Exploiting Residue Number System for Power-Efficient Digital Signal Processing in Embedded Processors*

Rooju Chokshi[1], Krzysztof S. Berezowski[2,3], Aviral Shrivastava[2], Stanisław J. Piestrak[4]
[1]Microsoft Corporation, Redmond, WA, USA
[2]Computer Science and Engineering, Arizona State University, Tempe, AZ, USA
[3]TIMA Laboratory, 38031 Grenoble, France
[4]LICM, University of Metz, 57070 Metz, France
rooju.chokshi@microsoft.com
{krzysztof.berezowski,aviral.shrivastava}@asu.edu
piestrak@univ-metz.fr

ABSTRACT

2's complement number system imposes a fundamental limitation on the power and performance of arithmetic circuits, due to the fundamental need of cross-datapath carry propagation. Residue Number System (RNS) breaks free of these bonds by decomposing a number into parts and performing arithmetic operations in parallel, significantly reducing the breadth of carry propagation. Consequently, RNS arithmetic has been proposed as a solution to improve the power-efficiency of arithmetic hardware. However, limitations of the expressiveness of RNS in terms of arithmetic operations together with overheads related to interaction with 2's complement arithmetic make programmable processor design that takes advantage of these benefits challenging. In this paper we meet this challenge by multi-tier synergistic co-design of architecture, micro-architecture, hardware components, as well as compilation techniques. Our experiments not only demonstrate simultaneous improvement of up to 30% in performance and 57% reduction in functional unit power consumption, but also that most of these benefits can be exploited with automatically generated code.

Categories and Subject Descriptors

C.1.m [**Processor Architectures**]: Miscellaneous; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms

Performance, Design, Algorithms

*This work was supported by Consortium for Embedded Systems, Arizona State University, Tempe, AZ, USA; National Science Foundation through grant number CCF-0916652; as well as was by European Commission through FP6 grant number MOIF-CT-2006-040672.

Keywords

Residue Number System, Processor, Compiler, Power, Performance

1. INTRODUCTION

The omnipresence of personal wireless communication and multimedia applications drives the demand for high performance, power efficient, and low cost embedded processors with rich digital signal processing (DSP) capabilities. Computationally dominated by discrete convolution, embedded DSP eventually boils down to the power efficiency of algorithms dominated by interleaved series of additions and multiplications. In this task, 2's complement arithmetic suffers from the fundamental limitation in delivering power efficient computations. First, high-performance logarithmic delay addition comes at the expense of exponential hardware complexity. Second, in multiplication the partial product reduction contributes quadratic hardware complexity and linear delay to the final 2-operand addition.

The exploration of number representations more suitable for high-performance power-efficient DSP has identified the Residue Number System (RNS) as a promising alternative for application specific DSP hardware [10, 18, 20]. In RNS an integer $X \in [0, M)$ can uniquely be represented by a set of residues $\{x_1, ..., x_l : \forall_{i=1}^{l} x_i = |X|_{P_i}\}$ characterized by the *moduli set* $P = \{P_1, ..., P_l\}$ of relatively prime integers, where $M = \prod_{i=1}^{l} P_i$ is system's *dynamic range*. Consequently, operations like $\circ \in \{+, -, \times\}$ on operands $X = \{x_1, ..., x_l\}$ and $Y = \{y_1, ..., y_l\}$ can be performed as $X \circ Y = \{|x_1 \circ y_1|_{P_1}, ..., |x_l \circ y_l|_{P_l}\}$, i.e. on smaller, independent, and parallel *channels*. Such decomposition greatly reduces *both* the length of the carry propagation chain in addition, as well as the size of partial product matrices (PPMs) in multiplication, leading to remarkable performance improvements along with area and power savings.

Along with these great advantages RNS manifests certain limitations. Magnitude comparison and division do not lend themselves to the same distributed and parallel computational structure as addition and multiplication [1, 19]. Transcoding results into 2's complement representation for that leverage engages computationally intensive *reverse conversion* [21]. Therefore, designing a programmable processor using RNS is a challenging task as limitations make an RNS-only processor too restrictive, while a hybrid processor that supports both 2's complement and RNS arithmetic may suffer from conversion overheads that can eventually outdo all advantages of fast and low-power computational units. Consequently, there have been very few previous approaches to RNS-based processor design, which are mostly focused on power and area reduction rather than on improving applications' performance [4, 9].

In this work we address the challenge of exploiting the power and performance benefits of RNS arithmetic in a RISC processor in a multi-tier manner. First, through a series of synergistic instruction-set-architecture, microarchitecture, and component design decisions we expose the conversion overheads to the software through instruction set augmentations. Second, we formulate the instruction selection problem for code generation for RNS processors, and perform instruction scheduling to hide the conversion overheads and optimize performance. For practical demonstration purposes, we extend the Simplescalar ARM simulator with RNS instructions, synthesize RNS components, and add their delay and power parameters into the modified simulator. Finally we enhance the GCC for ARM with the code generation and instruction scheduling optimizations and evaluate power and performance of several DSP benchmarks. As a result, we observe 21% improvement in performance and 51% reduction in functional unit power consumption during the execution of the compiled code.

The major novel contributions of this paper include:

1. a codesigned, optimized, architecture, microarchitecture component design, and compilation techniques for an RNS extension to the embedded RISC processor,
2. a design of an RNS multiplier that exploits properties of redundant representation utilized by the extension,
3. a formulation and solution of an application mapping problem on the proposed RNS extension that includes both instruction selection and instruction scheduling,
4. an extensive experimental evaluation demonstrating the undeniable performance gain and power efficiency.

The rest of this paper is organized as follows. First in Section 2 we summarize related work both in the area of RNS hardware component design, as well as few papers on RNS programmable processor design. Then, in Section 3 we discuss Instruction Set Architecture (ISA) extension. In Section 4 we elaborate on microarchitectural decisions. In Section 5 we give details of our RNS hardware components design. In Section 6 we discuss our synthesis and simulation setup. In Section 7 we introduce our novel compilation techniques. In Sections 8 and 9 we discuss our experimental results performed on SimpleScalar simulator across the set of benchmark applications and design corners. Finally, we conclude our findings in Section 10.

2. RELATED WORK

A volumenous body of RNS-related literature focusing on fast and power efficient design of computational and conversion units has recently been surveyed in [1, 12]. Furthermore, many high-performance RNS-based ASICs have been developed [10, 12, 18], as application specific hardware has long been believed to provide a natural ecosystem for RNS-based circuitry. However, these non-programmable architectures address different challenges than those of designing programmable RNS-based processors.

Although the idea of designing an RNS-based RISC processor was first stipulated almost 20 years ago [7], it has received very little attention so far. Ramirez et al. [16] have developed a 32-bit pure RNS SIMD architecture strictly tailored for DSP applications. It utilizes RNS addition, subtraction, and multiplication, with conversion units hardwired into the pipeline. With an advantage of being ISA compatible, such a design deals with conversion overheads only at the hardware level disallowing any speculative approaches through application mapping. Recently Chavez et al. [4] proposed more complete architecture of a RNS-based RISC DSP with composite multiply-and-accumulate block to perform additions, multiplications, and multiply-and-accumulate operations. However, in that work, the primary focus was on power and chip area reduction, rather than performance. In the follow-up paper they introduced the concept of *balanced* moduli set and the considerations about the implications on the multiplier design [5].

As the consequence of the absence of a programmable RNS processor, there has been no publicized work on developing compiler techniques for such architectures. The problem is quite similar to that of compilation in the presence of Instruction Set Extensions (ISEs) on which significant work has been done, summarized in [6]. However, while ISEs are composed of complex domain-specific operators optimized in hardware and exposed to the compiler through ISA, RNS extension improves performance of atomic operations and as such can be considered orthogonal to ISEs. Consequently, instruction set extension can still be defined over the top of RNS extension to better exploit its benefits in specific application domain — in fact the design of Chavez et al. [4] can be classified as such. However our problem differs from the traditional ISE problem where the objective is to find *maximal convex sub-graphs* of the application's dataflow graph, while compiling for the RNS extension boils down to finding *maximal connected sub-graphs* which is a problem of significantly lower computational complexity.

3. INSTRUCTION SET ARCHITECTURE

A first-cut approach to integrate RNS functional units into the processor pipeline is to pad their inputs and outputs with conversion logic [4]. The undeniable advantage of such approach is that all the necessary changes remain transparent to the software. However, it implies that every fast RNS operation involves necessary forward and reverse conversions. Since conversions are essentially costly and parasitic, the benefits of faster computations may eventually be overshadowed by these overheads. Consequently, the only choice is to separate computations and conversions. Hence we add separate instructions for RNS computations (***RADD***, ***RSUB***, ***RMUL***), and separate instructions for converting operands from 2's complement binary to RNS (***FC***), and vice versa (***RC***). The hope is that the conversion operations may be scheduled in parallel with some other computation to effectively hide the conversion latency.

4. MICROARCHITECTURE DECISIONS

Multi-operand addition (MOA) using Carry-Save Adders (CSA) plays a crucial role in the design of RNS functional units [17]. CSA-based MOA produces separate carry C and sum S vectors, that are typically reduced by a final 2-operand (modulo) adder to produce the final residue $x =$

$|2C+S|_P$. However, due to end-around carry propagation 2-operand modulo adders of width m are slower and consume more area and power than regular adders of same width [1]. Consequently, the advantage of datapath decomposition into narrower channels gets partially consumed by such limitations. It is obvious though, and widely exploited in application specific hardware [14, 15], that $2C + S$ is the value congruent to the residue R. Therefore, computations on (S, C) pairs can be aggregated and the final modulo addition can be delayed until the actual value of the residue is required [17]. Hence, if we choose to represent operands as (S, C) pairs, we may benefit from simultaneous speed, area, and power improvements at the expense of representation redundancy. This is an important consideration since although it reduces the complexity of the adder and the forward converter, such a solution introduces additional complexity to the multiplier and reverse converter as later discussed in Section 5. Eventually though, forward conversion is required per input operand, additions are typically much more frequent than multiplications, and reverse conversions are rather infrequent. Therefore, as this work demonstrates, such a representation boosts the profit of the RNS extension. The storage of double-width operands can be achieved with no or limited micro-architectural changes, as many RISC processors allow registers and memory accesses as double precision storage spanning two registers or memory locations at the possible expense of additional load/store cycles.

5. COMPONENT DESIGN

5.1 Moduli Selection

Moduli selection is a crucial decision in designing efficient RNS arithmetic and conversion units. Obviously, increasing the number of moduli uniformly increases the speed and reduces the complexity/power consumption of computational units [14]. Moreover, it provides quite an unique opportunity of conducting fast and low power computations on large dynamic ranges which finds its use in certain applications e.g. cryptography [3]. Unfortunately, it also considerably increases the complexity of the implementation of the reverse conversion. We have previously demonstrated [14, 15] that in the case of application-specific DSP hardware such an overhead actually is dissolved in the advantages of using small computational units, since the reverse converter can be arbitrarily pipelined to match the speed of computational stages, and power and speed advantages of very narrow channels make up for both the advantages as well as the pipelining overhead. On the other hand, for smaller dynamic ranges, the set $\{2^n - 1, 2^n, 2^n + 1\}$ is widely used as it enjoys certain properties that make the design of both computational units as well as the reverse converter unit uniformly efficient. However, due to the differences in $2^n - 1$, 2^n, and $2^n + 1$ channel implementations, this set is not *well-balanced* [5], i.e. typically features a significant amount of slack between the fastest 2^n and the slowest $2^n + 1$ channel. It was relatively straightforward to conclude that to avoid that disadvantage faster channels should operate on more input bits. Consequently, the moduli set $\{2^n - 1, 2^k, 2^n + 1\} : k > n$ as well as some amendments to the modulo $2^n + 1$ multiplier were proposed in [5] to address this issue. While we are facing the same challenge, our microarchitectural choice to use (C, S) pairs for internal representation allows us to mitigate multiplier imbalance more systematically (see discussion in Section 5.5). Besides, extending the concept of balancing we evaluate moduli sets $\{2^n - 1, 2^k, 2^n + 1\} : k > n$ not only for inter-channel balance in the multiplier, but also for balancing the delays between the critical path pipeline components in an attempt to efficiently utilize the slack in between their implementations. In this part of our exploration we target to balance the multiplier and reverse converter delays as they are much larger than that of the adder, and as such their speeds decide the system clock cycle. Then we are reusing the remaining slack in other faster RNS components armoring them with the capacity of performing more operations within a single clock cycle (e.g. 3-operand RNS addition).

5.2 Periodicity Property

We employ the periodicity property of RNS arithmetic to generate fast, efficient, and regular designs of RNS functional units. Both $2^n - 1$ and $2^n + 1$ moduli enjoy the periodicity of sequences of residues $|2^i|_{2^n \pm 1}$ with respect to n [17]. I.e. $\forall i, j \in N \cup \{0\}, n \in N : i < n$

$$\left|2^{i+nj}\right|_{2^n-1} = 2^{|i+jn|_n} = 2^i \tag{1}$$

$$\left|2^{i+nj}\right|_{2^n+1} = (-1)^j 2^{|i+jn|_n} = (-1)^j 2^i \tag{2}$$

Eq. (1) and (2) and the distributivity of $|\cdot|$ over modulo addition imply that in a binary weighted representation, bits at positions $i + nj$ are equivalent in weight (with respect to sign in case of $|\cdot|_{2^n+1}$ operations), when considered for modulo operations. Thus bits of equivalent weight can be arranged in width n end-around carry CSA structures [17].

5.3 Binary-to-RNS (Forward) Conversion

Forward converters compute residues of the 32-bit input integer X and produces a pair (S, C) for each channel. $|X|_{2^k}$ is as simple as a slice of k least significant bits of X with carry vector $C = 0$. For $2^n \pm 1$ channels we directly apply the technique from [17], i.e. the periodicity allows us to chop X into p slices of length n. Hence from Eq. (1) and (2):

$$|X|_{2^n-1} = \left|\sum_{i=0}^{p-1} 2^{in} X_i\right|_{2^n-1} = \left|\sum_{i=0}^{p-1} X_i\right|_{2^n-1}$$

$$|X|_{2^n+1} = \left|\sum_{i=0}^{p-1} 2^{in} X_i\right|_{2^n+1} = \left|\sum_{i=0}^{p-1} (-1)^i X_i\right|_{2^n+1}$$

Thanks to that, adding the terms X_1 through X_p is a straightforward application of end-around carry CSA tree that forms a Multi-Operand Modulo Adder (MOMA) [17]. In the second equation, for each negative vector, for each bit position converter employs the trivial equality:

$$-b_i 2^i = -(1 - \bar{b}_i)2^i = \bar{b}_i 2^i - 2^i \quad : b_i \in \{0, 1\}$$

Consequently, bits in the negative terms are inverted for the CSA addition, and for each inversion we accumulate a correction of -2^i for the inverted bit at position i. Since there is always a constant number of inversions in the CSA tree, the modulo of resultant correction is added as extra layer of full-adder cells [17]. In fact, adding a constant binary vector in this layer allows for further constant-0 or constant-1 simplifications of full-adder cells according to bits of the

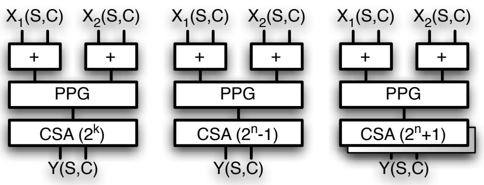


Figure 1: RNS Channel Multipliers

correction. It is important to note that the choice of representing operands in (S, C) form obviates the need to add a final 2-operand modulo adder stage for combining S and C.

5.4 RNS Adder

The high speed of RNS addition allows us to design a 3-operand RNS adder that is still significantly faster than its 2-operand 2's complement counterpart (Table 2). Obviously, for every channel it adds $X_1 = (S_1, C_1)$, $X_2 = (S_2, C_2)$, and $X_3 = (S_3, C_3)$, and produces $Y = (S, C)$. Therefore in fact, the adders for the $2^n \pm 1$ channels are structurally identical to the respective forward converters with $p = 6$ including the correction layer for the $2^n + 1$ channel. Again, the adder for the 2^k channel does not require end-around carry propagation and is a CSA tree in with all bits of weight greater than 2^k discarded. The 2-operand subtraction can be implemented by augmenting the inputs of the adders with logic to conditionally negate-and-correct the second operand, based on a control signal activated by the RSUB instruction.

5.5 RNS Multiplier

Multiplication is the first component for which even under redundant (S, C) representation delay varies with the input size. Hence it is also the first component to exercise the trade-offs springing out from the redundant representation as well as n and k selection. There are three architectural choices to consider. Carrying out the multiplication directly on the redundant (S_1, C_1) and (S_2, C_2) operands is definitely overly expensive as it roughly quadruples the size of partial product matrices. Computing residues $|S_1 + 2C_1|_{2^n-1}$ and $|S_2 + 2C_2|_{2^n+1}$ involves fairly expensive 2-operand modulo addition particularly unfavorable in the case of $2^n + 1$ modulus. However, we can take advantage of the identity $\big||X|_P * |Y|_P\big|_P = |X * Y|_P$, and produce intermediate operands in the form of $X'_1 = S_1 + 2C_1$ and $X'_2 = S_2 + 2C_2$. Hence we can use regular adders and save on modulo additions at the expense of one additional row and column of the PPM (Fig. 1). A by-product of that choice is also that it also efficiently reduces the imbalance between $2^n - 1$ and $2^n + 1$ channel delays, since it does not exercise the imbalance between the modulo 2^n+1 and 2^n-1 2-operand adders. In fact, with this design, $2^n - 1$ and $2^n + 1$ channels of the multiplier are structurally almost identical and the only imbalance is caused by the correction layer of $2^n + 1$ channel. Finally, the implementation is straightforward. Partial products are generated for each channel then aligned according to periodicity and added in an end-around CSA tree. As in the modulo $2^n + 1$ adder, the modulo $2^n + 1$ multiplier needs a fixed correction. Obviously in the 2^k channel bits at positions $i \geq k$ are being discarded throughout CSA addition.

With the design of a multiplier, we evaluate alternative moduli sets for balance. The RNS multiplier delays in different $\{2^n - 1, 2^k, 2^n + 1\}$ 32-bit dynamic range configurations are extracted into Table 1. Using the delay span across channels as the balancing metric, we selected the moduli set $\{2^n - 1, 2^k, 2^n + 1\} = \{2^9 - 1, 2^{15}, 2^9 + 1\}$ for implementation. Some additional discussion of the interaction between the multiplier and the reverse converter will also be given in the following section.

Table 1: Multiplier delays for different moduli sets

Moduli set	Multiplier delay (ns)	Del.span
$(2^9 - 1, 2^{18}, 2^9 + 1)$	$2.10, 2.55, 2.50$	0.45
$(2^8 - 1, 2^{16}, 2^8 + 1)$	$2.10, 2.40, 2.80$	0.70
$(2^9 - 1, 2^{15}, 2^9 + 1)$	$2.10, 2.20, 2.50$	0.40
$(2^8 - 1, 2^{17}, 2^8 + 1)$	$2.10, 2.45, 2.80$	0.70

5.6 RNS-to-Binary (Reverse) Conversion

RNS to binary conversion boils down to the hardware implementation of the equation stated by the Chinese Remainder Theorem (CRT). For our design we start from the so-called *new* formulation [21] of the CRT which states that given a set of residues $\{x_1, ..., x_l\}$ for the moduli set $\{P_1, ..., P_l\}$, the integer number X can be calculated as follows:

$$X = x_1 + P_1 \cdot \left| m_1(x_2 - x_1) + \sum_{i=2}^{l-1} m_i \Big(\prod_{j=2}^{i} P_j\Big)(x_{i+1} - x_i) \right|_{\prod_{j=2}^{l} P_j}$$

such that:

$$|m_1 P_1|_{P_2 P_3 \ldots P_l} \equiv 1, \ldots, |m_{l-1} P_1 P_2 \ldots P_{l-1}|_{P_l} \equiv 1$$

For $l = 3$, we have,

$$X = x_1 + P_1 \cdot |m_1(x_2 - x_1) + m_2 P_2 (x_3 - x_2)|_{P_2 P_3} \qquad (3)$$

such that

$$|m_1 P_1|_{P_2 P_3} \equiv 1, |m_2 P_1 P_2|_{P_3} \equiv 1$$

It is straightforward to observe that the assignment of moduli such that $P_1 = 2^k$ allows for the computation of X as a concatenation of the residue x_1 and the value Y:

$$Y = |m_1 x_2 - m_1 x_1 + m_2 P_2 x_3 - m_2 P_2 x_2|_{P_2 P_3} \qquad (4)$$

Similar considerations as in the case of the multiplier make us conclude that due to the redundant (C, S) representation, reverse converter has to be armored with input adders. However, unlike in the case of the multiplier, now we choose modulo adders that provide proper residues for each channel (Fig. 2) in order to allow for more aggressive optimization of the CRT equation as discussed below.

The hardware implementation of various forms of the CRT equation for $\{2^n - 1, 2^k, 2^n + 1\}$ moduli set for $k \neq n$ has been considered in the literature for both $k > n$ [5] and $k < n$ [13]. However neither the detailed derivation of the converter from [5] nor the qualitative comparison of the alternatives is known so far. Although the choice of $k < n$ [13] results in simpler hardware that essentially implements custom 3-operand addition, this design suffers from a very limited choice of n and k. In fact for 32-bit dynamic range, the sole possible choice is $n = 12$ and $k = 9$. As RNS multiplication is more likely to achieve the closest inter-channel balance between odd and even moduli for $n < k \leq 2n$, such a choice would rather impair the speed of the whole system. Consequently, the system as a whole does not really benefit from the high speed of the reverse converter.

For $k > n$, Eq. (3) can be reduced to 4-operand addition provided proper residues from each channel are fed to the converter [5]. That is likely to yield higher power consumption and larger delay than the converter of [13], but a full range of n and k choices are achievable, hence the multiplication can be optimally balanced, while still the converter's delay may match closely the multiplier delay. In fact, we found the delay of this design vary modestly with different choices of n and k. Consequently the input adders that reduce the (S, C) pairs in each channel are the components primarily impacted with the particular n and k choice. However, logarithmic in delay prefix adders are far less sensitive to small variations in bit-width than the partial product matrices of the multiplication. Thus, the multiplier remains the primary component to consider for inter-channel balance.

The derivation of the reverse converter for the selected n and k directly follows these straightforward congruences.

Lemma 1. $\forall n \in N; i, k \in N \cup \{0\} : 2in \geq k$

$$\left|2^{2in-k}2^k\right|_{(2^{2n}-1)} \equiv 1$$

Proof.

$$\left|2^{2in-k}2^k\right|_{(2^n-1)(2^n-1)} = \left|2^{2in}\right|_{(2^{2n}-1)}$$

$$= 2^{|2in|_{2n}} \equiv 2^0 \equiv 1 \quad \square$$

Lemma 2. $\forall n \in N; i, k \in N \cup \{0\} : in \geq k + 1$

$$\left|2^{in-k-1}2^k(2^n + 1)\right|_{(2^n-1)} \equiv 1$$

Proof.

$$\left|2^{in-k-1}2^k(2^n + 1)\right|_{(2^n-1)} = \left|2^{(i+1)n-1} + 2^{in-1}\right|_{(2^n-1)}$$

from Eq. (1):

$$= \left|2^{|(i+1)n-1|_n} + 2^{|in-1|_n}\right|_{(2^n-1)} = \left|2^{|-1|_n} + 2^{|-1|_n}\right|_{(2^n-1)} \equiv 1$$

$\square$

The lemmas lead directly to the following corollary.

Corollary 1. *If $P_1 = 2^k, P_2 = 2^n + 1, P_3 = 2^n - 1$ in the Eq. (3) then $\forall i \in N : 2in \geq k$, 2^{2in-k} is a valid integer assignment for m_1, and $\forall j \in N : in \geq k + 1$, 2^{jn-k-1} is a valid integer assignment for m_2 in the Eq. (3).*

A direct consequence of this corollary and Prop. 1 from [13] is that the form (4) of Eq. (3) can be implemented as:

$$Y = |y_1 + y_2 + y_3 + y_4|_{2^{2n}-1}$$

where the binary representation of each of the components can be obtained by bit manipulation on residues x_1, x_2, x_3:

$$y_1 = \left|-2^{2n-k}x_1\right|_{2^{2n}-1} = 2^{2n-k}\sum_{i=0}^{k-1} 2^i \bar{x}_{1,i} + \sum_{i=0}^{2n-k-1} 2^i$$

$$y_2 = \left|2^{2n-k-1}x_2\right|_{2^{2n}-1} = 2^{2^n-k-1}\sum_{i=0}^{n} 2^i x_{2,i}$$

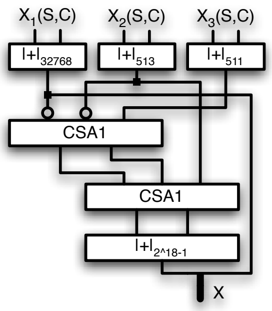


Figure 2: RNS-to-binary (reverse) converter

Table 2: Synthesis results

Arithmetic Unit	**Area (λ^2)**	**Pow. (mW)**	**Del. (ns)**	**P×D (mW·ns)**
32-bit Brent-Kung Adder	**11032**	**1.03**	**1.4**	**1.442**
Modulo $2^9 - 1$ Adder	2464	0.29	0.7	0.203
Modulo 2^{15} Adder	3982	0.45	0.7	0.315
Modulo $2^9 + 1$ Adder	5841	0.67	0.7	0.469
Modulo Adders	**12287**	**1.41**	**0.7**	**0.987**
32-bit Multiplier	**211927**	**48.52**	**4.2**	**203.78**
Modulo $2^9 - 1$ Multiplier	21647	4.97	2.8	13.92
Modulo 2^{15} Multiplier	30850	7.13	2.8	19.96
Modulo $2^9 + 1$ Multiplier	50915	11.95	2.8	33.46
Modulo Multipliers	**103412**	**24.05**	**2.8**	**67.34**
Binary-to-residue $2^9 - 1$	2287	0.26	0.7	0.182
Binary-to-residue 2^{15}	–	–	–	–
Binary-to-residue $2^9 + 1$	5081	0.61	0.7	0.427
Binary-to-residue	**7388**	**0.87**	**0.7**	**0.609**
Residue-to-binary	**52382**	**11.60**	**2.8**	**32.48**

$$y_3 = \left|-2^{3n-k-1}x_2\right|_{2^{2n}-1} = 2^{3n-k-1}\sum_{i=0}^{k-n} 2^i \bar{x}_{2,i}$$

$$+2^{2n-k}\sum_{i=0}^{n-2} 2^i + \sum_{i=0}^{2n-k-1} 2^i \bar{x}_{2,k-n+1+i}$$

$$y_4 = \left|2^{2n-k-1}\left(2^n x_3 + x_3\right)\right|_{2^{2n}-1} = 2^{3n-k-1}\sum_{i=0}^{k-n} 2^i x_{3,i}$$

$$+2^{2n-k-1}\sum_{i=0}^{n-1} 2^i x_{3,i} + \sum_{i=0}^{2n-k-2} 2^i x_{3,k-n+1+i}$$

Using 2 layers of end-around CSA of the width $2n$ on such permuted forms we can compute Y which, when concatenated with x_1 yields the final value of integer X. The organization of the circuit is shown in Fig. 2.

6. SYNTHESIS AND SIMULATION SETUP

We designed RNS components in RTL Verilog and synthesized using the 0.18μ OSU standard cell library with the *Cadence Encounter® RTL Compiler.* Along with those, we synthesized the 32-bit 2's complement Brent-Kung adder and the 32-bit 2's-complement multiplier obtained from the

ARITH project [8]. Timing constraints for the synthesis were initially set to minimum delay, then relaxed to maintain the integral ratios between components. As expected the final synthesis results (Table 2) indicated significant performance and power advantages: (i) 2-operand RNS addition and multiplication were $2X$ and $1.5X$ faster than their 2's complement counterparts; and (ii) power consumption of RNS multiplication was $\approx$ 50% of its 2's complement counterpart. Note also, that the total power consumption of RNS multiplication and reverse conversion was actually lower than the power consumption of the 2's complement multiplier. Hence, intuitively, application mapping on RNS extension should benefit from substantial reduction in computational units power consumption even when performance improvements were not possible to generate. This intuition will be strongly confirmed with our experimental results discussed further in the paper.

In the next step we have enhanced an existing RISC processor with the results of our architectural exploration. First, we have made the assumption that 2's complement adder determines the cycle duration in the pipeline. The rationale behind such assumption is straightforward. The adder quite commonly constitutes the critical path of the processor's datapath, hence no other component that operates in single cycle is slower than the adder. Although this leaves an RNS adder with a lot of unutilized slack, reducing the pipeline cycle time to match RNS adder delay would not only force making integer addition multi-cycle, but also imply the necessity of thorough consideration of the impact of this design decision on other pipeline components that we otherwise omit in our analysis. While this would be a valid design space point to explore, such efforts are beyond the scope of this work. Hence, in order to otherwise exploit unutilized slack in RNS components, based on the synthesis results, we fit into the pipeline binary-to-RNS conversion unit for single-cycle conversion of two binary operands, and a single-cycle 3-operand RNS adder. Consequently, RNS multiplication and RNS-to-binary conversion are 2 cycle operations. Finally, 2's complement multiplication is 3 cycle. We also include a separate RNS register file. RNS operations read from and write to this separate register file, while the forward conversion and the reverse conversion transfer data between the normal and the RNS register file.

Finally, in order to evaluate the proposed architecture we have augmented the SimpleScalar ARM simulator [2] with RNS arithmetic components, register file, and the instruction set extensions. We have also amended the GCC assembler for SimpleScalar ARM to recognize these new instructions. On the resultant platform we have run hand-optimized assembly code for kernels from several multimedia, image processing, and digital signal processing domains. The resulting runtimes as well as total functional unit power dissipation observed in the experiment are summarized in Fig. 3. For the purpose of power dissipation modeling, we consider the dynamic power of functional units estimated as the product of the functional unit power obtained from synthesis (Table 2), and the number of executions of the operation implemented by the functional unit.

The results of manually optimized runs (Fig. 3) demonstrate performance improvements that range from 8.62% for the LL-Hydro benchmark to more than 51% for 2D-DCT, with almost 30% improvement on average. Operations represented by these benchmarks form the core of the variety of image processing and DSP building blocks like color space transformation, geometric transformations, edge detection, digital filters, etc. Additionally, functional units power dissipation reduction of up to 57% (Matmul) and $\approx$ 52% on average, can be obtained as direct result of the lower power consumption in RNS functional units. Hence it is apparent that an RNS-equipped embedded microprocessor is able to offer simultaneous performance and power dissipation improvements. However, for the application programmer, these benefits may come at the price of tedious manual manipulation of assembly code and as such bear a prohibitive productivity overhead. In the following sections we demonstrate that to a significant extent these manipulations can be formalized as algorithmic compilation techniques and embedded into the standard C compiler.

7. CODE GENERATION

The two main tasks in generating code for a hybrid RNS processor are *instruction selection* and *instruction scheduling*. Instruction selection for RNS extension involves mapping arithmetic operations onto RNS instructions. However, since RNS hardware works only with RNS representation, conversions must be added at the beginning and at the end of each block of RNS instructions. Once mapping is completed, RNS instructions are subject to be scheduled with the objective to minimize runtime.

We formulate and solve these problems at the basic block level. A *basic block* is a continuous list of instructions, in which control flow has just one point of entry and one point of exit. In the basic block the data flow can be represented by a directed acyclic *Dataflow Graph* $G_{DFG}(V_{DFG}, E_{DFG})$, where a vertex $v \in V_{DFG}$ represents a computation and an edge $(u, v) \in E_{DFG}$ represents a dependency between vertices u and v, such that v consumes a value produced by u. We also define $PRED(v) = u \in V_{DFG} : (u, v) \in E_{DFG}$ and $SUCC(v) = w \in V_{DFG} : (v, w) \in E_{DFG}$. A DFG vertex $v \in V_{DFG}$ representing integer addition, multiplication, or subtraction is an *RNS eligible node*. A subgraph $G_{RES}(V_{RES}, E_{RES}) \subset G_{DFG}(V_{DFG}, E_{DFG})$ which is a connected component induced by a set of vertices $V_{RES} = \{v \in V_{DFG} : v$ is an *RNS eligible node*$\}$ is an *RNS eligible subgraph*. $G_{RES}(V_{RES}, E_{RES})$ that is not a proper subgraph of any other RNS eligible subgraph is a *maximal RNS eligible*

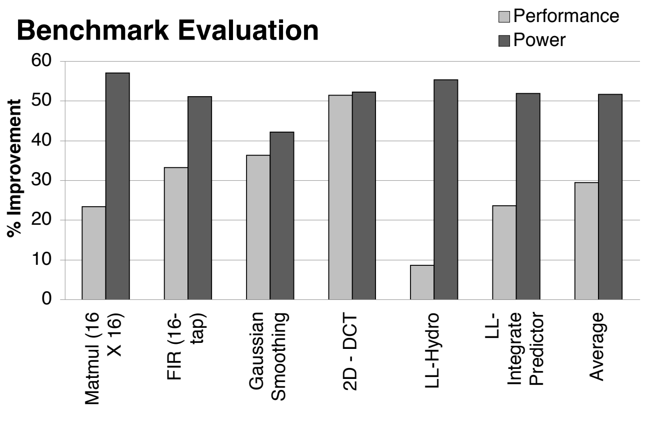


Figure 3: Performance and power improvements over various manually optimized benchmark kernels

subgraph $G_{MRES}(V_{MRES}, E_{MRES})$. $G_{RES}(V_{RES}, E_{RES})$ is *profitable* with respect to a metric M, if mapping of G_{RES} to RNS instructions improves the metric M. The instruction selection problem can be defined as follows: *Given* $G_{DFG}(V, E)$*, find all profitable* G_{MRES}. Each profitable G_{MRES} can be mapped onto the RNS extension, with appropriate conversions at the boundaries.

7.1 Instruction Selection

The instruction selection algorithm (Alg. 1) takes as input the G_{DFG} of the basic block and marks all MRESs. Then for each marked MRES it estimates its profitability with respect to cycle time using a profit model that includes conversion overheads. Profitable MRESs are then mapped to utilize RNS instructions. The algorithm *FIND_MRES* (Alg. 2), finds all MRESs, given the DFG of a basic block. First an RNS eligible node in the basic block is picked as the seed node, then the MRES is grown by the breadth-first search ignoring edge directions.

THEOREM 1. *Given a DFG* $G_{DFG}(V_{DFG}, E_{DFG})$*, algorithm FIND_MRES finds all subgraphs of* G_{DFG} *that are maximal RNS eligible subgraphs.*

PROOF. We state and prove the following statements:
The subgraphs marked by FIND_MRES *procedure in the graph* $G_{DFG}(V_{DFG}, E_{DFG})$ *are RNS eligible subgraphs.*

Lines 5-16 of *FIND_MRES* are the same as a breadth first traversal (BFT) in an undirected graph with v_s as the starting node. Lines 10-12 ensure that only nodes that are RNS eligible nodes are expanded in the BFT and only those are marked to be part of an MRES at Line 11.

2. *The RNS eligible subgraphs marked by* FIND_MRES *procedure in the graph* $G_{DFG}(V_{DFG}, E_{DFG})$ *are maximal.*

Suppose that a MRES $G_{MRES}(V_{MRES}, E_{MRES})$ that is a subgraph of the $G_{DFG}(V_{DFG}, E_{DFG})$ marked by the procedure *FIND_MRES* is indeed not maximal. Then $\exists v \in V_{MRES}$ s.t. $\exists u \in PRED(v) \cup SUCC(v)$ s.t. u is an RNS eligible node, but $u \notin V_{MRES}$. Since $v \in V_{MRES}$, it must have been queued at line 11 and eventually dequeued at line 6. There are two cases:

- u was already visited, in which case, it would also have been marked to be part of MRES at line 11 (because u is RNS eligible) in a previous iteration.
- u was not visited, in which case, it will now be marked to be part of the MRES, since it is RNS eligible.

In both cases, we find that $u \in V_{MRES}$, which is a contradiction of our original premise. Therefore G_{MRES} is maximal.

3. *The procedure* FIND_MRES *marks all possible maximal RNS eligible subgraphs.*

Suppose that there is an MRES $G_{MRES}(V_{MRES}, E_{MRES})$ of $G_{DFG}(V_{DFG}, E_{DFG})$ that was left unmarked after at the completion of *FIND_MRES* run. Since V_{MRES} only contains RNS eligible nodes and since they are unmarked, it means they were also not visited by *FIND_MRES*. Since line 3-4 ensure that all unvisited nodes are considered for seed nodes, G_{MRES} can remain unmarked only if it was not a subgraph of the original DFG, which is a contradiction. Thus all MRES that exist in the G_{DFG}, are marked. □

Algorithm 1 MAP2RNS

Require: Set S_{BB} consisting of DFGs for every basic block
Ensure: All graphs in S_{BB}, with all mapped profitable MRESs

```
1: for all $G \in S_{BB}$ do
2:     Do FIND_MRES($G$) [Alg. 2]
3:     for all MRESs, $G_{MRES}$ found in $G$ do
4:         profit ← ESTIMATE_PROFIT($G_{MRES}$)
5:         if profit > 0 then
6:             Transform $G_{MRES}$ to use RNS instructions
```

Algorithm 2 FIND_MRES

Require: $G_{DFG}(V_{DFG}, E_{DFG})$: DFG of basic block
Ensure: G_{DFG} with all MRESs marked

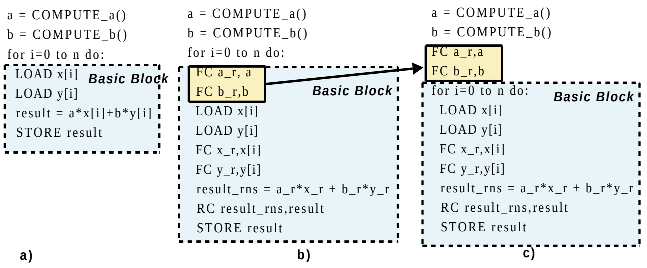
```
1: Let Q be a queue of nodes
2: MRES_ID ← 1
3: while All nodes in $V_{DFG}$ are not visited do
4:     Pick an unvisited RNS Eligible Node $v_s$ and add it to Q
5:     while Q is not empty do
6:         $v$ ← Node dequeued from Q
7:         for all $u \in PRED(v) \cup SUCC(v)$ do
8:             if $u$ is not visited then
9:                 Mark $u$ as visited.
10:                if $u$ is RNS Eligible then
11:                    $u.mres$ ← MRES_ID
12:                    Add $u$ to Q
13:    MRES_ID ← MRES_ID + 1
14: Return G
```

The *ESTIMATE_PROFIT* is a simple function that computes the profitability of MRES $G_{MRES}(V_{MRES}, E_{MRES})$ mapping on RNS extension in number of cycles. The following rules are applied to calculate the profit: (a) every pair of forward conversions is an overhead of 1 cycles; (b) every reverse conversion is an overhead of 2 cycles; (c) every 3-operand addition is a profit of 1 cycle; and (d) every multiplication is a profit of 1 cycle.

7.2 Instruction Scheduling

After instruction selection, we obtain a legitimate code for a hybrid RNS processor. However, there are at least two cases where the generated code is not optimal and requires explicit instruction scheduling. The first case is illustrated in Fig. 4, which shows a simple loop in C language (Fig. 4a). The basic block of this loop computes a weighted sum of elements of two arrays. Since our transformation operates at basic block level, it will place the forward conversions of constants a, and b inside the basic block, which is inside the loop (Fig. 4b). However, since a and b are not being changed inside the loop, their forward conversions can be moved outside the basic block (Fig. 4c).

The second issue is that for a series of addition, optimizing compilers will typically generate a balanced binary



```
a = COMPUTE_a()
b = COMPUTE_b()
for i=0 to n do:
  LOAD x[i]
  LOAD y[i]
  result = a*x[i]+b*y[i]
  STORE result
```

```
a = COMPUTE_a()
b = COMPUTE_b()
for i=0 to n do:
  FC a_r, a
  FC b_r,b
  LOAD x[i]
  LOAD y[i]
  FC x_r,x[i]
  FC y_r,y[i]
  result_rns = a_r*x_r + b_r*y_r
  RC result_rns,result
  STORE result
```

```
a = COMPUTE_a()
b = COMPUTE_b()
FC a_r,a
FC b_r,b
for i=0 to n do:
  LOAD x[i]
  LOAD y[i]
  FC x_r,x[i]
  FC y_r,y[i]
  result_rns = a_r*x_r + b_r*y_r
  RC result_rns,result
  STORE result
```

Figure 4: Moving forward conversions out of the loops

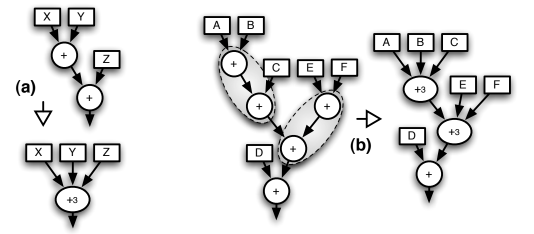


Figure 5: Sub-optimal pairing of additions

tree of additions in an attempt to optimize the number of 2-operand additions. However, since in our hybrid RNS processor, we have 3-operand adds, a binary tree does not yield optimal addition scheduling since consecutive 2-operand additions can be paired to form a 3-operand addition (Fig. 5a) resulting in possible reduction of the number of levels of the addition tree (Fig. 5b). Since we consider a hybrid processor with only one set of RNS resources we deal with the addition scheduling by simply linearizing the addition tree (Fig. 6). Hence, for an expression $\sum_{i=0}^{n} a_i$, containing n additions, the maximum number of addition pairs is $\lfloor \frac{n}{2} \rfloor$. Consider the DFG in Fig. 6 (left). There are 7 additions in the binary and only 2 pairs can be directly formed. However, if we linearize the structure of additions a maximum of 3 pairs can be formed (Fig. 6). Note however, that in case of a hybrid processor that employs more than one set of RNS resources reconstructing the addition tree as ternary would yield additional performance benefits.

8. EFFECTIVENESS OF COMPILER

We have implemented the discussed compilation techniques in GCC for ARM-SimpleScalar, and automatically generated code for all the benchmarks. As a result, we are able to compare the execution times of benchmarks on unmodified ARM architecture, along with three sets of RNS-optimized code: manually optimized, compiled with instruction selection only (basic), and compiled with instruction selection and scheduling (improved). Fig. 7 summarized the results of these runs as %-improvement over benchmarks running on an unmodified architecture. While manually optimized code yields an average performance improvement of almost 30%, just instruction selection achieves only ≈ 12% improvement as conversion overheads are not effectively managed. Enabling instruction scheduling enables more efficient handling of conversion overheads and exploits addition pairing, hence it is able to achieve ≈ 21% performance improvement on average. Note, that we provide no simulation results for the compiled LL-hydro benchmark. This is because the profit model does not report a performance profit for that

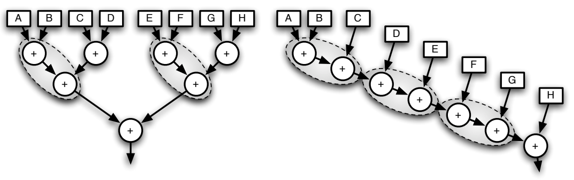


Figure 6: Linearizing the adds improves pairing

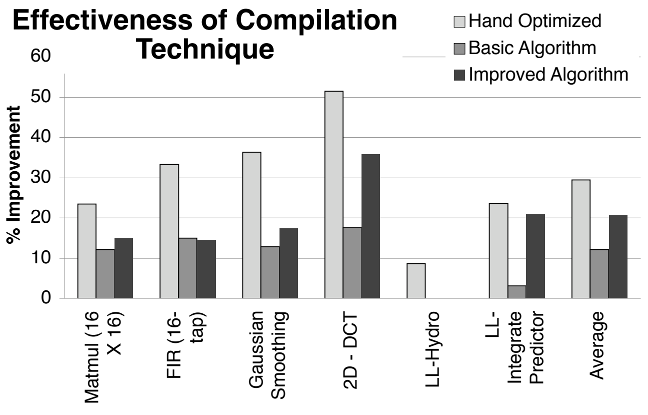


Figure 7: Performance improvements over various benchmark kernels and code generation techniques

benchmark, hence no mapping of code to RNS is done by the compiler. That somewhat matches the experience from manually optimized version of LL-Hydro, where the performance benefits are rather small and indeed not achieved at the basic block level. However, the power savings obtained from manually optimized run indicate possible augmentations to the profitability metric in order to enable these profits even with at performance improvement.

9. DESIGN SPACE EXPLORATION

Finally, we compare runtime vs. power consumption of the RNS-equipped ARM processor with processors having varying number of 2's complement adders and multipliers. The result of this exploration is depicted in Fig. 8. We observe that the power-performance product for RNS-equipped ARM is quite superior to those configurations having more adders and multipliers. This shows that in normal processors, while increasing the number of functional units does translate into improved performance, it comes at the cost of expending greater power. In contrast, the presence of faster, power efficient functional units and careful co-design enables RNS-equipped processor to achieve great performance (closely matching 2A,2M 2's complement integer configuration) while

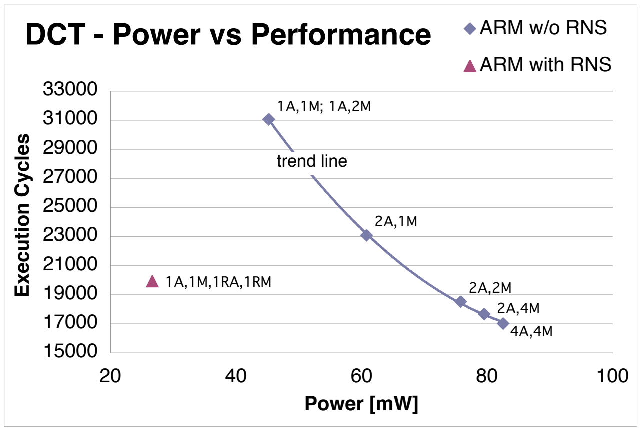


Figure 8: Performance vs power comparison with different adder/multiplier configurations.

dissipating in computational units *just a fraction* of power dissipated by 2M,2A configuration, as well as significantly smaller than 1A,1M configuration.

In general, with increasing amount of resources, application performance saturates when existing instruction level parallelism in the application is exhausted. RNS extension breaks the saturation barrier by exploiting parallelism inherently through RNS representation. As a result significant performance improvements can be delivered at much lower power consumption. Although power benefits are confined to arithmetic functional units only, they not only consume significant portion of the processor's power, but are also one of the most important hot-spots [11].

10. CONCLUSION AND FUTURE WORK

In this work we have significantly elevated the baseline for explorations of applicability of the Residue Number System to the embedded processor design. We demonstrated the benefits that RNS arithmetic has to offer to a low cost fixed-point DSP embedded processor, if only the trade-offs are vertically managed throughout the design. The maximum of $\approx 57\%$ power and $\approx 30\%$ performance improvements we obtained for manually optimized DSP kernels strongly disprove the sustaining stigma that RNS arithmetic is restricted to application specific hardware only. Moreover, our multi-tier synergistic design approach enabled us to exploit the benefits of RNS hardware through a handful of novel compiler techniques, that with average performance improvement of $\approx 21\%$ closely matches the manually optimized code and extend the leverage of RNS hardware to high-level programming languages.

The compiler technique introduced in this work could also benefit from improving the profit model to model instruction execution more accurately, as well as extending analysis of programs to larger blocks of the program graph, i.e. the hyperblock or superblock level. Resultantly, more subgraphs containing RNS-eligible operations could be mapped for performance and power benefits. Additionally, more aggressive optimizations of the ISA and its implementation still seem to be possible, including matching the system clock to RNS components' delays rather than 2's complement components' delays. That would allow to utilize the slack that still exists between these two alternatives. Finally, new language constructs can be introduced to guide the compiler in identifying code that could profitably be mapped to RNS.

In the application specific hardware inexpensive and easy to parallelize forward conversions are typically neglected and the reverse conversion is considered to be the dominant overhead of RNS. Interestingly, in our experiments we have found forward conversions to be much more likely to limit the achievable speed-up of certain algorithms than reverse conversions. With parallel execution limited by the memory interface and limited capacity of the register file, forward conversions are likely to outnumber reverse conversions by orders of magnitude. While compiler techniques can be devised to hide these latencies more efficiently, a natural exploration direction is to move them out of the processing pipeline — e.g. into the cache memory between levels 1 and 2 or integrate with loads and stores.

11. REFERENCES

[1] B. P. Amos Omondi. *Residue Number Systems Theory and Implementation.* Imperial College Press, 2007.

[2] T. Austin. Simplescalar. `http://www.simplescalar.com/`.

[3] J. Bajard, S. Duquesne, M. Ercegovac, and N. Meloni. Residue systems efficiency for modular products summation: Application to elliptic curves cryptography. *Proc. SPIE*, 6313:631304, 2006.

[4] R. Chaves and L. Sousa. RDSP: A RISC DSP based on residue number system. In *Proc. Europ. Symp. Dig. Syst. Des.*, pages 128–135, Sep. 2003.

[5] R. Chaves and L. Sousa. Improving residue number system multiplication with more balanced moduli sets and enhanced modular arithmetic structures. *IET Comp.s & Dig. Techn.*, 1(5):472–480, Sept. 2007.

[6] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. In *Int'l Workshop Applied Reconf. Comp. (ARC)*, pages 209–220, March 2008.

[7] M. Griffin and F. Taylor. A residue number system reduced instruction set computer (RISC) concept. In *Proc. Inter. Conf. Acoust., Speech, & Sign. Proc.*, pages 2581–2584, 1989.

[8] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal design of arithmetic circuits based on arithmetic description language. *IEICE Trans. Fundamentals*, E89-A:3500–3509, 2006.

[9] H.T.Vergos. A 200-MHz RNS core. In *Proc. Europ. Conf. Circ. Theory & Des.*, August 2001.

[10] W. K. Jenkins and B. J. Leon. The use of residue number systems in the design of finite impulse response digital filters. *IEEE Trans. Circ. & Syst.*, CAS-24:191–201, 1977.

[11] D. Kannan, A. Shrivastava, S. Bhardwaj, and S. Vrudhul. Power reduction of functional units considering temperature and process variations. In *Proc. VLSI Design*, volume 0, pages 533–539, Los Alamitos, CA, USA, 2008.

[12] P. V. A. Mohan. *Residue Number Systems: Algorithms and Architectures.* Kluwer Academic Publishers, Norwell, MA, 2002.

[13] A. Molahosseini, K. Navi, O. Hashemipour, and A. Jalali. An efficient architecture for designing reverse converters based on a general three-moduli set. *J. Syst. Archit.*, 54(10):929–934, 2008.

[14] S. J. Piestrak and K. S. Berezowski. Architecture of efficient RNS-based digital signal processor with very low-level pipelining. In *Proc. IET Irish Sign. & Syst. Conf.*, pages 127 –132, Galway, Ireland, 18-19 June 2008.

[15] S. J. Piestrak and K. S. Berezowski. Design of residue multipliers-accumulators using periodicity. In *Proc. IET Irish Sign. & Syst. Conf.*, pages 380–385, Galway, Ireland, 18-19 June 2008.

[16] J. Ramirez, A. Garcia, S. Lopez-Buedo, and A. Lloris. RNS-enabled digital signal processor design. *IEE Electr. Lett.*, 38(6):266–268, 2002.

[17] S.J.Piestrak. Design of residue generators and multioperand modular adders using carry-save adders. *IEEE Trans. Computers*, 43(1):68–77, Jan 1994.

[18] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor. *Residue number system arithmetic: Modern Applications in Digital Signal Processing.* IEEE Press, Piscataway, NJ, USA, 1986.

[19] T. Tomczak. Fast sign detection for RNS $(2^n - 1, 2^n, 2^n + 1)$. *IEEE Trans. Circ. and Syst. I*, 55(6):1502–1511, July 2008.

[20] T.Stouratitis and V.Paliouras. Considering the alternatives in low-power design. *IEEE Circ. & Dev.*, pages 23–29, 2001.

[21] Y. Wang. Residue-to-binary converters based on new Chinese Remainder Theorems. *IEEE Trans. Circ. Syst. II*, pages 197–206, Mar 2000.