20 Math Module (math.hhf)

The HLA math library module provides several large integer arithmetic/logical, trigonometric, and logarithmic routines that extend those provided directly in the CPU and FPU. Note that many of the transcendental functions place strict limits on the values of their parameters. See a reasonable math text or the Intel documentation for details.

A Note About the FPU: The Standard Library code (and the math module in particular) makes use of the FPU when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

20.1 The Math Module

To use the math functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "math.hhf" )
```

or

```
#include( "stdlib.hhf" )
```

20.2 Math Data Types

The math module of the HLA standard library works with seven different data types: 64-bit integers (signed or unsigned), 128-bit integers (signed or unsigned), and real values (32-bit, 64-bit, and 80 bits). Each function specifies the operand data types it expects to use.

20.3 64-Bit Arithmetic and Logical Operations

The HLA Standard Library provides a complete set of arithmetic and logical operations for 64-bit integers. Extended precision arithmetic (especially 64-bit) is fairly straight forward and an in-line coding of most of these functions will generally be faster than calling these functions. For example, a full 64-bit extended precision addition requires about the same number of instructions as it takes to simply pass the parameters to the *math.addq* routine. Therefore, do not call these routines if performance is important, use in-line code instead.

Another reason (beyond the procedure call overhead) that these procedures are slower than the in-line code is that the standard extended precision add sequence does not set the zero flag properly; these procedures have to execute several additional instructions to preserve the carry, sign, and overflow flags as well as properly set the zero flag. So, for example, if you don't use the value of the zero flag upon return, all this extra work goes to waste.

These procedures are convenient to use and are perfectly acceptable when performance is not an issue. Another advantage is that these routines work memory to memory and don't disturb the values in any registers; and also, these routines use a "three-address" form that allows a different destination address (i.e., the destination does not have to be the same as one of the source operands. Of course, as already mentioned, these functions tend to set the flags the same way the basic machine instructions would set them, a big advantage if you are testing the flags after extended-precision arithmetic operations.

```
math.addq( left:qword; right:qword; var dest:qword );
```

This routine adds two quad-word 64-bit integer values and stores the result in a 64-bit destination memory location. The values may be signed or unsigned. This routine computes the following:

```
dest := left + right;
```

This function sets the 80x86 flags exactly the same way that the standard ADD instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags denote unsigned or signed overflow (respectively).

```
HLA high-level calling sequence example:
```

```
    // Compute Sum64 := i64 + j64:

    math.addq( i64, j64, Sum64 );
```

HLA low-level calling sequence example:

```
    // Compute Sum64 := i64 + j64:

    push((type dword i64[4]));
    push((type dword i64));
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Sum64 );
    push( eax );
    call math.addq;
```

math.subq(left:qword; right:qword; var dest:qword);

This function subtracts two 64-bit integer values and stores their difference in *dest*. The values may be signed or unsigned. This function computes the following:

```
dest := left - right;
```

This function sets the 80x86 flags exactly the same way that the standard SUB instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags denote unsigned or signed overflow (respectively).

HLA high-level calling sequence example:

```
    // Compute Dif64 := i64 - j64:

    math.subq( i64, j64, Dif64 );
```

HLA low-level calling sequence example:

```
    // Compute Dif64 := i64 - j64:

    push((type dword i64[4]));
    push((type dword i64));
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Dif64 );
    push( eax );
    call math.subq;
```

math.divq(left:qword; right:qword; var dest:qword);

This routine divides one unsigned 64-bit value by another. It computes the following:

```
dest := left div right;
```

Since the 80x86 flags don't contain useful values after the execution of the DIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 64-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

Also note that this function does not automatically compute the remainder (as the x86 DIV instruction does). The math module provides the *math.modq* function if you need to compute the remainder of the division of two 64-bit values.

```
HLA high-level calling sequence example:

  // Compute Quo64 := i64 div j64:

  math.divq( i64, j64, Quo64 );


HLA low-level calling sequence example:

  // Compute Quo64 := i64 div j64:

  push((type dword i64[4]));
  push((type dword i64));
  push((type dword j64[4]));
  push((type dword j64));
  lea( eax, Quo64 );
  push( eax );
  call math.divq;
```

`math.idivq( left:qword; right:qword; var dest:qword );`

This routine divides one signed 64-bit value by another. It computes the following:

```
dest := left idiv right;
```

Since the 80x86 flags don't contain useful values after the execution of the IDIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 64-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

Also note that this function does not automatically compute the remainder (as the x86 IDIV instruction does). The math module provides the *math.modq* function if you need to compute the remainder of the division of two 64-bit values.

```
HLA high-level calling sequence example:

  // Compute Quo64 := i64 idiv j64:

  math.idivq( i64, j64, Quo64 );


HLA low-level calling sequence example:

  // Compute Quo64 := i64 idiv j64:

  push((type dword i64[4]));
  push((type dword i64));
  push((type dword j64[4]));
  push((type dword j64));
  lea( eax, Quo64 );
  push( eax );
```

```
    call math.idivq;
```

math.modq(left:qword; right:qword; var dest:qword);

This routine divides one unsigned 64-bit value by another and stores the remainder into a destination variable. It computes the following:

```
dest := left & right;// Unsigned modulo
```

Since the 80x86 flags don't contain useful values after the execution of the DIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 64-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

HLA high-level calling sequence example:

```
  // Compute Rem64 := i64 % j64:

  math.modq( i64, j64, Rem64 );
```

HLA low-level calling sequence example:

```
  // Compute Rem64 := i64 % j64:

  push((type dword i64[4]));
  push((type dword i64));
  push((type dword j64[4]));
  push((type dword j64));
  lea( eax, Rem64 );
  push( eax );
  call math.modq;
```

math.imodq(left:qword; right:qword; var dest:qword);

This routine divides one signed 64-bit value by another and stores the remainder into a destination variable. It computes the following:

```
dest := left % right;// Signed modulo
```

Since the 80x86 flags don't contain useful values after the execution of the IDIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 64-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

HLA high-level calling sequence example:

```
  // Compute Rem64 := i64 % j64:

  math.imodq( i64, j64, Rem64 );
```

HLA low-level calling sequence example:

```
    // Compute Rem64 := i64 % j64:

    push((type dword i64[4]));
    push((type dword i64));
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Rem64 );
    push( eax );
    call math.imodq;
```

math.mulq(left:qword; right:qword; var dest:qword);

This routine multiplies one unsigned 64-bit value by another and stores the product into a destination variable. It computes the following:

```
dest := left * right;// Unsigned multiplication
```

This function sets the carry and overflow flags if there was an unsigned overflow during the operation.

HLA high-level calling sequence example:

```
    // Compute Prod64 := i64 * j64:

    math.mulq( i64, j64, Prod64 );
```

HLA low-level calling sequence example:

```
    // Compute Prod64 := i64 % j64:

    push((type dword i64[4]));
    push((type dword i64));
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Prod64 );
    push( eax );
    call math.mulq;
```

math.imulq(left:qword; right:qword; var dest:qword);

This routine multiplies one signed 64-bit value by another and stores the signed product into a destination variable. It computes the following:

```
dest := left * right;// Signed multiplication
```

This function sets the carry and overflow flags if there was an unsigned overflow during the operation.

HLA high-level calling sequence example:

```
    // Compute Prod64 := i64 * j64:

    math.imulq( i64, j64, Prod64 );
```

HLA low-level calling sequence example:

```
    // Compute Prod64 := i64 % j64:

    push((type dword i64[4]));
    push((type dword i64));
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Prod64 );
    push( eax );
    call math.imulq;
```

math.negq(source:qword; var dest:qword);

This function negates (two's complement) the source operand and stores the result into the destination operand. It computes the following:

```
dest := -source;
```

This function leaves the 80x86 flags containing the same values one would expect after the execution of the NEG instruction

HLA high-level calling sequence example:

```
    // Compute Neg64 := -i64:

    math.negq( i64, Neg64 );
```

HLA low-level calling sequence example:

```
    // Compute Neg64 := -i64:

    push((type dword i64[4]));
    push((type dword i64));
    lea( eax, Neg64 );
    push( eax );
    call math.negq;
```

math.andq(left:qword; right:qword; var dest:qword);

This routine logically ANDs two quad-word 64-bit integer values. It computes the following:

```
    dest := left & right;// "&" implies bitwise AND operation
```

This routine sets the 80x86 flags exactly the same way that the standard AND instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
    // Compute Dest64 := i64 & j64:

    math.andq( i64, j64, Dest64 );
```

HLA low-level calling sequence example:

```
    // Compute Dest64 := i64 & j64:
```

```
    push((type dword i64[4]));
    push((type dword i64));
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Dest64 );
    push( eax );
    call math.andq;
```

math.orq(left:qword; right:qword; var dest:qword);

This function logically ORs two 64-bit values and stores the result into a 64-bit destination variable. It computes the following:

```
dest := left | right;
```

These routines set the 80x86 flags exactly the same way that the standard OR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
    // Compute Dest64 := i64 | j64:

    math.orq( i64, j64, Dest64 );
```

HLA low-level calling sequence example:

```
    // Compute Dest64 := i64 | j64:

    push((type dword i64[4]));
    push((type dword i64));
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Dest64 );
    push( eax );
    call math.orq;
```

math.xorq(left:qword; right:qword; var dest:qword);

This function logically XORx two 64-values and stores the result into a 64-bit variable. It computes the following:

```
dest := left ^ right;
```

This function sets the 80x86 flags exactly the same way that the standard XOR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
    // Compute Dest64 := i64 ^ j64:

    math.xorq( i64, j64, Dest64 );
```

HLA low-level calling sequence example:

```
    // Compute Dest64 := i64 ^ j64:
```

```
    push((type dword i64[4]));
    push((type dword i64));
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Dest64 );
    push( eax );
    call math.xorq;
```

math.notq(source:qword; var dest:qword);

This function inverts all the bits in the source operand and stores the result into the destination operand. It computes the following:

```
dest := ~source;
```

This function leaves the 80x86 flags containing the same values one would expect after the execution of the NOT instruction. Extended precision NOT is an especially trivial operation to compute manually; you should carefully consider whether you really want to use this function. Consistent flag results is probably the only good reason for using this function.

HLA high-level calling sequence example:

```
    // Compute Neg64 := not(i64):

    math.notq( i64, Not64 );
```

HLA low-level calling sequence example:

```
    // Compute Neg64 := not(i64):

    push((type dword i64[4]));
    push((type dword i64));
    lea( eax, Not64 );
    push( eax );
    call math.notq;
```

math.shlq(count:uns32; source:qword; var dest:qword);

This function logically shifts left a 64-bit value the number of bits specified by the *count* operand. It stores the result into the 64-bit dest operand. It computes the following:

```
dest := source << count;// Logical shift left operation
```

These routines set the 80x86 flags exactly the same way that the standard SHL instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry flag contains the last carry out of the H.O. bit, and the overflow flag is set if the last shift caused a sign change.

HLA high-level calling sequence example:

```
    // Compute Dest64 := j64 << i32:

    math.shlq( i32, j64, Dest64 );
```

HLA low-level calling sequence example:

```
    // Compute Dest64 := j64 << i32:

    push( i32 );
    push((type dword j64[4]));
    push((type dword j64));
    lea( eax, Dest64 );
    push( eax );
    call math.shlq;
```

math.shrq(count:uns32; source:qword; var dest:qword);

This function logically shifts right a 64-bit value the number of bits specified by the *count* operand. It stores the result into the 64-bit dest operand. It computes the following:

```
    dest := source >> count;// Logical shift right operation
```

These routines set the 80x86 flags exactly the same way that the standard SHR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry flag contains the last carry out of the H.O. bit, and the overflow flag is set if the last shift caused a sign change.

HLA high-level calling sequence example:

```
  // Compute Dest64 := j64 >> i32:

  math.shrq( i32, j64, Dest64 );
```

HLA low-level calling sequence example:

```
  // Compute Dest64 := j64 >> i32:

  push( i32 );
  push((type dword j64[4]));
  push((type dword j64));
  lea( eax, Dest64 );
  push( eax );
  call math.shrq;
```

20.4 128-Bit Arithmetic and Logical Operations

The HLA Standard Library provides a complete set of arithmetic and logical operations for 128-bit integers. Extended precision arithmetic is fairly straight forward and an in-line coding of some of these functions will generally be faster than calling these functions

Another reason (beyond the procedure call overhead) that these procedures are slower than the in-line code is that the standard extended precision add sequence does not set the zero flag properly; these procedures have to execute several additional instructions to preserve the carry, sign, and overflow flags as well as properly set the zero flag. So, for example, if you don't use the value of the zero flag upon return, all this extra work goes to waste.

These procedures are convenient to use and are perfectly acceptable when performance is not an issue. Another advantage is that these routines work memory to memory and don't disturb the values in any registers; and also, these routines use a "three-address" form that allows a different destination address (i.e., the destination does not have to be the same as one of the source operands. Of course, as already mentioned, these functions tend to set the flags the same way the basic machine instructions would set them, a big advantage if you are testing the flags after extended-precision arithmetic operations.

math.addl(left:lword; right:lword; var dest:lword);

This routine adds two 128-bit integer values and stores the result in a 128-bit destination memory location. The values may be signed or unsigned. This routine computes the following:

```
dest := left + right;
```

This function sets the 80x86 flags exactly the same way that the standard ADD instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags denote unsigned or signed overflow (respectively).

HLA high-level calling sequence example:

```
// Compute Sum128 := i128 + j128:

math.addl( i128, j128, Sum128 );
```

HLA low-level calling sequence example:

```
// Compute Sum128 := i128 + j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Sum128 );
push( eax );
call math.addl;
```

math.subl(left:lword; right:lword; var dest:lword);

This function subtracts two 128-bit integer values and stores their difference in *dest*. The values may be signed or unsigned. This function computes the following:

```
dest := left - right;
```

This function sets the 80x86 flags exactly the same way that the standard SUB instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags denote unsigned or signed overflow (respectively).

HLA high-level calling sequence example:

```
// Compute Dif128 := i128 - j128:

math.subl( i128, j128, Dif128 );
```

HLA low-level calling sequence example:

```
// Compute Dif128 := i128 - j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
```

```
    push((type dword i128));
    push((type dword j128[12]));
    push((type dword j128[8]));
    push((type dword j128[4]));
    push((type dword j128));
    lea( eax, Dif128 );
    push( eax );
    call math.subl;
```

math.divl(left:lword; right:lword; var dest:lword);

This routine divides one unsigned 128-bit value by another. It computes the following:

```
dest := left div right;
```

Since the 80x86 flags don't contain useful values after the execution of the DIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 128-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

Also note that this function does not automatically compute the remainder (as the x86 DIV instruction does). The math module provides the *math.modl* function if you need to compute the remainder of the division of two 128-bit values.

HLA high-level calling sequence example:

```
// Compute Quo128 := i128 div j128:

math.divl( i128, j128, Quo128 );
```

HLA low-level calling sequence example:

```
// Compute Quo128 := i128 div j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Quo128 );
push( eax );
call math.divl;
```

math.idivl(left:lword; right:lword; var dest:lword);

This routine divides one signed 128-bit value by another. It computes the following:

```
dest := left idiv right;
```

Since the 80x86 flags don't contain useful values after the execution of the IDIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 128-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

Also note that this function does not automatically compute the remainder (as the x86 IDIV instruction does). The math module provides the *math.modl* function if you need to compute the remainder of the division of two 128-bit values.

```
HLA high-level calling sequence example:

  // Compute Quo128 := i128 idiv j128:

  math.idivl( i128, j128, Quo128 );


HLA low-level calling sequence example:

  // Compute Quo128 := i128 idiv j128:

  push((type dword i128[12]));
  push((type dword i128[8]));
  push((type dword i128[4]));
  push((type dword i128));
  push((type dword j128[12]));
  push((type dword j128[8]));
  push((type dword j128[4]));
  push((type dword j128));
  lea( eax, Quo128 );
  push( eax );
  call math.idivl;
```

`math.modl( left:lword; right:lword; var dest:lword );`

This routine divides one unsigned 128-bit value by another and stores the remainder into a destination variable. It computes the following:

```
dest := left & right;// Unsigned modulo
```

Since the 80x86 flags don't contain useful values after the execution of the DIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 128-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

```
HLA high-level calling sequence example:

  // Compute Rem128 := i128 % j128:

  math.modl( i128, j128, Rem128 );


HLA low-level calling sequence example:

  // Compute Rem128 := i128 % j128:

  push((type dword i128[12]));
  push((type dword i128[8]));
  push((type dword i128[4]));
  push((type dword i128));
```

```
    push((type dword j128[12]));
    push((type dword j128[8]));
    push((type dword j128[4]));
    push((type dword j128));
    lea( eax, Rem128 );
    push( eax );
    call math.modl;
```

math.imodl(left:lword; right:lword; var dest:lword);

This routine divides one signed 128-bit value by another and stores the remainder into a destination variable. It computes the following:

```
dest := left % right;// Signed modulo
```

Since the 80x86 flags don't contain useful values after the execution of the IDIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 128-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

HLA high-level calling sequence example:

```
    // Compute Rem128 := i128 % j128:

    math.imodl( i128, j128, Rem128 );
```

HLA low-level calling sequence example:

```
    // Compute Rem128 := i128 % j128:

    push((type dword i128[12]));
    push((type dword i128[8]));
    push((type dword i128[4]));
    push((type dword i128));
    push((type dword j128[12]));
    push((type dword j128[8]));
    push((type dword j128[4]));
    push((type dword j128));
    lea( eax, Rem128 );
    push( eax );
    call math.imodl;
```

math.mull(left:lword; right:lword; var dest:lword);

This routine multiplies one unsigned 128-bit value by another and stores the product into a destination variable. It computes the following:

```
dest := left * right;// Unsigned multiplication
```

This function sets the carry and overflow flags if there was an unsigned overflow during the operation.

HLA high-level calling sequence example:

```
    // Compute Prod128 := i128 * j128:
```

```
    math.mull( i128, j128, Prod128 );
```

HLA low-level calling sequence example:

```
    // Compute Prod128 := i128 % j128:

    push((type dword i128[12]));
    push((type dword i128[8]));
    push((type dword i128[4]));
    push((type dword i128));
    push((type dword j128[12]));
    push((type dword j128[8]));
    push((type dword j128[4]));
    push((type dword j128));
    lea( eax, Prod128 );
    push( eax );
    call math.mull;
```

math.imull(left:lword; right:lword; var dest:lword);

This routine multiplies one signed 128-bit value by another and stores the signed product into a destination variable. It computes the following:

```
    dest := left * right;// Signed multiplication
```

This function sets the carry and overflow flags if there was an unsigned overflow during the operation.

HLA high-level calling sequence example:

```
    // Compute Prod128 := i128 * j128:

    math.imull( i128, j128, Prod128 );
```

HLA low-level calling sequence example:

```
    // Compute Prod128 := i128 % j128:

    push((type dword i128[12]));
    push((type dword i128[8]));
    push((type dword i128[4]));
    push((type dword i128));
    push((type dword j128[12]));
    push((type dword j128[8]));
    push((type dword j128[4]));
    push((type dword j128));
    lea( eax, Prod128 );
    push( eax );
    call math.imull;
```

math.negl(source:lword; var dest:lword);

This function negates (two's complement) the source operand and stores the result into the destination operand. It computes the following:

```
    dest := -source;
```

This function leaves the 80x86 flags containing the same values one would expect after the execution of the NEG instruction

```
HLA high-level calling sequence example:

  // Compute Neg128 := -i128:

  math.negl( i128, Neg128 );


HLA low-level calling sequence example:

  // Compute Neg128 := -i128:

  push((type dword i128[12]));
  push((type dword i128[8]));
  push((type dword i128[4]));
  push((type dword i128));
  lea( eax, Neg128 );
  push( eax );
  call math.negl;
```

`math.andl( left:lword; right:lword; var dest:lword );`

This routine logically ANDs two 128-bit values. It computes the following:

```
dest := left & right;// "&" implies bitwise AND operation
```

This routine sets the 80x86 flags exactly the same way that the standard AND instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

```
HLA high-level calling sequence example:

  // Compute Dest128 := i128 & j128:

  math.andl( i128, j128, Dest128 );


HLA low-level calling sequence example:

  // Compute Dest128 := i128 & j128:

  push((type dword i128[12]));
  push((type dword i128[8]));
  push((type dword i128[4]));
  push((type dword i128));
  push((type dword j128[12]));
  push((type dword j128[8]));
  push((type dword j128[4]));
  push((type dword j128));
  lea( eax, Dest128 );
  push( eax );
  call math.andl;
```

math.orl(left:lword; right:lword; var dest:lword);

This function logically ORs two 128-bit values and stores the result into a 128-bit destination variable. It computes the following:

```
dest := left | right;// "|" implies bitwise logical OR
```

This function sets the 80x86 flags exactly the same way that the standard OR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
// Compute Dest128 := i128 | j128:

math.orl( i128, j128, Dest128 );
```

HLA low-level calling sequence example:

```
// Compute Dest128 := i128 | j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Dest128 );
push( eax );
call math.orl;
```

math.xorl(left:lword; right:lword; var dest:lword);

This function logically XORs two 128-values and stores the result into a 128-bit variable. It computes the following:

```
dest := left ^ right;// "^" denotes bitwise exclusive-OR.
```

This function sets the 80x86 flags exactly the same way that the standard XOR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
// Compute Dest128 := i128 ^ j128:

math.xorl( i128, j128, Dest128 );
```

HLA low-level calling sequence example:

```
// Compute Dest128 := i128 ^ j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
```

```
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Dest128 );
push( eax );
call math.xorl;
```

math.notl(source:lword; var dest:lword);

This function inverts all the bits in the source operand and stores the result into the destination operand. It computes the following:

```
dest := ~source;
```

This function leaves the 80x86 flags containing the same values one would expect after the execution of the NOT instruction. Extended precision NOT is an especially trivial operation to compute manually; you should carefully consider whether you really want to use this function. Consistent flag results is probably the only good reason for using this function.

HLA high-level calling sequence example:

```
// Compute Neg128 := not(i128):

math.notl( i128, Not128 );
```

HLA low-level calling sequence example:

```
// Compute Neg128 := not(i128):

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
lea( eax, Not128 );
push( eax );
call math.notl;
```

math.shll(count:uns32; source:lword; var dest:lword);

This function logically shifts left a 128-bit value the number of bits specified by the *count* operand. It stores the result into the 128-bit dest operand. It computes the following:

```
dest := source << count;// Logical shift left operation
```

This function sets the 80x86 flags exactly the same way that the standard SHL instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry flag contains the last carry out of the H.O. bit, and the overflow flag is set if the last shift caused a sign change.

HLA high-level calling sequence example:

```
// Compute Dest128 := j128 << i32:

math.shll( i32, j128, Dest128 );
```

```
HLA low-level calling sequence example:

  // Compute Dest128 := j128 << i32:

  push( i32 );
  push((type dword j128[12]));
  push((type dword j128[8]));
  push((type dword j128[4]));
  push((type dword j128));
  lea( eax, Dest128 );
  push( eax );
  call math.shll;
```

`math.shrl( count:uns32; source:lword; var dest:lword );`

This function logically shifts right a 128-bit value the number of bits specified by the *count* operand. It stores the result into the 128-bit dest operand. It computes the following:

```
dest := source >> count;// Logical shift right operation
```

This function sets the 80x86 flags exactly the same way that the standard SHR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry flag contains the last carry out of the H.O. bit, and the overflow flag is set if the last shift caused a sign change.

```
HLA high-level calling sequence example:

  // Compute Dest128 := j128 >> i32:

  math.shrl( i32, j128, Dest128 );


HLA low-level calling sequence example:

  // Compute Dest128 := j128 >> i32:

  push( i32 );
  push((type dword j128[12]));
  push((type dword j128[8]));
  push((type dword j128[4]));
  push((type dword j128));
  lea( eax, Dest128 );
  push( eax );
  call math.shrl;
```

20.5 Transcendental, Logarithmic, and Other Floating-Point Operations

The HLA Standard Library contains a wide variety of transcendental and logarithmic functions. All of these instructions use the FPU for their computations, they do not use SSE-type floating-point instructions. These functions all assume that the CPU is in floating-point mode (that is, you've not executed any MMX instructions in your program or you haven't executed any MMX instructions since you last executed an EMMS instruction) and that the FPU stack is valid.

Because the x86 FPU supports three different real data types, each of the functions in the HLA Standard Library Math module provide variants that work on single-precision (32-bit), double-precision (64-bit), and extended-precision (80-bit) memory operands. The library also includes a version of each function that operates

on the (80-bit) value currently on the FPU's top of stack. If "*fcn*" represents a specific mathematical function name, then the HLA Standard Library generally provides the following four actual functions:

*fcn*32(r32:real32);// Expects a real32 operand
*fcn*64(r64:real64);// Expects a real64 operand
*fcn*80(r80:real80);// Expects a real80 operand
_fcn(); // Expects a real80 operand on the FPU top of stack

In addition to these four functions, the standard library will also provide a macro, simply named "*fcn*", that overloads these four functions and will automatically select the appropriate function to call based on the number of operands (zero or one) and the type of the operand (real32, real64, or real80).

Some of the functions in the Math module mirror x86 FPU instructions. The purpose of such functions is to handle range reduction and other operations needed to guarantee a correct or most precise result.

All of these functions leave an 80-bit result sitting on the top of the floating-point stack (except the *math.sincos* function, which leaves two values sitting on the FPU stack). In general, you should not count on any more significant bits than the number of bits in the original operand. That is, if you pass a 32-bit or 64-bit value to one of these functions, then you should save the result in a like-sized destination variable. Arithmetic precision is only as good as the original operand(s), so avoid false precision and store the results in appropriately-sized destination variables.

To save space, this document describes each class of functions that compute the same transcendental/ logarithmic value (except for size) in a single section.

```
#macro math.sin; @returns( "st0" );// Overloads the following functions:
procedure math._sin;  @returns( "st0" );
procedure math.sin32( r32: real32 );  @returns( "st0" );
procedure math.sin64( r64: real64 );  @returns( "st0" );
procedure math.sin80( r80: real80 );  @returns( "st0" );
```

These five functions compute the sine of their parameter value. The parameter value must specify an angle in radians.

The *math.sin* function is actually a macro that overloads the remaining four functions. If a *math.sin* invocation doesn't contain any parameters, then this macro expands to a call to the *math._sin* function which computes the sine of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math._sin();* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the sine via the FPU FSIN instruction.

```
HLA high-level calling sequence examples:

  // Compute y := sin(x):

  math.sin32( x32 );
  fstp( y32 );

  math.sin64( x64 );
  fstp( y64 );

  math.sin80( x80 );
  fstp( y80 );

  fld( x80 );
  math._sin();
  fstp( y80 );

  // Using the math.sin macro:
```

```
    math.sin( x32 );
    fstp( y32 );

    math.sin( x64 );
    fstp( y64 );

    math.sin( x80 );
    fstp( y80 );

    fld( x80 );
    math.sin();
    fstp( y80 );
```

HLA low-level calling sequence example:

```
    // Compute y := sin(x):

    push( x32 );
    call math.sin32;
    fstp( y32 );

    push( (type dword x64[4]) );
    push( (type dword x64[0]) );
    call math.sin64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.sin80;
    fstp( y80 );

    fld( x80 );
    call math.sin;
    fstp( y80 );
```

```
#macro math.cos; @returns( "st0" ); // Overloads the following functions:
procedure math._cos;  @returns( "st0" );
procedure math.cos32( r32: real32 );  @returns( "st0" );
procedure math.cos64( r64: real64 );  @returns( "st0" );
procedure math.cos80( r80: real80 );  @returns( "st0" );
```

These five functions compute the cosine of their parameter value. The parameter value must specify an angle in radians.

The *math.cos* function is actually a macro that overloads the remaining four functions. If a *math.cos* invocation doesn't contain any parameters, then this macro expands to a call to the *math._cos* function which computes the cosine of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math._cos();* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the cosine via the FPU FCOS instruction.

HLA high-level calling sequence examples:

```
  // Compute y := cos(x):

  math.cos32( x32 );
  fstp( y32 );

  math.cos64( x64 );
  fstp( y64 );

  math.cos80( x80 );
  fstp( y80 );

  fld( x80 );
  math._cos();
  fstp( y80 );

  // Ucosg the math.cos macro:

  math.cos( x32 );
  fstp( y32 );

  math.cos( x64 );
  fstp( y64 );

  math.cos( x80 );
  fstp( y80 );

  fld( x80 );
  math.cos();
  fstp( y80 );
```

HLA low-level calling sequence example:

```
  // Compute y := cos(x):

  push( x32 );
  call math.cos32;
  fstp( y32 );

  push( (type dword x64[4]) );
  push( (type dword x64[0]) );
  call math.cos64;
  fstp( y64 );

  pushw( 0 );             // Must dword align operand
  push( (type word x80[8]) );
  push( (type dword x80[4]) );
  push( (type dword x80[0]) );
  call math.cos80;
  fstp( y80 );

  fld( x80 );
  call math.cos;
  fstp( y80 );
```

```
#macro math.tan; @returns( "st0" ); // Overloads the following functions:
procedure math._tan;  @returns( "st0" );
procedure math.tan32( r32: real32 );  @returns( "st0" );
procedure math.tan64( r64: real64 );  @returns( "st0" );
procedure math.tan80( r80: real80 );  @returns( "st0" );
```

These five functions compute the tangent of their parameter value. The parameter value must specify an angle in radians.

The *math.tan* function is actually a macro that overloads the remaining four functions. If a *math.tan* invocation doesn't contain any parameters, then this macro expands to a call to the *math._tan* function which computes the tangent of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math._tan();* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the cosine via the FPU FTAN instruction.

HLA high-level calling sequence examples:

```
// Compute y := tan(x):

math.tan32( x32 );
fstp( y32 );

math.tan64( x64 );
fstp( y64 );

math.tan80( x80 );
fstp( y80 );

fld( x80 );
math._tan();
fstp( y80 );

// Utang the math.tan macro:

math.tan( x32 );
fstp( y32 );

math.tan( x64 );
fstp( y64 );

math.tan( x80 );
fstp( y80 );

fld( x80 );
math.tan();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := tan(x):

push( x32 );
call math.tan32;
fstp( y32 );

push( (type dword x64[4]) );
```

```
    push( (type dword x64[0]) );
    call math.tan64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.tan80;
    fstp( y80 );

    fld( x80 );
    call math.tan;
    fstp( y80 );
```

```
#macro math.sincos; // Overloads the following functions:
procedure math._sincos;
procedure math.sincos32( r32: real32 );
procedure math.sincos64( r64: real64 );
procedure math.sincos80( r80: real80 );
```

These five functions compute the sine and cosine of their parameter value. The parameter value must specify an angle in radians.

The *math.sincos* function is actually a macro that overloads the remaining four functions. If a *math.sincos* invocation doesn't contain any parameters, then this macro expands to a call to the *math._sincos* function which computes the sine and cosine of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math._sincos();* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the sine and cosine via the FPU FSINCOS instruction.

This function computes two return results (the sine and the cosine) and leaves the two values on the FPU stack at ST0 and ST1.

```
HLA high-level calling sequence examples:

    // Compute y := sincos(x):

    math.sincos32( x32 );
    fstp( y32 );

    math.sincos64( x64 );
    fstp( y64 );

    math.sincos80( x80 );
    fstp( y80 );

    fld( x80 );
    math._sincos();
    fstp( y80 );

    // Usincosg the math.sincos macro:

    math.sincos( x32 );
    fstp( y32 );

    math.sincos( x64 );
```

```
    fstp( y64 );

    math.sincos( x80 );
    fstp( y80 );

    fld( x80 );
    math.sincos();
    fstp( y80 );
```

HLA low-level calling sequence example:

```
    // Compute y := sincos(x):

    push( x32 );
    call math.sincos32;
    fstp( y32 );

    push( (type dword x64[4]) );
    push( (type dword x64[0]) );
    call math.sincos64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.sincos80;
    fstp( y80 );

    fld( x80 );
    call math.sincos;
    fstp( y80 );
```

```
#macro math.atan; @returns( "st0" ); // Overloads the following functions:
procedure math._atan;  @returns( "st0" );

procedure math.atan32( r32: real32 );  @returns( "st0" );
procedure math.atan64( r64: real64 );  @returns( "st0" );
procedure math.atan80( r80: real80 );  @returns( "st0" );
```

These five functions compute the arc tangent of their parameter value. The parameter value must specify an angle in radians.

The *math.atan* function is actually a macro that overloads the remaining four functions. If a *math.atan* invocation doesn't contain any parameters, then this macro expands to a call to the *math._atan* function which computes the arc tangent of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math._atan();* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the satandard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the cosine via the FPU FATAN instruction.

HLA high-level calling sequence examples:

```
    // Compute y := atan(x):

    math.atan32( x32 );
```

```
    fstp( y32 );

    math.atan64( x64 );
    fstp( y64 );

    math.atan80( x80 );
    fstp( y80 );

    fld( x80 );
    math._atan();
    fstp( y80 );

    // Uatang the math.atan macro:

    math.atan( x32 );
    fstp( y32 );

    math.atan( x64 );
    fstp( y64 );

    math.atan( x80 );
    fstp( y80 );

    fld( x80 );
    math.atan();
    fstp( y80 );
```

HLA low-level calling sequence example:

```
    // Compute y := atan(x):

    push( x32 );
    call math.atan32;
    fstp( y32 );

    push( (type dword x64[4]) );
    push( (type dword x64[0]) );
    call math.atan64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.atan80;
    fstp( y80 );

    fld( x80 );
    call math.atan;
    fstp( y80 );
```

```
#macro math.cot; @returns( "st0" ); // Overloads the following functions:
procedure math._cot;  @returns( "st0" );
procedure math.cot32( r32: real32 );  @returns( "st0" );
```

```
procedure math.cot64( r64: real64 );  @returns( "st0" );
procedure math.cot80( r80: real80 );  @returns( "st0" );
```

These five functions compute the cotangent (1/tan) of their parameter value. The parameter value must specify an angle in radians.

The *math.cot* function is actually a macro that overloads the remaining four functions. If a *math.cot* invocation doesn't contain any parameters, then this macro expands to a call to the *math._cot* function which computes the cotangent of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math._cot();* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := cot(x):

math.cot32( x32 );
fstp( y32 );

math.cot64( x64 );
fstp( y64 );

math.cot80( x80 );
fstp( y80 );

fld( x80 );
math._cot();
fstp( y80 );

// Using the math.cot macro:

math.cot( x32 );
fstp( y32 );

math.cot( x64 );
fstp( y64 );

math.cot( x80 );
fstp( y80 );

fld( x80 );
math.cot();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := cot(x):

push( x32 );
call math.cot32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.cot64;
fstp( y64 );

pushw( 0 );             // Must dword align operand
```

```
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.cot80;
    fstp( y80 );

    fld( x80 );
    call math.cot;
    fstp( y80 );
```

```
#macro math.csc // Macro that overloads the following four functions:
procedure math._csc;  @returns( "st0" );
procedure math.csc32( r32:real32 ); @returns( "st0" );
procedure math.csc64( r64: real64 );  @returns( "st0" );
procedure math.csc80( r80: real80 );  @returns( "st0" );
```

These five functions compute the cosecant (1/sin) of their parameter value. The parameter value must specify an angle in radians.

The *math.csc* function is actually a macro that overloads the remaining four functions. If a *math.csc* invocation doesn't contain any parameters, then this macro expands to a call to the *math._csc* function which computes the cosecant of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math._csc()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

```
HLA high-level calling sequence examples:

    // Compute y := csc(x):

    math.csc32( x32 );
    fstp( y32 );

    math.csc64( x64 );
    fstp( y64 );

    math.csc80( x80 );
    fstp( y80 );

    fld( x80 );
    math._csc();
    fstp( y80 );

    // Using the math.csc macro:

    math.csc( x32 );
    fstp( y32 );

    math.csc( x64 );
    fstp( y64 );

    math.csc( x80 );
    fstp( y80 );

    fld( x80 );
    math.csc();
    fstp( y80 );
```

```
HLA low-level calling sequence example:

  // Compute y := csc(x):

  push( x32 );
  call math.csc32;
  fstp( y32 );

  push( (type dword x64[4]) );
  push( (type dword x64[0]) );
  call math.csc64;
  fstp( y64 );

  pushw( 0 );             // Must dword align operand
  push( (type word x80[8]) );
  push( (type dword x80[4]) );
  push( (type dword x80[0]) );
  call math.csc80;
  fstp( y80 );

  fld( x80 );
  call math.csc;
  fstp( y80 );
```

```
#macro math.sec;// Macro that overloads the following four functions:
procedure  math._sec;  @returns( "st0" );
procedure  math.sec32( r32:real32 ); @returns( "st0" );
procedure  math.sec64( r64: real64 );  @returns( "st0" );
procedure  math.sec80( r80: real80 );  @returns( "st0" );
```

These five functions compute the secant (1/cos) of their parameter value. The parameter value must specify an angle in radians.

The *math.sec* function is actually a macro that overloads the remaining four functions. If a *math.sec* invocation doesn't contain any parameters, then this macro expands to a call to the *math._sec* function which computes the secant of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math._sec();*" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

```
HLA high-level calling sequence examples:

  // Compute y := sec(x):

  math.sec32( x32 );
  fstp( y32 );

  math.sec64( x64 );
  fstp( y64 );

  math.sec80( x80 );
  fstp( y80 );

  fld( x80 );
  math._sec();
  fstp( y80 );
```

```
    // Using the math.sec macro:

    math.sec( x32 );
    fstp( y32 );

    math.sec( x64 );
    fstp( y64 );

    math.sec( x80 );
    fstp( y80 );

    fld( x80 );
    math.sec();
    fstp( y80 );
```

HLA low-level calling sequence example:

```
    // Compute y := sec(x):

    push( x32 );
    call math.sec32;
    fstp( y32 );

    push( (type dword x64[4]) );
    push( (type dword x64[0]) );
    call math.sec64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.sec80;
    fstp( y80 );

    fld( x80 );
    call math.sec;
    fstp( y80 );
```

```
#macro math.asin // Macro that overloads the following four functions:
procedure  math._asin;  @returns( "st0" );
procedure  math.asin32( r32:real32 ); @returns( "st0" );
procedure  math.asin64( r64: real64 );  @returns( "st0" );
procedure  math.asin80( r80: real80 );  @returns( "st0" );
```

These five functions compute the arc sine ($\sin^{-1}$) of their parameter value. They return an angle in radians.

The *math.asin* function is actually a macro that overloads the remaining four functions. If a *math.asin* invocation doesn't contain any parameters, then this macro expands to a call to the *math._asin* function which computes the arc sin of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._asin();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
    // Compute y := asin(x):

    math.asin32( x32 );
    fstp( y32 );

    math.asin64( x64 );
    fstp( y64 );

    math.asin80( x80 );
    fstp( y80 );

    fld( x80 );
    math._asin();
    fstp( y80 );

    // Using the math.asin macro:

    math.asin( x32 );
    fstp( y32 );

    math.asin( x64 );
    fstp( y64 );

    math.asin( x80 );
    fstp( y80 );

    fld( x80 );
    math.asin();
    fstp( y80 );
```

HLA low-level calling sequence example:

```
    // Compute y := asin(x):

    push( x32 );
    call math.asin32;
    fstp( y32 );

    push( (type dword x64[4]) );
    push( (type dword x64[0]) );
    call math.asin64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.asin80;
    fstp( y80 );

    fld( x80 );
    call math.asin;
    fstp( y80 );
```

```
#macro math.acos // Macro to overload the following four functions:
procedure  math._acos;  @returns( "st0" );
```

```
procedure  math.acos32( r32:real32 ); @returns( "st0" );
procedure  math.acos64( r64: real64 );  @returns( "st0" );
procedure  math.acos80( r80: real80 );  @returns( "st0" );
```

These five functions compute the arc cosine ($\cos^{-1}$) of their parameter value. They return an angle in radians.

The *math.acos* function is actually a macro that overloads the remaining four functions. If a *math.acos* invocation doesn't contain any parameters, then this macro expands to a call to the *math._acos* function which computes the arc cosine of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._acos();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := acos(x):

math.acos32( x32 );
fstp( y32 );

math.acos64( x64 );
fstp( y64 );

math.acos80( x80 );
fstp( y80 );

fld( x80 );
math._acos();
fstp( y80 );

// Using the math.acos macro:

math.acos( x32 );
fstp( y32 );

math.acos( x64 );
fstp( y64 );

math.acos( x80 );
fstp( y80 );

fld( x80 );
math.acos();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := acos(x):

push( x32 );
call math.acos32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.acos64;
fstp( y64 );
```

```
    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.acos80;
    fstp( y80 );

    fld( x80 );
    call math.acos;
    fstp( y80 );
```

```
#macro math.acot// Macro that overloads the following four functions:
procedure math._acot;  @returns( "st0" );
procedure math.acot32( r32:real32 ); @returns( "st0" );
procedure math.acot64( r64: real64 );  @returns( "st0" );
procedure math.acot80( r80: real80 );  @returns( "st0" );
```

These five functions compute the arc cotangent (cot^{-1}) of their parameter value. They return an angle in radians.

The *math.acot* function is actually a macro that overloads the remaining four functions. If a *math.acot* invocation doesn't contain any parameters, then this macro expands to a call to the *math._acot* function which computes the arc cotangent of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._acot();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

```
HLA high-level calling sequence examples:

    // Compute y := acot(x):

    math.acot32( x32 );
    fstp( y32 );

    math.acot64( x64 );
    fstp( y64 );

    math.acot80( x80 );
    fstp( y80 );

    fld( x80 );
    math._acot();
    fstp( y80 );

    // Using the math.acot macro:

    math.acot( x32 );
    fstp( y32 );

    math.acot( x64 );
    fstp( y64 );

    math.acot( x80 );
    fstp( y80 );

    fld( x80 );
    math.acot();
    fstp( y80 );
```

```
HLA low-level calling sequence example:

  // Compute y := acot(x):

  push( x32 );
  call math.acot32;
  fstp( y32 );

  push( (type dword x64[4]) );
  push( (type dword x64[0]) );
  call math.acot64;
  fstp( y64 );

  pushw( 0 );             // Must dword align operand
  push( (type word x80[8]) );
  push( (type dword x80[4]) );
  push( (type dword x80[0]) );
  call math.acot80;
  fstp( y80 );

  fld( x80 );
  call math.acot;
  fstp( y80 );
```

```
#macro math.acsc// Overload macro that expands to one of the following:
procedure math._acsc;  @returns( "st0" );
procedure math.acsc32( r32:real32 ); @returns( "st0" );
procedure math.acsc64( r64: real64 );  @returns( "st0" );
procedure math.acsc80( r80: real80 );  @returns( "st0" );
```

These five functions compute the arc cosecant (csc^{-1}) of their parameter value. They return an angle in radians.

The *math.acsc* function is actually a macro that overloads the remaining four functions. If a *math.acsc* invocation doesn't contain any parameters, then this macro expands to a call to the *math._acsc* function which computes the arc cosecant of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._acsc();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

```
HLA high-level calling sequence examples:

  // Compute y := acsc(x):

  math.acsc32( x32 );
  fstp( y32 );

  math.acsc64( x64 );
  fstp( y64 );

  math.acsc80( x80 );
  fstp( y80 );

  fld( x80 );
  math._acsc();
  fstp( y80 );
```

```
    // Using the math.acsc macro:

    math.acsc( x32 );
    fstp( y32 );

    math.acsc( x64 );
    fstp( y64 );

    math.acsc( x80 );
    fstp( y80 );

    fld( x80 );
    math.acsc();
    fstp( y80 );
```

HLA low-level calling sequence example:

```
    // Compute y := acsc(x):

    push( x32 );
    call math.acsc32;
    fstp( y32 );

    push( (type dword x64[4]) );
    push( (type dword x64[0]) );
    call math.acsc64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.acsc80;
    fstp( y80 );

    fld( x80 );
    call math.acsc;
    fstp( y80 );
```

```
#macro math.asec // Overloading macro that expands to one of:
procedure math._asec;  @returns( "st0" );
procedure math.asec32( r32:real32 ); @returns( "st0" );
procedure math.asec64( r64: real64 );  @returns( "st0" );
procedure math.asec80( r80: real80 );  @returns( "st0" );
```

These five functions compute the arc secant (sec^{-1}) of their parameter value. They return an angle in radians.

The *math.asec* function is actually a macro that overloads the remaining four functions. If a *math.asec* invocation doesn't contain any parameters, then this macro expands to a call to the *math._asec* function which computes the arc secant of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._asec();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
  // Compute y := asec(x):

  math.asec32( x32 );
  fstp( y32 );

  math.asec64( x64 );
  fstp( y64 );

  math.asec80( x80 );
  fstp( y80 );

  fld( x80 );
  math._asec();
  fstp( y80 );

  // Using the math.asec macro:

  math.asec( x32 );
  fstp( y32 );

  math.asec( x64 );
  fstp( y64 );

  math.asec( x80 );
  fstp( y80 );

  fld( x80 );
  math.asec();
  fstp( y80 );
```

HLA low-level calling sequence example:

```
  // Compute y := asec(x):

  push( x32 );
  call math.asec32;
  fstp( y32 );

  push( (type dword x64[4]) );
  push( (type dword x64[0]) );
  call math.asec64;
  fstp( y64 );

  pushw( 0 );             // Must dword align operand
  push( (type word x80[8]) );
  push( (type dword x80[4]) );
  push( (type dword x80[0]) );
  call math.asec80;
  fstp( y80 );

  fld( x80 );
  call math.asec;
  fstp( y80 );
```

```
#macro math.twoToX // Macro that overloads the following functions:
procedure math._twoToX;  @returns( "st0" );
procedure math.twoToX32( r32: real32 );  @returns( "st0" );
procedure math.twoToX64( r64: real64 );  @returns( "st0" );
procedure math.twoToX80( r80: real80 );  @returns( "st0" );
```

These five functions compute 2^x of their parameter value (which is the value of x).

The *math.twoToX* function is actually a macro that overloads the remaining four functions. If a *math.twoToX* invocation doesn't contain any parameters, then this macro expands to a call to the *math._twoToX* function which computes 2^x of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._twoToX();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := 2**x:

math.twoToX32( x32 );
fstp( y32 );

math.twoToX64( x64 );
fstp( y64 );

math.twoToX80( x80 );
fstp( y80 );

fld( x80 );
math._twoToX();
fstp( y80 );

// Using the math.twoToX macro:

math.twoToX( x32 );
fstp( y32 );

math.twoToX( x64 );
fstp( y64 );

math.twoToX( x80 );
fstp( y80 );

fld( x80 );
math.twoToX();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := twoToX(x):

push( x32 );
call math.twoToX32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.twoToX64;
fstp( y64 );
```

```
    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.twoToX80;
    fstp( y80 );

    fld( x80 );
    call math.twoToX;
    fstp( y80 );
```

```
#macro math.TenToX // Overloads the following functions:
procedure math._tenToX;  @returns( "st0" );
procedure math.tenToX32( r32:real32 ); @returns( "st0" );
procedure math.tenToX64( r64: real64 );  @returns( "st0" );
procedure math.tenToX80( r80: real80 );  @returns( "st0" );
```

These five functions compute 10^x of their parameter value (which is the value of x).

The *math.tenToX* function is actually a macro that overloads the remaining four functions. If a *math.tenToX* invocation doesn't contain any parameters, then this macro expands to a call to the *math._tenToX* function which computes the 10^x of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._tenToX();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
    // Compute y := 10**x:

    math.tenToX32( x32 );
    fstp( y32 );

    math.tenToX64( x64 );
    fstp( y64 );

    math.tenToX80( x80 );
    fstp( y80 );

    fld( x80 );
    math._tenToX();
    fstp( y80 );

    // Using the math.tenToX macro:

    math.tenToX( x32 );
    fstp( y32 );

    math.tenToX( x64 );
    fstp( y64 );

    math.tenToX( x80 );
    fstp( y80 );

    fld( x80 );
    math.tenToX();
    fstp( y80 );
```

```
HLA low-level calling sequence example:

  // Compute y := tenToX(x):

  push( x32 );
  call math.tenToX32;
  fstp( y32 );

  push( (type dword x64[4]) );
  push( (type dword x64[0]) );
  call math.tenToX64;
  fstp( y64 );

  pushw( 0 );             // Must dword align operand
  push( (type word x80[8]) );
  push( (type dword x80[4]) );
  push( (type dword x80[0]) );
  call math.tenToX80;
  fstp( y80 );

  fld( x80 );
  call math.tenToX;
  fstp( y80 );
```

```
#macro math.exp // Overloads the following functions:
procedure math._exp;  @returns( "st0" );
procedure math.exp32( r32:real32 ); @returns( "st0" );
procedure math.exp64( r64: real64 );  @returns( "st0" );
procedure math.exp80( r80: real80 );  @returns( "st0" );
```

These five functions compute e^x of their parameter value (which is the value of x).

The *math.exp* function is actually a macro that overloads the remaining four functions. If a *math.exp* invocation doesn't contain any parameters, then this macro expands to a call to the *math._exp* function which computes e^x of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._exp();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

```
HLA high-level calling sequence examples:

  // Compute y := e**x:

  math.exp32( x32 );
  fstp( y32 );

  math.exp64( x64 );
  fstp( y64 );

  math.exp80( x80 );
  fstp( y80 );

  fld( x80 );
  math._exp();
  fstp( y80 );

  // Using the math.exp macro:
```

```
    math.exp( x32 );
    fstp( y32 );

    math.exp( x64 );
    fstp( y64 );

    math.exp( x80 );
    fstp( y80 );

    fld( x80 );
    math.exp();
    fstp( y80 );
```

HLA low-level calling sequence example:

```
    // Compute y := e**x:

    push( x32 );
    call math.exp32;
    fstp( y32 );

    push( (type dword x64[4]) );
    push( (type dword x64[0]) );
    call math.exp64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.exp80;
    fstp( y80 );

    fld( x80 );
    call math.exp;
    fstp( y80 );
```

```
#macro math.ytoX // Macro that overloads the following functions:
procedure math._yToX;        // Y is at ST1, X is at ST0.
procedure math.yToX32( y32Var, x32Var ); @returns( "st0" );
procedure math.yToX64( y64Var, x64Var ); @returns( "st0" );
procedure math.yToX80( y80Var, x80Var ); @returns( "st0" );
```

These five functions compute Y^X of their parameter values (which are the values of *y* and *x*).

The *math.yToX* function is actually a macro that overloads the remaining four functions. If a *math.yToX* invocation doesn't contain any parameters, then this macro expands to a call to the *math._yToX* function which computes Y^X using the values on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._yToX();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

Because this function uses logarithms to compute its result, the *y* argument must be a non-negative value.

HLA high-level calling sequence examples:

```
    // Compute z := y**x:
```

```
  math.yToX32( y32, x32 );
  fstp( z32 );

  math.yToX64( y64, x64 );
  fstp( z64 );

  math.yToX80( y80, x80 );
  fstp( z80 );

  fld( y80 );
  fld( x80 );
  math._yToX();
  fstp( z80 );

  // Using the math.yToX macro:

  math.yToX( y32, x32 );
  fstp( z32 );

  math.yToX( y64, x64 );
  fstp( z64 );

  math.yToX( y80, x80 );
  fstp( z80 );

  fld( y80 );
  fld( x80 );
  math.yToX();
  fstp( z80 );
```

HLA low-level calling sequence example:

```
  // Compute z := yToX( y, x ):

  push( y32 );
  push( x32 );
  call math.yToX32;
  fstp( z32 );

  push( (type dword x64[4]) );
  push( (type dword x64[0]) );
  push( (type dword y64[4]) );
  push( (type dword y64[0]) );
  call math.yToX64;
  fstp( z64 );

  pushw( 0 );            // Must dword align operand
  push( (type word y80[8]) );
  push( (type dword y80[4]) );
  push( (type dword y80[0]) );
  pushw( 0 );            // Must dword align operand
  push( (type word x80[8]) );
  push( (type dword x80[4]) );
  push( (type dword x80[0]) );
  call math.yToX80;
  fstp( z80 );

  fld( y80 );
```

```
    fld( x80 );
    call math.yToX;
    fstp( z80 );
```

```
#macro math.log // Overloads the following functions:
procedure math._log;  @returns( "st0" );
procedure math.log32( r32: real32 );  @returns( "st0" );
procedure math.log64( r64: real64 );  @returns( "st0" );
procedure math.log80( r80: real80 );  @returns( "st0" );
```

These five functions compute $\log_{10}(x)$ of their parameter value (which is the value of *x*).

The *math.log* function is actually a macro that overloads the remaining four functions. If a *math.log* invocation doesn't contain any parameters, then this macro expands to a call to the *math._log* function which computes the base 10 log of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._log();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
    // Compute y := log(x):

    math.log32( x32 );
    fstp( y32 );

    math.log64( x64 );
    fstp( y64 );

    math.log80( x80 );
    fstp( y80 );

    fld( x80 );
    math._log();
    fstp( y80 );

    // Using the math.log macro:

    math.log( x32 );
    fstp( y32 );

    math.log( x64 );
    fstp( y64 );

    math.log( x80 );
    fstp( y80 );

    fld( x80 );
    math.log();
    fstp( y80 );
```

HLA low-level calling sequence example:

```
    // Compute y := log(x):

    push( x32 );
    call math.log32;
```

```
    fstp( y32 );

    push( (type dword x64[4]) );
    push( (type dword x64[0]) );
    call math.log64;
    fstp( y64 );

    pushw( 0 );             // Must dword align operand
    push( (type word x80[8]) );
    push( (type dword x80[4]) );
    push( (type dword x80[0]) );
    call math.log80;
    fstp( y80 );

    fld( x80 );
    call math.log;
    fstp( y80 );
```

```
#macro math.ln // Overloads the following functions:
procedure math._ln;  @returns( "st0" );
procedure math.ln32( r32: real32 );  @returns( "st0" );
procedure math.ln64( r64: real64 );  @returns( "st0" );
procedure math.ln80( r80: real80 );  @returns( "st0" );
```

ln(x) [$\log_e(x)$]

These five functions compute $\log_e(x)$ of their parameter value (which is the value of *x*).

The *math.ln* function is actually a macro that overloads the remaining four functions. If a *math.ln* invocation doesn't contain any parameters, then this macro expands to a call to the *math._ln* function which computes the base e log of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The *"math._ln();"* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

```
  HLA high-level calling sequence examples:

    // Compute y := ln(x):

    math.ln32( x32 );
    fstp( y32 );

    math.ln64( x64 );
    fstp( y64 );

    math.ln80( x80 );
    fstp( y80 );

    fld( x80 );
    math._ln();
    fstp( y80 );

    // Using the math.ln macro:

    math.ln( x32 );
    fstp( y32 );

    math.ln( x64 );
    fstp( y64 );
```

```
  math.ln( x80 );
  fstp( y80 );

  fld( x80 );
  math.ln();
  fstp( y80 );
```

HLA low-level calling sequence example:

```
  // Compute y := ln(x):

  push( x32 );
  call math.ln32;
  fstp( y32 );

  push( (type dword x64[4]) );
  push( (type dword x64[0]) );
  call math.ln64;
  fstp( y64 );

  pushw( 0 );             // Must dword align operand
  push( (type word x80[8]) );
  push( (type dword x80[4]) );
  push( (type dword x80[0]) );
  call math.ln80;
  fstp( y80 );

  fld( x80 );
  call math.ln;
  fstp( y80 );
```