

# Chapter 2

## RNS-Based Embedded Processor Design

Pedro Miguens Matutino, Ricardo Chaves, and Leonel Sousa

### 2.1 Introduction

Digital consumer electronics represents a major sector of today's world economy in a wide range of products. Digital Signal Processors (DSPs) are a key component of many digital consumer products in several application domains, such as telecommunications, digital audio, video and imaging, speech processing, cryptography, and multimedia [1–9]. The design of DSPs has evolved rapidly over the last decade, driven by the ever-increasing need to improve performance and balance power consumption, flexibility, and integration of more features.

Conventional carry propagation arithmetic, based on a weighted number system, is a widely employed and well-studied approach. However, the dependencies and the need to perform the full weighted propagation of the carry cause a significant delay in arithmetic computation, preventing the design of arithmetic units with improved performance and enhanced efficiency. Therefore, Residue Number System (RNS) has been proposed as an alternative arithmetic system for computational intensive applications. RNS is a non-weighted numbering system that uses the remainders of the division by co-prime moduli, which compose a moduli set, to represent an integer value. The multiple and smaller values used in the RNS representation allow parallelism, high-speed, and low energy computation by reducing the hardware requirements to process data. The higher parallelism results from the fact that the multiplications and additions are performed independently on each individual residue channel.

---

P.M. Matutino (✉)

ISEL - IPL, INESC-ID, Rua Alves Redol 9, Lisboa, Portugal

e-mail: [pmmm@sips.inesc-id.pt](mailto:pmmm@sips.inesc-id.pt)

R. Chaves • L. Sousa

IST - Universidade de Lisboa, INESC-ID, Rua Alves Redol 9, Lisboa, Portugal

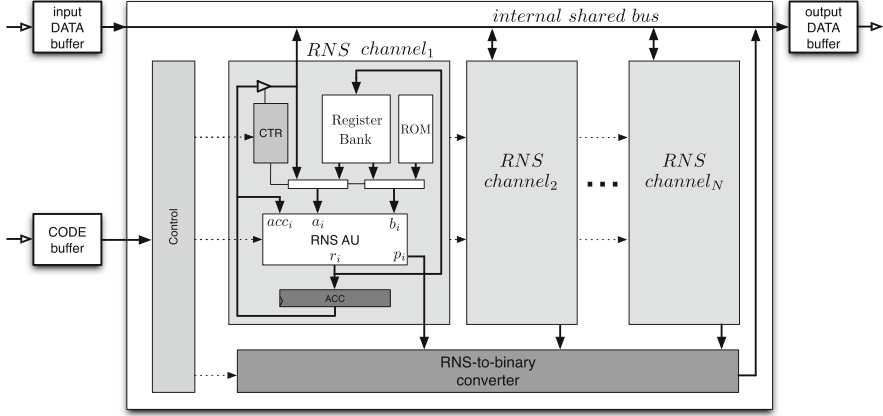
e-mail: [ricardo.chaves@inesc-id.pt](mailto:ricardo.chaves@inesc-id.pt); [las@inesc-id.pt](mailto:las@inesc-id.pt)

In order to facilitate the usage of RNS, its adaptability, and widen its applicability, a generic, efficient, and scalable RNS architecture supporting moduli sets with an arbitrary number of channels is required. In this chapter, a unified structure for a scalable RNS processor based on  $\{2^n \pm k_i\}$  ( $k_i \leq 2^{n/2} - 1$ ) moduli channels is proposed, allowing the design of RNS with any moduli set of the form  $\{2^n \pm k_0, \dots, 2^n \pm k_j\}$ , where  $j \in \mathbb{N}_0^+$ . The considered moduli set allows to arbitrarily increase the number of RNS channels and consequently, increase the Dynamic Range (DR) or reduce the width of the channels leading to a reduction in delay and area cost. The proposed RNS architecture provides a complete processing system supporting the computation of conversions, namely from binary-to-RNS, RNS-to-binary, and base extension. The proposed architecture has the capability of computing the conversion by reusing the arithmetic units of each channel, thus allowing the design of a more compact RNS processor. The arithmetic operations supported at the channel level include the addition, subtraction, and multiplication, with accumulation capability. For the reverse conversion two algorithms are considered: one based on the Chinese Remainder Theorem (CRT) and another on the Mixed Radix Converter (MRC). An Instruction Set Architecture (ISA) is also proposed, in order to provide a simple and independent interface to the proposed generic RNS architecture. The proposed RNS architecture and resulting processor implement the ISA without the need of explicitly defining the number of channels, allowing to generate generic code independently of the DR, making this transparent to the programmer.

This chapter is organized as follows. Section 2.2 presents the proposed RNS architecture, followed by the definition of the ISA in Sect. 2.3. The arithmetic operations including the conversions to and from binary are described in Sect. 2.4. Section 2.5 details the control units of the proposed processor. Section 2.6 presents a performance comparison with the most relevant state of the art, and Sect. 2.7 summarizes the main conclusions regarding the work herein proposed.

## 2.2 Processor Architecture

The RNS processor herein described is to be used as a coprocessor of generic Central Processing Units (CPUs), targeting intensive arithmetic computations. The first step of an RNS computation process is the conversion from binary-to-RNS, followed by the arithmetic operations performed in each channel, and finally the conversion from RNS-to-binary. Consequently, one of the main tasks of the RNS processor is to receive code and data (typical in binary) from the CPU and send it back after processing. Towards a more compact structure the approach herein presented considers that the number of required conversions is less than the number of arithmetic operations performed in the channels, which is typical of several applications, such as cryptographic operations. Furthermore, this allows for more compact structures since conversion steps can be executed in the hardware resources of the modular channels. Consequently, these conversions are executed in a sequential manner.



**Fig. 2.1** Proposed RNS architecture

The proposed architecture is organized in three main blocks, depicted in Fig. 2.1: (1) channel arithmetic blocks; (2) RNS-to-binary converter; and (3) global control unit. The channel arithmetic blocks perform the modular additions, subtractions, and multiplications on each channel. These were the chosen modular operations, since they are the basic arithmetic operations required in digital signal processing [3, 6, 7, 10, 11] and in applications, such as in asymmetrical cryptography [8, 9]. These arithmetic blocks are also used to perform the binary-to-RNS conversion.

The conversion from RNS-to-binary is herein performed in a two step algorithm. The first step converts the RNS representation into a Mixed Radix System (MRS) representation, followed by a second step that converts the MRS values into the binary value. The MRS conversion is fully computed in the arithmetic moduli channels. The second step of the reverse conversion, which cannot be performed using just the arithmetic operations available in the RNS channels, is computed on the RNS-to-binary converter module, containing the additional logic required for the conversion.

The MRS operation is herein also used to perform base extensions, implementing the conversion between moduli sets. The base extension allows to convert number representation between two moduli sets without performing the conversion from RNS-to-binary in the origin moduli set, and the conversion from binary-to-RNS in the destination. The base extension operations are intensively used in the Montgomery Modular Multiplication [Montgomery Multiplication (MM)] [12, 13], one of the main operations when computing asymmetrical cryptographic algorithms [14] on RNS. These base extension operations require more than one moduli set to be computed, although each RNS processor only has one moduli set, consequently the proposed RNS computation requires more than one processor. Given this, the proposed RNS architecture allows multiple RNS processors in the RNS computing system.

Considering that the RNS architecture is to be used as a coprocessor of generic CPUs, the communication between them can be established by First In First Out (FIFO) buffers, implementing the communication and data flow control, also acting as synchronization points.

The following sections describe the considered ISA and the proposed architecture, detailing the designed computation logic used to perform the binary-to-RNS conversion, the modular arithmetic operations, and RNS-to-binary conversion.

## 2.3 Instruction Set Architecture

In order to provide a simple and independent interface to the proposed RNS architecture, an ISA is defined. This ISA considers a scalable RNS architecture, without the need to explicitly define the number of channels of the implemented RNS and resulting DR. This allows to generate generic code, making it transparent to the programmer. Apart from the conversion operations, the programmer can code as for a typical arithmetic binary processor, without the need to know any details of RNS arithmetic. The defined ISA supports the following arithmetic operations: addition, subtraction, and multiplication, with and without accumulation capability, as further described in Sect. 2.4.2. Alongside these arithmetic operations, conversion operations to and from binary, and base extension, are also considered and further detailed in Sects. 2.4.1 and 2.4.3. Since these conversion operations are implemented by arithmetic computations, the number of these computations and consequently the number of clocks cycles are directly dependent on the number of channels. The conversion computation requires specific constants, which depend on the used moduli set and do not change for the entire life-cycle of the RNS processor. If these are defined programmatically and sent by the CPU to the coprocessor, overhead is introduced due to data transfers. Considering this, and in order to maintain a simple interface, these constants and the operation micro-code are built-in into the RNS processor. With this, the computation of the conversions to and from binary, and possible base extension, can be performed without external support. As stated before, the conversion operations are implemented in a sequential manner, requiring more than one cycle to be completed. Given this, the RNS processor instructions are divided into two main categories: namely, single cycle and multi-cycle instructions. The first one requires a single clock cycle to complete the arithmetic computation, independently of the number of RNS channels, while the multi-cycle instructions directly dependent on the number of channels, imposing a number of cycles equal to the number of channels.

Herein, a 32-bit instruction size is considered, since it is the most common length used nowadays on embedded CPUs [15]. The proposed instruction format is divided in four fields, namely: (1) register source one (*rs1*); (2) register source two (*rs2*); (3) destination register (*rd*); and (4) operation code (*opcode*), as depicted in Fig. 2.2. Each register field has a length of 8 bits, allowing to address up to 256 registers. The operation code field is divided into two sub-fields: the first one used for RNS processor enabling, and the second one to define the arithmetic operation.

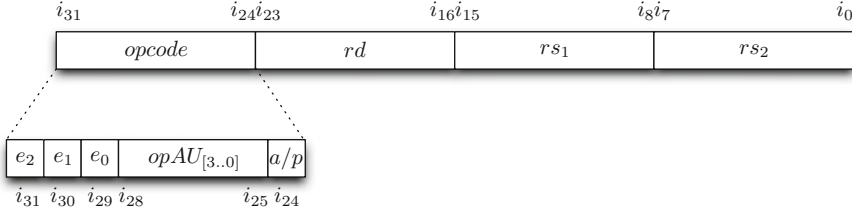


Fig. 2.2 Instruction format of the proposed RNS processor

Table 2.1 RNS processor ISA

#	mnemonic	instruction											
		$i_{31}$	$i_{30}$	$i_{29}$	$i_{28}$	$i_{27}$	$i_{26}$	$i_{25}$	$i_{24}$	$i_{23} \cdots i_{16}$	$i_{15} \cdots i_8$	$i_7 \cdots i_0$	
1	add/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	0	0	0	0	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
2	sub/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	0	0	0	1	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
3	mul/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	0	0	1	0	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
4	aadd/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	1	0	0	0	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
5	asub/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	1	0	0	1	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
6	amul/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	1	0	1	0	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
7	sadd/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	1	1	0	0	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
8	ssub/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	1	1	0	1	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
9	smul/a rd,rs1,rs2	$e_2$	$e_1$	$e_0$	1	1	1	0	$a$	$rd_{[7:0]}$	$rs1_{[7:0]}$	$rs2_{[7:0]}$	
10	bToRNS rd	$e_2$	$e_1$	$e_0$	0	1	0	0	$p$	$rd_{[7:0]}$			
11	mToRNS rd	$e_2$	$e_1$	$e_0$	0	1	0	1	$p$	$rd_{[7:0]}$			
12	rsToM rd,rs1	$e_2$	$e_1$	$e_0$	0	1	1	0	$p$	$rd_{[7:0]}$	$rs1_{[7:0]}$		
13	mToBIN rs1	$e_2$	$e_1$	$e_0$	0	1	1	1	0		$rs1_{[7:0]}$		

The RNS architecture allows multiple RNS processors in the RNS computing system using processor enabling flags, up to three are allowed with this codification. This maximum value is considered given the typical RNS based cryptographic implementations [16]. The arithmetic operations ( $opAU$ ) are specified by four bits, with an additional bit to enable accumulation, as depicted in Fig. 2.2 and detailed in Table 2.1. This additional bit is also used to define if the internal shared bus propagates the data, used in multi-cycle operations such as the base extension, allowing for data transfers between two RNS processors.

Table 2.1 details the proposed ISA, divided into single and multi-cycle instructions. The first three operations (1–3) implement the modular addition, subtraction, and multiplication between  $rs1$  and  $rs2$ , storing the result in the defined  $rd$ , and in the accumulator if the accumulator flag  $a$  is active. The following three operations (4–6) preceded by ‘ $a$ ’ execute the same operations as the previous ones, with the particularity that the accumulator value is also added. Once more the final value is stored in the destination register, and in the accumulator if the flag  $a$  is set. The last

three single cycle instructions (7–9), preceded by ‘s’, have the same behaviour as the previous ones, but instead of adding the result of the operation between  $rs1$  and  $rs2$ , subtracts that value from the accumulator. In these instructions, the flag  $a$  has the same functionality, i.e., besides the destination register, the result of the operation is also stored in the accumulator if  $a$  is set.

The first two multi-cycle instructions (10–11) convert from binary and from a mixed-radix representation, respectively, to the moduli set in the RNS processor. These instructions receive data from the shared bus and require only the definition of the destination register. The third multi-cycle instruction (*mToRNS rd,rs1*) executes the conversion from RNS-to-MRS, requiring a source register and a destination register. This instruction is described in detail in Sect. 2.4.3. Note that this conversion changes the representation from RNS to a Mixed Radix System. The instruction *mToBin* (MRS-to-binary) is used to perform the second step of the conversion from RNS-to-binary, that was started by a RNS-to-MRS instruction. Note that, herein the RNS-to-binary conversion is considered as a two step operation, as further described in Sect. 2.4.3. The MRS-to-binary instruction only requires the definition of  $rs1$ , without any destination register, since the computed values are written in output data buffer.

In order to support this ISA, each RNS channel is organized in four main blocks, as depicted in Fig. 2.1, namely: (1) register bank and accumulator, for data storage; (2) constant memory Read-Only-Memory (ROM); (3) RNS Arithmetic Unit (AU), where the modular arithmetic operations are executed; and (4) control (CTR).

The data information flow from and to the CPU defines the behaviour of the global control unit. Remember that the RNS architecture is to be used as a coprocessor of generic CPUs, and one of its main tasks is to receive code and data from the CPU and send it back after processing. In order to simplify the communication and data flow control, buffers are herein used as synchronization points. With this, the RNS processor stalls whenever data or code to be processed are missing. Furthermore, when the output data buffers are full the RNS processor also stalls, avoiding in this way losing processed data. In order to implement this data control flow, one unidirectional and one bidirectional FIFO buffers are herein considered for code and data, respectively, as depicted in Fig. 2.1.

## 2.4 RNS Arithmetic Operations

This section describes the formulation and resulting modular hardware structures required to compute the binary-to-RNS conversion, the modular arithmetic operations in each moduli channel, the RNS-to-binary conversion, and base extension operation.

### 2.4.1 Binary-to-RNS Conversion

The first typical operation in an RNS processor is the conversion to the RNS representation. This is achieved by computing the remainder of the division of each operand by  $\{2^n \pm k_i\}$ . The residue modulo  $\{2^n \pm k_i\}$  of the input value  $X$  can be achieved by computing the integer division of  $X$  by  $\{2^n \pm k_i\}$ . However, obtaining the remainder in this way is a costly operation. Nevertheless, this operation can be implemented considering only modular addition operations, such that the residue  $x_i$  of an integer  $X$  with  $(N \cdot n)$ -bit inputs can be calculated as:

$$x_i = \langle X \rangle_{2^n \pm k_i} = \left\langle \sum_{b=0}^{N \cdot n - 1} \langle 2^b \cdot X_{[b]} \rangle_{2^n \pm k_i} \right\rangle_{2^n \pm k_i}, \quad (2.1)$$

where  $X_{[b]}$  represents the  $b$ th-bit of the binary integer  $X$ , and  $\langle X \rangle_{2^n \pm k_i}$  represents the residue of  $X$  modulo  $\{2^n \pm k_i\}$ .

Given Eq. (2.1), the conversion of a binary number  $X$  with  $(N \cdot n)$  bits to RNS for modulo  $\{2^n - k\}$  can be computed as:

$$\langle X \rangle_{2^n - k} = \left\langle \sum_{i=0}^{N-1} k^i X_{[(i+1) \cdot n - 1 : i \cdot n]} \right\rangle_{2^n - k}. \quad (2.2)$$

where  $X_{[b:c]}$  represents the bit vector from  $b$ th-bit to  $c$ th-bit. Similarly to modulo  $\{2^n - k\}$ , the binary-to-RNS conversion modulo  $\{2^n + k\}$  can be computed as:

$$\begin{aligned} \langle X \rangle_{2^n + k} = & \left\langle \sum_{i=0}^{\lfloor \frac{N-1}{2} \rfloor} k^{2i} X_{[(2i+1) \cdot n - 1 : 2i \cdot n]} + \sum_{i=0}^{\lfloor \frac{N-2}{2} \rfloor} k^{2i+1} \overline{X}_{[(2i+2) \cdot n - 1 : (2i+1) \cdot n]} + \right. \\ & \left. + \sum_{i=0}^{\lfloor \frac{N-2}{2} \rfloor} k^{2i+1} (k + 1) \right\rangle_{2^n + k}. \end{aligned} \quad (2.3)$$

Note that  $\overline{X}$  represents the not bitwise operation, and the last term is a constant value that can be pre-calculated.

### 2.4.2 Arithmetic Channels

Arithmetic operations in Residue Number System are one of the most important aspects to take into account when optimizing computations. An RNS is composed of several smaller arithmetic channels. Typically, the modular arithmetic structures, implementing these channels, are more complex than the binary equivalent with the

same bit width. A suitable implementation of these arithmetic structures can lead to overall improvements, compensating the conversion overheads. Even though most of the state of the art still focus on modulo channels of the form  $\{2^n \pm 1\}$ , given their simplicity, the use of moduli sets with modulo  $\{2^n \pm k\}$  channels, with unrestricted  $k$  values, can be rather useful in the definition of RNS with larger moduli sets. With this, arithmetic systems with better performances can be obtained, given that the operands in each channel require a smaller number of bits.

The following presents the formulation and resulting structures used in the arithmetic channels performing the required addition, subtraction, and multiplication, with and without accumulation, for modulo  $\{2^n - k\}$  and  $\{2^n + k\}$ .

#### 2.4.2.1 Modulo $\{2^n - k\}$

The addition modulo  $\{2^n - k\}$  of two residue values, can be easily formulated as:

$$\langle a + b \rangle_{2^n - k} = \begin{cases} a + b & , a + b < 2^n - k \\ a + b - (2^n - k) & , a + b \geq 2^n - k \end{cases} . \quad (2.4)$$

For the subtraction of two residue values:

$$\begin{aligned} \langle -b_{[n-1:0]} \rangle_{2^n - k} &= \langle 2^n - 1 - b_{[n-1:0]} + 1 - k \rangle_{2^n - k} \\ &= \langle \overline{b_{[n-1:0]}} + 1 - k \rangle_{2^n - k} . \end{aligned} \quad (2.5)$$

Therefore:

$$\begin{aligned} \langle a - b \rangle_{2^n - k} &= \langle a - b \rangle_{2^n - k} \\ &= \langle a_{[n-1:0]} + \overline{b_{[n-1:0]}} + 1 - k \rangle_{2^n - k} . \end{aligned} \quad (2.6)$$

The subtraction between the residue  $a$  and  $b$  with accumulation can be formulated as:

$$\begin{aligned} \text{acc}_{q+1} &= \langle \text{acc}_q + a - b \rangle_{2^n - k} \\ &= \langle \text{acc}_{q[n-1:0]} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + 1 - k \rangle_{2^n - k} . \end{aligned} \quad (2.7)$$

Identically, the subtraction of  $(a + b)$  from the accumulated value is given by:

$$\begin{aligned} \text{acc}_{q+1} &= \langle \text{acc}_q - (a + b) \rangle_{2^n - k} \\ &= \langle \text{acc}_{q[n-1:0]} + \overline{a_{[n-1:0]}} + \overline{b_{[n-1:0]}} + 2 \cdot (1 - k) \rangle_{2^n - k} . \end{aligned} \quad (2.8)$$

The multiplication of the residue  $a$  by  $b$ , with positive accumulation, can be formulated as:

$$\begin{aligned}
 \text{acc}_{q+1} &= \langle \text{acc}_q + a \times b \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} + p_{[2n-1:0]} \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} + 2^n \cdot p_{[2n-1:n]} + p_{[n-1:0]} \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} + (2^n - k + k) \cdot p_{[2n-1:n]} + p_{[n-1:0]} \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} + k \cdot p_{[2n-1:n]} + p_{[n-1:0]} \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} + m_{[n+w_k-1:0]}^1 + p_{[n-1:0]} \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} + k \cdot m_{[n+w_k-1:n]}^1 + m_{[n-1:0]}^1 + p_{[n-1:0]} \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} + m_{[2w_k-1:0]}^2 + m_{[n-1:0]}^1 + p_{[n-1:0]} \rangle_{2^n-k} . \quad (2.9)
 \end{aligned}$$

Note that the width of  $k$  is represented by  $w_k = \lceil \log_2(k) \rceil \leq n/2$  bit, in order to use the arithmetic units proposed in [17, 18], and the value  $p_{[2n-1:0]}$  results from the binary multiplication of  $a \times b$ , that can be computed by an  $n \times n$ -bit binary multiplier. Consequently, the  $m_{[n+w_k-1:0]}^1$  represents the result of the multiplication of the  $k$  constant by  $p_{[2n-1:n]}$ , and similarly the  $m_{[2w_k-1:0]}^2$  represents the constant multiplication of the constant  $k$  by  $m_{[n-1:0]}^1$ .

The same multiplication but with negative accumulation can be obtained by computing:

$$\begin{aligned}
 \text{acc}_{q+1} &= \langle \text{acc}_q - a \times b \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} - (m_{[2w_k-1:0]}^2 + m_{[n-1:0]}^1 + p_{[n-1:0]}) \rangle_{2^n-k} \\
 &= \langle \text{acc}_{q[n-1:0]} + \overline{m_{[2w_k-1:0]}^2} + \overline{m_{[n-1:0]}^1} + \overline{p_{[n-1:0]}} + 3 \cdot (1 - k) \rangle_{2^n-k} . \quad (2.10)
 \end{aligned}$$

#### 2.4.2.2 Modulo $\{2^n + k\}$

Similarly to modulo  $\{2^n - k\}$ , the addition modulo  $\{2^n + k\}$  of two residue values can be formulated as:

$$\langle a + b \rangle_{2^n+k} = \begin{cases} a + b & , a + b < 2^n + k \\ a + b - (2^n + k) & , a + b \geq 2^n + k \end{cases} . \quad (2.11)$$

Once more to formulate the subtraction of two residue values, let us start by deriving the symmetric of a residue as:

$$\begin{aligned}
 \langle -b_{[n:0]} \rangle_{2^n+k} &= \langle -2^n \cdot b_{[n]} - b_{[n-1:0]} \rangle_{2^n+k} \\
 &= \langle -(2^n + k - k) \cdot b_{[n]} + 2^n - 1 - b_{[n-1:0]} + 1 + k \rangle_{2^n+k} \\
 &= \langle k \cdot b_{[n]} + \overline{b_{[n-1:0]}} + 1 + k \rangle_{2^n+k} .
 \end{aligned} \tag{2.12}$$

Thus, the subtraction between the residue  $a$  and  $b$  can be described as:

$$\begin{aligned}
 \langle a - b \rangle_{2^n+k} &= \langle a - b \rangle_{2^n+k} \\
 &= \langle a_{[n-1:0]} + \overline{b_{[n-1:0]}} + 1 + k - k \cdot a_{[n]} + k \cdot b_{[n]} \rangle_{2^n+k} \\
 &= \langle a_{[n-1:0]} + \overline{b_{[n-1:0]}} + 1 + k \cdot (1 - a_{[n]} + b_{[n]}) \rangle_{2^n+k} .
 \end{aligned} \tag{2.13}$$

The subtraction operations, with accumulation, for channels modulo  $\{2^n + k\}$  can be formulated as:

$$\begin{aligned}
 \text{acc}_{q+1} &= \langle \text{acc}_q + a - b \rangle_{2^n+k} \\
 &= \langle \text{acc}_{q[n:0]} + a_{[n:0]} - b_{[n:0]} \rangle_{2^n+k} \\
 &= \langle 2^n \cdot (\text{acc}_{q[n]} + a_{[n]} - b_{[n]}) + \text{acc}_{q[n-1:0]} + a_{[n-1:0]} - b_{[n-1:0]} \rangle_{2^n+k} \\
 &= \langle (2^n + k - k) \cdot (\text{acc}_{q[n]} + a_{[n]} - b_{[n]}) + \text{acc}_{q[n-1:0]} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + \\
 &\quad + 1 + k \rangle_{2^n+k} \\
 &= \langle (-k) \cdot (\text{acc}_{q[n]} + a_{[n]} - b_{[n]}) + \text{acc}_{q[n-1:0]} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + \\
 &\quad + 1 + k \rangle_{2^n+k} \\
 &= \langle \text{acc}_{q[n-1:0]} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + k \cdot (-\text{acc}_{q[n]} - a_{[n]} + b_{[n]}) + \\
 &\quad + k + 1 \rangle_{2^n+k} \\
 &= \langle \text{acc}_{q[n-1:0]} + a_{[n-1:0]} + \overline{b_{[n-1:0]}} + \\
 &\quad + (k \cdot (1 + b_{[n]} - \text{acc}_{q[n]} - a_{[n]}) + 1) \rangle_{2^n+k} .
 \end{aligned} \tag{2.14}$$

Note that the last term is one of the eight possible constants dependent on  $a_{[n]}, b_{[n]}$ , and  $\text{acc}_{q[n]}$ . These eight values can be pre-calculated.

Using the same approach, the subtraction of  $(a + b)$  with accumulation can be derived as:

$$\begin{aligned}
 \text{acc}_{q+1} &= \langle \text{acc}_q - (a + b) \rangle_{2^n+k} \\
 &= \langle \text{acc}_{q[n:0]} - a_{[n:0]} - b_{[n:0]} \rangle_{2^n+k}
 \end{aligned}$$

$$\begin{aligned}
&= \langle 2^n \cdot (\text{acc}_{q[n]} - a_{[n]} - b_{[n]}) + \text{acc}_{q[n-1:0]} - a_{[n-1:0]} - b_{[n-1:0]} \rangle_{2^n+k} \\
&= \langle -k \cdot (\text{acc}_{q[n]} - a_{[n]} - b_{[n]}) + \text{acc}_{q[n-1:0]} + \overline{a_{[n-1:0]}} + 1 + k + \overline{b_{[n-1:0]}} + \\
&\quad + 1 + k \rangle_{2^n+k} \\
&= \langle k \cdot (-\text{acc}_{q[n]} + a_{[n]} + b_{[n]}) + \text{acc}_{q[n-1:0]} + \overline{a_{[n-1:0]}} + \overline{b_{[n-1:0]}} + \\
&\quad + 2 + 2k \rangle_{2^n+k} \\
&= \langle \text{acc}_{q[n-1:0]} + \overline{a_{[n-1:0]}} + \overline{b_{[n-1:0]}} + \\
&\quad + (2 + k \cdot (2 + b_{[n]} - \text{acc}_{q[n]} + a_{[n]})) \rangle_{2^n+k} .
\end{aligned} \tag{2.15}$$

In a similar manner to the multiplication modulo  $\{2^n - k\}$  depicted in (2.9), the multiplication of  $a$  by  $b$ , with positive accumulation modulo  $\{2^n + k\}$ , can be formulated as:

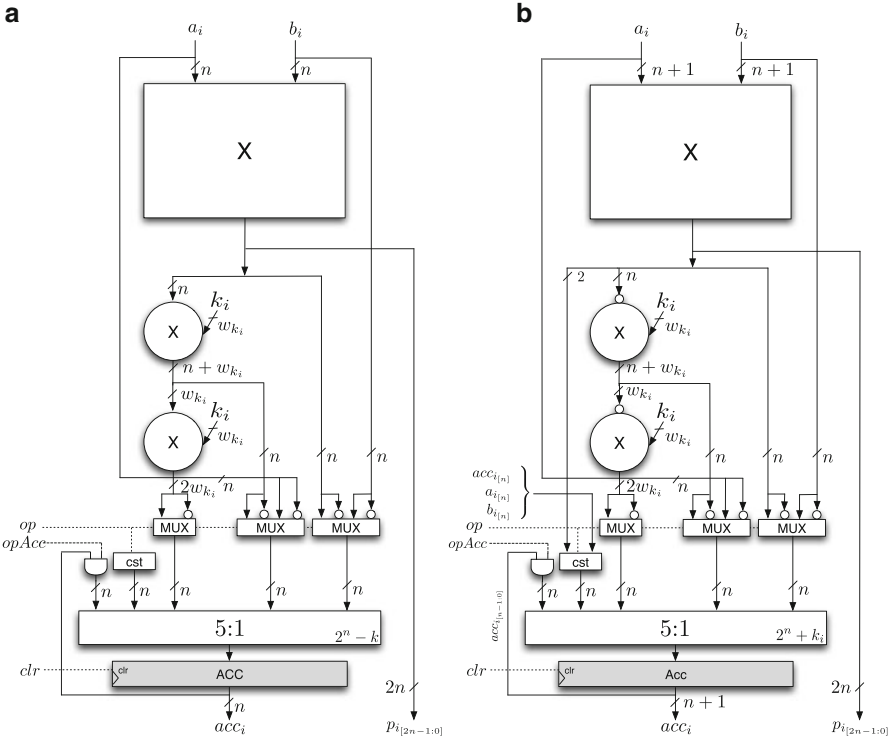
$$\begin{aligned}
\text{acc}_{q+1} &= \langle \text{acc}_q + a \times b \rangle_{2^n+k} \\
&= \langle \text{acc}_{q[n:0]} + a_{[n:0]} \times b_{[n:0]} \rangle_{2^n+k} \\
&= \langle \text{acc}_{q[n:0]} + p_{[2n+1:0]} \rangle_{2^n+k} \\
&= \langle \text{acc}_{q[n:0]} + 2^{2n} \cdot p_{[2n+1:2n]} + 2^n \cdot p_{[2n-1:n]} + p_{[n-1:0]} \rangle_{2^n+k} \\
&= \langle \text{acc}_{q[n:0]} + k^2 \cdot p_{[2n+1:2n]} - k \cdot p_{[2n-1:n]} + p_{[n-1:0]} \rangle_{2^n+k} \\
&= \langle 2^n \cdot \text{acc}_{q[n]} + \text{acc}_{q[n-1:0]} + k^2 \cdot p_{[2n+1:2n]} + k \cdot \overline{m_{[n+w_k-1:n]}^1} + m_{[n-1:0]}^1 + \\
&\quad + p_{[n-1:0]} + (k + 1) \rangle_{2^n+k} \\
&= \langle \text{acc}_{q[n-1:0]} + m_{[2w_k-1:0]}^2 + m_{[n-1:0]}^1 + p_{[n-1:0]} + k^2 \cdot p_{[2n+1:2n]} + \\
&\quad + (k + 1) - k \cdot \text{acc}_{q[n]} \rangle_{2^n+k} .
\end{aligned} \tag{2.16}$$

Using the subtractive inverse to rewrite (2.16) with negative accumulation results in:

$$\begin{aligned}
\text{acc}_{q+1} &= \langle \text{acc}_q - a \times b \rangle_{2^n+k} \\
&= \langle \text{acc}_{q[n-1:0]} + \overline{m_{[2w_k-1:0]}^2} + \overline{m_{[n-1:0]}^1} + \\
&\quad + \overline{p_{[n-1:0]}} + (k + 1) - k^2 \cdot p_{[2n+1:2n]} - k \cdot \text{acc}_{q[n]} \rangle_{2^n+k} .
\end{aligned} \tag{2.17}$$

### 2.4.2.3 Proposed Arithmetic Structures

Given the above formulation, it is possible to derive a single structure capable of computing all these operations. This structure uses a binary multiplier to compute  $a_i \times b_i$ , two constant multipliers [for computing  $(k \cdot p_{[2n-1:n]})$  and  $(k \cdot m_{[n+w_k-1:n]}^1)$ ], and one 5:1 modular adder (for computing the addition of five input vectors to one output vector) to add all the resulting terms. This 5:1 modular adder can be implemented using one 3:2 modular compression and one 4:1 modular adder, proposed in [17]. The arithmetic structure for channels modulo  $\{2^n - k\}$  and  $\{2^n + k\}$  are similar. However, for channels modulo  $\{2^n + k\}$ , a  $((n + 1) \times (n + 1))$ -bit binary multiplier is used, instead of a  $(n \times n)$ -bit multiplier. For the  $\{2^n + k\}$  modulo, two constant multipliers are also used, but the constant block depends on more selection bits ( $acc_{i[n]}$ ,  $a_{i[n]}$ ,  $b_{i[n]}$ , and  $p_{[2n+1:2n]}$ ). The resulting structures are depicted in Fig. 2.3.



**Fig. 2.3** Channel structure. (a) Modulo  $\{2^n - k\}$ . (b) Modulo  $\{2^n + k\}$

### 2.4.3 RNS-to-Binary Conversion

Most reverse converters of the state of the art are based on the Chinese Remainder Theorem (CRT) [19], on the Mixed Radix Converter (MRC) [19], and on the more recent New CRT [20]. Herein, both the CRT and MRC algorithms are considered. The first approach, based on the CRT, results in a more parallel computation while the second one, based on the MRC, results in a more sequential approach but requiring simpler modular arithmetic operations. Both algorithms allow to reuse the modular arithmetic units of each channel to perform part of the conversion from RNS-to-binary. This allows to reduce the system's overall circuit area. Nevertheless, the CRT approach requires additional hardware to implement a modular adder modulo  $M$ , necessary for the computation of the final binary value. The New Chinese Remainder Theorem I (new-CRT-I) [20] and the New Chinese Remainder Theorem II (new-CRT-II) [20] are herein not considered. This is due to the fact that new-CRT-I requires modulo channels satisfying the condition  $m_i > 2m_{i-1}$ , that imposes unbalanced systems, and the new-CRT-II requires intermediate modular operations, which are not suitable to be implemented in the modular channels.

#### 2.4.3.1 Proposed CRT Approach

The computation of the binary result ( $X$ ) using the CRT algorithm, for  $N$  residues, can be described by [19]:

$$\begin{aligned} \langle X \rangle_M &= \left\langle \sum_{i=1}^N \left\langle x_i \cdot \langle M_i^{-1} \rangle_{m_i} \cdot M_i \right\rangle_M \right\rangle_M \\ &= \left\langle \sum_{i=1}^N x_i \cdot \langle M_i^{-1} \rangle_{m_i} \cdot M_i \right\rangle_M \\ &= \left\langle \sum_{i=1}^N x_i \cdot W_i \right\rangle_M, \end{aligned} \quad (2.18)$$

where  $W_i$  is the weight of the residue  $m_i$ , given by:

$$W_i = \langle M_i^{-1} \rangle_{m_i} \cdot M_i, \quad (2.19)$$

where  $M_i^{-1}$  represents the multiplicative inverse of  $M_i$ , such that  $M_i = M/m_i$  and  $\langle M_i^{-1} M_i \rangle_{m_i} = 1$ .

Given  $x_i$ , the  $n$ -bit remainder of the division of  $X$  by  $m_i$ , and  $W_i$ , a constant multiplication factor, with a dynamic range of  $(N \cdot n)$  bits:

$$x_i = \sum_{j=0}^{n-1} x_{i[j]} 2^j, \quad W_i = \sum_{j=0}^{N \cdot n - 1} W_{i[j]} 2^j, \quad (2.20)$$

the product  $x_i \cdot W_i$  can be computed by decomposing the operand  $W_i$  into blocks of  $n$  bits, from most to least significant bits, where  $W_{i_l}$  represents the  $[l \cdot (n + 1) - 1 : l \cdot n]$  bits of  $W_i$ , given by:

$$\begin{aligned} W_i &= (2^n)^{N-1} \cdot \sum_{j=(N \cdot n)-n}^{N \cdot n-1} W_{i_{[j]}} 2^{j-(N \cdot n)+n} + \dots + \sum_{j=0}^{n-1} W_{i_{[j]}} 2^j \\ &= 2^{(N-1) \cdot n} \cdot W_{i_{(N-1)}} + \dots + 2^n \cdot W_{i_1} + W_{i_0} . \end{aligned} \quad (2.21)$$

Applying this decomposition to the modular multiplication in (2.18), the computation of  $X$  is given by:

$$\begin{aligned} \langle X \rangle_M &= \left\langle \sum_{i=1}^N x_i \cdot (2^{(N-1) \cdot n} \cdot W_{i_{(N-1)}} + \dots + 2^n \cdot W_{i_1} + W_{i_0}) \right\rangle_M \\ &= \left\langle 2^{(N-1) \cdot n} \cdot \sum_{i=1}^N x_i \cdot W_{i_{(N-1)}} + \dots + \sum_{i=1}^N x_i \cdot W_{i_0} \right\rangle_M . \end{aligned} \quad (2.22)$$

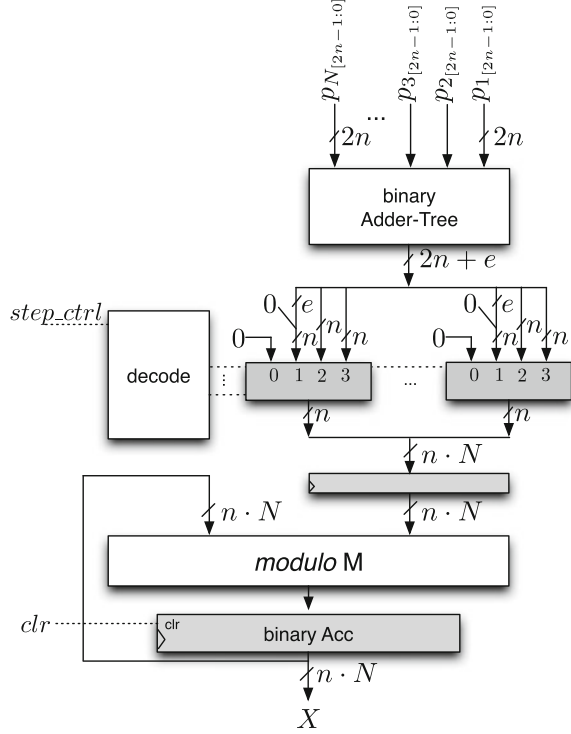
In the proposed RNS architecture, this computation can be performed by the multiplication units of each RNS channel, computing  $x_i W_i$ , followed by a final dedicated adder-tree modulo  $M$ . Note that the complexity of the final modular adder increases with the growth of the DR. To compute (2.22),  $N$  steps are required to perform all the constant multiplications. On each iteration the resulting constant multiplication values are added on a binary adder-tree, compressing the  $N$  input values, with  $2n$ -bit length, into one vector of  $2n + e$  bits, as depicted in Fig. 2.4. Note that  $e = \lceil \log_2(N) \rceil$  represents the extra bits resulting from the binary addition tree. This result is then shifted to compute the multiplication by the values  $2^n$  to  $2^{(N-1)n}$ , depending on the iteration. The selection of the values to be used is performed using  $N$  4:1 multiplexers. These multiplexers split the  $2n + e$  bits vector into two groups of  $n$  bits and one of  $e$  bits, and each group is then selected by the step decode circuit, as depicted in Fig. 2.4. The shifted value is fed into the  $m$ -bit modulo  $M$  adder to compute the binary value of  $X$ . After these  $N$  steps, an additional step is required to perform the last modulo  $M$  reduction, totalling  $N + 1$  steps to compute the conversion of  $X$  from RNS into its binary representation, using the CRT conversion algorithm.

#### 2.4.3.2 Proposed MRC Approach

The other considered reverse conversion approach uses the MRC algorithm [19]. This approach starts by computing a mixed-radix representation from a residue representation. Considering the moduli set  $\{m_1, m_2, \dots, m_N\}$ , with  $N$  channels, and  $z_i$  the mixed-radix related to  $m_i$  ( $0 \leq z_i < m_i$ ),  $X$  can be computed as:

$$X = z_N \cdot m_{N-1} \cdot m_{N-2} \dots m_1 + \dots + z_2 \cdot m_1 + z_1 . \quad (2.23)$$

**Fig. 2.4** CRT based conversion block



The mixed-radix digits ( $z_i$ ) can be iteratively calculated by [19]:

$$z_i = \left\langle \left\langle (m_{i-1} \cdots m_1)^{-1} \right\rangle_{m_i} (x_i - \langle z_{i-1} \cdot m_{i-2} \cdots m_1 + \cdots + z_2 \cdot m_1 + z_1 \rangle_{m_i}) \right\rangle_{m_i}. \quad (2.24)$$

Note that this algorithm is sequential, since the computation of  $z_i$  is dependent of  $z_{i-1}$ .

This iterative process requires  $2N$  cycles to compute all the mixed-radix digits, positioning  $X$  in MRS representation. This can be performed using only the arithmetic channels of the proposed RNS structure. The needed iterations are illustrated in Table 2.2. All the multiplicative inverse values required in (2.24) can be pre-computed and stored in memory.

The final  $X$  value can be computed by multiplying the mixed-radix digits ( $z_i$ ) by a constant factor ( $W'_i$ ), as shown in (2.23). Considering  $z_i$  and  $W'_i$  as:

$$z_i = \sum_{j=0}^{n-1} z_{i[j]} 2^j, \quad W'_i = \prod_{j=1}^{i-1} m_j = \sum_{j=0}^{n \cdot (i-1)} W'_{i[j]} 2^j. \quad (2.25)$$

**Table 2.2** Mixed Radix Converter algorithm execution on the proposed RNS architecture

step	channel N	...	channel 3	channel 2	channel 1
0	$x_N$		$x_3$	$x_2$	$x_1$
1	$x_N - z_1$		$x_3 - z_1$	$x_2 - z_1$	$z_1$
2				$acc_2 \cdot \langle m_1^{-1} \rangle_{m_2}$	
3	$acc_N - z_2 \cdot m_1$		$acc_3 - z_2 \cdot m_1$	$z_2$	
4			$acc_3 \cdot \langle m_2 m_1^{-1} \rangle_{m_2}$		
5	$acc_N - z_3 \cdot m_2 m_1$		$z_3$		
...					
2N-1	$acc_N - z_{N-1} \cdot m_{N-2} \cdots m_1$				
2N	$acc_N \cdot \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N}$				
2N	$z_N$		$z_3$	$z_2$	$z_1$
2N+1	$z_N \cdot m_N \cdots m_2 m_1 [n-1:0]$		$z_3 \cdot m_2 m_1 [n-1:0]$	$z_2 \cdot m_1$	$z_1$
2N+2	$z_N \cdot m_N \cdots m_2 m_1 [2n-1:n]$		$z_3 \cdot m_2 m_1 [2n-1:n]$		
...					
3N-1	$z_N \cdot m_N \cdots m_2 m_1 [N-n-1:(N-1) \cdot n]$				

The multiplication of a digit  $z_i$  by its weight  $W'_i$  can be computed identically to CRT (2.22), consequently  $X$  can be computed as:

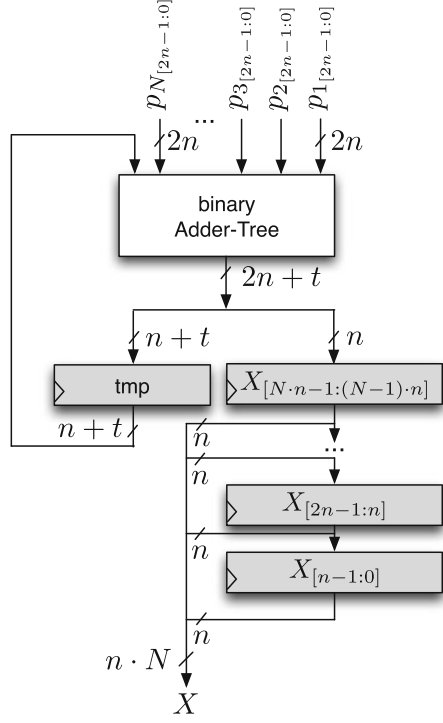
$$X = 2^{(N-1) \cdot n} \cdot \sum_{i=1}^N z_i \cdot W'_{i(N-1)} + \cdots + \sum_{i=1}^N z_i \cdot W'_{i_0}. \quad (2.26)$$

The final value  $X$  can thus be computed by a binary adder-tree, compressing the  $N + 1$  input values, with  $2n$  bits length, into one vector of  $2n + t$  bits, with  $t = \lceil \log_2(N + 1) \rceil$  representing the resulting additional bits. The binary adder-tree is fed with the results given by the binary multiplier output  $p_{i[2n-1:0]}$  of each arithmetic channel, as depicted in Figs. 2.3 and 2.5.

In each cycle, the  $n$  Least Significant Bits (LSB) of the binary adder-tree are stored into a register, and the previous value is shifted into another register. The computation of (2.26) requires  $N$  additional steps to conclude the computation of  $X$ . With this approach, the complete conversion from RNS-to-binary requires a total of  $3N$  cycles to compute  $X$ . The scheduling of operations used to perform the MRC conversion, based on the proposed RNS architecture, is depicted in Table 2.2.

The hardware overhead of this solution (Fig. 2.5) is reduced when compared with the CRT algorithm, which requires a final modulo  $M$  adder (Fig. 2.4). However, to compute  $X$  a total of  $3N$  cycles are required. Nevertheless, this computational

**Fig. 2.5** MRC based conversion block



cost can be optimized, if the last modular multiplication in the mixed-radix-digit computation is performed on each iteration, as given by:

$$\begin{aligned}
 z_i = & \left\langle x_i \left\langle (m_{i-1} \cdots m_1)^{-1} \right\rangle_{m_i} - z_{i-1} \cdot \left\langle (m_{i-1} \cdots m_1)^{-1} \cdot m_{i-2} \cdots m_1 \right\rangle_{m_i} - \cdots - \right. \\
 & \left. - z_2 \cdot \left\langle (m_{i-1} \cdots m_1)^{-1} \cdot m_1 \right\rangle_{m_i} - z_1 \cdot \left\langle (m_{i-1} \cdots m_1)^{-1} \right\rangle_{m_i} \right\rangle_{m_i}. \quad (2.27)
 \end{aligned}$$

With this optimization the iterative process requires only  $N$  cycles to compute the  $z_i$  mixed-radix digits, as depicted in Table 2.3.

The final computation of  $X$  is performed using the same approach as in the non-optimized conversion computation, and thus performing the RNS-to-binary conversion in  $2N$  cycles.

The following section describes the control unit for the proposed scalable RNS processor.

**Table 2.3** Optimized Mixed Radix Converter algorithm execution on the proposed RNS architecture

step	channel N	...	channel 2	channel 1
0	$x_N$		$x_2$	$x_1$
1	$x_N \cdot \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N}$		$x_2 \cdot \langle m_1^{-1} \rangle_{m_2}$	$x_1 \cdot 1$
2	$acc - \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N} \cdot z_1$		$acc_2 - \langle m_1^{-1} \rangle_{m_2} \cdot z_1$	$z_1$
3	$acc_N - \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N} \cdot m_1 \cdot z_2$		$z_2$	
4	$acc_N - \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N} \cdot m_2 m_1 \cdot z_3$			
...				
N	$acc_N - \langle m_{N-1} \cdots m_1^{-1} \rangle_{m_N} \cdot m_{N-2} \cdots m_1 \cdot z_{N-1}$			
N	$z_N$		$z_2$	$z_1$
N+1	$z_N \cdot m_N \cdots m_2 m_1 [n-1:0]$		$z_2 \cdot m_1$	$z_1$
N+2	$z_N \cdot m_N \cdots m_2 m_1 [2n-1:n]$			
...				
2N-1	$z_N \cdot m_N \cdots m_2 m_1 [N \cdot n-1:(N-1) \cdot n]$			

## 2.5 Control Units

In the RNS processor herein described, two types of control units are used: one for processor control denominated global control and another distributed control block (*CTR*, in Fig. 2.1) used in each arithmetic channel. The global control unit manages the fetching of code and the decoding of the respective instructions, alongside the management of the transfers to and from the data buffers. The single cycle arithmetic instructions perform the computation using data from registers and store the result into a destination register, without accessing the data bus, nor the constant memory. The multi-cycle instructions are controlled by a step counter, incremented at each iteration of the operation. When this counter achieves the number of iterations required to complete a given instruction (which depends on the number of channels in the RNS processor) the multi-cycle instruction is concluded, and another instruction can be fetched from the code buffer. The multi-cycle instructions are implemented by arithmetic operations partially supported by the arithmetic channels. The definition of these multi-cycle instructions is set by the micro-code to be executed. This micro-code is stored on the dedicated micro-code memory.

Table 2.4 depicts the operations performed on the arithmetic channels for the multi-cycle instructions, namely: (1) binary-to-RNS (mnemonic *bToRNS*); (2) MRS-to-RNS (*mToRNS*), used to convert from a mixed-radix representation to the RNS processor moduli set; (3) RNS-to-MRS (*rnsToM*); and (4) MRS-to-binary (*mToBIN*). Since only three types of arithmetic operations are required, and they only differ in the first steps of each instruction, the needed micro-code can be

**Table 2.4** Condensed micro-code for multi-cycle operations

Mnemonic	Operation step	Instructions
bToRNS rd, #, #	0	mula rd, #, #
mToRNS rd, #, #	$\neq 0$	amula rd, #, #
rnsToM rd, rs1, #	0	mula rd, rs1, #
mToBIN #, rs1, #	$\neq 0$	smula rd, rs1, #

reduced to a relative small memory, as described in Table 2.4. Instructions *bToRNS* and *mToRNS* actually have the exact same micro-code, differing only in the used constants. The same happens with the instructions *rnsToM* and *mToBIN*.

Alongside the arithmetic operations, used in each multi-cycle instruction, the operands source is also defined, namely: (1) the data buffer (for external data); (2) the internal shared bus; (3) the register file; and (4) the internal memory (for the constants stored in memory).

The binary-to-RNS and the MRS-to-RNS conversions instructions receive data from the data buffer and the internal memory, as source one and two, respectively. The RNS-to-MRS conversion instruction uses the internal shared bus as the source for the first operand and the internal memory as source for the second operand. The data presented in the internal bus is obtained from the arithmetic channels and defined by the operation step. This is used to propagate the mixed-radix-digits,  $z_i$ , through the other RNS channels, as depicted in Table 2.3. The last multi-cycle instruction, the MRS-to-binary computation, uses the register file and the internal memory as the operands source. Additionally, this last instruction disables the register update flag, since no register is to be updated. This conversion operation only uses the binary multipliers in the channels to compute  $X$ , as described in (2.23). Recall that the RNS-to-binary conversion is computed by two multi-cycle instructions: the first converts from RNS-to-MRS, computing the mixed-radix digits, and the second instruction converts from MRS-to-binary. The herein proposed architecture allows to implement Base Extension operations, supported by the RNS-to-MRS instruction in the source moduli set and by the MRS-to-RNS instruction in the destination moduli set.

Given the considered approach, the conversion instruction only depends on the step counter and on the stored constants for the chosen moduli set. With this, the proposed architecture becomes generic and scalable, supporting moduli sets of the form  $\{2^n \pm k_i\}$ . However, the number of moduli in the moduli set is limited by the fact that the moduli must be co-prime and of the restriction of  $w_k = \lceil \log_2(k) \rceil \leq n/2$ , given the used arithmetic units [17, 18]. As such, the larger the width of each channel,  $n$ , the higher the number of existing co-prime numbers and moduli that can co-exist.

Table 2.5 depicts the number of relative co-prime numbers that can be defined according to the allowed channel width. This table illustrates the number of existing co-prime values when considering moduli channels with only moduli of the form  $\{2^n - k_i\}$  and with moduli of the form  $\{2^n \pm k_i\}$ .

**Table 2.5** Number of relative prime numbers available for  $n$ -bit channel length

Channel width $n$ [bit]	8	12	16	20	...	32
$\{2^n - k\}$	6	14	43	129	...	4715
$\{2^n \pm k\}$	11	25	83	235	...	8677

Moduli sets composed only of  $\{2^n - k_i\}$  moduli are particularly interesting, given that they allow for simpler and more compact arithmetic units when compared with modulo  $\{2^n + k_i\}$  arithmetic units, as described in Sect. 2.4.2.

## 2.6 State-of-the-Art Analysis

In order to compare the herein described scalable and programmable RNS processor, the relevant state of the art is herein analysed, namely other programmable RNS architectures. Three other such systems are presented in the existing state of the art that satisfy this, namely: Residue Digital Signal Processor (RDSP), a general processing architecture proposed in [21]; the well-known Cox–Rower architecture proposed in [22]; and a more recent work implementing a unified computing system [23], where the code is defined as data transfers between registers and functional units. The last two architectures target the computation of asymmetric encryption algorithms, such as RSA [24] and ECC [25] algorithms.

### 2.6.1 State of the Art

The RDSP [21] is a 32-bit pipelined Reduced Instruction Set Computer (RISC) based processor with 5 pipeline stages. The arithmetic operations are performed by arithmetic units supporting the balanced moduli set  $\{2^n - 1, 2^{2n}, 2^n + 1\}$ , with  $n = 8$ . A particularity of this processor is that it also supports binary arithmetic operations. This processor supports the needed conversion operations, from binary-to-RNS and RNS-to-binary. For this, dedicated arithmetic units are used. The processor supports modular addition and multiplication operations, with and without accumulation. Furthermore, this processor offers flow control instructions (such as brunch instructions), feature not present in the remaining state of the art and in the processor herein described.

The main drawback of this architecture is that the supported moduli set only allows for the parallelism of three channels, providing a relatively small DR. In order to increase the DR and/or to reduce the width of the RNS channels, different RNS moduli sets could be considered for this processor, such as  $\{2^n - 3, 2^n - 1, 2^n + 1, 2^n + 3\}$  [26, 27]. However, this moduli set has the disadvantage of more complex modulo arithmetic, since the modular reduction of  $\{2^n \pm 3\}$  requires more area and imposes higher delay costs [28]. Considering only moduli of the form  $\{2^n\}$  and

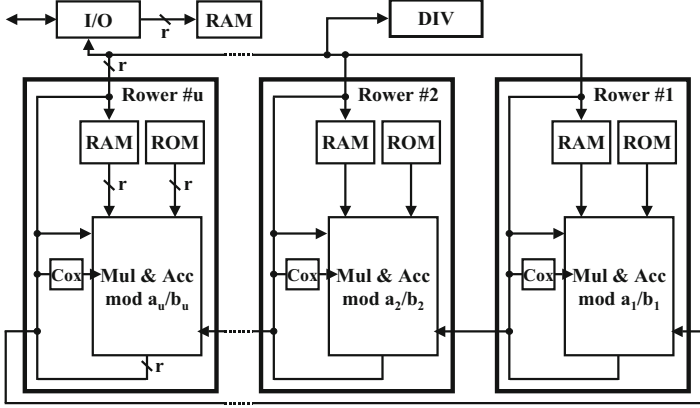


Fig. 2.6 Cox-Rower architecture, from [16]

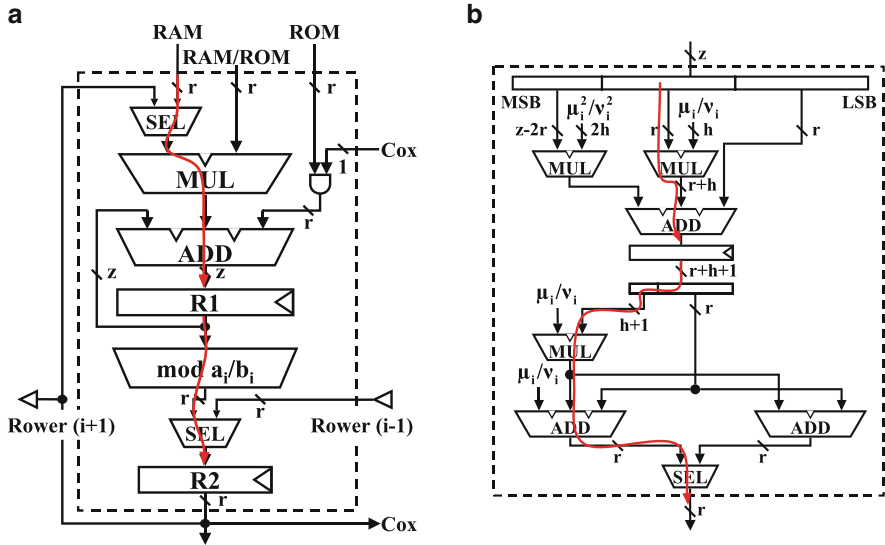
$\{2^n \pm 1\}$ , given its less complex structures, moduli sets with DRs up to  $(8n)$ -bit can be deployed, such as  $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1\}$  [29],  $\{2^n - 1, 2^n, 2^n + 1, 2^{2n} + 1\}$  [30],  $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1, 2^{n-1} - 1\}$  [31], and  $\{2^{n-5} - 1, 2^{n-3} - 1, 2^{n-3} + 1, 2^{n-2} + 1, 2^{n-1} - 1, 2^{n-1} + 1, 2^n, 2^n + 1\}$  [32]. Nevertheless, the predefined datapath of the RDSP processor does not allow to easily scale to larger DR.

The initial Cox-Rower architecture [22] is composed of a single Cox unit and by several Rower units. Each Rower unit is composed of a modular multiplier-and-accumulator, computing the channels RNS arithmetic. Note that each Rower unit is a dual-moduli channel, able to perform computations in two moduli sets (base one and two for the Montgomery Multiplication (MM) [13, 14]). The Cox unit is used to compute a correction factor, required in the MM. The Cox implementation is based on a simpler adder unit, computing the correction factor with bits received each cycle from the Rower units. An optimization to the initial architecture is proposed in [16], and corresponds to embedding a Cox unit in each Rower, as depicted in Fig. 2.6.

This approach avoids the broadcast of the correction factor data. To further improve data transfers, a ring connection is used, instead of a shared bus.

The authors also proposed an improved Rower unit, considering a three stage pipeline unit, towards a higher throughput and allowing to reduce the number of required Rower units in half. This unit implements a 32 bit long modular multiplier-and-accumulator, as depict in Fig. 2.7a. In the first stage of the pipeline, the Rower unit computes the binary multiplication and addition operations, with or without accumulation. The modular reduction is computed in the second and third stages of the pipeline. This reduction unit is implemented by three dual-constant multipliers and three adders, as depicted in Fig. 2.7b. Note that the Rower units are dual-moduli ( $2^r - \mu_i$  and  $2^r - \nu_i$  modulo), each supporting only two moduli.

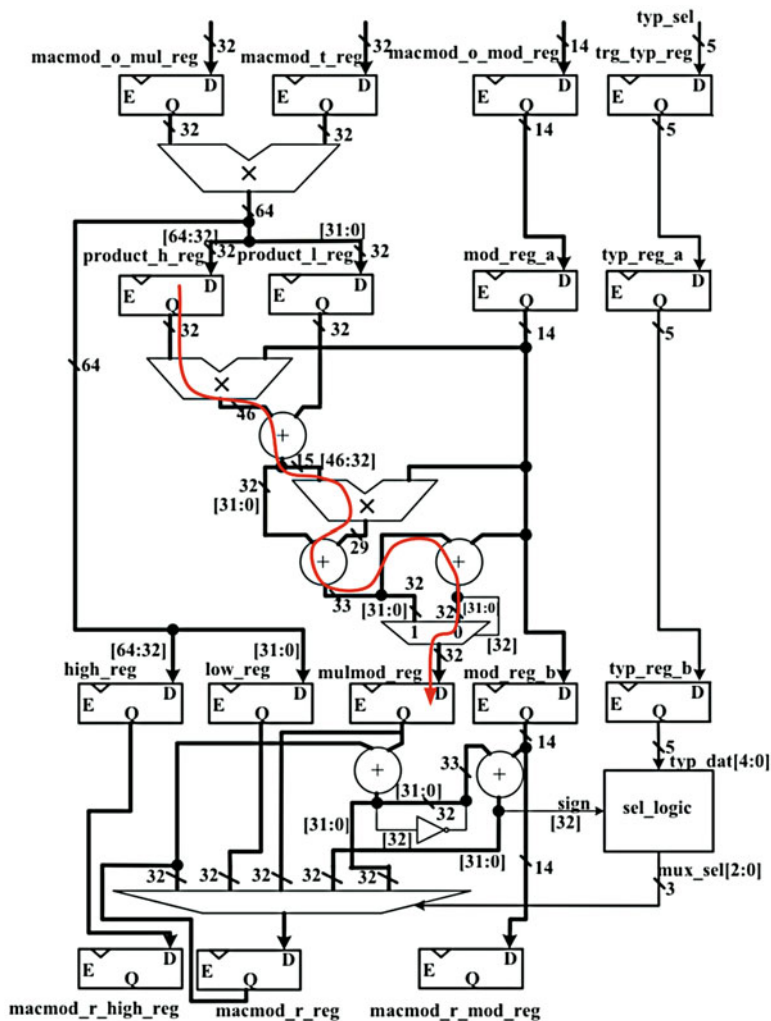
More recently, [33, 34] increased the number of pipeline stages of the Rower units, reaching higher frequencies. The pipeline was increased in four stages, implemented in the arithmetic block of the Rower unit. It also implements the



**Fig. 2.7** Rower architecture, from the Cox–Rower [16]. (a) Rower architecture. (b) Modular reduction

modular computation using the leak resistant arithmetic proposed in [35]. This modular arithmetic implements a countermeasure in the Cox–Rower architecture to provide better protections against side channel attacks.

Another RNS processor has been proposed in [36], based on a Transport Triggered Architecture (TTA), computing the base extensions using the same approach as [37]. The TTA approach allows for a higher instruction level parallelism, where the code is defined as data transports between registers and functional units. Operations start as a side-effect of transporting an operand to a “triggered port” of the functional unit. The presented structure deploys eight main functional units for the RNS channel arithmetic computation, for any modulo  $m_i$  of the used base. The number of functional units is upper bounded to eight, given that more functional units increase the number of buses. Moreover, in TTA [36] more buses require wider instruction words, resulting in a larger instruction memory. The main functional units of this RNS processor [36] are implemented by the Modular Multiplication-and-Accumulation (MMAC) units, as depicted in Fig. 2.8. However, these eight MMAC usually are less than the number of modulus, reducing the exploited RNS parallelism, since only eight modulo operations can be done simultaneously. In order to overcome this limitation, the MMAC is implemented as a three stage pipeline unit, allowing to take advantage of the inherent independence between RNS channels. The critical path of the MMAC functional unit is in the second pipeline stage, responsible for computing the modular reduction, as depicted in Fig. 2.8. The critical path is given by the delay of two multipliers, with  $32 \times 14$  and  $15 \times 14$  bits, and three binary adders, one with 46 bit length and two with 32 bit length.



**Fig. 2.8** MMAC architecture, from *uRNS* [23]

This work [36] has been further extended to a unified cryptographic processor [23], herein designated as *uRNS*, capable of computing RSA and ECC. In the *uRNS* the base extension is computed by the method proposed in [16]. The MMAC units are distributed in groups of four units each, up to 16 groups, allowing for different optimizations and performance levels, according to the key lengths. However, in the presented implementation the number of groups is restricted to 2, in order to maintain the number of buses and the instruction memory bounded, to allow it to be implemented.

### 2.6.2 Analysis Based on the Arithmetic Units

Given the related state of the art, only the two last processors are herein considered for a theoretical assessment, due to their support of large Dynamic Ranges. In order to compare these two last processors with the proposed RNS processor, only their modular arithmetic units are considered, since only these units are detailed in the literature [16, 23]. Note that the bus architectures details and operations scheduling are missing in the relevant related state of the art, not allowing for a full processor analysis. To obtain a technology independent assessment of the resulting arithmetic RNS architecture, an analysis is carried out using a neutral Full Adder based model [27, 38–40]. The estimation model considers that the area of 1-bit Full-Adder (FA) is represented by  $\Delta_{\text{FA}}$ , and the  $\tau_{\text{FA}}$  represents its delay. For the  $n$ -bit binary multipliers a delay of  $2n\tau_{\text{FA}}$  and an area of  $n^2 \Delta_{\text{FA}}$  are considered [41]. For the multiplication of an  $n$ -bit operand by a  $w_k$ -bit constant, a similar binary multiplier estimation is considered, with  $nw_k \Delta_{\text{FA}}$  of area resources usage and  $(n + w_k)\tau_{\text{FA}}$  of delay. The area resources and delay cost of registers are not taken into account in this estimation model. Considering this, the assessment of the related state of the art is based on the evaluation of the estimation costs for area and delay of the channel's arithmetic structures.

The evaluation of the described RNS architecture is based on the estimation cost of the channel's arithmetic structures, described in Sect. 2.4.2. Given that the area cost estimation for the modulo  $\{2^n - k\}$  channel arithmetic block is imposed by the area of one binary multiplier ( $n^2 \Delta_{\text{FA}}$ ), two constant multipliers, contributing with  $2nw_k \Delta_{\text{FA}}$ , and one 5:1 modular adder, implemented with one 4:1 modular adder and one modular Carry-Save-Adder (CSA) [41], contributing with  $(8n + 2w_k) \Delta_{\text{FA}}$ . As illustrated in Fig. 2.3, the critical path of the resulting channel arithmetic structure is imposed by the binary multiplier, contributing with  $2n\tau_{\text{FA}}$ , the two constant multipliers, contributing with  $2(n + w_k)\tau_{\text{FA}}$ , and the 5:1 modular adder, imposing a delay of  $(n + 5)\tau_{\text{FA}}$ . Note that these units do not have internal registers, since no pipeline is considered for them in the herein proposed architecture. An identical analysis can be performed for the arithmetic channel modulo  $\{2^n + k\}$ , resulting in the values depicted in Table 2.6.

The estimated area cost for the Rower unit of the Cox–Rower architecture, using the same FA model, is imposed by the area of the three pipeline stages. For the total area estimation cost the first pipeline stage contributes with the area of one binary multiplier ( $r^2 \Delta_{\text{FA}}$ , where  $r$  is the bit width of the moduli) and one 3:1 binary compression unit ( $2z \Delta_{\text{FA}}$ , where  $z$  represents a bit length greater than  $2r$ , considered to allow accumulation operations). The second pipeline stage is implemented by two binary multiplier and one 3:1 binary compression unit, contributing to the estimation

**Table 2.6** Area and delay estimation for the channel arithmetic structures

Modulo	$\Delta$ (area) [ $\Delta_{\text{FA}}$ ]	$\tau$ (delay) [ $\tau_{\text{FA}}$ ]
$2^n - k$	$n^2 + 8n + (2n + 2)w_k$	$5n + 2w_k + 5$
$2^n + k$	$(n + 1)^2 + (2n + 2)w_k + 8n$	$5n + 2w_k + 7$

cost with  $(2h(z - 2r)\Delta_{FA})$ ,  $((h \cdot r)\Delta_{FA})$ , and  $(2(r + h)\Delta_{FA})$ , respectively (note that  $h$  represents the bit length of the  $\mu_i$  and  $v_i$ ), as depicted in Fig. 2.7b. Finally, the third stage contributes with the area of one binary multiplier, one 3:1 binary compression unit, and one binary adder, which are given by  $((h^2 + h)\Delta_{FA})$ ,  $(2r\Delta_{FA})$ , and  $(r\Delta_{FA})$ , respectively. The estimation delay cost is given by the maximum critical path, located in the first pipeline stage as depicted in Fig. 2.7 and (2.29). The total estimations for area and delay, for the Rower unit of the Cox–Rower architecture, are given by:

$$\begin{aligned}\Delta_{\text{Rower}} &= \Delta_{\text{1st pipeline stage}} + \Delta_{\text{2nd pipeline stage}} + \Delta_{\text{3rd pipeline stage}} \\ &= \left[ (r^2 + 2z) + (2h(z - 2r) + h \cdot r + 2(r + h)) + (h^2 + h + 2r + r) \right] \cdot \Delta_{FA} \\ &= (r^2 + h^2 + 2z + 5r + 3h + 2hz - 3hr) \cdot \Delta_{FA}\end{aligned}\quad (2.28)$$

$$\begin{aligned}\tau_{\text{Rower}} &= \max(\tau_{\text{1st pipeline stage}}, \tau_{\text{2nd pipeline stage}}, \tau_{\text{3rd pipeline stage}}) \\ &= \max(2r + z + 1, 2r + 2h + 1, 2h + 2r + 2) \cdot \tau_{FA} \\ &= (2r + z + 1) \cdot \tau_{FA}.\end{aligned}\quad (2.29)$$

where  $z = 72$ ,  $r = 32$ , and  $h = 10$ , for a 32 bit channels. Note that the area and delay of the registers are not taken into account in the considered model.

Similar analysis, to the Cox–Rower, is applied to the unified cryptographic processor ( $uRNS$ ) [23], where the estimation area cost is given by: (1) one 32 bit binary multiplier, in the first pipeline stage; (2) one 32 by 14 bit binary multiplier, one 46 bit binary adder, one 15 by 14 bit binary multiplier, and two 32 bit binary adders, in the second stage; and (3) two 32 bit binary adders, in the third pipeline stage. Note that the MMAC unit proposed in  $uRNS$  [23] has a fixed bit width. Given this, fixed values are herein considered in the estimated cost analysis, instead of a generic form. Recall that the estimation delay cost for the  $uRNS$  is given by the critical path of the second pipeline stage, as depicted in Fig. 2.8. The estimation costs for the  $uRNS$  [23] are summarized as:

$$\begin{aligned}\Delta_{uRNS} &= \Delta_{\text{1st pipeline stage}} + \Delta_{\text{2nd pipeline stage}} + \Delta_{\text{3rd pipeline stage}} \\ &= [(32^2) + (32 \cdot 14 + 46 + 15 \cdot 14 + 32 + 32) + (32 + 32)] \cdot \Delta_{FA} \\ &= 1856\Delta_{FA}\end{aligned}\quad (2.30)$$

$$\begin{aligned}\tau_{uRNS} &= \max(\tau_{\text{1st pipeline stage}}, \tau_{\text{2nd pipeline stage}}, \tau_{\text{3rd pipeline stage}}) \\ &= \max(2 \cdot 32, 32 + 14 + 46 + 15 + 14 + 32 + 32, 32 + 32) \cdot \tau_{FA} \\ &= \max(64, 185, 64) \cdot \tau_{FA} \\ &= 185\tau_{FA}\end{aligned}\quad (2.31)$$

**Table 2.7** Theoretical analysis for the RNS arithmetic channels

Architecture	Pipeline stages	$\Delta$ (area) [ $\Delta_{FA}$ ]	$\tau$ (delay) [ $\tau_{FA}$ ]
Cox–Rower [16]	3	1938	137
<i>uRNS</i> [23]	3	1856	185
Proposed	0	1940	185

Table 2.7 summarizes and compares the estimated cost for these architectures, considering channels with 32 bit of length, the channel’s fixed width in the *uRNS* [23] architecture. Additionally, in order to make a fair comparison,  $k$  values up to 10 bits in length ( $w_{k_i} \leq 10$ ) are herein considered for the proposed arithmetic unit, since this is the maximum bit length allowed in the Cox–Rower architecture, namely as  $h$  in [16]. However, in *uRNS* [23] the bit length is less restricted, achieving values up to 14 bits.

From this theoretical analysis it can be concluded that the proposed RNS arithmetic units require similar area resources to those in the state of the art. Delay wise, the obtained metrics suggest that the Cox–Rower units are about 25% faster when using a three stage pipeline. However, these units are only able to compute two distinct modulus, not allowing to fully take advantage of the existing pipeline. On the other hand, the arithmetic units from the *uRNS* architecture are equally fast and also support generic modulus. However, this delay is achieved by using a three stage pipeline, while the arithmetic unit herein presented achieves identical metrics without pipeline. Furthermore, besides also allowing the usage of pipeline the herein considered RNS arithmetic units implement several operations (such as modular addition, subtraction, multiplication, and accumulation), unlike the one in the related state of the art that supports only one single operation. These features facilitate the implementation of more complex operations and a higher level of programmability.

Architecture wise, the Cox–Rower and *uRNS* architectures are very oriented to the computation of asymmetric cryptographic algorithms lacking adaptability to the computation of other algorithms. Data transfer wise, the optimized Cox–Rower architecture considers a ring bus, minimizing the impact of the bus in the system but significantly restricting the possible data transfers. On the other hand, the *uRNS* architecture presents a very powerful bus network, allowing the wide data transfer between units, needed for the TTA approach. The consequence of this highly interconnected bus is the lack of scalability, only supporting up to 8 arithmetic units, without causing significant data transfer delays.

The RNS architecture herein considered presents a trade-off in terms of data transfer and performance, allowing for a versatile data transfer and scalability. Moreover, the processor herein presented is fully programmable architecture, allowing it to be used in the computation of a wide range of algorithms, thus offering a programmable computing system supported by RNS.

## 2.7 Summary

The work described in this chapter presents the first version of a flexible and adaptable comprehensive framework to automatically design RNS processors. This RNS architecture can also be used to promote the development of novel moduli sets and optimized reverse converters.

Herein a compact and scalable RNS architecture is described, allowing for the design of RNS-based processors for any co-prime moduli set using moduli of the form  $\{2^n \pm k_i\}$ . This processor allows for modular additions, subtractions, and multiplication operations, with or without accumulation, supported by the proposed unified arithmetic units, while also providing the needed forward, reverse, and base extension conversions. With the proposed architecture the delay and area cost of dedicated processors are reduced while further exploring the parallelism and carry-free characteristic of RNS.

In conclusion, the proposed RNS processor allows for generic, scalable, and compact implementations with competitive performances when compared with the commonly used binary based systems and the related RNS state of the art, showing that efficient programmable RNS architectures can be designed.

## References

1. S.R. Barraclough, M. Sotheran, K. Burgin, A.P. Wise, A. Vadher, W.P. Robbins, R.M. Forsyth, The design and implementation of the IMS A110 image and signal processor, in *Proceedings of the IEEE Custom Integrated Circuits Conference*, May 1989, pp. 24.5/1–24.5/4
2. W.A. Chren, RNS-based enhancements for direct digital frequency synthesis. *IEEE Trans. Circuits Syst. II Analog Digit. Signal Process.* **42**(8), 516–524 (1995)
3. F. Piazza, E. Di Claudio, G. Orlandi, Fast combinatorial RNS processor for DSP applications. *IEEE Trans. Comput.* **44**(5), 624–633 (1995)
4. T.J. Slegel, R.J. Veracca, Design and performance of the IBM enterprise system/9000 type 9121 vector facility. *IBM J. Res. Dev.* **35**, 367–381 (1991)
5. C.-L. Wang, New bit serial VLSI implementation of RNS FIR digital filters. *IEEE Trans. Circuits Syst. II Exp. Briefs* **41**(11), 768–772 (1994)
6. P.M. Matutino, L. Sousa, An RNS based specific processor for computing the minimum sum-of-absolute-differences, in *11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, September 2008, pp. 768–775
7. J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, T. Wedi, Video coding with H.264/AVC: tools, performance, and complexity. *IEEE Circuits Syst. Mag.* **4**(1), 7–28 (first 2004)
8. J. Bajard, L. Imbert, A full RNS implementation of RSA. *IEEE Trans. Comput.* **53**(6), 769–774 (2004)
9. Antão, S., Bajard, J.-C., Sousa, L.: RNS based elliptic curve point multiplication for massive parallel architectures. *Comput. J.* 2011 - *Oxf. J.* **55**(5), 629–647 (2011)
10. G.C. Cardarilli, A. Nannarelli, M. Re, Residue number system for low-power DSP applications, in *Asilomar Conference on Signals, Systems and Computers - ACSSC 2007* (2007), pp. 1412–1416
11. P. Garai, C.B. Dutta, RNS based reconfigurable processor for high speed signal processing, in *IEEE Conference TENCON 2014*, October 2014, pp. 1–6

12. P.L. Montgomery, Modular multiplication without trial division. *Math. Comput.* **44**(170), 519–521 (1985)
13. J. Bajard, L.-S. Didier, P. Kornerup, An RNS montgomery modular multiplication algorithm, in *13th IEEE Symposium on Computer Arithmetic (ARITH)*, July 1997, pp. 234–239
14. K.C. Posch, R. Posch, Modulo reduction in residue number systems. *IEEE Trans. Parallel Distrib. Syst.* **6**(5), 449–454 (1995)
15. Advanced RISC Machines Ltd (ARM), *ARM7 Data sheet*. Advanced RISC Machines Ltd (ARM), ARM DDI 0020C edition, December 1994
16. H. Nozaki, M. Motoyama, A. Shimbo, S. Kawamura, Implementation of RSA algorithm based on RNS montgomery multiplication, in *Cryptographic Hardware and Embedded Systems - CHES 2001*, ed. by Ç. Koç, D. Naccache, C. Paar. *Lecture Notes in Computer Science*, vol. 2162 (Springer, Berlin, Heidelberg, 2001), pp. 364–376
17. P. Miguens Matutino, R. Chaves, L. Sousa, An efficient scalable RNS architecture for large dynamic ranges. *J. Signal Process. Syst.* **77**(1–2), 191–205 (2014)
18. P.M. Matutino, R. Chaves, L. Sousa, Arithmetic-based Binary-to-RNS converter modulo  $\{2^n \pm k\}$  for  $j$ -bit dynamic range. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **23**(3), 603–607 (2015)
19. A. Omondi, B. Premkumar (eds.), *Residue Number Systems: Theory and Implementation* (Imperial College Press, London, 2007)
20. Y. Wang, Residue-to-binary converters based on new Chinese remainder theorems. *IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process.* **47**(3), 197–205 (2000)
21. R. Chaves, L. Sousa, RDSP: A RISC DSP based on Residue Number System, in *Proceedings of the Euromicro Symposium on Digital System Design*, September 2003, pp. 128–135, ed. by IEEE
22. S. Kawamura, M. Koike, F. Sano, A. Shimbo, Cox-Rower architecture for fast parallel montgomery multiplication, in *Advances in Cryptology - EUROCRYPT 2000*, ed. by B. Preneel. *Lecture Notes in Computer Science*, vol. 1807 (Springer Berlin, Heidelberg, 2000), pp. 523–538
23. J. Wei, W. Guo, H. Liu, Y. Tan, A unified cryptographic processor for RSA and ECC in RNS, in *Computer Engineering and Technology*, ed. by W. Xu, L. Xiao, C. Zhang, J. Li, L. Yu. *Communications in Computer and Information Science*, vol. 396 (Springer, Berlin, Heidelberg, 2013), pp. 19–32
24. R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **26**(1), 96–99 (1983)
25. R. Schroepel, H. Orman, S. O'Malley, O. Spatscheck, Fast key exchange with elliptic curve systems, in *Advances in Cryptology - Crypto '95*. *Lecture Notes in Computer Science*, vol. 963 (Springer, Berlin, 1995), pp. 43–56
26. P.V. Ananda Mohan, Reverse converters for the moduli sets  $\{2^{2N} - 1, 2^N, 2^{2N} + 1\}$  and  $\{2^N - 3, 2^N + 1, 2^N - 1, 2^N + 3\}$ , in *SPCOM '04*, December 2004, pp. 188–192
27. M.-H. Sheu, S.-H. Lin, C. Chen, S.-W. Yang, An efficient VLSI design for a residue to binary converter for general balance moduli  $\{2^n - 3, 2^n + 1, 2^n - 1, 2^n + 3\}$ . *IEEE Trans. Circuits Syst. II: Express Briefs* **51**(3), 152–155 (2004)
28. P. Miguens Matutino, R. Chaves, L. Sousa, Arithmetic units for RNS moduli  $\{2^n - 3\}$  and  $\{2^n + 3\}$  operations, in *13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, September 2010, pp. 243–246
29. A.B. Premkumar, A.P. Vinod, A memoryless reverse converter for the 4-moduli superset  $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1\}$ . *J. Circuits Syst. Comput.* **10**(01n02), 85–99 (2000)
30. B. Cao, C.-H. Chang, T. Srikanthan, An efficient reverse converter for the 4-moduli set  $\{2^n - 1, 2^n, 2^n + 1, 2^{2n} + 1\}$  based on the new Chinese Remainder Theorem. *IEEE Trans. Circuits Syst. I Fundam. Theory Appl.* **50**(10), 1296–1303 (2003)
31. B. Cao, C.-H. Chang, T. Srikanthan, A residue-to-binary converter for a new five-moduli set. *IEEE Trans. Circuits Syst. I: Regul. Pap.* **54**(5), 1041–1049 (2007)
32. A. Skavantzios, M. Abdallah, T. Stouraitis, D. Schinianakis, Design of a balanced 8-modulus RNS, in *16th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2009*, December 2009, pp. 61–64

33. N. Guillermin, A high speed coprocessor for elliptic curve scalar multiplications over  $F_p$ , in *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems, CHES'10* (Springer-Verlag, Berlin, Heidelberg, 2010) pp. 48–64
34. N. Guillermin, A coprocessor for secure and high speed modular arithmetic. Cryptology ePrint Archive, Report 2011/354, July 2011
35. J.-C. Bajard, L. Imbert, P.-Y. Liardet, Y. Tegliah, Leak resistant arithmetic, in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ed. by M. Joye, J.-J. Quisquater. Lecture Notes in Computer Science, vol. 3156 (Springer, Berlin, Heidelberg, 2004), pp. 62–75
36. W. Guo, Y. Liu, S. Bai, J. Wei, D. Sun, Hardware architecture for rsa cryptography based on residue number system. *Trans. Tianjin Univ.* **18**(4), 237–242 (2012)
37. J.-C. Bajard, L.-S. Didier, P. Kornerup, Modular multiplication and base extensions in residue number systems, in *IEEE 15TH Symposium on Computer Arithmetic* (IEEE, New York, 2001), pp. 59–65
38. P.V. Ananda Mohan, New reverse converters for the moduli set  $\{2^n - 3, 2^n - 1, 2^n + 1, 2^n + 3\}$ . *AEU - Int. J. Electron. Commun.* **62**(9), 643–658 (2008)
39. H. Pettenghi, R. Chaves, L. Sousa, RNS reverse converters for moduli sets with dynamic ranges up to  $(8n+1)$ -bit. *IEEE Trans. Circuits Syst. I Regul. Pap.* **PP**(99), 1–14 (2012)
40. G. Jaberipur, H. Ahmadifar, A rom-less reverse RNS converter for moduli set  $\{2^q \pm 1, 2^q \pm 3\}$ . *IET Comput. Digit. Tech.* **8**(1), 11–22 (2014)
41. P.M. Matutino, H. Pettenghi, R. Chaves, L. Sousa, RNS arithmetic units for modulo  $\{2^n \pm k\}$ , in *15th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, September 2012, pp. 795–802

Embedded Systems Design with Special Arithmetic and  
Number Systems

Molahosseini, A.S.; de Sousa, L.S.; Chang, C.-H. (Eds.)

2017, X, 389 p. 127 illus., 48 illus. in color., Hardcover

ISBN: 978-3-319-49741-9