Instructions for asmlib

A multi-platform library of highly optimized functions for C and C++.

Version 2.21. 2011-08-21
By Agner Fog. © 2003-2011. GNU General Public License.

Contents

1 Introduction

Asmlib is a function library to call from C or C++ for all x86 and x86-64 platforms. It is not intended to be a complete function library, but contains mainly:

- Faster versions of several standard C functions.

- Useful functions that are difficult to find elsewhere.

- Functions that are best written in assembly language.

These functions are written in assembly language for the sake of optimizing speed. Many of the functions have multiple branches for different instruction sets, such as SSE2, SSE4.2, etc. These functions will automatically detect which instruction set is supported by the computer it is running on and select the optimal branch.

This library is also intended as a showcase to illustrate the optimization methods explained in my optimization manuals and an example of how to make a cross-platform function library.

The latest version of asmlib is always available at www.agner.org/optimize.

1.1 Support for multiple platforms

Different operating systems and compilers use different object file formats and different calling conventions. Asmlib is available in different versions, supporting 32-bit and 64-bit Windows, Linux, BSD and Mac running Intel or AMD x86 and x86-64 family processors. The following object file formats are supported: OMF, COFF, ELF, Mach-O. Almost all C and C++ compilers for these platforms support at least one of these object file formats. Processors running other instruction sets, such as Itanium or Power-PC are not supported.

Version 2.20 and later of asmlib is written in the NASM/YASM dialect of assembly syntax because the NASM and YASM assemblers support multiple platforms. The position-independent 32-bit versions can only be built with the YASM assembler.

See page 18 for a list of asmlib versions for different platforms.

1.2 Calling from other programming languages

Asmlib is designed for calling from C and C++. Calling the library functions from other programming languages can be quite difficult. It is necessary to use dynamic linking (DLL) under Windows if the compiler doesn't support static linking or if the static link library is incompatible.

A DLL under 32-bit Windows uses the `stdcall` calling convention by default. Most of the functions in asmlib have no `stdcall` version. See the description of each function.

Strings and arrays are represented differently in other programming languages. It is not possible to use string and memory functions in other programming languages unless there is a feature for linking with C. See the manual for the specific compiler to see how to link with C code.

Linking with Java is particularly difficult. It is necessary to use the Java Native Interface (JNI).

1.3 Position-independent code

Shared objects (*.so) in 32-bit Linux, BSD and Mac require position-independent code. Special position-independent versions of asmlib are available for building shared objects. Not all functions in asmlib are available in the position-independent versions. See the description of each function.

1.4 Overriding standard function libraries

The standard libraries that are included with common compilers are not always fully optimized and may not use the latest instruction set extensions. It is sometimes possible to improve the speed of a program simply by using a faster function library.

You may use a profiler to measure how much time a program spends in each function. If a significant amount of time is spent executing library functions then it may be possible to improve performance by using faster versions of these functions.

There are two ways to replace a standard function with a faster version:

1. Use a different name for the faster version of the function. For example call `A_memcpy` instead of `memcpy`. Asmlib have functions with `A_` prefix as replacements for several standard functions.

2. Asmlib is available in an "override" version that uses the same function names as the standard libraries. If two function libraries contain the same function name then the linker will take the function from the library that is linked first.

If you use the "override" version of the asmlib library then you don't have to modify the program source code. All you have to do is to link the appropriate version of asmlib into your project. See page 18 for available versions of asmlib. If standard libraries are included explicitly in your project then make sure asmlib comes before the standard libraries.

The override method will replace not only the function calls you write in the source code, but also function calls generated implicitly by the compiler as well as calls from other libraries. For example, the compiler may call `memcpy` when copying a big object. The override version of asmlib accepts function names both with and without the `A_` prefix.

The override method sometimes fails to call the asmlib function because the compiler uses built-in inline codes for some common functions rather than calling a library. The built-in codes are not optimal on modern microprocessors. Use option `-fno-builtin` on the Gnu compiler or `/Oi-` on the Microsoft compiler to make sure the library functions are called.

The override method may fail if the standard library has multiple functions in the same module. If the standard library has two functions in the same module, and your program uses both functions, then you cannot replace one without replacing the other. If asmlib replaces one, but not the other, then the linker will then generate an error message saying that there are two definitions of the replaced function.

If the override method fails or if you don't want to override the standard library then use the no-override version of asmlib and call the desired functions with the `A_` prefix.

1.5 Comparison with other function libraries

Test	Processor	Microsoft	CodeGear	Intel	Mac	Gnu 32-bit	Gnu 32-bit -fno-builtin	Gnu 64 bit -fno-builtin	Asmlib
`memcpy` 16kB aligned operands	Intel Core 2	0.12	0.18	0.12	0.11	0.18	0.18	0.18	0.11

memcpy 16kB unaligned op.	Intel Core 2	0.63	0.75	0.18	0.11	1.21	0.57	0.44	0.12
memcpy 16kB aligned operands	AMD Opteron K8	0.24	0.25	0.24	n.a.	1.00	0.25	0.28	0.22
memcpy 16kB unaligned op.	AMD Opteron K8	0.38	0.44	0.40	n.a.	1.00	0.35	0.29	0.28
strlen 128 bytes	Intel Core 2	0.77	0.89	0.40	0.30	4.5	0.82	0.59	0.27
strlen 128 bytes	AMD Opteron K8	1.09	1.25	1.61	n.a.	2.23	0.95	0.6	1.19

Comparing performance of different function libraries.
Numbers in the table are core clock cycles per byte of data (low numbers mean good performance). Aligned operands means that source and destination both have addresses divisible by 16.
Library versions tested (not up to date):
Microsoft Visual studio 2008, v. 9.0
CodeGear Borland bcc, v. 5.5
Mac: Darwin8 g++ v 4.0.1.
Gnu: Glibc v. 2.7, 2.8.
Asmlib: v. 2.10.
Intel C++ compiler, v. 10.1.020. Functions _intel_fast_memcpy and __intel_new_strlen in library libircmt.lib (undocumented function names).

See my manual Optimizing software in C++ for a discussion of the different function libraries.

1.6 Exceptions

Asmlib does not support exception handling. A general protection violation exception can occur if any of the functions in asmlib attempts to access non-existing or invalid memory addresses. The division functions can generate an exception in case of division by zero or a divisor out of range. Such an exception is likely to be the result of a programming error rather than intended behavior. The exception will cause a fatal error message but it is not possible to catch the exception and recover from it. The exception-handling methods are platform specific, and I have given higher priority to fast execution and portability than to support an exception catching that is unlikely to be useful.

1.7 String instructions and safety precautions

The string instructions in this library use the old fashioned C language way of handling strings because this is much faster than the C++ style string classes with dynamic memory allocation (see my manual "Optimizing software in C++"). The strings are stored in char arrays with the end of each string marked by a zero. Before storing a string in an array, the program must check that the size of the array is at least the length of the string plus one in order to hold the terminating zero. Writing beyond the boundaries of an array can cause malfunctions elsewhere in the program that are difficult to diagnose. This applies to the functions strcopy, strcat, substring, and any other functions that write strings.

Some of the string functions in the asmlib library can read beyond the end of a string (but never write beyond the end of a string). This is because they use the very efficient SSE4.2 instructions (if available) which will handle 16 characters at a time. The following asmlib functions can read up to 15 bytes beyond the end of a string: strstr, strcmp, strspn, strcspn, strtolower, strtoupper, strcount_UTF8, strCountInSet. Reading irrelevant bytes will not normally cause a problem as long as nothing is written to the irrelevant addresses. But this can possibly cause an error if the string is placed at the very end of data memory so that it attempts to read into a non-existing address space. This will cause the program to stop immediately with an error message.

If we wanted to prevent the library functions from reading non-existing memory addresses then we would have to check for memory page boundaries for every 16-bytes read. This would cause the functions to be much slower. Since the main focus of the asmlib library is to improve speed, we have chosen a different solution to this problem, namely to make sure that no string is placed at the very end of valid data memory. The functions simply add 16 bytes of unused memory to the uninitialized data section (`.bss`) which comes after the normal data section (`.data`). This will most likely prevent any error, but the programmer should take care of the following considerations if you want to be absolutely safe:

String literals, static arrays, and global arrays are stored in a static data section, which is followed by the `.bss` section and often several other sections. This is safe to use if one of the abovementioned functions is included in the same executable. A DLL or shared object has its own data sections. These data sections are usually followed by import tables, exception handler tables, etc. To be absolutely sure, you may link one of the above functions into the DLL/shared object by making a (dummy) call to it, for example `A_strcmp("","")`.

An array declared inside a function is a good and efficient place to store a string. The array is stored on the stack (unless declared `static`) and deallocated when the function returns. Reading beyond the end of the string array will not cause problems because there will always be something else (parameters and return addresses) at the end of the stack section.

Strings that are dynamically allocated with `new` or `malloc` or use C++ style string classes are stored on the heap. I don't have detailed information about the implementation of the heap in various systems to tell whether there is an end node of at least 15 bytes. It is recommended to allocate sufficient heap space if you are using dynamically allocated strings. A safer and more efficient solution is to allocate a memory pool of sufficient size and store multiple strings in the same memory pool. An implementation of such a string pool is provided in www.agner.org/optimize/cppexamples.zip, which also has support for using asmlib.

Most of the string functions can be used with either ASCII strings or UTF-8 encoded strings or any code page that uses single-byte codes. The UTF-8 coding system uses a single byte for the most common characters and multiple bytes for the less common characters. The UTF-8 is designed so that no part of a multi-byte code will in itself be a valid UTF-8 code. Thanks to this feature, it is possible to use search functions such as `strcmp` and `strstr` with UTF-8 strings. It is not safe to use the `substring` function on UTF-8 strings, unless you make special checks to avoid cutting off part of a multi-byte character code.

2 Memory and string functions

2.1 memcpy

Function prototype

```
void * A_memcpy(void * dest, const void * src, size_t count);
```

Description

Fast implementation of the standard `memcpy` function. Copies `count` bytes from `src` to `dest`. It is the responsibility of the programmer to make sure `count` does not exceed the size in bytes of `dest`. If the beginning of the destination block overlaps with the source then it is possible that part of the source is overwritten before it is copied. The programmer cannot rely on the data being copied in any particular order.

Uncached writes
This function can write either via the data cache or directly to memory. Writing to the cache is usually faster, but it may be advantageous to write directly to memory when the size of the data block is very big, in order to avoid polluting the cache.

The `A_memcpy` function will use uncached writes when the size specified by `count` is bigger than a certain limit. This limit is set by default to half the size of the largest level cache. The limit can be read with `GetMemcpyCacheLimit` and changed with `SetMemcpyCacheLimit`. These functions are defined as:

```
size_t GetMemcpyCacheLimit(void);
void   SetMemcpyCacheLimit(size_t limit);
```

The latter function will restore the default value (half the size of the largest level cache) when called with `limit` = 0.

Versions included
Standard library override version: Yes
Position-independent version: Yes
Stdcall version: No

2.2 memmove

Function prototype

```
void * A_memmove(void * dest, const void * src, size_t count);
```

Description
Fast implementation of the standard `memmove` function. Copies `count` bytes from `src` to `dest`. It is the responsibility of the programmer to make sure `count` does not exceed the size in bytes of `dest`. Allows overlap between `src` and `dest` by copying backwards if the first part of destination overlaps with the source and forwards if the last part of destination overlaps with the source. If there is no overlap then it may copy the data in any order.

Uncached writes
The `A_memmove` function will use uncached writes when the size specified by `count` is bigger than a certain limit. This limit is the same as for `A_memcpy`, see page 6.

Versions included
Standard library override version: Yes
Position-independent version: Yes
Stdcall version: No

2.3 memset

Function prototype

```
void * A_memset(void * dest, int c, size_t count);
```

Description
Fast implementation of the standard `memset` function. Inserts `count` copies of the lower byte of `c` into `dest`. It is the responsibility of the programmer to make sure `count` does not exceed the size in bytes of `dest`.

Uncached writes

This function can write either via the data cache or directly to memory. Writing to the cache is usually faster, but it may be advantageous to write directly to memory when the size of the data block is very big, in order to avoid polluting the cache.

The `A_memset` function will use uncached writes when the size specified by `count` is bigger than a certain limit. This limit is set by default to half the size of the largest level cache. The limit can be read with `GetMemsetCacheLimit` and changed with `SetMemsetCacheLimit`. These functions are defined as:

```
size_t GetMemsetCacheLimit(void);
void   SetMemsetCacheLimit(size_t limit);
```

The latter function will restore the default value (half the size of the largest level cache) when called with `limit` = 0.

Versions included
Standard library override version: Yes
Position-independent version: Yes
Stdcall version: No

2.4 strcat

Function prototype

```
char * A_strcat(char * dest, const char * src);
```

Description
Fast implementation of the standard `strcat` function. Concatenates two zero-terminated strings by inserting a copy of `src` after `dest` followed by a terminating zero. It is the responsibility of the programmer to make sure that `strlen(dest)+strlen(src)+1` does not exceed the size in bytes of the array containing `dest`.

Uncached writes
Extremely long strings can bypass the cache, see page 6.

Versions included
Standard library override version: Yes
Position-independent version: Yes
Stdcall version: No

2.5 strcopy

Function prototype

```
char * A_strcpy(char * dest, const char * src);
```

Description
Fast implementation of the standard `strcopy` function. Copies a zero-terminated string `src` into an array `dest` followed by a terminating zero. It is the responsibility of the programmer to make sure that `strlen(src)+1` does not exceed the size in bytes of the array `dest`.

Uncached writes
Extremely long strings can bypass the cache, see page 6.

Versions included
Standard library override version: Yes
Position-independent version: Yes
Stdcall version: No

2.6 strlen

Function prototype

```
size_t A_strlen(const char * str);
```

Description

Fast implementation of the standard strlen function. Returns the length of a zero-terminated string str, not counting the terminating zero.

If str is an ASCII string then the return value is the number of characters. If str is UTF-8 encoded then the return value is the number of code bytes, not the number of Unicode characters. See also the function strcount_UTF8 on page 11.

Versions included

Standard library override version: Yes
Position-independent version: Yes
Stdcall version: No

2.7 strstr

Function prototype

```
char * A_strstr (char * haystack, const char * needle);
const char * A_strstr (const char * haystack, const char * needle);
```

Description

Searches for the first occurrence of the substring needle in the string haystack. The return value is a pointer to the first occurrence of the substring needle in haystack, or zero (NULL) if not found. This function is particularly fast if the SSE4.2 instruction set is supported by the processor.

The two parameters can be zero-terminated ASCII or UTF-8 strings. It works with UTF-8 strings because no part of a multi-byte UTF-8 character can be a valid character. This implementation is useful for speeding up lexical processing, text parsing and DNA analysis applications.

Note

This function may read up to 15 bytes beyond the ends of the two strings. See page 4 for necessary precautions.

Versions included

Standard library override version: No, because of the special precautions needed.
Position-independent version: Yes.
Stdcall version: No.

2.8 strcmp

Function prototype

```
int A_strcmp (const char * string1, const char * string2);
```

Description

Compares two strings with case sensitivity. The two parameters can be zero-terminated ASCII or UTF-8 strings.

The return value is negative if string1 < string2, zero if string1 = string2, and positive if string1 > string2. The comparison is based on the unsigned ASCII or Unicode value of the first character that differs between string1 and string2.

Note
This function may read up to 15 bytes beyond the ends of the two strings. See page 4 for necessary precautions.

Versions included
Standard library override version: No, because of the special precautions needed as explained in the above note.
Position-independent version: Yes.
Stdcall version: No.

2.9 stricmp

Function prototype

```
int A_stricmp(const char *string1, const char *string2);
```

Description
String comparison without case sensitivity. This is similar to the standard library function variously named `stricmp`, `_stricmp`, `strcmpi` or `strcasecmp`, but it differs by not depending on locale settings or codepages. The two parameters are zero-terminated ASCII or UTF-8 strings.

`A_stricmp` is faster than the standard function `stricmp` etc. when a locale or codepage is defined because it does not have to look up all characters in tables. The letters A-Z are compared as if they were lower case, but other letters such as Á, á, Ä, ä, Å, å, etc. are regarded as all different and unique.

The return value is negative if `string1` < `string2`, zero if `string1` = `string2`, and positive if `string1` > `string2`. The comparison is based on the unsigned ASCII or Unicode value of the first character that differs between `string1` and `string2` converted to lower case.

If multiple comparisons are needed then it is faster to convert both strings to lower case with `A_strtolower` and then compare with `A_strcmp`.

Versions included
Standard library override version: No, because not exactly identical function.
Position-independent version: Yes.
Stdcall version: No.

2.10 strspn, strcspn

Function prototype

```
size_t strspn  (const char * str, const char * set);
size_t strcspn (const char * str, const char * set);
```

Description
`strspn` finds the length of the initial portion of `str` which consists only of characters that are part of `set`. (This is the same as the zero-based index to the first character not contained in of `set`).

`strcspn` finds the length of the initial portion of `str` which consists only of characters that are *not* part of `set`. (This is the same as the zero-based index to the first character that is contained in `set`).

The two parameters are zero-terminated ASCII strings. The functions will not work correctly if `set` contains multi-byte UTF-8 encoded characters.

These functions are useful for string parsing and finding whitespace, delimiters, etc. The functions are particularly fast if the SSE4.2 instruction set is supported by the microprocessor.

Note

This function may read up to 15 bytes beyond the ends of the two strings. See page 4 for necessary precautions.

Versions included

Standard library override version: No, because of the special precautions needed as explained in the above note.
Position-independent version: Yes.
Stdcall version: No.

2.11 substring

Function prototype

```
size_t A_substring(char * dest, const char * source, size_t pos,
size_t len);
```

Description

Makes a substring from `source`, starting at position `pos` (zero-based), and length `len` and stores it in the array `dest`. It is the responsibility of the programmer that the size of the `dest` array is at least `len+1` in order to make space for the string and the terminating zero. The return value is the actual length of the substring. This may be less than `len` if the length of `source` is less than `pos+len`. `source` must be a zero-terminated ASCII string. The substring stored in `dest` will be zero-terminated, even if its length is zero. This function is not found in standard C libraries, though it is often needed.

It is not safe to use this function for UTF-8 encoded strings because it may cut off part of a multi-byte character code. Such a partial character code will surely mess up the subsequent processing of the substring.

Versions included

Standard library override version: No.
Position-independent version: Yes.
Stdcall version: No.

2.12 strtolower, strtoupper

Function prototype

```
void A_strtolower(char * string);
void A_strtoupper(char * string);
```

Description

Converts a zero-terminated string to lower or upper case. Only the letters a-z or A-Z are converted. Other letters such as á, ä, å, α are not converted. The functions save time by not looking up locale-specific characters. The parameter can be a zero-terminated ASCII or UTF-8 string.

Note

This function may read up to 15 bytes beyond the end of the string. See page 4 for necessary precautions.

Versions included

Standard library override version: No.
Position-independent version: Yes.
Stdcall version: No.

2.13 strcount_UTF8

Function prototype

```
size_t strcount_UTF8(const char * str);
```

Description

Counts the number of characters in a zero-terminated UTF-8 encoded string. This value is less than the string length if the string contains multi-byte character codes. The terminating zero is not included in the count. Any byte order mark (BOM) is counted as one character.

This function does not check if the string contains valid UTF-8 code. It only counts the number of bytes, excluding continuation bytes.

Note

This function may read up to 15 bytes beyond the end of the string. See page 4 for necessary precautions.

Versions included

Position-independent version: Yes.
Stdcall version: No.

2.14 strCountInSet

Function prototype

```
size_t strCountInSet(const char * str, const char * set);
```

Description

Counts how many characters in the string `str` that belong to the set defined by the characters in `set`. Both strings are zero-terminated ASCII strings. Does not work if `set` contains multi-byte UTF-8 characters.

Note

This function may read up to 15 bytes beyond the ends of the two strings. See page 4 for necessary precautions.

Versions included

Position-independent version: Yes.
Stdcall version: No.

3 Integer division functions

These functions are intended for fast integer division when the same divisor is used multiple times. Division is slow on most microprocessors. In floating point calculations, we can do multiple divisions with the same divisor faster by multiplying with the reciprocal, for example:

```
    float a, b, d;
    a /= d;  b /= d;
```

can be changed to:

```
    float a, b, d, r;
    r = 1.0f / d;
    a *= r;  b *= r;
```

If we want to do something similar with integers then we have to scale the reciprocal divisor by 2^n and then shift n places to the right after the multiplication. A good deal of sophistication is needed to determine a suitable value for n and to compensate for rounding errors. The following functions implement this method in such a way that the result is truncated towards zero in order to get exactly the same result as we get with the '/' operator.

Most compilers will actually use this method automatically if the value of the divisor is a constant known at compile time. However, if the divisor is known only at runtime and you are doing multiple divisions with the same divisor then it is faster to use the functions described below.

The same method is useful for integer division in vector registers because there are currently no assembly instructions or intrinsic functions for integer vector division. The functions below satisfy this need.

3.1 Signed and unsigned integer division

Function prototype, signed version

```
void setdivisori32(int buffer[2], int d);
int dividefixedi32(const int buffer[2], int x);
```

Function prototype, unsigned version

```
void setdivisoru32(unsigned int buffer[2], unsigned int d);
unsigned int dividefixedu32(const unsigned int buffer[2], unsigned
int x);
```

Description

The buffer parameter is used internally for storing the reciprocal divisor and the shift count. setdivisor.. must be called first with the desired divisor d. Then dividefixed.. can be called for each x that you want to divide by d. Note that the divisor d must be positive, while the dividend x can have any value. If you need a negative divisor then change the sign of the divisor to positive and change the sign of the result. You may use multiple buffers if you have multiple divisors.

Wrapper class and overloaded '/' operator

A wrapper class with an overloaded '/' operator is included when using C++. The name of this wrapper class is div_i32 for the signed version and div_u32 for the unsigned version. It can be used in the following way:

```
    int a, b, d;
    div_i32 div(d);          // Object div represents divisor d
    a = a / div;             // Same as a/d but faster
    b = b / div;             // Same as b/d but faster
```

You may have multiple instances of the class if you have different divisors, or change the divisor with `div.setdivisor(NewDivisor);`

Error conditions

d = 0 will generate a divide-by-zero exception. d < 0 will generate a division overflow exception in the signed version.

Versions included

Position-independent versions: Yes.
Stdcall versions: No.

3.2 Integer vector division

Function prototype, vector of 8 signed 16-bit integers

```
void setdivisorV8i16(__m128i buf[2], int16_t d);
__m128i dividefixedV8i16(const __m128i buf[2], __m128i x);
```

Function prototype, vector of 8 unsigned 16-bit integers

```
void setdivisorV8u16(__m128i buf[2], uint16_t d);
__m128i dividefixedV8u16(const __m128i buf[2], __m128i x);
```

Function prototype, vector of 4 signed 32-bit integers

```
void setdivisorV4i32(__m128i buf[2], int32_t d);
__m128i dividefixedV4i32(const __m128i buf[2], __m128i x);
```

Function prototype, vector of 4 unsigned 32-bit integers

```
void setdivisorV4u32(__m128i buf[2], uint32_t d);
__m128i dividefixedV4u32(const __m128i buf[2], __m128i x);
```

Description

The buf parameter is used internally for storing the reciprocal divisor and the shift count. setdivisor.. must be called first with the desired divisor d. Then dividefixed.. can be called for each vector x that you want to divide by d. Note that the divisor d must be positive, while the dividend x can have any value. If you need a negative divisor then change the sign of the divisor to positive and change the sign of the result. You may use multiple buffers if you have multiple divisors. The 16-bit versions are faster than the 32-bit versions, measured by the time it takes to divide a whole vector.

The header file emmintrin.h must be included before asmlib.h in order to enable the vector type __m128i if you use these functions.

Wrapper classes and overloaded '/' operator

Wrapper classes with overloaded '/' operators are included when using C++. The name of these wrapper classes are as follows:

div_v8i16:	vector of 8 signed 16-bit integers
div_v8u16:	vector of 8 unsigned 16-bit integers
div_v4i32:	vector of 4 signed 32-bit integers
div_v4u32:	vector of 4 unsigned 32-bit integers

These wrapper classes can be used in the following way:

```
    #include <emmintrin.h>  // Include emmintrin.h before asmlib.h
    #include <asmlib.h>     // Header file for asmlib
    __m128i a;              // Integer vector
    int d;                  // Divisor
    div_v8i16 div(d);       // Object div represents divisor d
    a = a / div;            // Each element of a is divided by d
```

You may have multiple instances of the class if you have different divisors, or change the divisor with div.setdivisor(NewDivisor);

If Intel's header file dvec.h is available then you may use the vector classes in dvec.h instead of __m128i to get the operators +, -, *, etc. as well. Include dvec.h before asmlib.h.

Error conditions

d = 0 will generate a divide-by-zero exception. d < 0 will generate a division overflow exception in the signed versions.

Versions included

Position-independent versions: Yes.
Stdcall versions: No.

4 Other functions

4.1 round

Function prototypes

```
int RoundF(float  x);
int RoundD(double x);
int Round(float x);     // C++ overloaded
int Round(double x);    // C++ overloaded
```

Description

Converts a floating point number to the nearest integer. When two integers are equally near, then the even integer is chosen (provided that the current rounding mode is set to default). This function does not check for overflow. The default way of converting floating point numbers to integers in C++ is truncation. Rounding is much faster than truncation in 32 bit mode when the SSE2 instruction set is not enabled.

Versions included

Position-independent versions: Yes
Stdcall versions: No

Alternatives

Compilers with C99 or C++0x support have the identical functions `lrint` and `lrintf`. Compilers with intrinsic functions support have `_mm_cvtsd_si32` and `_mm_cvt_ss2si` when SSE2 is enabled.

4.2 popcount

Function prototype

```
unsigned int A_popcount(unsigned int x);
```

Description

Population count. Counts the number of 1-bits in a 32-bit integer.

Versions included

Position-independent versions: Yes
Stdcall versions: No

4.3 InstructionSet

Function prototype

```
int InstructionSet(void);
```

Description

This function detects which instructions are supported by the microprocessor and the operating system. The return value is also stored in a global variable named `IInstrSet`. If `IInstrSet` is not negative then `InstructionSet` has already been called and you don't need to call it again.

Return values:

Return value	Meaning
0	80386 instruction set only
1 or above	MMX instructions supported
2 or above	conditional move and FCOMI supported
3 or above	SSE (XMM) supported by processor and enabled by Operating system
4 or above	SSE2 supported
5 or above	SSE3 supported
6 or above	Supplementary-SSE3 supported
8 or above	SSE4.1 supported
9 or above	POPCNT supported
10 or above	SSE4.2 supported
11 or above	AVX (YMM) supported by processor and enabled by Operating system
12 or above	PCLMUL and AES supported
13 or above	AVX2 supported

The return value will always be 4 or above in 64-bit systems.

This function is intended to indicate only instructions that are supported by Intel, AMD and VIA and instructions that might be supported by all these vendors in the future. Each level is reported only if all the preceding levels are also supported.

Instructions and features that do not form a natural sequence or which may not be supported in future processors are not included here.

Vendor-specific instructions (e.g. XOP for AMD) are not included here.

The future FMA instruction sets are not yet included because it is presently unknown (July 2011) whether FMA3 or FMA4 will be the common norm.

Versions included
Position-independent version: Yes
Stdcall version: Same version can be used.

4.4 ProcessorName

Function prototype

```
char * ProcessorName(void);
```

Description
Returns a pointer to a static zero-terminated ASCII string with a description of the microprocessor as returned by the CPUID instruction.

Versions included
Position-independent versions: Yes
Stdcall version: Same version can be used.

4.5 CpuType

Function prototype

```
void CpuType(int * vendor, int * family, int * model);
```

Description
Determines the vendor, family and model number of the current CPU and returns these to the variables pointed to by the parameters.
Values of vendor:
0 = unknown, 1 = Intel, 2 = AMD, 3 = VIA/Centaur, 4 = Cyrix, 5 = NexGen.

The value returned as family is the sum of the family and extended family numbers as given by the cpuid instruction.
The value returned as model is the model number + (extended model number << 8), as given by the cpuid instruction.

Null pointers are allowed for values that are not needed.

Versions included
Position-independent versions: Yes
Stdcall versions: No

4.6 DataCacheSize

Function prototype

```
size_t DataCacheSize(int level);
```

Description
Gives the size in bytes of the level-1, level-2 or level-3 data cache, for `level` = 1, 2, or 3, respectively. The size of the largest-level cache is returned when `level` = 0. This function does not tell the size of the code cache.

A value of 0 is returned if there is no cache or the function fails to determine the cache size.

Versions included
Position-independent versions: Yes
Stdcall versions: No

4.7 cpuid_abcd

Function prototype

```
void cpuid_abcd(int abcd[4], int eax);
```

Description
This function calls the `CPUID` machine instruction.
The input value of register `eax` is in `eax`.
The output value of register `eax` is returned in `abcd[0]`.
The output value of register `ebx` is returned in `abcd[1]`.
The output value of register `ecx` is returned in `abcd[2]`.
The output value of register `edx` is returned in `abcd[3]`.
The use of the CPUID instruction is documented in manuals from Intel and AMD.

Alternative
Compilers with support for intrinsic functions may have the similar function `__cpuid`.

Versions included
Position-independent version: Yes
Stdcall version: No.

4.8 cpuid_ex

Function prototype

```
void cpuid_ex(int abcd[4], int eax, int ecx);
```

Description
This function calls the `CPUID` machine instruction.
The input value of register `eax` is in `eax`.

The input value of register ecx is in ecx.
The output value of register eax is returned in abcd[0].
The output value of register ebx is returned in abcd[1].
The output value of register ecx is returned in abcd[2].
The output value of register edx is returned in abcd[3].
The use of the CPUID instruction is documented in manuals from Intel and AMD.

Alternative
Compilers with support for intrinsic functions may have the similar function __cpuidex.

Versions included
Position-independent versions: Yes
Stdcall versions: No

4.9 ReadTSC

Function prototype

```
uint64_t ReadTSC(void);
```

Description
This function returns the value of the internal clock counter in the microprocessor. Execution is serialized before and after reading the time stamp counter in order to prevent out-of-order execution. Does not work on 80386 and 80486. A 32-bit value is returned if the compiler doesn't support 64-bit integers.

To count how many clock cycles a piece of code takes, call ReadTSC before and after the code to measure and calculate the difference.

You may see that the count varies a lot because you may not be able to prevent interrupts during the execution of your code. If the measurement is repeated then you will see that the code takes longer time the first time it is executed than the subsequent times because code and data are not cached at the first execution.

Time measurements with ReadTSC() may not be fully reproducible on Intel processors with SpeedStep technology (i.e. Core and later) because the clock frequency is variable.

ReadTSC() is also useful for generating a seed for a random number generator.

Versions included
Position-independent version: Yes
Stdcall version: Same version can be used.

4.10 DebugBreak

Function prototype

```
void A_DebugBreak(void);
```

Description
Makes a debug breakpoint for testing purposes. Will not work when the program is not running in a debugger.

Versions included
Position-independent version: Yes
Stdcall version: Same version can be used.

4.11 Random number generators

Several random number generators are provided in a separate function library available from www.agner.org/random.

5 Library versions

The asmlib library has many versions for compatibility with different platforms and compilers. Use the tables below to select the right version for a particular application.

Library version selection guide: Windows				
Compiler/language	File format	Override standard library	32 bit	64 bit
MS C++ unmanaged, Intel, Gnu	COFF	yes	alibcof32o.lib	alibcof64o.lib
		no	alibcof32.lib	alibcof64.lib
Borland C++, Watcom, Digital Mars	OMF	yes	alibomf32o.lib	
		no	alibomf32.lib	
MS C++ .net, C#, VB	DLL	no	alibd32.dll	alibd64.dll
Borland Delphi	DLL	no	alibd32.dll	
Other languages	DLL	no	alibd32.dll	alibd64.dll

Library version selection guide: Linux and BSD (x86 and x86-64)					
Compiler/language	File format	Override standard library	32 bit executable	32 bit shared object	64 bit
Gnu, Intel C++	ELF	yes	alibelf32o.a	alibelf32op.a	alibelf64o.a
		no	alibelf32.a	alibelf32p.a	alibelf64.a

Library version selection guide: Mac (Intel based)					
Compiler/language	File format	Override standard library	32 bit executable	32 bit shared object	64 bit
Gnu, Intel C++	MachO	yes	alibmac32o.a	alibmac32op.a	alibmac64o.a
		no	alibmac32.a	alibmac32p.a	alibmac64.a

Explanation of the column headings:

Compiler/language: The compiler and programming language used. Different compilers may use different object file formats.

File format: It is necessary to select a library in the right object file format, or a dynamic link library if static linking is not possible.

Override standard library: Libraries with suffix o use the same names for standard functions as standard libraries. If this library is linked before the standard library then it will replace the standard functions. Libraries without suffix o use different names for the standard functions.

32 bit / 64 bit: Use the appropriate version when compiling for 32-bit mode or 64-bit mode.

32 bit executable: Use this version when making a main executable binary file.

32 bit shared object: Use this version when position-independent code is needed. Position-independent code is needed when building a shared object for 32-bit mode, but it is slightly slower.

6 File list

Files in asmlib.zip

asmlib-instructions.pdf	This file
asmlib.h	C/C++ Header file for asmlib functions
alibelf32.a	Library 32-bit ELF format
alibelf32o.a	Library 32-bit ELF format, override standard library
alibelf32op.a	Library 32-bit ELF format, override, position-independent
alibelf32p.a	Library 32-bit ELF format, position-independent
alibelf64.a	Library 64-bit ELF format
alibelf64o.a	Library 64-bit ELF format, override standard library
alibmac32.a	Library 32-bit Mach-O format
alibmac32o.a	Library 32-bit Mach-O format, override standard library
alibmac32op.a	Library 32-bit Mach-O format, override, position-independent
alibmac32p.a	Library 32-bit Mach-O format, position-independent
alibmac64.a	Library 64-bit Mach-O format
alibmac64o.a	Library 64-bit Mach-O format, override standard library
alibd32.dll	Library 32-bit Windows DLL
alibd32.lib	Import library for alibd32.dll
alibd64.dll	Library 64-bit Windows DLL
alibd64.lib	Import library for alibd64.dll
alibcof32.lib	Library 32-bit COFF format
alibcof32o.lib	Library 32-bit COFF format, override standard library
alibcof64.lib	Library 64-bit COFF format
alibcof64o.lib	Library 64-bit COFF format, override standard library
alibomf32.lib	Library, 32-bit OMF format
alibomf32o.lib	Library, 32-bit OMF format, override standard library
license.txt	Gnu general public license
asmlibSrc.zip	Source code

Files in asmlibSrc.zip

alibd32.asm alibd64.asm	Source code for DLL entry
instrset32.asm instrset64.asm	Source code for InstructionSet function
cachesize32.asm cachesize64.asm	Source code for DataCacheSize function
cpuid32.asm cpuid64.asm	Source code for cpuid... functions
cputype32.asm cputype64.asm	Source code for CpuType function
debugbreak32.asm debugbreak64.asm	Source code for DebugBreak function
divfixedi32.asm divfixedi64.asm	Source code for integer division functions
divfixedv32.asm divfixedv64.asm	Source code for integer vector division functions
instrset32.asm instrset64.asm	Source code for InstructionSet function
memcpy32.asm memcpy64.asm	Source code for memcpy function
memmove32.asm memmove64.asm	Source code for memmove function
memset32.asm memset64.asm	Source code for memset function
popcount32.asm popcount64.asm	Source code for popcount function
procname32.asm procname64.asm	Source code for ProcessorName function
rdtsc32.asm rdtsc64.asm	Source code for ReadTSC function
round32.asm round64.asm	Source code for Round functions
serialize32.asm serialize64.asm	Source code for Serialize function
strcat32.asm strcat64.asm	Source code for strcat function
strcmp32.asm strcmp64.asm	Source code for strcmp function
strcountset32.asm strcountset64.asm	Source code for strCountInSet function
strcountutf832.asm strcountutf864.asm	Source code for strcount_UTF8 function
strcpy32.asm strcpy64.asm	Source code for strcpy function
stricmp32.asm stricmp64.asm	Source code for stricmp function

strlen32.asm strlen64.asm	Source code for strlen function
strspn32.asm strspn64.asm	Source code for strspn and strcspn functions
strstr32.asm strstr64.asm	Source code for strstr function
strtouplow32.asm strtouplow64.asm	Source code for strtolower and strtoupper functions
substring32.asm substring64.asm	Source code for substring function
unalignedisfaster32/64.asm	Source code for internal function
testalib.cpp	Test example
alibd32.def	Exports definition function for alibd32.dll
alibd64.def	Exports definition function for alibd64.dll
MakeAsmlib.bat	Batch file for making asmlib
asmlib.make	Makefile for making asmlib

7 License conditions

These software libraries are free: you can redistribute the software and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the license, or any later version.

Commercial licenses are available on request to www.agner.org/contact.

This software is distributed in the hope that it will be useful, but without any warranty. See the file `license.txt` or www.gnu.org/licenses for the license text.

8 No support

Note that asmlib is a free library provided without warranty or support. This library is for experts only, and it may not be compatible with all compilers and linkers. If you have problems using it, then don't.

I am sorry that I don't have the time and resources to provide support for this library. If you ask me to help with your programming problems then you will not get any answer. Bug reports are welcome, though.