Extending Moore's Law via Computationally Error Tolerant Computing

BOBIN DENG, Georgia Institute of Technology*
SRISESHAN SRIKANTH, Georgia Institute of Technology*
ERIC R. HEIN, Georgia Institute of Technology
THOMAS M. CONTE, Georgia Institute of Technology
ERIK DEBENEDICTIS, Sandia National Laboratories**
JEANINE COOK, Sandia National Laboratories
MICHAEL P. FRANK, Sandia National Laboratories

Dennard scaling has ended. Lowering the voltage supply (V_{dd}) to sub volt levels causes intermittent losses in signal integrity, rendering further scaling (down) no longer acceptable as a means to lower the power required by a processor core. However, it is possible to correct the occasional errors caused due to lower V_{dd} in an efficient manner, and effectively lower power. By deploying the right amount and kind of redundancy, we can strike a balance between overhead incurred in achieving reliability and energy savings realized by permitting lower V_{dd}. One promising approach is the Redundant Residue Number System (RRNS) representation. Unlike other error correcting codes, RRNS has the important property of being closed under addition, subtraction and multiplication, thus enabling computational error correction at a fraction of an overhead compared to conventional approaches. We use the RRNS scheme to design a Computationally-Redundant, Energy-Efficient core, including the microarchitecture, ISA and RRNS centered algorithms. From the simulation results, this RRNS system can reduce the energy-delay-product (EDP) by about 3× for multiplication intensive workloads and by about 2× in general, when compared to a non-error-correcting binary core.

Extension of Conference Paper: This paper is an extension of "Computationally-Redundant Energy-Efficient Processing for Y'all (CREEPY)" [11]. This submission adds the following:

(1) Correction factor analysis for RRNS signed arithmetic, including an improved correction factor computation for signed multiplication via an LUT based mechanism. (Section 4.8.5)
(2) Design and evaluation of an efficient RRNS multiplier unit by using the index-sum technique, along with associated re-derivation of suitable RRNS bases. (Sections 4.4 and 4.5)
(3) A novel adaptive check insertion strategy that leverages hardware/software runtime or compiler. (Section 4.6.3)
(4) Impact of multi-domain voltage supply to further lower energy consumption. (Sections 4.7, 6.1 and 6.3)
(5) Improved evaluation accuracy by simulating an LLC-main memory hierarchy instead of a perfect cache. (Section 5)
(6) Energy limit analysis for binary, RNS and RRNS cores. (Section 6)

These add significantly more than 30% new material and provide greater insight into RRNS core design. *W.r.t.* written content, every section has been revamped to better present the new findings above.

* These authors contributed equally to this work.
** This work is supported by Laboratory-Directed Research and Development (LDRD) Project 180819 from Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525. Approved for public release SAND2018-0257 J.

DOI: 10.1145/nnnnnnn.nnnnnnn

CCS Concepts: •**Computer systems organization** →**Reliability; Processors and memory architectures; Redundancy;**

Additional Key Words and Phrases: Error-tolerant computing, low energy, reliability

1 INTRODUCTION

Dennard scaling [12] has been one of the main phenomena driving efficiency improvements of computers through several decades. The main idea of this law is that transistors consume the same amount of power per unit area as they scale down in size. However, leakage current and threshold voltage limits caused Dennard scaling to end [46] about a decade ago. This essentially negates any performance benefits that Moore's law may provide in the future; power considerations dictate that a higher transistor density results in either a lower clock rate or a reduction in active chip area.

Theis and Solomon [82] suggest that new device concepts within the purview of two-dimensional lithography technology, such as tunneling FETs, enable reduction of the $\frac{1}{2}CV^2$ energy to small multiples of kT, without resulting in low switching speed [81]. Similarly, research on ferroelectric transistors, *aka* negative capacitance FETs (NCFETs) demonstrates a sub-$60mV/dec$ slope as well as a higher drive current [34–36, 65], both of which are necessary in rendering V_{dd} reduction beneficial to energy reduction without sacrificing performance.

These next generation devices are fast switching even at few tens of millivolts, but as a result, are vulnerable to thermal noise perturbations. This translates into *intermittent, stochastic bit errors in logic*. With signal energies approaching the kT noise floor, future architectures will need to treat reliability as a first-class citizen, by employing efficient computational error correction.

In this paper, we propose a scalable architectural technique to effectively extend the benefits of Moore's law. We enable reducing the supply voltage beyond conservative thresholds by efficiently correcting intermittent computational errors that may arise as a result of thermal noise. Energy benefits are observed as long as the overhead incurred in error correction is less than that saved by lowering V_{dd}. The *creepy* approach of introducing error correcting hardware to lower energy is demonstrated as beneficial in this paper.

1.1 Contributions

(1) Development of microarchitecture, ISA and RRNS centered algorithms towards a Computationally-Redundant, Energy-Efficient core design.
(2) Design and analysis of an efficient RRNS multiplier unit using the index-sum technique.
(3) Novel RRNS-check-insertion heuristics to optimize performance/energy/reliability trade-offs.
(4) Derivation of an estimated lower limit on signal energies via stochastic fault injecting simulation.

We first introduce some mathematical background and notation in Section 2 and then provide a high level overview of a CREEPY core in Section 3 before describing the RRNS algorithms and several other aspects towards designing a CREEPY core in Section 4. We then describe our evaluation methodology, results before discussing related work and concluding in Sections 5, 6, 7 and 8 respectively.

2 BACKGROUND

2.1 Triple Modular Redundancy (TMR)

Error-correcting codes (ECC) are widely used in modern processors to improve reliability. However, these are limited to memory/communication systems and are unable to achieve computational

Table 1. A (4, 2)-RRNS example with the simplified base set (3, 5, 2, 7, 11, 13). Range is 210, with 11 and 13 being the redundant bases.

Decimal	mod 3	mod 5	mod 2	mod 7	mod 11	mod 13
13	1	3	1	6	2	0
14	2	4	0	0	3	1
13+14=27	(1+2)mod 3=0	2	1	6	5	1
	All columns function independently of one another.					
	An error in any one of these columns (residues) can be corrected by the remaining columns.					

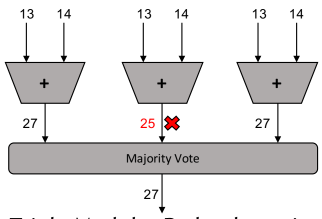


Fig. 1. Triple Modular Redundancy in action.

fault tolerance. The conventional approach to computational fault tolerance is TMR [86]. As shown in Figure 1, the idea is to replicate the computation twice (for a sum total of three computations per computation) and then take a majority vote. With a model that assumes that at most one of these three computations can be in error at any given point in time, it follows that at least two of the computations are error-free; this can thus be used to detect and correct a single error, assuming an error-free voter.

While simple to understand and implement, this introduces more than 200% overhead in area and power leaving plenty of room for improvement. Any energy savings from lowering V_{dd} would be eclipsed due to this overhead in correcting resultant errors.

2.2 Residue Number System (RNS)

The Residue Number System has been used as an alternative to the binary number system chiefly to speed up computation [1, 51]. This increased efficiency comes from the fact that a large integer can be represented using a set of smaller integers, with arithmetic operations permissible on the set in parallel. We present some of the properties of RNS below.

Let $B = \{m_i \in \mathbb{N}\ for\ i = 1, 2, 3, \ldots, n\}$ be a set of n co-prime natural numbers, which we shall refer to as bases or moduli. $M = \prod_{i=1}^{n} m_i$ defines the range of natural numbers that can be injectively represented by RNS that is defined by the set of bases B. Specifically, for x such that $x \in \mathbb{N}\ and\ x < M$, then, $x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, \ldots, |x|_{m_n})$, where $|x|_m = x\ mod\ m$. Each term in this n-tuple is referred to as a residue.

We also note that addition, subtraction and multiplication are closed under RNS. This is because of the following observation: given $x, y \in \mathbb{N}\ and\ x, y < M$, we have $|x\ op\ y|_m = ||x|_m\ op\ |y|_m|_m$, where op is any add/subtract/multiply operation.

2.3 Redundant RNS (RRNS)

To augment RNS with fault tolerance, r redundant bases are introduced. The set of moduli now contains n non-redundant and r redundant moduli: $B = \{m_i \in \mathbb{N}\ for\ i = 1, 2, 3, \ldots, n, n+1, \ldots, n+r\}$. The reason these extra bases are redundant is because any natural number smaller than M $(= \prod_{i=1}^{n} m_i)$ can still be represented uniquely by its n non-redundant residues. Intuitively, the r redundant residues form a sort of *error code* because all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N}$, $x < M$, then,

$x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, \ldots, |x|_{m_n}, |x|_{m_{n+1}}, \ldots, |x|_{m_{n+r}})$ contains n non-redundant residues as well as r redundant residues. For convenience, we further define $M_R = \prod_{i=n+1}^{n+r} m_i$.

Upon applying arithmetic transformations to an RRNS number, any error that occurs in one of the residues is contained within that residue and does not propagate to other residues. When required, such an error can be corrected with the help of the remaining residues. Specifically, an RRNS system with $(n, r) = (4, 2)$, a single errant residue can be corrected, or, two errant residues can be detected. Table 1 provides a simple example, Section 4.8 outlines necessary algorithms to do so. Research by Watson and Hastings [25, 89, 90] lays the foundation for the underlying theoretical framework that is used and extended in our work. Their work also details algorithms to handle RRNS scaling and fractional multiplication. They used (199, 233, 194, 239, 251, 509) as the (4, 2)-RRNS system, providing a range $M = 199 \times 233 \times 194 \times 239 \in (2^{31}, 2^{32})$. In Section 4.5, we discuss the methodology and implications of choosing a different set of RRNS bases for the purposes of trading range with overhead.

Not only does a residue number system achieve a higher efficiency due to enhanced bit-level parallelism (also, no carries required for addition), but also that introducing 50% of overhead is sufficient to provide resiliency. As the granularity of an error is that of an entire residue, RRNS is capable of potentially correcting multi-bit errors as well, for *free*.

We design a computer based on these properties.

3 CREEPY OVERVIEW

Given new device concepts that enable device operation at signal energies close to the kT noise floor [34–36, 65, 81, 82], CREEPY aims to achieve lower energy consumption by lowering V_{dd} in such a manner that the intermittent errors that thereby arise are corrected efficiently.

We take note of the compute preserving properties of RRNS (*cf.* Section 2.3) and propose building a Turing-complete computer around this idea.

A CREEPY core consists of 6 *subcores*, an Instruction Register (IR) and a Residue Interaction Unit (RIU), as depicted in Figure 2.

Each subcore consists of an adder, a multiplier, a portion of the distributed register file and a portion of the distributed data cache. The bit-width of these components is same as that of their corresponding residue (8-bit or 9-bit in this example). Each subcore is fault-isolated from the other because it is designed to operate on a single residue of data(analogous to a bit-slice processor, with bolsters). Post a successful instruction fetch (the instruction cache stores instructions in binary, and is ECC-protected), the ECC-checked instruction is dispatched onto the 6 subcores, which then proceed to operate on their corresponding slice of data. For example, adding two registers is done on a per residue basis; the register file is itself distributed across the 6 subcores. Similarly, the data cache is also distributed across the 6 subcores and stores RRNS protected data. The RIU is then responsible to perform any operations that involve more than a single residue.

Section 4 evaluates several aspects of designing such a core, and provides solutions. For example, conventional multipliers incur high cost in both energy and area, therefore, we leverage RRNS properties to provide an efficient solution in Section 4.4. The RRNS base selection is also very important in CREEPY core design because it directly affects the computational range and energy efficiency, which we discuss in Section 4.5. The RIU logic includes 3 parts: RRNS consistency check logic, RRNS comparison logic and RRNS to binary conversion logic.For the RRNS consistency check logic and RRNS comparison logic designs, we have detailed discussions in Section 4.8.1 and Section 4.8.4 respectively. The RRNS to binary conversion logic is relatively less important as it is used only to support operations that are not native to RRNS, such as bit-shifting and division, which are relatively few in number. Furthermore, the circuitry to convert from RRNS to binary is identical to

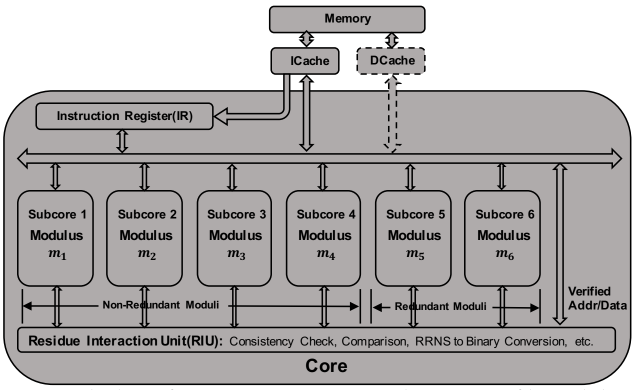


Fig. 2. The CREEPY Core with the reference RRNS system. The register file and data cache are distributed across the subcores.

that found in the literature[7, 25, 76] to convert from RNS to binary, therefore we omit it for space constraints.

Because conversions to and from binary are expensive and rather unnecessary for RRNS data, a CREEPY core operates entirely on RRNS data and literals. An upshot of this is that control-path errors manifest themselves as data errors, meaning that they can be handled simply by handling the data error. For example, if there is an error in bypass logic in a subcore, or, if a faulty decoder in one of the subcores causes it to perform a multiplication instead of an addition, the resultant residue for that subcore would have an erroneous value, but can be recovered from the remaining 5 residues that were a result of the correct addition operation.

Although operating entirely on RRNS data and literals avoids the significant overheads of converting to/from binary, representing a memory address (for the purposes of PC, LD and ST) in an RRNS format naively may cause significant degradation in locality and changes memory access patterns, which is fundamental to memory systems performance. This issue has already been handled by Srikanth et. al. [70], where they propose bit-manipulation techniques as well as a compiler based approach, with little to no overhead. Of the techniques proposed, their rns_sub scheme, which essentially subtracts the least significant residue from the others, renders the most energy efficient architecture along with the added advantage of requiring no support from the software stack.

Interfacing a CREEPY core with heterogeneous accelerators (such as those on an SoC) that may or may not be RRNS based, is a more involved issue, and we leave that to future work.

CREEPY employs standard ECC-protected main memory because of ECC's compactness and efficiency when it comes to protecting stored data. However, standard ECC isn't amenable to computational fault tolerance and therefore, the representation of data is in RNS form (as opposed to binary). The memory controller checks ECC on a processor load and generates the two redundant residues before loading the resultant RRNS data into the last level cache. Similarly, it generates ECC upon a processor store (and the redundant residues are not stored into the main memory). The exact choice of ECC is not relevant to this article; any of the existing schemes [43] may be used.

4 CREEPY CORE

In this section, we present several considerations for the design of the CREEPY core.

4.1 Instruction Set Architecture (ISA)

The description of CREEPY ISA is laid out in a manner similar to that of the MIPS ISA, for explanatory purposes. To simplify instruction fetch and decode, all instructions are of fixed length; 32 bits. The

ISA expects 32 registers (R0-R31), with R0 hard-wired to zero, R30 being the link register and R31 storing the default next PC (= $PC + 4$). In our micro-architecture, each register is 49 bits long (*i.e.,* it contains the RRNS redundant residues as well) and is sliced on a per-modulus (sub-core) basis. The data cache is also implemented in a similar manner, as it stores data in an RRNS format.

(1) **R-Format (ADD/SUB/MUL)**

These instructions assume that the destination operand as well as both source operands are registers.

Opcode	Src Reg1	Src Reg2	Dest Reg	Reserved
6b	5b	5b	5b	11b

(2) **I-Format (ADDI/SUBI/MULI)**

For instructions that require compiler generated immediate literals, two new instructions (that always occur in succession without exception) are defined. Telescopic op-codes are employed to facilitate implementation of such *set* instructions. The fundamental need for the *set* instruction arises from the fact that literals are 49 bit RRNS values and would not otherwise simply fit within a 32 bit field (next to an immediate instruction, for example).

Set123 sets the the first 3 residues of the immediate value into the first 3 sub-core slices of the destination register and *Set456* sets the remaining 3 residues of the immediate value into the other three sub-core slices of the destination register.

Opcode	Dest	Reserved	Residue3	Residue2	Residue1
11[2b]	5b	0[1b]	8b	8b	8b

Opcode	Dest	Residue6	Residue5	Residue4
11[2b]	5b	9b	8b	8b

For an example, consider the immediate instruction *Addi R1, R2, 0x020202020202*. A CREEPY program would implement this instruction as follows:

(a) Set123 R3, 020202
(b) Set456 R3, 020202
(c) Add R1, R3, R2

(3) **Branch**

Opcode	Reg1	Reg2	Reg3	Link	Reserved
6b	5b	5b	5b	1b	10b

Recall that R0 = 0, R31 = PC + 4 and that R30 is the link register. A CREEPY branch follows one of the following semantics:

(a) Reg1 = R0 and Reg3 = R0 and Link = 0: An unconditional branch that always jumps to the address in Reg2.
(b) Link = 0: A conditional branch that jumps to the address in Reg2 (base) + Reg3 (offset) if Reg1 is 0. This is otherwise known as a *beqz* instruction.
(c) Link = 1: A branch and link instruction to enable sub-routine calls and returns. The default next PC is stored into the link register and the program jumps to the address in Reg2.

(4) **Load/Store**

Opcode	Reg1	Reg2	Reg3	Reserved
6b	5b	5b	5b	11b

Reg3 is the destination for a load and is the source register for a store. The source/destination address for a load/store is given by Reg1 (base) + Reg2 (offset). Note that the memory address is hereby stored in an RRNS format. Recall from Section 3 that efficiently handling RRNS addresses without conversion to binary is critical to application performance. Tradeoffs and methodologies in this space have been handled by Srikanth et. al. [70].

(5) **RRNS Check**

Opcode	Reg1	Reserved
6b	5b	21b

Reg1 is the register that needs to be checked. Once an error is detected, the system would try to correct it, for example, by performing the RRNS Single Error Detection and Correction algorithm (Section 4.8). Candidate usage scenarios are discussed in Section 4.6 and evaluated in Section 6. Helper instructions such as *mov*, *ret* etc. also exist, but are omitted from this description for brevity.

4.2 Error Model

First, we distinguish fault, error and failure as follows:

Fault. A single bit flips, but is not stuck-at, *i.e.,* only intermittent / transient faults are considered. Causes may range from unreliable devices to low supply voltage to particle strikes to random noise and any combination therein.

Error. One or more faults in a single residue that show up during a consistency check.

Failure. Error uncorrectable and no recovery mechanism, or error undetectable.

Faults may lead to errors which may lead to failures. *We can guarantee the system is reliable if at most one error per core occurs between two RIU checks.* Multiple bit flips are rare but this phenomenon occurs if a circuit in the carry chain fails [40]. In our design, carry chains are limited to a residue as there are no carries between residues. Therefore, any resulting multi-bit errors would be localized to a single residue, which we can correct. If this RRNS system needs to detect and correct multiple error residues, an extra checkpoint and rollback mechanism is necessary. However, based on the discussion above, the case of multiple residues in error is extremely rare. So we ignore the checkpoint mechanism design in current system and leave it to future work.

Redundancy in time, *i.e.*, check at cycle x, check again at cycle y, check again at cycle z, and vote, does not apply to this model as it is possible that the three checks suffer 3 independent 1 bit faults, rendering voting useless. The *transient* clause in the model rules out stuck-at faults. An implication of this is that we cannot achieve reliability by merely trading performance alone. Additional resources in terms of spatial redundancy are necessary, which is exactly what has been designed.

Different components of the core are protected via specialized means that target each component. The guiding principle is to design a system that uses the more efficient of RRNS/ECC based redundancy based on the range and nature of data being protected. Where both techniques are deemed insufficient to prevent the fault from metastasizing into an error, and eventually into a failure, the more conventional (and expensive) method: Triple Modular Redundancy (TMR), is employed. An alternative is to prevent the fault from occurring in the first place by using high V_{dd} (and/or circuit hardening). Choosing optimally between the latter expensive techniques is beyond the scope of this document but we assume that the RIU uses a high V_{dd} / hardened circuitry. We assume that error in control signals manifest themselves as errors in data (for example, a control error causing one of the subcores to operate on the wrong opcode will be caught as a data error); however, one can potentially further improve the control signals' integrity by using either TMR or intelligent state assignment, and that the RIU uses a high V_{dd} / hardened circuitry.

4.3 Signed Number Representation

There are three competing ways of representing signed numbers, given an RRNS framework as presented in Section 2.3. Each presents its set of trade-offs, which we now detail.

M is the product of all the non-redundant moduli (M = m1*m2*m3*m4) and MR is the product of all the redundant moduli (MR = m5*m6).

(1) **Complement M*MR Signed Representation**
The M*MR complement signed representation is depicted by Figure 3. To provide a few examples, 0 is represented by 0, 1 is represented by 1, $\frac{M}{2}-1$ is represented by $\frac{M}{2}-1$, -1 is represented by $M*MR-1$ and $-\frac{M}{2}$ is represented by $M*MR-\frac{M}{2}$. This is similar to signed binary representation. However, representing numbers in this manner breaks known error correction algorithms[89].

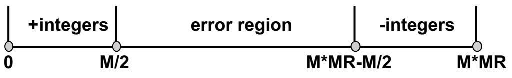


Fig. 3. Complement M*MR signed representation

(2) **Complement M Signed Representation**
The M complement signed representation is depicted in Figure 4. This is similar to the M*MR complement representation, except that the wrap-around occurs at M as opposed to M*MR. This representation does not break error correction algorithms, provided that some correction factors (scaling and offset) are applied to the result of each arithmetic operation. However, further analysis indicates that these correction factors require knowledge of the signs of the operands, which are not trivial to determine like in binary. The RRNS sign determination is a time-consuming algorithm. Moreover. arithmetic operation overflow detection is unknown for this representation.

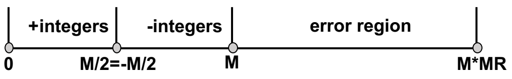


Fig. 4. Complement M signed representation

(3) **Excess-$\frac{M}{2}$ Signed Representation**
The Excess-$\frac{M}{2}$ signed representation is depicted in Figure 5. The excess notation, sometimes known as offset notation, merely shifts each number by $\frac{M}{2}$. To further elaborate, 0 is represented by $\frac{M}{2}$, 1 is represented by $\frac{M}{2}+1$ and -1 is represented by $\frac{M}{2}-1$. Similar to the M Complement representation, the results of arithmetic operations must be offset by a correction factor before they can be corrected. However, these correction factors turn out to be independent of the sign of the operands. We also find that this representation enables simple algorithms for comparison (and thereby sign detection) and arithmetic operation overflow detection. In fact, these algorithms make use of a technique used in the error correction algorithm itself. These algorithms are discussed in detail in Section 4.8.

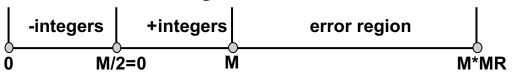


Fig. 5. Excess -M/2 signed representation

We choose Excess-$\frac{M}{2}$ to be the de facto signed representation scheme for CREEPY.

4.4 Optimized Multiplier Unit Design

Many workloads in the domains of multimedia, image processing and digital signal processing are highly multiplication intensive [87] . Index-sum multiplication has been proposed in the past [57, 58] to achieve multiplication via simple addition and table lookup operations, thereby rendering it more efficient than traditional binary multiplication provided the size of the LUT is not too large. The principle is analogous to using a *logarithm* operation, *i.e.,* a multiplication can be achieved via a table lookup, addition and a reverse table lookup, as summarized as follows for the product of two numbers X and Y:

Table 2. Mapping table of $GF(59)$ with a primitive root of 11 ($g = 11$)

X	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
α	0	7	2	14	42	9	10	21	4	49	1	16	25	17	44	28	48	11	34	56
X	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**
α	12	8	47	23	26	32	6	24	22	51	53	35	3	55	52	18	37	41	27	5
X	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**		
α	40	19	57	15	46	54	45	30	20	33	50	39	38	13	43	31	36	29		

Ex: $X = 3,\ Y = 6 \implies \alpha_X = 2, \alpha_Y = 9$, whose sum is 11, which reverse maps to 18 and is indeed the desired product.

(1) Use a pre-defined mapping table to generate $index(X)$ and $index(Y)$.
(2) Compute the sum $Z = index(X) + index(Y)$.
(3) Use a pre-defined reverse mapping table to return the product XY as $reverse\_index(Z)$.

While realizing an LUT that is addressable by a 32-bit input is rather expensive, leveraging RNS properties allows us to slice this table into a few tables with address sizes closer to 8 bits, as outlined by Preethy *et al.* [57, 58] . We extend this idea into RRNS by adjusting the RRNS bases (*cf.* 4.5) to be amenable to index-sum LUTs, the requirements for which, are summarized below.

Index-sum multiplication is based on the theory of Galois fields, which can be classified into 3 types: $GF(p)$, $GF(p^m)$ and $GF(2^m)$, where, p is an odd prime number and $m \in \mathbb{Z}^+$. The range of integers that can be represented bijectively in Galois fields, and the encoding methodology depends on the GF type[58]: (We skip the methodology of deriving $GF(p^m)$ as we don't utilize this for CREEPY.)

$GF(p)$: Any integer $x \in [1, p-1]$ can be uniquely coded as a single integral index code α by the relationship $X = |g^\alpha|_p$, where $\alpha \in [0, p-2]$, and g is a primitive root such that $|g^{p-1}|_p = 1$. See Table 2 for an example.

$GF(2^m)$: Any integer $x \in [1, 2^m - 1]$ can be coded as a triple integral index code $< \alpha, \beta, \gamma >$ by the relationship $X = 2^\alpha |5^\beta (-1)^\gamma|_{2^m}$, where $\alpha \in [0, m-1]$, $\beta \in [0, 2^{m-2} - 1]$ and $\gamma \in [0, 1]$. See Table 3 for an example.

Therefore, the relative preference of GF types are $GF(p) > GF(p^m) > GF(2^m)$ as they require 1, 2 and 3 index codes respectively. Furthermore, a smaller value of p and m leads to a smaller LUT. These considerations impact the choice of RRNS bases, as discussed in Section 4.5 / Table 4.

By using the index-sum technique in conjunction with RRNS, we greatly simplify the complexity of multiplication. Index-sum multiplication can be efficiently performed via a simple addition and two modest table lookup operations. We achieve a reduction in ALU gate count using this approach by about 87% when compared to using a traditional multiplier in RRNS, which itself reduces the gate count by 52% when compared to a traditional non-error-correcting binary ALU, thereby realizing area, energy and reliability improvements, as we demonstrate in Section 6.

4.5 Selecting RRNS Bases

Watson [89] used the base set (199, 233, 194, 239, 251, 509) in his paper. However, the range rendered by this set is larger than 2^{31} but smaller than that of a 32-bit unsigned integer: 2^{32}. Furthermore, these bases are not amenable to designing index-sum based multipliers, as discussed in Section 4.4. These limiting necessary and sufficient conditions can be summarized as follows:

(1) Each pair of bases m_i, m_j must be relatively prime. (For RRNS representation [89].)
(2) $max_{n+1 \leq i \leq n+r} \frac{M_R}{m_i} \geq max_{1 \leq i \leq n} m_i$
(3) $M_R \geq max_{1 \leq i \neq j \leq n} m_i m_j$
(4) $M_R \neq 2m_i m_j - n_1 m_i - n_2 m_j$; $1 \leq i \neq j \leq n; 1 \leq n_1 \leq m_j - 1; 1 \leq n_2 \leq m_i - 1$
(5) $M_R \geq 2 \sum_{i=1}^{n} (m_i - 1) + \sum_{i=n+1}^{n+r} (m_i - 1)$. (For RRNS single error correction [89].)

Table 3. Mapping table of $GF(2^6)$

X	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
α, β, γ	0,0,0	1,0,0	0,3,1	2,0,0	0,1,0	1,3,1	0,10,1	3,0,0	0,6,0	1,1,0	0,5,1	2,3,1
X	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
α, β, γ	0,15,0	1,10,1	0,4,1	4,0,0	0,12,0	1,6,0	0,7,1	2,1,0	0,13,0	1,5,1	0,14,1	3,3,1
X	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
α, β, γ	0,2,0	1,15,0	0,9,1	2,10,1	0,11,0	1,4,1	0,8,1	5,0,0	0,8,0	1,12,0	0,11,1	2,6,0
X	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**
α, β, γ	0,9,0	1,7,1	0,2,1	3,1,0	0,14,0	1,13,0	0,13,1	2,5,1	0,7,0	1,14,1	0,12,1	4,3,1
X	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**
α, β, γ	0,4,0	1,2,0	0,15,1	2,15,0	0,5,0	1,9,1	0,6,1	3,10,1	0,10,0	1,11,0	0,1,1	2,4,1
X	**61**	**62**	**63**									
α, β, γ	0,3,0	1,8,1	0,0,1									

Ex: $X = 3$, $Y = 6$ map to, respectively, $< 0, 3, 1 >$, $< 1, 3, 1 >$, whose sum results in $< 1, 6, 2 >$. Since $\gamma \in [0, 1]$, the modulo sum results in $< 1, 6, 0 >$, which reverse maps to 18 and is the desired product.

Table 4. New base sets that satisfy all conditions listed in Section 4.5. For reference, the range of Watson's base set (8-8-8-8-8-9) is 2,149,852,322, and 2^{32} is 4,294,967,296.

Subcore bits	Total bits	Range	Possible base sets	Bases' format
6-8-7-7-8-8	44	82,600,832	**(61,149,128,71,179,181)**	(p,p,2^7,p,p,p)
7-8-7-8-8-8	46	467,921,792	(97,223,128,169,239,241)	(p,p,2^7,13^2,p,p)
7-8-7-8-8-9	47	729,405,056	(113,239,128,211,251,263)	(p,p,2^7,p,p,p)
8-8-7-8-8-8	47	635,871,872	(151,167,128,197,211,223)	(p,p,2^7,p,p,p)
7-8-8-7-9-9	48	430,002,432	(89,233,256,81,283,293)	(p,p,2^7,3^4,p,p)
8-8-8-8-8-9	49	2,149,852,322	**(199,233,194,239,251,509)**	Watson[89]**
9-6-9-5-10-10	49	251,904,512	(269,59,512,31,521,523)	(p,p,2^9,p,p,p)
9-7-8-8-9-9	50	1,230,080,256	(433,81,256,137,439,443)	(p,3^4,2^7,p,p,p)
9-7-9-7-10-10	52	1,719,885,312	(367,113,512,81,521,523)	(p,p,2^9,3^4,p,p)
9-8-8-9-9-10	53	7,891,035,392	**(421,211,256,347,503,521)**	(p,p,2^8,p,p,p)
9-9-8-9-9-9	53	7,710,332,672	(277,317,256,343,409,421)	(p,p,2^8,7^3,p,p)

** these bases do not satisfy index sum constraint.

(6) $|m_1 m_2 - m_3 m_4| = 1$; also known as the $K, K-1$ property. (For RRNS fractional multiplication [89].)

(7) $m_i \in$ {x | x is either (p) prime, (p^m) a power of prime, or (2^m) a power of 2}. Recall that the relative order of preference is $p > p^m > 2^m$, and that smaller bases result in smaller ROMs. (For index sum multiplication (Section 4.4).)

The proofs of Condition (1)-(6) are available in Waston's thesis[89] and Condition (7) is based on the theory of Galois fields which has been discussed in Section 4.4. We limit our analysis to $(n, r) = (4, 2)$ for simplicity and find bases that satisfy the conditions summarized above, while keeping the overhead to a minimum. Table 4 lists several such possibilities. (61, 149, 128, 71, 179, 181) is the set of bases that offers least overhead, whereas (421, 211, 256, 347, 503, 521) on the other hand, offers a range superior to 2^{32} at additional overhead.

4.6 RRNS Check Insertion Strategies

Given that the CREEPY microarchitecture supports the error model outlined in Section 4.2, it is necessary to carefully insert RRNS_check instructions as they have a direct impact on the performance-energy-reliability metrics of the core. In this section, we outline the following check insertion schemes.

4.6.1 ***Periodic check.*** Insert a single check instruction after every n instructions. When $n = \infty$, this is an unchecked core and when $n = 1$, every instruction is checked. Note that lowering the value of n increases the check insertion frequency, raising performance overhead. While increased check insertion frequency typically provides increased reliability, one must be vary of the fact that the check instruction itself is of non-zero latency (*cf.* Section 4.8), meaning that, the longer the core spends in consistency checking, the longer it leaves its state vulnerable for errors to *creep* in. On the other hand, not checking every instruction also increases the probability of errors manifesting into multiple residues, leading to core failure.

4.6.2 ***Pipelined check.*** Insert a pipelined check that checks n instructions after every n instructions. This approach has the performance advantage of amortizing the latency of RRNS_check via pipelining as well as the reliability advantage of being able to increase state coverage of consistency check.

4.6.3 ***StateTable guided adaptive check.*** We define a bookkeeping entity known as $StateTable$ in Section 5 to maintain temporal information of the vulnerability of processor state. Whenever the probability of a register exceeds a certain threshold, an RRNS_check is inserted for that register. Naturally, this can be extended to insert pipelined checks if more than one register is in need of a check. This $StateTable$ itself is assumed to be an error-free entity that can either be implemented in software or hardware. Like the other two schemes, this insertion scheme can be implemented by the compiler or by the runtime (hardware or software); however, it is likely that utilizing a runtime component for this purpose would yield greater accuracy, which translates to improved efficiency and reliability, although subject to the overhead the $StateTable$ itself introduces.

Irrespective of the check strategy, we acknowledge that the following need to be error-free for correct execution; however, for the purposes of this simulation, we ignore their overheads/implications on control flow by assuming periodic checkpointing for potential rollbacks: (1) Effective address of each memory access (RRNS check), (2) Instruction contents (standard ECC check), and, (3) Main memory contents (standard ECC check). For a low-overhead checkpoint mechanism candidate, one can use an incremental checkpoint scheme to save energy and reduce storage overhead, when compared to using full checkpoints alone. The incremental checkpoints only record the modified entries from the last checkpoint (the last checkpoint could either be a full checkpoint or an incremental checkpoint). Once the rollback operation is necessary, the system can then use the last full checkpoint and the subsequent incremental checkpoints to recovery the machine state. A detailed trade-off analysis of the size, frequency, reliability and energy of such a scheme is beyond the scope of this paper.

4.7 Multi-Domain Voltage Supply

The error distribution for each domain of a CREEPY core, *viz.*, computational logic, SRAM cells and RIU logic, are different. In an SRAM device, any fault occurring in one of its transistors gets latched, thereby resulting in an error. To contrast, glitches in logic transistors get masked if the glitch does not occur close to the clock edge. Also, to avoid having to 'check a check instruction', we assume the RIU logic is error-free protected via TMR, hardened logic, and/or higher signal energies, with the latter sufficient to model the energy effects of the former. Given that the vulnerability of these domains increases from computational logic to SRAM cells to RIU logic, it is inefficient to assume a uniformly high signal energy across these domains. We model this phenomena by independent voltage rails for each of these domains. Shimazaki [68] and Rusu [64] proposed some multi-voltage domain designs. The voltage domains referred to in CREEPY are coarse-grained (module-based), rendering the implementation feasible.

Table 5. Error Correction table of RRNS System with Moduli (3,5,2,7,11,13)

Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ
1 , 10	4 6	4 , 5	1 1	5 , 12	3 1	7 , 7	4 3	9 , 3	2 2
2 , 10	2 3	4 , 6	4 4	6 , 1	3 1	7 , 8	1 2	10 , 3	4 1
2 , 12	4 6	4 , 7	2 1	6 , 4	2 4	8 , 1	2 2		
3 , 3	1 1	4 , 11	4 5	6 , 5	4 3	8 , 4	4 2		
3 , 9	4 5	5 , 8	4 4	7 , 2	4 2	8 , 10	1 2		
3 , 12	2 3	5 , 9	2 1	7 , 6	4 2	9 , 1	4 1		

4.8 RIU Algorithms

The algorithm for single error correction was originally given by Watson [89]. However, RNS renders comparison and arithmetic overflow detection to be a non-trivial exercise. We present algorithms to perform these RIU functions by augmenting the consistency checking algorithm. This way, no extra hardware is warranted beyond that required by the error check.

4.8.1 ***Single Error Detection and Correction Algorithm.*** The single error detection and correction algorithm proposed by Watson [89] is based on an error correction table. The working of this algorithm for a system with 4 non-redundant moduli (m_1, m_2, m_3, m_4) and 2 redundant moduli (m_5, m_6), for any given integer X ($< M = m_1 m_2 m_3 m_4$) is as follows: (a) Use a base-extension algorithm [25, 52, 89] to compute $|X'|_{m_5}$ and $|X'|_{m_6}$, where $|X'|_{m_5}$ and $|X'|_{m_6}$ are the outputs of the parallel RIU base-extension algorithm. The inputs to the RIU base-extension algorithm are the outputs of non-redundant subcores: $|X|_{m_1}$, $|X|_{m_2}$, $|X|_{m_3}$ and $|X|_{m_4}$, where $|X|_m = X \ mod \ m$; the computational output of the subcore with m_i modulus. (b) For $i = 5, 6$: compute $\Delta m_i = |X'|_{m_i} - |X|_{m_i}$. (c) A non-zero difference indicates the presence of an error. This pair of differences indexes into an entry of a pre-computed (fixed) error correction table, which contains the index of the residue that is in error and a correction offset that needs to be added to that residue to correct said error.

The RRNS_check instruction performs this RRNS Single Error Detection and Correction algorithm. For the error detection step, the system would perform (a) and (b) to the get values of Δm_5 and Δm_6. For the error correction step (if necessary), it performs (c). Analysis of the algorithm reveals that the error detection step would take 8 cycles while the correction step takes 2 cycles. Therefore, once the system inserts an RRNS_check instruction, the first step is to execute the 8-cycle error detection procedure. If no error is found, then this RRNS_check instruction is complete and it takes 8 cycles in total. But if an error is detected, then we need 2 more cycles for the RRNS correction operation to complete (resulting in 10 cycles in total).

For ease of presentation, we present such an error correction table for a smaller (toy) set of RRNS base moduli in Table 5. The total entries in such a table is at most $2 \sum_{i=1}^{4}(m_i - 1)$. For the remainder of this section, these set of bases are used for explanatory purposes.

4.8.2 ***Unsigned Number Overflow Detection.*** In the absence of any error or overflow, adding 2 unsigned RRNS numbers results in both Δm_5 and Δm_6 being zero. As has been just explained, presence of an error is handled by the error correction table. In the absence of error, we observe that any overflow manifests itself as a fixed index into the error correction table, with the entry not corresponding to any error. Table 6 provides some examples of this observation. While computation of the deltas is most efficient using a base-extension algorithm, we use Chinese Remainder Theorem(CRT) or the Mixed-Radix Conversion (MRC) method to first convert the RRNS number to binary, before computing deltas. This is solely for explanatory purposes; binary conversion is not actually necessary to detect overflow.

Iterating through all possible combinations of numbers and operations, we observe that the value pair of (Δm_5, Δm_6) is fixed. Moreover, (Δm_5, Δm_6) = (10,11) is not a legitimate address of the

Table 6. Unsigned Number Overflow Examples in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	CRT/MRC	$\lvert X'\rvert m_5,\lvert X'\rvert m_6$	Δm_5, Δm_6
2+209	(2,2,0,2,2,2)	(2,4,1,6,0,1)	(1,1,1,1,2,3)	(1, 1, 1, 1) $\Leftrightarrow$ 1	$\lvert 1\rvert_{11}=1,\lvert 1\rvert_{13}=1$	10 11
3+209	(0,3,1,3,3,3)	(2,4,1,6,0,1)	(2,2,0,2,3,4)	(2, 2, 0, 2) $\Leftrightarrow$ 2	$\lvert 2\rvert_{11}=2,\lvert 2\rvert_{13}=2$	10 11
...	...	...	...	...	...	10 11
209+209	(2,4,1,6,0,1)	(2,4,1,6,0,1)	(1,3,0,5,0,2)	(1, 3, 0, 5) $\Leftrightarrow$ 208	$\lvert 208\rvert_{11}=10,\lvert 208\rvert_{13}=0$	10 11

Table 7. Excess-$\frac{M}{2}$ Overflow Examples for addition of two positive numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\lvert X'\rvert m_5,\lvert X'\rvert m_6$	$\Delta m_5,\Delta m_6$
1+104	(1,1,0,1,7,2)	(2,4,1,6,0,1)	(0,0,1,0,7,3)	(0,0,0,0,1,2)	(0, 0, 0, 0) $\Leftrightarrow$ 0	$\lvert 0\rvert_{11}=0,\lvert 0\rvert_{13}=0$	10 11
2+104	(2,2,1,2,8,3)	(2,4,1,6,0,1)	(1,1,0,1,8,4)	(1,1,1,1,2,3)	(1, 1, 1, 1) $\Leftrightarrow$ 1	$\lvert 1\rvert_{11}=1,\lvert 1\rvert_{13}=1$	10 11
...	...	...	...	...	...	...	10 11

Table 8. Excess-$\frac{M}{2}$ Overflow Examples for addition of two negative numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\lvert X'\rvert m_5,\lvert X'\rvert m_6$	$\Delta m_5,\Delta m_6$
-1-105	(2,4,0,6,5,0)	(0,0,0,0,0,0)	(2,4,0,6,5,0)	(2,4,1,6,10,12)	(2, 4, 1, 6) $\Leftrightarrow$ 209	$\lvert 209\rvert_{11}=0,\lvert 209\rvert_{13}=1$	1 2
-3-104	(0,2,0,4,3,11)	(1,1,1,1,1,1)	(1,3,1,5,4,12)	(1,3,0,5,9,11)	(1, 3, 0, 5) $\Leftrightarrow$ 208	$\lvert 208\rvert_{11}=10,\lvert 208\rvert_{13}=0$	1 2
...	...	...	...	...	...	...	1 2

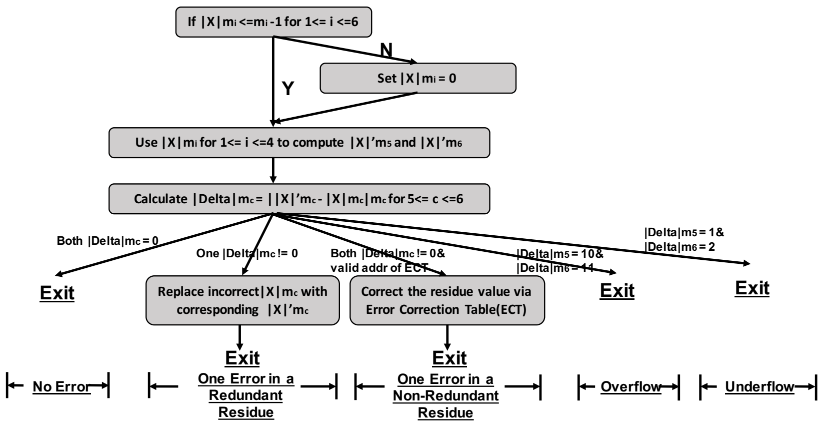

If |X|mi <=mi -1 for 1<= i <=6
N
Y
Set |X|mi = 0
Use |X|mi for 1<= i <=4 to compute |X|'m5 and |X|'m6
Calculate |Delta|mc = ||X|'mc - |X|mc|mc for 5<= c <=6
Both |Delta|mc = 0
One |Delta|mc != 0
Both |Delta|mc != 0& valid addr of ECT
|Delta|m5 = 10& |Delta|m6 = 11
|Delta|m5 = 1& |Delta|m6 = 2
Exit
Replace incorrect |X|mc with corresponding |X|'mc
Correct the residue value via Error Correction Table(ECT)
Exit
Exit
Exit
Exit
No Error
One Error in a Redundant Residue
One Error in a Non-Redundant Residue
Overflow
Underflow

Fig. 6. Single error detection and correction algorithm with overflow/underflow detection

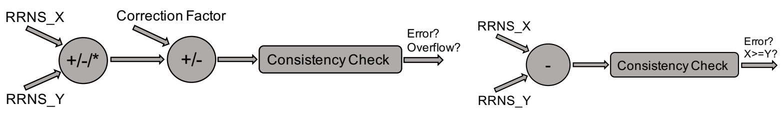
error correction table (Table 5), thus enabling a distinction between an error and an overflow. This approach, however, does not apply to multiplication.

4.8.3 ***Signed Number Overflow Detection.*** Recall from Section 4.3 that CREEPY uses the Excess-$\frac{M}{2}$ signed representation. We discuss the two sources of overflow independently:

(1) *Add two positive numbers.* Table 7 provides a few examples illustrating the algorithm (Correction factors are explained in detail in Section 4.8.5). The $1 + 104$ in the first column is represented in decimal. After Excess-$\frac{M}{2}$ mapping, the computing equation is transformed to $106 + 209$ since $\frac{M}{2} = 105$ for the toy set of moduli. Therefore, the X RRNS value is the the RRNS of 106 and Y RRNS value is the the RRNS of 209. We observe that the pair (Δm_5,Δm_6) remains at a fixed value (10,11).
(2) *Add two negative numbers.* Similarly, examples for adding two negative numbers are shown in Table 8. In this case, we observe that the pair (Δm_5,Δm_6) is fixed to (1,2).

Note that neither (10, 11) nor (1, 2) are legitimate addresses in Table 5, thereby enabling a distinction between an error and an overflow. However, while this method works for both addition and subtraction, it does not hold for detection of multiplication overflow as the delta-pair is not constant and sometimes indexes into a legal error correction table entry.

Figure 6 shows the overview of the whole algorithm.



(a) Signed Overflow Detection (b) Signed Comparison

Fig. 7. Signed Overflow Detection and Comparison

We observe that the described algorithm works in a similar manner even with the base sets in Table 4. E.g. Waston's bases (199,233,194,239,251,509), an overflow results in a delta-pair of (77, 289), whereas an underflow results in (174, 220). Both these pairs do not index into legitimate entries of the error correction table for these set of bases (*cf.* Appendix E, Watson [89]).

4.8.4 ***Comparison***. Comparison is an important operation because of its use in determining control flow. In a manner similar to overflow detection, we explore potential algorithms to perform RRNS comparison without incurring unnecessary hardware overhead.

Jen-shiun et al. [9] and Omondi [52] proposed number comparison methods for residue numbers based on parity bits. However, a prerequisite of these parity comparison methods is that all moduli are supposed to be odd (in addition to being pair-wise relatively prime). In CREEPY, one of the non-redundant moduli is even (to enable fast fractional multiplication [89]), therefore this approach is not suitable.

Instead, we propose leveraging the error check algorithm itself to check for an overflow post a subtraction: To compare X and Y, perform $X - Y$ and derive the delta-pair (Δm_5, Δm_6). Then, $X \geq Y$ iff the delta-pair is (0, 0) (*i.e.,* no overflow) and $X < Y$ iff the delta-pair is (174, 220) (*i.e., $X - Y$* results in an underflow).

This new residue number comparison method can be used for both unsigned and Excess-$\frac{M}{2}$ signed numbers. It is easy to understand that this idea is suitable for unsigned residue numbers: if $X < Y$, then $X - Y \notin [0, M)$, thereby resulting in an underflow. For an Excess-$\frac{M}{2}$ signed number X, an injective mapped residue number can be defined as follows: $X_{mapped} = \frac{M}{2} + X$. Therefore, $X \geq Y$ iff $X_{mapped} \geq Y_{mapped}$, which reduces to an unsigned comparison. A caveat to note is that correction factors should not be added for a comparison operation. These are summarized in Figures 7a and 7b.

4.8.5 ***Correction Factors***. In this section, we are concerned with the addition, subtraction and multiplication operations on two numbers that do not generate any overflow. Recall from Section 4.3 that CREEPY uses the Excess-$\frac{M}{2}$ notation, which means that there is a bijective mapping from any number x such that $-\frac{M}{2} < x < \frac{M}{2}$ to $x + \frac{M}{2}$. Because of this offset, arithmetic operations results need to be re-adjusted using what we term as *correction factors*. [However, this has nothing to do with the RRNS error correction operation.]

Addition Consider the addition of two numbers x and y. To represent the mapping, define a and b such that $0 \leq a, b < \frac{M}{2}$ so that there is no overflow.

Case 1: $x, y \geq 0$

Consider $x = a$ and $y = b$. The sum $x + y$ can be represented for each subcore $1 \leq i \leq n + r$ as follows:

$$\left| |\frac{M}{2} + a|_{m_i} + |\frac{M}{2} + b|_{m_i} \right|_{m_i} = \left| |M|_{m_i} + |a + b|_{m_i} \right|_{m_i} \tag{1a}$$

$$= |a + b|_{m_i} \; for \; 1 \leq i \leq n \tag{1b}$$

However, the expected addition result is:

$$\left| \frac{M}{2} + a + b \right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |a + b|_{m_i} \right|_{m_i} \tag{2}$$

It follows that:

(1) $1 \le i \le n$ and m_i is odd: Examining equations 1b and 2 imply that no correction factor is necessary.
(2) $1 \le i \le n$ and m_i is even: Examining equations 1b and 2 implies that a constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.
(3) $n+1 \le i \le n+r$: Examining equations 1a and 2 imply that a constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be subtracted from the result.

Case 2: $x, y < 0$
Setting $x = -a$ and $y = -b$, and re-working equations similar to Equations 1a, 1b and 2 result in correction factors that are identical to Case 1.

Case 3: $x > 0$, $y < 0$ (Without loss of generality.)
Setting $x = a$ and $y = -b$, and re-working equations similar to Equations 1a, 1b and 2 result in correction factors that are identical to Case 1.

Subtraction Due to the symmetric and offset based nature of the Excess-$\frac{M}{2}$ representation, we again present the working of just one of the cases; without loss of generality: $x = a$ and $y = b$. Then, $x - y$ becomes:

$$\left| |\frac{M}{2} + a|_{m_i} - |\frac{M}{2} + b|_{m_i} \right|_{m_i} = |a-b|_{m_i} \tag{3}$$

However, the expected subtraction result is:

$$\left|\frac{M}{2} + a - b\right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |a-b|_{m_i} \right|_{m_i} \tag{4}$$

From examining equations 3 and 4, it follows that:
(1) $1 \le i \le n$ and m_i is odd: No correction factor is necessary.
(2) $1 \le i \le n$ and m_i is even: A constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.
(3) $n+1 \le i \le n+r$: A constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.

Multiplication Again, for brevity, we only present the case where two positive integers are multiplied; without loss of generality: $x = a$ and $y = b$; the product xy becomes:

$$\left| |\frac{M}{2} + a|_{m_i} |\frac{M}{2} + b|_{m_i} \right|_{m_i} = \left| |\frac{M^2}{4} + \frac{(a+b)M}{2}|_{m_i} + |ab|_{m_i} \right|_{m_i} \tag{5}$$

However, the expected multiplication result is:

$$\left|\frac{M}{2} + ab\right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |ab|_{m_i} \right|_{m_i} \tag{6}$$

As residues are typically 8-bit wide, consider a 511 entry LUT per subcore that stores the following:

$$LUT(s) = \left|\frac{M^2}{4} + \frac{(s-1)(M)}{2}\right|_{m_i} \tag{7}$$

From examining equations 5, 6 and 7, it follows that:
(1) $1 \le i \le n$ and m_i is odd: No correction factor is necessary.
(2) $1 \le i \le n$ and m_i is even: The correction factor can be effected by computing $s = a + b$ and then subtracting $LUT(s)$ from the result of the multiplier.
(3) $n+1 \le i \le n+r$: The correction factor can be effected by computing $s = a + b$ and then subtracting $LUT(s)$ from the result of the multiplier.

The correction factors for the addition and subtraction operations require a single, constant addition/subtraction operation, whereas for multiplication, 2 additions/subtractions and a modest table lookup are required. Another advantage of the schemes presented here is that sign determination is not necessary and that they can be performed at the subcore level, without the involvement of the RIU.

5 EVALUATION METHODOLOGY

To measure the performance-energy-reliability trade-off of a CREEPY core, we augment a stochastic fault injection mechanism into a cycle-accurate in-order trace-based simulator. We abstract the notion of using next-generation devices operating at low signal energies (E_s) and the resulting interaction with the kT noise floor into P_e, the probability of an error occurring in a transistor state in any given cycle. E_s, provided as an input to the simulation, is a measure of the signal energy at the input of a transistor; P_e is the probability of a fault occurring at the output of a transistor in any given cycle. The relationship of E_s and P_e can be defined by the following relation: $P_e = exp(\frac{-E_s}{kT})$. From Section 4.7, these inputs are vectors as they denote the signal energies and error probabilities for each voltage domain, however, for explanatory purposes, we present them as scalars for the remainder of this section. Also input to the simulator is the check insertion strategy, as discussed in Section 4.6. Because we are evaluating a very different number system, we simulated an unpipelined microarchitecture with no branch prediction and a 2-level memory hierarchy (LLC-DRAM, with latencies of 12 cycles and 100 cycles for LLC hit and miss respectively) to maintain our primary focus in this paper. Adding more features to our design has been left as future work.

We first introduce a series of error events and their probabilities.

P_e Probability of an error occurring in a transistor state in any given cycle. This is provided as an input to the simulation, as just discussed.

P_{add} Probability of at least a single error in an adder (each sub-core has an adder). If there are N_{add} transistors in an adder, the probability of each of these transistors being free of error is $(1 - P_e)^{N_{add}}$. Therefore, P_{add}=1-$(1 - P_e)^{N_{add}}$. Similarly, P_{sub} and P_{mul} are calculated. For multi-cycle operations, this definition holds as long as the state of each transistor is used exactly once for the operation. This is true for the said operators. Note that this is a conservative (pessimistic) estimate in our evaluation because we ignore any error masking that may potentially occur.

P_{R_i} Probability of at least 1 error being present in a slice (sub-core/residue) of register R_i since its last write. To compute this, we devise a $StateTable$, the i^{th} entry of which holds the tuple (P, $cycle$), where, P is the probability of R_i having atleast 1 error being present in the corresponding residue upon its most recent update at cycle $cycle$. This $StateTable$ is updated for each register write.

For example, consider the register R_0. 1) At cycle 0, the default value of R_0 tuple is (P=0, cycle=0). 2) At cycle 10, assume that we have an ADD instruction: ADD R0, R1, R2, and that it is the first instruction writing to R_0. We then update the tuple value to (Error_Probability_ADD, 10). It is necessary to update the P value here because the error probability of this ADD instruction should be taken into account. P value would then be set back to 0 once an RRNS check is inserted for that register and no error is detected, and then set the current system cycle value to the cycle field. This way, the P field in the $StateTable$ always reflects the probability of that register of having at least 1 error being present in one of its residues, given its most recent update at the cycle field.

Assuming an SRAM implementation of 8-bit wide R_i, the number of transistors is $8 \times 6 = 48$. The probability of R_i being error free is subject to two probabilities: (1) probability of an error-free write, ($P_1 = 1 - StateTable[R_i].P$) and, (2) probability of no error creeping into it since its last write ($P_2 = (1 - P'_e)^{48(c - StateTable[R_i].cycle)}$), where, c is the current cycle and P'_e is the probability of an error occurring in the state of an SRAM transistor. Due to the nature of an SRAM device, any fault occurring in one of its transistors gets latched, resulting in a higher probability of an error (when compared with glitches in logic transistors getting masked if the glitch does not occur close to the clock edge). As such, we assume $P'_e = 100P_e$. Putting it all together, we have $P_{R_i} = 1 - P_1 * P_2$.

$P_{LOAD\ X}$ Probability of at least 1 error being present in the loaded data of address X. This is analogous to P_{R_i}, with the extended $StateTable$ storing an entry for each cache line. As we assume a perfect off-chip (ECC protected) main memory, cache miss repairs are initialized with a zero probability in error, and cache replacement victims' entries are evicted from the $StateTable$. Finally, $P_{LOAD\ X}$ encapsulates the probability of an error in the implicit computation of the address X itself (from its base and offset) during the execution of the load, in addition to the probability of an error in the loaded data from the cache line.

P_{SC} Probability of at least 1 error occurring in a sub-core from the last time it was checked. To illustrate, consider the following add instruction: $ADD\ R_3, R_2, R_1$. Then, at the end of instruction, $P_{SC} = 1 - (1 - P_{add})(1 - P_{R_2})(1 - P_{R1})$.

P_C Probability of exactly 1 error occurring in a CREEPY core from the last time it was checked. This translates to exactly 1 sub-core being in error (where the sub-core error itself may be of multi-bit form; RRNS can tolerate multi-bit flips within a single residue). Therefore, $P_C = 6C_1 \times P_{SC}(1 - P_{SC})^5$, where the combinatorial choose operator nC_r enumerates the number of ways in which r items can be chosen from n distinct items.

P_C^0 Probability of no error in a CREEPY core from the last time it was checked. $P_C^0 = 6C_0 \times (1 - P_{SC})^6 = (1 - P_{SC})^6$.

P_C^{fail} Probability of a CREEPY core failing at any given cycle, since the last time it was checked. The current version of the CREEPY micro-architecture is unable to correct more than 1 error occurring in the core, and assumes a recovery mechanism such as checkpointing is in place. As such, we deem ≥ 2 errors in the core as amounting to a failure. Therefore, $P_C^{fail} = \sum_{2 \leq r \leq 6} 6C_r \times P_{SC}^r (1 - P_{SC})^{6-r} = 1 - P_C^0 - P_C$.

Note that the computation of these error probabilities is done after every instruction (irrespective of the check insertion strategy) for the purposes of bookkeeping such as $StateTable$ update and to estimate the probability of a failure $P_{C,i}^{fail}$ at each time step t_i. We use a typically used reliability metric, Mean Time Between Failure (MTBF) [77], which can be defined as follows: $MTBF = \frac{Total\ Cycles}{CPU\ Frequency \times \sum_i P_{C,i}^{fail}}$. The subscript i in $P_{C,i}^{fail}$ represents the i^{th} instruction of the instruction stream. MTBF also corresponds to mean time to checkpoint recovery.

6 SIMULATION RESULTS

6.1 Signal Energy Limits

From an independent set of simulations of a non-error-correcting core operating on binary data, we find that the minimal signal energy required for ensuring its reliable operation is $48kT$. In our previous design of an error correcting RRNS core [11], we assumed a single voltage domain across computational logic, SRAM cells and RIU logic. Together with a traditional multiplier (i.e., without index-sum), a pipelined check insertion strategy (with a frequency of 5 instructions) and a 16MB LLC, the result is that we can tolerate gate signal energies of $42 - 43kT$, as shown in Figure 8.

However, given the dissimilarity in error distribution across computation, SRAM and RIU (Section 4.7), we consider independent voltage domains for these. For simplicity, we conservatively set the RIU gate signal energy to be $48kT$ (i.e., same as that required for a non-error correcting binary core), although it can be potentially lowered as its functionality is a subset of that of a binary core. We find that the relative impact of energy savings in the RIU is rather limited (Section 6.5), and therefore restrict the RIU gate signal energy to $48kT$ in our evaluations.

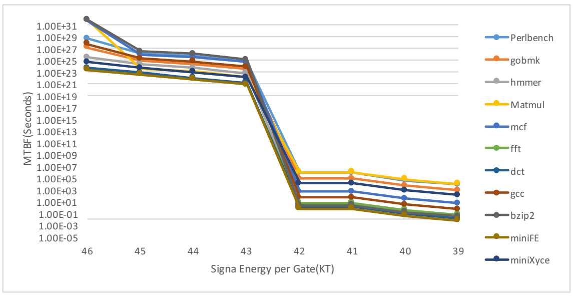


Fig. 8. Reliability when a single voltage domain is used.

Table 9. Sensitivity of MTBF (seconds) to benchmarks and gate signal energies of computational logic, SRAM cells and RIU logic. For example, 30-43-48 denotes the gate signal energy for computational logic to be $30kT$, SRAM cells to be $43kT$, and RIU logic to be $48kT$.

Benchmarks	36-43-48	35-43-48	34-43-48	33-43-48	32-43-48	31-43-48	30-43-48	29-43-48	28-43-48	27-43-48
perlbench	1.70E+17	2.29E+16	3.10E+15	4.20E+14	5.69E+13	7.70E+12	1.04E+12	1.41E+11	**1.91E+10***	2.62E+09
gobmk	1.07E+17	1.44E+16	1.95E+15	2.64E+14	3.58E+13	4.85E+12	6.55E+11	8.87E+10	**1.22E+10***	1.97E+09
hmmer	4.08E+16	5.53E+15	7.48E+14	1.01E+14	1.37E+13	1.85E+12	2.51E+11	**3.40E+10***	6.23E+09	9.69E+08
matmul	1.08E+16	1.47E+15	1.99E+14	2.69E+13	3.64E+12	4.93E+11	6.66E+10	**9.02E+09***	1.22E+09	2.76E+08
mcf	1.81E+17	2.44E+16	3.31E+15	4.47E+14	6.06E+13	8.20E+12	1.11E+12	1.50E+11	**2.03E+10***	2.80E+09
fft	7.63E+14	1.03E+14	1.40E+13	1.89E+12	2.56E+11	**3.47E+10***	4.69E+09	6.35E+08	1.86E+08	2.32E+07
dct	1.33E+15	1.80E+14	2.44E+13	3.30E+12	4.47E+11	6.05E+10	**8.18E+09***	1.11E+09	3.11E+08	4.02E+07
gcc	1.41E+17	1.91E+16	2.59E+15	3.50E+14	4.75E+13	6.42E+12	8.68E+11	1.18E+11	**1.60E+10***	2.37E+09
bzip2	1.73E+17	2.34E+16	3.17E+15	4.29E+14	5.81E+13	7.86E+12	1.06E+12	1.44E+11	**1.95E+10***	2.65E+09
miniFE	6.94E+14	9.39E+13	1.27E+13	1.72E+12	2.33E+11	**3.15E+10***	4.26E+09	5.77E+08	1.69E+08	2.11E+07
miniXyce	1.09E+16	1.47E+15	1.99E+14	2.69E+13	3.64E+12	4.93E+11	6.66E+10	**9.02E+09***	2.04E+09	3.06E+08

* We use these signal energies for the remainder of this paper, as they render reasonable reliability.

We abstract these voltage domains as a triplet; for example, $30 - 43 - 48$ denotes the gate signal energy for computational logic to be $30kT$, SRAM cells to be $43kT$, and RIU logic to be $48kT$. For the purposes of this evaluation, we assume a target MTBF of $1E + 10$ seconds (over 300 years) and find that the gate signal energy for computational logic can be lowered all the way to $28 - 31kT$, depending upon the benchmark, as shown in Table 9.

Given these minimum signal energies, we evaluate the performance, efficiency and reliability of various core configurations in Sections 6.2, 6.3 and 6.4 respectively. The core configurations presented are as follows:

Binary A non-error-correcting core operating on binary data. This is the baseline and requires signal energies of at least $48kT$ in order to achieve reasonable reliability.

RNS A non-error-correcting core operating on RNS data. In other words, an RRNS core without redundant subcores and error correction capabilities.

RRNS_pipe5 An error correcting RRNS core with a pipelined check insertion strategy (with a frequency of 5 instructions), as was determined as the most optimal strategy in our previous RRNS core design [11].

Index-sum_pipe5 Similar to RRNS_pipe5, except that the traditional multiplier is replaced with an index-sum multiplier.

RRNS_Adapt_1e-9 An error correcting RRNS core with an adaptive check insertion strategy (with an error probability threshold of $1e-9$. We found that $1e-9$ was the optimal threshold obtained via simulation for target MTBF/signal energy).

Index-sum_Adapt_1e-9 Similar to RRNS_Adapt_1e-9 that uses an index-sum multiplier.

6.2 Performance

Figure 9 presents the performance of various core configurations listed in Section 6.1, normalized to that of a non-error-correcting binary core.

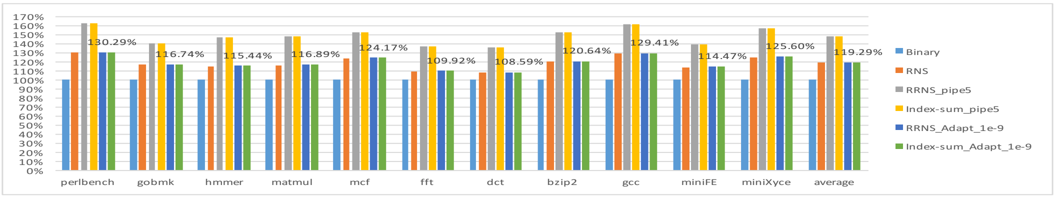


Fig. 9. Performance of various core configurations, normalized to an non-error-correcting binary core

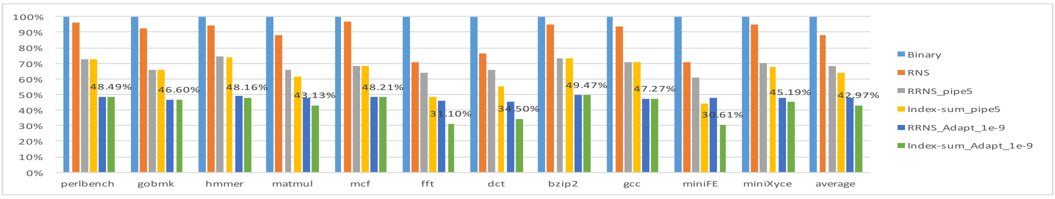


Fig. 10. Energy Comparison for Different Strategies

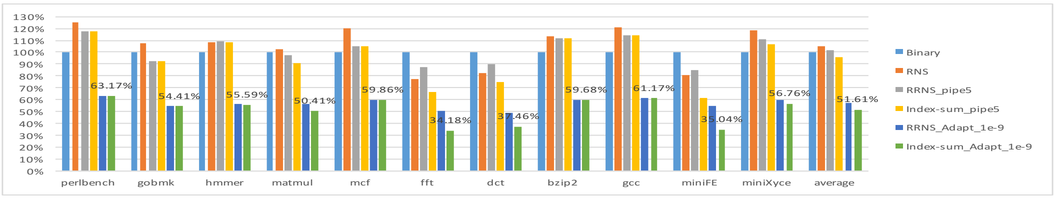


Fig. 11. EDP Comparison for Different Strategies

There is an inherent performance degradation in running binary-optimized code on an (R)RNS-based core because position-based bit manipulation techniques are expensive in (R)RNS, however, this is limited to about 20% on average. Introducing error correction may further degrade performance if naive or static check insertion strategies are used. The overhead due to error correction is amortized when the check insertion strategy is adaptive instead.

6.3 Energy

The primary concern of CREEPY core design is reducing the core energy overhead. Figure 10 shows the normalized energy consumption of the aforementioned configurations.

The non-error-correcting binary core requires high gate signal energies in order to be reliable. Given the low-bit-width and carry-free nature of RNS arithmetic, RNS based cores are inherently more energy efficient than their binary counterparts. When *efficient* error correction is introduced, further energy savings can be achieved as the supply voltage can be turned down while still maintaining reliable functionality. We ensure that the overhead of error correction is minimal by using an adaptive check insertion strategy. Finally, using index-sum multipliers enables further energy savings as they are more efficient than traditional multipliers (savings of over 3× for multiplication intensive benchmarks and over 2.3× on average).

6.4 Energy Delay Product(EDP)

Figure 11 shows the Energy Delay Product (EDP) of these core configurations, normalized to that of a non-error-correcting binary core. With the exception of arithmetic intensive workloads, RNS cores typically have a higher EDP than binary cores. However, via efficient error correction, our RRNS cores show significantly improved EDP. Specifically, by utilizing our best optimization scheme (index-sum multiplier and adaptive check insertion), we see EDP benefits of about 2× on average, or about 3× for multiplication intensive workloads.

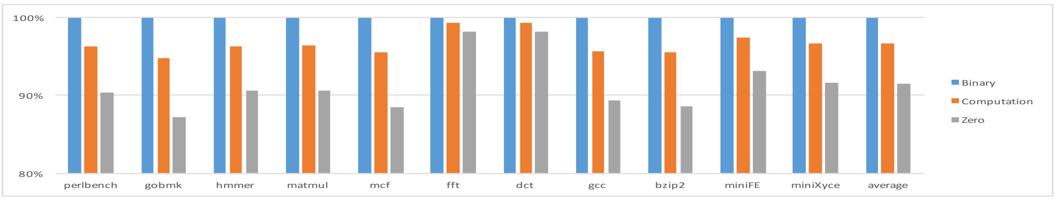


Fig. 12. The Potential of RIU Energy Optimization

6.5 Energy Potential of RIU Optimizations

As described in Sections 4.7 and 6.1, we conservatively choose the gate signal energy for RIU logic to be that necessary for reliable operation of a Turing complete non-error-correcting binary core, i.e., $48kT$. One of the reasons for this is to side-step the issue of 'checking the checker'. However, if we were to deploy self-checking logic or some other optimizations in the RIU, it may no longer be necessary to use a high voltage supply for the RIU domain. In this limits study, we evaluate 3 possibilities of the gate signal energy to RIU logic: *Binary* - $48kT$, *Computation* - same as that of RRNS subcore computational logic, *Zero* - $0kT$. From an Amdahl's law perspective, we find that optimizing RIU logic has limited impact on core energy, as shown in Figure 12, thanks to our judicious RIU usage via adaptive check insertion.

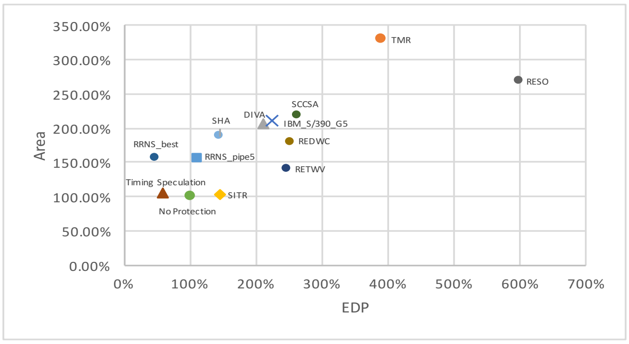
7 RELATED WORK

RNS and RRNS The energy efficient properties of RNS due to its low-bit-width operations and absence of carries across residues has found applications in the digital signal processing (DSP) [10, 14, 60] domain. Furthermore, the representability of high bit-width integers as a tuple-of-resides has been leveraged by the cryptography (RSA) [4, 28, 94] community. Anderson [1] proposed an architecture and ISA for an RNS co-processor designed to run datapath operations in tandem with a general-purpose processor running binary instructions, where the primary role of the general purpose processor is to handle control flow. The RNS co-processor uses an accumulator based ALU and does not support caching or computational error correction (RRNS). Furthermore, it requires a conversion to binary (and vice-versa) for comparison operations, which is expensive. Clearly, our CREEPY architecture is significantly more efficient. A unique feature of their ISA is their ability to encode instructions targeting two ALUs simultaneously. But this can easily be extended to our architecture and enable such Superscalar-like capabilities if need be.

Chiang et al. [9] provide RNS algorithms for comparison and overflow detection, but assume all bases to be odd and do not consider error correction. Similarly, Preethy et al. [57, 58] integrate index-sum multiplication into RNS, but do not consider its impact on the properties of RRNS bases critical to CREEPY.

Ever since Watson and Hastings [25, 89, 90] introduced RRNS as an efficient means for computational error correction, there has been a significant body of research [3, 5, 8, 13, 17, 21, 22, 24, 32, 38, 39, 42, 53, 59, 61, 67, 71–73, 75, 76, 78–80, 91–93, 95] that strives to improve upon it. These are orthogonal to CREEPY, and further such algorithmic research can be used to optimize aspects of the core itself, such as the RIU.

Computational Error Correction Standard error correcting codes (ECC)[43] have already been adopted into modern memory systems. These codes accommodate errors occurring in storage and communication/network traffic, but are not able to protect computational logic. The naive approach to computational error correction is triple modular redundancy (TMR)[86], requiring over a 200% overhead in area and energy for single error correcting capability. Several techniques in the form of arithmetic codes such as AN codes[6, 18, 19, 41, 66, 88], self-checking[30, 33, 44, 48–50, 84] and self-correcting[15, 20, 26, 37, 45, 55, 62, 63, 74, 83] adders and multipliers have since been devised.



RESO[54], REDWC[31], RETWV[27], SCCSA[85], SHA[56], DIVA[2], SITR[47], Timing Speculation[16, 23]

Fig. 13. First order comparison of area overhead and energy-delay product (EDP) of various mechanisms for computational error correction, depicting the superiority of RRNS. Computational error correction techniques use a combination of spatial and temporal redundancy techniques. While temporal redundancy allows for a low area overhead, they suffer from a significant performance penalty. Timing speculation techniques seem more efficient than RRNS, however, their error model assumes all bit errors manifest as circuit timing errors, which is not sufficient to work with ultra low energy logic devices.

Orthogonally, proposals employ redundancy at a higher granularity, such as timing speculation (wherein error correction capability is limited to circuit timing violations)[16, 23], partial pipeline replication[2] or checkpoint-rollback-recovery such as those in IBM Power8 processors[29]. While these are more efficient than naive TMR, they come with limitations on their error model, or, their area overheads are still over 100% and/or incur a significant performance penalty, owing to the fact that they leverage temporal redundancy in an effort to minimize area overhead[69]

Figure 13 summarizes some of these techniques in comparison with RRNS. We refer the interested reader to Srikanth et al.[69] for a more detailed survey on some of these non-residue techniques, but the takeaway is that RRNS is generally considered superior in terms of capability and efficiency for computational error resilience.

Approaches that employ timing speculation[16, 23] may seem superior to RRNS at first glance. However, the error model that can be supported by an RRNS error correcting microarchitecture is orthogonal to theirs, if not broader. For example, razor[16] uses conventional transistors, therefore lowering V_{dd} lowers MOSFET switching speed, resulting in a frequency drop, which could cause setup time violations that they handle via a delayed latch mechanism. They assume that any error manifests itself as a timing error. Similarly, decor[23] uses a delayed commit approach (with rollback support) to handle violations in timing margins. However, with emerging devices (Section 1), V_{dd} can be lowered to few tens of millivolts without frequency loss, meaning that operating at the resultant thermal noise floor leads to *stochastic, intermittent* bit flips, which cannot be captured as circuit timing errors. Unlike such approaches, a CREEPY core can not only tolerate such errors in the data path, but also in the control path between memory accesses.

In terms of being able to tolerate control path errors, approaches such as DIVA[2] that replicate parts of the pipeline are capable. Their design provides recovery by having a simple core recalculate results of an out-of-order core. In this approach, the simple core is assumed to be error-free. This is similar to a ”double-modular-redundancy” approach with a rad-hard node, implying a relatively high overhead. Furthermore, if the rad-hard simple core is instead prone to error, checkpoint and re-execute methods would need to be employed, similar to the IBM POWER7/8 processors[29]. On the other hand, a CREEPY core is able to tolerate errors in its redundant as well as non-redundant computations.

8 CONCLUSION

The advent of next generation device concepts such as tunneling FETs and ferroelectric/negative-capacitance FETs enables reduction of supply voltage to few tens of millivolts without degradation in switching speed. However, as a result of operating close the the kT noise floor, computational logic is subject to intermittent, stochastic errors. The RRNS representation is a promising approach towards using such ultra low power devices, by employing efficient computational error correction.

In this paper, we design a Compuationally-Redundant, Energy-Efficient core, including the microarchitecture, ISA and RRNS centered algorithms. We elucidate several novel optimizations and RRNS-based design considerations to demonstrate significant improvements over a non-error-correcting binary core.

9 ACKNOWLEDGMENT

We sincerely thank the anonymous reviewers of the TACO review process for their constructive feedback on this research.

REFERENCES

[1] Daniel Anderson. 2014. Design and Implementation of an Instruction Set Architecture and an Instruction Execution Unit for the REZ9 Coprocessor System. *M.S. Thesis, U of Nevada LV* (2014).

[2] Todd M Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, MICRO-32. Proceedings. 32nd Annual International Symposium on.* IEEE, 196–207.

[3] Jean-Claude Bajard, Julien Eynard, and Nabil Merkiche. 2016. Multi-fault Attack Detection for RNS Cryptographic Architecture. In *Computer Arithmetic (ARITH), 2016 IEEE 23nd Symposium on.* IEEE, 16–23.

[4] J-C Bajard and Laurent Imbert. 2004. A full RNS implementation of RSA. *IEEE Trans. Comput.* 53, 6 (2004), 769–774.

[5] Ferruccio Barsi and Piero Maestrini. 1974. Error detection and correction by product codes in residue number systems. *IEEE Trans. Comput.* 100, 9 (1974), 915–924.

[6] David T Brown. 1960. Error detecting and correcting binary codes for arithmetic operations. *IRE Transactions on Electronic Computers* 3 (1960), 333–337.

[7] YG.C. Cardarilli and M. Re ; R. Lojacono. 1998. RNS-to-binary conversion for efficient VLSI implementation. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 45 (1998), 667–669.

[8] Chip-Hong Chang, Amir Sabbagh Molahosseini, Azadeh Alsadat Emrani Zarandi, and Tian Fatt Tay. 2015. Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. *IEEE circuits and systems magazine* 15, 4 (2015), 26–44.

[9] Jen-Shiun Chiang and Mi Lu. 1991. Floating-point numbers in residue number systems. *Computers & Mathematics with Applications* 22, 10 (1991), 127–140.

[10] Rooju Chokshi, Krzysztof S Berezowski, Aviral Shrivastava, and Stanislaw J Piestrak. 2009. Exploiting residue number system for power-efficient digital signal processing in embedded processors. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems.* ACM, 19–28.

[11] B. Deng, S. Srikanth, E. R. Hein, P. G. Rabbat, T. M. Conte, E. DeBenedictis, and J. Cook. 2016. Computationally-redundant energy-efficient processing for y'all (CREEPY). In *IEEE International Conference on Rebooting Computing (ICRC).* 1–8.

[12] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (Oct 1974), 256–268.

[13] Elio D Di Claudio, Gianni Orlandi, and Francesco Piazza. 1993. A systolic redundant residue arithmetic error correction circuit. *IEEE Trans. Comput.* 42, 4 (1993), 427–432.

[14] Elio D Di Claudio, Francesco Piazza, and Gianni Orlandi. 1995. Fast combinatorial RNS processors for DSP applications. *IEEE transactions on computers* 44, 5 (1995), 624–633.

[15] Shlomi Dolev, Sergey Frenkel, Dan E Tamir, and Vladimir Sinelnikov. 2013. Preserving Hamming Distance in Arithmetic and Logical Operations. *Journal of Electronic Testing* 29, 6 (2013), 903–907.

[16] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and others. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *Microarchitecture,MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on.* IEEE, 7–18.

[17] M Etzel and W Jenkins. 1980. Redundant residue number systems for error detection and correction in digital filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28, 5 (1980), 538–545.

[18] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. 2009. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 283–296.

[19] Ph Forin. 1989. Vital coded microprocessor principles and application for various transit systems. *IFAC Control, Computers, Communications* (1989), 79–84.

[20] Swaroop Ghosh, Patrick Ndai, and Kaushik Roy. 2008. A novel low overhead fault tolerant Kogge-Stone adder using adaptive clocking. In *Design, Automation and Test in Europe, 2008. DATE'08*. IEEE, 366–371.

[21] Vik Tor Goh and Mohammad Umar Siddiqi. 2008. Multiple error detection and correction based on redundant residue number systems. *IEEE Transactions on Communications* 56, 3 (2008).

[22] Oded Goldreich, Dana Ron, and Madhu Sudan. 1999. Chinese remaindering with errors. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*. ACM, 225–234.

[23] Meeta S Gupta, Krishna K Rangan, Michael D Smith, Gu-Yeon Wei, and David Brooks. 2008. DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors. In *High Performance Computer Architecture, HPCA, IEEE 14th International Symposium on*. IEEE, 381–392.

[24] Nor Zaidi Haron and Said Hamdioui. 2011. Redundant residue number system code for fault-tolerant hybrid memories. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 7, 1 (2011), 4.

[25] C. W. Hastings. 1966. Automatic detection and correction of errors in digital computers using residue arithmetic. In *Region Six Annu. Conf.* IEEE, 429–464.

[26] Yuang-Ming Hsu and EE Swartzlander. 1992. Time redundant error correcting adders and multipliers. In *Defect and Fault Tolerance in VLSI Systems, 1992. Proceedings., 1992 IEEE International Workshop on*. IEEE, 247–256.

[27] Yuang-Ming Hsu and EE Swartzlander. 1992. Time redundant error correcting adders and multipliers. In *Defect and Fault Tolerance in VLSI Systems, Proceedings., International Workshop on*. IEEE, 247–256.

[28] Ching Yu Hung and Behrooz Parhami. 1994. Fast RNS division algorithms for fixed divisors with application to RSA encryption. *Inform. Process. Lett.* 51, 4 (1994), 163–169.

[29] IBM. 2014. IBM Power System E880 server, an IBM POWER8 technology-based system, addresses the requirements of an industry-leading enterprise class system. (2014).

[30] Barry W Johnson, James H Aylor, and Haytham H Hana. 1988. Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder. *IEEE journal of solid-state circuits* 23, 1 (1988), 208–215.

[31] Barry W Johnson, James H Aylor, and Haytham H Hana. 1988. Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder. *journal of solid-state circuits* 23, 1 (1988), 208–215.

[32] Rajendra S. Katti. 1996. A new residue arithmetic error correction scheme. *IEEE transactions on computers* 45, 1 (1996), 13–19.

[33] Osnat Keren, Ilya Levin, Vladimir Ostrovsky, and Beni Abramov. 2008. Arbitrary error detection in combinational circuits by using partitioning. In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*. IEEE, 361–369.

[34] Asif Islam Khan, Korok Chatterjee, Juan Pablo Duarte, Zhongyuan Lu, Angada Sachid, Sourabh Khandelwal, Ramamoorthy Ramesh, Chenming Hu, and Sayeef Salahuddin. 2016. Negative capacitance in short-channel FinFETs externally connected to an epitaxial ferroelectric capacitor. *IEEE Electron Device Letters* 37, 1 (2016), 111–114.

[35] Asif Islam Khan and Sayeef Salahuddin. 2015. 4 Extending CMOS with negative capacitance. *CMOS and Beyond: Logic Switches for Terascale Integrated Circuits* (2015), 56–76.

[36] Asif I Khan, Chun W Yeung, Chenming Hu, and Sayeef Salahuddin. 2011. Ferroelectric negative capacitance MOSFET: Capacitance tuning & antiferroelectric operation. In *Electron Devices Meeting (IEDM), 2011 IEEE International*. IEEE, 11–3.

[37] EV Krekhov, Al-r A Pavlov, AA Pavlov, PA Pavlov, DV Smirnov, AN Tsar'kov, PA Chistopol'skii, AV Shandrikov, BA Sharikov, and DA Yakimov. 2008. A method of monitoring execution of arithmetic operations on computers in computerized monitoring and measuring systems. *Measurement Techniques* 51, 3 (2008), 237–241.

[38] Hari Krishna, Bal Krishna, Kuo-Yu Lin, and Jenn-Dong Sun. 1994. *Computational Number Theory and Digital Signal Processing: Fast Algorithms and Error Control Techniques*. Vol. 6. CRC Press.

[39] Hari Krishna, K-Y Lin, and J-D Sun. 1992. A coding theory approach to error control in redundant residue number systems. I. Theory and single error correction. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39, 1 (1992), 8–17.

[40] Daniel Lipetz and Eric Schwarz. Self Checking in Current Floating-Point Units. In *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic (ARITH '11)*. IEEE Computer Society, Washington, DC, USA, 73–76.

[41] Chao-Kai Liu. 1972. *Error-correcting-codes in computer arithmetic*. Technical Report. DTIC Document.

[42] Hao-Yung Lo and Ting-Wei Lin. 2013. Parallel Algorithms for Residue Scaling and Error Correction in Residue Arithmetic. *Wireless Engineering and Technology* 4, 04 (2013), 198.

[43] Florence Jessie MacWilliams and Neil James Alexander Sloane. 1977. *The theory of error-correcting codes*. Elsevier.

[44] Daniel Marienfeld, Egor S Sogomonyan, Vitalij Ocheretnij, and M Gossel. 2005. New self-checking output-duplicated booth multiplier with high fault coverage for soft errors. In *Test Symposium, 2005. Proceedings. 14th Asian*. IEEE, 76–81.

[45] J Mathew, S Banerjee, P Mahesh, DK Pradhan, AM Jabir, and SP Mohanty. 2010. Multiple bit error detection and correction in GF arithmetic circuits. In *Electronic System Design (ISED), 2010 International Symposium on*. IEEE, 101–106.

[46] A. McMenamin. 2013. The End of Dennard Scaling. (2013).

[47] Elias Mizan, Tileli Amimeur, and Margarida F Jacome. 2007. Self-imposed temporal redundancy: An efficient technique to enhance the reliability of pipelined functional units. In *Computer Architecture and High Performance Computing, SBAC-PAD. 19th International Symposium on*. IEEE, 45–53.

[48] Michael Nicolaidis. 2003. Carry checking/parity prediction adders and ALUs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11, 1 (2003), 121–128.

[49] Michael Nicolaidis and Hakim Bederr. 1994. Efficient implementations of self-checking multiply and divide arrays. In *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings*. IEEE, 574–579.

[50] Michael Nicolaidis and RO Duarte. 1998. Design of fault-secure parity-prediction booth multipliers. In *Design, Automation and Test in Europe, 1998., Proceedings*. IEEE, 7–14.

[51] Eric B Olsen. 2015. Introduction of the Residue Number Arithmetic Logic Unit With Brief Computational Complexity Analysis (Rez-9 soft processor). *Whitepaper, Digital System Research* (2015).

[52] Amos Omondi and Benjamin Premkumar. Residue Number Systems: Theory and Implementation. Imperial College Press.

[53] Glenn A. Orton, Lloyd E. Peppard, and Stafford E. Tavares. 1992. New fault tolerant techniques for residue number systems. *IEEE transactions on computers* 41, 11 (1992), 1453–1464.

[54] Janak H. Patel and Leona Y. Fung. 1982. Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Trans. Computers* 31, 7 (1982), 589–595.

[55] Song Peng and Rajit Manohar. 2005. Fault tolerant asynchronous adder through dynamic self-reconfiguration. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 171–178.

[56] Song Peng and Rajit Manohar. 2005. Fault tolerant asynchronous adder through dynamic self-reconfiguration. In *Computer Design: VLSI in Computers and Processors, ICCD. Proceedings.International Conference on*. IEEE, 171–178.

[57] AP Preethy and D Radhakrishnan. 1999. A 36-bit balanced moduli MAC architecture. In *Circuits and Systems, 1999. 42nd Midwest Symposium on*, Vol. 1. IEEE, 380–383.

[58] AP Preethy and D Radhakrishnan. 2000. RNS-based logarithmic adder. *IEE Proceedings-Computers and Digital Techniques* 147, 4 (2000), 283–287.

[59] Vijaya Ramachandran. 1983. Single residue error correction in residue number systems. *IEEE transactions on computers* 32, 5 (1983), 504–507.

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

[61] Thammavarapu RN Rao. 1970. Biresidue error-correcting codes for computer arithmetic. *IEEE Transactions on computers* 100, 5 (1970), 398–402.

[62] Wenjing Rao and Alex Orailoglu. 2008. Towards fault tolerant parallel prefix adders in nanoelectronic systems. In *Design, Automation and Test in Europe, 2008. DATE'08*. IEEE, 360–365.

[63] Wenjing Rao, Alex Orailoglu, and Ramesh Karri. 2006. Fault identification in reconfigurable carry lookahead adders targeting nanoelectronic fabrics. In *Test Symposium, 2006. ETS'06. Eleventh IEEE European*. IEEE, 63–68.

[64] Stefan Rusu. 2010. Multi-Domain Processors Design Overview. *ISCA tutorial on Multi-domain Processors: Challenges, Design Methods, and Recent Developments* (June 2010).

[65] Sayeef Salahuddin and Supriyo Datta. 2008. Can the subthreshold swing in a classical FET be lowered below 60 mV/decade?. In *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*. IEEE, 1–4.

[66] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. 2010. ANB-and ANBDmem-encoding: detecting hardware errors in software. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 169–182.

[67] Avik Sengupta and Balasubramaniam Natarajan. 2013. Performance of systematic RRNS based space-time block codes with probability-aware adaptive demapping. *IEEE Transactions on Wireless Communications* 12, 5 (2013), 2458–2469.

[68] Y. Shimazaki, R. Zlatanovici, and B. Nikolic. 2004. A shared-well dual-supply-voltage 64-bit ALU. *Journal of Solid-State Circuits* 39, 3 (2004), 494–500.

[69] Sriseshan Srikanth, Bobin Deng, and Thomas M Conte. 2016. A Brief Survey of Non-Residue Based Computational Error Correction. *arXiv preprint arXiv:1611.03099* (2016).

[70] S. Srikanth, P. G. Rabbat, E. R. Hein, B. Deng, T. M. Conte, E. DeBenedictis, J. Cook, and M Frank. Memory System Design for Ultra Low Power, Computationally Error Resilient Processor Microarchitectures. In *International Symposium on High Performance Computer Architecture (HPCA),2018*. [to appear].

[71] C-C Su and H-Y Lo. 1990. An algorithm for scaling and single residue error correction in residue number systems. *IEEE Trans. Comput.* 39, 8 (1990), 1053–1064.

[72] J-D Sun and Hari Krishna. 1992. A coding theory approach to error control in redundant residue number systems. II. Multiple Error detection and correction. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39, 1 (1992), 18–34.

[73] J-D Sun, Hari Krishna, and KY Lin. 1992. A superfast algorithm for single-error correction in RRNS and hardware implementation. In *Circuits and Systems, 1992. ISCAS'92. Proceedings., 1992 IEEE International Symposium on*, Vol. 2. IEEE, 795–798.

[74] Yan Sun, Minxuan Zhang, Shaoqing Li, and Yali Zhao. 2010. Cost effective soft error mitigation for parallel adders by exploiting inherent redundancy. In *IC Design and Technology (ICICDT), 2010 IEEE International Conference on*. IEEE, 224–227.

[75] Andraos Sweidan and Ahmad A Hiasat. 2001. On the theory of error control based on moduli with common factors. *Reliable computing* 7, 3 (2001), 209–218.

[76] Nicholas S Szabo and Richard I Tanaka. 1967. *Residue arithmetic and its applications to computer technology.* McGraw-Hill.

[77] J. Tan and O. Rosen. 2004. Process for determining competing cause event probability and/or system availability during the simultaneous occurrence of multiple events. (April 22 2004). https://www.google.com/patents/US20040078167 US Patent App. 10/272,156.

[78] Yangyang Tang, Emmanuel Boutillon, Christophe Jégo, and Michel Jézéquel. 2010. A new single-error correction scheme based on self-diagnosis residue number arithmetic. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*. IEEE, 27–33.

[79] Thian Fatt Tay and Chip-Hong Chang. 2014. A new algorithm for single residue digit error correction in Redundant Residue Number System. In *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*. IEEE, 1748–1751.

[80] Thian Fatt Tay and Chip-Hong Chang. 2016. A non-iterative multiple residue digit error detection and correction algorithm in RRNS. *IEEE transactions on computers* 65, 2 (2016), 396–408.

[81] T. N. Theis. 2012. (keynote) in quest of a fast, low-voltage digital switch. *ECS Transactions* 45, 6 (2012), 3–11.

[82] T. N. Theis and P. M. Solomon. 2010. In Quest of the Next Switch: Prospects for Greatly Reduced Power Dissipation in a Successor to the Silicon Field-Effect Transistor. *Proc. IEEE* 98, 12 (Dec 2010), 2005–2014.

[83] Mojtaba Valinataj and Saeed Safari. 2007. Fault tolerant arithmetic operations with multiple error detection and correction. In *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*. IEEE, 188–196.

[84] Dilip P Vasudevan and Parag K Lala. 2005. A technique for modular design of self-checking carry-select adder. In *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on*. IEEE, 325–333.

[85] Dilip P Vasudevan, Parag K Lala, and James Patrick Parkerson. 2007. Self-checking carry-select adder design based on two-rail encoding. *IEEE Transactions on Circuits and Systems I: Regular Papers* 54, 12 (2007), 2696–2705.

[86] John Von Neumann. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies* 34 (1956), 43–98.

[87] E George Walters III, Mark G Arnold, and Michael J Schulte. 2003. Using truncated multipliers in DCT and IDCT hardware accelerators. In *Optical Science and Technology, SPIE's 48th Annual Meeting*. International Society for Optics and Photonics, 573–584.

[88] Ute Wappler and Christof Fetzer. 2007. Hardware failure virtualization via software encoded processing. In *Industrial Informatics, 2007 5th IEEE International Conference on*, Vol. 2. IEEE, 977–982.

[89] R. W. Watson. 1965. Error detection and correction and other residue interacting operations in a residue redundant number system. Univ. California, Berkeley.

[90] Richard W Watson and Charles W Hastings. 1966. Self-checked computation using residue arithmetic. *Proc. IEEE* 54, 12 (1966), 1920–1931.

[91] Hanshen Xiao, Hari Krishna Garg, Jianhao Hu, and Guoqiang Xiao. 2016. New Error Control Algorithms for Residue Number System Codes. *ETRI Journal* 38, 2 (2016), 326–336.

[92] Li Xiao and Xiang-Gen Xia. 2015. Error correction in polynomial remainder codes with non-pairwise coprime moduli and robust Chinese remainder theorem for polynomials. *IEEE Transactions on Communications* 63, 3 (2015), 605–616.

[93] SS-S Yau and Yu-Cheng Liu. 1973. Error correction in redundant residue number systems. *IEEE Trans. Comput.* 100, 1 (1973), 5–11.

[94] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. 2003. RSA speedup with Chinese remainder theorem immune against hardware fault cryptanalysis. *IEEE Transactions on computers* 52, 4 (2003), 461–472.

[95] Pengsheng Yin and Lei Li. 2013. A new algorithm for single error correction in RRNS. In *Communications, Circuits and Systems (ICCCAS), 2013 International Conference on*, Vol. 2. IEEE, 178–181.