RESIDUE NUMBER SYSTEM ENHANCEMENTS FOR PROGRAMMABLE PROCESSORS

by

Rooju Gnyanesh Chokshi

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2008

RESIDUE NUMBER SYSTEM ENHANCEMENTS FOR PROGRAMMABLE PROCESSORS

by

Rooju Gnyanesh Chokshi

has been approved

November 2008

Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Sarma Vrudhula
Rida Bazzi

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Today's embedded processors face multi-faceted challenges in the form of stringent performance, area, power and time-to-market requirements. There is a perpetual demand for embedded processors with lower power, yet higher performance, especially in personal wireless communication and multimedia domains.

The 2's complement number system imposes a fundamental limitation on power-performance efficiency as a result of sequential carry propagation. The Residue Number System (RNS) breaks this limitation by partitioning operations into parallel independent components resulting in fast and power-efficient hardware. While the performance and power benefits of these RNS components have been exploited in application specific hardware, it has not been possible to capitalize on their advantages in a programmable processor. The main challenge lies in mitigating overheads of conversion operations that are needed to transform operands from 2's complement to RNS and vice-versa.

This work meets this challenge by synergistic architecture, micro-architecture, component co-design which enables hiding these overheads effectively, paving the way for significant performance benefits. Additionally, compiler techniques that rely on these enhancements are also proposed for automated code mapping. Concentration on mitigating overheads across design levels results in an RNS-based extension to Reduced Instruction Set Computer (RISC) processors that demonstrates application performance improvement of 21 percent and functional-unit power improvement of 54 percent in the average.

To

my family

ACKNOWLEDGMENTS

I would like to thank my advisor and committee chair Dr. Aviral Shrivastava but for whom, this work would have been an unplanted seed. I would like to thank him for those energetic and stimulating discussions we had. His infinite patience, support, invaluable guidance and encouragement has been the backbone in this extremely crucial, yet most exciting endeavor of my master's degree. His never-say-die attitude has never ceased to amaze me.

In the same breath, I would also like to thank Dr. Krzysztof Berezowski, for not only bringing to attention the powers of Residue Number System initially, but also for lending his experience to bring me up to date with the state-of-the-art. This work would have never developed a strong foundation if it were not for his continuous and zealous interest in it.

I would also like to thank Dr. Sarma Vrudhula and Dr. Rida Bazzi for the invaluable support they provided me as committee members. I also thank my colleagues at the Compiler-Microarchitecture Laboratory, Jong-eun Lee, Sai Mylavarapu, Reiley Jeyapaul, Amit Pabalkar and Arun Kannan for their interesting perspectives on my research. My heartfelt gratitude also extends to Amit, Arun, Khushboo and Vivek for making our lab an energetic and lively place with humor that ranged from rib-tickling to side-splitting. Being picked on has never been so much fun!

I would like to acknowledge the Engineering Career Center for providing me with an internship opportunity which lent completeness to my education. I would like to thank Microchip Technology, Inc, Chandler and my mentor Nicky Ayoub for exposure to current technologies, which broadened my engineering perspective.

I am very grateful to my parents, Gnyanesh and Mona Chokshi, and sisters, Prachi and Pratichi. Their unflinching love, support and belief in me has made life's challenges less difficult to surmount. To them, I dedicate this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. Introduction

Embedded systems today have transformed from simple, single-function control systems to highly complex, multipurpose computing platforms. Higher performance is no longer the only important criterion in such designs. The advent and popularity of personal wireless communication and handheld, portable multimedia and communication devices in the last decade has created stringent requirements on performance, power, cost and time-to-market. The omnipresence of such battery-powered devices has created a perpetual demand for cheap, high performance, and power efficient embedded processors.

A. *Shortcomings Of 2's Complement Arithmetic*

The binary 2's complement number system forms the basis of contemporary computer arithmetic. However, it imposes a fundamental limit on the achievable performance and power. Consider the n-bit Ripple Carry Adder (RCA). Although this n-bit adder is simplest and most efficient in terms of area and power, the carry propagation chains lead to an $O(n)$ delay in computation, which is too slow for the performance requirements of the present day. As a result, a large number of sophisticated adders employing complex carry prediction schemes have been developed. For example, the Brent-Kung Adder [4] and the Kogge-Stone Adder [20], employ complex carry prefix networks to speed up computations by reducing the delay to $O(\log n)$. However, these parallel prefix networks are much more expensive in terms of area and power consumption.

Furthermore, multipliers for 2's complement binary arithmetic also have serious area and power requirements. The speed of the multiplier crucially depends on the depth of the partial product reduction tree as well as the final fast carry propagate adder (CPA). The reduction of the $O(n^2)$ partial product bits with $O(logn)$ delay is quite expensive in terms of area and power.

These limitations have led to exploration of alternative representations for doing computer arithmetic [41] and the residue number system (RNS) has been identified as a very promising alternative.

B. *Residue Number System*

In a RNS, integers are broken into smaller components such that important operations, can be performed on smaller components independently of each other.

B.1. *Definition*

A residue number system is characterized by a set of relatively prime integers (i.e., no pair from the set contains a common nonunity factor) $P = (p_1, p_2, ..., p_k)$, the *moduli set*. Any number $N \in [0, M)$ where $M = \prod_{i=1}^{k} p_i$ is the *dynamic range* of the system, can be uniquely represented in RNS as a k-tuple $(n_1, n_2, ..., n_k)$, where $n_i = |N|_{p_i}$ is the remainder of dividing N by p_i. Resultantly, certain operations $\circ \in \{+, -, \times\}$ on two RNS operands $X = (x_1, x_2, ..., x_k)$ and $Y = (y_1, y_2, ..., y_k)$ can be performed as $X \circ Y = (x_1 \circ y_1, x_2 \circ y_2, ..., x_k \circ y_k)$. The conversion from residue number system to 2's complement is done by applying the Chinese Remainder Theorem.

This form of computation has a very important property that the processing of the i-th component of the result depends only on the i-th components of the operands. This distribution of computation into smaller bit-width and independent *channels*, not only reduces the carry propagation length in adders, but also dramatically reduces the number of partial products that need to be accumulated in multipliers, leading to remarkably faster and low power implementations of these operations. Also, Freking and Parhi [8] argue that RNS may reduce circuit's switching activity, and may benefit from more aggressive supply voltage scaling.

B.2. *Conversions Into And From Residue Number System*

Although residue number systems have unique properties that enable faster computations on integers, there are also overheads associated, in the form of transforming representation from binary to residue (called *forward conversion*) and vice-versa (called *reverse conversion*). While forward conversion involves computing residues by the modulo operation, reverse conversion is a more complex operation. There are two general methods to obtain a binary number X from its residue representation, *Mixed Radix Conversion* (MRC) and *Chinese Remainder Theorem* (CRT).

a. *Mixed Radix Conversion*

Given a RNS defined on the moduli-set $P = (p_1, p_2, ..., p_k)$ and a number $(x_1, x_2, ..., x_k)$ in residue representation, the binary number X is computed as

$$X = a_k \prod_{i=1}^{k-1} p_i + \cdots + a_3 p_1 p_2 + a_2 p_1 + a_1 \tag{I.1}$$

where the a_i, called the Mixed Radix Coefficients, are calculated as follows:

$a_1 = \Big|X\Big|_{p_1} = x_1$, $a_2 = \Big|\frac{X}{p_1}\Big|_{p_2}$, $\ldots$, $a_i = \Big|\frac{X}{\prod_{j=1}^{i-1} m_j}\Big|_{m_i}$

b. *Chinese Remainder Theorem*

Given a RNS defined on the moduli-set $P = (p_1, p_2, ..., p_k)$ and a number $(x_1, x_2, ..., x_k)$ in residue representation, the binary number X is computed as

$$X = \Big|\sum_{i=1}^{n} P_i |P_i^{-1}|_{p_i} x_i\Big|_M \tag{I.2}$$

where $M = \prod_{i=1}^{k} p_i$, $P_i = \frac{M}{p_i}$ and $|P_i^{-1}|_{p_i}$ is the multiplicative inverse of P_i modulo p_i.

Both these methods involve complex, modular calculations which result in complex and large circuit implementations that are expensive in both speed and power. Therefore, any RNS based design has to take into consideration the high overheads of conversion. A computation can only be beneficial if the profits gained in the computational steps exceed the overheads associated with conversions.

B.3. *Major Research Areas In RNS*

A significant amount of research has focused on designing extremely fast RNS computational and conversion hardware [25], especially for the conversion from RNS to binary, since that is a very costly operation.

Another major research area lies in the application of RNS to digital signal and image processing. The advantages of RNS in application specific hardware for digital filters, matrix operations, and image processing operations have been well appreciated [23,41,47].

An area which is important but has not received much attention is incorporating RNS arithmetic into general programmable processors. Although RNS enables much faster additions and multiplications, magnitude comparison and division are expensive as they do not lend themselves to the same distributed and parallel computational structure as addition and multiplication. This implies that a RNS-only processor would be too restrictive in its computational capabilities. Therefore, any processor that seeks to take advantage of RNS should need to have a hybrid design that supports both 2's complement and RNS arithmetic. However, in such hybrid processor, the conversion operations have very significant overhead, and can outdo all advantages of fast and low-power RNS arithmetic units. Consequently, there have been few previous approaches to RNS-based processor design and they too have focused on power and area reduction and not on application performance [5,16].

Through this work, we seek to bring forth the immense benefits of residue number system so that a large variety of existing algorithms can take advantage of high performance computation. Through a series of synergistic instruction-set-architecture, microarchitecture, and component design decisions, we produce an RNS extension to an existing RISC processor that is able to demonstrate both performance and power improvements. However, having a microarchitecture extension is just one side of the coin and it naturally leads to the problem of compilation of programs to these RNS extensions. The key question to be addressed is "Given a program, what portions of code should be mapped to RNS extension?". Manual analysis and optimization of code is tedious, at best, for small programs and certainly infeasible for large programs. Therefore an automated compilation technique that can analyze and map code is needed. A naive approach to automation would be to instruct the compiler to map all additions and multiplications to RNS and provide appropriate conversions operations at the boundaries. However, this highly optimistic approach may, indeed, deteriorate performance, if the cost of conversion overheads exceeds the benefits of faster arithmetic. Therefore, we propose novel compilation techniques to tackle the RNS code mapping problem. To the best of our knowledge, this is the first attempt at designing compilation techniques for RNS-enabled microprocessors.

C. *Summary Of Major Contributions*

- We propose a residue number system based extension for a 32-bit RISC architecture. Synergistic co-design of architecture, microarchitecture and instruction set allows us to achieve simultaneous performance and power benefits.
- We define the problem of code mapping and propose compilation techniques for profitable mapping.

D. *Organization Of The Thesis*

In Chapter II, we present a summary of previous research in RNS-based design. Details of our design decisions, workings of our RNS functional units and our architecture model are presented in Chapter III. In Chapter IV we give a formal definition of the RNS code mapping problem and our approach to solve it. Chapter V contains a discussion of our experimental setup, evaluation and results. Future work and conclusions are described in Chapter VI.

II. Previous RNS Research

Research in RNS-based processor design can be classified into 3 main categories.

A. *RNS Functional Units*

A.1. *Conversion Units*

Any RNS-based design involves conversions from binary to RNS, commonly known as *forward conversion*, and vice versa, known as *reverse conversion.* The conversion units need to be extremely efficient as conversion operations are an overhead in performance and power in RNS-based systems. Reverse conversion, which is an application of the Chinese Remainder Theorem, is especially expensive. It is therefore natural that a lot of research has addressed not only discovering efficient moduli-set, but also faster and power-efficient reverse converter circuits.

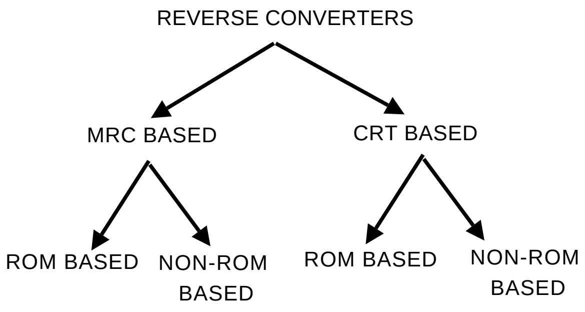


Fig. 1. Classification of Reverse Converters

A classification of the various kinds of techniques is shown in Figure 1. The first level of classification is made on the basis of what mechanism, whether the mixed radix conversion (Equation (I.1)) or Chinese Remainder theorem (Equation (I.2)), is used to compute the binary number from residues. An additional classification can be made on the actual hardware used in the implementation of the reverse converter. The first reverse converters were based on read-only memories (ROMs) to hold pre-computed moduli used in computation of the terms of Equation (I.1) and Equation (I.2). These converters, though fast, do not scale well for large modulo operations as the size of the ROM is directly proportional to the modulus. Non-ROM based converters, instead, compute all the terms of the equations and are more scalable.

More recent reverse converters [43,45,46] have utilized an alternative simpler formulation of the Chinese Remainder Theorem, called the New Chinese Remainder Theorem [44]. This new formulation transforms the original CRT equation, which needs a large modulo operation, to use terms with smaller values of moduli.

A.2. *Computational Units*

A large number of adders and multipliers have been proposed over the years for a variety of moduli. These can, in general, be classified into ROM and non-ROM based implementations. A review of some of the early modulo adders and multipliers can be found in [26]. More recently, Hiasat [13] has proposed high-speed modular adders suitable for VLSI implemenation. Efstathiou et. al [7] have proposed fast modulo adders based on parallel-prefix carry computation units. Zimmermann [48] proposes circuits for addition and multiplication modulo $2^n - 1$ and $2^n + 1$. Adders are based on parallel-prefix architectures and multipliers are enhanced so that speed-up techniques like Wallace trees and Booth-recoding can be used.

B. *Applications Of RNS Designs*

RNS has long been applied in implementations of digital signal processing hardware like Digital-to-Analog converters, Finite Impulse Response (FIR) filters, Infinite Impulse Response (IIR) filters, 2-D FIR filters.

Jenkins and Leon [18], Soderstrand [35], Wang [42], Miller and Polsky [24], have constructed FIR filters based on residue number system. Huang et al [17], Shanbhag and

Siferd [33] have utilized RNS to realize efficient pipelined 2-D FIR filters. Soderstrand and Sinha [37], Sinha and Loomis [22] have suggested the implementation of IIR filters using RNS. Moreover, RNS has also been used in the design of 1-dimensional [27, 28, 30, 31] and 2-dimensional Discrete Wavelet Transform architectures [21].

RNS has also been used to speed up RSA encryption and decryption [2] and for high-speed Elliptic Curve Cryptography (ECC) [32]

RNS-based architectures have also been explored for lesser-known applications like adaptive beamforming, a form of modified Gauss elimination [19].

C. *Programmable RNS Architectures*

A comparitively lesser amount of research has focussed on the development of programmable RNS microprocessors.

Griffin and Taylor [9] first proposed the RNS RISC architecture as a future research area. Vergos [16] developed an RNS core supporting RNS addition and multiplication. The motivation for that work was not the development of a self-contained programmable RNS-based processor, but to provide a pre-designed IP block for use in IC designs. E.g. their core does not include conversion from RNS to binary and hence is not complete in RNS functionality. Also, it does not process operands for the 3 channels in parallel, but samples them serially and processes them in a pipelined fashion.

Ramirez et al [29] have developed a 32-bit SIMD RISC processor which is purely based on RNS arithmetic and highly tailored for DSP applications. Their architecture supports only RNS addition, subtraction and multiplication. Moreover, conversion operations are not included in their instruction set architecture. This means that hardware units for conversion need to be present as additional stages in the dataflow. Although programmable, this architecture is limited in the kind of applications it can execute.

Chavez and Sousa [5] proposed a complete architecture for a RNS-based RISC DSP, primarily focussed on the reduction of power and chip area. They proposed a composite multiply-and-accumulate block to perform additions, multiplications, and multiply-and-accumulate operations. Although they did report some performance improvements, they did not describe any architectural or microarchitectural details of their design.

III. Residue Number System Extension to a RISC Processor

A. *Instruction Set Architecture*

A first-cut approach to integrate RNS functional units into an existing RISC architecture, could be to attach conversion logic at their inputs and outputs. While this solution has the advantage of being transparent to system software, it implies that every fast RNS operation involves forward and reverse conversions as shown in Figure 2(a). As a result, benefits of faster computation are overshadowed by the cost of conversions. We conclude that it is important to separate conversion of operands from computations as shown in Figure 2(b). This implies that there will be separate instructions for computations, and separate instructions for converting operands from 2s complement binary to RNS, and vice versa. This chief advantage of this is that now it is possible to reduce conversion overhead through appropriate instruction scheduling and automated code transformation. Table I describes the new instructions that we add to the RISC ISA.

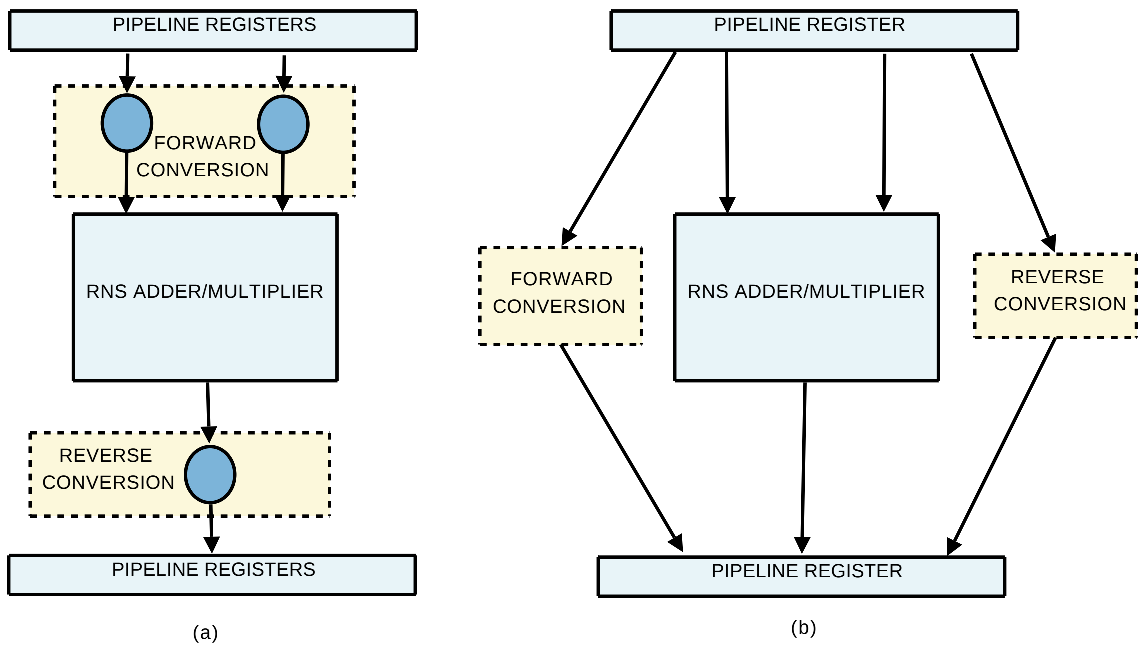


Fig. 2. (a) Naive integration (b) Separation of conversion from computation

TABLE I
Instructions For RNS Computation

Operation	**Mnemonic**
Forward Conversion	FC
RNS Addition	RADD
RNS Subtraction	RSUB
RNS Multiplication	RMUL
Reverse Conversion	RC

B. *Microarchitecture Decisions*

Multi-operand addition (MOA) using Carry-Save Adders (CSA) play an important role in the design of RNS functional units. CSA-based addition produces separate carry C and sum S vectors, that are typically reduced by a final modulo adder to produce the final residue $R = |C + 2S|_P$ (Figure 3(a)). However, modulo adders are slower and consume more area and power than regular adders. It is well known though, that the C and S vectors represent 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 [34]. Hence, if we choose to represent operands as (S, C) pairs, we may benefit from speed, area, and power advantages at the same time as the final modulo adder stage can now be removed (Figure 3(b)). Although this reduces the complexity of the adder and the forward converter, it introduces additional complexity in the multiplier and reverse converter. Hence, this is an important system-level design consideration. However, since forward conversion is required per operand, additions are typically much more frequent than multiplications, and reverse conversions are rather infrequent, this representation boosts the overall benefit of RNS extension.

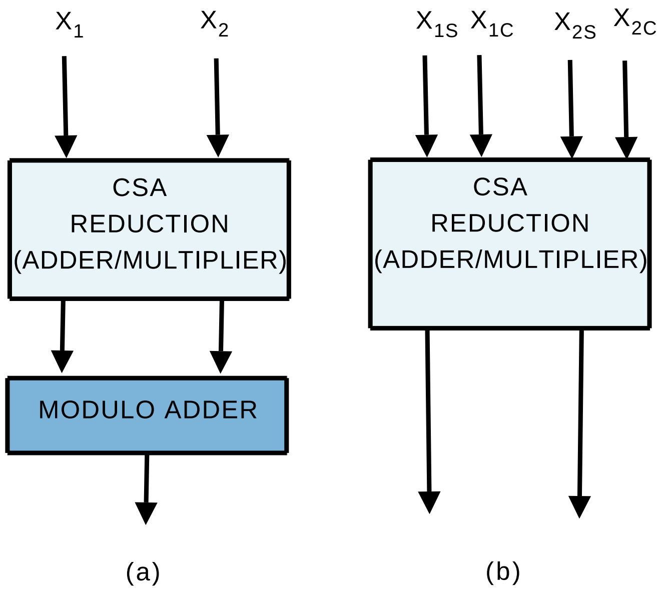

X_1
X_2
CSA REDUCTION (ADDER/MULTIPLIER)
MODULO ADDER
(a)
X_{1S}
X_{1C}
X_{2S}
X_{2C}
CSA REDUCTION (ADDER/MULTIPLIER)
(b)

Fig. 3. (a) Conventional CSA reduction with modulo adder (b) CSA Reduction with the (S, C) representation

The storage of these double-width operands can be achieved without any micro-architectural changes, as RISC processors often allow registers and memory to be accessed as double precision storage spanning 2 registers or memory locations.

C. *Moduli Set Selection*

The selection of moduli-set is crucial to designing efficient arithmetic and conversion units. $(2^n - 1, 2^n, 2^n + 1)$ is a very widely used set as it is easy to design efficient functional and conversion units for it. However it is not *well-balanced* [6]. For effective balancing, we desire faster channels to operate on more inputs bits. $(2^n - 1, 2^{2n}, 2^n + 1)$ has been proposed by [6]. The power-of-2 channel, which is the fastest among the 3 channels, operates on twice the number of bits as each of the other two channels. However, our experiments reveal that, for our design of functional units, the moduli-set $(2^n - 1, 2^k, 2^n + 1), k > n$ achieves a better balancing of channels. We evaluate the channel balance in the multiplier for different configurations of (k, n) values, including $(k, n) = (2n, n)$, to find the most balanced configuration for our component design. This evaluation is presented later in this section, when the design of the multiplier is presented.

A well-known property of numbers of the form $2^n \pm 1$ is that there is a periodicity of values [34] of $|2^{i+nj}|_{2^n \pm 1}$, with period depending on n. That is:

$$\forall i, j \in N \cup \{0\}, 2^{i+nj} \equiv |2^i|_{2^n-1} \tag{III.1}$$

$$\forall i, j \in N \cup \{0\}, 2^{i+nj} \equiv (-1)^j |2^i|_{2^n+1} \tag{III.2}$$

Equation (III.1) and Equation (III.2) imply that in a binary weighted representation of a number, any bit at position $i + nj$ is equivalent, in weight, to a bit at position i, when considered in the ring of numbers modulo $2^n \pm 1$. Piestrak [34] has used these properties enable fast and regular designs for RNS functional units.

D. *Binary-to-RNS (Forward) Conversion*

The forward converter computes residues of the 32-bit input X and produces a pair (S, C) for every channel. $|X|_{2^k}$ is simply the least significant k bits of X. The design for computation of $|X|_{2^n \pm 1}$ channels is shown in Figure 4

Periodicity of residues allows us to do the computation on X as follows:

$$\begin{aligned} |X|_{2^n-1} &= |2^{pn}X_p + \cdots + 2^{2n}X_2 + 2^nX_1 + X_0|_{2^n-1} \\ &= |X_p + \cdots + X_2 + X_1 + X_0|_{2^n-1} \end{aligned} \tag{III.3}$$

$$\begin{aligned} |X|_{2^n+1} &= |2^{pn}X_p + \cdots + 2^{2n}X_2 + 2^nX_1 + X_0|_{2^n+1} \\ &= |(-1)^pX_p + \cdots + X_2 - X_1 + X_0|_{2^n+1} \end{aligned} \tag{III.4}$$

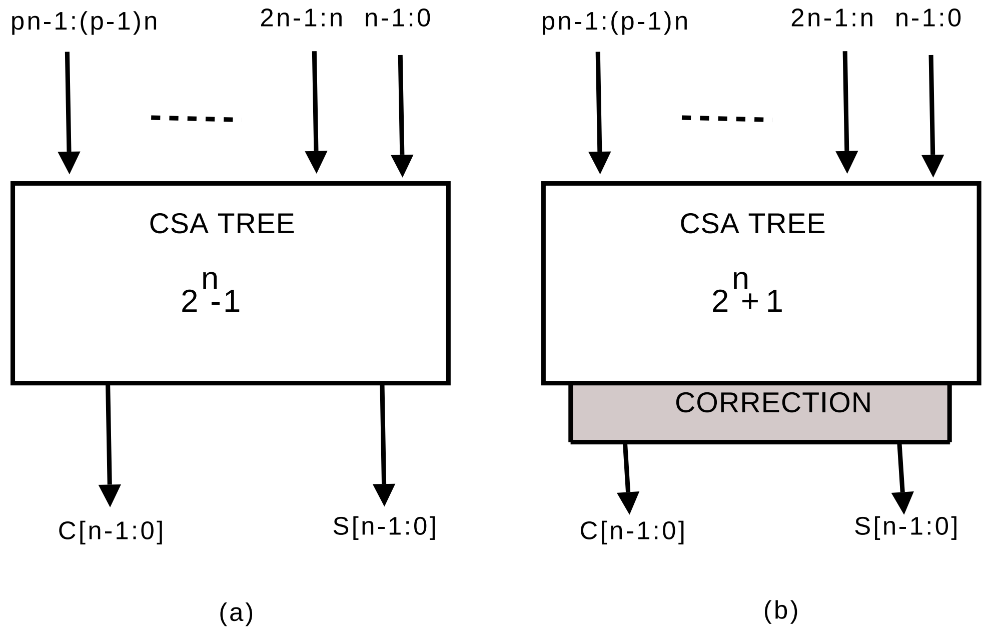


Fig. 4. Binary-to-RNS (Forward) Converter

Thus, we can align the terms X_p through X_1 and add them in a CSA tree. In Equation (III.4), the bits in the negative terms are inverted when they are added in the CSA tree. For each inversion of a bit at weight 2^{i+nj}, we need to apply a correction of -2^i to the final result. Since, there is always a constant number of inversions in the CSA tree, the constant correction is added as extra layer of full-adder cells simplified with the carry input as 0 or 1 according to bits of the correction (hatched layer in Figure 4)

It is important to note that the choice of representing operands in (S, C) form obviates the need to add a final modulo adder stage to combine S and C vectors.

E. *RNS Adder Design*

The RNS adder for every channel takes in 2 operands in the form $X_1 = (S_1, C_1)$ and $X_2 = (S_2, C_2)$, and produces a result $X_3 = (S_3, C_3)$. The adder for the 2^k channel is a CSA tree in which all bits with weight greater than 2^k are discarded.

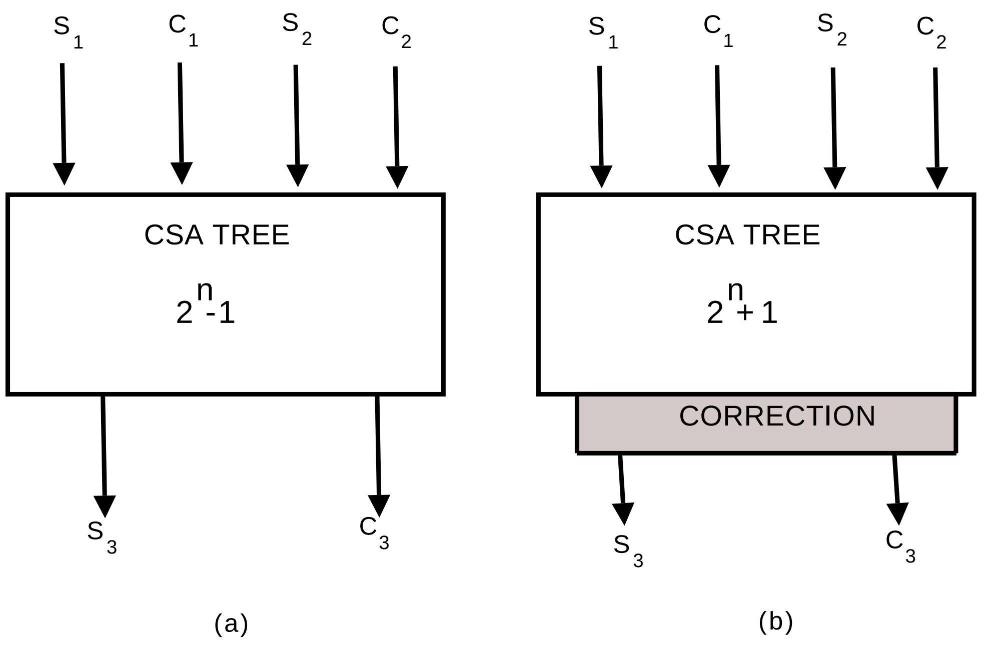


Fig. 5. RNS Channel Adders

The adders for the $2^n \pm 1$ channels are shown in Figure 5. Again, the bits are aligned according to their weights as given by the periodicity and added in a CSA tree. For the $2^n + 1$ channel adder, a constant correction needs to be added to the result of the CSA tree. The subtraction operation can be implemented by augmenting the inputs of the adders with logic to conditionally negate the second operand, based on a control signal activated by the RSUB instruction (omitted from figures for simplicity).

F. *RNS Multiplier Design*

One way to multiply two RNS numbers (S_1, C_1) and (S_2, C_2) is to multiply them in the form: $(S_1 + 2C_1) \cdot (S_2 + 2C_2) = S_1 \cdot S_2 + 2S_1 \cdot C_2 + 2S_2 \cdot C_1 + 4C_1 \cdot C_2$. This method involves the generation and addition of partial products for the four product terms in the expansion. To avoid this, we first produce two intermediate operands $X_1 = S_1 + 2C_1$ and $X_2 = S_2 + 2C_2$. Partial products are generated X_1 and X_2, aligned according to periodicity,

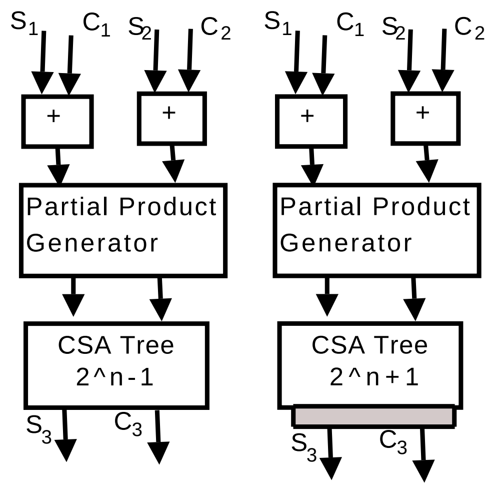


Fig. 6. RNS Channel Multipliers

and added in a CSA tree. As in the $2^n + 1$ channel adder, the $2^n + 1$ channel multiplier also needs a fixed correction. The multipliers are shown in Figure 6. Multiplier for the 2^k channel follows the same structure except bits at positions greater than or equal to k in the CSA tree are discarded.

Note that overhead in the multiplier, of having adders to compute X_1 and X_2 results from the choice of having the (S, C) format. However this format enables faster adders and since additions are more frequent than multiplications in typical applications, it results in faster application execution.

With this design of RNS multipliers, we can now evaluate various moduli-set configurations for balance. The various configurations for a 32-bit dynamic range, and the multiplier delay for the 3 channels is shown in Table II. The span in the delays of the 3 channels is the least for the moduli-set $(2^9 - 1, 2^{15}, 2^9 + 1)$, which corresponds to $n = 9$ and $k = 15$ in the moduli-set $(2^n - 1, 2^k, 2^n + 1)$ and we choose this to be our moduli-set.

TABLE II
MULTIPLIER CHANNEL DELAY FOR DIFFERENT 32-BIT 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

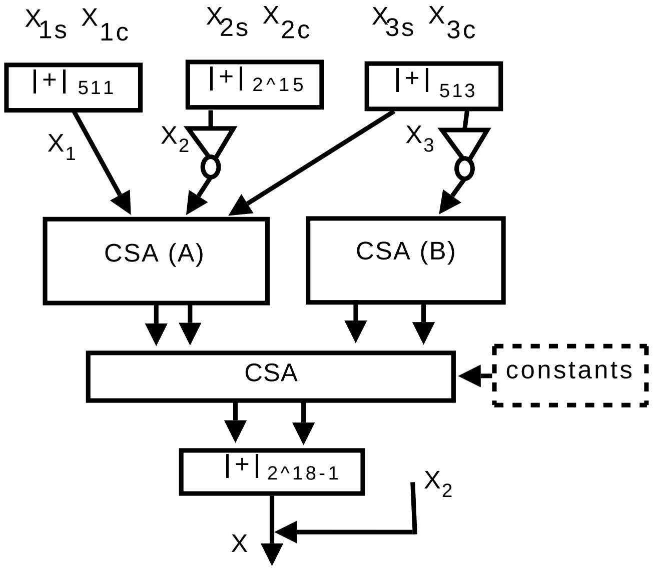


Fig. 7. RNS-to-binary Converter

G. *RNS-to-Binary (Reverse) Conversion*

For the RNS-to-Binary converter, we employ the new Chinese Remainder Theorem-I [45], similar to that in [46]. The New CRT-I states that, given a residue number $(x_1, x_2, ..., x_n)$ for the moduli set $(P_1, P_2, ..., P_n)$, the binary number X can be calculated as follows:

$$X = x_1 + P_1 \cdot |k_1(x_2 - x_1) + \sum_{i=2}^{n-1} k_i(\prod_{j=2}^{i} P_j)(x_{i+1} - x_i)|_{\prod_{j=2}^{n} P_j} \qquad \text{(III.5)}$$

where,

$k_1P_1 \equiv |1|_{P_2P_3...P_n}$,

$k_2P_2P_3 \equiv |1|_{P_3...P_n}$, and

$k_{n-1}P_1P_2...P_{n-1} \equiv |1|_{P_n}$.

The smallest k_i satisfying the above congruences are chosen. For $n = 3$, we have,

$$X = x_1 + P_1 \cdot |k_1(x_2 - x_1) + k_2P_2(x_3 - x_2)|_{P_2P_3}$$

For our moduli-set, we take $P_1 = 2^{15}$, $P_2 = 2^9 + 1$, $P_3 = 2^9 - 1$ and correspondingly, $k_1 = 2^3$ and $k_2 = 2^2$. Following a derivation similar to [46], we can calculate X as:

$$X = x_2 + 2^{15}|A + B|_{2^{18}-1} \tag{III.6}$$

where,

$A = 2^3\lfloor\frac{x_1+2(\overline{x_2})+x_3+2^9z_0}{2}\rfloor + (2^3 - 1)$

$B = 2^{12}\lfloor\frac{x_1+(\overline{x_3})+(2^{10}-1)}{2}\rfloor + 2^4(2^3 - 1) + 2^2z_0$

and $z_0 = x_{30} \bigoplus x_{10}$, x_{30} and x_{10} are the LSBs of x_3 and x_1. Note that x_1, x_2 and x_3 are 9-bit, 15-bit and 10-bit numbers respectively. Using CSA trees, and bit-alignment, we can compute A and B.

The other terms of A and B are constants which are inserted at the correct bit positions. To compute $|A + B|_{2^{18}-1}$ in Equation (III.6), we use two 18-bit Kogge-Stone adders in parallel, one computing $Y_1 = A + B$ and other $Y_2 = A + B + 1$. If Y_1 generates a carry, Y_2 is the result of modulo addition, else it is Y_1. Finally, since, x_2 is a 15-bit number, it can just be appended to the bits of the second term in Equation (III.6) to get X. The organization is shown in Figure 7.

H. *Synthesis Of RNS Units*

The RNS functional units are implemented in Verilog and synthesized using the 0.18μ OSU standard cell library with the *Cadence Encounter*® *RTL Compiler*. The designs were initially synthesized to achieve the lowest possible delay. Then timing constraints on the various functional units were relaxed for power and area optimizations such that the integral ratios were maintained for the delays. We also synthesized 2's complement 32-bit Brent-Kung adder and 32-bit multiplier whose structural HDL descriptions are provided by ARITH project [15].

TABLE III
SYNTHESIS RESULTS

Arithmetic Unit	**Area (λ^2)**	**Power (mW)**	**Delay (ns)**
32-bit Brent-Kung Adder	11032	1.03	1.4
Modulo $2^9 - 1$ Adder	2464	0.29	0.7
Modulo 2^{15} Adder	3982	0.45	0.7
Modulo $2^9 + 1$ Adder	5841	0.67	0.7
Total: Modulo Adders	12287	1.41	–
32-bit Multiplier	211927	48.52	4.2
Modulo $2^9 - 1$ Multiplier	21647	4.97	2.8
Modulo 2^{15} Multiplier	30850	7.13	2.8
Modulo $2^9 + 1$ Multiplier	50915	11.95	2.8
Total: Modulo Multipliers	103412	24.05	–
Binary-to-residue $2^9 - 1$	2287	0.26	0.7
Binary-to-residue 2^{15}	–	–	–
Binary-to-residue $2^9 + 1$	5081	0.61	0.7
Total: Binary-to-residue	7388	0.87	–
Residue-to-binary	52382	11.60	2.8

From the results (Table III), we note that:

- RNS addition and multiplication are $2X$ and $1.5X$ faster than their 2's complement counterparts; and

- Power consumption of RNS multiplication is *less than half* of its 2's complement counterpart.

Hence we can design a binary-to-RNS conversion unit for single-cycle conversion of two binary operands. Similarly, a single-cycle 3-operand RNS adder can be designed. Multiplication and RNS-to-binary conversion are 2 cycle operations.

I. *Processor Model*

We integrate the functional units above into the pipeline of a typical RISC processor as shown in Figure 8.

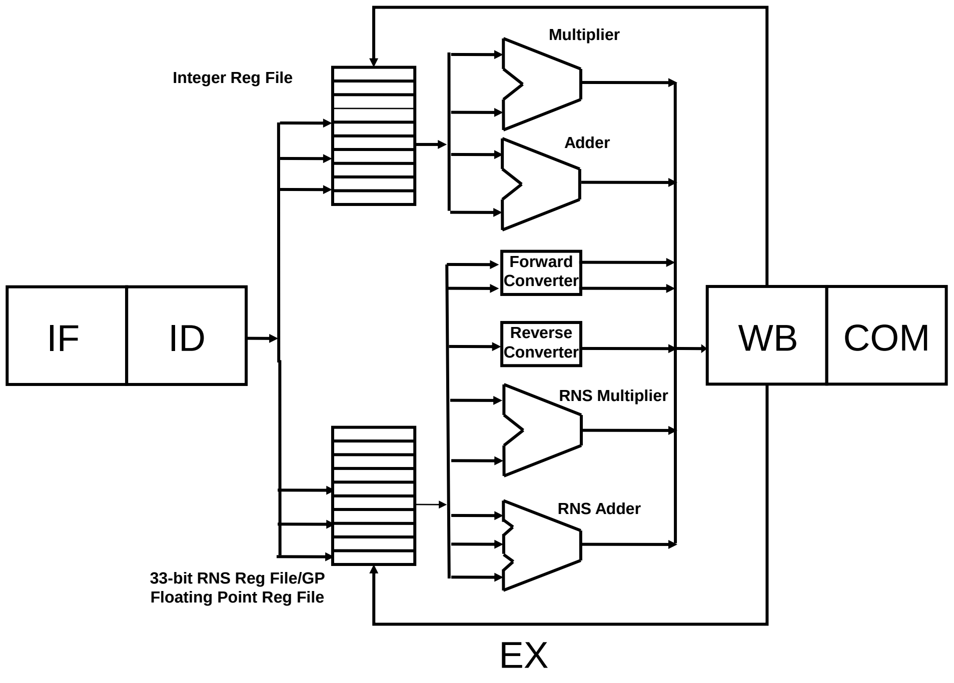


Fig. 8. Typical RISC Pipeline with RNS Functional Units

I.1. *Case Study: Matrix Multiplication*

Computation in RNS can be beneficial if the program structure is such that there are enough computation operations whose faster execution compensates for the conversion overheads. For multiplication of two $N \times N$ matrices A and B, each cell of the result matrix C is computed as follows:

$C_{ij} = A_{i0} \cdot B_{0j} + A_{i1} \cdot B_{1j} + \cdots + A_{iN} \cdot B_{Nj}$

The same row in matrix A is used in the computation of an entire row in the result matrix C, and hence before computing row i of matrix C, we can forward-convert the corresponding row of matrix A and store it. This avoids the repeated forward-conversion of a row in A for computing every cell of the corresponding row in C. The computational requirements of $C = A \times B$ are shown in Table IV

TABLE IV
Computational requirements for Matrix Multiplication

Operation	**Amount RNS**	**Amount Normal**	**Cycles RNS**	**Cycles Normal**
Loads (A)	N^2	N^2	N^2	N^2
Loads (B)	N^3	N^3	N^3	N^3
FC (A)	$\frac{N^2}{2}$	0	$\frac{N^2}{2}$	0
FC (B)	$\frac{N^3}{2}$	0	$\frac{N^3}{2}$	0
Mults	N^3	N^3	$2 \cdot N^3$	$3 \cdot N^3$
Adds	$\frac{N^3}{2}$	$N^2(N-1)$	$\frac{N^3}{2}$	$N^2(N-1)$
RC	N^2	0	$2 \cdot N^2$	0
Stores	N^2	N^2	N^2	N^2

Therefore the normalized benefit is $\frac{(CyclesRNS - CyclesNormal)}{(CyclesNormal)} = \frac{(N^3 - 3.5N^2)}{(4N^3 + 4.5N^2)}$, which is positive for $N \geq 4$, and 25% as $N \rightarrow \infty$. Thus we can see that in spite of overheads, RNS computation has significant performance improvements over 2's complement computation. Although this analysis does not model the complexities of pipeline execution, it highlights essential aspects of RNS computation.

An important observation from Table IV, is that the number $(O(n^3))$ of forward conversions is an order of magnitude greater than that $(O(n^2))$ of reverse conversions. So, although reverse conversion is a costlier operation, forward conversion tends to have a greater contribution to overhead due to its sheer volume.

IV. Compilation for Residue Number Extension

This chapter describes the basic compilation technique developed for mapping application code to RNS instructions and also discusses improvements to it. Our goal is to increase the number of computations that can be done in the RNS domain while reducing the overhead of conversions.

A. *Problem Definition*

In order to define the problem of code mapping to RNS, we introduce certain definitions.

Dataflow Graph (DFG) - is a graph $G(V, E)$, where each vertex $v \in V$ represents a computation and each edge $e = (u, v) \in E$ represents a dependency between u and v where v consumes a value produced by u. We also define $PRED(v) : u \in V/(u, v) \in E$ and $SUCC(v) : w \in V/(v, w) \in E$.

RNS Eligible Node (REN) - For a DFG $G(V, E)$, a node $v \in V$ is an *RNS Eligible Node* if the computation it represents is an addition, multiplication or subtraction of integer values.

RNS Eligible Subgraph (RES) - For a DFG $G(V, E)$, a subgraph $G_{RES}(V_{RES}, E_{RES})$ is called *RNS Eligible* if $\forall v \in V_{RES}$, v is an RNS Eligible Node.

Maximal RNS Eligible Subgraph (MRES) - For a DFG $G(V, E)$, an RNS Eligible Subgraph $G_{RES}(V_{RES}, E_{RES})$ is *Maximal* if $\forall v \in V_{RES}$, there does not exist any edge $e = (u, v)$ or $e\prime = (v, u)$, such that e is an RNS Eligible Node. A maximal RNS eligible subgraph G_{MRES} is *profitable* with respect to a metric M (power, cycle time, etc.), if mapping G_{MRES} to RNS instructions improves the metric M, when applied to G_{MRES}.

With this, the RNS code mapping problem can be defined as follows:

Given a dataflow graph $G(V, E)$, find out all profitable Maximal RNS Eligible Subgraphs

Each profitable MRES can be mapped to the RNS extension, with appropriate conversion operations at its boundaries.

B. *Basic Code Mapping Algorithm*

B.1. *Top Level Algorithm*

The basic code mapping algorithm (Algorithm 1) takes as input the DFGs of all basic blocks in the program and traverses each DFG to mark all MRESs. For each marked MRES, it then estimates its profitability with respect to cycle time, using a profit model that includes conversion overheads. Profitable MRESs are then mapped to utilize RNS-specific instructions.

Algorithm 1 MAP2RNS

```
Require: Set $S_{BB}$ consisting of DFGs for every basic block
Ensure: All graphs in $S_{BB}$, with all profitable MRESs mapped to RNS.
1: for all $G \in S_{BB}$ do
2:    Do FIND_MRES($G$) [Algorithm 2]
3:    for all MRESs, $G_{MRES}$ found in $G$ do
4:       profit $\leftarrow$ ESTIMATE_PROFIT($G_{MRES}$) [Algorithm 3]
5:       if profit $> 0$ then
6:          Transform $G_{MRES}$ to use RNS instructions.
7:       end if
8:    end for
9: end for
```

The operation is illustrated in Figure 9

B.2. *Finding Maximal RNS Eligible Subgraphs*

The algorithm FIND_MRES (Algorithm 2), finds all Maximal RNS Eligible Subgraphs, given the DFG of a basic block. An RNS Eligible Node in the basic block is picked as the seed node and the MRES is grown by performing a breadth-first search ignoring the direction of the edges.

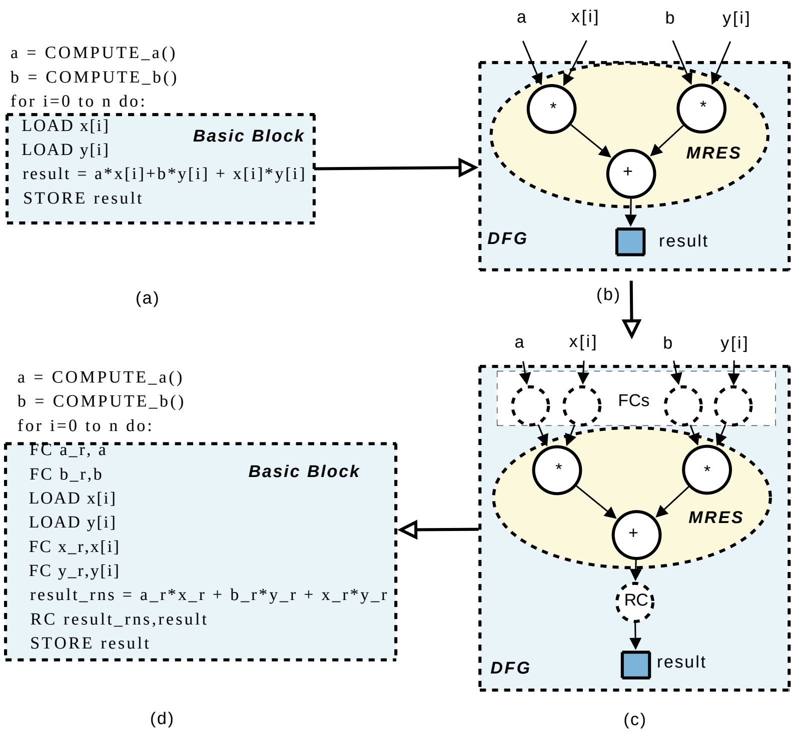


Fig. 9. Mapping to RNS

Theorem 1. *Given a DFG $G(V,E)$, algorithm FIND_MRES finds all subgraphs of G that are Maximal RNS Eligible Subgraphs.*

Proof. We state and prove the following three statements:

The subgraphs FIND_MRES marks are RNS Eligible Subgraphs

Lines 5-16 of FIND_MRES are the same as a Breadth First Traversal (BFT) on 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.

The RNS Eligible Subgraphs FIND_MRES marks, are maximal.

Suppose that a RES $G_{RES}(V_{RES}, E_{RES})$, that is a subgraph of the DFG $G(V,E)$ of the basic block, is not maximal. Then, $\exists v \in V_{RES}$ s.t. $\exists u \in PRED(v) \cup SUCC(v)$ s.t. u is an

Algorithm 2 FIND_MRES

```
Require: $G(V, E)$: DFG of basic block with nodes $V$ and edges $E$.
Ensure: Graph $G$, with all MRESs marked.
1: Let Q be a queue of nodes.
2: MRES_ID $\leftarrow$ 1
3: while All nodes in $V$ 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:       $v \leftarrow$ 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:               $u.mres \leftarrow$ MRES_ID
12:               Add $u$ to Q.
13:            end if
14:         end if
15:      end for
16:   end while
17:   MRES_ID $\leftarrow$ MRES_ID + 1
18: end while
19: Return G.
```

RNS Eligible Node, but $u \notin V_{RES}$.

Now, since $v \in V_{RES}$, it must have been queued at Line 11 and eventually unqueued at Line 6. There are two possibilities:

- u was already visited, in which case, it would also have been marked to be part of RES 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 RES, since it is RNS Eligible.

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

All possible Maximal RNS Eligible Subgraphs are marked.

Suppose that there is an MRES $G_{MRES}(V_{MRES}, E_{MRES})$ of $G(V, E)$ that was unmarked by

FIND_MRES. Since V_{MRES} only contains RNS Eligible Nodes and since they are unmarked, it means they are 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 were not a subgraph of the original DFG, which is a contradiction. Thus all MRES that exist in the DFG G, are marked. $\square$

B.3. *Estimating Profitability Of A MRES*

The algorithm ESTIMATE_PROFIT estimates the profitability of RNS conversion of an MRES $G_{MRES}(V_{MRES}, E_{MRES})$, with respect to cycle-time. Since the RNS extension includes a single-cycle 3-input addition of RNS operands, we can pair adds of the form $x + y + z$ and transform them into a single 3-input addition as shown in Figure 10. The algorithm MARK_ADD_PAIRS, traverses the MRES and marks all eligible pairs of additions. Existence of any dataflow (u, v), such that $u \notin V_{MRES}$ and $v \in V_{MRES}$ implies that the value produced by u needs to be forward converted, while that of any dataflow (u', v'), such that $u' \in V_{MRES}$ and $v' \in V_{MRES}$ implies that the value produced by v' needs to be reverse converted. The following rules are applied to calculate the profit:

- Every pair of forward conversions is an overhead of 1 cycles.
- Every reverse conversion is an overhead of 2 cycles.
- Every 3-operand addition is a profit of 1 cycle.
- Every multiplication is a profit of 1 cycle.

C. *Improvements Upon Basic Algorithm*

C.1. *Forward Conversions Within Loops*

Consider the example shown in Figure 11(a). For sake of simplicity we ignore the profitability of conversion and map the computation of *result* to RNS with the basic algorithm

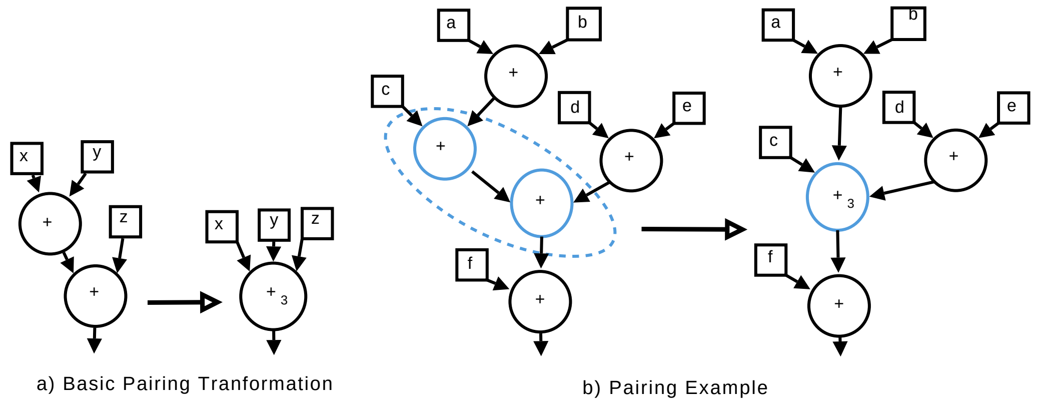
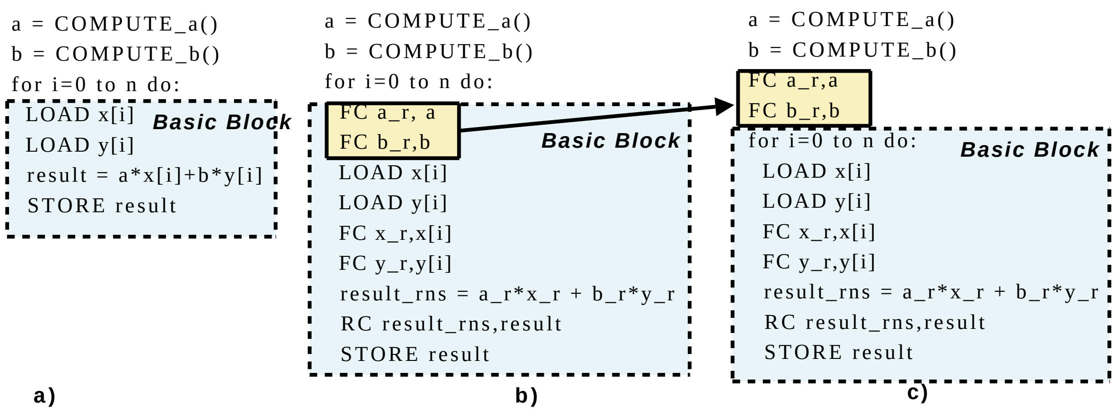


Fig. 10. Pairing of Additions



a)
```
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
```

b)
```
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
```

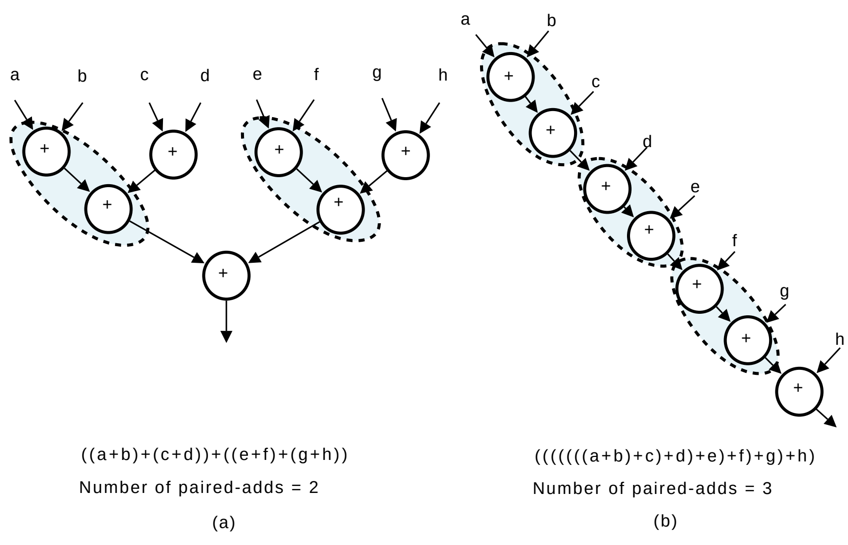
c)
```
a = COMPUTE_a()
b = COMPUTE_b()
FC a_r,a
FC b_r,b
for i=0 to n do:
  LOAD x[i]
  LOAD y[i]
  FC x_r,x[i]
  FC y_r,y[i]
  result_rns = a_r*x_r + b_r*y_r
  RC result_rns,result
  STORE result
```

Fig. 11. Moving Foward Conversions Out of Loops

(Figure 11(b)). Since the basic algorithm works purely at the basic block level, it inserts the forward conversions of a and b within the basic block in which they are used. However, since a and b are only written outside and not within the basic block, their forward conversions can also be moved outside the basic block (Figure 11(c)). This prevents unnecessary forward conversions at the boundaries of the basic block.

C.2. *Improved Pairing Of Additions*

The structure of addition expressions in the DFG constructed by the compiler is not amenable for optimal pairing of additions. Ideally for an expression $\sum_{i=0..n} a_i$, containing



a
b
c
d
e
f
g
h
((a+b)+(c+d))+((e+f)+(g+h))
Number of paired-adds = 2
(a)
(((((((a+b)+c)+d)+e)+f)+g)+h)
Number of paired-adds = 3
(b)

Fig. 12. Linearizing Adds

n additions, the maximum number of add-pairs possible is $\lfloor \frac{n}{2} \rfloor$. Consider the DFG in Figure 12(a). There are 7 additions and although ideally there should be 3 pairs ($\lfloor \frac{7}{2} \rfloor$), the maximum pairs that can be formed for this organization of additions is 2. However, if we make a linear structure of additions, as in Figure 12(b), then optimal pairing can be done. We formalize these concepts below.

Addition Cluster - For a DFG $G(V, E)$, a subgraph $G_{ac}(V_{ac}, E_{ac})$ is an *Addition Cluster* if the $\forall v \in V_{ac}$, v is represents an integer addition. An add cluster is a DAG that represents a series summation of integers.

Addition Tree - An *Addition Tree*, is an Addition Cluster which is a binary tree.

Fully Linear Addition Tree - A *Fully Linear Addition Tree*, representing an n additions is an Addition Tree with n levels.

The algorithm LINEARIZE_ADDS (Algorithm 5), an algorithm that converts an addition tree to a fully linear addition tree. The algorithm carries out the transformation shown in Figure 13 repeatedly on the addition tree until it is fully linear.

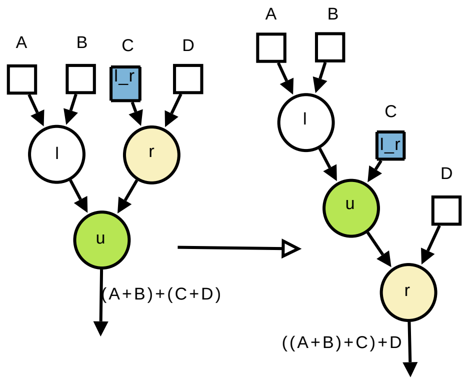


Fig. 13. Addition Linearizing Transformation

Theorem 2. *LINEARIZE_ADDS transforms a given addition tree $G_{at}(V_{at}, E_{at})$, representing n series additions, to a linearized addition tree.*

Proof. Every single application of the transformation shown in Figure 13, increases the number of levels in G_{at} by 1. Therefore, at the most n applications of the transformation are required to transform it to a addition tree having n levels. This is, by definition, a linearized addition tree. □

Algorithm 3 ESTIMATE_PROFIT

Require: $G(V, E)$: DFG for basic block.
Require: $G_{MRES}(V_{MRES}, E_{MRES})$: MRES to estimate profit for.
Require: cFC,cRC: Cycles that one forward and reverse conversion costs, respectively
Ensure: Estimated profit P on mapping G_{MRES} to RNS.

```
1: Let $nGain$ denote number of cycles saved by RNS mapping.
2: Let $nOvhd$ denote the number of cycles spent on conversion overheads.
3: Let $nFC$ denote number of forward conversions.
4: Let $nRC$ denote number of reverse conversions.
5: Let $P$ denote the estimated profit in cycle time.
6: $nFC \leftarrow 0$,$nRC \leftarrow 0$,$P \leftarrow 0$
7: MARK_ADD_PAIRS($G_{MRES}$) [Algorithm 4]
8: while there is atleast one unvisited node in $V_{MRES}$ do
9:    Pick an unvisited node $v \in V_{MRES}$
10:   Mark $v$ as visited.
11:   for all nodes $u \in V, (u, v) \in E$ && $u \notin V_{MRES}$ do
12:      $nFC \leftarrow nFC + 1$
13:   end for
14:   if $\exists w \in V, (v, w) \in E$ && $w \notin V_{MRES}$ then
15:      $nRC \leftarrow nRC + 1$
16:   end if
17:   $nOvhd \leftarrow (\frac{nFC}{2} + nFC \mod 2) + 2 \times nRC$
18:   if $v$ is an addition && $(v, w)$ is a paired add for some $w$ then
19:      $nGain \leftarrow nGain + 1$
20:   end if
21:   if $v$ is a multiplication then
22:      $nGain \leftarrow nGain + 1$
23:   end if
24:   $P \leftarrow nGain - nOvhd$
25: end while
26: Return $P$.
```

Algorithm 4 MARK_ADD_PAIRS

Require: $G(V, E)$: DFG for basic block.
Require: $G_{MRES}(V_{MRES}, E_{MRES})$: MRES in which to mark add-pairs.
Ensure: All possible add-pairs in G_{MRES} are marked.

```
1: for all nodes $u \in V_{MRES}$ that are not yet part of an add-pair do
2:    if $u$ is an addition then
3:       if $\exists v \in V_{MRES}, s.t. (u, v) \in E_{MRES}$ && $v$ is an addition && $\nexists w \in V, w \neq v, s.t. (u, w) \in E$ then
4:          Mark the add-pair $(u, v)$.
5:       end if
6:    end if
7: end for
```

Algorithm 5 LINEARIZE_ADDS

```
Require: $G(V, E)$: DFG for basic block.
Require: $G_{at}(V_{at}, E_{at})$: Addition Tree to linearize
Ensure: $G_{at}$ is a Linearized Addition Tree.
 1: while $G_{at}$ contains an addition node $u$ s.t. both its operands are also results of additions
    do
 2:   $l \leftarrow$ Left predecessor of $u$, $r \leftarrow$ Right predecessor of $u$, $l_r \leftarrow$ Left predecessor of $r$.
 3:   Let $r \leftarrow$ Right predecessor of $u$.
 4:   Let $l_r \leftarrow$ Left predecessor of $r$.
 5:   Remove edges $(r, u)$ and $(r_l, r)$.
 6:   Let $S \leftarrow SUCC(u)$.
 7:   for all $w \in S$ do
 8:     Remove edge $(u, w)$.
 9:   end for
10:   Add edge $(l_r, u)$ and $(u, r)$.
11:   for all $w \in S$ do
12:     Add edge $(r, w)$.
13:   end for
14: end while
```

V. Experimental Results

A. *Simulation Model*

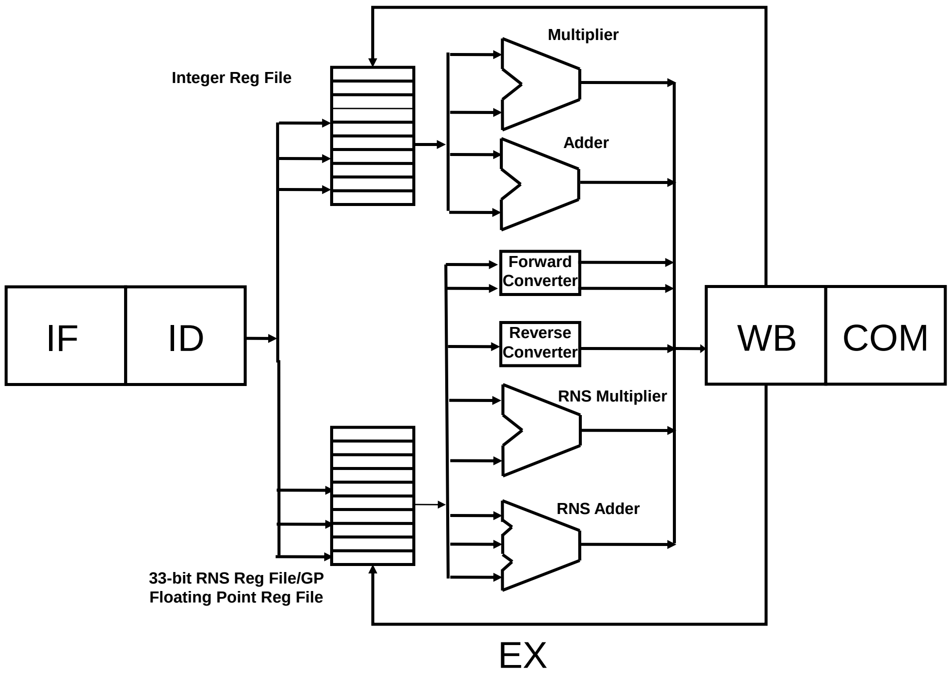


Fig. 14. Typical RISC Pipeline with RNS Functional Units

Using the synthesis results discussed in Section H (repeated in Table V for convenience), we incorporate the RNS functional unit model in the SimpleScalar simulator [1]. The organization is shown in Figure 14. Delay and power of RNS functional units are considered according to Table V. The simulation is run in an in-order execution as typical for embedded microprocessors.

B. *Benchmark Kernels*

In order to evaluate our RNS extension and compiler extension we choose benchmark kernels that represent core operations in a wide variety of digital signal processing, image processing, graphics and multimedia algorithms. Convolution, 2D-Discrete Cosine Transform (2D-DCT), matrix multiplication, finite impulse response (FIR), Gaussian Smoothing,

TABLE V
Synthesis Results

Arithmetic Unit	**Area (λ^2)**	**Power (mW)**	**Delay (ns)**
32-bit Brent-Kung Adder	11032\\11	1.03	1.4
Modulo $2^9 - 1$ Adder	2464	0.29	0.7
Modulo 2^{15} Adder	3982	0.45	0.7
Modulo $2^9 + 1$ Adder	5841	0.67	0.7
Total: Modulo Adders	12287	1.41	–
32-bit Multiplier	211927	48.52	4.2
Modulo $2^9 - 1$ Multiplier	21647	4.97	2.8
Modulo 2^{15} Multiplier	30850	7.13	2.8
Modulo $2^9 + 1$ Multiplier	50915	11.95	2.8
Total: Modulo Multipliers	103412	24.05	–
Binary-to-residue $2^9 - 1$	2287	0.26	0.7
Binary-to-residue 2^{15}	–	–	–
Binary-to-residue $2^9 + 1$	5081	0.61	0.7
Total: Binary-to-residue	7388	0.87	–
Residue-to-binary	52382	11.60	2.8

etc are central to these domains. Therefore, we use the following kernels as our benchmark set to evaluate our RNS extension and compiler techniques.

- Matrix Multiplication
- Finite Impulse Response filter
- Gaussian Smoothing
- 2-dimensional Discrete Cosine Transform
- Livermore Loops - Hydro kernel
- Livermore Loops - Integrate Predictor kernel

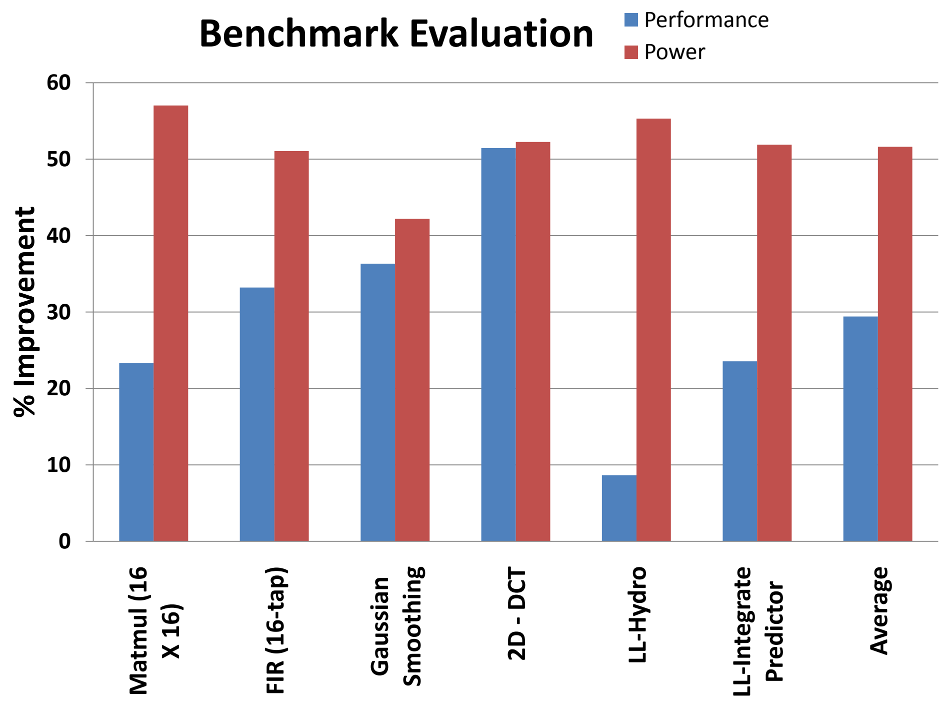


Fig. 15. Execution times for manually-optimized benchmarks

C. *Simulation Results*

C.1. *Evaluation Of Manually Mapped Benchmarks*

The above benchmarks are optimized manually to incorporate RNS computation instructions and conversion instructions as applicable. The simulator functional unit configuration consists of 1 RNS adder, 1 RNS multiplier, 1 forward converter and 1 reverse converter as well as 1 2's complement adder and multiplier each. Simulation results (Figure 15) show an average performance improvement of 29.4%. Furthermore, power reduction up to 57.03% (16×16 matrix multiplication) and 51.6% in the average, can be obtained in the functional units using RNS. This substantial power reduction is the direct result of the lower power consumption seen in RNS functional units. While these are considerable power savings, we acknowledge that the power consumption in arithmetic units is a small percentage of the total power consumption of a microprocessor. Hence these improvements might not form a significant portion of total power consumption. Since several low power and low cost embedded processors are typically employed in execution of applications having these kernels at their core, the advantages of an RNS-equipped processor are immediately obvious.

C.2. *Evaluation Of Compiler Mapped Code*

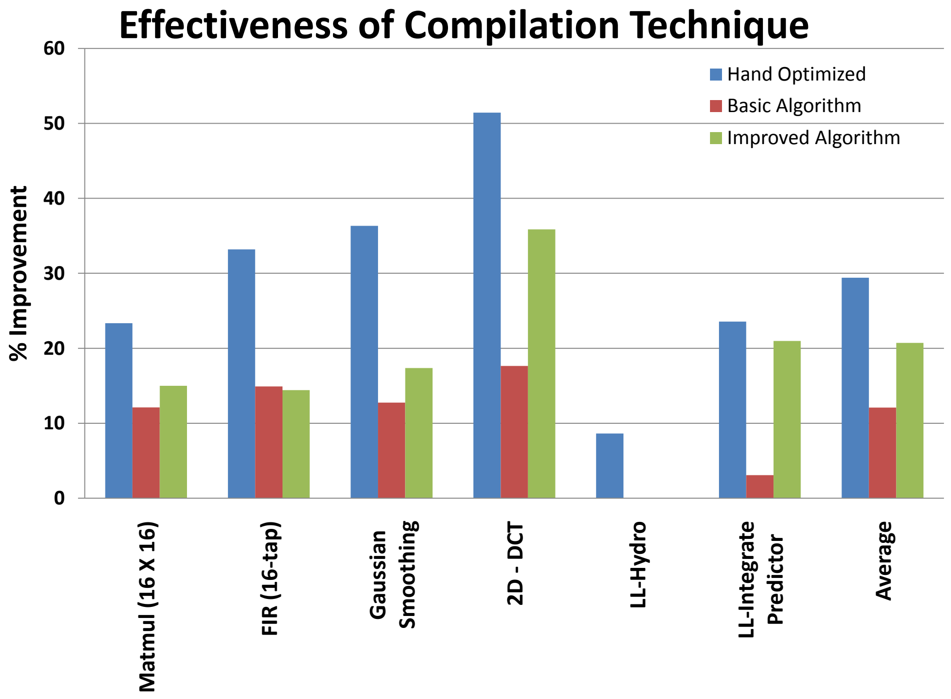


Fig. 16. Percentage improvement in execution times for compiler-optimized benchmarks

Next, we evaluate our compiler technique to see the performance and power benefits obtained by automation of mapping. We compare the execution times of RNS-optimized benchmarks with original benchmarks. To gauge the effectiveness of our compiler technique we also compare performance with manually optimized benchmarks of Section C.1. While hand-optimized code yields an average performance improvement of 29.4%, our basic technique achieves a 12.1% improvement as conversion overheads are not effectively managed. The improved technique, with its better handling of conversion overheads and improved pairing of additions, is able to achieve 20.7% performance improvement on average.

Note that there are no compiler-based simulation results for the LL-hydro benchmark. This is because our profit model does not report a profit for that benchmark and hence no mapping of code to RNS is done by the compiler.

C.3. *Evaluation across design corners*

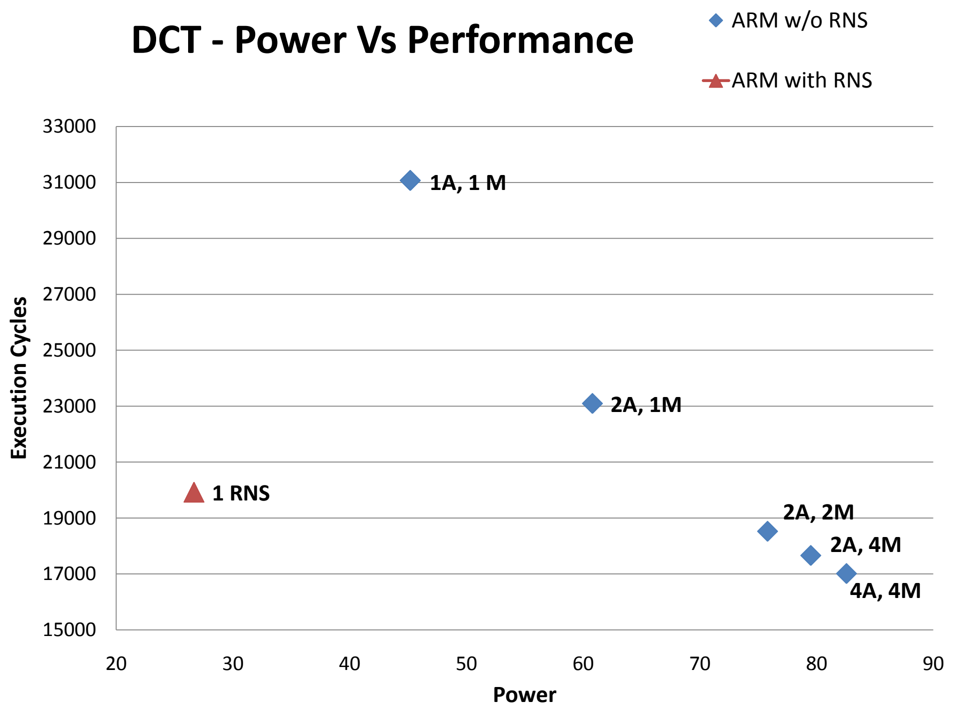


Fig. 17. Power vs. Performance - DCT benchmark

We compare runtime vs power consumption of the RNS-equipped ARM processor with processors having varying number of 2's complement adders and multipliers. From Figure 17, we see that the performance-to-power ratio 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, RNS is able to achieve significantly greater performance by utilizing lesser power owing to the the presence of faster functional units and careful co-design. Also, in general, application performance saturates as resources are increased, when existing instruction level parallelism in the application is exhausted. RNS breaks this saturation barrier by exploiting parallelism that the RNS representation inherently creates, resulting in the significant performance improvements at the expense of much lower power.

VI. Conclusion and Future Work

In this work, we have demonstrated that by synergistic system design, it is possible to overcome challenges of incorporating RNS arithmetic in embedded microprocessors. The 51.6% power and 29.4% performance improvements we obtain for several hand-optimized DSP and image processing kernels, demonstrate that RNS arithmetic is not restricted only to the domain of application specific non-programmable designs but also holds great potential as an efficient computational platform in embedded RISC processors.

We also developed, what is to the best of our knowledge, the first compilation technique targeted towards exploiting the benefits of RNS computation in processors. Our basic technique lends an average performance improvement of 12%. By improving the basic technique to amortize the overhead of forward conversions and to improve pairing of additions, we realize an average performance improvement of 20.72%.

Interestingly, while in the application specific hardware the reverse conversion is considered to be the dominant bottleneck of RNS hardware and the overhead of inexpensive and easy to parallelize forward conversions would typically be neglected, in a programmable processor it is the forward conversions that are likely to limit the achievable speedup of certain algorithms. While their parallel execution is limited by memory interface, with limited size of register file, they are likely to outnumber reverse conversions by orders of magnitude. While compiler techniques can be devised to hide the forward conversion latencies, a possible exploration direction is to move them out of the processing pipeline — e.g. into the cache memory between levels 1 and 2 or integrating them with loads and stores.

The compiler technique in this work can be extended in several aspects:

- Improve the profit model to model instruction execution more accurately.
- Extend analysis of programs to larger blocks of the program graph, i.e., at the hyper-

block or superblock level instead of just at the basic block level, so that subgraphs containing more number of RNS operations can be mapped for greater performance and power benefits.

- Explore the use of code annotation for improved program analysis. Language constructs can be introduced to guide the compiler in identifying blocks of code that can be profitably mapped to RNS.

REFERENCES

[1] Simplescalar Simulator. http://www.simplescalar.com.

[2] J. Bajard and L. Imbert. A Full RNS Implementation of RSA. *IEEE Transactions On Computers*, pages 769–774, 2004.

[3] D. Banerji. A novel implementation method for addition and subtraction in residue number systems. *Transactions on Computers*, C-23(1):106–109, 1974.

[4] R. Brent and H. Kung. A regular layout for parallel adders. *Transactions on Computers*, C-31(3):260–264, 1982.

[5] R. Chaves and L. Sousa. RDSP: A RISC DSP based on residue number system. *Proc. Euro. Symp. Digital System Design: Architectures, Methods, and Tools*, pages 128–135, Sep. 2003.

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

[7] C. Efstathiou, H. Vergos, and D. Nikolos. Fast Parallel-Prefix Modulo $2^n + 1$ Adders. *IEEE Transactions On Computers*, pages 1211–1216, 2004.

[8] W. Freking and K. Parhi. Low-Power FIR Digital Filters Using Residue Arithmetic. *Proc. 31st Asilomar Conference on Signals, Systems, and Computers*, 1:128–135, 1997.

[9] M. Griffin and F. Taylor. A Residue Number System Reduced Instruction Set Computer (RISC) Concept. *Proc. Inter. Conf. Acoustis, Speech, and Signal Processing*, pages 2581–2584, 1989.

[10] A. Hiasat. New memoryless, mod $(2^n \pm 1)$ residue multiplier. *Electronics Letters*, 28(3):314–315, 1992.

[11] A. Hiasat. New Efficient Structure for a Modular Multiplier for RNS. *IEEE Transactions On Computers*, pages 170–174, 2000.

[12] A. Hiasat. RNS arithmetic multiplier for medium and large moduli. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, 47(9):937–940, 2000.

[13] A. Hiasat. High-Speed and Reduced-Area Modular Adder Structures for RNS. *IEEE Transactions On Computers*, pages 84–89, 2002.

[14] R. Hohne and R. Siferd. A programmable high performance processor using the residue number system and CMOS VLSI technology. *Aerospace and Electronics Conference, 1989. NAECON 1989., Proceedings of the IEEE 1989 National*, pages 41–43, 1989.

[15] 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.

[16] H.T.Vergos. A 200-MHz RNS Core. *European Conference on Circuit Theory and Design*, August 2001.

[17] C. Huang, D. Peterson, H. Rauch, J. Teague, and D. Fraser. Implementation of a fast digital processor using the residue number system. *Circuits and Systems, IEEE Transactions on*, 28(1):32–38, 1981.

[18] W. Jenkins and B. Leon. The use of residue number systems in the design of finite impulse response digital filters. *Circuits and Systems, IEEE Transactions on*, 24(4):191–201, 1977.

[19] B. Kirsch, P. Turner, U. Center, and P. Warminster. Adaptive beamforming using RNS arithmetic. *Computer Arithmetic, 1993. Proceedings., 11th Symposium on*, pages 36–43.

[20] P. Kogge and H. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 22(8):786–793, 1973.

[21] Y. Liu and E. Lai. Design and implementation of an RNS-based 2-D DWT processor. *Consumer Electronics, IEEE Transactions on*, 50(1):376–385, 2004.

[22] H. Loomis and B. Sinha. High-speed recursive digital filter realization. *Circuits, Systems, and Signal Processing*, 3(3):267–294, 1984.

[23] 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.

[24] D. Miller and J. Polky. An implementation of the LMS algorithm in the residue number system. *Circuits and Systems, IEEE Transactions on*, 31(5):452–461, 1984.

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

[26] P. Mohan. *Residue Number Systems: Algorithms and Architectures.* Kluwer Academic Publishers, 2002.

[27] J. Ramirez, P. Fernandez, U. Meyer-Base, F. Taylor, A. Garcia, and A. Lloris. Index-based rns dwt architectures for custom ic designs. *Proc. IEEE Workshop on Signal Processing Systems*, pages 70–79, 26–28 Sept. 2001.

[28] J. Ramirez, A. Garcia, P. Fernandez, L. Parrilla, and A. Lloris. Implementation of canonical and retimed rns architectures for the orthogonal 1-d dwt over fpl devices. *Conference Record of the Thirty-Fourth Asilomar Conference on Signals, Systems and Computers*, 1:384–388, 29 Oct.–1 Nov. 2000.

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

[30] J. Ramirez, A. Garcia, U. Meyer Base, F. Taylor, P. Fernandez, and A. Lloris. Design of rns-based distributed arithmetic dwt filterbanks. *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '01)*, 2:1193–1196, 7–11 May 2001.

[31] J. Ramirez, A. Garcia, L. Parrilla, A. Lloris, and P. Fernandez. Implementation of rns analysis and synthesis filter banks for the orthogonal discrete wavelet transform over fpl devices. *Proc. 43rd IEEE Midwest Symposium on Circuits and Systems*, 3:1170–1173, 8–11 Aug. 2000.

[32] D. Schinianakis, A. Kakarountas, and T. Stouraitis. A New Approach to Elliptic Curve Cryptography: an RNS Architecture. *IEEE Mediterranean electrotechnical conference, Benalmádena (Málaga), Spain*, pages 1241–5, 2006.

[33] N. Shanbhag and R. Siferd. A single-chip pipelined 2-D FIR filter using residue arithmetic. *Solid-State Circuits, IEEE Journal of*, 26(5):796–805, 1991.

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

[35] M. Soderstrand. A high-speed low-cost recursive digital filter using residue number arithmetic. *Proceedings of the IEEE*, 65(7):1065–1067, 1977.

[36] M. Soderstrand. A new hardware implementation of modulo adders for residue number systems. *IEEE Press Reprint Series*, pages 72–75, 1986.

[37] M. Soderstrand and B. Sinha. A pipelined recursive residue number system digital filter. *Circuits and Systems, IEEE Transactions on*, 31(4):415–417, 1984.

[38] F. Taylor. Large moduli multipliers for signal processing. *Circuits and Systems, IEEE Transactions on*, 28(7):731–736, 1981.

[39] F. Taylor. A VLSI Residue Arithmetic Multiplier. *IEEE Transactions on Computers*, 31(6):540–546, 1982.

[40] F. Taylor. An overflow-free residue multiplier. *IEEE Transactions On Computers*, 32(5):501–504, 1983.

[41] T.Stouratitis and V.Paliouras. Considering the alternatives in lowpower design. *IEEE Circuits and Devices*, pages 23–29, 2001.

[42] C. Wang. New bit-serial VLSI implementation of RNS FIR digital filters. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, 41(11):768–772, 1994.

[43] W. Wang, M. Swamy, M. Ahmad, and Y. Wang. The applications of the new chinese remainder theorems for three moduli sets. *Proc. IEEE Canadian Conference on Electrical and Computer Engineering*, 1:571–576 vol.1, 1999.

[44] Y. Wang. New chinese remainder theorems. *Conference Record of the Thirty-Second Asilomar Conference on Signals, Systems & Computers*, 1:165–171 vol.1, 1998.

[45] Y. Wang. Residue-to-binary converters based on new chinese remainder theorems. 47(3):197–205, 2000.

[46] Y. Wang, X. Song, M. Aboulhamid, and H. Shen. Adder based residue to binary number converters for $(2^n - 1, 2^n, 2^n + 1)$. 50(7):1772–1779, 2002.

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

[48] R. Zimmermann. Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication. *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 158–167, 1999.