Optimizing Residue Arithmetic on FPGAs

Haohuan Fu, Oskar Mencer, Wayne Luk
Department of Computing, Imperial College
London, United Kingdom
{*hfu,oskar,wl*}*@doc.ic.ac.uk*

Abstract

Residue Number System (RNS), which originates from the Chinese Remainder Theorem, is regarded as a promising number representation in the domain of Digital Signal Processing (DSP). This paper describes our work on optimizing residue arithmetic units on the platform of reconfigurable devices, such as FPGAs. First, we provide improved designs for residue arithmetic units. For reverse converters from RNS to binary numbers, we propose a novel design that uses only n-bit additions. Compared to previous work, the design consumes up to 14.3% less area and provides lower latency. Second, we develop a reconfigurable RNS arithmetic library generator for the moduli set $\{2^n-1, 2^n, 2^n+1\}$. *The generator supports a wide range of RNS numbers, and enables us to perform an extensive comparison between RNS and other number representations at both the arithmetic unit level and the application level. The comparison shows that, for applications involving a large number of multiplications, the RNS designs can reduce up to 1/2 DSP48s for large bit-width settings.*

1. Introduction

Dated back to the Chinese Remainder Theorem (CRT) in the ancient Chinese Mathematic book "Sun Zi Suan Jing" [1] (with a recent English translation in [2]), the theory behind Residue Number System (RNS) has existed for over hundreds of years. By decomposing one large number into a number of small residue values, RNS greatly reduces the carry chain length of adders and the size of multipliers. Compared to conventional binary representations, RNS provides low latency in addition and multiplication, plus possible reduction in area and power consumption [3]. Due to these special features, RNS is long regarded as a promising number format in Digital Signal Processing (DSP) [4, 5].

However, RNS also has its inherent disadvantages when compared to binary representations. As the residue values do not contain any magnitude information, comparison, scaling and division are difficult. Conversions between binary and residue numbers are costly. These difficulties constrain the utilization of RNS to practical applications.

As a reconfigurable hardware device, FPGA is an ideal platform to evaluate the RNS representations. There are existing work on implementing residue arithmetic on FPGAs [6], and also research efforts that use RNS to implement applications such as Intellectual Property Protection (IPP) procedures [7] and RSA algorithm [8] on FPGAs. However, there still lacks a general support for RNS arithmetic.

To facilitate further investigations on the potential of RNS in different applications, we work on improving the arithmetic designs of RNS numbers, and developing an optimized RNS arithmetic library generator that targets the FPGA platform. The optimized arithmetic library generator enables a fast prototyping of efficient RNS designs. By studying the tradeoffs between RNS and other number representations, we can apply RNS to suitable applications to achieve improvement in performance and precision or reduction in resource consumption.

Our major contributions are:

1. improvement of RNS arithmetic on FPGAs. For reverse converters from RNS to binary numbers, we propose a novel design which consumes up to 14.3% less area and provides lower latency than previous work [9]. The forward converter are implemented with simple modular additions. We also provide simplified solutions for specific cases of scaling and magnitude comparison operations.

2. an RNS arithmetic library generator for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. We keep n as a reconfigurable parameters, so that the generator supports a wide range of RNS numbers.

3. significant resource reduction by using RNS for multiplications. We can fit more multipliers into one FPGA.

Using the library generator, we perform an extensive comparison between RNS and other number representations

on both arithmetic units and benchmarks. The comparison shows that, the major advantage of RNS lies in the reduction of resources in multiplication operations. In applications that involves a large number of multiplications, the RNS representation can reduce up to 1/2 DSP48s for large bit-widths, thus fit more designs into the FPGA device.

2. Background

2.1. RNS Notations

$MS = \{M_1, M_2, \cdots M_n\}$ is the moduli set which contains n different moduli that are pairwise co-prime to each other. The representation range of the moduli set is $M = M_1 \cdot M_2 \cdots M_n$. A number X in the range of $[0, M-1]$ can be represented by $\{x_1, x_2, \cdots x_n\}$, where x_i is the residue value of X mod M_i, denoted as $|X|_{M_i}$. For the cases that we need to represent negative values, the range becomes $[-M/2, M/2-1]$. For the convenience of discussions, we also define S_i as $S_i = \prod_{j \neq i}^{n} M_j = M/M_i$, and the value $|x^{-1}|_M$ as the multiplicative inverse of $|x|_M$ that satisfies $\left||x^{-1}|_M \cdot x\right|_M = 1$.

2.2. Selection of Moduli Set

In our work, we choose to use the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ for the following reasons: firstly, it provides simpler designs for converters and magnitude-related operations, thus is more possible to provide efficient designs for common applications; secondly, although power-of-two moduli set can not use the index calculus approach to implement efficient multipliers [10], we can utilize the dedicated hardware multipliers on FPGA platforms to implement multiplications with acceptable cost; thirdly, as it is the most commonly used one in previous work, using this moduli set makes our work applicable to most existing designs.

2.3. Previous Work on RNS Arithmetic

Forward converter

As most existing devices and applications use binary representations, such as fixed-point or floating-point numbers, the first part of an RNS design is usually a converter that converts binary numbers into the residue form, usually denoted as forward converter or residue generator.

The early efforts [11, 12] on forward converters decompose the binary value into an array of power-of-two values, compute the residue of each power-of-two value, and sum them up with modular adders. The basic idea is as follows: assume $r_{ji} = |2^j|_{M_i}$, where $0 \leq j \leq n-1$, and $X = \sum_{i=0}^{n-1} b_i \cdot 2^i$. The residue x_i can then be processed as $x_i = |\sum_{j:b_j=1} r_{ji}|_{M_i}$.

Based on the above approach, S. Piestrak [13] proposes the concept of periodic properties of modular operations in his converter designs. Based on the periodic property, the input bits can be divided into a number of groups and handled similarly, which greatly simplify the design.

Reverse converter

Compared to the forward converter, the reverse converter (converts RNS numbers back into binary form, also referred to as RNS decoder) is more complicated and costs more resources to implement.

The earliest algorithm of reverse conversion dates back to the Chinese Remainder Theorem (CRT), which is proposed by an ancient Chinese mathematician Tzu Sun [1, 2]. The idea of CRT can be illustrated as equation (1). Multiplicative inverse values are used to reconstruct the binary value from residue values.

Y. Wang proposes another reverse conversion algorithm which he names as the New Chinese Remainder Theorems [14]. This algorithm calculates the binary value as shown in equation (2). Similar ideas are proposed for conversion from RNS numbers to Mixed-Radix (MR) representations [15]. Applying this algorithm to the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, Y. Wang provides an adder-based reverse converter [9], which is one of the most efficient reverse converter design for this moduli set.

Magnitude Comparison

The most straightforward way of magnitude comparison is to reverse-convert the residue representation into binary representation (or Mixed-Radix (MR) representations) and perform the comparison.

There are also other methods to derive a monotonic growing function from the residue values. Based on the observation of a "diagonal function" in the sequence of RNS numbers, G. Dimauro et. al. [16] propose a function that use residue values and an extra modulus to compute a value that grows monotonically with the magnitude of RNS numbers. However, the computation of the "diagonal" value still involves costly multiplication and modular operations.

3. Design of RNS Arithmetic Units

As noted in section 2.2, in order to reduce the complexity of certain RNS arithmetic units and make the representation more applicable to common applications, we select the most commonly-used moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ in our RNS arithmetic design. In the following sections, we show in details how this moduli set simplifies the design of arithmetic units, such as converters between binary and RNS numbers, scaling operator and comparison operator.

$$
\begin{aligned}
X &= \{x_1, x_2, \cdots, x_n\} = |\{x_1, 0, 0, \cdots, 0\} + \{0, x_2, 0, \cdots, 0\} + \cdots + \{0, 0, \cdots, x_n\}|_M \\
&= \left|x_1 \cdot |S_1^{-1}|_{M_1} \cdot S_1 + x_2 \cdot |S_2^{-1}|_{M_2} \cdot S_2 + \cdots + x_n \cdot |S_n^{-1}|_{M_n} \cdot S_n\right|_M = \left|\sum_{i=1}^{n} (|S_i^{-1}|_{M_i} \cdot S_i \cdot x_i)\right|_M \qquad (1)
\end{aligned}
$$

$$
\begin{aligned}
X &= |x_1 + k_1 M_1 (x_2 - x_1) + k_2 M_1 M_2 (x_3 - x_2) + \cdots + k_{n-1} M_1 M_2 \cdots M_{n-1} (x_n - x_{n-1})|_{M_1 M 2 \cdots M_{n-1} M_n} \\
\text{where} \quad & |k_1 M_1|_{M_2 \cdots M_n} = 1, |k_2 M_1 M_2|_{M_3 \cdots M_n} = 1, \cdots, |k_{n-1} M_1 \cdots M_{n-1}|_{M_n} = 1 \qquad (2)
\end{aligned}
$$

3.1. Forward Converter

The previous designs of forward converters (detailed in section 2.3) breaks the binary number into bits, and calculate the residue values of each bit separately. In our design, as we use the specific moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, the generation of the residue values is greatly simplified.

The representation range of the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ is $2^{3n} - 2^n$, which corresponds to binary numbers with $3n$ bits. If we divide the $3n$ bits into three groups of n bits and denote each group with A, B, and C respectively, then the binary number X can be expressed as $X = A \cdot 2^{2n} + B \cdot 2^n + C$. The residue values of the three moduli can then calculated as follows:

$$
\begin{aligned}
x_1 &= |A \cdot 2^{2n} + B \cdot 2^n + C|_{2^n - 1} \\
&= |A \cdot (2^n - 1)(2^n + 1) + A + B \cdot (2^n - 1) + B + C|_{2^n - 1} \\
&= |A + B + C|_{2^n - 1} \qquad (3) \\
x_2 &= |A \cdot 2^{2n} + B \cdot 2^n + C|_{2^n} = C \qquad (4) \\
x_3 &= |A \cdot 2^{2n} + B \cdot 2^n + C|_{2^n + 1} \\
&= |A \cdot (2^n - 1)(2^n + 1) + A + B \cdot (2^n + 1) - B + C|_{2^n + 1} \\
&= |A - B + C|_{2^n + 1} \qquad (5)
\end{aligned}
$$

Therefore, the forward converter of this moduli set can be easily implemented through modular adders. For all the FPGA designs in this paper and our RNS arithmetic library, we use A Stream Compiler (ASC) [17] as the hardware compiler that maps the design into FPGA implementations. ASC is a high-level FPGA programming tool using C++ alike syntax. The compiler works as an add-on library to the standard C++ compilers, and supports hardware data-types, such as integer, fixed-point and floating-point number, with configurable bit-widths. In our experiments, all the designs are targeted on the Xilinx Virtex IV FX100 FPGA board . Table 1 shows the area cost and latency of RNS forward converters with typical bit-width settings.

3.2. Reverse Converter

As mentioned in section 2.3, the previous reverse converter designs are based on the CRT or the new CRT by Y. Wang [14], and usually involve multiplications or at least 2n-bit modular additions. In our design, we try to acquire the binary value with only n-bit adders.

Table 1. RNS Forward Converters: area cost and latency for different n values (the moduli set we use is $\{2^n - 1, 2^n, 2^n + 1\}$).

value of n	4	8	12	16	20
area cost / # slices	113	186	236	318	392
latency / ns	14.2	17.2	18.7	19.7	22.4

We describe our algorithm based on using the same A, B, C denotations as in section 3.1. As A, B, C are all values in the range of $[0, 2^n - 1]$, the equations for forward conversion ((3), (4), (5)) can be converted as follows (as noted in section 2.1, we denote three moduli values as M_1, M_2, and M_3):

$$
\begin{aligned}
A + B + C &= x1 + c_1 \cdot M_1, \quad \text{where} \quad c_1 \in \{0, 1, 2\} \qquad (6) \\
C &= x2 \qquad (7) \\
A - B + C &= x3 + c_2 \cdot M_3, \quad \text{where} \quad c_2 \in \{-1, 0, 1\} \qquad (8)
\end{aligned}
$$

Altogether, c_1 and c_2 have nine different combinations. However, given one set of residue values $\{x_1, x_2, x_3\}$, there is always only one valid combination of c_1 and c_2, as the above equations include hidden constraints among x_1, x_2, x_3 and c_1, c_2 values. The first kind of constraints is about the parity of the numbers. As $A+B+C$ and $A-B+C$ have the same parity, if x_1 and x_3 also have the same parity, then c_1 and c_2 shall have the same parity. Otherwise, if x_1 and x_3 have different parities, then c_1 and c_2 shall have different parities. The second kind of constraints relate to the range of the values. We can derive the expressions of $2A$ and $2B$ from the above equations (6), (7), and (8). As $2A$ and $2B$ shall both be in the range of $[0, 2^{n+1} - 2]$, we can derive another set of equations regarding the relationship between x_1, x_2, x_3 and c_1, c_2.

Applying all the above hidden constraints in the equations (6), (7), and (8), we can acquire a full set of complete

Table 2. Conditions for selecting valid c_1 and c_2 values. For each different c_1 or c_2 value, the multiple rows on the right describes the corresponding cases that the value is valid, i.e. if the x_i values satisfy any of the rows on the right, c_1 or c_2 are determined to be the value on the left. 'SAME' and 'DIFF' denote that x_1 and x_3 have the same or different parity.

$c_1 = 0$	$x_1 \geq x_3$ & $x_1 + x_3 \geq 2 \cdot x_2$ & SAME
	$x_1 + x_3 \geq 2 \cdot x_2 + M_3$ & DIFF
$c_1 = 2$	$x_1 < x_3$ & $x_1 + x_3 \leq 2 \cdot x_2$ & SAME
	$x1 + x3 \leq 2 \cdot x_2 - M_3$ & DIFF
$c_1 = 1$	all other cases
$c_2 = -1$	$x_1 < x_3$ & $x_1 + x_3 > 2 \cdot x_2$ & SAME
	$x_1 + x_3 \geq 2 \cdot x_2 + M_3$ & DIFF
$c_2 = 1$	$x_1 \geq x_3$ & $x1 + x3 > 2 \cdot x_2$ & SAME
	$x1 + x3 \leq 2 \cdot x_2 - M_3$ & DIFF
$c_2 = 0$	all other cases

conditions that correspond to different combinations of c_1 and c_2 values, as shown in Table 2.

Applying the above conditions, we can determine the values of c_1 and c_2 according to residue values $\{x_1, x_2, x_3\}$. As C equals to the value of x_2, we can then calculate the values A and B from the values of $A+B+C$ and $A-B+C$.

As shown in Figure 1, we implement the above reverse conversion algorithm as a circuit structure with two major parts. The first part is the condition decoder that takes the residue values $\{x_1, x_2, x_3\}$ as input, compare all the different value combinations as shown in Table 2, and figure out the correct coefficients c_1 and c_2. The second part is the binary generator that uses the coefficients c_1 and c_2 to construct the A and B parts. As part C comes from the value of x_2, we can combine them to produce the binary result.

To evaluate the cost and performance, we implement the reverse converter design on FPGAs with different n values (the n value in the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$), and compare the experiment results with the design by Wang et. al. [9], which is one of the most efficient existing designs for the $\{2^n - 1, 2^n, 2^n + 1\}$ moduli set. As shown in Fig. 2, compared to Wang's design, our reverse converters consume 7.7% to 14.3% less area on FPGAs. Our design consumes 7.7% for the case of $n = 4$, and consumes 14.3% less area for the case of $n = 20$. Thus, our reduction in area cost increases with the size of the moduli set. On the latency side, our design also takes less time to produce the output from the input values in most cases.

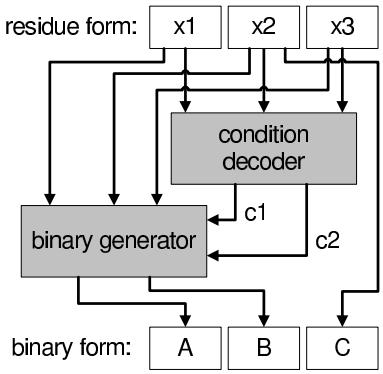


Figure 1. General structure of our reverse converter.

3.3. Addition and Multiplication

RNS's advantage in addition and multiplication comes from the following property:

$$|a + b|_M = \big| |a|_M + |b|_M \big|_M \quad (9)$$

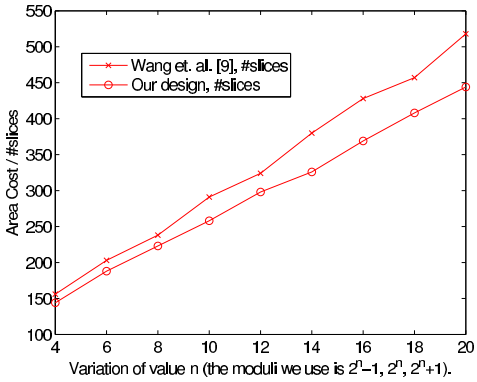
$$|a * b|_M = \big| |a|_M * |b|_M \big|_M \quad (10)$$

Thus, to add or multiply two RNS numbers $\{x_1, x_2, x_3\}$ and $\{y_1, y_2, y_3\}$, we only need to add or multiply the corresponding value pairs x_i and y_i. When implementing the adders and multipliers, we can apply the same techniques for binary adders and multipliers, such as carry-save adders and Booth's multiplication algorithms [18]. However, as our arithmetic library targets the FPGA platforms, we use the adder and multiplier cores provided by ASC [17] directly.

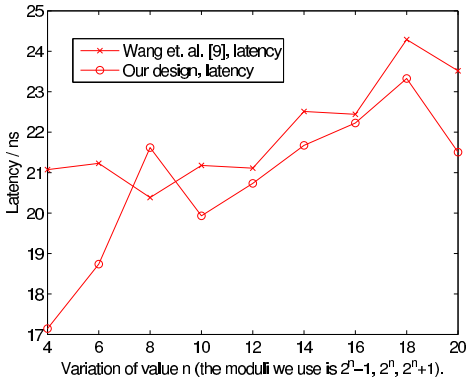
4. Comparing RNS and Integer Arithmetic Units

The previous section describes the algorithm design of the major units in our RNS arithmetic library. By implementing the designs using ASC syntax descriptions, we develop a generator for RNS arithmetic libraries. The generator provides arithmetic units for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, and takes n as an input parameter that the users can configure.

Combining this library generator and ASC, we have a tool that can automatically generate RNS arithmetic units for different bit-width, and evaluate the area cost and latency of the units. To compare RNS and common binary representations, we perform a comparison between RNS and integer arithmetic units with different bit-width settings.



(a) Comparison of area cost.



(b) Comparison of latency.

Figure 2. Comparing our reverse converter and the design in [9]: area cost and latency.

Figure 3 shows the comparison between RNS and integer adders. We compare RNS operations for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ to 3n-bit integer operations, as they have a similar representation range.

The area cost of both RNS and integer adders increase almost linearly to the bit-width of operands. The RNS adders consume around 50 slices more than the integer adders. The integer adder performs a 3n-bit addition, while the RNS adder performs three n-bit additions separately. However, the RNS adder also needs to perform a modular operation after the addition, which incurs more area consumption. On the latency side, the RNS adders also do not show the expected advantage. This is mainly because the adders are implemented on the FPGA with fast-carry chains, which greatly reduce the latency difference between the carry chain of a 3n-bit adder and the carry chain of a n-bit adder. Meanwhile, the extra modular step of RNS adders introduce more latency. Thus, in most cases, the RNS adders show a higher latency than integer adders. Only when the value n increases to 18, which corresponds to integers over 54 bits, the short carry chain of RNS starts to outweigh the extra latency of the modular step, and the RNS adders show a lower latency than the integer adders.

As RNS multiplier is quite different from integer multiplier, it is not clear how to perform an unbiased comparison of the multipliers of the two different number systems. RNS multiplications are bounded by the range of the moduli, i.e. the multiplication of two 3n-bit values only produce a 3n-bit result. For integers, the multiplication of two 3n-bit values produce a 6n-bit result. In our comparison, we try to make the computation complexity in the two number systems as equivalent as possible. Thus, we reduce the the multiplication of two 3n-bit integers into a partial multiplication that only produces to output the lower 3n bits as the result. The same reduction to integer or fixed-point multipliers is performed in the case studies in Section 5.

Figure 4 shows the comparison between RNS and integer multipliers. The multipliers can be implemented using only logic slices, or using both logic slices and DSP48s.

When using only logic slices, the area consumption for integer and RNS multipliers are similar. For bit-widths smaller than 10, RNS multiplier consumes more area due to the extra modular operations to the results. For bit-widths larger than 10, RNS multipliers start to bring area reductions. However, the latency of RNS multipliers are higher than integer multipliers, which is also because of the modular operation needed after the multiplication.

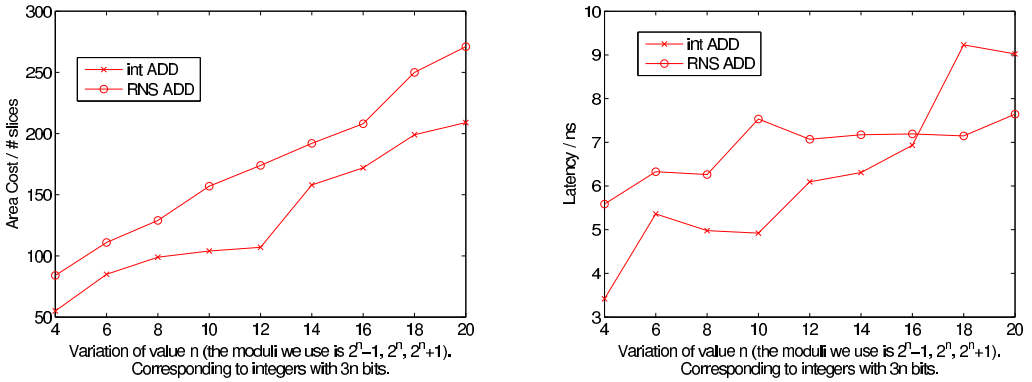
When using both logic slices and DSP48s, the RNS multipliers consume 50 to 200 more slices than integer multipliers, but the number of DSP48s is reduced greatly. For large bit-width ($n = 12, 14, 16$), the RNS multipliers only consume 1/2 of the number of DSP48s consumed by integer multipliers. When using DSP48s, the latency of RNS multipliers are still higher than integer multipliers.

The comparison results show that the major advantage of RNS lies in the reduction of resource consumptions for multipliers. Because of the extra modular operations after addition, RNS adders consume slightly more area than integer adders. For similar reasons, RNS adders and multipliers also show a higher latency than integer units. For large bit-widths, RNS adders produce a similar or even lower latency.

5. Evaluating RNS on Practical Benchmarks

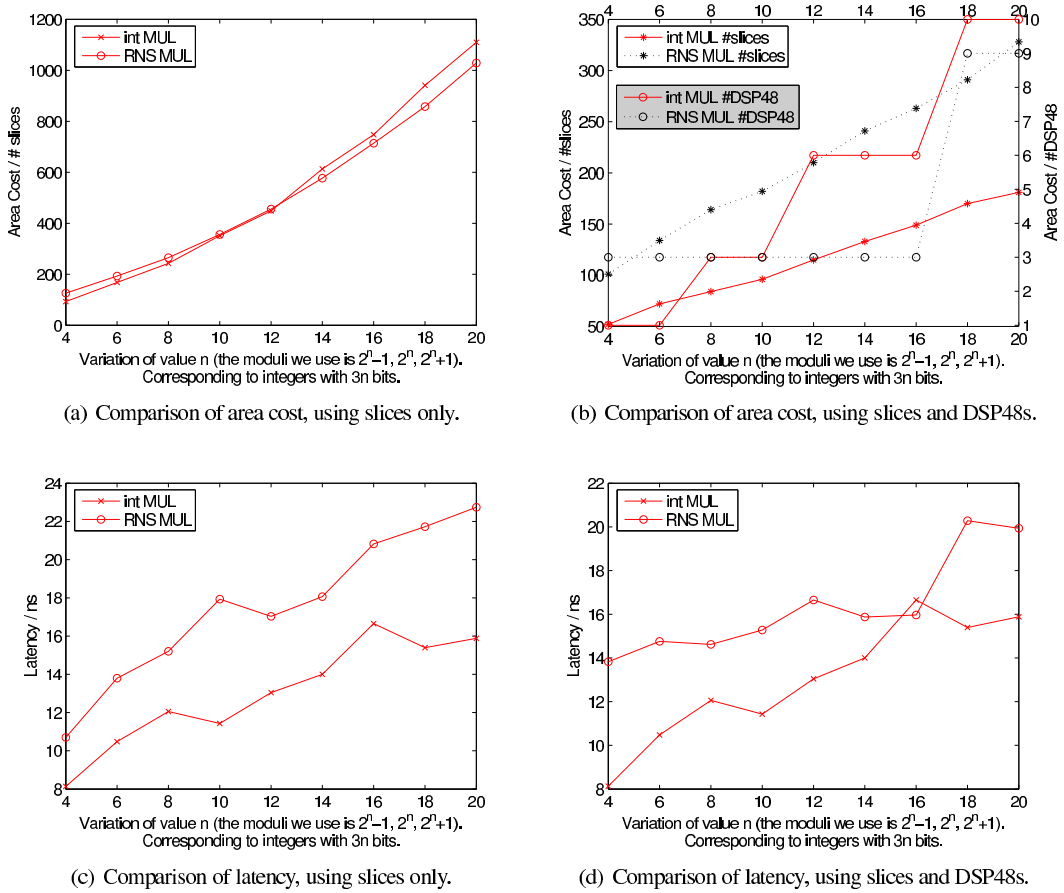
5.1. 2 by 2 Matrix Multiplication: RNS versus Integer

We compare the RNS designs and integer designs for a 2 by 2 matrix multiplication processing core. As we focus on the area cost and latency of the processing core itself, the



(a) Comparison of area cost. (b) Comparison of latency.

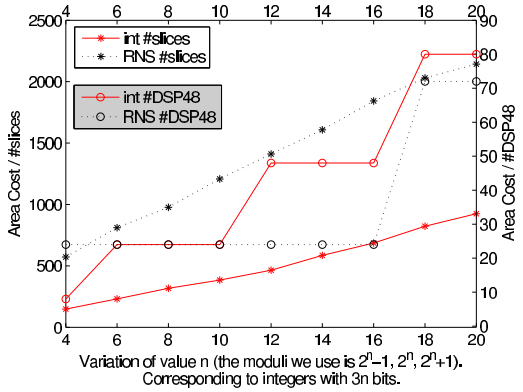
Figure 3. RNS adders versus integer adders: comparison of area cost and latency.



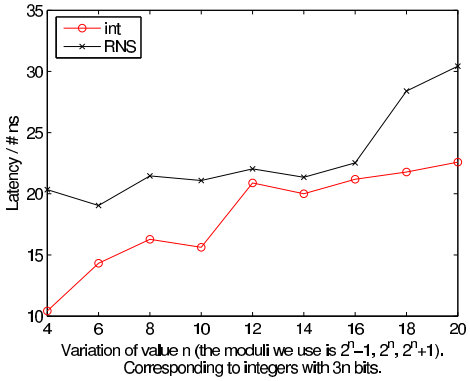
(a) Comparison of area cost, using slices only. (b) Comparison of area cost, using slices and DSP48s.

(c) Comparison of latency, using slices only. (d) Comparison of latency, using slices and DSP48s.

Figure 4. RNS multipliers versus integer multipliers: comparison of area cost and latency. The RNS multipliers can save up to 100 slices or 50% DSP48s for large bit-width settings.



(a) Comparison of area cost, using slices and DSP48s.



(b) Comparison of latency.

Figure 5. 2 by 2 matrix multiplication: comparison of area cost and latency between RNS and integer designs.

conversions between RNS and binary forms are not considered.

The processing core consists of 8 multiplications and 4 additions. As shown in Figure 5, the consumption of DSP48s is greatly reduced in RNS designs. For the cases that $n = 12, 14, 16$, the RNS designs consume only 1/2 DSP48s of the integer designs. Similarly, for the cases that $n = 12, 14, 16$, the RNS designs provide a similar latency to integer designs. As there are far less DSP48s involved in the design, the critical path is reduced. In the other cases, the latency introduced in the extra modular operations outweighs the reduced portion in DSP48s, the RNS designs show a higher latency. The RNS designs also consume around 500 to 1000 extra slices than integer designs due to the extra modular operations.

5.2. FIR Filter: RNS versus Fixed-point

In the second case study, we compare RNS and fixed-point representation in a 11-tap Finite Impulse Response (FIR) filter design. The computation involves 11 multiplications and 10 additions. We assume that the input and output values are in the form of fixed-point, thus forward and reverse converters are included in the RNS design. DSP48s are used for multipliers.

As shown in Figure 6, the usage of DSP48s is again greatly reduced for large bit-width cases. In RNS designs, the number of consumed DSP48s is reduced by 1/2 when $n = 12, 14, 16$. However, due to the converters added into the design and the extra modular operations, the RNS designs consume around 2000 more slices and show a much higher latency.

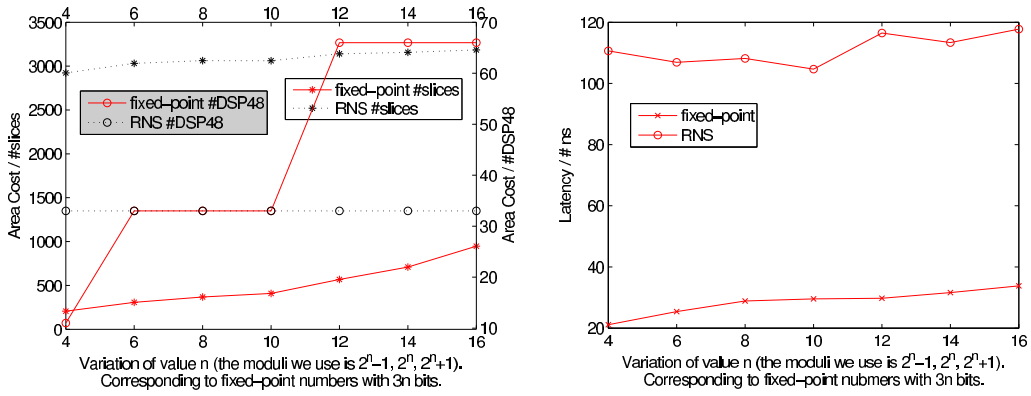
In this specific case, we implement the design on a Virtex IV FX100 FPGA, which consists of 42176 slices and 192 DSP48s. By using RNS representation, we can cut the number of DSP48s down to 1/2 for large bit-width cases ($n = 12, 14, 16$), and fit 5 copies of the same design into one FPGA, while the fixed-point version can only support 2 copies in one FPGA. On the other hand, we can also have more taps in the RNS design to produce a better FIR filter.

6. Conclusion

This paper optimizes RNS arithmetic designs on the platform of reconfigurable devices, such as FPGAs. In particular, we propose a novel design for reverse converters from RNS to binary numbers, which consumes up to 14.3% less area and provides lower latency. Based on the optimized RNS arithmetic units, we develop an RNS arithmetic library generator for the moduli set $\{2^n - 1, 2^n, 2^n + 1\}$. The generator takes the value n as a configurable parameter and supports a wide range of RNS numbers. This library generator enables us to perform an extensive comparison between RNS and other number representations at both the arithmetic unit level and the application level. The comparison shows that, the major advantage of RNS lies in the reduction of resources in multiplication operations. In applications that involves a large number of multiplications, the RNS representation can reduce up to 1/2 DSP48s for large bit-widths, thus fit more designs into the FPGA.

References

[1] T. Sun, *Sun Zi Suan Jing (The Mathematical Classic by Sun Zi)*, around 400 AD.



(a) Comparison of area cost.

(b) Comparison of latency.

Figure 6. 11-tap FIR filter: comparison of area cost and latency between RNS and fixed-point designs.

[2] L. R. Lam and T. S. Ang, *Fleeting Footsteps: Tracing the Conception of Arithmetic and Algebra in Ancient China*. World Scientific Publishing Company, 2004.

[3] G. Cardarilli, A. D. Re, A. Nannarelli, and M. Re, "Low-Power Implementation of Polyphase Filters in Quadratic Residue Number System," in *Proc. IEEE International Symposium on Circuits and Systems*, 2004, pp. 725–728.

[4] M. Mahesh and M. Mehendale, "Improving Performance of High Precision Signal Processing Algorithms on Programmable DSPs," in *Proc. IEEE International Symposium on Circuits and Systems*, 1999, pp. 488–491.

[5] R. Charles and L. Sousa, "RDSP: A RISC DSP Based on Residue Number System," in *Proc. Euromicro Symposium on Digital System Design*, 2003, pp. 128–135.

[6] T. Tomczak, "Optimizing Residue Arithmetic on FPGAs," in *Proc. International Conference on Dependability of Computer Systems*, 2006, pp. 297–305.

[7] L. Parrilla, E. Castillo, A. Garcia, and A. Lloris, "Intellectual Property Protection for RNS Circuits on FPGAs," in *Proc. FPL*, 2004, pp. 1139–1141.

[8] M. Ciet, M. Neve, E. Peeters, and J.-J. Quisquater, "Parallel FPGA Implementation of RSA with Residue Number Systems," in *Proc. 46th IEEE International Midwest Symposium on Circuits and Systems*, 2003, pp. 806–810.

[9] Y. Wang, X. Song, M. Aboulhamid, and H. Shen, "Adder Based Residue to Binary Number Converters for $(2^n - 1, 2^n, 2^n + 1)$," *IEEE Trans. Signal Processing*, pp. 1772–1779, 2002.

[10] A. Garcia and A. Lloris, "RNS Scaling based on Pipelined Multipiers for Prime Moduli," in *Proc. IEEE Workshop on Signal Processing Systems*, 1998, pp. 459–468.

[11] G. Alia and E. Martinelli, "A VLSI Algorithm for Direct and Reverse Conversion from Weighted Binary NUmber System to Residue Number System," *IEEE Trans. Circuits Syst.*, pp. 1033–1039, 1984.

[12] R. Capocelli and R. Giancarlo, "Efficient VLSI Networks for Converting an Integer from Binary System to Residue Number System and Vice Versa," *IEEE Trans. Circuits Syst.*, pp. 1425–1430, 1988.

[13] S. Piestrak, "Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders," *IEEE Trans. Comput.*, pp. 68–77, 1994.

[14] Y. Wang, "New Chinese Remainder Theorems," in *Proc. 32nd Asilomar Conference on Signals, Systems and Computers*, 1998, pp. 165–171.

[15] C. Huang, "A Fully Parallel Mixed-Radix Conversion Algorithm for Residue Number Applications," *IEEE Trans. Comput.*, pp. 398–402, 1983.

[16] G. Dimauro, S. Impedovo, and G. Pirlo, "A New Technique for Fast Number Comparison in the Residue Number System," *IEEE Trans. Comput.*, pp. 608–612, 1993.

[17] O. Mencer, "ASC, A Stream Compiler for Computing with FPGAs," *IEEE Trans. Computer-Aided Design*, vol. 25, no. 9, pp. 1603–1617, Sept. 2006.

[18] M. Flynn and S. Oberman, *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.