# Modular Multiplication and Squaring on ARM-NEON Revisited

2014/10/13

## Outline

## Motivations

- Montgomery et al. [11] (1985) $\rightarrow$ Barrett et al. [2] (1987) $\rightarrow$ Quisquater et al. [13, 14] (1990, 1992)
  Modular operation is a performance-critical operation for public key cryptography.
  Particularly, Montgomery algorithm is widely used in practice.
- Intel inc. [9] (2000) $\rightarrow$ Lin et al. [5] (2006) $\rightarrow$ Bos et al. [6] (2010) $\rightarrow$ Bernstein et al. [4] (2012) $\rightarrow$ Gueron et al. [8] (2012) $\rightarrow$ Pabbeleti et al. [12] (2013) $\rightarrow$ Bos et al. [7] (2013) $\rightarrow$ Bernstein et al. [3] (2014)
  Not much works focus on non-redundant modular multiplication and squaring. Enhance both implementations on SIMD architecture are still an open problem.
  Let us make an effort to solve this problem !
- Montgomery curve, Edwards curve (e.g. Edward25519, Curve25519, Curve41417 [4, 3].. However,
  We still like widely used NIST curves !

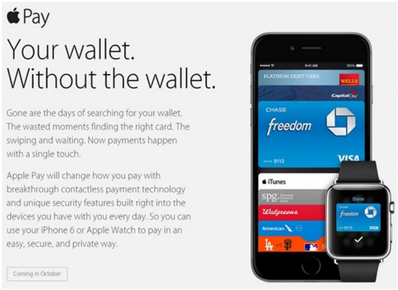
## Contributions

- Novel approach for efficient implementation of multi-precision multiplication/squaring and Montgomery algorithms.
- Fast finite field multiplications for the SGCM, NIST P-192 and P-256 fields.
- Record-setting execution times for scalar multiplication over 192-bit security prime fields.

## Outline

## Outline

## Security in Mobile


Pay

Your wallet.
Without the wallet.

Gone are the days of searching for your wallet. The wasted moments finding the right card. The swiping and waiting. Now payments happen with a single touch.

Apple Pay will change how you pay with breakthrough contactless payment technology and unique security features built right into the devices you have with you every day. So you can use your iPhone 6 or Apple Watch to pay in an easy, secure, and private way.

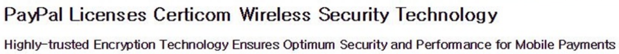
Coming in October

Prime Numbers Hide Your Secrets

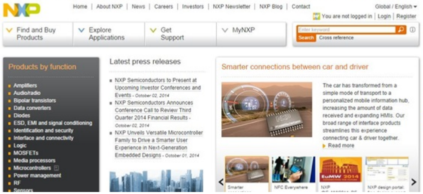
What happens when you enter your credit card number online?

By Edward Frenkel

How exactly are credit card purchases kept secure?


PayPal Licenses Certicom Wireless Security Technology

Highly-trusted Encryption Technology Ensures Optimum Security and Performance for Mobile Payments



Latest press releases

- NXP Semiconductors to Present at Upcoming Investor Conferences and Events - October 02, 2014
- NXP Semiconductors Announces Conference Call to Review Third Quarter 2014 Financial Results - October 02, 2014
- NXP Unveils Versatile Microcontroller Family to Drive a Smarter User Experience in Next-Generation Embedded Designs - October 01, 2014

Smarter connections between car and driver

The car has transformed from a simple mode of transport to a personalized mobile information hub, increasing the amount of data received and expanding HMIs. Our broad range of interface products streamlines this experience connecting car & driver together.

Read more

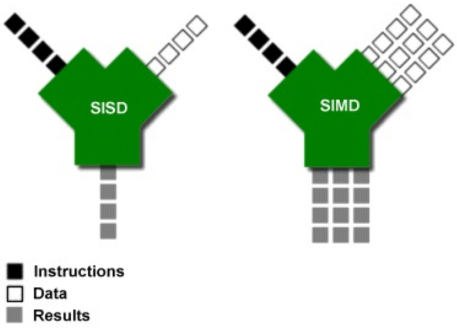
The element has its own embedded operating system, power supply (a small battery) and its own processors and RAM memory. All data storage areas, which is where your credit card and fingerprint information will reside, are protected against both physical and software attacks, and the flow of data goes through a number of interfaces and micro-controllers, protected by various types of end-to-end encryption (Public Key Infrastructure, Dual / Triple key DES-3). Apple Pay takes your banking information and generates an unique Device Account Number, which is encrypted and stored inside the Secure Element. The chip is included in both the iPhone and the Apple Watch.

## Outline

## Parallel Computation with SIMD

SIMD architecture provides further speed-up with parallel computations on modern processors.

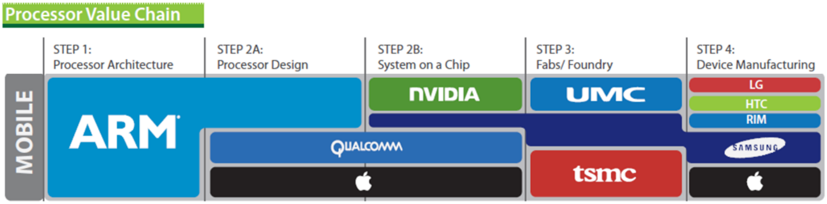
Figure: Comparison between SISD and SIMD



## NEON Engine ∈ Smartphone

Smartphone application processors(AP) designed by ARM(Cortex), Qualcomm(Snapdragon) and Apple(A series). All these processors are based on ARM processor. For this reason, they can take an advantage of ARM-NEON engine.

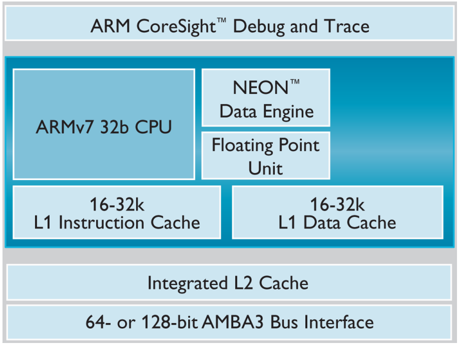
Figure: Mobile Processor Technology


Processor Value Chain
STEP 1: Processor Architecture
STEP 2A: Processor Design
STEP 2B: System on a Chip
STEP 3: Fabs/ Foundry
STEP 4: Device Manufacturing
MOBILE
ARM
NVIDIA
UMC
LG
HTC
RIM
QUALCOMM
SAMSUNG
tsmc

## NEON Engine: Architecture

ARM-NEON engine includes expanded double(D) and quadruple(Q) word size registers which represent 64-bit and 128-bit word registers. Each register provides vector wise computations by 8-bit, 16-bit, 32-bit and 64-bit.

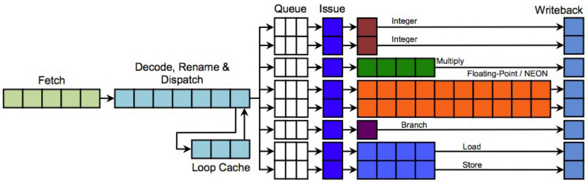
Figure: Architecture of ARM-NEON


ARM CoreSight™ Debug and Trace
ARMv7 32b CPU
NEON™ Data Engine
Floating Point Unit
16-32k L1 Instruction Cache
16-32k L1 Data Cache
Integrated L2 Cache
64- or 128-bit AMBA3 Bus Interface

## NEON Engine: Pipeline and pipeline stall

For high performance instructions should be pipelined. If pipeline stall occurs, performance is significantly degraded.

Figure: Pipeline of ARM-NEON



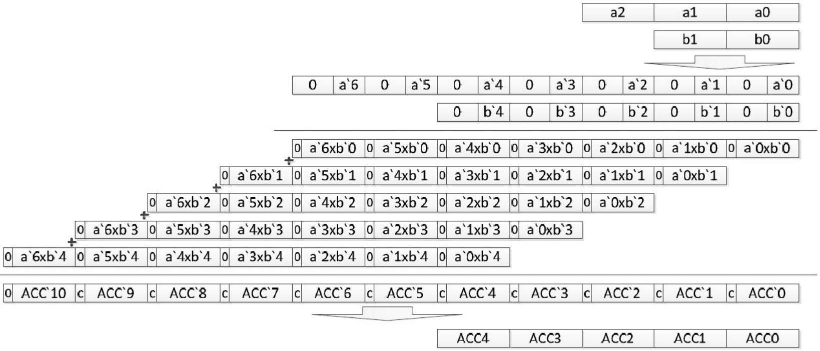
| Mnemonics | Description | Cycles | Source | Result | Writeback |
|---|---|---|---|---|---|
| VADD | Vector Addition | 1 | -,2,2 | 3 | 6 |
| VSUB | Vector Subtraction | 1 | -,2,1 | 3 | 6 |
| VEOR | Vector Exclusive-or | 1 | -,2,2 | 3 | 6 |
| VMULL | Vector Multiplication | 2 | -,2,1 | 7 | 7 |
| VMLAL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

Table: Instruction set summary for NEON [1]

## Outline

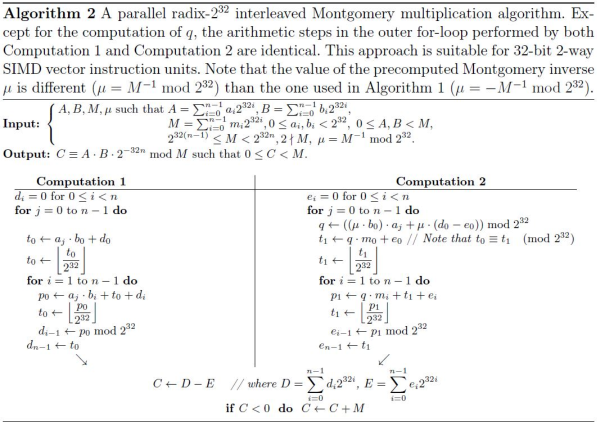
## WAIFI’12 Gueron et al. [8]

Redundant representation(29-radix rather than 32-radix) can reserve the carry bits and prevent continuous carry propagations. However it can impose more number of partial products. (For 192-bit operand, the number of limbs/products are 8(192/24)/64($8 \times 8$) in 24-radix and 6(192/32)/36($6 \times 6$) in 32-radix).


a2 a1 a0
b1 b0
0 a`6 0 a`5 0 a`4 0 a`3 0 a`2 0 a`1 0 a`0
0 b`4 0 b`3 0 b`2 0 b`1 0 b`0
0 a`6xb`0 0 a`5xb`0 0 a`4xb`0 0 a`3xb`0 0 a`2xb`0 0 a`1xb`0 0 a`0xb`0
+
0 a`6xb`1 0 a`5xb`1 0 a`4xb`1 0 a`3xb`1 0 a`2xb`1 0 a`1xb`1 0 a`0xb`1
+
0 a`6xb`2 0 a`5xb`2 0 a`4xb`2 0 a`3xb`2 0 a`2xb`2 0 a`1xb`2 0 a`0xb`2
+
0 a`6xb`3 0 a`5xb`3 0 a`4xb`3 0 a`3xb`3 0 a`2xb`3 0 a`1xb`3 0 a`0xb`3
+
0 a`6xb`4 0 a`5xb`4 0 a`4xb`4 0 a`3xb`4 0 a`2xb`4 0 a`1xb`4 0 a`0xb`4
0 ACC`10 c ACC`9 c ACC`8 c ACC`7 c ACC`6 c ACC`5 c ACC`4 c ACC`3 c ACC`2 c ACC`1 c ACC`0
ACC4 ACC3 ACC2 ACC1 ACC0

## SAC'13 Bos et al. [7]

2-way Montgomery multiplication, one for multiplication and one for reduction, is efficient approach. However, this has interdependency between former results and latter source data, causing pipeline stalls.



**Algorithm 2** A parallel radix-$2^{32}$ interleaved Montgomery multiplication algorithm. Except for the computation of $q$, the arithmetic steps in the outer for-loop performed by both Computation 1 and Computation 2 are identical. This approach is suitable for 32-bit 2-way SIMD vector instruction units. Note that the value of the precomputed Montgomery inverse $\mu$ is different ($\mu = M^{-1} \bmod 2^{32}$) than the one used in Algorithm 1 ($\mu = -M^{-1} \bmod 2^{32}$).

**Input:** $\begin{cases} A, B, M, \mu \text{ such that } A = \sum_{i=0}^{n-1} a_i 2^{32i}, B = \sum_{i=0}^{n-1} b_i 2^{32i}, \\ M = \sum_{i=0}^{n-1} m_i 2^{32i}, 0 \le a_i, b_i < 2^{32},\ 0 \le A, B < M, \\ 2^{32(n-1)} \le M < 2^{32n}, 2 \nmid M,\ \mu = M^{-1} \bmod 2^{32}. \end{cases}$

**Output:** $C \equiv A \cdot B \cdot 2^{-32n} \bmod M$ such that $0 \le C < M$.

**Computation 1**

```
$d_i = 0$ for $0 \le i < n$
for $j = 0$ to $n-1$ do
    $t_0 \leftarrow a_j \cdot b_0 + d_0$
    $t_0 \leftarrow \left\lfloor \frac{t_0}{2^{32}} \right\rfloor$
    for $i = 1$ to $n-1$ do
        $p_0 \leftarrow a_j \cdot b_i + t_0 + d_i$
        $t_0 \leftarrow \left\lfloor \frac{p_0}{2^{32}} \right\rfloor$
        $d_{i-1} \leftarrow p_0 \bmod 2^{32}$
    $d_{n-1} \leftarrow t_0$
```

**Computation 2**

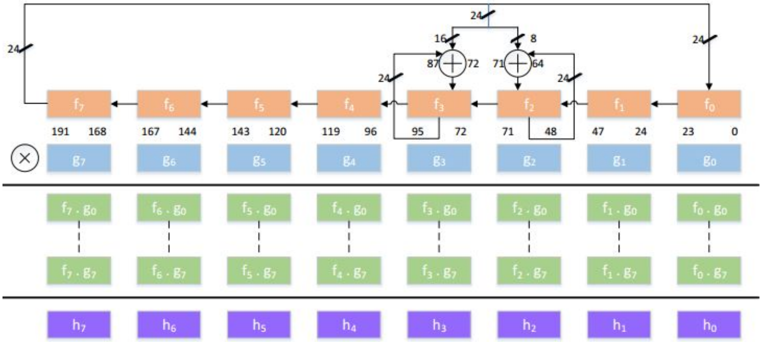
```
$e_i = 0$ for $0 \le i < n$
for $j = 0$ to $n-1$ do
    $q \leftarrow ((\mu \cdot b_0) \cdot a_j + \mu \cdot (d_0 - e_0)) \bmod 2^{32}$
    $t_1 \leftarrow q \cdot m_0 + e_0$ // Note that $t_0 \equiv t_1 \pmod{2^{32}}$
    $t_1 \leftarrow \left\lfloor \frac{t_1}{2^{32}} \right\rfloor$
    for $i = 1$ to $n-1$ do
        $p_1 \leftarrow q \cdot m_i + t_1 + e_i$
        $t_1 \leftarrow \left\lfloor \frac{p_1}{2^{32}} \right\rfloor$
        $e_{i-1} \leftarrow p_1 \bmod 2^{32}$
    $e_{n-1} \leftarrow t_1$
```

```
$C \leftarrow D - E$    // where $D = \sum_{i=0}^{n-1} d_i 2^{32i}$, $E = \sum_{i=0}^{n-1} e_i 2^{32i}$
if $C < 0$ do $C \leftarrow C + M$
```

## HPEC’13 Pabbuleti et al. [12]

Former works conduct reduction on intermediate results. In this work, reduction is directly conducted on operand.

Figure: Reduction on NIST192



## Outline

## Outline

## Cascade Operand Scanning Multiplication

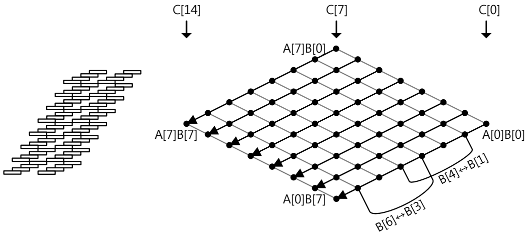


Figure: COS Multiplication in 256-bit case with 32-bit word size

- Unlike previous approach, we conduct 2-way approach for multiplication in non-redundant form. 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 `0xffff _ffff` $(2^{32} - 1)$.

## Cascade Operand Scanning Multiplication

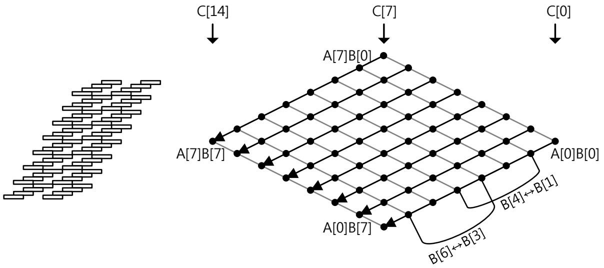


Figure: COS Multiplication

- Multiplication $A[0]$ with $((B[0], B[4]), (B[2], B[6]), (B[1], B[5]), (B[3], B[7]))$ is computed, generating 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 `0xffff _fffe _0000 _0001` $(2^{64} - 2^{33} + 1)$.

## Cascade Operand Scanning Multiplication

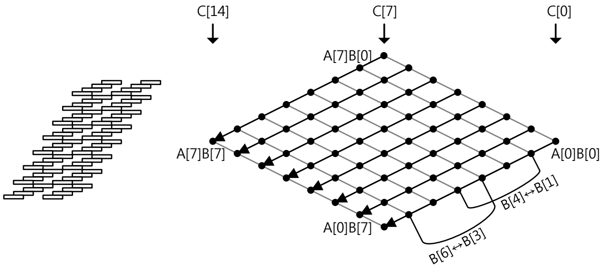


Figure: COS Multiplication

- 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 `0x0000 _0000 _0000 _0000` , which outputs 32-bit results ranging from 0 to `0xffff _ffff` $(2^{32} - 1)$.

## Cascade Operand Scanning Multiplication

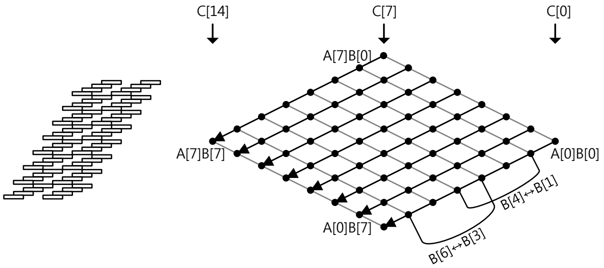


Figure: COS Multiplication

- 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 $((C[1], C[5]), (C[2], C[6]), (C[3], C[7]), (C[4]))$.

## Cascade Operand Scanning Multiplication

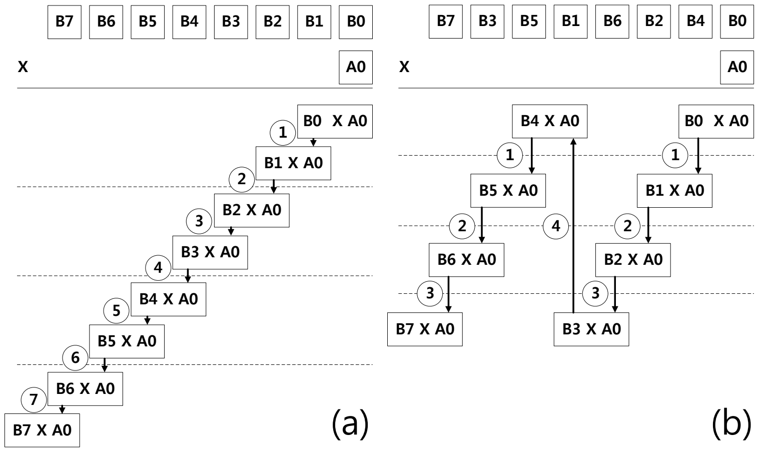


Figure: Left: pipeline stall, Right: pipelined computations

## Outline

## Coarsely Integrated COS Montgomery Multiplication

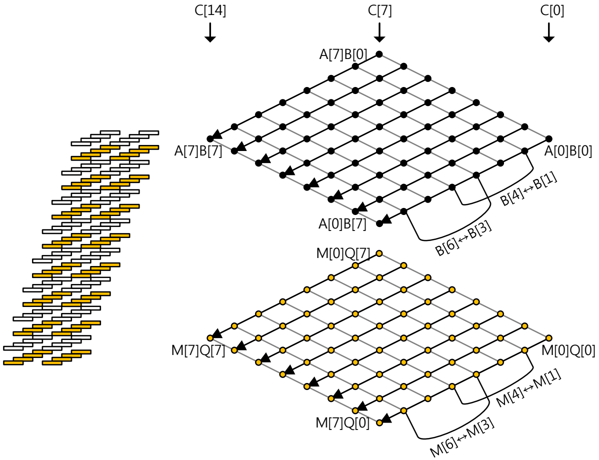


Figure: Coarsely Integrated Cascade Operand Scanning Montgomery Multiplication

## Coarsely Integrated COS Montgomery Multiplication

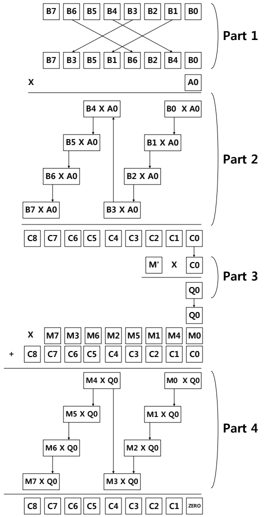


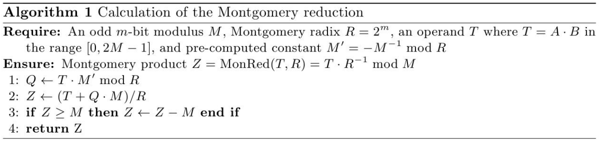
Table: Comparison of pipeline stall for Montgomery multiplication

| Our CICOS MM | | Bos's 2-way MM [7] | |
|---|---|---|---|
| Pipelined | Pipeline Stall | Pipelined | Pipeline Stall |
| $n^2$ | $2n$ | - | $n^2+n$ |

## Coarsely Integrated COS Montgomery Multiplication

Final Subtraction without Conditional Statements

- The calculation of the Montgomery multiplication may require a final subtraction of the modulus $M$ to get a fully reduced result in range of $[0, M)$.
- If most significant bit ($z_m$) is set, modulus remains. If not, modulus $M$ is set to zero. We conduct operand masking method as suggested in [10].



**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 $M' = -M^{-1} \bmod R$

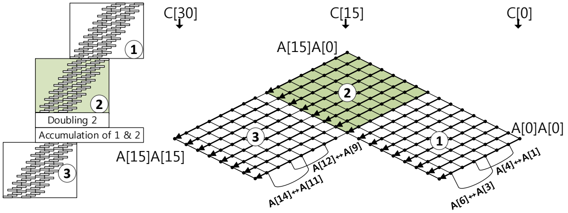
**Ensure:** Montgomery product $Z = \text{MonRed}(T, R) = T \cdot R^{-1} \bmod M$

```
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
```

## Outline

## Traditional Squaring for SIMD

Duplicated partial products $(A[0] \times A[1] = A[1] \times A[0])$



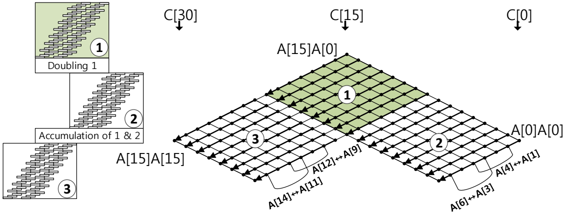
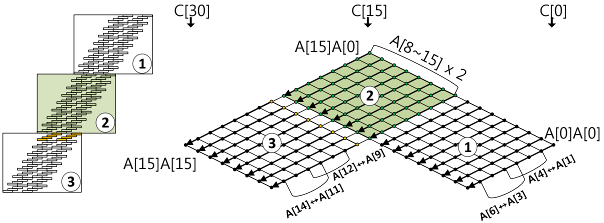


Figure: Upper: Lazy Doubling, Lower: Sliding Block Doubling on SIMD

## Double Operand Scanning Squaring

Double the operand variables($A$) instead of intermediate results. Carry bit($m+1$-th bit for doubled operand $A$) from doubling the operand is handled in constant time computation with masked operand approach.



```
Algorithm 3 Double Operand Scanning squaring
Require: An even $m$-bit operand $A$
Ensure: $2m$-bit result $C$
 1: $(A_{hbit}, A_{DBL}) = A_{[\frac{m}{2},m-1]} \times 2$
 2: $C = A_{[0,\frac{m}{2}-1]} \times A_{[0,\frac{m}{2}-1]}$
 3: $C = C + A_{[0,\frac{m}{2}-1]} \times A_{DBL}$
 4: $C = C + (A_{hbit} \& A_{[0,\frac{m}{2}-1]}) + A_{[\frac{m}{2},m-1]} \times A_{[\frac{m}{2},m-1]}$
 5: return $C$
```

Figure: Upper: Double Operand Scanning Squaring, Lower: Algorithm for DOS

## Comparison between COS and DOS in 1024-bit

Each block represents single 256-bit squaring. Two level DOS for 1024-bit squaring vanishes 6 256-bit wise multiplication blocks rather than COS.

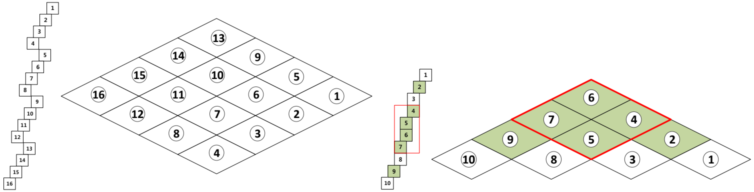
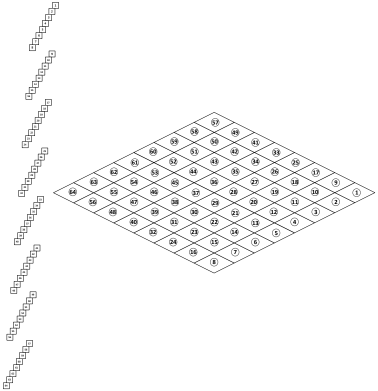


Figure: Left: 1024-bit COS, Right: 1024-bit DOS

## Comparison between COS and DOS in 2048-bit

Three level DOS for 2048-bit squaring vanishes 28 256-bit wise multiplication blocks rather than COS.



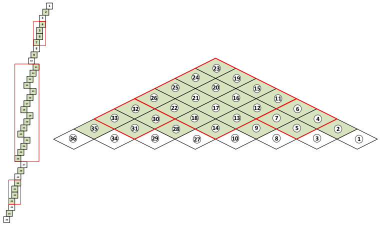


Figure: Left: 2048-bit COS, Right: 2048-bit DOS

## Outline

## SGCM: $P = 2^{128} + 2^{13} + 2^{12} + 2^{7} + 2^{5} + 2 + 1$

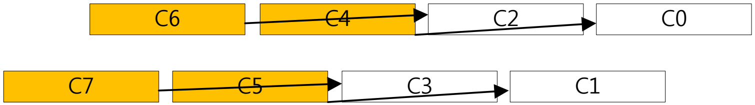


Figure: Fast reduction for SGCM

- Outputs ranging from 129-bit to 256-bit ($C4 \sim C7$) are multiplied by 12451 and then the results are subtracted to ($C0 \sim C3$).
- After then carry propagations from 1-bit to 128-bit ($C0 \sim C3$) are conducted. If there are carry/borrow bit (129-th) set, additional reduction are conducted once again.

## secp192r1: $P = 2^{192} - 2^{64} - 1$

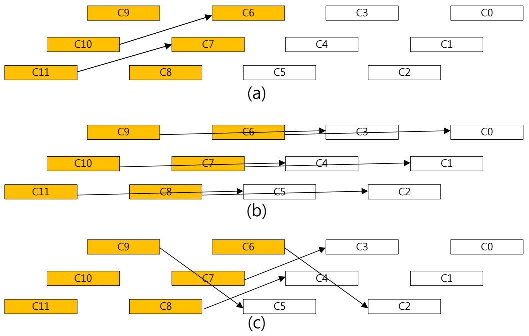


Figure: Fast reduction for secp192r1

- (a) intermediate results from 321-bit to 384-bit are added to $193 \sim 256$-bit. The reason to conduct $193 \sim 256$-bit first is to accumulate same bit position and compute at once.

## secp192r1: $P = 2^{192} - 2^{64} - 1$

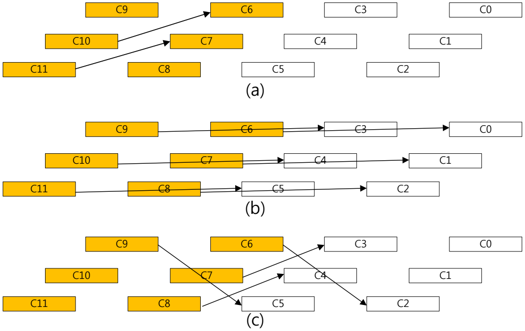


Figure: Fast reduction for secp192r1

- (b) the results from 193-bit to 384-bit are added to $1 \sim 192$-bit.

## secp192r1: $P = 2^{192} - 2^{64} - 1$

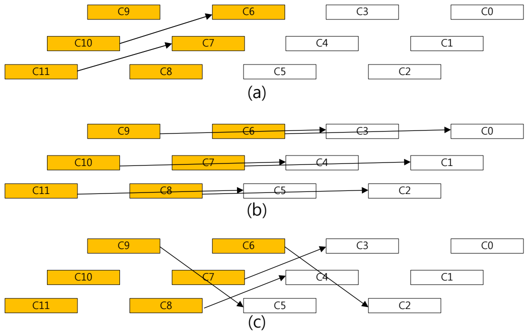


Figure: Fast reduction for secp192r1

- (c) results from 193-bit to 320-bit are added to $65 \sim 192$-bit. If there are carry bits set, additional reduction is conducted.

## secp256k1: $P = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

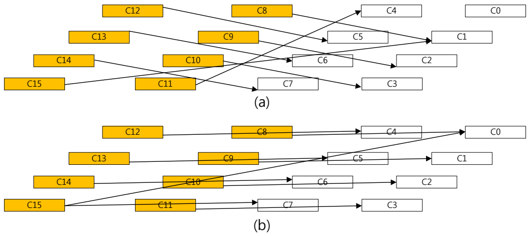


Figure: Fast reduction for secp256k1

- (a) we firstly conduct reduction on $2^{32}$ so intermediate results from 257-bit to 480-bit are added to $33 \sim 256$-bit. The range of $[481, 512]$-bit should be added to $[257, 288]$-bit and this is again added to $33 \sim 64$-bit. We directly add the results to $33 \sim 64$-bit.

## secp256k1: $P = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

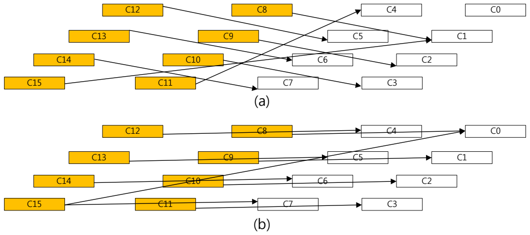


Figure: Fast reduction for secp256k1

- (b) After then intermediate results from 257 to 512-bit are multiplied by 977($2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 1$) and then added to $1 \sim 256$-bit. As a similar approach of previous step, the range of $[481, 512]$-bit is directly multiplied and added to bits from 1 to 32-bit. If there is carry bit set, additional reduction is conducted more.

## Outline

## Evaluation of Modular Multiplication/Squaring

- CICOS Montgomery multiplication outperforms Bos et al(SAC'13) by $48\% \sim 57\%$(A9) and $21\% \sim 35\%$(A15). DOS squaring is faster than our COS multiplication by $23.8\%$(A9) $\sim$ $19.1\%$(A15).
- COS multiplication is faster than latest GMP 6.0.0 by $55\%$(A9) and $52\%$(A15).

Table: Results of multiplication, squaring and Montgomery multiplication in clock cycle

| Bit | Cortex-A9 | | | Cortex-A15 | | |
|---|---|---|---|---|---|---|
| | Our NEON | NEON [7] | ARM [7] | Our NEON | NEON [7] | ARM [7] |
| Cascade Operand Scanning Multiplication | | | | | | |
| 256 | 308 | n/a | n/a | 219 | n/a | n/a |
| 512 | 1050 | n/a | n/a | 643 | 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 |
| Karatsuba Multiplication($t = 1/t = 0$) | | | | | | |
| 512 | 1026/1144 | n/a | n/a | 625/697 | n/a | n/a |
| Double Operand Scanning Squaring | | | | | | |
| 512 | 800 | n/a | n/a | 520 | n/a | n/a |
| 1024 | 3500 | n/a | n/a | 2275 | 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 |

## Evaluation of Elliptic Curve Cryptography

- We achieved 43% enhancements in finite field multiplication of secp192r1 than HPEC'13. The performance enhancement impacts directly on scalar multiplication operations by 16.5%.
- Since SGCM128 has two's complement conversion overheads, it shows similar performance with secp192r1 consisting of additions for reduction.

Table: Results of field/scalar multiplication on Qualcomm Snapdragon

| Bit | APQ8060 | | |
|---|---|---|---|
| | Our NEON | NEON [12] | ARM [12] |
| Prime Field Multiplication with Fast Reduction | | | |
| SGCM128 | 228 | n/a | n/a |
| secp192r1 | 228 | 404 | 574 |
| secp256k1 | 402 | n/a | n/a |
| Scalar Multiplication | | | |
| secp192r1 | 1,035K | 1,243K | 1,591K |

## Outline

## Conclusions and Future Work

Conclusions:

- Introduced new techniques for multi-precision multiplication/squaring and Montgomery multiplication, only 32732 and 26232 clock cycles are required for Montgomery multiplication at the length of 2048-bit, which produces the fastest implementation published for NEON platform.
- New speed records for scalar multiplication over NIST P-192 security prime fields.

Recent News:

- We extended COS/DOS methods to redundant representation(26-radix).

Future work:

- OPF-Montgomery Multiplication/Squaring.
- Twisted Edwards curve and Montgomery curve on P-192 and P-256.

## Thanks and Questions

Thanks for your attention!

Questions?

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.

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.

D. J. Bernstein, C. Chuengsatiansup, and T. Lange.
Curve41417: Karatsuba revisited.
In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 316–334. Springer, 2014.

D. J. Bernstein and P. Schwabe.
Neon crypto.

In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 320–339. Springer, 2012.

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.

J. W. Bos and M. E. Kaihara.
Montgomery multiplication on the cell.
In *Parallel Processing and Applied Mathematics*, pages 477–485. Springer, 2010.

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.

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.

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.

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.

P. L. Montgomery.
Modular multiplication without trial division.
*Mathematics of Computation*, 44(170):519–521, Apr. 1985.

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.

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.

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.