A Karatsuba-based Montgomery Multiplier

Gary C.T. Chow*, Ken Eguro†, Wayne Luk* and Philip Leong‡
*Department of Computing, Imperial College London, UK
Email: {cchow, wl}@doc.ic.ac.uk
†Embedded and Reconfigurable Computing Group, Microsoft Research, Redmond, WA, USA
Email: eguro@microsoft.com
‡School of Electrical and Information Engineering, University of Sydney, Australia
Email: philip.leong@sydney.edu.au

***Abstract*—Modular multiplication of long integers is an important building block for cryptographic algorithms. Although several FPGA accelerators have been proposed for large modular multiplication, previous systems have been based on $O(N^2)$ algorithms. In this paper, we present a Montgomery multiplier that incorporates the more efficient Karatsuba algorithm which is $O(N^{(\log 3/\log 2)})$. This system is parameterizable to different bit-widths and makes excellent use of both embedded multipliers and fine-grained logic. The design has significantly lower LUT-delay product and multiplier-delay product compared with previous designs. Initial testing on a Virtex-6 FPGA showed that it is 60-190 times faster than an optimized multi-threaded software implementation running on an Intel Xeon 2.5 GHz CPU. The proposed multiplier system is also estimated to be 95-189 times more energy efficient than the software-based implementation. This high performance and energy efficiency makes it suitable for server-side applications running in a datacenter environment.**

I. Introduction

Montgomery multiplication [1] is critical for many cryptographic algorithms such as RSA, Digital Signature Algorithm (DSA), Elliptic Curve DSA and other emerging cryptographic algorithms, such as pairing-based systems. Despite improvements in the clock frequency and the level of parallelism of conventional microprocessors, software-based implementations of Montgomery multiplication remain insufficient, both in terms of performance and energy efficiency. This has led to research investigating more efficient hardware accelerators. We focus on Montgomery multiplication for long integers e.g. such as required for operations over the finite field GF(p) where p is a 512-bit prime number. These computations have long carry chains and their implementations are fundamentally different from systems that use composite fields with small characteristic (e.g. GF(2^{167})).

Existing FPGA Montgomery multiplier implementations for large integer systems have one major weakness: they all use algorithms with O(N^2) complexity i.e. the area and/or runtime increases quadratically with the bit-width of the computation. In this paper, we improve the implementation of Montgomery multiplication of long integers at the algorithmic level in order to lower complexity and increase throughput. We have developed a parameterized Karatsuba multiplier using a combination of multiple-precision and coarse-grained carry-save addition techniques. This method has complexity $O(N^{(\log 3/\log 2)})$.

The major contributions of this work are:

- An FPGA-optimized design for irregular, recursive Karatsuba multiplication that is parameterizable to different bit-widths and a batch-pipelined Montgomery multiplier based on the Karastuba multiplier. (Section III)
- We compare the performance and energy efficiency of the proposed Montgomery multiplier with existing software and hardware implementations. (Section IV)

II. Background

A. Software-Based Montgomery Multiplication

Software implementations of Montgomery multiplication usually utilize the straightforward algorithm described in [1] or the multiple-precision algorithms discussed in [2]. Algorithm 1 shows the straightforward Montgomery multiplication algorithm for two k-bit inputs $\bar{a}$ and $\bar{b}$. It requires three k-bit multiplications, one $2k$-bit addition and one k-bit subtraction. The complexity of the straightforward Montgomery multiplication depends on the complexity of the underlying integer multiplications.

Algorithm 1 Montgomery multiplication algorithm

Input: $0 \leq (\bar{a}, \bar{b}) < p, 2^{k-1} \leq p < 2^k$
Precompute: $r = 2^k$, p' and r^{-1} satisifying $r \cdot r^{-1} - p \cdot p' = 1$
Output: $\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1}$

1: $t_1 \leftarrow \bar{a} \cdot \bar{b}$
2: $t_2 \leftarrow t_1 \cdot p' \pmod r$
3: $u \leftarrow (t_1 + t_2 \cdot p)/r$
4: if $u \geq p$ then return $u - p$ else return u

Multiple-precision algorithms are also commonly used for computing Montgomery multiplication. Large bit-width integers are broken down into smaller limbs that are computed sequentially. Different multiple-precision algorithms have been proposed. Koc et al. analyzed these algorithms and classified them into 5 categories. All of them require $(2s^2 + s)$ w-bit multiplications[1].

[1]Number of limb (s) = ⌈ Total bit-width (N) / word-width (w) ⌉.

B. Related Hardware Implementations

Existing FPGA implementations of Montgomery multiplication can be classified into two major categories: bit-wise and block-wise approaches. Bitwise implementations can be found in [3], [4], [5], [6], [7], [8]. These systems do not use any explicit multiplication, implementing the entire algorithm using solely addition. They require O(N) additions, each with bit-width of N. Thus, they have at least O(N^2) complexity where N is the bit-width of the computation. Such systems typically need to use carry save techniques to overcome the problem of long carry chains. Not only do these systems suffer from problems stemming from the massive summations involved, they are unable to make efficient use of a resource that most modern FPGAs provide. Since they do not use multiplication, the embedded multipliers in these FPGAs are wasted.

Block-wise Montgomery multiplier designs typically use one of the five multiple-precision algorithms from [2] or similar algorithms with quotient pipelining from [9]. Found in [10], [9], [11], [12], these systems use embedded multipliers and have O(s^2) multiplication complexity.

III. Parametric Karatsuba integer multiplier

The key to our Montgomery multiplier design is the recursive Karatsuba algorithm. This allows us to compute modular multiplication with a complexity approaching $O(N^{(\log 3/\log 2)})$. Background on the Karatsuba algorithm is provided in Section A. Our design uses multiple-precision arithmetic techniques so that the critical path delay is independent of the multiplier's bit-width.

Unless stated otherwise, we assume we are multiplying two $2k$-bit unsigned integers and the limb-widths of all components are w. The number of limbs in a $2k$-bit word is ($2s = \lceil 2k/w \rceil$). We use either a coarse-grained carry-save technique or a pipelined multiple-precision technique in all of our adders and subtracters. The critical path of the circuit primarily depends upon the limb-width w.

A. Karatsuba algorithm

The Karatsuba multiplication algorithm was proposed by Karatsuba and Ofman in 1962 [13].To illustrate the algorithm, we let X and Y be two $2k$-bit unsigned integers and split them both in half.

$$X = 2^k X_1 + X_0 \text{ and } Y = 2^k Y_1 + Y_0$$

In conventional long multiplication, the product XY is computed with four k-bit multiplications and three additions as shown Equation 1.

$$\begin{aligned} XY &= 2^{2k} z_2 + 2^k z_1 + z_0 \\ z_2 &= X_1 Y_1, z_1 = X_0 Y_1 + X_1 Y_0, z_0 = X_0 Y_0 \end{aligned} \quad (1)$$

As shown in Equation 2, Karatsuba noticed that the middle term z_1 can be computed reusing the terms z_2 and z_0. Reusing these terms allow us to replace two of the multiplications and one addition with four additions/subtractions and one multiplication.

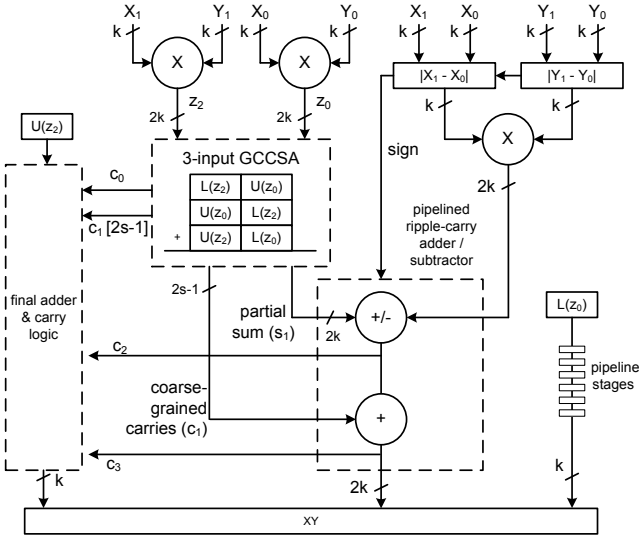

Fig. 1. Block diagram of recursive Karatsuba multiplier.

$$\begin{aligned} z_1 &= X_0 Y_1 + X_1 Y_0 \\ &= X_1 Y_1 + X_0 Y_0 - (X_1 - X_0)(Y_1 - Y_0) \\ &= z_2 + z_0 - (X_1 - X_0)(Y_1 - Y_0) \\ XY &= T_1 - T_2 \\ T_1 &= 2^{2k} z_2 + z_0 + 2^k (z_2 + z_0) \\ T_2 &= 2^k (X_1 - X_0)(Y_1 - Y_0) \end{aligned} \quad (2)$$

B. High-level architecture

We designed a parameterizable Karatsuba layer and apply it recursively. In our prototype architecture, we achieve high performance by fully parallelizing and pipelining all Karatsuba layers.

We also optimized the original Karatsuba algorithm to suit our hardware implementation. First, instead of computing the product of two 2's complement numbers (T_2) in Equation 2, we compute the absolute value of $(X_1 - X_0)$ and $(Y_1 - Y_0)$. These absolute values are then added to or subtracted from the first term (T_1). Second, we take advantage of the fact that the sum (lower half z_2 + upper half z_0) actually appears twice and rearrange T_1 as Equation 3. This allows us to remove some logic required for the addition.

$$\begin{aligned} T_1 &= z_{00} + (2^{2k} + 2^k)(z_{01} + z_{20}) \\ &\quad + 2^k z_{00} + 2^{2k} z_{20} + 2^{3k} z_{21} \\ z_0 &= 2^k z_{01} + z_{00} \end{aligned} \quad (3)$$

Figure 1 shows the block diagram of a single level of our recursive Karatsuba multiplier for $2k$-bit inputs. It is constructed from three k-bit Karatsuba multipliers (or embedded multipliers if k is equal to or smaller than the native multiplier bit-width). We use a coarse-grained carry-save adder

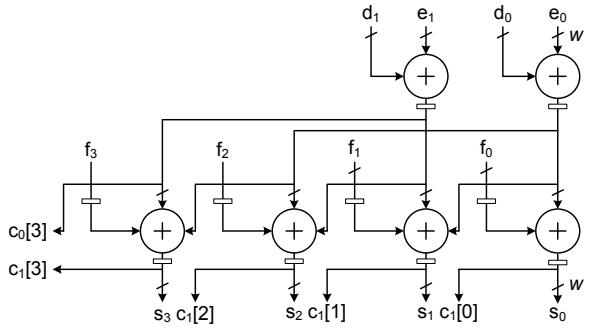

Fig. 2. An example 3-input coarse grained carry-save adder with 4 limbs.

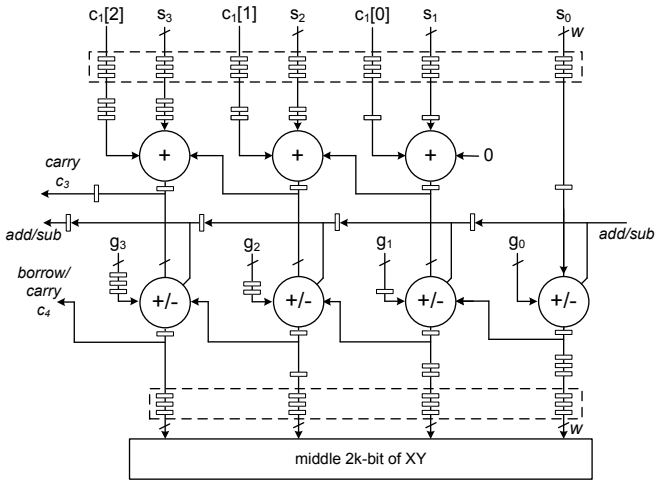

Fig. 3. Multiple-precision 3-input adder / subtracter.

(CGCSA) for the first three additions in the middle $2k$-bits. Pipelined multiple-precision adders are used in the 3-input adder, the final adder logic and the absolute value units. They are carefully pipelined to minimize latency while achieving high clock frequency.

C. Detailed implementation

We denote (d = L(z_2),U(z_0)) as the first input to the coarse-grained carry-save adder, (e = U(z_0),L(z_2)) as the second and (f = U(z_2),L(z_0)) as the third. Figure 2 shows a block diagram of an example 3-input CGCSA with 4 limbs. In the first cycle, the lower half of inputs d and e are summed limb-wise, producing a k-bit partial sum and an s-bit coarse-grained carry. These sums and carries are then duplicated and added to a registered version of input f. The 3-input adder generates a $2k$-bit partial sums (s), coarse-grained carries ($c_1[2s-2..0]$) and two MSB carries (c_0 and $c_1[2s-1]$). Shown in Figure 1, the partial sum and coarse-grained carries are fed into the 3-input multiple-precision adder/subtracter, while the two MSB carries are pipelined and fed into the final adder.

As shown in Figure 1, the 3-input pipelined ripple-carry adder / subtracter is used to combine the partial sum and coarse-grained carries from the CGCSA and add or subtract the third partial product $(X_1-X_0)(Y_1-Y_0)$, depending on the signs from the absolute value units. Figure 3 shows an example

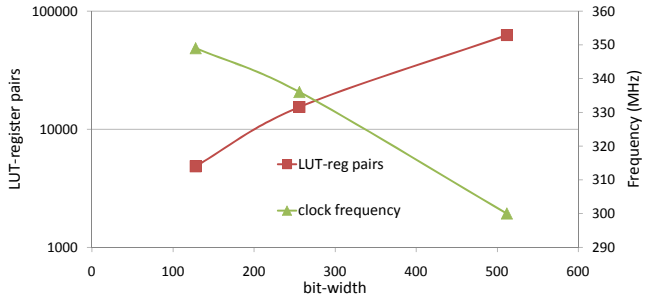

Fig. 4. Result of our batch-pipelined Montgomery multipliers.

3-input adder/subtracter with 4 limbs. g, s, and $C_1[i]$ represent the partial product from the third multiplier, and the partial sum and coarse-grained carries from the GCCSA, respectively. The absolute value units and the final adder have a similar construction as the ripple-carry adder. A special circuit is used to reduce the four 1-bit carries to a 2-bits sum before it is fed into a k-bit adder.

Although our recursive Karatsuba multiplier provides integer multiplication with low complexity, its input to output latency grows quickly with the number of bits. To solve this problem, we propose a batch-pipelined architecture that can hide the long latency with data level parallelism. A universal datapath is constructed using the Karatsuba multiplier, ripple adder / subtracter and shifter to compute step 1, 2, (3 and 4) of Algorithm 1 in three passes. We fed batch data into the datapath to keep the datapath fully utilized in every clock cycle. The Karatsuba-based Montgomery multiplier has an average throughtput of $f/3$ where f is the operating clock freqnecy.

IV. Result and Comparison

We synthesized our design on a Xilinx Virtex-6 XC6VSX475T-2 FPGA using ISE 11.5. Karatsuba multipliers are constructed by applying Karatsuba layers recursively down to 32-bit multipliers generate by Xilinx coregen. We found that a 32-bit fast carry-chain has a maximum clock frequency of 400 MHz, which is close to the maximum frequency of the 32-bit multipliers. Thus, we select 32-bits as our limb-width in every Karatsuba layer. Figure 4 shows the resource utilization of batch-pipelined Montgomery multipliers with different bit-widths.

We found the designs' clock frequencies are lower than the theoretical 400 MHz limit. A major contributor to this is likely increased routing delay in large designs as the average separation between components increases. We could reduce this routing delay by inserting additional pipelining registers.

We implemented software Montgomery multiplication using Algorithm 1 with the long integer multiplication functions provided by the GMP multiple-precision library [14]. We benchmarked this software implementation on an Intel Xeon E5420 CPU running at 2.5 GHz. The single-threaded performance was multiplied by 4 to estimate the maximum aggregate performance of using all four cores. The CPUs thermal design power (TDP) specification was used to estimate power

TABLE I
COMPARISON OF DIFFERENT OUR MONTGOMERY MULTIPLIERS WITH SOFTWARE IMPLEMENTATION.

bit-width	num of cores	freq (MHz)	FPGA throughput (M MontMul / s)	FPGA power consumption (W)	FPGA energy efficiency (μJ / Mul)	software throughput (M MontMul / s)	software energy efficiency (μJ / Mul)	speedup (times)	energy efficiency gain (times)
128	55	349	6400	80.5	0.0126	33.6	2.38	190	189
256	17	336	1900	67	0.0352	17.9	4.48	106	127
512	4	300	400	51	0.127	6.6	12.12	60.6	95

TABLE II
COMPARISON TO PREVIOUS 512-BIT BLOCK-WISE MONTGOMERY MULTIPLIERS.

Design	normalized LUTs	embedded multiplier	iterations	normalized frequency (MHz)	area$_L$-delay product	area$_M$-delay product
Tang [10]	8235	32	74	500	1219	4.74
Suzuki [9]	3937	17	152	500	1197	5.2
Ours	62557	324	3	300	625	3.24

consumption. We estimate the maximum number of different bit-width multiplier cores that could be mapped to the Virtex-6 and their performance. The power consumption of these FPGA implementations is estimated using the Xilinx power estimator tool with an activity rate of 50 %, a very pessimistic estimate. Table I shows the performance speedup and per-multiplication energy efficiency of our FPGA implementations against the software implementations.

We normalized the performance and resource requirements of previous 512-bit block-wise Montgomery multiplier designs with the architecture proposed in this paper. It is impossible to determine the exact operating frequency of these designs, we optimistically assume they can all operate at 500 MHz when mapped to our target Virtex-6 device. We also normalized the fine-grained logic usage of [10] to the Virtex-6 LUT by assuming that each Virtex-6 LUT can realize one Virtex-2 slice. We calculated both the normalized area$_L$-delay product (i.e. LUTs) and area$_M$-delay product (i.e. embedded multiplier) as shown in Table II. We achieve significantly lower area-delay products thanks to the $O(N^{(\log 3/\log 2)})$ complexity of the Karatsuba algorithm.

One major disadvantage of our Karatsuba-based Montgomery multiplier is the size. Although fully parallelized designs provide high throughput and require little to no control logic, large multipliers may encounter difficulty operating at high clock frequencies due to routing delay. This may be improved in future work by exploring different ways of serializing parts of the architecture.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented a Karatsuba-based Montgomery multiplier for cryptography applications using long integers. The multipliers have significantly lower area-delay products compared with previous designs. They also provide excellent performance and energy efficiency compared with software implementations. Future work includes research on serializing the design so that it can reach parts of the design space using fewer resources.

REFERENCES

[1] P. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[2] C. Kaya Koc, T. Acar, and J. Kaliski, B.S., "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26 –33, jun 1996.

[3] A. L. Masle, W. Luk, J. Eldredge, and K. Carver, "Parametric encryption hardware design," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, vol. 5992. Springer Berlin / Heidelberg, 2010, pp. 68–79.

[4] Y.-Y. Zhang, Z. Li, L. Yang, and S.-W. Zhang, "An efficient CSA architecture for Montgomery modular multiplication," *Microprocessors and Microsystems*, vol. 31, no. 7, pp. 456 – 459, 2007.

[5] C. McIvor, M. McLoone, and J. McCanny, "Modified Montgomery modular multiplication and RSA exponentiation techniques," in *IEE Proceedings of Computers and Digital Techniques*, vol. 151, no. 6, nov. 2004, pp. 402 – 408.

[6] A. Daly and W. Marnane, "Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic," in *Proc. FPGA*. New York, NY, USA: ACM, 2002, pp. 40–49.

[7] D. Narh Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler, "Efficient hardware architectures for modular multiplication on FPGAs," in *Proc. FPL*, aug. 2005, pp. 539 – 542.

[8] S. Ors, L. Batina, B. Preneel, and J. Vandewalle, "Hardware implementation of a Montgomery modular multiplier in a systolic array," in *in Proc. International Parallel and Distributed Processing Symposium*, april 2003, p. 8 pp.

[9] D. Suzuki, "How to maximize the potential of FPGA resources for modular exponentiation," *Cryptographic Hardware and Embedded Systems-CHES 2007*, pp. 272–288, 2007.

[10] S. Tang, K. Tsui, and P. Leong, "Modular exponentiation using parallel multipliers," in *Proc. ICFPT*, dec. 2003, pp. 52 – 59.

[11] N. Jiang and D. Harris, "Quotient Pipelined Very High Radix Scalable Montgomery Multipliers," in *Asilomar Conference on Signals, Systems and Computers*, 29 2006-nov. 1 2006, pp. 1673 –1677.

[12] E. Oksuzoglu and E. Savas, "Parametric, secure and compact implementation of RSA on FPGA," in *International Conference on Reconfigurable Computing and FPGAs*, 2008, pp. 391 –396.

[13] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akad. Nauk SSSR*, vol. 145, no. 293-294, 1962, p. 85.

[14] T. Granlund and the GMP development team, *The GNU Multiple Precision Arithmetic Library, Edition 5.0.1*, Feb 2010.