FPGA Implementation of a Multiplier in MLRNS

Ratko Pilipovic[1], Zdenka Babic[2], Patricio Bulic[3]

[1,2]*University of Banja Luka, Faculty of Electrical Engineering Banja Luka, Bosnia and Hercegovina*
[3]*University of Ljubljana, Faculty of Computer and Information Science Ljubljana, Slovenia*
E-mail: [1]ratko.pilipovic@etfbl.net, [2]zdenka@etfbl.net, [3]patricio.bulic@fri.uni-lj.si

Abstract

Multiplication is one of the four elementary, mathematical operations on which are based many DSP applications. Since the multiplication dominates in many DSP operation, there is high demand for high speed multiplier. In this paper we try to implement multiplier in Residue Number System with multiple levels (Multi-Level Residue Number System - MLRNS) using FPGA. Since the moduli at each level have the simple form, conversions from weighted number system to MLRNS and vice-versa are simple and fast. Each high-to-low level conversion in MLRNS decreases dynamic ranges of operands and increases the level of parallelism. In other words, we can observe multiplication of large numbers as multiple parallel multiplications of smaller numbers in MLRNS.

Keywords — Residue number system (RNS), moduli sets, FPGA, parallelism, weighted-to-RNS conversion, large numbers, digital signal processing.

1 Introduction

Residue arithmetic is an ancient branch of mathematics, dated to the 3rd Century A.D. [1]. Addition and multiplication in Residue number systems (RNS) are carry-free operations and can be performed in parallel. A few decades ago, since these great advantages, RNS were investigated for use in the design of general purpose computers [2], and specialized RNS processors [3]. But, being a non-weighted number system, RNS suffers from complicated division and magnitude comparison. Moreover, conversion from weighted number system to RNS with arbitrary set of moduli and vice-versa is cumbersome.

Although these shortcomings prohibit its widespread use in general-purpose computer systems, because of the ability to perform high-speed concurrent arithmetic, the RNS has been proposed as a means to efficiently perform computations in digital signal processing (DSP) and other applications which require only multiplication and addition [4, 5, 6, 7]. Nowadays, RNS is becoming more and more popular for image processing in embedded systems, which have limited processing power that is constrained by power consumption [8, 9].

An RNS moduli base is a set $S = \{m_1, m_2, .., m_N\}$ composed of positive coprime integers. We say that m_l, m_k are coprime when their only common divisor is one. An unique RNS representation for every integer X belonging to the dynamic range $[0, M-1]$ defined by the product of all moduli $M = \prod_{i=1}^{n} m_i$, is given by RNS digits $X_i, i = 1, 2, ..., N$:

$$X = (\langle X \rangle_{m_1}, \langle X \rangle_{m_2}, ..., \langle X \rangle_{m_N}) \tag{1}$$

where $\langle X \rangle_{m_k}$ denotes arithmetic moduo of X by m_k

According to the Chinese Remaining Theorem (CRT) [1], conversion of RNS number representation to binary number system is given by:

$$X = \left\langle \sum_{i=1}^{N} \left\langle X_i \left\langle \hat{m_i}^{-1} \right\rangle_{m_i} \hat{m_i} \right\rangle_{m_i} \right\rangle_M \tag{2}$$

where $\hat{m_i} = \frac{M}{m_i}, i = 1, 2, .., N$ and $\hat{m_i}^{-1}$ denotes the multiplicative inverse of $\hat{m_i}$ modulo m_i.

Residue number systems with a lot of moduli are more efficient, because more moduli means smaller resulting dynamic range and greater level of parallelism. But, because of the constraint that moduli must be positive coprime integers, construction of moduli base is not a simple task. Because of simple conversion to and from RNS and arithmetic units, the triple moduli set $\{2^n - 1, 2^n, 2^n + 1\}$ have unprecedented importance in RNS. Main advantage of this set is simple and straight forward conversion, but main disadvantage lies in limited dymanic range. If large dynamic ranges are desirable, we must consider a RNS with more than three moduli set. Paper [10] presents a comparative study on different moduli set in RNS. We can divide all moduli sets in four groups, moduli sets with $3n$, $4n$, $5n$ and $6n$ dynamic range. Moduli sets with $3n$ dynamic ranges can be divided into two groups, ones that uses $2^n + 1$ channel and others that don't. Although RNS without $2^n + 1$ channel perform faster, the reverse converters are more complicated compared to RNS with $2^n + 1$ channel. For a faster multiplication and increase of dynamic range, RNS with four moduli set is introduced. The main disadvantages of this moduli set is complicated reverse conversion and usage of moduli is dependent with value of n. RNS with moduli with $5n$ dynamic range is used to improve residue arithmetic, but it also suffers from complicated reverse conversion. To improve reverse conversion, moduli set with

$6n$ dynamic range. As concerned residue arithmetic, they show acceptable delay.

This paper presents challenges of FPGA implementation of multiplier in Multi-Level Residue Number System (MLRNS). MLRNS uses the multiple binary-to-RNS conversions with several moduli sets in the same form $\{2^n-1, 2^n, 2^n+1\}$, presented in [11]. This set is used to overcome limited range of three-moduli and at the same time to overcome complexity of RNS-to-Binary and Binary-to-RNS conversion of four-, five- and six- moduli set.

2 Multi-level residue number system

Let $\langle X_i \cdot Y_i \rangle_{m_i}$ denotes modulo m_i multiplication in RNS, where X_i and Y_i represents RNS operands. For these operations, if dynamic ranges of inner RNS operands are known, it is possible to calculate dynamic range of inner RNS results before modulo m_i residue reduction, e.g. dynamic range of $\cdot$ operation results. Dynamic range of inner RNS operands is $[0, m_i - 1]$. In three moduli RNS with $S = \{2^n-1, 2^n, 2^n+1\}$ dynamic ranges of inner RNS operands are $[0, 2^n-2], [0, 2^n-1], [0, 2^n]$ per moduli respectively. For that case, maximally dynamic ranges of inner RNS results, before modulo m_i residue reduction, is $[0, 2^{2n}]$.

Let's define a RNS subsystem, $RNS^{(1)}$, with moduli set $S^{(1)} = \{2^{n^{(1)}}-1, 2^{n^{(1)}}, 2^{n^{(1)}}+1\}$. New moduli should be as small as possible, and it must not interfere with inner operand product. To achieve second goal, new moduli must respect constraint:

$$M^{(1)} = (2^{n^{(1)}}-1)(2^{n^{(1)}})(2^{n^{(1)}}+1) > 2^{2n} \quad (3)$$

Following Eq. 4 we can make relation between $n^{(1)}$ and n:

$$n^{(1)} = \left\lfloor \frac{2n}{3} \right\rfloor + 1 \quad (4)$$

The moduli from the set $S^{(1)}$ are smaller than the moduli from the set S. This procedure can be repeated, defining the $RNS^{(2)}$ as the subsystem of $RNS^{(1)}$, and so on, until we reach moduli of desired size. These multiple $RNS^{(k)} - to - RNS^{(k+1)}$, conversions lead to the *Multi-Level Residue Number System (MLRNS)*. Because of the moduli set in each RNS subsystems are in the form $\{2^n-1, 2^n, 2^n+1\}$, these $RNS^{(k)} - to - RNS^{(k+1)}$ and vice versa conversions are as simple as binary-to-RNS and RNS-to-binary conversions. By each such conversion, dynamic range of inner operands in RNS subsystems is decreased and level of parallelism is increased. For example, in MLRNS system with K levels, multiplication of X and Y can be done with 3^K multiplication, that can be executed in parallel, where operands have lower dynamic range than X and Y.

3 Proposed architecture

In this section we will go trough the architecture of MLRNS based multiplier. Proposed architecture is based on discussion presented in Section 2 with minor changes that were done in order to make it more suitable for FPGA. We designed three level MLRNS based multiplication, which is consisted of several moduli sets in the same form $\{2^n-1, 2^n, 2^n+1\}$. In three level MLRNS with moduli set in the form $\{2^n-1, 2^n, 2^n+1\}$, multiplication is replaced with $3^3 = 27$ fully parallel multiplications. MLRNS based multiplication can be done with multiple multiplications performed in parallel where operands have lower dynamic ranges than initial operands before MLRNS conversion. Having this in mind we can say that MLRNS based multiplication is suitable for implementing on FPGA, which is popular platform for parallel processing. Figure 1. illustrates multiplication in MLRNS system. X and Y represents input operands and Z represent output. First step in MLRNS based multiplication is Binary-to-MLRNS conversion of input operand. Result of Binary-to-MLRNS conversion are vectors $(X_1, X_2, ..., X_{27})$, $(Y_1, Y_2, ..., Y_{27})$ which are MLRNS representation of numbers X and Y, respectively. Multiplier conducts, in our case, 27 parallel multiplication, where operands have lower dynamic range than X and Y. Output of multiplier is vector $(Z_1, Z_2, ..., Z_{27})$ where $Z_i = X_i \cdot Y_i$. Resulting vector $(Z_1, Z_2, ..., Z_{27})$ is MLRNS representation of product $Z = X \cdot Y$, so as final step we need to apply MLRNS-to-Binary conversion on vector $(Z_1, Z_2, ..., Z_{27})$ to get product of X and Y.

3.1 Binary-to-MLRNS conversion

Figure 2. a) illustrates Binary-to-MLRNS conversion, where $n^{(i)}$ determines $\{2^{n^{(i)}}-1, 2^{n^{(i)}}, 2^{n^{(i)}}+1\}$, module set in i-th layer. To achieve that Binary-to-MLRNS doesn't have influence on multiplication, each MLRNS layer must respect condition given by Eq. 4. Architecture of Binary-to-MLRNS conversion is pretty simple, each output from one level of MLRNS conversion represents input for next level of MLRNS conversion. One level of MLRNS conversion, represents one or more Binary-to-RNS conversion that can be process in parallel, so the problem of implementing Binary-to-MLRNS conversion is reduced to problem implementing Binary-to-RNS conversion on FPGA. Let's write binary integer in the form:

$$X = A + B2^n + C2^{2n} \quad (5)$$

With such notation, three moduli RNS representation of number X is given by:

$$X_{RNS} = (X_1, X_2, X_3) = (\langle X \rangle_{2^n-1}, \langle X \rangle_{2^n}, \langle X \rangle_{2^n+1}) \quad (6)$$

where $X_1 = \langle A + B + C \rangle_{2^n-1}$, $X_2 = A$ and $X_3 = \langle A - B + C \rangle_{2^n-1}$ Problem of determining A, B and C can be done very easily in VHDL, taking first, second and third $n^{(i)}$-bit parts of input operand X.

Let's observe $(i+1)$-th level of Binary-to-MLRNS conversion, with the appropriate RNS representation $(\langle X_l \rangle_{2^{n^{(i+1)}}-1}, \langle X_l \rangle_{2^{n^{(i+1)}}}, \langle X_l \rangle_{2^{n^{(i+1)}}+1})$ of a number X_l which represents output from i-th level. Using $2^{n^{(i+1)}}$ basis on $(i+1)$-th level we can represent number X_l as $X_l = A + B2^{n^{(i+1)}} + C2^{2n^{(i+1)}}$. Based on Eq. 4, we can conclude that $2^{2n^{(i+1)}} > 2^{n^{(i)}}$, and also, according

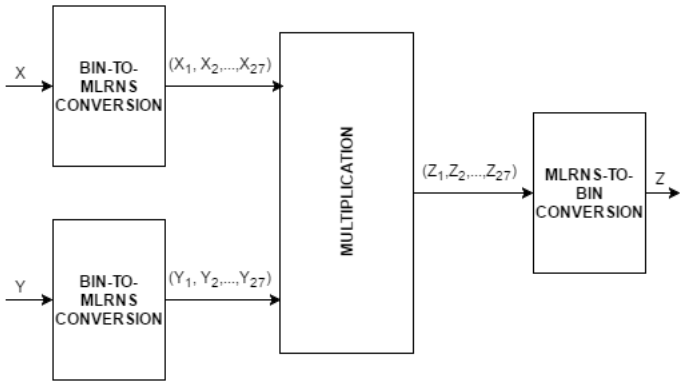

Figure 1: MLRNS based multiplier

to Section 2., we can say that maximal value for X_l is $2^{n^{(i)}}$. Having these two observation in mind, we can say that C is equal to 0 in Eq. 6 for every level of Binary-to-MLRNS conversion. Due to previous discussion, in order to implement Binary-to-RNS conversion, we only need to design block that calculates $\langle A+B\rangle_{2^{n^{(i)}}-1}$ and $\langle A-B\rangle_{2^{n^{(i)}}+1}$.

$$\langle A+B\rangle_{2^{n^{(i)}}-1} = \begin{cases} \langle (A+B-1)\rangle_{2^{n^{(i)}}} & \text{if } (A+B) \geq 2^{n^{(i)}} \\ (A+B), & \text{otherwise} \end{cases} \quad (7)$$

Implentation of Eq. 7 consists of adder, comparator and incrementer. Although this implementation is pretty simple, the main problem is conditional branching, which can cause very big delay. To overcome this problem we use Kogge-Stone parrallel prefix adder structure [12]. Parallel prefix adder is type of adder where carry bits are generated using parallel-prefix carry-computation unit. To use parallel-prefix adder as modulo adder we connect output carry bit with its input carry bit. Resulting structure is more known as EAC (End Around Carry) parallel prefix adder. Only disadvantage of this approach is double representation of zero, so we always have in mind that $2^{n^{(i)}}-1$, is equal to 0.

Following Eq. 5 finding residue of X by module $2^{n^{(i)}}+1$ can be expressed as $\langle A-B\rangle_{2^{n^{(i)}}+1}$. This expression can be written as follows:

$$\langle A+B\rangle_{2^{n^{(i)}}+1} = \begin{cases} (A+B-1)-2^{n^{(i)}} & \text{if } (A+B-1) > 2^{n^{(i)}} \\ (A+B-1), & \text{otherwise} \end{cases} \quad (8)$$

Looking at Eq.8 we can see that for residue generation we need to implent addition and subtraction with conditional branching. This method is time consuming and it's inefficient when A and B have great dynamic ranges. To overcome these disadvantages we use residue generators described in [13]. Proposed generator consist of full adders and Augmented Diminished-1 adder. In the diminished-1 representation, X is represented by $x_z X^*$ where x_z is single bit, known as zero indication bit and X^* is $n^{(i)}$-bit vector known as number part. If $X > 0$ the $X^* = A-1$ and $a_z = 0$, whereas if $X = 0$ then $X^* = 0$ and $x_z = 1$. Working with diminished-1 notation eases and simplifies finding residue by module $2^{n^{(i)}}+1$, than normal weighted representation. At first level of residue generation we translate the A and B into theirs diminished-1 forms A^* and B^*. Diminished-1 form A^* and B^* serve as input operand for augmented Diminished-1 adder which consist of parallel prefix IEAC (Inverted End Around Carry) adder and AND logical circuit with multiple inputs. Logical AND circuit detects when the residue is equal to $2^{n(i)}$, while in other cases the output is equal to 0. Resulting residue is made by concatenation between output of AND logical circuit and the output of of parallel prefix IEAC adder.

Proposed implementation of Binary-to-MLRNS consist of combinatorial logic, and doesn't need external synchronization. For this approach we used structural design, where building blocks represent Binary-to-RNS conversion. In this conversion we achieve local and global parallelism. Local parallelism consist of parallel residue generation by moduli set $\{2^n-1, 2^n, 2^n+1\}$ inside RNS-to-Binary conversion. Global parallelism is achieved by parallel execution of Binary-to-RNS conversion inside one layer, e.g. in our case we can achieve 9 parallel Binary-to-RNS conversion at third level of MLRNS.

3.2 MLRNS-to-Binary conversion

Figure 2 b) illustrates MLRNS-to-Binary conversion, where $n^{(i)}$ determines $\{2^{n^{(i)}}-1, 2^{n^{(i)}}, 2^{n^{(i)}}+1\}$ module set, in i-th layer. Function of this block is transformation of vector $(Z_1, Z_2, ..., Z_{27})$ into its appropriate binary representation Z. As we stated before we designed three level MLRNS based multiplication, so our conversion block consist of three step RNS-to-Binary conversion. Architecture of MLRNS-to-Binary conversion is pretty similar to architecture of Binary-to-MLRNS conversion although they have inverse functions. Outputs from i-th layer represents RNS representation for moduli set $\{2^{n^{(i+1)}}-1, 2^{n^{(i+1)}}, 2^{n^{(i+1)}}+1\}$ in $(i+1)$ layer. Based on described architecture, we only need to implement block that converts RNS-to-Binary which is main building block of proposed MLRNS-to-Binary conversion block.

Let's examine dynamic range of output numbers in $(i+1)$-th layer. These numbers belong to dynamic range $[0, M^{(i)}-1]$ where $M^{(i)} = 2^{n^{(i)}}(2^{n^{(i)}}+1)(2^{n^{(i)}}-1)$,

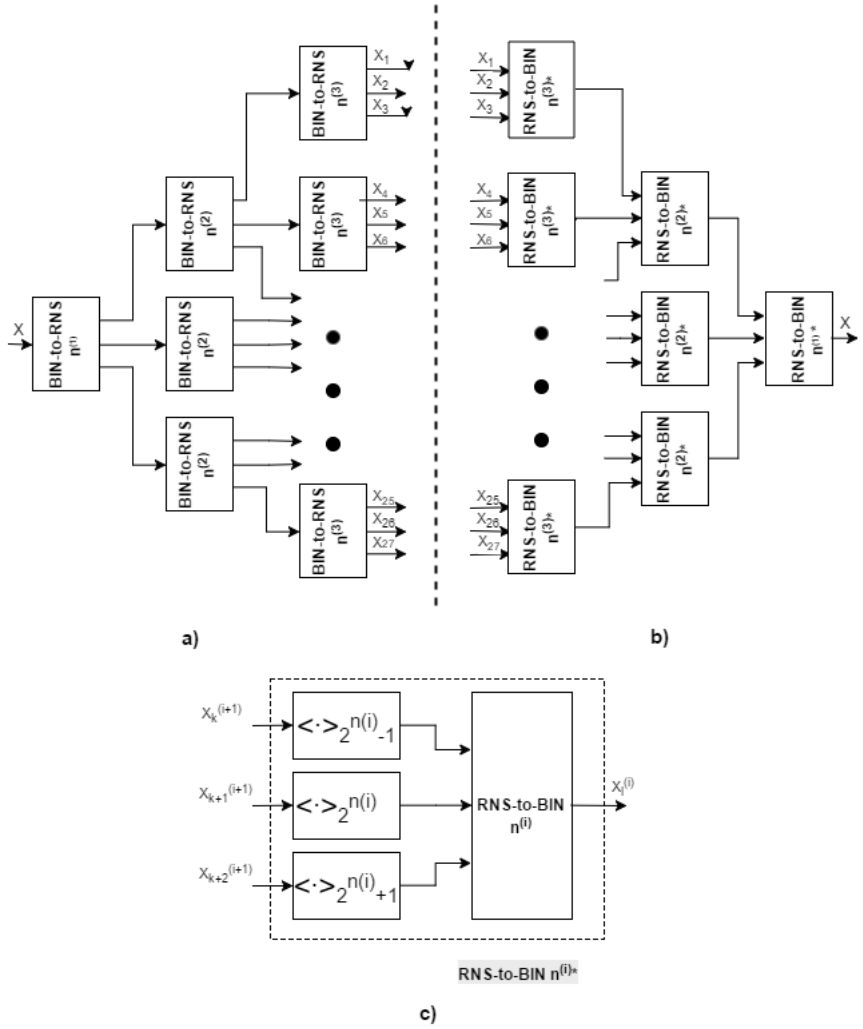

Figure 2: Block sheme: a) Bin-to-MLRNS conversion b) MLRNS-to-bin conversion c) Modified RNS-to-Binary conversion

which is approximately equal to $2^{3n^{(i)}}$. Analyzing dynamic range of output number from $(i+1)$-th layer and having in mind Eq. 4, we can conclude that output of $(i+1)$-th layer has greater dynamic range than input of i-th layer. Because of this it's necessary to find residue by moduo $\left\{2^{n^{(i)}}-1, 2^{n^{(i)}}, 2^{n^{(i)}}+1\right\}$, before applying RNS-to-Binary conversion. Modification of RNS-to-Binary is illustrated on Figure 2 c).

The traditional implementation of RNS-to-Binary conversion is based on inverse CRT (*Chinese Remainder Theorem* conversion based on Eq. 2 and multiplicative inverses that can be expresed as $\hat{m_1} = 2^n(2^n+1)$ $\hat{m_1}^{-1} = 2^{n-1}$, $\hat{m_2} = 2^{2n}-1$ $\hat{m_2}^{-1} = 2^n - 1$ and $\hat{m_3} = 2^n(2^n-1)$ $\hat{m_3}^{-1} = 2^{n-1}+1$. Analyzing Eq. 2 and the forms of multiplicative inverses, for implementing inverse CRT we need to perform multiple modulo additions, modulo subtractions, and bit shifting. This kind of implementation requires many operators with long latencies and heavy resource cost. To avoid these problems we implemented RNS-to-Binary conversion described in [14]. In [14] is presented highly efficient and general purpose RNS-to-Binary converter which is implemented using bitwise arithmetic. General idea of this work is based on following properties:

$$\langle -v \rangle_{2^n-1} = \overline{v_{n-1}v_{n-2}...v_1v_0} \tag{9}$$

$$\langle 2^p v \rangle_{2^n-1} = v_{n-p-1}v_{n-p-2}...v_1v_0v_{n-1}v_{n-2}...v_{n-p} \tag{10}$$

$$\langle 2^p v \rangle_{2^n-1} = v_{n-p-1}v_{n-p-2}...v_1v_0v_{n-1}v_{n-2}...v_{n-p} \tag{11}$$

$$\left\langle 2^{n+1} v \right\rangle_{2^n-1} = v_{n-1}v_{n-2}...v_1v_0v_{n-1}v_{n-2}...v_1v_0 \tag{12}$$

where v represents number with its binary representation $v_{n-p-1}v_{n-p-2}...v_1v_0$. Based on properties expressed by Eq. 9. - 12., X can be calculated from its RNS represantation (X_1, X_2, X_3) as follows:

$$X = \langle v_1 + v_{21} + v_{22} + v_3 \rangle_{2^{2n}-1} + X_2 \tag{13}$$

where v_1, v_{21}, v_{22}, and v_3 are calculated as follow:

$$v_1 = \langle -2^n X_2 \rangle_{2^{2n}-1} \tag{14}$$

$$v_{21} = \left\langle 2^{n-1} X_1 \right\rangle_{2^{2n}-1} \tag{15}$$

$$v_{22} = \left\langle -2^{2n-1} X_1 \right\rangle_{2^{2n}-1} \tag{16}$$

$$v_3 = \left\langle 2^{n-1}(2^n+1)X_1 \right\rangle_{2^{2n}-1} \quad (17)$$

Having in mind equations expressed by Eq.13 - 17, RNS-to-Binary conversion can be design with 2 complement operation, 4-left shift operations and 1 concatenation operation.

Proposed architecture is entirely based on combinatorial logic, and doesn't need external synchronization. For this approach we used structural design, where building blocks represent RNS-to-binary conversion. These blocks are depicted in Figure 2 c), and are connected as depicted in Figure 2 b). Like in Binary-to-MLRNS, we achieve a great level of parallelism using simultaneous RNS-to-Binary conversion inside one layer.

4 Experimental results

In this section we will first shortly describe used test prototype and then will briefly explain its building parts. In order to evaluate our implementation we will investigate hardware utilization and delay. It is of great importance to evaluate hardware utilization in order to observe overall scalability of proposed approach. On the other hand goal of MLRNS approach is speeding up the arithmetic operations, so it's of the crucial importance to observe the delay.

Our test prototype is the Digilent Atlys Spartan-6 Trainer board equipped with Spartan 6 XC6SLX45 FPGA chip. This board contains 128 MB DDR2 and 16 MB ROM. Spartan 6 XC6SLX45 FPGA chip contains 43,661 Logic, 6,822 CLB (Complex Logic Blocks) slices, 54,576 FF's, 18 Kb RAM, and with 58 DSP slices. LUT represents a flexible resource capable of implementing logic functions to six inputs, small ROM, small RAM and shift registers. Unlike LUTs which function can be programmed by user, DSP slices are specially designed for fast multiplication and addition. These unit can compute functions that are in form $P = (A+D)*B$, $P = P' + C$ and many similar functions. It is also capable of SIMD processing, implementing 2 or 4 shorter addition/subtraction/ accumulation operations of 24 or 12 bits, respectively.

Table 1: Analysis of resource usage and delay of MLRNS based multiplier

n	**Resource usage**		**Delay[ns]**	
	LUTs	DSP slices	Logic	Propagation
32	4264	9	37.399	143.112
64	8708	27	50.631	187.084
128	28074	32	62.233	243.441

Table 1. shows hardware resource utilization and delay of MLRNS multiplier in dependence of dynamic range of product $X \cdot Y$. Both input operands X and Y are represented with n bits. For hardware resources we analyzed number of used LUTs and DSP slices. LUTs are used for building combinatorial logic for implementing MLRNS forward and reverse conversion. In this work we tried to emphasize on MLRNS so we used DSP slices for multiplication of appropriate MLRNS representation X and Y.

Looking at the hardware utilization of MLRNS multiplier we conclude that system is very demanding. Efficiency of hardware utilization is decreased due to used board which was available at the time. The Atlys board is mainly used for introducing FPGA technologies for students. Furthermore we didn't used any optimization in order to contribute scalability of our system.

The most interesting part is delay. We differ delay that is caused by logic and delay that is caused by signal propagation between building FPGA elemensts in forward and reverse conversion. Analyzing the delay in Table 1. we can conclude that about 80% of delay is caused by signal propagation, while 20% of delay is caused by used logic. From aspect of maximal speed we succeeded in creating high speed logic for fast multiplying, although we have problems with signal propagation. This situation is natural consequence of FPGA design which consists of CLBs which are interconnect by connection matrix. Our design, in general, will be spread on multiple CLBs so the communications between CLBs causes delay. This is sign to map our implementation into ASIC structures.

5 Conclusion

In this paper we present a FPGA implementation of multiplier in Multilevel Residue Number System. Because of constraint that moduli in the standard RNS must be positive coprime integers, construction of more efficient RNS with lot of moduli is not a simple task. RNS with lot of moduli has larger dynamic range, but suffers from complex and slow realizations of both to and from RNS conversions. For MLRNS based multiplier implementation we use RNS with $\{2^n-1, 2^n, 2^n+1\}$ moduli set as base for constructing MLRNS. To increase level of parallelism and decrease dynamic range of operands we only need to add new level to MLRNS. The number of levels in MLRNS and sets of moduli should be designed in accordance to the dynamic range of product factors.

Based on FPGA, we developed an multiplier based on Three-Level MLRNS. Although experimental results show high usage of hardware resources and relative big delay, proposed approach has great potential to become efficient solution for design high speed multipliers. This presumption is based on fact that in MLRNS common multiplication is done as several multiplication executed in parallel with operands that have lower dynamic range than inital product factors. This feature of MLRNS will be visible when we deal with large numbers multiplication. This potential is also confirmed by logic delay of implemented multiplier.

Future improvements would contribute to hardware scalability and faster multiplication allowing multiplier to use all key-advantages of MLRNS. In future work we will explore further optimization of proposed multiplier and try to map it into ASIC structures.

Acknowledgments

This research was supported by Slovenian Research Agency (ARRS) under Grants P2-0359 (National research program Pervasive computing), and by Ministry of Civil Affairs, Bosnia and Herzegovina, under grants BI-BA/16-17-029 (Bilateral Collaboration Project).

References

[1] J. H. McClellan and C. M. Rader, "Number theory in digital signal processing," 1979.

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

[3] V. Paliouras and T. Stouraitis, "Multifunction architectures for rns processors," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 46, no. 8, pp. 1041–1054, 1999.

[4] E. B. Olsen, "Introduction of the residue number arithmetic logic unit with brief computational complexity analysis," *arXiv preprint arXiv:1512.00911*, 2015.

[5] G. C. Cardarilli, A. Nannarelli, M. Petricca, and M. Re, "Characterization of rns multiply-add units for power efficient dsp," in *2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1–4, IEEE, 2015.

[6] D. R, B. V, S. K. Sahoo, N. R. Samhitha, N. A. Cherian, and P. M. Jacob, "Implementation of floating point mac using residue number system," in *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*, pp. 461–465, Feb 2014.

[7] C. H. Chang, A. S. Molahosseini, A. A. E. Zarandi, and T. F. Tay, "Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications," *IEEE Circuits and Systems Magazine*, vol. 15, pp. 26–44, Fourthquarter 2015.

[8] W. Wang, M. Swamy, and M. O. Ahmad, "Rns application for digital image processing," in *System-on-Chip for Real-Time Applications, 2004. Proceedings. 4th IEEE International Workshop on*, pp. 77–80, IEEE, 2004.

[9] D. K. Taleshmekaeil and A. Mousavi, "The use of residue number system for improving the digital image processing," in *IEEE 10th INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING PROCEEDINGS*, pp. 775–780, IEEE, 2010.

[10] D. Younes and P. Steffan, "A comparative study on different moduli sets in residue number system," in *Computer Systems and Industrial Informatics (ICCSII), 2012 International Conference on*, pp. 1–6, Dec 2012.

[11] Z. Babic, "Recurent rns and its application to digital signal processing," in *XXXVIII Etran confernce paper antology*, pp. 109–110, ETRAN, June 1994.

[12] H. T. Vergos, C. Efstathiou, and D. Nikolos, "Diminished-one modulo 2 n+ 1 adder design," *IEEE Transactions on Computers*, vol. 51, no. 12, pp. 1389–1399, 2002.

[13] H. T. Vergos and D. Bakalis, "On the use of diminished-1 adders for weighted modulo 2n+ 1 arithmetic components," in *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on*, pp. 752–759, IEEE, 2008.

[14] B. Liu, H. Fu, L. Gan, W. Zhao, and G. Yang, "Optimizing residue number reverse converters through bitwise arithmetic on fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pp. 236–243, IEEE, 2015.