LBA (Long Binary Architecture)
by Theo Kortekaas

Purpose

Purpose of the Lba architecture is to perform in a simple way arithmetic operations on very large numbers in an assembler or C^{++} program. Such a method is also referred to as arbitrary-precision arithmetic, or multiple precision arithmetic. Another common term for big numbers in computer arithmetic is Bignum.
Simple operations, like addition, subtraction and multiplication, usually take place in integer registers that have a limited capacity. Lba provides the ability to perform the same operations on integer variables with a large capacity. To this end, an LBA variable is defined, called Lba-field, with virtually unlimited capacity. Also several LBA functions are is defined that operate on the Lba-fields. The aim is to make the implementation of the LBA operations as fast as possible. Therefore it has been decided to work only with natural numbers and zero, not with negative numbers.
The LBA architecture is designed for a computing environment based on an X86 processor running under a 32-bit Windows system. The used program development environment is a Windows XP system with Borland CBuilder 4.0 that contains the tools TASM32, BCC32, ILINK32 and BRC32.
This development environment is rather outdated; however, the so produced programs run smoothly on Windows 7, Windows 8 and Windows 10.
A modernization would be desirable; as well as a transition to 64-bit. The prospect, however, to have to revise thousands of *lines of code*, is still a big barrier.
The capacity of an Lba-field is at present only limited by some practical limits, such as the maximum size of an address-space under a Win32 operating system and the maximum number that is possible for bitnumber in a 32-bit integer.
A practical limit of an Lba-field is therefore 100,000,000 integers of 32 bits. No check is performed on this limit, but excess of it can lead to unpredictable results.

History

The beginning of the development of LBA dates from the time that the 80386 processor was introduced, and OS/2 was still a promising operating system. Based on this platform (80386 and OS/2), a library was set up with key arithmetic functions for Lba-fields. This library has been carried out in the form of assembly language macros.
Further development in the computer world made it necessary to carry out several times a migration for this function-library: from an OS/2 environment to Windows 3.1; from Windows 3.1 to the Win32 application program interface. A transition to a 64-bit environment is not provided yet.

Further enhancements in LBA offer special features such as calculation of Greatest Common Divisor; the Extended Euclidean Algorithm and the Montgomery Modular Multiplication. Also an interface between LBA and C^{++} is developed.
The documentation, available with this publication, is mainly limited to operations, equivalent with X86 instructions.

X86 terminology

In this description the X86 terminology is used. In this terminology, computer memory consists of bytes. Each byte is 8 bits and is separately addressable. The bits are referred to as bit 0 to bit 7.
A word has 16 bits and can be located in a 16-bit register or in two consecutive bytes in the computer memory. The bitnumbering in a word is bit 0 to bit 15. Bit 0 has the value 1 if the bit is "on" and bit 15 has a value of 32,768 if the bit is in the "on" state.
A double word is 32 bits and can contain a 32-bit integer. Usually a double word is called a dword. A dword may be located in a 32-bit register or in four consecutive bytes in computer memory. The bitnumbering in a dword is bit 0 to bit 31.
If a dword is located in four consecutive bytes in computer memory, bit 0 to bit 7 are in the first byte; bit 8 to bit 15 in the second byte; bit 16 to bit 23 in the third byte and bit 24 to bit 31 in the fourth byte. This method of storage is called little-endian.
X86 has eight 32-bit registers (seven of which are for general use). These are called general registers. These registers are named: EAX, EBX, ECX, EDX, EDI, ESI, EBP and ESP (the latter serves as the stack pointer). Furthermore, there is a flag register with one-bit indicators (flags) such as zero-flag; carry-flag etc.

Lba-field

Within the LBA architecture multiple formats are used. The main format is the Lba-field. This description is limited to the Lba-field. The Lba-field is a field in computer memory in which a number can be stored. (An Lba-field can also be saved in an external file). The Lba-field consists of two parts: the LBA-length field and the LBA-body. The LBA-length field consists of a single dword in which the length *n* of the LBA-body is stored. The LBA-body is composed of *n* dwords which are positioned directly adjacent to the LBA-length field in the computer memory. A practical limit for *n* is 100,000,000. This is not checked. All (arithmetic) operations are performed, in principle, on Lba-fields, and take place in units of dwords. In the literature, these units are called limbs.
The bitnumbering runs for the first limb (after the LBA-length field) just from bit number zero to bit number 31. In the following limbs the bitnumbering continues from 32 to bit number 63 for the second limb, from bit number 64 to bitnumber 95 for the third limb...etc. A practical limit for bit number is 3,199,999,999. However, this is not checked.
The definition of an Lba-field determines its length; this length can not be changed during the life of the program. (There are ways to dynamically create and destroy Lba-fields; that is further not discussed here). In the definition of an Lba-field the content of the LBA-body is undefined. Lba-fields should not overlap.

LBA components

LBA takes shape in the form of this description. Further components are a file assembler sources (lba.asm); a file assembler macros (lba.mac) and a file with C^{++} definitions (lba.h). These files can be viewed and downloaded here.

- tonjanee.home.xs4all.nl/lba.asm
- tonjanee.home.xs4all.nl/lba.mac
- tonjanee.home.xs4all.nl/lba.h

Furthermore, there is a compiled version of the LBA library in the form of an object file: it can be linked with another program.

- tonjanee.home.xs4all.nl/lba.obj

This object file does not contain the reporting functions "Report" and "Mes". These functions can be used only in the "LBA operating environment".
This means a combined Assembler and C^{++} program with menu structure and provision for creating and displaying a log file.

Description LBA definitions and operations

The operations described here are mainly simple operations for which no special services are needed such as memory management, interruptions or display services. These are operations for which, in general, an X86 instruction equivalent is available. In that case, the parameters of the operations are as much as possible similar to the equivalent X86 instructions.
If an invalid condition is detected, the result field is, usually, set to zero. The Lba-fields should not overlap.
In the description of each operation the following information is given consecutively:

The first line (s):	The name of the operation and the parameters. The parameterdescription can mention a type: lba; the address of an Lba-field; int; the contents of a 32-bit integer; intad; the address of a 32-bit integer.
The second line:	the call information from a C++ environment. This information is based on the Borland compiler bcc32 and refers to a function in the file lba.h
The third line:	the call information from an Assembler environment. This information is based on Borland tasm32 Macro Assembler, and refers to a macro in the lba.mac file. The name of the macro is, if possible, the name of the equivalent X86 instruction, supplemented by a M.
The fourth line:	the mnemonic of the equivalent X86 instruction. This is the mnemonic as used in the Intel documentation [1].

The remaining lines describe the operation.

LBA definitions

Define LBA variabele   Parameters: LBA-name,Length

C^{++}:	**Ldefb (Name,n) ;**
Asm:	**LDEFB Name,n**
X86:	**none**

LDEFB is used to define an LBA variabele with Name as name and Length as size in dwords (this length is excluding the length-field itself). The first dword gets the content *n*; the next *n* dwords has content undefined.

LBA Operations

Move **Parameters: To(lba),From(lba)**
C^{++}: **Mov (To,From)**
Asm: **MOVM To,From**
X86: **MOV**

The content of the from-field is copied to the to-field. The content of the first element (dword) of the LBA-body of the from-field goes to the first element (dword) of the LBA-body of the to-field, and so forth. If the length of the to-field is greater than the length of the from-field, than the remaining part of the to-field is filled with elements with the content of zero. If the length of the from-field is greater than the length of the to-field then only the part of the from-field that fits in the to-field is transferred to the to-field. Of the portion of the from-field that can not be transferred to the to-field the contents of the elements are checked for zero. If one or more of these elements does not have content equal to zero, the entire operation is considered invalid and the to-field is set to zero.
The from-field may not be the same as the to-field.
The contents of all general registers remains unaffected, as well as the status of the flags.

Copy **Parameters: To(lba),From(lba)**
C^{++}: **Copy (To,From)**
Asm: **Copy To,From**
X86: **None**

The content of the from-field is copied to the to-field. It is assumed that both fields are equally large. This operation is the fastest way to move an LBA field. However, no check is performed on the length of both fields. The contents of all general registers remains unaffected, as well as the status of the flags.

Load	**Parameters: To-field(lba),From-field(int)**	
C++:	**Load**	**(To,From)**
Asm:	**LOAD**	**To,From**
X86:	**None**	

The Lba-field "To" is set to zero; then the contents of the 32 bits integer "From" is placed in the first 32 bits of the LBA field. The term "int" may contain one of the general registers, or a value up to 2^{32} - 1. The contents of all general registers remains unaffected. Also, the content of the integer field remains unchanged.
The status of the flags is uncertain.

Store	**Parameters: To-field(intAd),From-field(lba)**	
C++:	**Store**	**(To,From)**
Asm:	**STORE**	**To,From**
X86	**None**	

The first 32 bits of the LBA-field "From" are placed in the 32-bit integer field "To". "To" contains the address of a 32-bit memory field. The content of the LBA field remains unchanged. The contents of all general registers remains unaffected, as well as the status of the flags.

Add **Parameters: Sum(lba),Num(lba)**
C^{++}: **Add (Sum,Num)**
Asm: **ADDM Sum,Num**
X86: **ADD**

The contents of the Num-field is added to the Sum-field. If the Num-field is smaller than the Sum-field, the Num-field is assumed to be supplemented by elements having the value zero. If the two fields are equal in size, then the result is reliable, unless there is a carry at the last element. In this case (and also in the event that a carry occurs at the last element when the Num-field is smaller than the Sum-field) the result of the addition is invalid and the whole sum-field is set to zero.
When the Sum-field is smaller than the Num-field, the rest of the Num field, that exceeds the size of the Sum-field, may contain only elements with the value zero. This will be checked and if this is not the case, the result of the addition is invalid and the whole sum-field is set to zero.
The Num-field will remain unchanged.
The contents of all general registers remains unaffected, as well as the status of the flags.

Subtract **Parameters: Sum(lba),Num(lba)**
C^{++}: **Sub (Sum,Num)**
Asm: **SUBM Sum,Num**
X86: **SUB**

The contents of the Num-field is subtracted from the contents of the Sum-field. If the Num-field is smaller than the Sum-field, the Num-field is assumed to be complemented with elements having the value zero. If the two fields are equal in size, the result is correct unless there is a borrow-situation at the last limb. In this case (and also in the case that a borrow occurs when the Num-field is smaller than the Sum-field), the result of the subtraction is invalid and the entire Sum-field is set to zero. When the Sum-field is smaller than the Num-field, the part of the Num field that the length of the Sum-field exceeds may contain only elements with the value zero. This will be checked and if this is not so, the result of the subtraction is invalid and the entire Sum-field is set to zero. The Num field will remain unchanged. The contents of the general registers remains unaffected.

Multiply	**Parameters: Product(lba),Mc(lba),Mr(lba)**
C++:	**Mul (Product,Mc,Mr)**
Asm:	**MULM Product,Mc,Mr**
X86:	**MUL**

The contents of Multiplicand (Mc) and Multiplier(Mr) are multiplied according to the classical method[2] en the result is placed in Product. If the sum of the size (in number of dwords) of Mr and Mc is greater than the size (in number of dwords) of the Lba-field Product it is assumed that the size of the Lba-field Product is not sufficient is to contain the value of Product. In that case the operation is invalid and Product will be set to zero.
The contents of Mc and Mr are preserved.The contents of the general registers remains unaffected. The status of the flags is changed.

Divide	**Parameters: Quotiënt(lba),Dividend(lba),Divisor(lba)**
C++:	**Div (Quotiënt,Dividend,Divisor)**
Asm:	**DIVM Quotiënt,Dividend,Divisor**
X86:	**DIV**

The content of Dividend is divided by the contents of Divisor and the result is placed in Quotient. Any remainder is placed in Dividend. The content of Divisor can not be zero. If Divisor is zero anyhow, the operation is considered invalid and Quotient is set to zero. The content of Divisor remains unchanged. The contents of the general registers remains unaffected. The status of the flags is changed.

Increase	**Parameters: Lba-field(lba)**
C^{++}:	**Inc (Lba-field) ;**
Asm:	**INCM Lba-field**
X86:	**INC**

Lba-field contains the address of the Lba-field. The contents of the Lba-field is incremented by one. If the Lba-field contains, before the operation, all 1's, then content is entirely put to zero and the carry-flag is set to 1. In any other situation, the carry flag is reset to 0.
The contents of all general registers remains unaffected.

Decrease	**Parameters: Lba-field(lba)**
C^{++}:	**Dec (Lba-field)**
Asm:	**DECM Lba-field**
X86:	**DEC**

Lba-field contains the address of the Lba-field. The contents of the Lba-field is decremented by one. If the content of the Lba-field, before the operation, is zero, then content remains zero and the carry-flag is set to 1. In any other situation, the carry flag is reset to 0.
The contents of all general registers remains unaffected.

Shift Left Parameters: Lba-field(lba),Nbits(int)
C^{++}: Shl (Lba-field,Nbits)
Asm: SHLM Lba-field,Nbits
X86: SHL

Lba-field contains the address of the Lba-field and Nbits, as 32-bit integer, contains the number of bits that the content of the Lba-field is to be shifted to the left. A shift of *n* bits to the left is equivalent to a multiplication of the contents of the Lba-field by a factor of 2^n. If the number of bits to shift in Nbits is greater than the number of bits that make up the Lba-field, then the operation is invalid and is not executed. This is not signaled.
The content of the bits that is shifted out on the left side is lost.
The bits that come in to the right get content 0.
The contents of all general registers remains unaffected, as well as the status of the flags.

Shift Right Parameters: Lba-field(lba),Nbits(int)
C^{++}: Shr (Lba-field,Nbits)
Asm: SHRM Lba-field,Nbits
X86: SHR

Lba-field contains the address of the Lba-field and Nbits, as 32-bit integer, contains the number of bits that the content of the Lba-field is to be shifted to the right. A shift of *n* bits to the right is equivalent to a division of the contents of the Lba-field by a factor of 2^n. If the number of bits to shift in Nbits is greater than the number of bits that make up the Lba-field, then the operation is invalid and is not executed. This is not signaled.
The content of the bits that is shifted out on the right side is lost.
The bits that come in to the left side get content 0.
The contents of all general registers remains unaffected, as well as the status of the flags.

Set Bit	**Parameters: Lba-field(lba), Bitnr(int)**	
C^{++}:	**Bts**	**(Lba-field,Bitnr) ;**
Asm:	**BTSM**	**Lba-field,Bitnr**
X86:	**BTS**	

Lba-field contains the address of the Lba-field. Bitnr contains, as 32-bit integer, the number of the bit in the Lba-field, which should be put on 1.
Bitnr can not contain a value greater than the size of the Lba-field (in number of bits = (number of dwords * 32) - 1). If Bitnr is greater, then the operation is invalid and is not executed. However, this is not signaled.
The content of all general registers remain unchanged.

Complement Bit		**Parameters: Lba-field(lba), Bitnr(int)**
C^{++}:	**Btc**	**(Lba-field,Bitnr) ;**
Asm:	**BTCM**	**Lba-field,Bitnr**
X86:	**BTC**	

Lba-field contains the address of the Lba-field. Bitnr contains, as 32-bit integer, the number of the bit in the Lba-field, which should be “flipped” (a 0 becomes an 1; an 1 becomes a 0).
Bitnr can not contain a value greater than the size of the Lba-field (in number of bits = (number of dwords * 32) - 1). If Bitnr is greater, then the operation is invalid and is not executed. However, this is not signaled.
The content of all general registers remain unchanged.

Reset Bit	**Parameters: Lba-field(lba), Bitnr(int)**	
C^{++}:	**Btr**	**(Lba-field,Bitnr) ;**
Asm:	**BTRM**	**Lba-field,Bitnr**
X86:	**BTR**	

Lba-field contains the address of the Lba-field. Bitnr contains, as 32-bit integer, the number of the bit in the Lba-field, which should be reset to zero.
Bitnr can not contain a value greater than the size of the Lba-field (in number of bits = (number of dwords * 32) - 1). If Bitnr is greater, then the operation is invalid and is not executed. However, this is not signaled.
The content of all general registers remain unchanged.

Bit Scan Forward		**Parameters: Bitnr(intAd),Lba-field(lba)**
C^{++}:	**Bsf**	**(Bitnr,Lba-field)**
Asm:	**BSFM**	**Bitnr,Lba-field**
X86:	**BSF**	

Lba-field contains the address of the Lba-field. Bitnr contains the address of a 32-bit integer. The Lba-field is scanned from bit number zero (the bit having the smallest value) to the first 1-bit. The number of this bit is placed in Bitnr. If the Lba-field as a whole is zero, then Bitnr is set to -1 in order to distinguish this situation from the situation that the zeroth bit is 1.
The content of all general registers remain unchanged.

Bit Scan Reverse		**Parameters: Bitnr(intAd),Lba-field(lba)**
C^{++}:	**Bsr**	**(Bitnr,Lba-field) ;**
Asm:	**BSRM**	**Bitnr,Lba-field**
X86:	**BSR**	

Lba-field contains the address of the Lba-field. Bitnr contains the address of a 32-bit integer. The Lba-field is scanned from the bit with the highest value to the first bit which is set to 1. The number of this bit is set in Bitnr. If the Lba-field as a whole is zero, then Bitnr is set to -1 in order to distinguish this situation from the situation in which the zero bit is set to 1 and this is the only bit of the Lba-field which is set to 1.
The content of all general registers remain unchanged.

Modulo	**Parameters:**	**Base(lba),Modulus(lba)**
C^{++}:	**Mod**	**(Base,Modulus) ;**
Asm:	**MOD**	**Base,Modulus**
X86:	**none**	

The number in Lba-field Base is divided by the number in Lba-field Modulus. The quotient is irrelevant and will not be saved. The remainder of the division is placed in Lba-field Base and replaces the original content. (This residue may be zero.)
The content of Modulus remains unchanged. If the Lba-field Modulus contains zero then the operation is invalid and the Lba-field Base is set to zero.
The contents of the general registers remains unaffected. The status of the flags is changed.

Compare Zero   Parameters: Lba-field(lba)
C^{++}:   EqualZero   (Lba-field) ;
Asm: EqualZero   Lba-field
X86:   None

The contents of the Lba-field is compared with the value 0. In the the assembler environment the AL register functions as a result-code.
AL = 0 if the contents ≠ 0.
AL = 1 if the content has the value 0.
In a C^{++} environment, the instruction can be used as follows:
if (EqualZero(Lba-field)) {This is done if content = 0}
else {this will be carried out if content ≠ 0};
In an assembler environment, the instruction is used as follows:

```
EqualZero  Lba-field
JE         label   ; go to label if content = 0
```

The contents of all general registers, except the EAX register, remains unaffected. The status of the flags is changed.

Compare One   Parameters: Lba-field(lba)
C^{++}:   EqualOne   (Lba-field) ;
Asm: EqualOne   Lba-field
X86:   None

The contents of the Lba-field is compared with the value 1. In the the assembler environment the AL register functions as a result-code.
AL = 0 if the contents ≠ 1.
AL = 1 if the content has the value 1.
In a C^{++} environment, the instruction can be used as follows:
if (EqualOne(Lba-field)) {This is done if content = 1}
else {this will be carried out if content ≠ 1};
In an assembler environment, the instruction is used as follows:

```
EqualOne   Lba-field
JE         label   ; go to label if content = 1
```

The contents of all general registers, except the EAX register, remains unaffected. The status of the flags is changed.

Compare Equal **Parameters: Lba-field1(lba), Lba-field2(lba)**
C++: **EqualTo** **(Lba-field1,Lba-field2) ;**
Asm: **EqualTo** **Lba-field1,Lba-field2**
X86: **none**

If the contents of the first Lba-field is equal to the contents of the second Lba-field, then the condition is true and will be handled in a C++ program as such. In an assembler program, the zero flag is put on if the contents of the first Lba-field is equal to the contents of the second field-Lba.
If the Lba fields are unequal in length, then the shortest field is assumed to be complemented with zeros.
In a C++ environment, the instruction can be used as follows:
if (EqualTo (Lba-field1, LBA-field2)) {this will be executed if the contents is the same}
else {this will be executed if the contents is unequal};
In an assembler environment, the instruction is used as follows:

```
EqualTo   Lba-field1, LBA-field2        ;go to label if contents
JE        label                         ;equal
```

The contents of all general registers, with exception of the EAX register, remains unaffected.

Compare LessThan **Parameters: Lba-field1(lba), Lba-field2(lba)**
C++: **LessThan** **(Lba-field1,Lba-field2) ;**
Asm: **LessThan** **Lba-field1,Lba-field2**
X86: **none**

If the contents of the first Lba-field is smaller than the contents of the second Lba-field, then the condition is true and will be handled in a C++ program as such. In an assembler program, the zero flag is put on if the contents of the first Lba-field is smaller than the contents of the second Lba-field.
If the Lba fields are unequal in length, then the shortest field is assumed to be complemented with zeros.
In a C++ environment, the instruction can be used as follows:

```
if (LessThan (Lba-field1,Lba-field2)) {This is done if contents
              1st Lba-field smaller than contents 2nd Lba-field}
else          {this will be executed if contents of 1st Lba-field
              is greater than or equal to contents of 2nd Lba-field};
```

In an assembler environment, the instruction is used as follows:

```
LessThan  Lba-field1, Lba-field2
JE        label       ;go to label if contents 1st Lba-field is
                      ;smaller than contents of 2nd Lba-field
```

The contents of all general registers, but EAX, remains unchanged.

Bit test **Parameters: Lba-field(lba),Bitnr(int)**
C^{++}: **Bt (Lba-field,Bitnr) ;**
Asm: **BTM Lba-field,Bitnr**
X86: **BT**

Lba-field contains the address of the Lba-field and Bitnr contains, as a 32-bit integer, the bit number of the bit whose value is to be tested. If the bit number is beyond the length of the Lba-field (in number of bits), then the instruction is invalid and is not executed. This is not signaled in a C^{++} environment. In the assembler environment the AL register functions as result code:
AL = 0 if the selected bit has a value of 0.
AL = 1 if the selected bit has a value of 1.
AL = 9 if the statement is not valid.
In a C^{++} environment, the result of the instruction can be processed as follows:

```
if (Bt (Lba-field,Bitnr)) {this is executed if the bit = 1}
        else {this is executed if the bit = 0};
```

In an assembler environment, the instruction is used as follows:

```
        BTM         Lba-field,Bitnr
        JE          label               ;go to label if bit = 1
```

The contents of all general registers, except the EAX register, remains unaffected. The status of the flags is changed.

Number Of Significant Bits **Parameters: Nbits(intAd),Lba-field(lba)**
C^{++}: **Nosb (Nbits,Lba-field) ;**
Asm: **NOSB Nbits,Lba-field**
X86: **none**

The number of significant bits of the Lba-field is determined and this number is placed in the 32-bits integer who's address is specified in Nbits.
The number of significant bits is the total number of bits of the Lba-field less the number of leading bits that is set to zero.
The Lba-field remains unchanged. The contents of all general registers remains unaffected as well as the status of the flags.

Special LBA Operations

Greatest Common Divisor Parameters: Gcd(lba),Factor1(lba),Factor2(lba)

C^{++}:	**Gcd**	**(Gcd,Factor1,Factor2)**
Asm	**GCD**	**Gcd,Factor1,Factor2**
X86:	**none**	

This function calculates the greatest common divisor (GCD) [3] of Factor1 and Factor2. The three parameters are the addresses of Lba-fields. The two factors may not be zero. If one of the factors, or both factors are zero, then the operation is invalid, and GCD is put to zero.
The content of all general registers remains unaffected.

Extended Euclidean algorithm[4]

Parameters: Pos(lba),Neg(lba),Factor1(lba),Factor2(lba)

C^{++}:	**Eucal**	**(pos,neg,f1,f2) ;**
Asm:	**Eucal**	**Pos,Neg,F1,F2**
X86	**none**	

The operation Eucal calculates according to the Extended Euclidean Algorithm: xF1 - yF2 = Gcd.
Factor1 and Factor2 (F1 and F2) are input parameters. Pos and Neg are output parameters. All parameters are addresses of Lba-fields.
First is calculated the Greatest Common Divisor of F1 and F2.
Then, x and y are calculated. Of these output variables is x positive and y is negative. Because LBA does not use negative numbers, the positive value x is placed in the Lba-field **Pos** and the negative value y (as a positive number) is placed in the Lba-field **Neg.**
If necessary, F1 and F2 values are changed to allow correct calculation of xF1 - yF2 = Gcd.
LBA workfields that are required are dynamically created during the calculation.
The contents of all general registers remains unaffected.
The content of F1 and F2 can be optionally interchanged.

Square Multiplication Parameters: Result(lba),Multiplier(lba)

C^{++}:	**Smul**	**(Result,Multiplier) ;**
Asm	**SMUL**	**Result,Multiplier**
X86:	**none**	

SMUL calculates the square of Multiplier and places the result in Result. Result and Multiplier both contain the addresses of Lba-fields. If the Result field is too small to be able to contain the square then Result is entirely put to zero. The contents of all general registers remains unaffected.

Montgomery Square Multiplication Parameters:
Result(lba),Mr(lba),Modulus(lba),ModInv(int),NS(int)

C++:	**Mgs (Result,Mr,Modulus,ModInv,NS)**
Asm	**MGS Result,Mr,Modulus,ModInv,NS**
X86:	**none**

This function calculates the square of Mr. (mod Modulus) according to the Montgomery method [5]. Result is the output, and Mr is the input. Modulus is the modulus. These three parameters are addresses of Lba-fields. The parameters ModInv and NS are 32-bit integer fields. ModInv is the modular multiplicative inverse function, and NS contains the number of shifts that must be performed in order to carry out the entire Montgomery function. NS functions as a count-down counter, and is modified during the execution of MGS.
The contents of all general registers remains unaffected

Montgomery Multiplication Parameters: Result(lba), Multiplier(lba), Multiplicand(lba),Modulus(lba),ModInverse(int),NS(int)

C++:	**Mgm (Result,Mr,Mc,Mod,ModInv,NS)**
Asm	**MGM Result,Mr,Mc,Mod,ModInv,NS**
X86:	**none**

From this function Montgomery Square Multiplication (MGS) is called.
This function calculates the product of Mr and Mc (mod Modulus) according to the Montgomery methode [5]. Result is the output, and Mr and Mc is the input. Mod is the modulus. This four parameters are addresses of Lba-fields. The parameters ModInv en NS are 32-bit integer fields. ModInv is the modular multiplicative inverse function and NS contains the number of shifts to be executed to complete the whole Montgomery Function. NS is a count-down counter that is updated during execution of MGS that is called from MGM.
The contents of all general registers remains unaffected.

Modular Exponentiation	**Parameters:**	
	RC(intAd),Result(lba),Base(lba),Exponent(lba),Modulus(lba)	
C^{++}:	**Modex**	**(RC,Result,Base,Exponent,Modulus)**
Asm	**MODEX**	**RC,Result,Base,Exponent,Modulus**
X86:	**none**	

This function calculates Base to the power Exponent (mod Modulus) according to the Montgomery methode[6].
RC (result code) is an address of an integer containing the result-code:
RC = 0 operation succesfully completed
RC = 8 wrong input
RC = 9 calculation cancelled
Result, Base, Exponent and Modulus are all addresses to Lba-fields. The Lba-field Modulus may not be zero. The content of the Lba-field Base must be less than the content of Modulus. If these conditions are not met, then the operation is invalid. The Lba-field Result is set to zero and RC = 8.
During execution of Modex the functions Mgs and Mgm are called and the Lba-field Exponent is corrupted. The Lba-field Modulus is unchanged. The contents of all general registers remains unaffected.

Pseudo Random Number Generator		**Parameter: Result(lba)**
C^{++}:	**Prng**	**(Result)**
Asm	**PRNG**	**Result**
X86:	**geen**	

This function generates a string pseudorandom bits and insert this bits in the Lba-field Result. Pseudo random means that the string of bits is determined and is not completely random. The output, however, complies with the (outdated) standard in accordance with FIPS 140-1 [7].
All dwords of Result are filled with bits except the last dword (the dword with highest value).
The contents of all general registers remains unaffected.

LBA reporting

Report LBA variabele Parameter: Lba-name
C^{++}: Report (Lba-name) ;
Asm: REPORT Lba-name

With the Report statement, the contents of an LBA-variable is shown in the program log.
The form in which it is reported:
"Lba-name" = xx.xxx.xxx.xxx.xxx ....
The numerical contents of the LBA-variable is shown in decimal form, in groups of three numbers, each separated by a separation character. This separator character can be a comma or a point, depending on the language which is active at that moment.

Show messages Parameters: message-nr,integer
C^{++}: Mes (msg-nr,integer) ;
Asm: Msg msg-nr,integer

With the message statement a predefined message with message-number is shown in the program-log. Along with the message the content of an integer is shown (optional). This can be a number with a maximum value of 2^{32} - 1.
With the assembler form the integer may be the specification of a general register.

References:

1. *Intel documentation* (http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf)
2. Theo Kortekaas (http://tonjanee.home.xs4all.nl/SSAdescription.pdf) page. 3 *Classical Method*
3. *Greatest Common Divisor* (https://en.wikipedia.org/wiki/Greatest_common_divisor
4. *Extended Euclidean algorithm* (https://en.wikipedia.org/wiki/Extended_Euclidean_algorthm)
5. Theo Kortekaas *Montgomery multiplication* (http://tonjanee.home.xs4all.nl/Montgomerydescription.pdf) page 6
6. Theo Kortekaas *Modular Exponentiation* (http://tonjanee.home.xs4all.nl/Montgomerydescription.pdf) page 11
7. *FIPS 140-1 Federal Information Processing Standards* (https://en.wikipedia.org/wiki/FIPS_140)

Author: Theo Kortekaas
Email: t.kortekaas@xs4all.nl
Web-pages: http://tonjanee.home.xs4all.nl