Modular Exponentiation via Nested Reduction in a Residue Number System

Phil Sun

January 30, 2020

Abstract

We present an algorithm that uses a residue number system to compute modular squarings with low latency. This algorithm is highly amenable to hardware implementation and can be used for fast verifiable delay function (VDF) computation.

1 Introduction

A verifiable delay function (VDF) is a function whose computation is inherently sequential in nature and whose output can be quickly verified. An example of such a function is $x^{2^T} \pmod M$ where M is a semiprime whose factorization is kept private. This VDF can only be computed by squaring x and reducing it modulo M for T iterations, but the result can be verified in $O(\log T)$ operations. Efficient computation of this VDF requires a low-latency algorithm for modular squaring. In this paper we present a novel approach for modular squaring that achieves $O(\log n)$ circuit depth and $O(n^2)$ area for an n-bit modulus. The algorithm uses a residue number system (RNS) and performs no base changes unlike previous RNS algorithms such as [1].

2 Background

A residue number system (RNS) represents an integer x by its residues modulo a collection of bases. Specifically, an RNS uses L bases $m_1, \ldots, m_L$, each of B bits[1], and computes $r_i = x \bmod m_i$ for $i \in 1, \ldots L$. Define P as the product of the bases: $P \triangleq \prod_{i=1}^{L} m_i$. So long as the m_i are pairwise relatively prime and the inequality

$$0 \leq x < P$$

holds, r_i provides a unique and recoverable representation of x. The recovery function g that computes x given $r_1, \ldots, r_L$ is defined below; it's a result of the Chinese Remainder Theorem.

[1]The bases can be of different sizes, but latency is determined by the critical path and therefore operations involving the largest bases. As a result it is best to make the RNS bases uniformly sized.

$$f(r_1 \ldots, r_L) \triangleq \sum_{i=1}^{L} r_i \cdot \frac{P}{m_i} \cdot \left((P/m_i)^{-1} \bmod m_i \right), \tag{1}$$

$$g(r_1, \ldots, r_L) \triangleq f(r_1, \ldots, r_L) \bmod P \tag{2}$$

The return value of g will be an integer x in the range $[0, P)$ such that $x \equiv r_i \bmod m_i$ for i in $1, \ldots, L$.

2.1 RNS Arithmetic

Suppose there is an RNS with L bases, $m_1, \ldots, m_L$. Consider two integers x and y that are represented in this system as $r_1, \ldots, r_L$ and $r'_1, \ldots, r'_L$, respectively. We now examine the expressions for $x - y$, $x + y$, and xy in the RNS.

By the rules of modular arithmetic, we know that if $r_i = x \bmod m_i$ and $r'_i = y \bmod m_i$ then $r_1 + r'_1 = x + y \bmod m_i$. Addition of x and y therefore turns into a pointwise addition in RNS-space. Similarly $x - y$ is a pointwise subtraction, and xy is a pointwise multiplication. Compared to a traditional positional integer representation such as binary or decimal, RNS holds two advantages.

The first is that addition, subtraction, and multiplication are implemented via L independent operations, leading to $O(1)$ latency.[2] In contrast, a positional integer representation has carries that can propagate from the lowest bit to the highest, resulting in $O(n)$ worst-case latency. A redundant representation like the one used in [5] can be used to cut down on latency, but such a representation presents its own complications.

Operation	Positional	Redundant Positional	RNS
Addition	$O(n)$	$O(1)$	$O(1)$
Subtraction	$O(n)$	$O(1)$	$O(1)$
Multiplication	$O(n)$	$O(\log n)$	$O(1)$

Table 1: Latency of arithmetic in various integer representations

RNS's second advantage is its ability to implement multiplication in a linear number of gates. In the classical schoolbook multiplication algorithm used with positional integer representation, each digit must be multiplied by every other digit, leading to $O(n^2)$ gate usage. This can be reduced by algorithms such as Karatsuba or Schonhage-Strassen, but such algorithms have greater constant factors and circuit depth. In addition these algorithms are still superlinear in area.

Operation	Positional	Redundant Positional	RNS
Addition	$O(n)$	$O(n)$	$O(n)$
Subtraction	$O(n)$	$O(n)$	$O(n)$
Multiplication	$O(n^2)$	$O(n^2)$	$O(n)$

Table 2: Implementation size of arithmetic in various integer representations

Both advantages of RNS over positional representation are leveraged in our algorithm.

[2] The following analysis assumes that we can represent an n-bit integer in an RNS with $L = O(n)$ and $B = O(1)$. This is a simplification that is discussed in the appendix.

3 Notation

Below is a reference for the notation used throughout the paper. We also provide each symbol's value as tuned for a 1024-bit modulus to give a sense of relative magnitudes.

Symbol	Definition	Value
M	Modulus provided as VDF parameter	Approx 2^{1024}
n	Bits in M	1024
L	Number of RNS bases	128
B	Bits in each RNS base	17
m_i	i^{th} RNS base	Around 77000
r_i	i^{th} residue in RNS representation	Ranges from 0 to $m_i - 1$
P	Product of RNS bases	Approx 2^{2076}
x	Number represented by the RNS	Ranges from 0 to $P - 1$

4 Algorithm in Detail

At a high level, the algorithm takes in an input x, squares it, and reduces it by subtracting the largest multiple of M that is less than or equal to x.

Algorithm 1 Algorithm Overview

Input: x, T, M
Output: $x^{2^T} \bmod M$

1: **for** t in $1, \ldots T$ **do**
2: $\quad x := x^2$
3: $\quad x := x - M\lfloor x/M \rfloor$
4: **end for**
5: **return** x

However, our algorithm operates in RNS-space. Using the RNS recovery function g defined in equation 2, our pseudocode becomes:

Algorithm 2 Algorithm Overview in RNS Space

Input: x, T, M; RNS parameters L and $m_1, \ldots, m_L$
Output: $x^{2^T} \bmod M$

1: **for** i in $1, \ldots L$ **do**
2: $\quad r_i := x \bmod m_i$
3: **end for**
4: **for** t in $1, \ldots T$ **do**
5: $\quad$ **for** i in $1, \ldots L$ **do**
6: $\quad\quad r_i := r_i^2 \bmod m_i$
7: $\quad\quad r_i := r_i - M\lfloor g(r_1, \ldots, r_L)/M \rfloor \bmod m_i$
8: $\quad$ **end for**
9: **end for**
10: **return** $g(r_1, \ldots, r_L)$

Line 6 in algorithm 2 corresponds to line 2 in algorithm 1; they perform squaring, which is

trivial to implement in RNS. Line 7 in algorithm 2 corresponds to line 3 in algorithm 1 and performs modular reduction. This line involves division, which is difficult to implement in RNS. The primary contribution of this paper is an efficient method of doing this reduction.

To computationally expedite line 7, we can begin with the following:

$$\begin{aligned} r_i &:= r_i - M\lfloor g(r_1, \ldots, r_L)/M \rfloor \bmod m_i \\ &= r_i - (M \bmod m_i) \left\lfloor \frac{g(r_1, \ldots, r_L)}{M} \bmod m_i \right\rfloor \end{aligned}$$

Where we define $a \bmod m$ to be b such that $0 \leq b < m$ and $a = b + km$ for some integer k. Note this definition applies even to non-integer a such as the expression in the floor function above.

Now recall that $g = f \bmod P$ so $g = f - P\lfloor f/P \rfloor$. Note this is the same technique we used to reduce x, hence the nested reduction that gives the algorithm its name. This allows us to decompose our expression further:

$$r_i := r_i - (M \bmod m_i) \left\lfloor \frac{f(r_1, \ldots, r_L)}{M} - \frac{P\lfloor f(r_1, \ldots, r_L)/P \rfloor}{M} \bmod m_i \right\rfloor \tag{3}$$

If we define

$$p_1 \triangleq \frac{f(r_1, \ldots, r_L)}{M}; \quad p_2 \triangleq \frac{P\lfloor f(r_1, \ldots, r_L)/P \rfloor}{M}$$

then inside the floor function of equation 3 we have the difference of two fractional quantities: $\lfloor (p_1 \bmod m_i) - (p_2 \bmod m_i) \rfloor$. We now prove that this difference can be decomposed into the difference of two floored quantities. For real numbers a and b, let $\{a\}$ and $\{b\}$ represent the fractional parts of each, respectively: $a = \lfloor a \rfloor + \{a\}$, $b = \lfloor b \rfloor + \{b\}$.

$$\begin{aligned} \lfloor a - b \rfloor &= \lfloor \lfloor a \rfloor + \{a\} - (\lfloor b \rfloor + \{b\}) \rfloor \\ &= \lfloor \lfloor a \rfloor - \lfloor b \rfloor + \{a\} - \{b\} \rfloor \\ &= \lfloor a \rfloor - \lfloor b \rfloor + \lfloor \{a\} - \{b\} \rfloor \end{aligned}$$

By definition $0 \leq \{a\}, \{b\} < 1$, so $-1 < \{a\} - \{b\} < 1$ and therefore $\lfloor \{a\} - \{b\} \rfloor$ is either 0 or -1. Applying this to our equation, we know:

$$\lfloor p_1 \bmod m_i \rfloor - \lfloor p_2 \bmod m_i \rfloor + 1 = \lfloor p_1 \bmod m_i - p_2 \bmod m_i \rfloor + (0 \text{ or } 1)$$

We can therefore decompose equation 3 as follows:

$$r_i := r_i - \underbrace{(M \bmod m_i)\lfloor p_1 \bmod m_i \rfloor}_{s_i} + \underbrace{(M \bmod m_i)\lfloor p_2 \bmod m_i \rfloor}_{t_i} + (M \bmod m_i) \tag{4}$$

The off-by-one from the decomposition of the floor leads to a potential added $M \bmod m_i$ to each r_i. This is equivalent to adding M to the integer our RNS represents, which is harmless because all results are taken modulo M regardless. So long as we select RNS parameters that have the capacity to handle the incomplete reduction, the correctness of the result is not impacted. We now go into detail describing the computation of the two components of our decomposed expression, s_i and t_i.

4.1 Computation of s_i

Our goal is to compute $M\lfloor f(r_1,\ldots,r_L)/M\rfloor \bmod m_i$ for all $i \in \{1,\ldots,L\}$. We do this by computing $\lfloor f(r_1,\ldots,r_L)/M\rfloor \bmod m_i$ then passing the result into a lookup table that given x returns $Mx \bmod m_i$. We can compute $\lfloor f(r_1,\ldots,r_L)/M\rfloor \bmod m_i$ as follows:

$$\begin{aligned}
\lfloor f(r_1,\ldots,r_L)/M\rfloor \bmod m_i &= \left\lfloor \frac{\sum_{j=1}^{L} r_j \cdot \dfrac{P}{m_j} \cdot ((P/m_j)^{-1} \bmod m_j)}{M} \right\rfloor \bmod m_i \\
&= \left\lfloor \sum_{j=1}^{L} r_j \cdot \frac{P/M}{m_j} \cdot \left((P/m_j)^{-1} \bmod m_j\right) \right\rfloor \bmod m_i \\
&= \left\lfloor \sum_{j=1}^{L} r_j \cdot \left[\frac{P/M}{m_j} \cdot \left((P/m_j)^{-1} \bmod m_j\right) \bmod m_i \right] \right\rfloor \qquad (5)
\end{aligned}$$

We can see in equation (5) that r_j is multiplied by a precomputable constant. This allows us to formulate the problem as a matrix-vector multiplication. Consider matrix A where $A_{i,j} = \dfrac{P/M}{m_j}((P/m_j)^{-1} \bmod m_j) \bmod m_i$; let $\vec{r}$ be the column vector of all r_i. Then $A\vec{r}$ equals our desired quantity $\lfloor f(r_1,\ldots,r_L)/M\rfloor \bmod m_i$.

In practice we use lookup tables to compute the matrix-vector product. We also decompose the floor in equation (5) and quantize each product individually before summing:

$$\lfloor f(r_1,\ldots,r_L)/M\rfloor \bmod m_i \approx \sum_{j=1}^{L} \left\lfloor r_j \cdot \left[\frac{P/M}{m_j} \cdot \left((P/m_j)^{-1} \bmod m_j\right) \bmod m_i \right] \right\rfloor$$

By quantizing individually we no longer need as much precision in each $A_{i,j}r_i$ product. This shrinks our LUT which is very important because this matrix-vector product is the dominant contributor to gate usage in our algorithm. Using the same analysis as we did when decomposing equation (3) to (4), we see that our decomposed expression is somewhere between 0 and $L-1$ under the actual expression. This in effect means we are under-reducing the integer we squared by an additive factor of up to $(L-1)M$. As long as our RNS has sufficient capacity to represent $(LM)^2$ this doesn't affect accuracy, and the increase in computational efficiency this decomposition grants makes this tradeoff worthwhile.

After computing $\lfloor f(r_1,\ldots,r_L)/M\rfloor \bmod m_i$, we use an LUT to multiply the quantity by M and reduce it modulo m_i. This then gives us s_i as desired. Overall this computation contributes $\log L$ gate delay due to the summation of L integers in the matrix-vector product. It consumes $O(L^2)$ area.

4.2 Computation of t_i

First note that $f(r_1,\ldots,r_L)/P$, which is required in the evaluation of t_i, is much smaller than $f(r_1,\ldots,r_L)/M$, the quantity used in s_i:

$$f(r_1, \ldots, r_L)/P = \frac{\sum_{i=1}^{L} r_i \cdot \frac{P}{m_i} \cdot \left((P/m_i)^{-1} \bmod m_i\right)}{P}$$
$$= \sum_{i=1}^{L} r_i \cdot \frac{(P/m_i)^{-1} \bmod m_i}{m_i} \tag{6}$$

Recall that $(P/m_i)^{-1} \bmod m_i$ by the definition of modular inverse must be in the range $[0, m_i)$. This leads to

$$f(r_1, \ldots, r_L)/P < \sum_{i=1}^{L} r_i \cdot \frac{m_i}{m_i} = \sum_{i=1}^{L} r_i$$

so the quantity is only a few bits larger than r_i. In comparison, $f(r_1, \ldots, r_L)/M$ is P/M times larger, which for typical M implies hundreds to thousands of bits. Such a large number would be computationally intractable which is why we computed $f(r_1, \ldots, r_L)/M \bmod c_i$ in the previous section; we split a single large number into L numbers of B bits each. This time, however, we have a single number not much longer than B bits; taking it modulo c_i would actually increase our work because we'd have L numbers instead of one, but each wouldn't be much smaller.

For this reason we compute $f(r_1, \ldots, r_L)/P$ directly. From equation (6) we can see this can be viewed as a dot product; our desired result equals $\vec{r} \cdot \vec{v}$ where $\vec{r}$ stores the current RNS residues and $\vec{v}_i = \frac{(P/m_i)^{-1} \bmod c_i}{m_i}$.

Another crucial difference from computing s_i is that $\lfloor f(r_1, \ldots, r_L)/P \rfloor$ must be calculated exactly. Looking at the definition of t_i:

$$t_i \triangleq M \left\lfloor \frac{P}{M} \lfloor f(r_1, \ldots, r_L)/P \rfloor \right\rfloor \bmod m_i$$

we see that an approximation error of δ in $\lfloor f(r_1, \ldots, r_L)/P \rfloor$ leads to a final error of $\delta \cdot \frac{P}{M} \cdot M$. In comparison, when computing s_i the error was only δM. The multiplier of P/M is so large that even $\delta = 1$ would lead to catastrophic overflow and an incorrect result.

To get an exact result, we define $\vec{v}_i$ as

$$\left\lfloor 1 + 2^k \cdot \frac{(P/m_i)^{-1} \bmod m_i}{m_i} \right\rfloor$$

and then divide our dot product by 2^k. This provably gives the correct answer for sufficiently large k; for a 1024-bit modulus, we need $k = 24$. This technique is known as using a redundant modulus and is introduced and analyzed in [4].

After computing our exact dot product, we use L LUTs to give us t_i. The i^{th} LUT takes in an integer x and returns $M\lfloor Px/M \rfloor \bmod m_i$. Feeding the dot product into each of the LUTs gives us all $t_1, \ldots, t_L$ as desired.

5 FPGA Implementation

We implemented the algorithm on a Xilinx XCVU9P FPGA. The FPGA's DSP48E2 DSPs performed the squaring of the r_i (line 6 of algorithm 2). All other multiplications in the algorithm are multiplications by a constant and were implemented via lookup table. Lookup-based multiplication is advantageous because it can pre-reduce the output; DSP based multiplication outputs a full multiplication result that requires reduction, which would then necessitate the use of LUTs again. In general we aimed to use LUTs of 5 and 6 bits which best align with the FPGA's internally available resources. We chose not to perform the squaring via LUT because squaring is a non-linear operation that isn't easily decomposed into smaller LUTs.

Our FPGA implementation used a purely combinatorial design. Every other clock cycle, the output of the combinatorial circuit was assigned back to the input. The modular squaring latency was therefore twice the clock period. The RNS parameters chosen for these three implementations were optimized for a 1024-bit modulus and we believe significantly better performance could have been achieved with proper tuning for these smaller modulus sizes.

For the 128 and 256 bit implementations, the matrix multiplication LUT used to calculate s_i was implemented as $3BL^2$ LUT6 primitives; each of the L^2 matrix entries used $3B$ LUT6 primitives to compute the multiplication. For the 256-bit implementation this required $3 \cdot 17 \cdot 32^2 = 52224$ LUT6s. Scaling this to 512 bits would've quadrupled usage and prevented the logic from fitting in one SLR. We planned to circumvent this issue by using $3BL^2$ LUT5 primitives and $3BL^2$ registers. An LUT6 can be used as two LUT5s so this should've halved our LUT usage and allowed our design to fit in a single SLR. However, we were unable to get the Xilinx synthesizer to infer the LUT arragement we desired. We believe with a few simple hints to the synthesizer it would infer correctly and likely lead to a design with a 15-20ns latency.

For the 1024-bit implementation, we determined that even the $3\times5+3\times1$ arrangement used for the 512-bit multiplier would use too many LUTs. Further decomposition of the 18-bit product into smaller LUTs wasn't possible because smaller lookups don't physically exist in the FPGA fabric; they're simply implemented as 5-bit LUTs and therefore save no space. The only solution was to "temporally multiplex" the LUT and read it over several cycles. For example, an LUT6 can provide 16 bits of lookup over 4 cycles by providing 4 bits per cycle. However, this was a significant redesign that wasn't finished in time for the competition deadline.

5.1 Implementation Results

We achieved the following latencies:

Bits in modulus	Gate Delay	Wire Delay	Achieved Latency
121	5.38ns	7.78ns	13.53ns
254	5.91ns	8.19ns	14.62ns

Table 3: FPGA implementation latencies for various sized moduli

6 Comparison with Alternative Algorithms

The below table lists five possible combinations of integer representation and reduction technique for fast modular squaring:

Integer Representation	**Reduction Method**		
	Barrett	Montgomery	LUT-based
RNS	(Ours)	[1]	N/A
Traditional	[2]	[3]	[5]

Table 4: Modular squaring algorithms categorized by approach

An LUT-based reduction technique like the one used in [5] can't be effectively applied to an RNS representation, so among the three reduction methods and two representation systems we get five valid combinations.

All five algorithms can be implemented to use $O(n^2)$ area with $O(\log n)$ gate delay; comparisons come down to constant factors. We will first compare our algorithm to [5] and [1]. Collectively these three algorithms all heavily utilize LUTs while [2] and [3] are purely multiplication based.

6.1 LUT-centric Algorithm Comparison

LUT area consumption can be variably configured by decomposing one LUT into several smaller ones. For example, the original implementation of [5] used the combination of an 8-bit with a 9-bit LUT to reduce 17 bits of the output. This would use $2^8 + 2^9 = 768$ entries. However, the implementation also could've used two 6-bit and one 5-bit LUT to accomplish the same thing. This would've only used $2 \cdot 2^6 + 2^5 = 160$ entries at the cost of using more adders. For sake of comparison we will compare all three LUT-heavy algorithms assuming LUTs with B-bit inputs. While this may be an unrealistically large LUT size, all algorithms scale similarly when decomposing to smaller LUTs so this is a fair basis for evaluation.

We found Montgomery reduction a poor match for RNS representation. We developed an approach, described in the appendix, that improves on [1] by allowing the two RNS base changes to occur parallel. This allowed the two asymptotically signficant contributors to gate delay to happen in parallel rather than in sequence. Even with this improvement, we found the algorithm too complex and high in constant factor relative to the one described in this paper. We therefore focus on comparison with [5].

If each base is B bits, our algorithm requires L bases such that $BL \geq 2n$ so that our RNS system has enough capacity to store M^2. We use $O(n)$ gates and incur $O(1)$ gate delay in the first portion of our algorithm where we square and reduce each r_i. In the reduction phase, we compute s_i and t_i in parallel. The s_i computation is the dominant source of gate usage; it requires L^2 lookups of B bit to B bit integers. Implementing an adder tree to sum the results then uses L^2 B-bit adders and requires $O(\log n)$ gate delay. The combination and reduction after computing s_i and t_i are asymptotically insignificant in terms of latency and area.

The algorithm described in [5] splits the integer into L' pieces of B bits each such that $L'B = n$, so $L' = L/2$. This algorithm uses L'^2 multipliers to compute the components of

the squared input. This results in L'^2 results of size $2B$ each, which requires $2L'^2$ B-bit adders to combine. For reduction, the high n bits of the result are reduced by LUT. If we implement this with B-bit LUTs, this requires L' such LUTs, each $n = BL'$ bits wide. This is equivalent in area to $L'^2 = L^2/4$ LUTs of B bits to B bits. These LUT results then have to be combined, which uses L'^2 B-bit adders.

Comparing our algorithm with [5], we see that ours is superior in gate delay while the comparison in terms of area is ambiguous. Our algorithm has a single adder tree of L elements in its critical path, while [5] has two trees of $L/2$ elements, one from the squaring step and one from the reduction step. For all relevant L, $\log L < 2 \log \frac{L}{2}$ and in fact $\lim_{L \to \infty} \frac{\log L}{2 \log(L/2)} = \frac{1}{2}$. Meanwhile, gate consumption is summarized in the table belows. Depending on the relative size of multipliers compared to LUTs and adders, it's unclear which has the advantage.

	Ours	[5]
$B \times B$ Multipliers	L	$L^2/4$
B-bit to B-bit LUT	L^2	$L^2/4$
B-bit adder	L^2	$0.75L^2$

Table 5: Modular squaring algorithms categorized by approach

6.2 Non-RNS Barrett and Montgomery Reduction

Algorithms [2] and [3] are purely multiplication based. They both use two multiplications to reduce their input. This means their critical path has three $O(\log n)$ components: one multiplication to square the input, and two multiplications that must be done in sequence for the reduction. In comparison the RNS-based approaches have one $O(\log n)$ component while [5] has two. It is difficult to compare constant factors but we predict that implementations of [2] and [3] have a disadvantage in gate delay. However, these two algorithms can easily be implemented in subquadratic area by using a fast multiplication algorithm such as Karatsuba. Such an approach decreases wire delay and shows promise once wire delay becomes the dominant contributor to latency. We are unsure if this happens at scales relevant for VDF computation or not until much larger n.

7 Conclusion

We present a novel algorithm for low-latency multiplication using an RNS integer representation. The algorithm has lower gate delay than [5] and comparable area usage. Preliminary FPGA implementation of this algorithm shows great promise.

References

[1] J. . Bajard and L. Imbert. A full rns implementation of rsa. *IEEE Transactions on Computers*, 53(6):769–774, June 2004.

[2] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in Cryptology—CRYPTO '86*, page 311–323, Berlin, Heidelberg, 1987. Springer-Verlag.

[3] Peter L. Montgomery. Modular multiplication without trial division. 1985.

[4] A. P. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in rns. *IEEE Transactions on Computers*, 38(2):292–297, Feb 1989.

[5] Erdinç Öztürk. Modular multiplication algorithm suitable for low-latency circuit implementations. Cryptology ePrint Archive, Report 2019/826, 2019. `https://eprint.iacr.org/2019/826`.

A Analysis of B and L in Relation to n

In this section we show that aymptotically, we require $B = O(\log n)$ in order to find sufficient bases to represent an n-bit integer. In practice this asymptotic analysis isn't very predictive of behavior in the range of n typically used for VDFs, and $B = 32$ is sufficient to represent integers of any practical size, so the rest of the paper assumes $B = O(1)$ and $L = O(n)$.

Let $\pi(x)$ be the prime counting function; it returns the number of primes below x. The prime number theorem states that

$$\lim_{x \to \infty} \pi(x) = \frac{x}{\ln x}$$

The number of prime numbers representable with exactly B bits equals $\pi(2^B) - \pi(2^{B-1})$ which using the prime number theorem we can show approaches

$$\begin{aligned}
\frac{2^B}{\ln 2^B} - \frac{2^{B-1}}{\ln 2^{B-1}} &= \log_2 e \left(\frac{2^B}{B} - \frac{2^{B-1}}{B-1} \right) \\
&= \log_2 e \cdot \frac{(B-1) \cdot 2^B - B \cdot 2^{B-1}}{B(B-1)} \\
&= \log_2 e \cdot \frac{(B-2) \cdot 2^{B-1}}{B(B-1)}
\end{aligned}$$

Each of these primes is greater than 2^{B-1} so each provides at least $B - 1$ bits of information. The total amount of information expressible in an RNS with all of these primes as bases is therefore at least

$$\log_2 e \cdot (B-1) \cdot \frac{(B-2) \cdot 2^{B-1}}{B(B-1)} = O(2^B)$$

Setting $O(2^B) = n$ gives $B = O(\log n)$ as desired.

B A Montgomery RNS Algorithm with Parallel Base Changes

We present a sketch of an improvement over [1] that allows the RNS base changes to occur in parallel. This shortens the critical path of the algorithm. In this section of the paper we adopt the notation of [1]. Below is a reproduction of Algorithm 1 from [1] to serve as a basis of comparison:

Algorithm 3 Algorithm 1 from [1]

Input: Two RNS bases $B = (m_1, \ldots, m_k)$ and $B' = (m_{k+1}, \ldots, m_{2k})$, such that $M = \prod_{i=1}^{k} m_i < M' = \prod_{i=1}^{k} m_{k+i}$ and $\gcd(M, M') = 1$; a redundant modulus m_r, $\gcd(m_r, m_i) = 1 \forall i = 1...2k$; a positive integer N represented in RNS in both bases such that $0 < (k+2)^2 N < M$ and $\gcd(N, M) = 1$; two positive integers a, b represented in RNS in both bases with $ab < MN$.

Output: A positive integer $\hat{r} \equiv abM^{-1} \pmod{N}$ represented in RNS in both bases, with $\hat{r} < (k+2)N$.

1: $q \leftarrow (a \times b) \times (-N^{-1})$ in B
2: $[q \text{ in } B] \longrightarrow [\hat{q} \text{ in } B']$ *First base extension*
3: $\hat{r} \leftarrow (a \times b + \hat{q} \times N) \times M^{-1}$ in B'
4: $[\hat{r} \text{ in } B] \longleftarrow [\hat{r} \text{ in } B']$ *Second base extension*

Note that lines 2 and 4 are $O(\log n)$ delay steps while lines 1 and 3 are $O(1)$ delay steps, so performing the base extensions in parallel is important. The current algorithm requires q to be computed in B and $\hat{r}$ to be computed in B' so parallelization isn't possible.

In our new approach, we choose $m_{k+i} = m_i^2$ for all $i = 1...k$ instead of choosing arbitrary bases in B'; this means $M' = M^2$. Now consider line 3; suppose we have already computed $(a \times b + \hat{q} \times N)$ and are about to divide by M. Looking at the modular congruence this implies on a single base, we see:

$$\begin{aligned} \hat{r}_i &\equiv (a \times b + \hat{q} \times N) \pmod{m_{k+i}} \\ &\equiv (a \times b + \hat{q} \times N) \pmod{m_i^2} \end{aligned}$$

When we multiply by M^{-1} this is equivalent to dividing by m_i and then multiplying by $(M/m_i)^{-1}$. Crucially, dividing by m_i changes our modulus from m_i^2 to m_i, and thereby doing our B' to B conversion "for free." In return, we only have computed $[\hat{r} \text{ in } B]$ but never $[\hat{r} \text{ in } B']$, which we still need. So we still need to do two base changes, but they are now arranged in a way that allows them to be done in parallel:

Algorithm 4 Algorithm 1 from [1], with parallel base changes

1: $q \leftarrow (a \times b) \times (-N^{-1})$ in B
2: $[q \text{ in } B] \longrightarrow [\hat{q} \text{ in } B']$ *First base extension*
3: $[(a \times b) \text{ in } B] \longrightarrow [(a \times b) \text{ in } B']$ *Second base extension*
4: $\hat{r} \leftarrow (a \times b + \hat{q} \times N) \times M^{-1}$ in B'

We chose not to implement this algorithm because of its high constant factor. Each base extension involves a matrix-vector and a vector-vector product, and two extensions required. Line 4 also requires a fair amount of arithmetic.