A Proposal for Modular Multiplication

Ali Şentürk and Mustafa Gök
Department of Computer Engineering
Cukurova University Balcalı, 01330, Turkey

***Abstract*—Modular multiplication is an important operation used in cryptography. This paper presents a novel modular multiplication algorithm that uses mainly multiply-accumulate operation. The proposed algorithm uses a small-size look-up table that can be stored into a general-purpose processor's cache. The algorithm can be used in any microarchitecture that has high-performance multiply-add hardware units in their datapaths.**

***Index Terms*—RSA algorithm, modular multiplication.**

I. Introduction

RSA and eliptic curve cryptography algorithms mostly rely on modular multiplication [1], [2], [3]. RSA algorithm encrypts a plain text T as $T^E mod\ M$ where E is the public key and M is a large modulus. RSA decrypts a cyphertext, C, as $C^D mod\ M$ where D is the private key. Since the exponentiation operation can be performed by executing a series of modular multiplications, the efficient execution of this operation improves the performance of the algorithm. Direct computation of the modular multiplication requires one multiplication and one division operations. To compute modular exponent, most algorithms use the result of the modular multiplication iteratively which causes a data dependency. Because of the data dependency, hardware may waste many cycles if a division instruction is used in the computation. The previous work favored the Montgomery's algorithm [4], since it does not require trial divisions. Over the years it is also accepted that the Montgomery's algorithm is the best fit for hardware implementations and significant research effort is dedicated on efficient hardware implementation of this algorithm [5], [6], [7]. Efficient hardware implementations are important because they decrease the time needed for the decryption of the cipher by brute force. To make brute force attacks difficult, the preferred bit sizes for the modulus, public key, and private key are increased. Recently, applications that use 1024 to 2048 bit sizes have come to scene. On the other hand this trend introduces an important drawback. Public users do not have special hardware for cryptography applications and general purpose processors do not have arithmetic units that can directly operate on large operands. A strategy to process the large operand sizes is dividing the operand into sub-operands equal to the processor's word size and operating on them and combining the sub-results to get the final result. Though this method causes significant performance degradation, it is the only choice if there is no special hardware support.

Modern general purpose processsor architectures have multiple high-performance multiply and multiply-add units in their datapaths due to extensive use of this operation in all kinds of applications. Thus, significant research effort is dedicated to decrease the latency of these instructions. For example, current Intel Core Duo processor family that uses 65nm technology has a 32-bit integer multiply instruction with 3 cycle latency whereas the addition instructions in this processor family have 1 cycle latency [8]. Including popular Montgomery algorithm most modular multiplication algorithms rely on addition instruction. This study presents a modular multiplication algorithm that uses multiplication operation to fill this gap. The rest of the paper is organized as follows: Section 2 presents the proposed algorithms, Section 3 discusses the efficiency of the algorithm. Section 4 presents the conclusion.

II. The Proposed Algorithm

Modern general purpose processsor architectures have multiple high-performance multiply and multiply-add units in their datapaths. The proposed algorithm uses these units to perform modular multiplication for large moduli. The steps of the proposed algorithm are given below.

Algorithm

Assume A and B are two n-bit numbers and M is the moduli.

- Multiply A and B, $P = A \cdot B$. Perform $Q = W(1)$ where Q is used to accumulate modular equivalent of the constant values that are discarded in this step. $W(1)$ is obtained from a look-up table (LUT) as $W(1) = LUT(P_{(n+k-1):n})$. The LUT has 2^k rows where each row contains an n-bit precomputed $P_{(n+k-1):n})\ mod\ M$ value.
- Test if $PL(1) = P_{2n-1:(n+k)}$ is equal to zero, if the test fails, do $(PL1 \cdot W(2)) + P_{(n-1):0}$ where and $W(2) = LUT(2^k)$, otherwise, go to the final step.
- Repeat computation until $PL(i+1) = 0$ when this occurs sum the value of the accumulated Q with $PL(i)_{(n-1):0}$ to get the final result.

The flow chart of the algorithm is given in Figure 1. The algorithm requires pre-computation of moduli values for a selected range of values as explained above. These elements are computed only once for each new moduli. An important challenge is the size of the look-up tables since very large look-up tables create memory bottleneck. However, for practical operand sizes look-up tables can be small enough to fit in L1 caches. In the worst case, the algorithm computes the modular multiplication in $\lceil n/k \rceil$ iterations and requires one multiply-add, one addition and one access to LUT for each iteration. A numerical example that demonstrates the worst case for an 8-bit moduli is given below.

$$A^2 \equiv (7F)^2_{16} \quad mod\ (81)_{16}$$

is computed using the presented algorithm. To demonstrate the algorithm more clearly hexadecimal representation of the

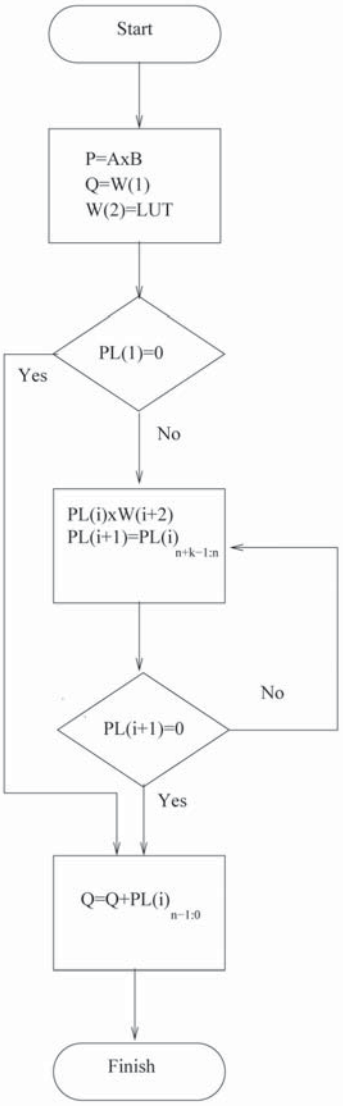

Fig. 1. The Flow Chart.

numbers are used.

Example

- Step 1:
  $P = A^2 = 3F01$
  $PL1 = 0F$
  $Q = W1 = 7B$, $W2 = 79$.
- Step 2:
  $(PL1 \cdot W2) + P_{(n-1):0} = (F \cdot 79) + 01 = 718$
  $W3 = 73$, $W4 = 0$
  $PL2 = 0$
- Step 3:
  $Q \equiv Q + W3 \ mod\ 81 \equiv (7B + 73) \ mod\ 81 \equiv 6D$
  $Q \equiv (Q + PL1_{(n-1):0}) \ mod\ 81$
  $\equiv (6D + 18) \ mod\ 81 \equiv 4$

III. Analysis of The Algorithm

Most modern processor's support 32-bit operand sizes. The proposed algorithm can be adjusted to perform 32-bit modular multiplication. As it is stated before, the number of the required iterations to generate the result mainly depends on the size of the LUT. For 32-bit applications, to keep the look-up table small enough to fit in a processor's cache a look-up table that has 256 rows with 32-bit row size is used. Total size of the LUT is 8KBs which can easily be stored even in an L1 cache of a modern general purpose processor. In this set-up, the proposed algorithm computes the result in $32/8 = 4$ iterations when the worst case occurs. Table 1 presents number of operations required to compute a 32-bit modulus with the proposed method and with Montgomery's method. In both methods, there is a data dependency between each iterations. This means the operations cannot be executed in parallel. An estimate for processors that has 65 nm Intel Core is given based on the worst case latencies for the operations [8]. The latency for a memory access is accepted as one cycle assuming the look-up table is stored in L1 cache. The latency for add and shift operations are one cycle, the latency for multiply instruction is three cycles. There is a packed multiply-add instruction (PMADDWD) among MMX instructions for this family with three cycles latency. Based on these latency values Montgomery algorithm computes the product of 32-bit modular multiplication in 96 cycles whereas the proposed algorithm generates the same result in 20 cycles. On the other hand it should be noted that there is extra overhead for the proposed algorithm due to the computation of the LUT values. A fair concern about the applicability of the algorithm may arise considering the large operand sizes used in practice such as 1024 bits or 2048 bits. Since the available operations are dictated by the processor's word size, the large size operands have to be split into word size suboperands. For example a 128-bit operand must be splitted into four 32-bit operands, to be processed which is true for all algorithms.

TABLE I
A Comparision of Modular Multiplication Methods

Operation	M.A. [4]	P. A.
Add	64	4
Multiply-Add	-	4
Shift	32	-
LUT Access	-	4

IV. Conclusion

This paper presented a straightforward algorithm to compute modular multiplication. The algorithm exploits the current improvements in the performance of multiply-add implementations. Preliminary analysis shows that it has potential to speed-up the computation of modular multiplication in general-purpose architectures. The proposed algorithm can be combined with Montgomery's algorithm in order to increase the instruction level parallelism, since these two algorithms use different resources. Planned future work will search the direct applicabality of the proposed algorithm on hardware and more specifically on FPGA platforms.

REFERENCES

[1] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, p. 126, 1978.

[2] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[3] V. Miller, "Use of elliptic curves in cryptography," in *Advances in CryptologyCRYPTO85 Proceedings*. Springer, 1986, pp. 417–426.

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

[5] S. Eldridge and C. Walter, "Hardware implementation of Montgomery's modular multiplication algorithm," *IEEE Transactions on Computers*, vol. 42, no. 6, pp. 693–699, 1993.

[6] E. Brickell, "A survey of hardware implementations of RSA," in *Advances in CryptologyCRYPTO89 Proceedings*. Springer, 1986, pp. 368–370.

[7] N. Nedjah and L. Mourelle, "Three hardware architectures for the binary modular exponentiation: Sequential, parallel, and systolic," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 3, pp. 627–633, 2006.

[8] "Intel 64 and IA-32 architectures optimization reference manual," Intel Corporation, November 2009.