Instructions for asmlib

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

By Agner Fog. Technical University of Denmark
Version 2.52. 2022-11-13

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
- Efficient random number generators

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, AVX, AVX2, AVX512 etc. These functions will detect which instruction set is supported by the microprocessor it is running on and select the optimal branch. This detection is done automatically the first time such a function is called, and an internal pointer is set to the optimal version of the function so that no detection is required when the same function is called again.

This library is also intended as a showcase to illustrate the optimization methods explained in my optimization manuals and as 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, AMD and VIA 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, Power-PC or ARM 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.
Version 2.50 and later no longer includes position-independent 32-bit versions of the libraries because these can only be built with the YASM assembler, which is no longer maintained.

See page 6 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 does not support static linking or if the static link library is incompatible.

A DLL under 32-bit Windows uses the `stdcall` calling convention by default. Only some of the functions in asmlib have a `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.

For example, to call the Mersenne twister random number generator from Borland Delphi Pascal, use the function declarations:

```
Procedure MersenneRandomInitD(seed:integer); stdcall;
  external 'libad32.dll';
Procedure MersenneRandomInitByArrayD(seeds:PInteger;
  NumSeeds:integer); stdcall; external 'libad32.dll';
  { seeds must point to first element of array }
Function MersenneRandomD: double; stdcall; external 'libad32.dll';
Function MersenneIRandomD(min,max:integer):integer; stdcall;
  external 'libad32.dll';
Function MersenneIRandomXD(min,max:integer):integer; stdcall;
  external 'libad32.dll';
Function MersenneBRandomD:integer; stdcall; external 'libad32.dll';
```

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. Position-independent 32-bit code is no longer supported in asmlib.

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 do not 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 6 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 do not 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 structured exception handling. A general protection violation exception can occur if any of the functions in asmlib attempts to access 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 not likely to be useful.

1.7 String instructions and safety precautions

The string instructions in this library use the traditional 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 do not 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 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	libacof32o.lib	libacof64o.lib
		no	libacof32.lib	libacof64.lib
Borland C++, Watcom, Digital Mars	OMF	yes	libaomf32o.lib	
		no	libaomf32.lib	
MS C++ .net, C#, VB	DLL	no	libad32.dll	libad64.dll
Borland Delphi	DLL	no	libad32.dll	
Other languages	DLL	no	libad32.dll	libad64.dll

Library version selection guide: Linux and BSD (x86 and x86-64)				
Compiler/language	File format	Override standard library	32 bit executable	64 bit
Gnu, Clang, Intel C++	ELF	yes	libaelf32o.a	libaelf64o.a
		no	libaelf32.a	libaelf64.a

Library version selection guide: Mac (Intel based)				
Compiler/language	File format	Override standard library	32 bit executable	64 bit
Gnu, Clang, Intel C++	MachO	yes	libamac32o.a	libamac64o.a
		no	libamac32.a	libamac64.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 standard functions such as `memcpy`, `memset`, `strlen`, etc.. 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.

3 Memory and string functions

3.1 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. This function uses different methods for different CPU models, based on tests of which method is fastest on each type of CPU.

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 last 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 last level cache) when called with `limit` = 0.

Versions included

Standard library override version: Yes
Stdcall version: No

3.2 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 that `count` does not exceed the size in bytes of `dest`. This function allows overlap between `src` and `dest` by making sure that overlapping memory positions are read before they are written. The programmer cannot rely on the data being copied in any particular order, except for this rule. This function uses different methods for different CPU models, based on tests of which method is fastest on each type of CPU.

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 7.

Versions included
Standard library override version: Yes
Stdcall version: No

3.3 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`. This function uses different methods for different CPU models, based on tests of which method is fastest on each type of CPU.

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 last 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 last level cache) when called with `limit` = 0.

Versions included
Standard library override version: Yes
Stdcall version: No

3.4 memcmp

Function prototype

```
int A_memcmp(const void * buf1, const void * buf2, size_t count);
```

Description

Fast implementation of the standard `memcmp` function. Compares two blocks of memory of size `count` bytes. The return value is zero if the two memory blocks `ptr1` and `ptr2` are equal. The return value is positive if the first differing byte is bigger in `ptr1` than in `ptr2` when compared as unsigned bytes. The return value is negative if the first differing byte is smaller in `ptr1` than in `ptr2` when compared as unsigned bytes.

Versions included
Standard library override version: Yes
Stdcall version: No

3.5 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 7.

Versions included
Standard library override version: Yes
Stdcall version: No

3.6 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 7.

Versions included
Standard library override version: Yes
Stdcall version: No

3.7 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 12.

Versions included
Standard library override version: Yes
Stdcall version: No

3.8 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 5 for necessary precautions.

Versions included
Standard library override version: No, because of the special precautions needed.
Stdcall version: No.

3.9 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 5 for necessary precautions.

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

3.10 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`, with A-Z 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.
Stdcall version: No.

3.11 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 5 for necessary precautions.

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

3.12 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.
Stdcall version: No.

3.13 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 5 for necessary precautions.

Versions included
Standard library override version: No.
Stdcall version: No.

3.14 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 value returned by `strlen` 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 5 for necessary precautions.

Versions included
Stdcall version: No.

3.15 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 5 for necessary precautions.

Versions included

Stdcall version: No.

4 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. This is implemented in the vector class library, available from www.agner.org/optimize/#vectorclass. If you do not want to use the vector class library then you may use the vector division functions below.

4.1 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

Stdcall versions: No.

4.2 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.

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

Stdcall versions: No.

Vector classes and overloaded '/' operator

Vector classes with overloaded operators for integer vector division are provided in the vector class library www.agner.org/optimize/VectorClass.zip.

These vector classes can be used as shown in the following example:

```
// Example for dividing 400 integers by 10
#include <vectorclass.h>     // Header file for vector classes

int dividends[400], quotients[400]; // numbers to work with
Divisor4i div10(10);                // make divisor object
Vec4i temp;                         // temporary vector of 4 int

// loop for 4 elements per iteration
for (int i = 0; i < 400; i += 4) {
    temp.load(dividends+i);         // load 4 elements
    temp /= div10;                  // divide each element by 10
    temp.store(quotients+i);        // store 4 elements
}
```

If Intel's header file `dvec.h` is available then you may use the vector classes in `dvec.h` instead, though it is less versatile. This example illustrates how:

```
#include <dvec.h>                   // Intel vector classes
#include "asmlib.h"                 // asmlib header file

// define class for encapsulating division parameters
class Divisor_Is32vec4 {
public:
    __m128i buf[2];                 // parameters
    Divisor_Is32vec4(int d){        // constructor
        setdivisorV4i32(buf,d);     // calculate parameters
    }
};

// define operator for dividing vector by divisor object
Is32vec4 operator / (Is32vec4 const & a, Divisor_Is32vec4 const & d) {
    return dividefixedV4i32(d.buf, a);
}

int dividends[400], quotients[400]; // numbers to work with
Divisor_Is32vec4 div10(10);         // make divisor object
Is32vec4 temp;                      // temporary vector of 4 int

// loop for 4 elements per iteration
for (int i = 0; i < 400; i += 4) {
    temp = *(Is32vec4*)(dividends+i);  // load 4 elements
    temp = temp / div10;               // divide each element by 10
    *(Is32vec4*)(quotients+i) = temp;  // store 4 elements
}
```

5 Miscellaneous functions

5.1 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

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.

5.2 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

Stdcall versions: No

5.3 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 do not 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 (SSSE3)
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
14 or above	FMA3, F16C, BMI1, BMI2, LZCNT
15 or above	AVX512F
16 or above	AVX512BW, AVX512DQ, AVX512VL

The return value will always be 4 or more 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.

Versions included
Stdcall version: Same version can be used.

5.4 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
Stdcall version: Same version can be used.

5.5 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
Stdcall versions: No

5.6 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 if the function fails to determine the cache size.

Versions included
Stdcall versions: No

5.7 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
Stdcall version: No.

5.8 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
Stdcall versions: No

5.9 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 the old 80386 and 80486 processors. A 32-bit value is returned if the compiler does not 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 less likely to be in the cache 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.

Versions included
Stdcall version: Same version can be used.

5.10 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
Stdcall version: Same version can be used.

6 Random number generator functions

These random number generators form a part of the random number generator library, available from www.agner.org/random. Please see the random number generator library for instructions, theoretical details and for generating different probability distributions.

Large applications may instead use the random number vector generator in the vector class library http://www.agner.org/optimize/#vectorclass.

There are several different pseudo random number generators available in asmlib:

Pseudo random number generator	Description
Mersenne twister	This generator has become very popular because of its good randomness, long cycle length and high speed.
SFMT generator	A further development of the Mersenne twister, specially designed for computers with vector instructions. Better and faster than the standard Mersenne twister.
Mother-of-all generator	An older generator with a small memory footprint. Has higher bifurcation but lower cycle length than the other generators.
Combined SFMT and Mother-of-all generator	Combines the SFMT generator and the Mother-of-all generator for the most demanding applications.

There is also a non-deterministic random number generator `PhysicalSeed`, see page 24.

Each of the pseudo random number generators are available in different implementations:

Variant	Description
Single threaded, C, static link	Simple to call from `*.lib` or `*.a` library. Not for multiple threads.
Single threaded, DLL	Windows DLL. To call from programming languages that cannot link to a `*.lib` library. Not for multiple threads.
Thread-safe, C, static link	For multi-threaded applications in C language. Caller must supply a storage buffer for each thread. Link with `*.lib` or `*.a` library.
Thread-safe, C++, static link	Convenient C++ class for single and multi-threaded applications, linking to `*.lib` or `*.a` library.
Thread-safe, C++ source	Source code in C++. Does not need external library. Provided in `randomc.zip` package, www.agner.org/random.

Pseudo-random numbers are generated by calling the following functions. Each function has different variants for the different generators. You must initialize the random number generator before generating the first random number.

Function	Purpose
RandomInit	Initialize random number generator with a 32-bit integer as seed. Each seed generates a different sequence of pseudo-random numbers. Starting again with the same seed will generate the same sequence.
InitByArray	Initialize random number with an array of multiple integers as seed. The generated sequence depends on all values in the array. (Not available for Mother-of-all generator).
Random	Generates a random floating point number with uniform distribution in the interval $0 \le x < 1$. Double precision. Resolution: 32 bits in Mersenne Twister and Mother-Of-All generator, 52 bits in SFMT and combined generator.
RandomL	Same as Random, but with long double precision. Only available with SFMT generator, and only for compilers that support long double precision. Resolution: 63 bits.
IRandom	Generates a random integer with uniform distribution in an arbitrary interval. The distribution may be slightly biased due to rounding errors, especially for very large intervals that are not a power of 2. Restrictions: max ≥ min and max - min + 1 < 2^{32}.

IRandomX	Same as IRandom, but with exactly uniform distribution. Slower than IRandom, especially when the interval length is changing. (Not available for Mother-of-all generator).
BRandom	Generates 32 random bits as a 32-bit integer.

These function names have different prefixes etc. for the different generator variants. The complete function declarations are listed below.

6.1 Mersenne twister

Single threaded, C language, static link version

Function prototypes in `asmlibran.h`.

```
void MersenneRandomInit(int seed);
void MersenneRandomInitByArray(int const seeds[], int NumSeeds);
int  MersenneIRandom (int min, int max);
int  MersenneIRandomX(int min, int max);
double MersenneRandom();
uint32_t MersenneBRandom();
```

Single threaded, Windows DLL version

Function prototypes in `asmlibran.h`.

```
void __stdcall MersenneRandomInitD(int seed);
void __stdcall MersenneRandomInitByArrayD(int const seeds[],
               int NumSeeds);
int  __stdcall MersenneIRandomD (int min, int max);
int  __stdcall MersenneIRandomXD(int min, int max);
double   __stdcall MersenneRandomD();
uint32_t __stdcall MersenneBRandomD();
```

Thread-safe, C language, static link version

Function prototypes in `asmlibran.h`.

```
void   MersRandomInit(void * Pthis, int seed);
void   MersRandomInitByArray(void * Pthis, int const seeds[],
       int NumSeeds);
int    MersIRandom (void * Pthis, int min, int max);
int    MersIRandomX(void * Pthis, int min, int max);
double MersRandom  (void * Pthis);
uint32_t MersBRandom (void * Pthis);
```

`Pthis` must point to a storage buffer of size `MERS_BUFFERSIZE` bytes. Use a separate buffer for each thread.

Thread-safe, C++, static link version

Class definition in `asmlibran.h`.

```
class CRandomMersenneA {
public:
   CRandomMersenneA(int seed);
   void RandomInit(int seed);
```

```
   void RandomInitByArray(int const seeds[], int NumSeeds);
   int IRandom (int min, int max);
   int IRandomX(int min, int max);
   double Random();
   uint32_t BRandom();
private:
   char internals[MERS_BUFFERSIZE];
};
```

Make one instance of the class for each thread.

Thread-safe, C++ source code

Class definition in `randomc.h`. Source code in `mersenne.cpp`.
See `randomc.zip`.

6.2 Mother-of-all generator

Single threaded, C language, static link version

Function prototypes in `asmlibran.h`.

```
void     MotherRandomInit(int seed);
int      MotherIRandom (int min, int max);
double   MotherRandom();
uint32_t MotherBRandom();
```

Single threaded, Windows DLL version

Function prototypes in `asmlibran.h`.

```
void     __stdcall MotherRandomInitD(int seed);
int      __stdcall MotherIRandomD (int min, int max);
double   __stdcall MotherRandomD();
uint32_t __stdcall MotherBRandomD();
```

Thread-safe, C language, static link version

Function prototypes in `asmlibran.h`.

```
void   MotRandomInit(void * Pthis, int seed);
int    MotIRandom(void * Pthis, int min, int max);
double MotRandom (void * Pthis);
uint32_t MotBRandom(void * Pthis);
```

`Pthis` must point to a storage buffer of size `MOTHER_BUFFERSIZE` bytes. Use a separate buffer for each thread.

Thread-safe, C++, static link version

Class definition in `asmlibran.h`.

```
class CRandomMotherA {
public:
   CRandomMotherA(int seed);
   void RandomInit(int seed);
   int IRandom(int min, int max);
```

```
   double Random();
   uint32_t BRandom();
private:
   char internals[MOTHER_BUFFERSIZE];
};
```

Make one instance of the class for each thread.

Thread-safe, C++ source code

Class definition in `randomc.h`. Source code in `mersenne.cpp`.
See `randomc.zip`.

6.3 SFMT generator and combined generator

Set the parameter `IncludeMother` to 0 (default) to get the SFMT generator alone, or 1 to combine the SFMT generator with the Mother-of-all generator.

Single threaded, C language, static link version

Function prototypes in `asmlibran.h`.

```
void SFMTgenRandomInit(int seed, int IncludeMother = 0);
void SFMTgenRandomInitByArray(int const seeds[], int NumSeeds,
     int IncludeMother = 0);
int    SFMTgenIRandom (int min, int max);
int    SFMTgenIRandomX(int min, int max);
double SFMTgenRandom();
long double SFMTgenRandomL();
uint32_t SFMTgenBRandom();
```

Single threaded, Windows DLL version

Function prototypes in `asmlibran.h`.

```
void __stdcall SFMTgenRandomInitD(int seed, int IncludeMother);
void __stdcall SFMTgenRandomInitByArrayD(int const seeds[],
               int NumSeeds, int IncludeMother);
int  __stdcall SFMTgenIRandomD (int min, int max);
int  __stdcall SFMTgenIRandomXD(int min, int max);
double   __stdcall SFMTgenRandomD();
uint32_t __stdcall SFMTgenBRandomD();
```

Thread-safe, C language, static link version

Function prototypes in `asmlibran.h`.

```
void   SFMTRandomInit(void * Pthis, int ThisSize, int seed,
       int IncludeMother = 0);
void   SFMTRandomInitByArray(void * Pthis, int ThisSize,
       int const seeds[], int NumSeeds, int IncludeMother = 0);
int    SFMTIRandom (void * Pthis, int min, int max);
int    SFMTIRandomX(void * Pthis, int min, int max);
double SFMTRandom  (void * Pthis);
long double SFMTRandomL (void * Pthis);
uint32_t SFMTBRandom (void * Pthis);
```

`Pthis` must point to a storage buffer of size `SFMT_BUFFERSIZE` bytes. Use a separate buffer for each thread.

Thread-safe, C++, static link version

Class definition in `asmlibran.h`.

```
class CRandomSFMTA {
public:
   CRandomSFMTA(int seed, int IncludeMother = 0);
   void RandomInit(int seed, int IncludeMother = 0);
   void RandomInitByArray(int const seeds[], int NumSeeds,
        int IncludeMother = 0);
   int IRandom (int min, int max);
   int IRandomX(int min, int max);
   double Random();
   long double RandomL();
   uint32_t BRandom();
private:
   char internals[SFMT_BUFFERSIZE];
};
```

Make one instance of the class for each thread.

```
class CRandomSFMTA1 : public CRandomSFMTA {
public:
   CRandomSFMTA1(int seed) : CRandomSFMTA(seed,1) {}
};
```

Combined generator. Same as `CRandomSFMTA` with Mother-of-all generator included.

Thread-safe, C++ source code

Class definition in `sfmt.h`. Source code in `sfmt.cpp`.
See `randomc.zip`.

6.4 PhysicalSeed

This function generates non-deterministic random integers based on the random motion of electrons if this feature is supported by the CPU. It is useful for generating seeds for the pseudo random number generators listed above.

Function prototype

```
int PhysicalSeed(int seeds[], int NumSeeds);
```

Description

Generates random integers. This function uses a physical and non-deterministic source of random numbers if possible. The array `seeds` will be filled with random 32-bit integers. `NumSeeds` is the desired number of random integers to put into the array `seeds`. It is the responsibility of the programmer that the `seeds` array has at least `NumSeeds` elements.

The return value indicates the method used:
0 Failure. No suitable instruction available, or method failed.
1 No physical random number generator available. Uses internal clock counter instead.
2 VIA physical random number generator used.
3 RDRAND instruction used.
4 RDSEED instruction used.

The return value will indicate the best available method if `NumSeeds` is zero.

All modern microprocessors can be expected to return a value of at least 1. You need to consider the situation where the return value is 1. This indicates that the processor has no physical random number generator. Instead the function uses the internal clock counter in the CPU. The clock counter returns the number of clock cycles since the computer was last turned on.

The resolution of the internal clock counter is determined by the CPU clock frequency. If, for example, the CPU frequency is 2 GHz, then the resolution is 0.5 nanoseconds. This can provide a good seed for a random number generator if the event somehow depends on the time of a command from a human user. No human is able to press a button with nanosecond precision. However, any subsequent calls will not be independent of the first one because it will be equal to the preceding value plus the time it takes to execute the code until the next call. While the time it takes to execute a piece of code is not always the same, it might possibly be exactly the same as last time this code executed. If you need another seed that is independent of the first one then you must wait for a new user input or some other external event that has no precise timing. Only the first two elements in `seeds` will have nonzero values if the return value is 1. The first element has a resolution defined by the internal clock frequency. The second element contains the upper 32 bits of the 64-bit clock count if `NumSeeds` > 1.

If the return value is 2 or more then you can generate any number of random integers. This function is slower than a pseudo random number generator. Therefore, it is recommended to use the `PhysicalSeed` function only to generate a seed for a pseudo random number generator and then use the pseudo random number generator for making a sequence of random numbers.

Versions included
Stdcall version: Yes (`PhysicalSeedD`)

7 Patches for Intel compiler and libraries

In many cases, Intel compilers generate code that performs poorly on non-Intel CPU's. The compiler inserts a function that checks the CPU brand and selects a code path for an inferior instruction set if the CPU is not an Intel. The same applies to some function libraries from Intel, even if they are used with a different compiler. See http://www.agner.org/optimize/blog/read.php?i=49 and http://www.agner.org/optimize/#manual_cpp for further discussion of this issue.

The included file inteldispatchpatch.zip includes code that can be used to circumvent this problem and improve the compatibility of the code with CPU's from other vendors than Intel. See the file dispatchpatch.txt for instructions.

8 File list

Files in asmlib.zip

asmlib-instructions.pdf	This file
asmlib.h	C/C++ Header file for asmlib functions
asmlibran.h	C/C++ Header file for random number generators
libaelf32.a	Library 32-bit ELF format
libaelf32o.a	Library 32-bit ELF format, override standard library
libaelf64.a	Library 64-bit ELF format

libaelf64o.a	Library 64-bit ELF format, override standard library
libamac32.a	Library 32-bit Mach-O format
libamac32o.a	Library 32-bit Mach-O format, override standard library
libamac64.a	Library 64-bit Mach-O format
libamac64o.a	Library 64-bit Mach-O format, override standard library
libad32.dll	Library 32-bit Windows DLL
libad32.lib	Import library for libad32.dll
libad64.dll	Library 64-bit Windows DLL
libad64.lib	Import library for libad64.dll
libacof32.lib	Library 32-bit COFF format
libacof32o.lib	Library 32-bit COFF format, override standard library
libacof64.lib	Library 64-bit COFF format
libacof64o.lib	Library 64-bit COFF format, override standard library
libaomf32.lib	Library, 32-bit OMF format
libaomf32o.lib	Library, 32-bit OMF format, override standard library
license.txt	Gnu general public license
asmlibSrc.zip	Source code
inteldispatchpatch.zip	Alternative CPU dispatchers for improving the compatibility of Intel function libraries with non-Intel CPUs

Files in asmlibSrc.zip

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
dispatchpatch32/64.asm	Source code for object files in inteldispatchpatch.zip
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
libad32.asm libad64.asm	Source code for DLL entry
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
mersenne32.asm mersenne64.asm	Source code for Mersenne twister
mother32.asm mother64.asm	Source code for Mother-of-all generator
physseed32.asm physseed64.asm	Source code for PhysicalSeed 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
sfmt32.asm sfmt64.asm	Source code for SFMT generator
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

randomah.asi	Source code for pseudo random number generators
testalib.cpp	Test example
testmem.cpp	Example for testing memory functions
testrandom.cpp	Example for testing random generator functions
libad32.def	Exports definition function for libad32.dll
libad64.def	Exports definition function for libad64.dll
MakeAsmlib.bat	Batch file for making asmlib
asmlib.make	Makefile for making asmlib

Files in inteldispatchpatch.zip

dispatchpatch.txt	Instructions: Patches for improving the compatibility of Intel function libraries and Intel compiler generated code with non-Intel CPUs
intel_cpu_feature_patch.c	Workaround for Intel compiler and SVML library
intel_mkl_feature_patch.c	Workaround for Intel math kernel library (MKL)
dispatchpatch32.o	Dispatchers for MKL/VML, Linux, 32 bit
dispatchpatch64.o	Dispatchers for MKL/VML, Linux, 64 bit
dispatchpatch32.obj	Dispatchers for MKL/VML, Windows, 32 bit
dispatchpatch64.obj	Dispatchers for MKL/VML, Windows, 64 bit
dispatchpatch32.mac.o	Dispatchers for MKL/VML, Mac, 32 bit
dispatchpatch64.mac.o	Dispatchers for MKL/VML, Mac, 64 bit

9 Change log

Version 2.52. 2018-04-25:
Minor changes in InstructionSet function.

Version 2.51. 2016-11-16:
AVX512F version of memmove.
Fixed bug in SetMemcpyCacheLimit.

Version 2.50. 2016-11-09:
AVX512F version of memcpy, memset, and memcmp.
AVX512BW version of memcpy, memmove, memset, and memcmp.
InstructionSet function added value 16.
Position-independent 32-bit versions of libraries no longer included.
Fixed bug in strCountInSetGeneric

Version 2.32. 2013-08-21:
AVX version of memcpy, memmove and memset improved for some Intel processors.
InstructionSet function added values 14 and 15.

Version 2.20. 2011-07-06:
Assembly code switched to NASM syntax.

10 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.

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.

11 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.