Multiprecision Multiplication on ARMv8

ZHE LIU\textsuperscript{1}, KIMMO JÄRVINENÄDL\textsuperscript{2}, WEIQIANG LIU\textsuperscript{3}, HWAJEONG SEO\textsuperscript{4}

\textsuperscript{1} APSIA, INTERDISCIPLINARY CENTRE FOR SECURITY, RELIABILITY AND TRUST (SNT), UNIVERSITY OF LUXEMBOURG, LUXEMBOURG
\textsuperscript{2} DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF HELSINKI, HELSINKI, FINLAND
\textsuperscript{3} COLLEGE OF ELECTRONIC AND INFORMATION ENGINEERING, NANJING UNIVERSITY OF AERONAUTICS AND ASTRONAUTICS
\textsuperscript{4} DEPARTMENT OF IT, HANSUNG UNIVERSITY
Motivation

• Cryptography degrades the performance of smartphone

• In particular, public key cryptography imposes high overheads

• Fast PKC implementation is important to achieve high availability
Motivation

• Multi-precision arithmetic operation (for PKC)
  • Compact big number implementation is an open problem

• Few works focus on ARMv8
  • GCM (CT-RSA’15) $\rightarrow$ Binary field multiplication (SCN’16) $\rightarrow$ Binary ECC (SG-CRC’17)
  • This work improves the performance of multiplication on ARMv8!
Contribution

• Compact implementations of multi-precision multiplication
  • Subtractive Karatsuba algorithm
  • Evaluation of multiple-level Karatsuba
  • Test input size (128, 256, 384, 512-bit)
  • Squaring dedicated routine
Target Platform – ARMv8

- 95% of smartphones based on ARM architecture
- Modern smartphone supports 64-bit ARMv8
Target Platform – ARMv8

- 32-bit mode (AArch32) & 64-bit mode (AArch64)
- 64-bit ARM & 128-bit NEON registers and instruction sets
- Crypto (AES and SHA) operation
Multiplication on ARMv8

\[
\begin{align*}
X0 & \quad a0 \\
X1 & \quad b0 \\
a1b0 \\
x3 \\
\end{align*}
\]

\[
\begin{align*}
X2 & \quad 64 \text{ bits} \\
\end{align*}
\]

\[
\begin{align*}
X0 & \quad a0 \\
X1 & \quad b0 \\
a0b0 \\
x3 \\
\end{align*}
\]

\[
\begin{align*}
X2 & \quad 64 \text{ bits} \\
\end{align*}
\]

\[
\begin{align*}
X0 & \quad a0 \\
X1 & \quad b0 \\
a0b0 \\
x3 \\
\end{align*}
\]

\[
\begin{align*}
X2 & \quad 64 \text{ bits} \\
\end{align*}
\]

MUL

UMULH
For 64-bit multiplication on ARMv8, NEON requires 4 UMULL routines but A64 only needs 1 MUL and 1 UMULH. A64 is more efficient than NEON for big integer multiplication.
Multi-precision Multiplication

256~2048-bit multiplication on 64-bit architecture

- divide big integer (256~2048-bit) into small integer (64-bit)

<table>
<thead>
<tr>
<th>Method</th>
<th>Operand-scanning</th>
<th>Product-scanning</th>
<th>Hybrid-scanning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation order</td>
<td>Row-wise</td>
<td>Column-wise</td>
<td>Mixture of row/column</td>
</tr>
<tr>
<td>Requirement</td>
<td>Many registers</td>
<td>Efficient MAC routine</td>
<td>General processor</td>
</tr>
</tbody>
</table>

- ARMv8 supports 31x64-bit registers $\rightarrow$ operand-scanning (previous works)
Multi-precision Multiplication

Operand-scanning method
Multi-precision Squaring

A special case of multiplication where both operands are the same (i.e., A = B)

Certain partial products become the same and need to be performed only once (i.e., $A[0] \times B[1] + A[1] \times B[0]$ becomes $2 \times A[0] \times A[1]$ if $A = B$)

Two approaches:
- Doubling the operand (i.e., $2 \times A[0] \rightarrow 2 \times A[0] \times A[1]$)
- Doubling the result (i.e., $A[0] \times A[1] \rightarrow 2 \times A[0] \times A[1]$) $\rightarrow$ Sliding-block-doubling
Multi-precision Squaring

Sliding-block-doubling method
Karatsuba-Ofman Algorithm

Number of partial product

<table>
<thead>
<tr>
<th>School-book</th>
<th>Karatsuba-Ofman</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N^2$</td>
<td>$N^{\log_2 3}$</td>
</tr>
</tbody>
</table>

The product $C = A \cdot B$ of two $n$-bit integers $A = A_L + A_H 2^{n/2}$ and $B = B_L + B_H 2^{n/2}$

$$C = A_H \cdot B_H 2^n + ((A_L + A_H) \cdot (B_L + B_H) - A_L \cdot B_L - A_H \cdot B_H) 2^{n/2} + A_L \cdot B_L$$
Subtractive Karatsuba Algorithm

\[ C = A_H \cdot B_H 2^n + \left( (A_L + A_H) \cdot (B_L + B_H) - A_L \cdot B_L - A_H \cdot B_H \right) 2^{n/2} + A_L \cdot B_L \]

\[ (A_L + A_H) \cdot (B_L + B_H) - A_L \cdot B_L - A_H \cdot B_H = A_L \cdot B_L + A_H \cdot B_H - |A_H - A_L| \cdot |B_H - B_L| \]

Advantage:
- constant size of operands \((n/2)\) → fast constant-time multiplication

Requirement:
- Absolute value in two’s complement representation
Multi-precision Multiplication on ARMv8

128-bit Karatsuba multiplication

```
1: LDP x4, x5, [x2]  // loading
2: LDP x2, x3, [x1]
3: MOV x1, #0
4: MUL x6, x2, x4  // A_L \cdot B_L low
5: UMULH x7, x2, x4 \{A_L \cdot B_L high\}
6: MUL x8, x3, x5  // A_H \cdot B_L low
7: UMULH x9, x3, x5 \{A_H \cdot B_H high\}
8: ADDS x10, x6, x8
9: ADCS x11, x7, x9
10: ADCS x12, x1, x1
11: ADDS x7, x7, x10
12: ADCS x8, x8, x11
13: ADCS x9, x9, x12
14: SUBS x2, x2, x3 \{absolute values\}
15: SBXCS x3, x3, x3
16: EOR x2, x2, x3
17: AND x3, x3, #1
18: ADD x2, x2, x3
19: SUBS x4, x4, x5
20: SBCS x5, x5, x5
21: EOR x4, x4, x5
22: AND x5, x5, #1
23: ADD x4, x4, x5
24: EOR x3, x3, x5
25: SUB x3, x3, #1 \{combining the signs\}
26: MUL x10, x2, x4 \{A_D \cdot B_D low\}
27: UMULH x11, x2, x4 \{A_D \cdot B_D high\}
28: EOR x10, x10, x3
29: EOR x11, x11, x3
30: AND x4, x3, #1
31: ADDS x10, x10, x4
32: ADCS x11, x11, x1
33: ADCS x3, x3, x1
34: ADDS x7, x7, x10
35: ADCS x8, x8, x11
36: ADCS x9, x9, x3 \{storing\}
37: STP x6, x7, [x0, #0]
38: STP x8, x9, [x0, #16]
```
Multi-precision Squaring on ARMv8

No need for absolute value handling $|A_H - A_L| \cdot |A_H - A_L| \rightarrow$ always positive value

```
1: ldp x3, x4, [x1]
2: mov x5, #0
3: mul x6, x3, x3
4: umulh x7, x3, x3
5: mul x8, x4, x4
6: umulh x9, x4, x4
7: subs x3, x3, x4
8: sbcs x4, x4, x4
9: eor x3, x3, x4
10: and x4, x4, #1
11: adds x3, x3, x4
12: mul x10, x3, x3
13: umulh x11, x3, x3
14: adds x12, x6, x8
15: adcs x13, x7, x9
16: adcs x9, x9, x5
17: adds x7, x7, x12
18: adcs x8, x8, x13
19: adcs x9, x9, x5
20: subs x7, x7, x10
21: sbcs x8, x8, x11
22: sbcs x9, x9, x5
23: stp x6, x7, [x0, #0]
24: stp x8, x9, [x0, #16]
```
Optimizations of instruction set

Generation of the carry register
- MOV X0, #0 → ... → ADDS X1, X1, X2 → ADCS X3, X0, X0

Two’s complement
- SBCS X2, X2, X2 → EOR X0, X0, X2 → AND X2, X2, #1 → ADD X0, X0, X2
Evaluation

IDE: Xcode

Target:
- 64-bit ARMv8-A architecture
- Apple A7 (APL0698) @1.3GHz

Program language: assembly

Optimization level: -Ofast
## Evaluation

### Multiplication

<table>
<thead>
<tr>
<th>Method</th>
<th>Input size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128</td>
</tr>
<tr>
<td><strong>Operand-scanning</strong></td>
<td></td>
</tr>
<tr>
<td>cycle</td>
<td>19</td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td><strong>This paper</strong></td>
<td></td>
</tr>
<tr>
<td>cycle</td>
<td>24</td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>Karatsuba</td>
<td>1-level</td>
</tr>
</tbody>
</table>

### Squaring

<table>
<thead>
<tr>
<th>Method</th>
<th>Input size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128</td>
</tr>
<tr>
<td><strong>Sliding Block Doubling</strong></td>
<td></td>
</tr>
<tr>
<td>cycle</td>
<td>17</td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td><strong>This paper</strong></td>
<td></td>
</tr>
<tr>
<td>cycle</td>
<td>20</td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>Karatsuba</td>
<td>1-level</td>
</tr>
<tr>
<td>Operation</td>
<td>ADD/ADC</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>Previous operand-scanning multiplication</strong></td>
<td></td>
</tr>
<tr>
<td>128-bit</td>
<td>7</td>
</tr>
<tr>
<td>256-bit</td>
<td>32</td>
</tr>
<tr>
<td>384-bit</td>
<td>77</td>
</tr>
<tr>
<td>512-bit</td>
<td>158</td>
</tr>
<tr>
<td><strong>Proposed Karatsuba multiplication</strong></td>
<td></td>
</tr>
<tr>
<td>128-bit</td>
<td>14</td>
</tr>
<tr>
<td>256-bit</td>
<td>45</td>
</tr>
<tr>
<td>384-bit</td>
<td>84</td>
</tr>
<tr>
<td>512-bit</td>
<td>182</td>
</tr>
<tr>
<td><strong>Previous sliding block doubling squaring</strong></td>
<td></td>
</tr>
<tr>
<td>128-bit</td>
<td>6</td>
</tr>
<tr>
<td>256-bit</td>
<td>27</td>
</tr>
<tr>
<td>384-bit</td>
<td>56</td>
</tr>
<tr>
<td>512-bit</td>
<td>94</td>
</tr>
<tr>
<td><strong>Proposed Karatsuba squaring</strong></td>
<td></td>
</tr>
<tr>
<td>128-bit</td>
<td>7</td>
</tr>
<tr>
<td>256-bit</td>
<td>32</td>
</tr>
<tr>
<td>384-bit</td>
<td>60</td>
</tr>
<tr>
<td>512-bit</td>
<td>124</td>
</tr>
</tbody>
</table>
Conclusion

Achievements

- Efficient implementations of multi-precision multiplication / squaring on ARMv8

Future works

- Cryptography implementations (ECC, RSA, SIDH)