White Paper

Erdinc Ozturk
James Guilford
Vinodh Gopal
Wajdi Feghali

IA Architects
Intel Corporation

New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors

August 2012

327831 -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.

To maximize performance of code using these instructions, users can program at the assembly level. However, intrinsic definitions of mulx, adcx and adox will also be integrated into compilers. This is the first example of an “add with carry” type instruction being implemented with intrinsics. The intrinsic support will enable users to implement large integer arithmetic using higher level programming languages such as C/C++.

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, as well as for public key cryptography (e.g., RSA). In this paper, we describe the critical operations required in large integer arithmetic and their efficient implementations using the new instructions.

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, an 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 optimized libraries, such as the GNU Multi-Precision (GMP) library. The optimized code in these libraries traditionally uses scalar (integer) instructions that work on the General Purpose registers (preferably 64-bit).

Large integer multiplication is one of the most interesting arithmetic operations to consider in this context. Many algorithms, such as modular exponentiation and modular reduction, are based on multiplication, and the cost of doing these multiplications becomes dominant for the entire algorithm.

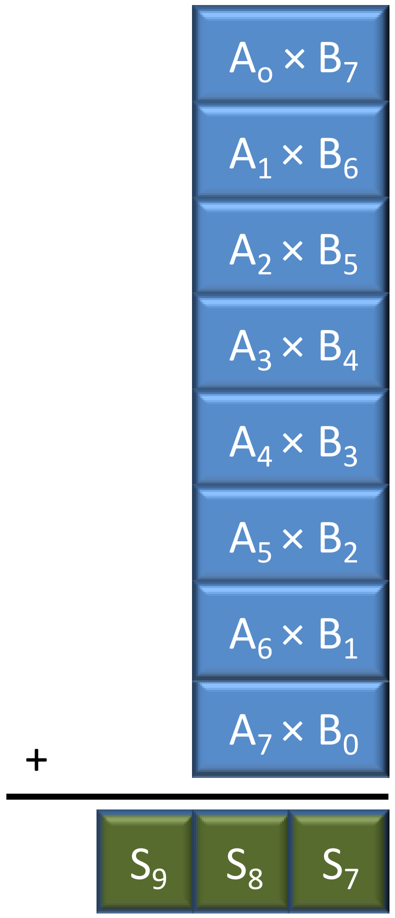
Introduction to Large Integer Multiplication

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 (depicted as green boxes in the diagrams below). In the case of IA, one "word" would be 64-bits. All N*M pairs of words are multiplied, generating two "words"-worth of bits (depicted in blue below) and the results are summed with the appropriate weights. If the sizes of the inputs are large enough (e.g. larger than 2048 bits), then an alternative approach, the Karatsuba Algorithm [1], becomes feasible. This is a more irregular method, which trades off fewer multiplications for more additions and complexity. In this paper, we focus only on efficient implementations of the school book method of multiplication.

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 a 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*, as shown in Figure 1.

Figure 1. Column-wise ordering for multiplication

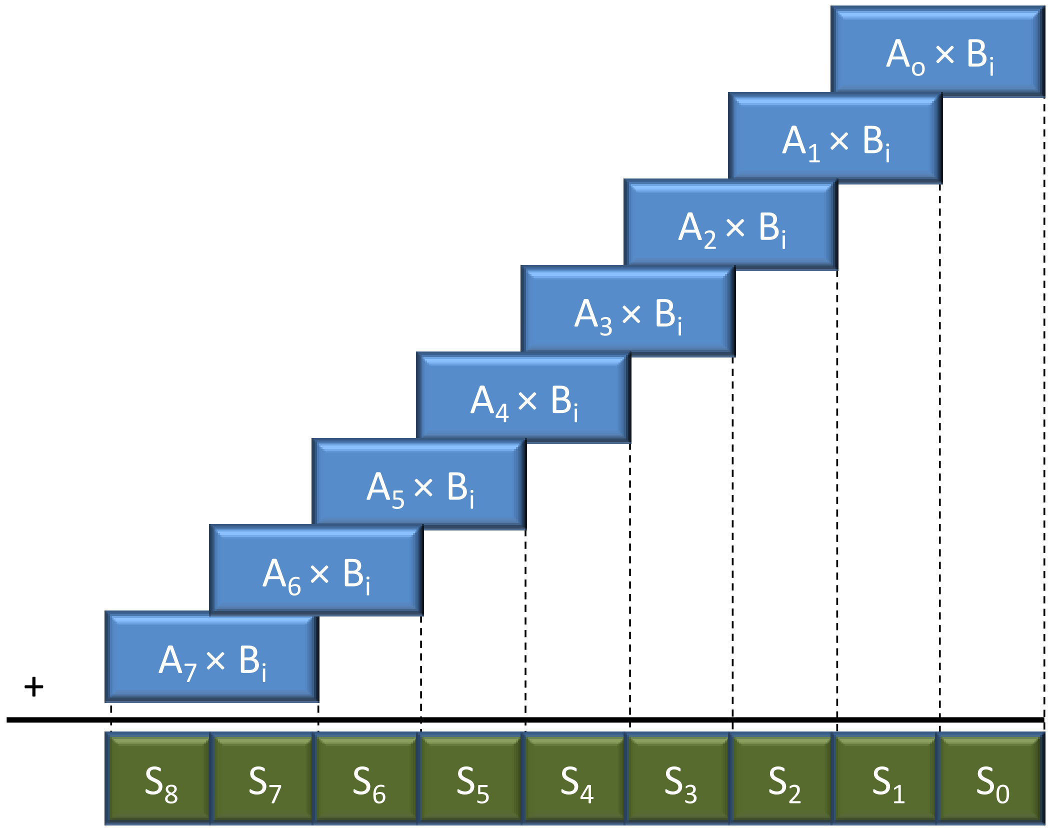


The main drawback to this approach is the number of add operations needed per multiply. Each product is two-words wide and needs two word-size adds (actually one add and one add-with-carry), with the high-order add generating a carry that then needs to be added into an accumulator.

Therefore, each multiplication requires one full word-size add (to S_7), one word-size add-with-carry (to S_8), and one add-carry-to-0 (to S_9).

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. $A_j \times B_i$ for all appropriate j and some fixed i. This is illustrated in Figure 2.

Figure 2. Row-wise ordering for multiplication (*diagonal*).



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 producing the S_n result, and then this sum needs to be added to a partial sum that accumulates the results from different diagonals.

The key point is that each addition can generate a carry to the next-higher word.

In one approach, the product results are added in a first pass, following the accumulation in a second pass. In this way, the high word result of A_0xB_i is added to the low word result of A_1xB_i, which generates S_1 and may also generate a carry out. Then the high word result of A_1xB_i is added to the low word result of A_2xB_i plus the carry in from the previous sum, which generates S_2 and may also generate a carry, and so on. Similarly, in the accumulation pass, S_n is added to Acc_n (not shown) and a carry in, which generates Acc_n and may generate a carry out. In the Intel® Architecture, transfer of the carry information is done through the carry flag of the eFlags register.

Each addition reads the carry flag to use as the carry in, and sets the carry flag for the carry out. Thus, in order to properly handle the carry scenario, the next higher order addition must be executed immediately before any other instruction modifies the carry flag. Simply put, the partial sum pass must be more-or-less contiguous in the instruction sequence, as is also true with the accumulation pass. An out-of-order machine can look ahead and process the accumulation pass in parallel with the partial sum pass using a renamed eFlags register. However, this look ahead is limited by the capacity of the machine's instruction scheduler. To start scheduling the accumulation pass in parallel with the partial sum pass, the remainder of the contiguous partial sum pass must be loaded into the scheduler, taking up valuable entries.

Alternatively, the instruction set ISA can include a mechanism to manage multiple independent carry indications. In this approach, the partial sum pass and the accumulation pass can be co-mingled, with one carry indication being used in the partial sum computations and one being used in the accumulation computations.

Another optimization is based on the observation that $(A \times B + C + D)$, where A, B, C, and D are one-word values, will never produce a result larger than two words. This means that one can add two low-order words to a two-word product (or one low-order word and a carry) without generating a carry-out of the high-order word. This allows one to efficiently consume the carry or in other words truncate the carry chain in some cases.

There are, of course, many variations on these themes. For example, the multiplication of two words by two words using diagonals can be used as a primitive. The above property is used internally to consume the carry at various points, and then this primitive can be used in a column-wise manner to form the full multiplication.

New Instruction Definitions

Three new integer 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:

- 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-out. 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”.

Programming Support and Tools

Compilers will have support for these instructions via intrinsics, allowing programmers to code in C/C++.

As an example, the following intrinsics can be used to develop 64-bit code:

```
unsigned __int64 umul128(unsigned __int64 a, unsigned __int64 b,
unsigned __int64 * hi);

unsigned char _addcarryx_u64(unsigned char c_in, unsigned __int64
src1, unsigned __int64 src2, unsigned __int64 *sum_out);
```

The first intrinsic provides support for mulx and the second provides support for the adcx and adox instructions.

Large Integer Multiplication

For many compute-intensive operations using large operands, the multiplication implementation 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.

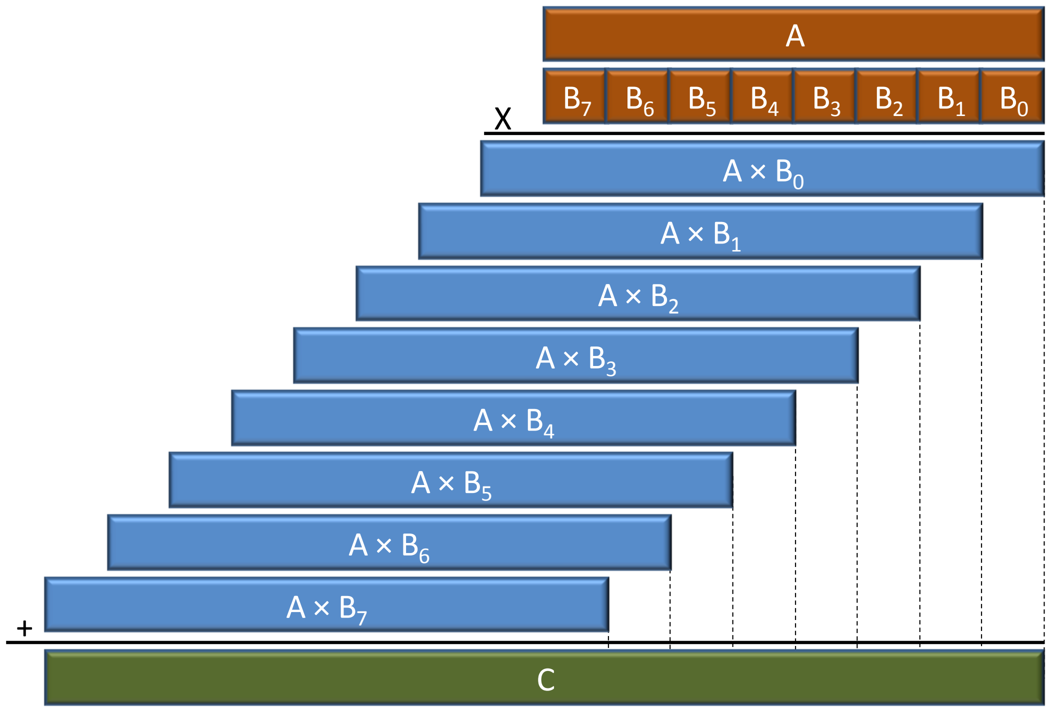
The main building block of the row-wise approach is the diagonal, as explained in Figure 2. As an example, we will show how to realize a 512x512-bit multiplication using 8 512x64-bit diagonals. High level breakdown of this approach is shown in Figure 3.

Algorithm: 512x512-bit multiplication

```
Input: A[7:0], B[7:0]

Output: C[15:0] = A * B

Step1: {R[7:0], C[0]} = A[7:0] * B[0]

for i from 1 to 7:

   Step2: {R’[7:0], C[i]} = A[7:0] * B[i] + R[7:0]

Step3: C[15:8] = R[7:0]
```

Here, R[7:0] are 64-bit registers and A, B, and C are memory locations (A and B are 512-bit operands and C is the 1024-bit result). At the end of the multiplication, the registers will be stored into the memory location C[15:8]. Note that R’[7:0] means the state of R[7:0] after an iteration of the loop.

Figure 3. High-level breakdown of a 512x512 multiplication



The first diagonal of the multiplication (Step 1) is shown in Figure 4.

Figure 4. First diagonal of a 512x64 bit multiplication for the row-wise approach

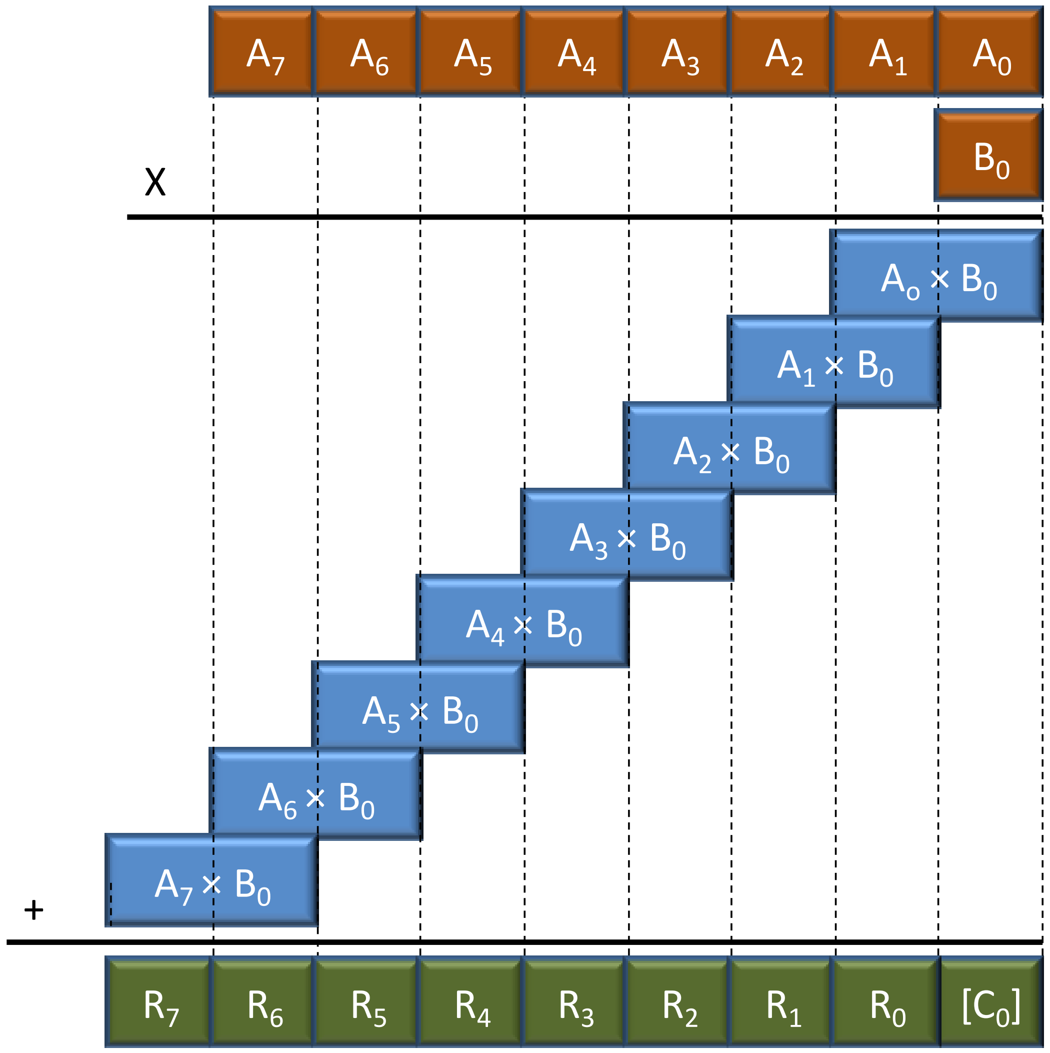


Table 1. Instruction sequence for Step 1

mul-based instruction sequence	mulx-based instruction sequence
mov OP, [pB+8*0]	mov rdx, [pB+8*0]
mov rax, [pA+8*0]	
mul OP	mulx R0, rax, [pA+8*0]
mov [pDst+8*0], rax	mov [pDst + 8*0], rax
mov R0, rdx	
mov rax, [pA+8*1]	
mul OP	mulx R1, rax, [pA+8*1]
add R0, rax	add R0, rax
adc rdx, 0	
mov R1, rdx	
mov rax, [pA+8*2]	
mul OP	mulx R2, rax, [pA+8*2]
add R1, rax	adc R1, rax
adc rdx, 0	...
mov R2, rdx	
...	

A comparison of the instruction sequences using the mul instruction and mulx instructions is shown in Table 1. Since there is only one carry chain, use of adcx/adox instructions is not necessary for this diagonal.

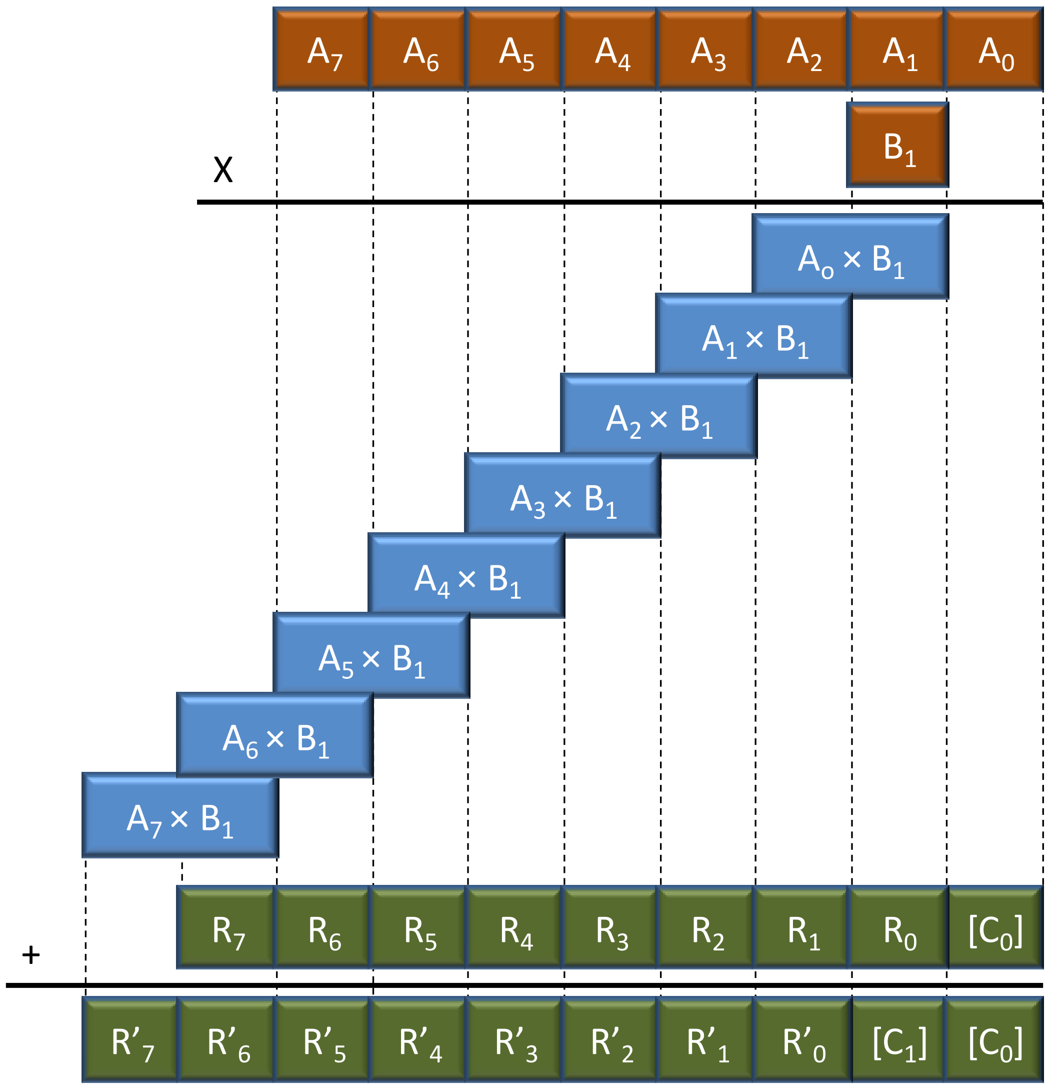
Figure 5 shows the diagonal structure of Step 2 of the multiplication algorithm. As seen in the figure, the intermediate result stored in registers R7:R0 is added to the current row of the multiplication A*B_i and the result stays in registers R7:R0, with the exception of the bottom word being stored into memory.

For step 2, a comparison of instruction sequences using the mul instruction, mulx instruction and adcx/adox instructions is shown in Table 2. It should be noted that R_i and R'_i are the same registers, R'_i shows the positions of the registers after a 512x64 diagonal multiplication. As can be seen from the figure, mulx instruction reduces the number of mov instructions required without any change in the method of implementation of a 512x64 diagonal, as a result of the structure of the instruction. Adcx/adox instructions further reduce the number of instructions required for a diagonal, by enabling the user to implement it with a more efficient code flow.

Table 2. Instruction sequence for Step 2

mul-based instruction sequence	mulx-based instruction sequence	mulx/adcx/adox based instruction sequence
mov OP, [pB+8*0]	mov OP, [pB+8*0]	xor rax, rax
		mov rdx, [pB+8*0]
mov rax, [pA+8*0]		
mul OP	mulx TMP1,rax, [pA+8*0]	mulx T1, T2, [pA+8*0]
add R0, rax	add R0, rax	adox R0, T2
adc rdx, 0	adc TMP1, 0	adcx R1, T1
mov TMP, rdx	mov pDst, R0	mov pDst, R0
mov pDst, R0		
mov rax, [pA+8*1]		
mul OP	mulx TMP2,R’0, [pA+8*1]	mulx T1, R’0, [pA+8*1]
mov R0, rdx	add R’0, R1	adox R’0, R1
add R1, rax	adc TMP2, 0	adcx R2, T1
adc R0, 0	add R’0, TMP1	
add R1, TMP	adc TMP2, 0	
adc R0, 0		
mov rax, [pA+8*2]		
mul OP	mulx TMP1,R’1, [pA+8*2]	mulx T1, R’1, [pA+8*2]
mov TMP, rdx	add R’1, R2	adox R’1, R2
add R2, rax	adc TMP1, 0	adcx R3, T1
adc TMP, 0	add R’1, TMP2	…
add R2, R0	adc TMP1, 0	
adc TMP, 0	…	
…		

Figure 5. Diagonal structure for Step 2



Conclusion

New integer 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 multiplication and shows how it can be efficiently implemented using the Intel® Architecture instruction set. We also demonstrate how the use of the new mulx, adcx, and adox instructions can result in an even more efficient solution. Complete source code for optimized implementations of modular exponentiation can be found in [3].

Acknowledgements

We thank Gilbert Wolrich, Sean Gulley, Sean Mirkes and Matthew Merten for their substantial contributions.

References

[1] http://en.wikipedia.org/wiki/Karatsuba_algorithm

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

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

Authors

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

Acronyms

IA	Intel® Architecture

§