White Paper

Erdinc Ozturk
James Guilford
Vinodh Gopal

IA Architects
Intel Corporation

Large Integer Squaring on Intel® Architecture Processors

January 2013

328569-001

Executive Summary

New instructions, mulx, adcx and adox, are being introduced on Intel® Architecture Processors. The adcx and adox instructions are being introduced one generation later than mulx.

These new instructions will enable users to develop high-performance implementations of large integer arithmetic on Intel® Architecture.

New instructions are being introduced on Intel® Architecture Processors to enable fast implementations of large integer arithmetic. Large Integer Arithmetic is widely used in multi-precision libraries for high-performance technical computing usages, as well as for public key cryptography (for example, RSA). In this paper, we describe some critical operations required in large integer arithmetic which can be efficiently implemented using the new instructions.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. www.intel.com/embedded/edc.

Contents

Overview

Large Integer Arithmetic refers to performing arithmetic operations on integers (typically unsigned) that are larger than the native word size of the processor. Typically, these integers are much larger than the maximum 64-bit words supported by most general purpose processors. In many applications, the large integers used may be 512-bit, 1024-bit, or larger.

One place that large integers are used is the RSA public key algorithm, which needs to perform a modular exponentiation of at least 512-bit operands. For example, a RSA private key operation using a 2048-bit key requires modular exponentiation of 1024-bit integers, assuming the use of the Chinese Remainder Algorithm.

Large integer arithmetic is also used for Elliptic Curve Cryptography (ECC) and Diffie-Hellman (DH) Key Exchange. Beyond cryptography, there are many use cases in complex research and high performance computing (HPC). The demand for this functionality is high enough to warrant a number of commonly used libraries, such as the GNU Multi-Precision (GMP) library.

Large integer multiplication and squaring are the most interesting arithmetic operations to consider in this context. Many algorithms, such as modular exponentiation and modular reduction, are based on these operations. In a previous paper [1], we described efficient methods to perform multiplication. In the current paper, we describe an efficient method to perform large integer squaring.

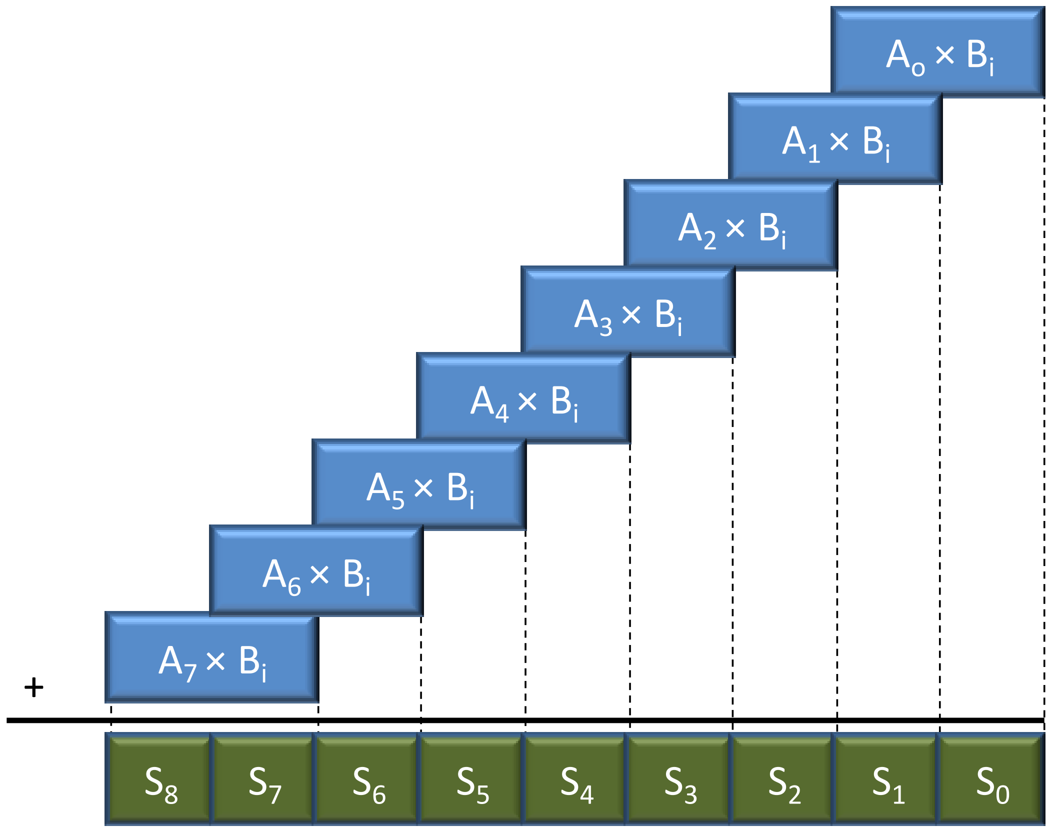
Large Integer Multiplication Basics

The school book method is a common way of doing large integer multiplication. Let the sizes of the two inputs be N and M words, where in this context, "word" means the size of the largest words that can be multiplied by the underlying hardware. In the case of IA, one "word" would be 64-bits. All N*M pairs of words are multiplied, and the results are summed with the appropriate weights.

Efficiency of the school book method depends on choosing the order in which the multiplications and subsequent additions are done. Two common orderings for school book multiplication are column-wise and row-wise (also called by-diagonals). In the column-wise approach, all of the products that are associated with a particular result position are computed and summed in a single pass. In general, this corresponds to computing one "column" in a single pass, where the column consists of the products $A_i \times B_{N-i}$ for all appropriate i.

The row-wise approach computes all of the products formed by one word of one source and all of the words of the other source, i.e. $A_i \times B_j$ for all appropriate *i* and some fixed *j*. This is illustrated in Figure 1.

Figure 1:: Row-wise (Diagonal) ordering for multiplication



Note that each product element of the "row" is two words wide, so it overlaps the adjacent elements by one word. To make it easier to visualize, we stagger the elements of the row, turning them into a "diagonal".

In general, each word in the product needs to be added to two different words. The high-half of one product term needs to be added to the low-half of the adjacent term, then this sum needs to be added to a partial sum that accumulates the results from different diagonals. The key point is there are two carry chains that need to be propagated.

Each multiplication only needs to have two associated additions. However, there is typically only a single carry flag in a general purpose processor. Therefore, the two carry-chains need to be done sequentially, or there needs to be a mechanism to support two independent carry chains. In [1], we focused on efficient implementations of the school book method of multiplication performed in a row-wise manner.

New Instruction Definitions

Three new instructions [2] are being introduced on Intel® Architecture Processors to enable fast implementations of large integer arithmetic.

MULX Instruction

The mulx instruction is an extension of the existing mul instruction, with the difference being in the effect on flags:

```
mulx dest_hi, dest_lo, src1
```

The instruction also uses an implicit src2 register, edx or rdx depending on whether the 32-bit or 64-bit version is being used.

The operation is:

```
dest_hi:dest_lo = src1 * r/edx
```

The reg/mem source operand `src1` is multiplied by rdx/edx, and the result is stored in the two destination registers `dest_hi:dest_lo`. No flags are modified.

This provides two key advantages over the existing mul instruction:

- There is greater flexibility in register usage, as current mul destination registers are implicitly defined. With mulx, the destination registers may be distinct from the source operands, so that the source operands are not over-written.
- Since no flags are modified, mulx instructions can be mixed with add-carry instructions without corrupting the carry chain.

ADCX/ADOX Instructions

The adcx and adox instructions are extensions of the adc instruction, designed to support two separate carry chains. They are defined as:

```
adcx dest/src1, src2
adox dest/src1, src2
```

Both instructions compute the sum of `src1` and `src2` plus a carry-in and generate an output sum `dest` and a carry. The difference between these two instructions is that adcx uses the CF flag for the carry in and carry out (leaving the OF flag unchanged), whereas the adox instruction uses the OF flag for the carry in and carry out (leaving the CF flag unchanged).

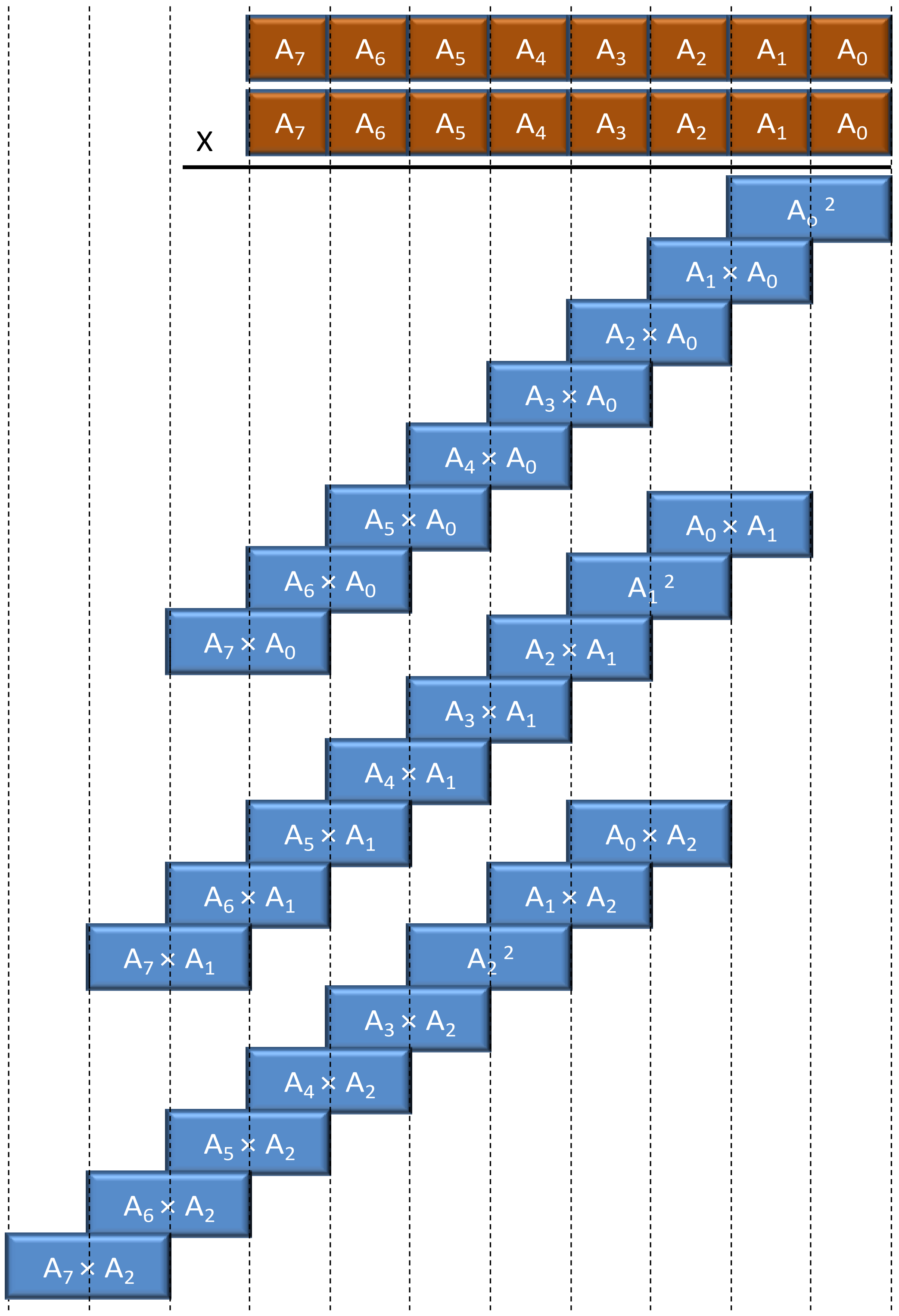
The primary advantage of these instructions over adc is that they support two independent carry chains.

Note that the two carry chains can be initialized by an instruction that clears both the CF and OF flags, for example “xor reg,reg”.

Large Integer Squaring

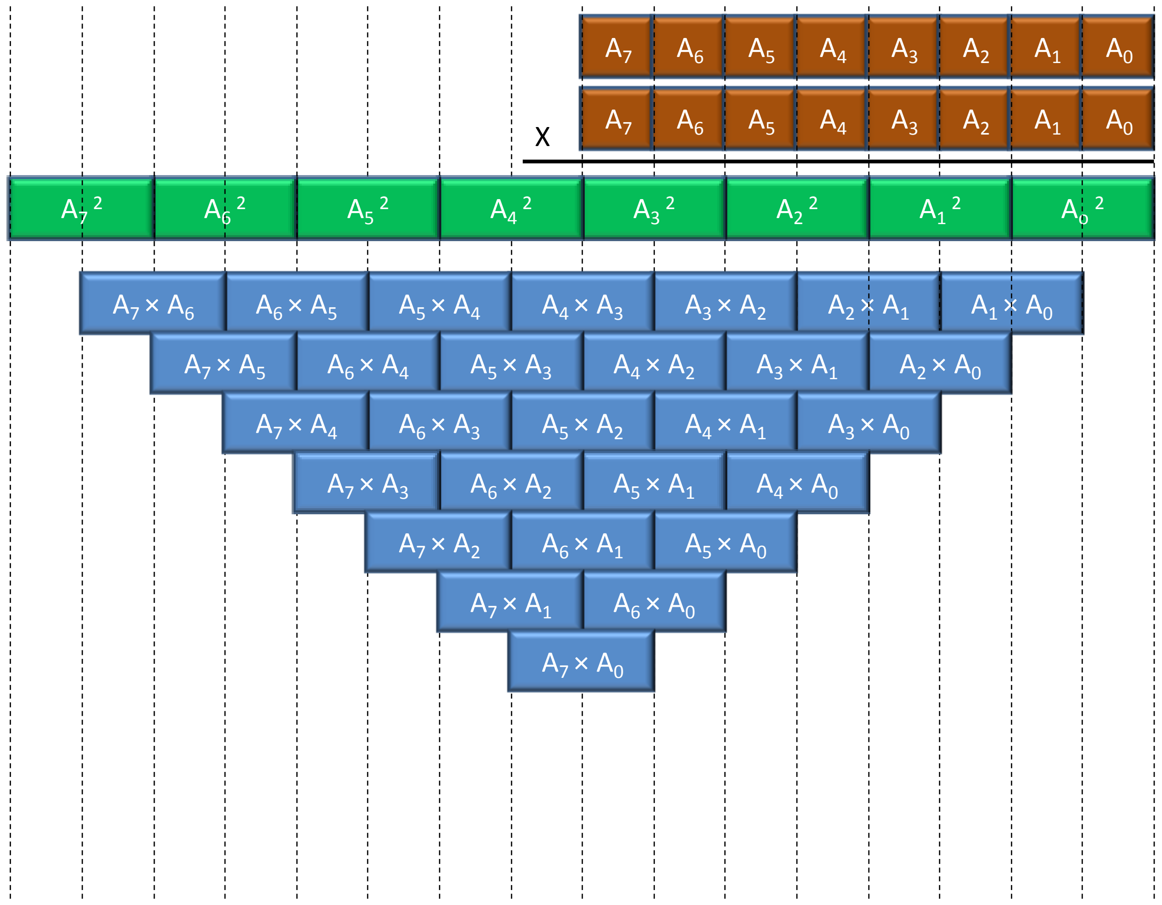
For many compute-intensive operations using large operands, the squaring is a crucial problem to solve efficiently. In this section, the row-wise approach will be examined, as it has proven to be the most efficient method for many cryptographic and HPC applications. Although squaring can be done using multiplication with identical operand inputs, we use a special squaring implementation which reduces the number of base multiplications significantly. Similar methods have been proposed in [4]. This will be illustrated with a 512-bit squaring example. Figure 2 shows the multiplication algorithm with identical input operands. Note that A_i is 64 bits since the core multiplier of the CPU is a 64-bit multiplier, and we show the first 3 diagonals of a 512-bit multiplication.

Figure 2: Multiplication algorithm with identical input operands



Note that the terms $A_i \times A_j$ for all appropriate i != j appear twice, and a naïve multiplication algorithm will multiply these terms twice. Instead we can optimize the number of multiplications of these terms. If we write down all the equations for the diagonals and group the identical weight products, we get a picture as shown in Figure 3 for 512-bit squaring.

Figure 3: Squaring algorithm. The bottom (blue) part needs to be multiplied by 2

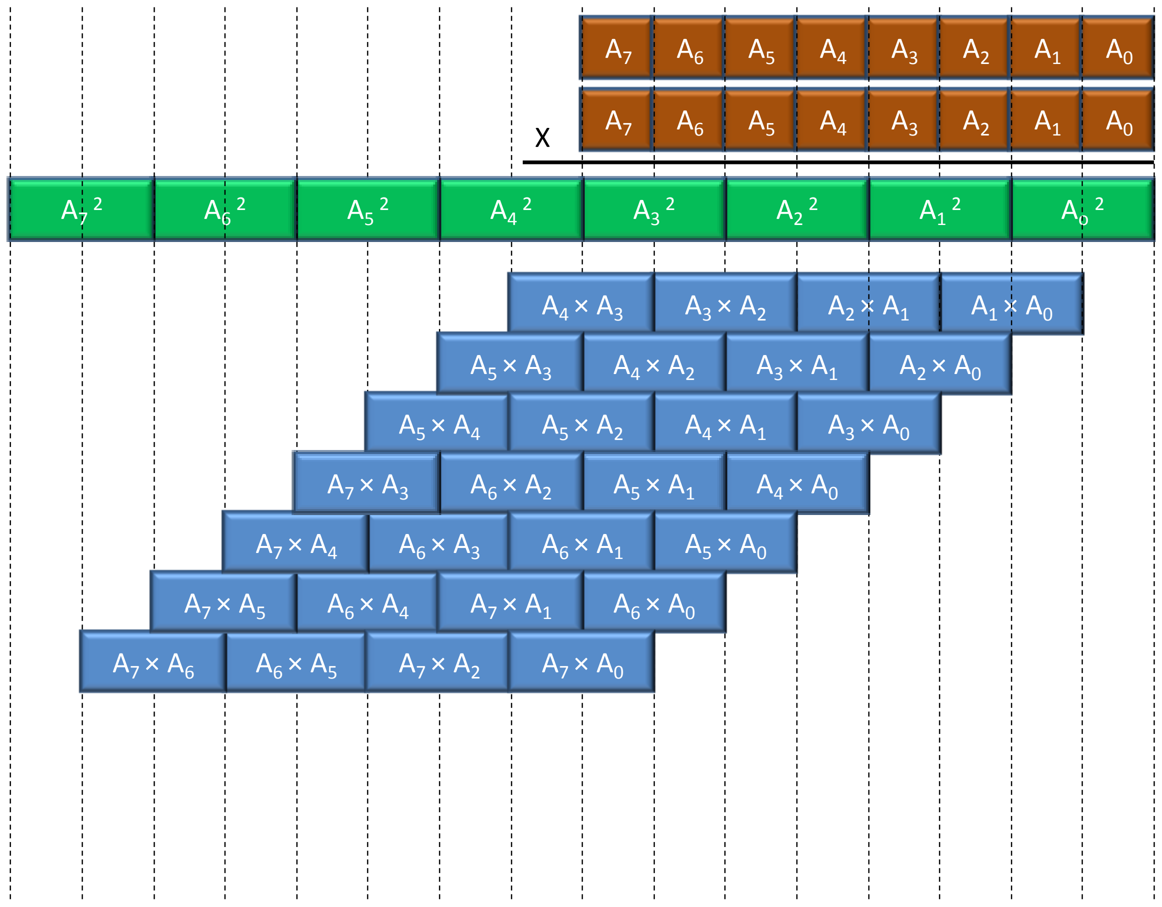

A7 A6 A5 A4 A3 A2 A1 A0
A7 A6 A5 A4 A3 A2 A1 A0
X
A7² A6² A5² A4² A3² A2² A1² A0²
A7×A6 A6×A5 A5×A4 A4×A3 A3×A2 A2×A1 A1×A0
A7×A5 A6×A4 A5×A3 A4×A2 A3×A1 A2×A0
A7×A4 A6×A3 A5×A2 A4×A1 A3×A0
A7×A3 A6×A2 A5×A1 A4×A0
A7×A2 A6×A1 A5×A0
A7×A1 A6×A0
A7×A0

The squaring approach in Figure 3 is better in terms of multiplications than a naïve multiplication as shown in Figure 2, however the diagonals are irregular and decrease in size.

Note that the blue terms need to be multiplied by 2 before being added to the green terms. As there are some inefficiencies and overheads associated with processing a diagonal, it is ideal to have equally sized diagonals.

We regroup the bottom diagonals to achieve a much better multiplication scheme as shown in Figure 4.

Figure 4: 512-bit squaring with the bottom diagonals reordered



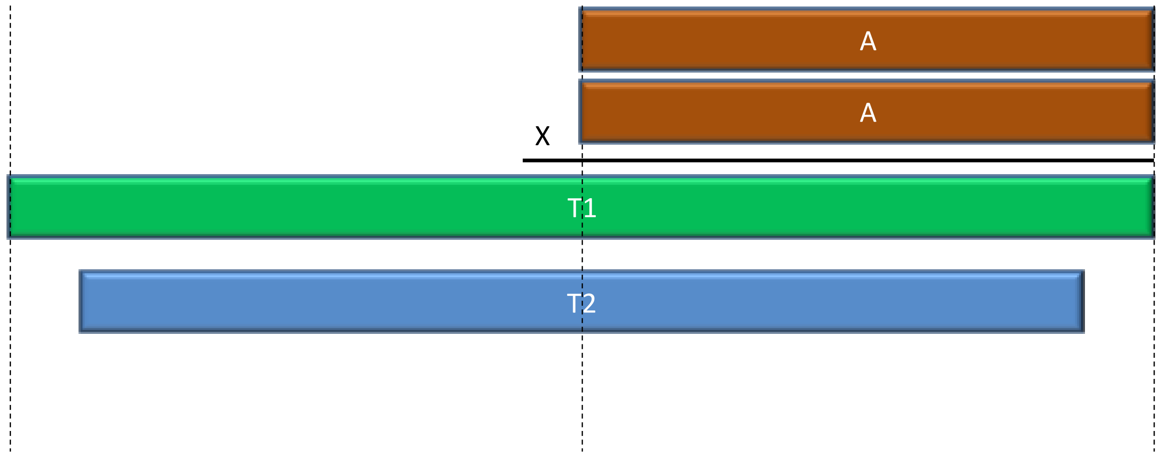
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
X
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
A_7^2 A_6^2 A_5^2 A_4^2 A_3^2 A_2^2 A_1^2 A_0^2
$A_4 \times A_3$ $A_3 \times A_2$ $A_2 \times A_1$ $A_1 \times A_0$
$A_5 \times A_3$ $A_4 \times A_2$ $A_3 \times A_1$ $A_2 \times A_0$
$A_5 \times A_4$ $A_5 \times A_2$ $A_4 \times A_1$ $A_3 \times A_0$
$A_7 \times A_3$ $A_6 \times A_2$ $A_5 \times A_1$ $A_4 \times A_0$
$A_7 \times A_4$ $A_6 \times A_3$ $A_6 \times A_1$ $A_5 \times A_0$
$A_7 \times A_5$ $A_6 \times A_4$ $A_7 \times A_1$ $A_6 \times A_0$
$A_7 \times A_6$ $A_6 \times A_5$ $A_7 \times A_2$ $A_7 \times A_0$

Four 7x1 diagonal multiplications can be performed efficiently to implement the bottom diagonals of squaring.

These symmetric diagonals can be processed efficiently with the mulx/adcx/adox instructions, exactly as described in [1] for large integer multiplication.

After the core 64x64 multiplications, we have 2 intermediate results T1 and T2 as shown in Figure 5.

Figure 5: Intermediate results



As a final computation, we need to perform the operation T1 + 2*T2. This can be done efficiently using the adcx and adox instructions. A quad-word of the T2 will be added to the corresponding quad-word of T1 with both adcx and adox instructions, creating two independent carry chains by adding T2 to T1 twice, without the overhead of a second pass of addition.

Conclusion

New instructions are being introduced on Intel® Architecture Processors to enable fast implementations of large integer arithmetic. This paper presents a brief introduction to large integer squaring and shows how it can be efficiently implemented. The complete source code for optimized implementations of modular exponentiation can be found in [3].

Contributors

We thank Gilbert Wolrich, Wajdi Feghali, Kirk Yap and Sean Gulley for their substantial contributions to this work.

References

[1] New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors
http://download.intel.com/embedded/processor/whitepaper/327831.pdf

[2] http://software.intel.com/file/45027

[3] RSAX Code -http://www.intel.com/p/en_US/embedded/hwsw/software/crc-license?id=6336

[4] Shay Gueron, Vlad Krasnov, "Speeding Up Big-Numbers Squaring," itng, pp.821-823, 2012 Ninth International Conference on Information Technology - New Generations, 2012

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://intel.com/embedded/edc.

Authors

Erdinc Ozturk, James Guilford, Vinodh Gopal are IA Architects with the IAG Group at Intel Corporation.

Acronyms

IA	Intel® Architecture
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

This paper is for informational purposes only. THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, Go to: http://www.intel.com/performance/resources/benchmark_limitations.htm

Hyper-Threading Technology requires a computer system with a processor supporting HT Technology and an HT Technology-enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. For more information including details on which processors support HT Technology, see here.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

Intel® Turbo Boost Technology requires a PC with a processor with Intel Turbo Boost Technology capability. Intel Turbo Boost Technology performance varies depending on hardware, software and overall system configuration. Check with your PC manufacturer on whether your system delivers Intel Turbo Boost Technology.For more information, see http://www.intel.com/technology/turboboost.

Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel Turbo Boost Technology, Intel Hyper Threading Technology, Intel Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2013 Intel Corporation. All rights reserved.