect of the Golden Model design.

5.1.9 Resource sharing n=8

In Figure 5.9 is reported the simulation result for the Resource sharing design with $n = 8$. To generate the output it requires 65286 clock cycles, in line with the expected results.

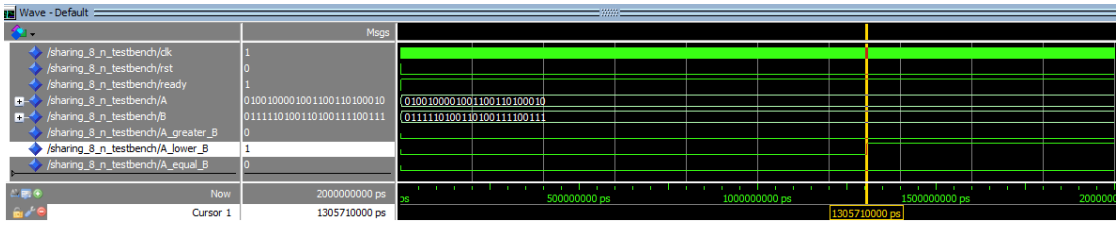

Figure 5.9: Resource sharing design for $n = 8$ and inputs $A = 2457$ and $B = 489321$: the result is $A < B$.

In Table 5.9 are reported the performances associated to this design.

Resource sharing design's results for $n = 8$	
f_{MAX}	165.8 MHz
Total Area	5817.6 μm^2
Total Power	211.9 μW

Table 5.9

The required time to generate the result is 393.67 μs.

5.1.10 Unfolded resource sharing n=8 with k=2

The first Unfolded level considered is $2^k = 4$ and the adding number in RNS is $[8,\ 4,\ 0]$ which is fully representable with this moduli set. The limiting value applied to the modulus 255's counter is 247: 8 is the adding number for all the values lower than this threshold, while 9 is the one for the higher values. In Figure 5.10 is reported the simulation in ModelSim associated to this design. It takes 16325 clock cycles to generate the results.

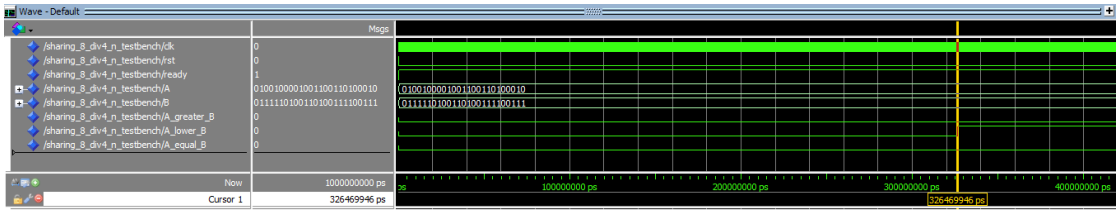

Figure 5.10: Unfolded resource sharing $n = 8$ with $k = 2$ design simulation with inputs $A = 2457$ and $B = 489321$: the result is $A < B$.

In Table 5.10 are reported the performances associated to the Unfolded resource sharing design with $n = 8$ and $k = 2$. As expected, this design occupies more area

and dissipates more power with respect to the resource sharing design (54% and 32% respectively), but its maximum frequency is decreased of about 29%.

Unfolded resource sharing design's results $n = 8$ with $k = 2$	
f_{MAX}	138.7 MHz
Total Area	12670.9 μm^2
Total Power	311.7 μW

Table 5.10

It takes 117.7 μs to generate the output, which is an improvement of the 70% with respect of the resource sharing design.

5.1.11 Unfolded resource sharing n=8 with k=3

The design with unfolding level equal to $2^3 = 8$ needs the counters to add the RNS number [16, 8, 0]. In this case the threshold number associated to the modulus 255's counter is 240: the counter adds 16 when the counting number is under the threshold, but it adds 17 when the counting number is higher than 240. In Figure 5.11 it is possible to observe that to generate the output are needed 8164 clock cycles.

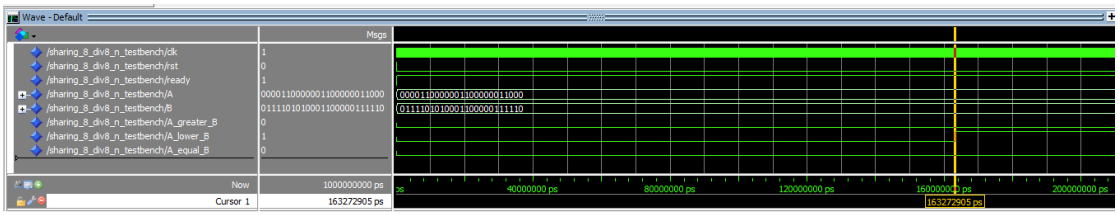

Figure 5.11: Unfolded resource sharing $n = 8$ with $k = 3$ design simulation with inputs $A = 24$ and $B = 75032$: the result is $A < B$.

In Table 5.11 are reported the performances of the Unfolding resource sharing design with $n = 8$ and $k = 3$. As expected it dissipates 50.5% more power with respect of the resource sharing design and the 27% more than the Unfolded level with $k = 2$. It also requires the 74% more area with respect of the resource sharing design and 43.4% more than the Unfolding level with $k = 2$. Although, it has a maximum frequency reduced of 26.9% with respect of the resource sharing design and of the 12.7% with respect of the unfolding level $k = 2$.

Unfolded resource sharing design's results $n = 8$ **with** $k = 3$	
f_{MAX}	121.1 MHz
Total Area	22387.7 μm^2
Total Power	428.1 μW

Table 5.11

It needs 67.43 μs to generate the output, improving the required time of 82.9% and 42.7% respectively for the resource sharing and the unfolding level $k = 2$.

5.1.12 Unfolded resource sharing n=8 with k=4

The Unfolded resource sharing with $k = 4$ uses counters that adds the RNS number equal to [32, 16, 0]. Since the moduli set used is the $n = 8$ one, it is possible to fully represent and add those numbers, even if the modulus 255's counter has a limiting threshold of 224 so that it is added 32 to all the counted numbers lower than this value, but when the counted values are higher it is added 33. In Figure 5.12 it is reported that this comparator needs 4084 clock cycles to generate the result.

Figure 5.12: Unfolded resource sharing $n = 8$ with $k = 4$ design simulation with inputs $A = 24$ and $B = 75032$: the result is $A < B$.

In Table 5.12 are reported the design's performances.

Unfolded resource sharing design's results $n = 8$ **with** $k = 4$	
f_{MAX}	112.5 MHz
Total Area	45565.2 μm^2
Total Power	676.1 μW

Table 5.12

As expected, its area and power consumption are higher than all the previous analyzed designs: 87.2% more area and 68.7% more power dissipated with respect of the resource sharing design, 72% more area and 53.9% more power dissipated

with respect of the unfolding with $k = 2$ design and finally 50.8% more area and 36.6% more power dissipated with respect of the unfolding with $k = 3$ design. Its maximum frequency is reduced of 32.2% with respect of the resource sharing design, 18.9% with respect of the unfolding with $k = 2$ design and 7.1% with respect of the unfolding with $k = 3$ design.

However this is the design that takes less time to generate the result: 36.3 μs, reduced of 90.7% with respect of the resource sharing design, 69.1% with respect of the unfolding with $k = 2$ design and 46.2% with respect of the unfolding with $k = 3$ design.

5.2 Comparison with existing methods

The designs' results have been compared with respect of the actual comparison circuits in RNS with moduli set $[2^n - 1,\ 2^n,\ 2^n + 1]$. The analysis has been performed with some selected articles that reports all the results in terms of area, delay and power consumption of the proper VLSI designs synthesized. Moreover, the proposed architectures results have been adapted in order to correctly handle the comparison.

The chosen parameters also depends on the results reported in the selected articles. Considering [20], the comparison has been conducted on the parameter $Area \times Time$, evaluated as the product between the total area and the time period, and the total power consumption.

Besides, even if the proposed designs are synthesised on a 65 nm technology, not all the actual comparators are synthesised using the same technology. This is the reason why the values of delay and area are reconverted using respectively the Equations (5.1) and (5.2), while the total power consumption is normalized with respect of the technology as in Equation (5.3).

$$DELAY_{TARGET} = \frac{DELAY}{SIZE_{FEATURE}} \times SIZE_{TARGET} \tag{5.1}$$

$$AREA_{TARGET} = \frac{AREA}{SIZE^2_{FEATURE}} \times SIZE^2_{TARGET} \tag{5.2}$$

$$POWER_{NORMALIZED} = POWER_{TOT} \frac{V^2_{DD_{TARGET}} \times SIZE_{TARGET}}{SIZE_{FEATURE} \times V^2_{DD_{FEATURE}}} \tag{5.3}$$

After fixing the $SIZE_{TARGET} = 65\ nm$ parameter, the $SIZE_{FEATURE}$ depends on the used technology: for each considered comparator's implementation the sythesised technology is reported in the last column of both Table 5.14 (for $n = 8$

designs) and Table 5.15 (for $n = 5$ designs). To normalize the total power consumption the V_{DD} general values considered, which also depends on the synthesised technology, are reported in Table 5.13.

SIZE $[nm]$	V_{DD} $[V]$
65	1.1
90	1.3
130	1.5
180	1.8

Table 5.13

The VLSI designs compared with the proposed ones in Table 5.14 (considering $n = 8$) are described in the following selected articles:

- [20]: This VLSI architecture performs the comparison in RNS using an algorithm that evaluates the result of the subtraction between all the residues and reconvert this latter result with the Mixed Radix approach. The design' synthesis has been done using a 65 nm technology, however the only synthesised results reported are the $A \times T$ product and the total dissipated power.

- [3]: This VLSI implementation is one of the comparison technique most similiar to the PANC method. Indeed, this technique is based on partitioning the range in subranges and then evaluate the comparison between every subrange associated to the two lowest moduli in the moduli set. Its drawback is associated on the representation of different partitioning functions that can be aided thanks to the usage of the *CRT*. To achieve the conversion with the CRT, it is necessary to speed up the addition operation, so this paper presents the results of the performances of the comparator both using a *Regular Parallel Prefix*, RPP, or a *Totally Parallel Prefix*, TPP, modular adders. Summarizing, this algorithm uses the partial conversion in CRT to evaluate the subranges associated to each modulus. This architecture has been synthesised in a 130 nm technology and both delay and area are reported together with the power consumption syntheses' results. Even if after the conversion in a 65 nm technology the area results are similar to the ones of the proposed designs, the two article's architectures, RPP and TPP achieve a much lower delay with respect of our designs that is translated in a lower $A \times T$ product. However the power consumption is much higher as reported in Table 5.14.

- [17]: This VLSI architecture is the simplest since the evaluation of the comparison result is achieved by subtracting the two operands and then converting

the result to binary using the CRT reconversion method. This design is synthesized in 90 nm, but the associated results in terms of power consumption are not reported.

- [11]: This comparison method is based on the conversion of the two numbers to be compared using a combination of the CRT and MRC conversion methods and then it requires a binary comparison of the two values. The referring article provides FPGA performance measures that cannot be fairly compared with our results. Luckily, in [3] its ASIC realization has been synthesised in 130 nm. Even if this architecture uses uses a TPP, Total Parallel Prefix, adder structure that speed up the conversion, the power dissipated is still higher with respect of our solutions.

- [18]: This VLSI architecture is subtraction-based: after the subtraction the result is partially reconverted thanks to the CRT technique. This VLSI architecture has been synthesized in 90 nm in [17], so the power consumption results are not reported.

The comparison results reported in Table 5.14 are associated to the implementations with $n = 8$. Only the method in [17] and [18] have been implemented for $n = 5$, whose results are reported in Table 5.15.

$n = 8$				
Architecture	**Comparison method**	**A × T** $[\mu m^2 \cdot \mu s]$	**Normalized total power consumption** $[mW]$	**Techn.**
Resource sharing	PANC	35	0.212	65 nm
Unfolding $k = 2$	PANC	91	0.312	65 nm
Unfolding $k = 3$	PANC	184	0.428	65 nm
Unfolding $k = 4$	PANC	405	0.676	65 nm
[20]	Subtracting and analyzing the signs then converting in MR	≈ 0.3	≈ 15	65 nm
$[3]_{RPP}$	Partitioning via CRT conversion	5	3.8	130 nm
$[3]_{TPP}$		6.2	4.8	130 nm
[11]	CRT and MRC conversion	6	5.3	130 nm
[17]	Subtracting and converting via CRT	≈ 2.3	$N.A.$	90 nm
[18]	Subtracting and partially converting via CRT	≈ 3.3	$N.A.$	90 nm

Table 5.14

$n = 5$				
Architecture	**Comparison method**	**A** $\times$ **T** $[\mu m^2 \cdot \mu s]$	**Normalized total power consumption** $[mW]$	**Techn.**
Golden model	PANC	8.5×10^3	8.3	65 nm
Resource sharing	PANC	20	0.165	65 nm
Unfolding $k = 2$	PANC	56	0.222	65 nm
Unfolding $k = 3$	PANC	107	0.277	65 nm
Unfolding $k = 4$	PANC	226	0.430	65 nm
[17]	Subtracting and converting via CRT	≈ 1.2	$N.A.$	90 nm
[18]	Subtracting and partially converting via CRT	≈ 2.4	$N.A.$	90 nm

Table 5.15

To better understand those results, in Tables 5.16 and 5.17 are reported the starting values, the converted with the formulas in Equation (5.1) and (5.2) of both the results respectively of delay and area. In this analysis [20] is not considered since the separate results of area and delay are not reported.

Arch.	**T** $[ns]$	**Conv T** $[ns]$	**f** $[MHz]$	**Conv f** $[MHz]$	**Area** $[\mu m^2]$	**Conv Area** $[\mu m^2]$
Resource Sharing		6.03		165.8		5817.6
Unfolded k=2		7.2		138.7		12670.9
Unfolded k=3		8.26		121.1		22387.7
Unfolded k=4		8.89		112.5		45565.2
$[3]_{RPP}$	1.43	0.715	699	1398	27761	6940.25
$[3]_{TPP}$	1.42	0.71	704	1408	35157	8789.25
[11]	1.44	0.720	694	1388	33486	8371.5
[17]	1.48	1.07	675	935	4200	2190.7
[18]	1.6	1.2	625	865	5500	2868.8

Table 5.16

Arch.	**T** $[ns]$	**Conv T** $[ns]$	**f** $[MHz]$	**Conv f** $[MHz]$	**Area** $[\mu m^2]$	**Conv Area** $[\mu m^2]$
Golden Model		11.96		83.6		717799.3
Resource Sharing		6.03		165.8		5817.6
Unfolded k=2		7.2		138.7		12670.9
Unfolded k=3		8.26		121.1		22387.7
Unfolded k=4		8.89		112.5		45565.2
[17]	1.25	0.9	800	1111	2500	1304
[18]	1.55	1.1	645	909	4000	2086

Table 5.17

From both the Tables 5.16 and 5.17, it is possible to understand that the proposed designs have a larger area and an higher period with respect of the RNS comparison's state of the art. This translates in an $A \times T$ product greater of two order of magnitude, which is a main drawback of our designs.

However, considering the power consumption its highest contribute is the *dynamic power* evaluated as in Equation (5.4).

$$P_{dyn} = \alpha C f V_{DD}^2 \tag{5.4}$$

where αC represents the effective commuting capacity evaluated as the product of the capacitance C and the *switching activity* α, f is the frequency and V_{DD} is the supply voltage.

From this latter consideration it is possible to see that the low power consumption results, exploited in both Tables 5.14 and 5.15, are related not only to the operational frequency lower of one order of magnitude with respect of the RNS comparison's state of the art, but mostly to the reduced switching activity.

In the Resource sharing and Unfolding designs are indeed used counters that are switched-on only when necessary in the total computation, optimizing specifically the switching activity parameter.

Chapter 6

Conclusion and future implementations

All the RNS comparators created following the PANC method present a normalized power consumption lower than the the other implementations. As expected, the normalized power increases with n and with the unfolding level k, but reaches its highest value considering the Golden model implementation because of the large quantity of elements used.

However, the $A \times T$ values of our architecture are much higher than the ones associated to the other implementations. In the possible future implementations this drawback can be solved by speeding up the period either applying pipeline levels both internal or external to the combinatorial devices to speed up the computation or implementing the combinatorial elements, responsible for the slow clock cycle, with existing faster ones exploited in RNS.

To obtain instead an higher throughput, it is possible to increase the unfolding level k, conscious that the area will increase as well.

However, another option to be better exploited should be slightly modify the algorithm by avoiding the indexes generation and their comparison, but using their identification as flags so that the first one to be point out indicates that its input is the lower; if both those flags are in the same moment asserted, then the output of the comparison between the residues associated to the highest modulus is used. This structure should be particularly efficient for the resource sharing and the unfolded resource sharing structures, but should be better exploited for the golden model one.

In conclusion, this implementation is significantly advantageous in RNS datapath since all the structures can be reutilized. As exploited before, its performances can

be still improved in different ways, however its power results allow this architecture to be used in low-power applications, such as IoT.

Bibliography

[1] Amir Sabbagh Molahosseini, Azadeh Alsadat Emrani Zarandi, Paulo Martins, and Leonel Sousa. «A Multifunctional Unit for Designing Efficient RNS-Based Datapaths». In: *IEEE Access* 5 (2017), pp. 25972–25986. DOI: 10.1109/ACCESS.2017.2766841 (cit. on pp. ii, 3, 6).

[2] Victor Krasnobayev, Sergey Koshman, Kostiantyn Myslyvtsev, Kateryna Kuznetsova, Tetiana Ivko, and Tetiana Katkova. «Method of Arithmetic Comparison of Data in the Residue Numeral System». In: *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC ST)*. 2019, pp. 483–487. DOI: 10.1109/PICST47496.2019.9061381 (cit. on pp. ii, 1, 2, 12).

[3] Zeinab Torabi and Ghassem Jaberipur. «Low-Power/Cost RNS Comparison via Partitioning the Dynamic Range». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.5 (2016), pp. 1849–1857. DOI: 10.1109/TVLSI.2015.2484618 (cit. on pp. iii, 5, 7, 10, 11, 56–59).

[4] Chip-Hong Chang, Amir Sabbagh Molahosseini, Azadeh Alsadat Emrani Zarandi, and Tian Fatt Tay. «Residue Number Systems: A New Paradigm to Datapath Optimization for Low-Power and High-Performance Digital Signal Processing Applications». In: *IEEE Circuits and Systems Magazine* 15.4 (2015), pp. 26–44. DOI: 10.1109/MCAS.2015.2484118 (cit. on pp. 1, 3, 4, 6).

[5] P.A. Lyakhov, M.V. Valueva, D.I. Kaplun, and A.S. Voznesensky. «A New Method of Sign Detection in RNS Based on Modified Chinese Remainder Theorem». In: *2021 10th Mediterranean Conference on Embedded Computing (MECO)*. 2021, pp. 1–4. DOI: 10.1109/MECO52532.2021.9460255 (cit. on pp. 1, 22).

[6] Nikolay I. Chervyakov, Pavel A. Lyakhov, Nikolay N. Nagornov, Maria V. Valueva, and Dmitrii I. Kaplun. «High-Performance Hardware 3D Medical Imaging using Wavelets in the Residue Number System». In: *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. 2020, pp. 1–4. DOI: 10.1109/MECO49872.2020.9134123 (cit. on p. 1).

[7] Sahand Salamat, Mohsen Imani, Sarangh Gupta, and Tajana Rosing. «RNS-Snet: In-Memory Neural Network Acceleration Using Residue Number System». In: *2018 IEEE International Conference on Rebooting Computing (ICRC)*. 2018, pp. 1–12. DOI: 10.1109/ICRC.2018.8638592 (cit. on p. 1).

[8] P Athira, N R Deepa, Nimmy M Philip, and E G Anoop. «Modular Adder Designs Based On Thermometer Coding And One-Hot Coding». In: *2021 2nd International Conference on Advances in Computing, Communication, Embedded and Secure Systems (ACCESS)*. 2021, pp. 343–347. DOI: 10.1109/ACCESS51619.2021.9563300 (cit. on p. 1).

[9] Zabihollah Ahmadpour and Ghassem Jaberipur. «Up to 8k bit Modular Montgomery Multiplication in Residue Number Systems With Fast 16-bit Residue Channels». In: *IEEE Transactions on Computers* 71.6 (2022), pp. 1399–1410. DOI: 10.1109/TC.2021.3086071 (cit. on p. 1).

[10] Amos Omundi and Benjamin Premkumar. *Residue Number System: Theory and Implementation*. Jan. 2007 (cit. on p. 6).

[11] Shaoqiang Bi and Warren J. Gross. «The Mixed-Radix Chinese Remainder Theorem and Its Applications to Residue Comparison». In: *IEEE Transactions on Computers* 57.12 (2008), pp. 1624–1632. DOI: 10.1109/TC.2008.126 (cit. on pp. 6, 7, 11, 57–59).

[12] Zhongde Wang, G.A. Jullien, and W.C. Miller. «An improved residue-to-binary converter». In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 47.9 (2000), pp. 1437–1440. DOI: 10.1109/81.883343 (cit. on p. 7).

[13] Nikolay Chervyakov, Mikhail Babenko, Andrei Tchernykh, Anton Nazarov, and Anastasiia Garianina. «The Fast Algorithm for Number Comparing in Three-Modular RNS». In: *2016 International Conference on Engineering and Telecommunication (EnT)*. 2016, pp. 26–28. DOI: 10.1109/EnT.2016.014 (cit. on p. 11).

[14] Mikhail Babenko, Maxim Deryabin, Stanislaw J. Piestrak, Piotr Patronik, Nikolay Chervyakov, Andrei Tchernykh, and Arutyun Avetisyan. «RNS Number Comparator Based on a Modified Diagonal Function». In: *Electronics* 9.11 (2020). DOI: 10.3390/electronics9111784 (cit. on p. 11).

[15] Z. Torabi, G. Jaberipur, and S. A. Mirnaseri. «RNS Comparison via Shortcut Mixed Radix Conversion: The Case of Three 4 Moduli Sets». In: *IETE Journal of Research* 0.0 (2021), pp. 1–7. DOI: 10.1080/03772063.2021.1902865 (cit. on p. 11).

[16] Sachin Kumar and Chip-Hong Chang. «A VLSI-efficient signed magnitude comparator for { 2n-1, 2n, 2n +2n+1-1} RNS». In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016, pp. 1966–1969. DOI: 10.1109/ISCAS.2016.7538960 (cit. on p. 11).

[17] Lei Li, Guodong Li, Yingxu Zhao, Pengsheng Yin, and Wanting Zhou. «High Speed Comparator for the Moduli 2n, 2n-1, 2n+1». In: *IEICE Electronics Express* 10 (Nov. 2013), pp. 20130628–20130628. DOI: 10.1587/elex.10.20130628 (cit. on pp. 11, 16, 56–60).

[18] Shiva Taghipour Eivazi, Mehdi Hosseinzadeh, and Omid Mirmotahari. «Fully parallel comparator for the moduli set {2n, 2n-1, 2n+1}». In: *IEICE Electron. Express* 8 (2011), pp. 897–901 (cit. on pp. 11, 57–60).

[19] Sachin Kumar and Chip-Hong Chang. «A Scaling-Assisted Signed Integer Comparator for the Balanced Five-Moduli Set RNS {2{n}-1, 2{n}, 2{n}+ 1, 2{n+1}- 1, 2{n-1}- 1}». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25 (2017), pp. 3521–3533 (cit. on p. 11).

[20] Leonel Sousa and Paulo Martins. «Sign Detection and Number Comparison on RNS 3-Moduli Sets

$$\{2^n - 1, 2^{n+x}, 2^n + 1\}$$

2 n - 1 , 2 n + x , 2 n + 1 ». In: *Circuits, Systems, and Signal Processing* 36 (June 2016). DOI: 10.1007/s00034-016-0354-z (cit. on pp. 11, 55, 56, 58, 59).

[21] Vitor Krasnobayev, Alexandr Kuznetsov, Mihael Zub, and Kateryna Kuznetsova. «Methods for Comparing Numbers in Non-Positional Notation of Residual Classes». In: *CMIS*. 2019 (cit. on p. 13).

[22] Minghe Xu, Zhenpeng Bian, and Ruohe Yao. «Fast Sign Detection Algorithm for the RNS Moduli Set 2n+1-1, 2n-1, 2n». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23 (Jan. 2014), pp. 1–1. DOI: 10.1109/TVLSI.2014.2308014 (cit. on p. 16).