Montgomery Modular Multiplication on ARM-NEON Revisited

Hwajeong Seo[1], Zhe Liu[2], Johann Großschädl[2], Jongseok Choi[1], and Howon Kim[1]*

[1] Pusan National University,
School of Computer Science and Engineering,
San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,jschoi85,howonkim}@pusan.ac.kr
[2] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue R. Coudenhove-Kalergi, L–1359 Luxembourg-Kirchberg, Luxembourg
{zhe.liu,johann.groszschaedl}@uni.lu

Abstract. Montgomery modular multiplication constitutes the "arithmetic foundation" of modern public-key cryptography with applications ranging from RSA, DSA and Diffie-Hellman over elliptic curve schemes to pairing-based cryptosystems. The increased prevalence of SIMD-type instructions in commodity processors (e.g. Intel SSE, ARM NEON) has initiated a massive body of research on vector-parallel implementations of Montgomery modular multiplication. In this paper, we introduce the Cascade Operand Scanning (COS) method to speed up multi-precision multiplication on SIMD architectures. We developed the COS technique with the goal of reducing Read-After-Write (RAW) dependencies in the propagation of carries, which also reduces the number of pipeline stalls (i.e. bubbles). The COS method operates on 32-bit words in a row-wise fashion (similar to the operand-scanning method) and does not require a "non-canonical" representation of operands with a reduced radix. We show that two COS computations can be "coarsely" integrated into an efficient vectorized variant of Montgomery multiplication, which we call Coarsely Integrated Cascade Operand Scanning (CICOS) method. Due to our sophisticated instruction scheduling, the CICOS method reaches record-setting execution times for Montgomery modular multiplication on ARM-NEON platforms. Detailed benchmarking results obtained on an ARM Cortex-A9 and Cortex-A15 processors show that the proposed CICOS method outperforms Bos et al's implementation from SAC 2013 by up to 57% (A9) and 40% (A15), respectively. Furthermore, our COS multiplication is faster than lastest GMP 6.0.0 by up to 55% (A9) and 52% (A15), respectively.

Keywords: Public-key cryptography, modular arithmetic, SIMD-level parallelism, vector instructions, ARM NEON

1 Introduction

Despite more than three decades of research efforts, public-key cryptography (PKC) is still considered computation-intensive, especially when executed on embedded processors. This is mainly because the underlying arithmetic operations (e.g. exponentiation, scalar multiplication) are performed on operands of a size of several hundreds or even thousands of bits. Multi-precision modular arithmetic is a performance-critical building block of both traditional public-key algorithms (e.g. RSA) and elliptic curve cryptosystems. This is in particular the case for the modular multiplication, which demands careful optimization to achieve acceptable performance, especially on embedded processors. In order to reduce the execution time of modular multiplication, cryptographers have developed several efficient reduction algorithms, while software engineers made efforts to implement them in an optimal way. One of the most

* Corresponding Author

important modular reduction techniques is Montgomery's algorithm, which was originally introduced in 1985 [14] and has been widely deployed in real-world applications. Some other examples for reduction algorithms are the methods of Barrett [2] and Quisquater [16, 17].

In recent years, an increasing number of embedded microprocessors started to provide Single Instruction Multiple Data (SIMD) instructions to better support multimedia workloads. In order to exploit the parallel computing power of SIMD instructions, traditional algorithms need to be redesigned and software needs to be rewritten into a vectorized form. There exist a few papers related to the implementation of cryptographic algorithms; for example, the authors of [3, 8, 18, 7] propose ways to speed up cryptography using the NEON instruction set extensions, which is a relatively new SIMD (i.e. vector) architecture for mobile devices developed by ARM. In particular, to achieve fast public-key cryptography, it is important to develop optimized SIMD implementations of multi-precision modular multiplication. In [4], an efficient 128-by-128-bit integer multiplication using Freescale's SIMD extension is introduced. Various implementations, including [11], adopt a reduced-radix representation with 29 bits per word for a better handling of the carry propagation. In [5], vector instructions on the Cell microprocessor are used to perform multiplication on operands represented with a radix of 2^{16}. More recently, Gueron et al [10] described an implementation for the new AVX2 SIMD platform (Intel Haswell architecture) that uses 256-bit wide vector instructions and a reduced-radix representation for faster accumulation of partial products. At HPEC 2013, a novel modular reduction method was introduced for the NIST primes P192 and P224 [15], which is also based on a reduced-radix representation for the operands.

However, a reduced-radix representation (sometimes also called redundant representation) requires to compute more partial products and, thus, execute more multiply instructions compared to a canonical (i.e. non-redundant) representation. For example, if we use a radix-2^{24} representation (i.e. 24 bits per word) for 192-bit operands, the total number of partial products is $8 \times 8 = 64$. On the other hand, a conventional non-redundant representation based on a radix of 2^{32} reduces the number of partial products to only $6 \times 6 = 36$. At SAC 2013, Bos et al introduced a 2-way Montgomery multiplication for SIMD processors including ARM NEON [6]. Their implementation computes the multiplication and reduction operation simultaneously using a non-redundant representation, which allowed them to exploit the SIMD-level parallelism provided by the NEON engine. However, the performance of their implementation suffers from Read-After-Write (RAW) dependencies in the instruction flow. Such dependencies cause pipeline stalls since the instruction to be executed has to wait until the operands from the source registers are available to be read. For example, the `VMULL` instruction takes two clock cycles to issue the operation, but the result is only available after (at least) seven clock cycles, which means `VMULL` has a fairly long latency[3]. If a data conflict occurs, the pipeline is halted for seven clock cycles rather than just two clock cycles.

In this paper, we describe optimizations to further push the performance of multi-precision multiplication and Montgomery multiplication on ARM-NEON processors. We present a non-redundant Cascade Operand Scanning (COS) method for multiplication, which achieves record-setting execution times on ARM Cortex-A9 and Cortex-A15 processors. The COS method processes the partial products in a non-conventional order to reduce the number of data-dependencies in the carry propagation from less to more significant words, which also reduces the number of pipeline stalls. The same strategy can be applied for a two-way NEON-optimized Montgomery multiplication method, called Coarsely Integrated Cascade Operand Scanning (CICOS) method, which essentially consists of two COS computations, whereby one contributes to the multiplication and the second to the Montgomery reduction. Our experimental results show that a Cortex-A15 processor is able to execute a CICOS Montgomery multiplication with 1024-bit operands in only 5600 clock cycles, which is almost 40% faster

[3] A brief description of some important NEON instructions along with the instruction timings can be found in Appendix D.

than the NEON implementation of Bos et al (8527 cycles according to [6, Table 3][4]). Furthermore, our COS multiplication is faster than lastest GMP 6.0.0 by up to 52% on Cortex-A15 processor.

The remainder of this paper is organized as follows. In Section 2, we recap the previous best results for multiplication and Montgomery multiplication on 32-bit SIMD-based architectures. In Section 3, we present novel methods for multi-precision multiplication and Montgomery multiplication on SIMD-based processors, especially ARM-NEON. Thereafter, we will summarize our experimental results in Section 4. Finally, in Section 5, we conclude the paper.

2 Previous Work

Long integer arithmetic is not straightforward to implement on SIMD-based architectures, mainly due to the propagation of carries from one word to the next, which has to be carried out in addition, multiplication, and other operations. In order to deal with this problem, many recent SIMD implementations adopt a redundant representation with a reduced number of active bits per register with the goal of keeping the final result within remaining capacity of a register so that no carry propagations are needed. In [10], by exploiting the AVX2 instruction set extension with redundant representation, the authors showed a performance enhancement of 51% over the OpenSSL 1.0.1 implementation. In [15], Pabbuleti et al. implemented the NIST-recommended prime-field curve including P192 and P224 on the Snapdragon APQ8060 within 404, 405 clock cycles via applying multiplicand reduction method into SIMD-based machine. Recently, in SAC'13, a different approach to split the Montgomery multiplication into two parts, being computed in parallel, was introduced [6]. They flip the sign of the precomputed Montgomery constant and accumulate the result in two separate intermediate values that are computed concurrently while avoiding a redundant representation. This method is to compute the multiplication and reduction step simultaneously using 2-way SIMD instructions at the cost of some overhead and shows a performance increase of a factor of 1.5 or more than sequential implementation on the Atom platform for 2048-bit modulo. In this paper, we take a different approach computing the multiplication using 2-way SIMD instructions first and subsequently the reduction using 2-way SIMD. The approach uses non-redundant representation and computes the carry propagation using 2-way SIMD instructions

3 Proposed Method

Throughout the paper, we will use the following notations. Let A and B be two operands with a length of m-bit that are represented by multiple-word arrays. Each operand is written as follows: $A = (A[n-1], ..., A[2], A[1], A[0])$ and $B = (B[n-1], ..., B[2], B[1], B[0])$, whereby $n = \lceil m/w \rceil$, and w is the word size. The result of multiplication $C = A \cdot B$ is twice length of A, and represented by $C = (C[2n-1], ..., C[2], C[1], C[0])$. For clarity, we describe the method using a multiplication structure and rhombus form. The multiplication structure describes order of partial products from top to bottom and each point in rhombus form represents a multiplication $A[i] \times B[j]$. The rightmost corner of the rhombus represents the lowest indices $(i,\ j = 0)$, whereas the leftmost represents corner the highest indices $(i,\ j = n-1)$. The lowermost side represents result indices $C[k]$, which ranges from the rightmost corner $(k = 0)$ to the leftmost corner $(k = 2n-1)$. Particularly, SIMD architecture computes two 32-bit partial products with single instruction, so we use two multiplication structures to describe SIMD operations. These block structures placed in the row represent two partial products with single instruction.

[4] Note that the timings in the proceedings version of Bos et al's paper differ from the version in the IACR eprint archive at `https://eprint.iacr.org/2013/519`. We used the faster timings from the eprint version for comparison with our work.

3.1 Cascade Operand Scanning Multiplication for SIMD

SIMD architecture is able to compute multiple data with single instruction. However, SIMD instruction does not provide carry handling registers and therefore, results in imposing huge overheads on SIMD machine to manage carry propagations. In order to alleviate this problem, many of the previous work adopted the so-called redundant representation which absorbs carry propagations into remaining bits in the destination registers, but this architecture also has performance degradations because redundant representation increases number of partial products. In order to address both drawbacks, we choose non-redundant representation, suggesting an efficient carry handling with simple operand realignments.

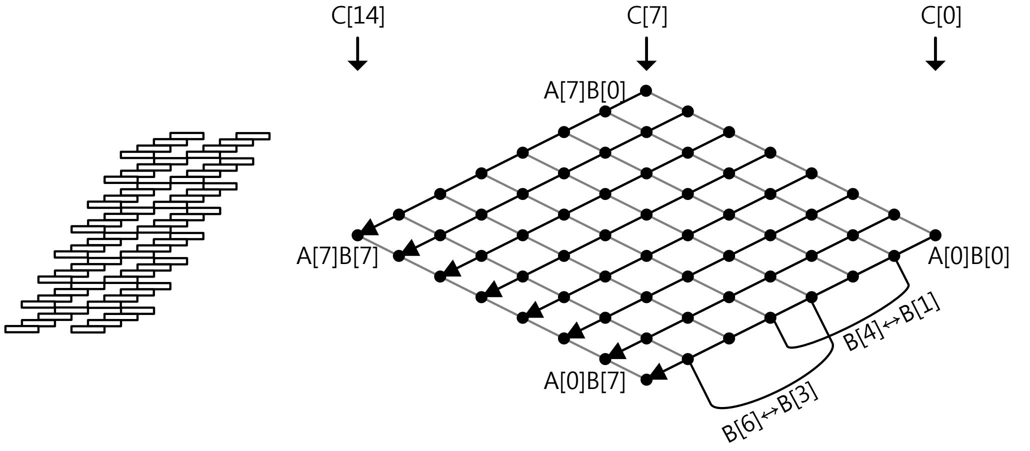

Fig. 1. Cascade operand scanning multiplication for 256-bit operand

In Figure 1, we designed multi-precision multiplication for SIMD architecture. Taking the 32-bit word with 256-bit multiplication as an example, our method works as follows[5]. Firstly, we re-organized operands by conducting transpose operation, which can efficiently shuffle inner vector by 32-bit wise. Instead of a normal order $((B[0], B[1]), (B[2], B[3]), (B[4], B[5]), (B[6], B[7]))$, we actually classify the operand as groups $((B[0], B[4]), (B[2], B[6]), (B[1], B[5]), (B[3], B[7]))$ for computing multiplication where each operand ranges from 0 to $2^{32} - 1$(`0xffff_ffff` in hexadecimal form). Secondly, multiplication $A[0]$ with $((B[0], B[4]), (B[2], B[6]), (B[1], B[5]), (B[3], B[7]))$ is computed, generating the partial product pairs including $((C[0], C[4]), (C[2], C[6]), (C[1], C[5]), (C[3], C[7]))$ where the results are located from 0 to $2^{64} - 2^{33} + 1$(`0xffff_fffe_0000_0001`). Third, partial products are separated into higher bits ($64 \sim 33$) and lower bits ($32 \sim 1$) by using transpose operation with 64-bit initialized registers having 0 value(`0x0000_0000_0000_0000`), which outputs a pair of 32-bit results ranging from 0 to $2^{32} - 1$(`0xffff_ffff`). After then the higher bits are added to lower bits of upper intermediate results. For example, higher bits of $((C[0], C[4]), (C[1], C[5]), (C[2], C[6]), (C[3]))$ are added to lower bits of $((C[1], C[5]), (C[2], C[6]), (C[3], C[7]), (C[4]))$. By referring Figure 2, our method establishes SIMD friendly addition process and the carry values are propagated in grouped cascade way. The proposed method finely rearranges each intermediate result and order of carry propagation to conduct carry handling with only four additions in non-redundant representation and no pipeline stalls. After addition, the least significant word ($C[0]$, lower bits of $B[0] \times A[0]$) is placed within 32-bit in range of [0, `0xffff_ffff`] so this is directly saved into temporal registers or memory. On the other hand, remaining inter-

[5] Operands $A[0 \sim 7]$ and $B[0 \sim 7]$ are stored in 32-bit registers. Intermediate results $C[0 \sim 15]$ are stored in 64-bit registers. The 32-bit register means that we use half capacity of 64-bit register.

mediate results from $C[1]$ to $C[7]$ are placed within [0, `0x1_ffff_fffe`][6], which exceed range of 32-bit in certain cases. Fortunately, the addition of intermediate results ($C[1 \sim 7]$) and 32-bit by 32-bit multiplication in next step are placed into 64-bit registers without overflowing, because addition of maximum multiplication result $2^{64} - 2^{33} + 1$(`0xffff_fffe_0000_0001`) and intermediate result $2^{33} - 2$(`0x1_ffff_fffe`) outputs the final results within 64-bit $2^{64} - 1$(`0xffff_ffff_ffff_ffff`)[7]. For this reason, we don't need to propagate 33th carry bit of intermediate results ($C[1 \sim 7]$) in each round but we delay the carry propagations to very end of round and conduct whole carry propagations at once. Before move to next round, lower bits of $C[4]$ and higher bits of $C[7]$ are re-grouped into $(C[4], C[8])$ and then intermediate result pairs are re-shaped in $((C[1], C[5]), (C[2], C[6]), (C[3], C[7]), (C[4], C[8]))$. This process is repeated with remaining operands ($A[1 \sim 7]$) by seven times more to complete the multiplication[8]. After eight rounds of multiplication, the results from $C[0]$ to $C[7]$ are perfectly fit into 32-bit, because the least significant word is outputted in 32-bit in every round. However remaining intermediate results ($C[8] \sim C[15]$) are not placed within 32-bit so we should process a chain of carry propagation over 32-bit by conducting final alignment. The final alignment executes carry propagation results from $C[8]$ to $C[15]$ with sequential addition and transpose instructions. This process causes pipeline stalls by 8 times, because higher bits of former results are directly added to next intermediate results. Therefore, proposed COS incurs pipeline stalls by the number of n for final alignment. In case of 512-, 1024- and 2048-bit COS multiplications, we should conduct 4, 16 and 64 times of 256-bit COS multiplications because the 256-bit COS multiplication is maximum operand size on NEON processor due to limited number of registers[9]. Unlike 256-bit version, intermediate result should be stored and re-loaded, so we assigned temporal memory and stack storages to retain intermediate results. Finally, several `load`, `store`, `push` and `pop` instructions are used to establish 512-, 1024- and 2048-bit implementations.

3.2 Coarsely Integrated Cascade Operand Scanning Multiplication for SIMD

In [6], Bos et al. introduced a 2-way Montgomery multiplication for SIMD architecture. However, the proposed 2-way Montgomery multiplication has high data interdependency because they used ordinary operand-scanning method for multiplication and reduction procedures which compute partial products in incremental order and previous partial product results are directly used in next step. Finally this resource access conflicts results in pipeline stalls. In order to resolve this problem, we rescheduled the order of operations to achieve a dependency-free design. We implemented the Coarsely Integrated Cascade Operand Scanning (CICOS) algorithm by using COS method since separated version needs to `load` and `store` intermediate results twice more than integrated version, while the finely integrated mode leads to high interdependency between each intermediate result. To describe Montgomery multiplication properly, we grouped two rhombus forms in Figure 3. Upper rhombus represents multi-precision multiplication and under rhombus represents Montgomery reduction. In order to distinguish both computations, we painted structure forms of multiplication process in white and reduction process in yellow.

[6] In the first round, the range is within [0, `0x1_ffff_fffd`], because higher bits and lower bits of intermediate results ($C[0 \sim 7]$) are located in range of [0, `0xffff_fffe`] and [0, `0xffff_ffff`], respectively. From second round, the addition of higher and lower bits are located within [0, `0x1_ffff_fffe`], because both higher and lower bits are located in range of [0, `0xffff_ffff`].

[7] In the first round, intermediate results ($C[0 \sim 7]$) are in range of [0, `0x1_ffff_fffd`] so multiplication and accumulation results are in range of [0, `0xffff_ffff_ffff_fffe`]. From second round, the intermediate results are located in [0, `0x1_ffff_fffe`] so multiplication and accumulation results are in range of [0, `0xffff_ffff_ffff_ffff`].

[8] Pseudo and program code of cascade operand scanning are available in Appendix. A and C, the rhombus form in 512-bit is depicted in Figure 5.

[9] NEON engine supports sixteen 128-bit registers. We assigned four registers for operands (A, B), four for intermediate results (C) and four for temporal storages.

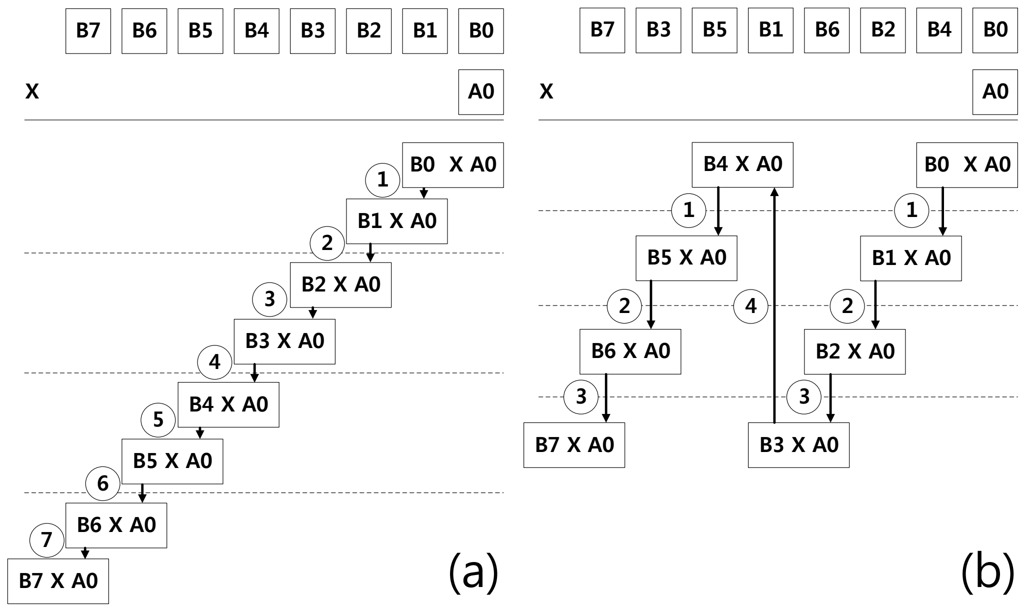

Fig. 2. Carry propagations in non-redundant representation, (a) ordinary operand scanning method, (b) proposed method(operand B is transposed before computations)

Algorithm 1 Calculation of the Montgomery reduction

```
Require: An odd $m$-bit modulus $M$, Montgomery radix $R = 2^m$, an operand $T$ where $T = A \cdot B$ in
    the range $[0, 2M - 1]$, and pre-computed constant $M' = -M^{-1} \bmod R$
Ensure: Montgomery product $Z = \text{MonRed}(T, R) = T \cdot R^{-1} \bmod M$
 1: $Q \leftarrow T \cdot M' \bmod R$
 2: $Z \leftarrow (T + Q \cdot M)/R$
 3: if $Z \geq M$ then $Z \leftarrow Z - M$ end if
 4: return Z
```

In the following, we will take a 256-bit case (M' has a length of 32-bit) as an examples to further explain our implementation of Montgomery reduction for Algorithm 1. As shown in Figure 3, part of the operand Q (i.e. $Q[0 \sim 7]$) are computed by multiplying intermediate results with M'. All partial products with $Q[0]$ and $M[0 \sim 7]$ are executed throughout the row. After then $T + Q \cdot M$ (i.e. Step 2 of Algorithm 1) can be computed using COS method. For Montgomery multiplication on SIMD architecture, we integrated two COS methods[10]. Firstly, we re-organized multiplicands (B) by conducting transpose operation. This process takes $((B[0], B[1]), (B[2], B[3]), (B[4], B[5]), (B[6], B[7]))$ as inputs and generates the new operand groups as $((B[0], B[4]), (B[2], B[6]), (B[1], B[5]), (B[3], B[7]))$ where each operand ranges from 0 to $2^{32} - 1$(`0xffff_ffff`). Secondly, multiply $A[0]$ with $((B[0], B[4]), (B[2], B[6]), (B[1], B[5]), (B[3], B[7]))$, and generating partial product pairs including $((C[0], C[4]), (C[2], C[6]), (C[1], C[5]), (C[3], C[7]))$ where multiplication results range from 0 to $2^{64} - 2^{33} + 1$(`0xffff_fffe_0000_0001`). Third, partial products are separated into higher bits and lower bits by using transpose operation with initialized registers, which outputs results in range of [0, `0xffff_ffff`]. After then the higher bits are added to lower bits of upper intermediate results. For example, higher bits of $(C[0], C[4])$, $(C[2], C[6])$, $(C[1], C[5])$, $(C[3])$ are added to lower bits of $(C[1], C[5])$, $(C[3], C[7])$, $(C[2], C[6])$, $(C[4])$. These interme-

[10] Operands $A[0 \sim 7]$, $B[0 \sim 7]$, $M[0 \sim 7]$, $Q[0 \sim 7]$ and M' are stored in 32-bit registers. Intermediate results $C[0 \sim 15]$ are stored in 64-bit registers.

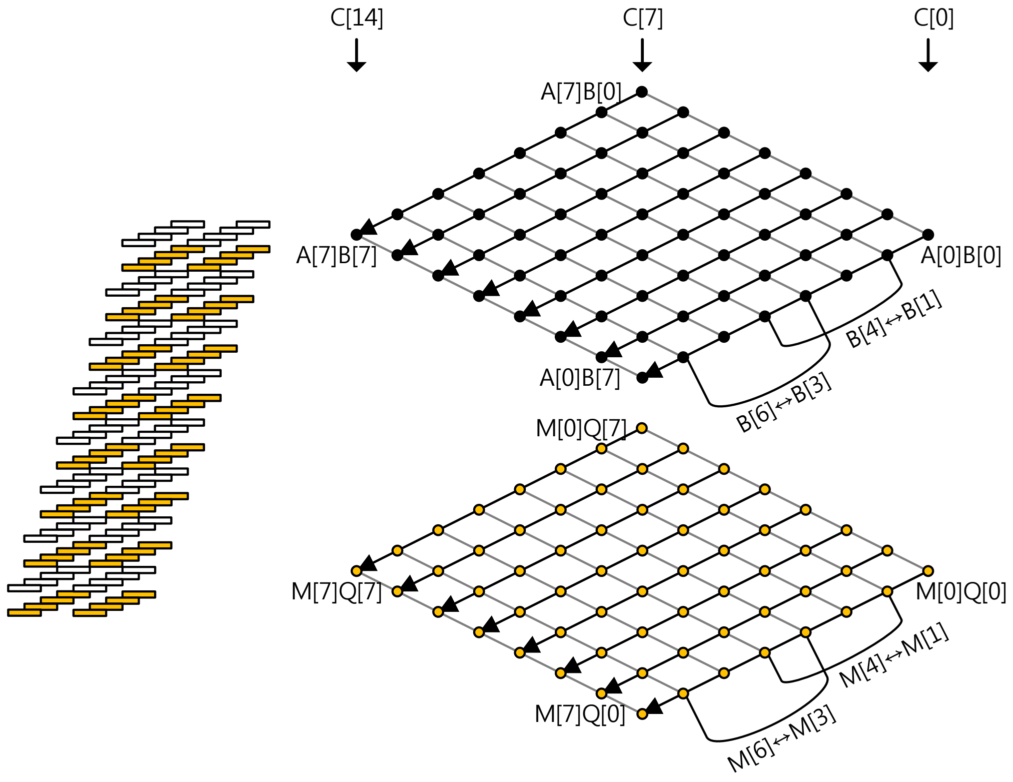

Fig. 3. Coarsely integrated cascade operand scanning for Montgomery multiplication in 256-bit

diate results are placed between 0 and $2^{32}-1$(`0x1_ffff_fffe`)[11]. The remaining higher bits of ($C[7]$) are copied to ($C[8]$). After then lower bits of $C[0]$ in range of [0, `0xffff_ffff`] is multiplied with operand M' to generate $Q[0]$ in range of [0, `0xffff_ffff`]. In case of operand (M), the variables are stored in ordered way like this (($M[0], M[4]$), ($M[2], M[6]$), ($M[1], M[5]$), ($M[3], M[7]$)). Fourthly, multiply $Q[0]$ with (($M[0], M[4]$), ($M[2], M[6]$), ($M[1], M[5]$), ($M[3], M[7]$)) are executed with intermediate results pairs including (($C[0], C[4]$), ($C[2], C[6]$), ($C[1], C[5]$), ($C[3], C[7]$)) where the results range from 0 to $2^{64}-1$(`0xffff_ffff_ffff_ffff`). Lastly, partial products are separated into higher bits and lower bits. After then the higher bits of (($C[0], C[4]$), ($C[1], C[5]$), ($C[2], C[6]$), and the lower bits of ($C[4]$)) are added to lower bits of (($C[1], C[5]$), ($C[2], C[6]$), ($C[3], C[7]$), and the higher bits of ($C[3]$)), which output the accumulated results ranging from 0 to $2^{33}-2$(`0x1_ffff_fffe`). Particularly, higher bits of ($C[7]$) are added to lower bits of ($C[8]$) and then higher bits of ($C[3]$) and lower bits of ($C[8]$) are re-grouped in ($C[4], C[8]$). Finally, the result $C[0]$ is set to zero which is discarded and intermediate results are re-constructed like this (($C[1], C[5]$), ($C[2], C[6]$), ($C[3], C[7]$), ($C[4], C[8]$)). The multiplication and reduction process is repeated with remaining operands ($A[1 \sim 7]$) by seven times more to complete the Montgomery multiplication[12]. After then, final alignment and final subtraction follow[13]. As like COS, maximum operand size of CICOS method is

[11] In the first round, the range is within [0, `0x1_ffff_fffd`], because higher bits and lower bits of intermediate results ($C[0 \sim 7]$) are located in range of [0, `0xffff_fffe`] and [0, `0xffff_ffff`], respectively. From second round, the addition of higher and lower bits are located within [0, `0x1_ffff_fffe`], because both higher and lower bits are located in range of [0, `0xffff_ffff`].

[12] The rhombus form in 512-bit is described in Figure 6

[13] Pseudo and program code of coarsely integrated cascade operand scanning is available in Appendix. B and C

256-bit. For longer integers such as 512-, 1024- and 2048-bit, we should conduct 4, 16 and 64 times of 256-bit CICOS Montgomery multiplication.

3.3 Final Subtraction without Conditional Statements

As shown in Algorithm 1, the calculation of the Montgomery multiplication may require a final subtraction of the modulus M to get a fully reduced result in range of $[0, M)$. In order to get the results, the final subtraction is conducted. The operation is computable with conditional branch by checking the carry bit. However, this method has two drawbacks. First two operands should be compared byte by byte via the compare function and the attacker can catch the leakage information because conditional statements consumes different clock cycles [19]. To resolve this, in [13], author suggested without conditional branch method for Montgomery multiplication. We adopted this method into SIMD architecture. Based on the concept of incomplete modular arithmetic, we don't compare exact value between Z and M, but we use most significant bit (z_m) of Z. If z_m is set, modulus remains. If not, modulus M is set to zero. In order to compute modulus, we perform SIMD multiplication M by z_n, because the M is saved in 32-bit packed way ($M[0], M[4]$) like this (`0x????_????_????_????`). This representation is not favorable to conduct 32-bit wise subtraction because it cannot handle borrow bit in non-redundant subtraction. By conducting multiplication by z_n, each modulus can get masked 64-bit operand with higher 32-bit in zero padding and lower 32-bit in values ($(0, M[0])$, $(0, M[4])$) like this (`0x0000_0000_????_????`). After then we perform the subtraction on intermediate results (Z) by modulus (M). Final result may not be the at least non-negative residue but this is always in the range $[0, 2^m)$. This incomplete reduction does not introduce any problems in practice because incomplete representation can still be used as operand in a subsequent Montgomery multiplication [20]. In the following subsections, we further explore the algorithm in instruction level to analyze the impacts of pipelined operation order.

3.4 Requirements for Pipeline

By referring to Appendix. D. Table 5, NEON engine consists of four timing models. Among them, we should carefully concern on both issue and result clock cycles. Issue clock is minimum clock cycles to trigger instruction. After the issue clock cycle, we can issue next instructions in pipelined way. In a contrast, result clock is time to get final results. If we access to result registers before result clock, we should wait until the data is ready during remaining result clock cycles. For fully pipelined computation, we should issue several instructions and then access to certain results which satisfy the result clock cycle. We analyze proposed method by referring Figure 4.

Part 1 This part starts with operand alignments. Operands B are transposed by conducting the single `VTRN` instruction[14] on eight 32-bit packed variables. The operation needs 1 clock cycle to issue the instruction and 2 clock cycles take to get final results.

Part 2 The second part consists of multiplication and carry propagation. The `VMULL` operation needs 2 clock cycles to issue the instruction and 7 clock cycle for result. By conducting four `VMULL` instructions for $((B[0] \times A[0]), (B[4] \times A[0]))$, $((B[1] \times A[0]), (B[5] \times A[0]))$, $((B[2] \times A[0]), (B[6] \times A[0]))$ and $((B[3] \times A[0]), (B[7] \times A[0]))$, we can get final results within 13 clock cycles, because 8 (2×4) clock cycles for triggering four `VMULL` instructions and 5 additional clock cycles to get last results are needed. Furthermore this computation order satisfies result clock cycle because we conduct four `VMULL` operations taking 8 clock cycles to issue so results of first instruction are ready. In a contrast, if there is interdependency between former destination

[14] Detailed descriptions of instructions are found in Table 5

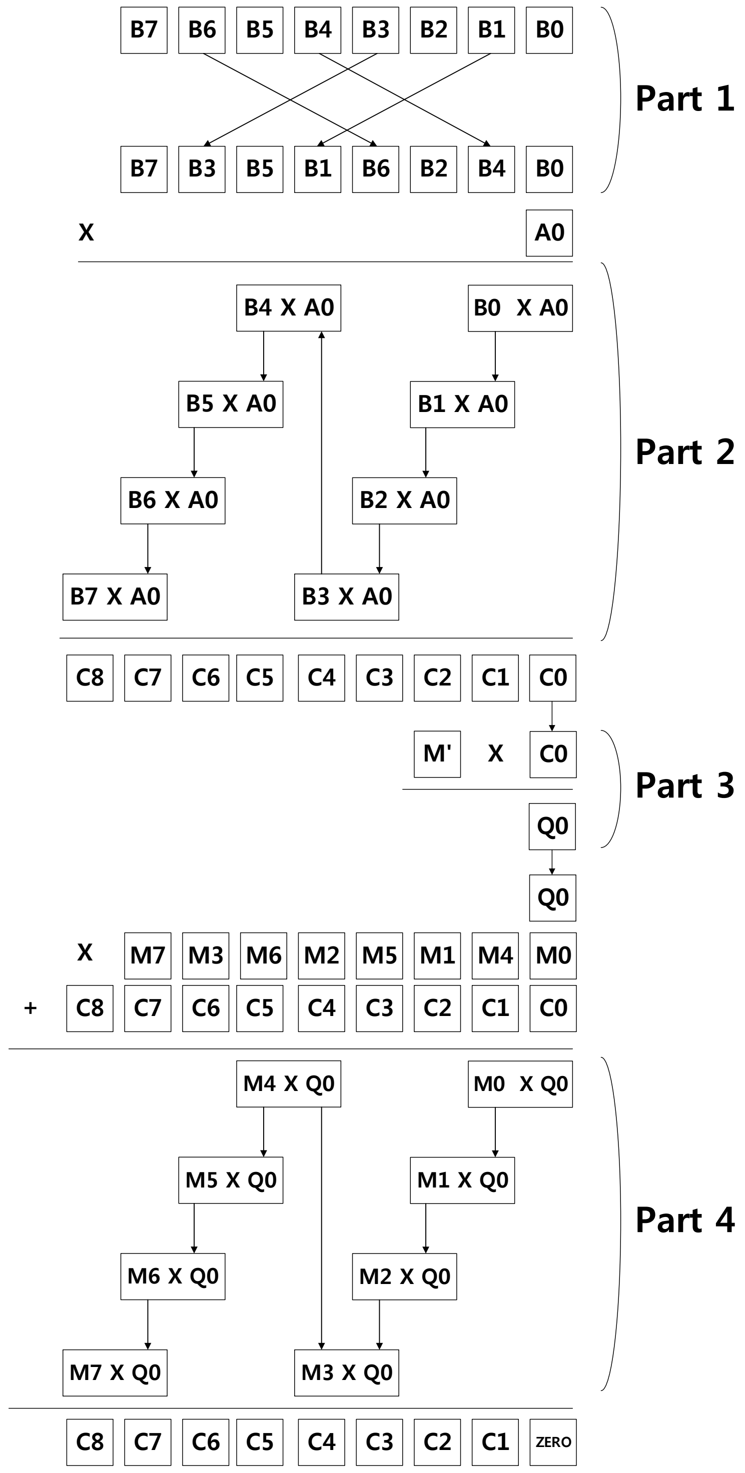

Fig. 4. Structure of coarsely integrated cascade operand scanning Montgomery multiplication

Table 1. Comparison of pipeline stall for Montgomery multiplication

Our CICOS MM		Bos's 2-way MM [6]	
Pipelined	Pipeline Stall	Pipelined	Pipeline Stall
n^2	$2n$	-	$n^2 + n$

and latter source registers, the costs rise to 28 clock cycles, because we should wait for four result clock cycles. After then four `VTRN` instructions for intermediate results on previous partial products are conducted. This takes 5 clock cycles for 4 and 1 clock cycles to trigger the instruction and wait for final results. Lastly, the five carry propagations are conducted with five `VADD` instructions. This takes 5 and 2 clock cycles to issue and get results, respectively. If this is not pipelined, the overheads are 15 clock cycles.

Part 3 In third part, the partial product of M' and $C[0]$ is executed. The result $Q[0]$ is directly used for Montgomery reduction in Part 4. If we conduct this instruction in Part 3, we should wait 7 clock cycles for results to be ready for next step. However, we can avoid the cost by conducting the product during Part 2. After obtaining $C[0]$ and before transpose operation, we conduct `VMULL` on M' and $C[0]$. This only takes 2 clock cycles to issue so we can save 5 clock cycles.

Part 4 The last part calculates the reduction process. In contrast to Part 2, `VMLAL` instruction is exploited to compute multiplication and addition simultaneously. The four partial products of $Q[0]$ and $M[0 \sim 7]$ are conducted, accumulating the intermediate results $C[0 \sim 7]$. The computation takes only 8 and 5 clock cycles to issue and get results. If we conduct the addition and multiplication computations separately in non-pipelined way, the overheads are up-to 40 clock cycles consisting of 12 and 28 for four `VADD` and four `VMULL` instructions, respectively. After then, four `VTRN` and five `VADD` instructions are conducted to manage carry propagation and accumulate intermediate results, which take 5 and 7 clock cycles, respectively.

Final Alignment and Subtraction/Addition The final results from $C[8]$ to $C[15]$ should be aligned to propagate carry bits from least significant word ($C[8]$) to most ($C[15]$). Firstly, higher bits of $C[8]$ are added to $C[9]$ and this is iterated to most significant intermediate result ($C[15]$). Unlike multiplication and reduction process, the final alignment occurs pipeline stalls due to interdependency between former and next variables. After final alignment, final subtraction follows and this also requires to conduct carry propagation incurring pipeline stalls. Finally, our Montgomery method includes the overheads of pipeline stalls by $2n$ times (n for final alignment and n for final subtraction). In case of pipelined instruction, we conduct n^2 times of carry propagations during multiplication and reduction process. On the other hand, Bos's 2-way Montgomery multiplication has n^2+n times of pipeline stalls because former multiplication results are directly used in next operations by following ordinary operand scanning. Particularly, Bos's method incurs pipeline stalls during multiplication and reduction process by n^2 times and final subtraction/addition is required to conduct sequential subtraction by n times. The comparison results are drawn in Table 1.

4 Results

In this section, we report the execution time of the proposed methods on 32-bit ARM Cortex-A9 and Cortex-A15 processors and compare our results with the related works.

4.1 Target Platforms

Cortex Series The ARM Cortex series are full implementation of the ARMv7 architecture including NEON engine. Register size is expanded to double(D) and quadruple(Q) word which represent 64-bit and 128-bit word registers. Each register provides short bit size computations such as 8-bit, 16-bit, 32-bit and 64-bit. This feature provide more precise operation and benefit to various word size computation. Particularly, Cortex-A9 provides the high efficiency, dual-issue super scalar, out-of-order execution and 8-11 stage pipeline. The processor is adopted in several devices including iPad 2, iPhone 4S, Galaxy S2, Galaxy S3, Galaxy Note 2, and Kindle Fire. The Cortex-A15 processor delivers twice the performance of Cortex-A9 processor. The processor provides multi-issue, out-of-order superscalar pipeline with a tightly coupled low latency. The pipeline is supported from 15 to 25 stages. The Cortex-A15 is used in Chromebook, NEXUS 10, Tegra 4 and Odroid-XU.

4.2 Evaluation

We prototyped our methods for ARM Cortex-A9 and A15 processors, which are equivalent to the target processors used in Bos et al's work [6]. We developed the source code using the NDK android library and measured the execution time using the system time function `clock_gettime()` and scaled to clock cycles based on the processor's frequency. We compared our results with best previous results from proceedings version of Bos et al's paper presented at SAC 2013 [6]. In Table 2, we categorize the timings with respect to the architecture that served as experimental platform. In the case of 1024-bit Montgomery multiplication, we achieve an execution time of 8358 clock cycles on the Cortex-A9 series, while Bos et al's SIMD implementation requires 17464 clock cycles. Furthermore, on a Cortex-A15, we compute a 1024-bit Montgomery multiplication within 5600 clock cycles rather than 8527 clock cycles as specified in [6, Table 3]. Thus, our 1024-bit implementation outperforms Bos et al's work by approximately 52% and 34% on a Cortex-A9 and Cortex-A15, respectively. The speed-ups are even more significant for 512-bit operands: 57% on the A9 and 40% on the A15. Our results show that the NEON instructions improve the execution time by 29% and 18% (512 and 1024-bit on Cortex-A9) as well as 37% and 35% (512 and 1024-bit on Cortex-A15) over a sequential implementation that does not take advantage of the NEON engine. The case for 2048-bit also shows, our method improves performance by 48.7% and 21.5% on A9 and A15, respectively. Compared with the sequential implementations, still our 2048-bit implementations have enhancements by 10.9% and 22.7%. The following is reason for the significant speed-up compared to Bos et al's NEON implementation. First, we process the operands in a special order so as to reduce pipeline stalls caused by RAW data dependencies. Second, we perform the carry propagation in an efficient fashion in the NEON engine by adding grouped intermediate results.

Comparison to GMP The GNU multiple precision arithmetic library (GMP) [9] ver 6.0.0 is also tested on same target devices including Cortex-A9 and A15. They used asymptotically fast method like Karatsuba multiplication [12]. The results of long integer multiplications on 512-, 1024- and 2048-bit are obtained within 2346, 6970 and 21386 clock cycles for A9 and 1376, 4224 and 13792 clock cycles for A15. Compared with our COS multiplication, GMP implementations are slower by 20∼55% and 22∼52% for A9 and A15, respectively. Furthermore, unlike our method, the GMP is not a cryptographic library and does not provide constant-time implementations.

5 Conclusions

We presented optimization techniques to improve the performance of multi-precision arithmetic operations (in particular multiplication and Montgomery multiplication) on 2-way

Table 2. Results of multiplication and Montgomery multiplication in clock cycle on ARM Cortex-A9/A15 platforms

Bit	Cortex-A9			Cortex-A15		
	Our NEON	NEON [6]	ARM [6]	Our NEON	NEON [6]	ARM [6]
Multiplication						
256	308	n/a	n/a	219	n/a	n/a
512	1050	n/a	n/a	658	n/a	n/a
1024	4298	n/a	n/a	2810	n/a	n/a
2048	17080	n/a	n/a	10672	n/a	n/a
Montgomery Multiplication						
256	658	n/a	n/a	308	n/a	n/a
512	2254	5236	3175	1485	2473	2373
1024	8358	17464	10167	5600	8527	8681
2048	32732	63900	36746	26232	33441	33961

SIMD platforms. More specifically, we introduced the COS method for multi-precision multiplication, which processes the words of one of the operands in a non-conventional order so as to reduce pipeline stalls caused by data dependencies. Furthermore, we described the CICOS method for performing Montgomery multiplication by coarsely interleaving COS-based multiplication and Montgomery reduction steps. Thanks to these optimizations, we were able to achieve record-setting execution times for conventional multiplication as well as Montgomery multiplication on ARM NEON platforms. For example, on an ARM Cortex-A15 processor, our CICOS method performs a 1024-bit Montgomery multiplication only 5600 clock cycles, which is roughly 34% faster than the NEON implementation of Bos et al (8527 cycles). On a Cortex-A9, the performance gain is even higher, namely 52% (8358 vs. 17464 cycles, i.e. we save 9106 cycles). In case of 2048-bit operands, our methods have improved performance by 21.5% (A15) and 48.7% (A9). These gaps further increase for 512-bit operands to 40% (A15) and 57% (A9). We also outperform Bos et al's non-vectorized implementation (which uses only standard ARM instructions) on the two platforms by between 11% and 37%. Based on these results, we can draw the following conclusion. Our re-ordering of operands along with a "coarse" integration of multiplication and reduction is significantly faster than the conventional operand ordering and "fine" integration approach followed by Bos et al. The interesting future work would be asymptotically faster integer multiplication method like Karatsuba multiplication on SIMD architecture. Karatsuba method replaces multiplication into several additions. However, when it comes to non-redundant representation, addition produces a chains of carry propagations which incur high overheads. For practical purposes, we should study further how to employ an efficient addition operation over non-redundant representation.

References

1. ARM. Cortex-A9 NEON Media Processing Engine Technical Reference Manual Revision: r4p1. Available for download at `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0409i/index.html`, June 2012.
2. P. D. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer Verlag, 1987.
3. D. J. Bernstein and P. Schwabe. Neon crypto. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 320–339. Springer, 2012.
4. Bo Lin. Solving Sequential Problems in Parallel: An SIMD Solution to RSA Cryptography. Available for download at `http://cache.freescale.com/files/32bit/doc/app_note/AN3057.pdf`, Feb. 2006.

5. J. W. Bos and M. E. Kaihara. Montgomery multiplication on the cell. In *Parallel Processing and Applied Mathematics*, pages 477–485. Springer, 2010.
6. J. W. Bos, P. L. Montgomery, D. Shumow, and G. M. Zaverucha. Montgomery multiplication using vector instructions. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography — SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 471–489. Springer Verlag, 2014.
7. D. Câmara, C. P. Gouvêa, J. López, and R. Dahab. Fast software polynomial multiplication on arm processors using the neon engine. In *Security Engineering and Intelligence Informatics*, pages 137–154. Springer, 2013.
8. A. Faz-Hernandez, P. Longa, and A. H. Sanchez. Efficient and secure algorithms for glv-based scalar multiplication and their implementation on glv-gls curves. Technical report, Cryptology ePrint Archive, Report 2013/158, 2013. http://eprint.iacr.org, 2013.
9. Free Software Foundation, Inc. GMP: The GNU Multiple Precision Arithmetic Library. Available for download at `http://www.gmplib.org/`, Aug. 2014.
10. S. Gueron and V. Krasnov. Software implementation of modular exponentiation, using advanced vector instructions architectures. In *Arithmetic of Finite Fields*, pages 119–135. Springer, 2012.
11. Intel Corporation. Using streaming SIMD extensions (SSE2) to perform big multiplications. Application note AP-941, available for download at `http://software.intel.com/sites/default/files/14/4f/24960`, July 2000.
12. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.
13. Z. Liu and J. Großschädl. New speed records for montgomery modular multiplication on 8-bit avr microcontrollers. In *Progress in Cryptology–AFRICACRYPT 2014*, pages 215–234. Springer, 2014.
14. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
15. K. C. Pabbuleti, D. H. Mane, A. Desai, C. Albert, and P. Schaumont. Simd acceleration of modular arithmetic on contemporary embedded platforms. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.
16. J.-J. Quisquater. Procédé de codage selon la méthode dite rsa, par un microcontrôleur et dispositifs utilisant ce procédé. *Demande de brevet français.(Dépôt numéro: 90 02274)*, 122, 1990.
17. J.-J. Quisquater. Encoding system according to the so-called rsa method, by means of a microcontroller and arrangement implementing this system, Nov. 24 1992. US Patent 5,166,978.
18. A. H. Sánchez and F. Rodríguez-Henríquez. Neon implementation of an attribute-based encryption scheme. Technical report, Technical Report CACR 2013-07, available at http://cacr.uwaterloo.ca/techreports/2013/cacr2013-07.pdf, 2013.
19. C. D. Walter and S. Thompson. Distinguishing exponent digits by observing modular subtractions. In *Topics in CryptologyCT-RSA 2001*, pages 192–207. Springer, 2001.
20. T. Yanik, E. Savas, and Ç. Koç. Incomplete reduction in modular arithmetic. In *Computers and Digital Techniques, IEE Proceedings-*, volume 149, pages 46–52. IET, 2002.

Appendix. A. Pseudo Code: COS Multiplication in 256-bit[15]

```
**Input:** word size 32-bit, Operands $A, B \in [0, 256)$.
**Output:** $C = A \cdot B$.
//Register $R$, Memory $M$, Temporal registers $R_{AC}$, $R_{LOW}$, $R_{HIGH} \in [0, 8]$[16].
$R_A[7, ..., 0] \leftarrow M_A[7, ..., 0]$.
$R_B[7, ..., 0] \leftarrow M_B[7, ..., 0]$.
$R_{AC}[8, ..., 0] \leftarrow 0$.
**transpose** $R_B$.
**for** $i = 0$ **to** 7
    $(R_{AC}[4], R_{AC}[0]) \leftarrow (R_{AC}[4], R_{AC}[0]) + (R_A[i], R_A[i]) \times (R_B[4], R_B[0])$.
    $(R_{AC}[5], R_{AC}[1]) \leftarrow (R_{AC}[5], R_{AC}[1]) + (R_A[i], R_A[i]) \times (R_B[5], R_B[1])$.
```

[15] 512-, 1024- and 2048-bit cases conduct 256-bit COS multiplication by 4, 16 and 64 times, respectively. The rhombus form of 512-bit COS is available in Figure 5.

[16] 32-bit register: $R_A[0 \sim 7]$, $R[0 \sim 7]$, 64-bit register: $R_{AC}[0 \sim 8]$, $R_{LOW}[0 \sim 8]$, $R_{HIGH}[0 \sim 8]$

```
    $(R_{AC}[6], R_{AC}[2]) \leftarrow (R_{AC}[6], R_{AC}[2]) + (R_A[i], R_A[i]) \times (R_B[6], R_B[2])$.
    $(R_{AC}[7], R_{AC}[3]) \leftarrow (R_{AC}[7], R_{AC}[3]) + (R_A[i], R_A[i]) \times (R_B[7], R_B[3])$.
    $R_{LOW}[8, ..., 0]$, $R_{HIGH}[8, ..., 0] \leftarrow 0$.
    **transpose** $R_{AC}$ **into** $R_{HIGH}$ **and** $R_{LOW}$.
    $(R_{LOW}[4]) \leftarrow (R_{LOW}[4]) + (R_{HIGH}[3])$.
    $(R_{LOW}[5], R_{LOW}[1]) \leftarrow (R_{LOW}[5], R_{LOW}[1]) + (R_{HIGH}[4], R_{HIGH}[0])$.
    $(R_{LOW}[6], R_{LOW}[2]) \leftarrow (R_{LOW}[6], R_{LOW}[2]) + (R_{HIGH}[5], R_{HIGH}[1])$.
    $(R_{LOW}[7], R_{LOW}[3]) \leftarrow (R_{LOW}[7], R_{LOW}[3]) + (R_{HIGH}[6], R_{HIGH}[2])$.
    $(R_{LOW}[8]) \leftarrow (R_{HIGH}[7])$.
    $M_C[i] \leftarrow R_{LOW}[0]$.
    **for** $j = 0$ **to** $3$
        $(R_{AC}[j+4], R_{AC}[j]) \leftarrow (R_{LOW}[(j+5)], R_{LOW}[(j+1)])$.
    **end for**
**end for**
**final alignment** $R_{AC}$.
$M_C[15, ..., 8] \leftarrow R_{AC}$.
**Return** $M_C$.
```

Appendix. B. Pseudo Code: CICOS Montgomery Multiplication in 256-bit[17]

```
**Input:** word size 32-bit, Operands $A, B$, Modulus $M^* \in [0, 256)$,
Inverse of Modulus $M' \in [0, 32)$.
**Output:** $C = MonRed(A \cdot B, R)$.
//Register $R$, Memory $M$, Temporal registers $R_{AC}$, $R_{LOW}$, $R_{HIGH} \in [0, 8]$[18].
$R_A[7, ..., 0] \leftarrow M_A[7, ..., 0]$.
$R_B[7, ..., 0] \leftarrow M_B[7, ..., 0]$.
$R_{M'} \leftarrow M_{M'}$.
$R_{M^*}[7, ..., 0] \leftarrow M_{M^*}[7, ..., 0]$.
$R_{AC}[8, ..., 0] \leftarrow 0$.
**transpose** $R_B$.
**for** $i = 0$ **to** 7
    $(R_{AC}[4], R_{AC}[0]) \leftarrow (R_{AC}[4], R_{AC}[0]) + (R_A[i], R_A[i]) \times (R_B[4], R_B[0])$.
    $(R_{AC}[5], R_{AC}[1]) \leftarrow (R_{AC}[5], R_{AC}[1]) + (R_A[i], R_A[i]) \times (R_B[5], R_B[1])$.
    $(R_{AC}[6], R_{AC}[2]) \leftarrow (R_{AC}[6], R_{AC}[2]) + (R_A[i], R_A[i]) \times (R_B[6], R_B[2])$.
    $(R_{AC}[7], R_{AC}[3]) \leftarrow (R_{AC}[7], R_{AC}[3]) + (R_A[i], R_A[i]) \times (R_B[7], R_B[3])$.
    $R_{LOW}[8, ..., 0], R_{HIGH}[8, ..., 0] \leftarrow 0$.
    **transpose** $R_{AC}$ **into** $R_{HIGH}$ **and** $R_{LOW}$.
    $(R_{LOW}[4]) \leftarrow (R_{LOW}[4]) + (R_{HIGH}[3])$.
    $(R_{LOW}[5], R_{LOW}[1]) \leftarrow (R_{LOW}[5], R_{LOW}[1]) + (R_{HIGH}[4], R_{HIGH}[0])$.
    $(R_{LOW}[6], R_{LOW}[2]) \leftarrow (R_{LOW}[6], R_{LOW}[2]) + (R_{HIGH}[5], R_{HIGH}[1])$.
    $(R_{LOW}[7], R_{LOW}[3]) \leftarrow (R_{LOW}[7], R_{LOW}[3]) + (R_{HIGH}[6], R_{HIGH}[2])$.
    $(R_{LOW}[8]) \leftarrow (R_{HIGH}[7])$.
    $R_Q[i] \leftarrow R_{LOW}[0] \times R_{M'}$.
    $(R_{AC}[4], R_{AC}[0]) \leftarrow (R_{LOW}[4], R_{LOW}[0]) + (R_Q[i], R_Q[i]) \times (R_{M^*}[4], R_{M^*}[0])$.
    $(R_{AC}[5], R_{AC}[1]) \leftarrow (R_{LOW}[5], R_{LOW}[1]) + (R_Q[i], R_Q[i]) \times (R_{M^*}[5], R_{M^*}[1])$.
    $(R_{AC}[6], R_{AC}[2]) \leftarrow (R_{LOW}[6], R_{LOW}[2]) + (R_Q[i], R_Q[i]) \times (R_{M^*}[6], R_{M^*}[2])$.
    $(R_{AC}[7], R_{AC}[3]) \leftarrow (R_{LOW}[7], R_{LOW}[3]) + (R_Q[i], R_Q[i]) \times (R_{M^*}[7], R_{M^*}[3])$.
    $R_{LOW}[8, ..., 0], R_{HIGH}[8, ..., 0] \leftarrow 0$.
    **transpose** $R_{AC}$ **into** $R_{HIGH}$ **and** $R_{LOW}$.
    $(R_{HIGH}[3]) \leftarrow (R_{HIGH}[3]) + (R_{LOW}[4])$.
    $(R_{LOW}[4]) \leftarrow (R_{HIGH}[3])$.
    $(R_{LOW}[5], R_{LOW}[1]) \leftarrow (R_{LOW}[5], R_{LOW}[1]) + (R_{HIGH}[4], R_{HIGH}[0])$.
    $(R_{LOW}[6], R_{LOW}[2]) \leftarrow (R_{LOW}[6], R_{LOW}[2]) + (R_{HIGH}[5], R_{HIGH}[1])$.
    $(R_{LOW}[7], R_{LOW}[3]) \leftarrow (R_{LOW}[7], R_{LOW}[3]) + (R_{HIGH}[6], R_{HIGH}[2])$.
    $(R_{LOW}[8]) \leftarrow (R_{LOW}[8]) + (R_{HIGH}[7])$.
    **for** $j = 0$ **to** 3
        $(R_{AC}[j + 4], R_{AC}[j]) \leftarrow (R_{LOW}[(j + 5)], R_{LOW}[(j + 1)])$.
    **end for**
**end for**
**final alignment** $R_{AC}$.
**final subtraction** $R_{AC}$.
$M_C \leftarrow R_{AC}$.
**Return** $M_C$.
```

[17] 512-, 1024- and 2048-bit cases conduct 256-bit CICOS Montgomery multiplication by 4, 16 and 64 times, respectively. The rhombus form of 512-bit CICOS is available in Figure 6.

[18] 32-bit register: $R_A[0 \sim 7]$, $R[0 \sim 7]$, $R_M[0 \sim 7]$, $R_Q[0 \sim 7]$, $R_{M'}$, 64-bit register: $R_{AC}[0 \sim 8]$, $R_{LOW}[0 \sim 8]$, $R_{HIGH}[0 \sim 8]$

Appendix. C. Program Code: COS Multiplication and CICOS Montgomery Multiplication in 256-bit

Table 3. Program code: COS multiplication in 256-bit

Operand A registers : q0, q1, Operand B registers : q2, q3

```
vtrn.32 q0, q1

//step 1
vmull.u32 q13, d0, d4[0]
vmull.u32 q12, d2, d4[0]
vmull.u32 q11, d1, d4[0]
vmull.u32 q10, d3, d4[0]

veor q5,q5,q5
veor q6,q6,q6
veor q7,q7,q7
veor q8,q8,q8

vtrn.32 q13, q5
vtrn.32 q12, q6
vtrn.32 q11, q7
vtrn.32 q10, q8

vadd.i64 q12, q12, q5
vadd.i64 q11, q11, q6
vadd.i64 q10, q10, q7
vqadd.u64 d16, d16, d27
vext.8 d28, d28, d26, #4

//step 2
vmlal.u32 q12, d0, d4[1]
vmlal.u32 q11, d2, d4[1]
vmlal.u32 q10, d1, d4[1]
vmlal.u32 q8 , d3, d4[1]

veor q5,q5,q5
veor q6,q6,q6
veor q7,q7,q7
veor q9,q9,q9

vtrn.32 q12, q5
vtrn.32 q11, q6
vtrn.32 q10, q7
vtrn.32 q8 , q9

vadd.i64 q11, q11, q5
vadd.i64 q10, q10, q6
vadd.i64 q8 , q8 , q7
vqadd.u64 d18, d18, d25
vext.8 d28, d28, d24, #4

...//skip step3-7

//step 8
vmlal.u32 q11, d0, d7[1]
vmlal.u32 q10, d2, d7[1]
vmlal.u32 q8 , d1, d7[1]
vmlal.u32 q9, d3, d7[1]

veor q5,q5,q5
veor q6,q6,q6
veor q7,q7,q7
veor q13,q13,q13

vtrn.32 q11, q5
vtrn.32 q10, q6
vtrn.32 q8, q7
vtrn.32 q9, q13

vadd.i64 q10, q10, q5
vadd.i64 q8 , q8 , q6
vadd.i64 q9 , q9 , q7
vqadd.u64 d26, d26, d23
vext.8 d31, d31, d22, #4

//final alignment
veor q5,q5,q5
veor q6,q6,q6
veor q7,q7,q7
veor q0,q0,q0

vtrn.32 d20, d10
vqadd.u64 d16, d16, d10
vext.8 d2, d2, d20, #4

vtrn.32 d16, d11
vqadd.u64 d18, d18, d11
vext.8 d2, d2, d16, #4

vtrn.32 d18, d12
vqadd.u64 d26, d26, d12
vext.8 d3, d3, d18, #4

vtrn.32 d26, d13
vqadd.u64 d21, d21, d13
vext.8 d3, d3, d26, #4

vtrn.32 d21, d14
vqadd.u64 d17, d17, d14
vext.8 d4, d4, d21, #4

vtrn.32 d17, d15
vqadd.u64 d19, d19, d15
vext.8 d4, d4, d17, #4

vtrn.32 d19, d0
vqadd.u64 d27, d27, d0
vext.8 d5, d5, d19, #4
vext.8 d5, d5, d27, #4

//output q14, q15, q1, q2
```

Table 4. Program code: CICOS Montgomery multiplication in 256-bit

Operand A registers : q0, q1, Operand B registers : q2, q3, Inverse of M register : d8
Operand M registers : q14, q15, r0 indicates pointer of value(`0x00000001ffffffff`)

```
vtrn.32 q0, q1

//step 1
vmull.u32 q13, d0, d4[0]
vmull.u32 q12, d2, d4[0]
vmull.u32 q11, d1, d4[0]
vmull.u32 q10, d3, d4[0]

veor q5,q5,q5
veor q6,q6,q6
veor q7,q7,q7
veor q8,q8,q8

vmull.u32 q4, d8, d26

vtrn.32 q13, q5
vtrn.32 q12, q6
vtrn.32 q11, q7
vtrn.32 q10, q8

vadd.i64 q12, q12, q5
vadd.i64 q11, q11, q6
vadd.i64 q10, q10, q7
vqadd.u64 d27, d27, d16

veor q5,q5,q5
veor q6,q6,q6
veor q7,q7,q7
veor q9,q9,q9

vmlal.u32 q13, d28, d8[0]
vmlal.u32 q12, d29, d8[0]
vmlal.u32 q11, d30, d8[0]
vmlal.u32 q10, d31, d8[0]

vtrn.32 q13, q5
vtrn.32 q12, q6
vtrn.32 q11, q7
vtrn.32 q10, q9

vadd.i64 q12, q12, q5
vadd.i64 q11, q11, q6
vadd.i64 q10, q10, q7
vqadd.u64 d18, d18, d27
vqadd.u64 d19, d19, d17

...//skip step2-8

//final alignment
veor q5,q5,q5
veor q6,q6,q6
veor q7,q7,q7
veor q0,q0,q0

vtrn.32 d20, d10
vqadd.u64 d18, d18, d10

vtrn.32 d18, d11
vqadd.u64 d26, d26, d11

vtrn.32 d26, d12
vqadd.u64 d24, d24, d12

vtrn.32 d24, d13
vqadd.u64 d21, d21, d13

vtrn.32 d21, d14
vqadd.u64 d19, d19, d14

vtrn.32 d19, d15
vqadd.u64 d27, d27, d15

vtrn.32 d27, d0
vqadd.u64 d25, d25, d0

vtrn.32 d25, d1

//final subtraction
vmull.u32 q1, d28, d1[0]
vmull.u32 q2, d29, d1[0]
vmull.u32 q3, d30, d1[0]
vmull.u32 q4, d31, d1[0]

vldmia r0!, {q8}

veor q5,q5,q5
veor q6,q6,q6
veor q7,q7,q7
veor q0,q0,q0

vsub.i64 q10, q10, q1
vsub.i64 q9, q9, q2
vsub.i64 q13, q13, q3
vsub.i64 q12, q12, q4

veor d20, d20, d16
vadd.i64 d20, d17, d20
vtrn.32 d20, d10

vqsub.u64 d18, d18, d10
veor d18, d18, d16
vadd.i64 d18, d18, d17
vtrn.32 d18, d11

vqsub.u64 d26, d26, d11
veor d26, d26, d16
vadd.i64 d26, d26, d17
vtrn.32 d26, d12

vqsub.u64 d24, d24, d12
veor d24, d24, d16
vadd.i64 d24, d24, d17
vtrn.32 d24, d13

vqsub.u64 d21, d21, d13
veor d21, d21, d16
vadd.i64 d21, d21, d17
vtrn.32 d21, d14

vqsub.u64 d19, d19, d14
veor d19, d19, d16
vadd.i64 d19, d19, d17
vtrn.32 d19, d15

vqsub.u64 d27, d27, d15
veor d27, d27, d16
vadd.i64 d27, d27, d17
vtrn.32 d27, d0

vqsub.u64 d25, d25, d0
veor d25, d25, d16
vadd.i64 d25, d25, d17

vtrn.32 q10, q9
vtrn.32 q13, q12

//output d20, d26, d21, d27
```

Appendix. D. Instruction Set Summary for NEON

Cycles This is the number of issue cycles the particular instruction consumes, and is the absolute minimum number of cycles per instruction if no operand interlocks are present.

Source The Source field indicates the execution cycle where the source operands must be available if the instruction is to be permitted to issue. The comma separated list matches that of the Format field, indicating the register that each value applies to. Where two lines are provided for a single format containing quad (Q) registers, this indicates that the source registers are handled independently between top and bottom half double (D) registers. The first line provides the information for the lower half of the quad register and the second line provides the same information for the upper half of the quad register.

Result The Result field indicates the execution cycle when the result of the operation is ready. At this point, the result might be ready as source operands for consumption by other instructions using forwarding paths. However, some pairs of instructions might have to wait until the value is written back to the register file.

Writeback The Writeback field indicates the execution cycle that the result is committed to the register file. From this cycle, the result of the instruction is available to all other instructions as a source operand.

Table 5. Instruction set summary for NEON [1]

Mnemonics	`Description`	`Cycles`	`Source`	`Result`	`Writeback`
`VADD`	Vector Addition	1	-,2,2	3	6
`VSUB`	Vector Subtraction	1	-,2,1	3	6
`VEOR`	Vector Exclusive-or	1	-,2,2	3	6
`VMULL`	Vector Multiplication	2	-,2,1	7	7
`VMLAL`	Vector Multiplication with Addition	2	3,2,1	7	7
`VTRN`	Vector Traspose	1	-,1	2	6

Appendix E. 512-bit COS Multiplication and CICOS Montgomery Multiplication in Rhombus Form[19]

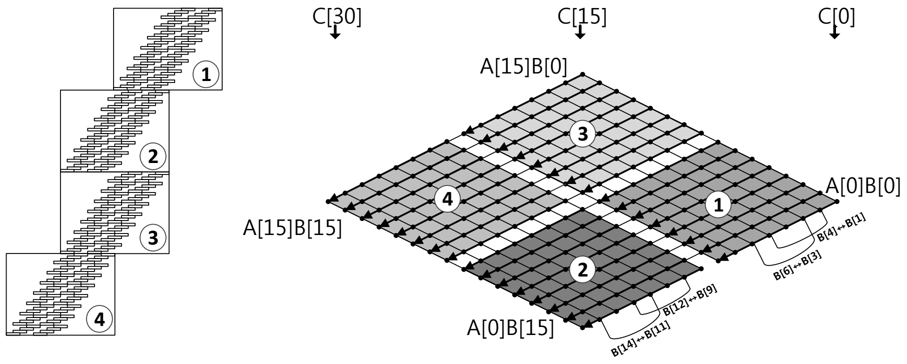

Fig. 5. Cascade operand scanning on NEON in 512-bit

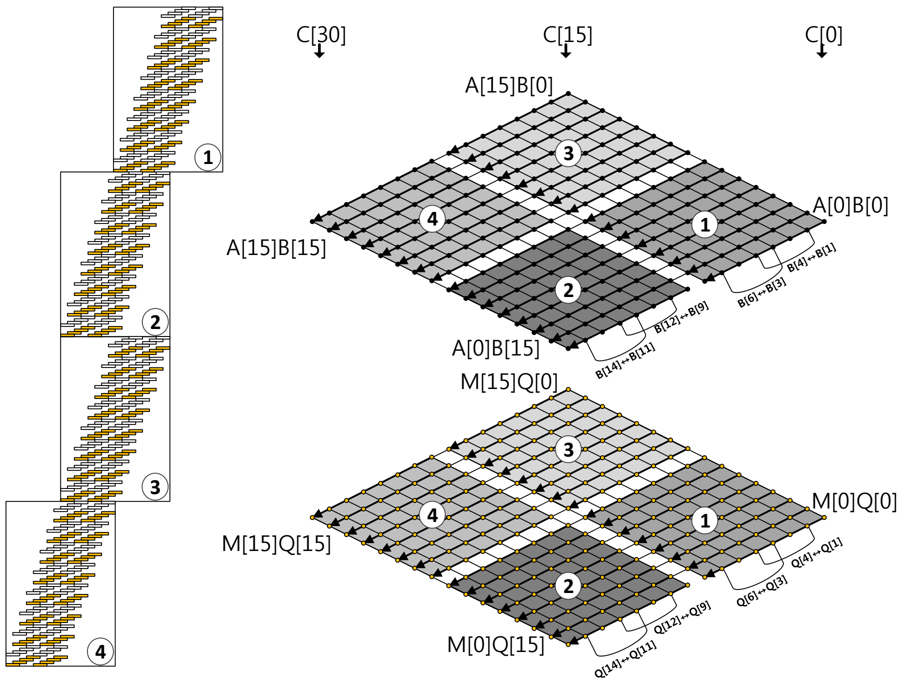

Fig. 6. Coarsely integrated cascade operand scanning on NEON in 512-bit

[19] 1024-and 2048-bit COS and CICOS methods can be established with 4 and 16 times of 512-bit COS and CICOS computations.