# Introduction to x64 Assembly

## Introduction

For years, PC programmers used x86 assembly to write performance-critical code. However, 32-bit PCs are being replaced with 64-bit ones, and the underlying assembly code has changed. This Gem is an introduction to x64 assembly. No prior knowledge of x86 code is needed, although it makes the transition easier.

x64 is a generic name for the 64-bit extensions to Intel's and AMD's 32-bit x86 instruction set architecture (ISA). AMD introduced the first version of x64, initially called x86-64 and later renamed AMD64. Intel named their implementation IA-32e and then EMT64. There are some slight incompatibilities between the two versions, but most code works fine on both versions; details can be found in the Intel® 64 and IA-32 Architectures Software Developer's Manuals and the AMD64 Architecture Tech Docs. We call this intersection flavor x64. Neither is to be confused with the 64-bit Intel® Itanium® architecture, which is called IA-64.

This Gem won't cover hardware details such as caches, branch prediction, and other advanced topics. Several references will be given at the end of the article for further reading in these areas.

Assembly is often used for performance-critical parts of a program, although it is difficult to outperform a good C++ compiler for most programmers. Assembly knowledge is useful for debugging code – sometimes a compiler makes incorrect assembly code and stepping through the code in a debugger helps locate the cause. Code optimizers sometimes make mistakes. Another use for assembly is interfacing with or fixing code for which you have no source code. Disassembly lets you change/fix existing executables. Assembly is necessary if you want to know how your language of choice works under the hood – why some things are slow and others are fast. Finally, assembly code knowledge is indispensable when diagnosing malware.

## Architecture

When learning assembly for a given platform, the first place to start is to learn the register set.

*General Architecture*

Since the 64-bit registers allow access for many sizes and locations, we define a byte as 8 bits, a word as 16 bits, a double word as 32 bits, a quadword as 64 bits, and a double quadword as 128 bits. Intel stores bytes "little endian," meaning lower significant bytes are stored in lower memory addresses.

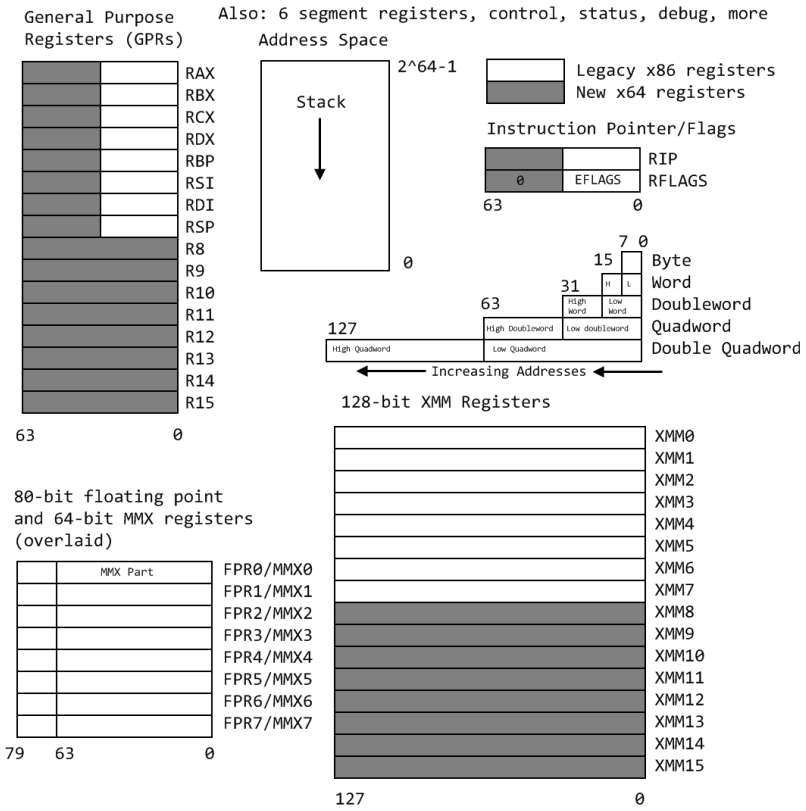


**Figure 1 – General Architecture**

Figure 1 shows sixteen general purpose 64-bit registers, the first eight of which are labeled (for historical reasons) RAX, RBX, RCX, RDX, RBP, RSI, RDI, and RSP. The second eight are named R8-R15. By replacing the initial R with an E on the first eight registers, it is possible to access the lower 32 bits (EAX for RAX). Similarly, for RAX, RBX, RCX, and RDX, access to the lower 16 bits is possible by removing the initial R (AX for RAX), and the lower byte of the these by switching the X for L (AL for AX), and the higher byte of the low 16 bits using an H (AH for AX). The new registers R8 to R15 can be accessed in a similar manner like this: R8 (qword), R8D (lower dword), R8W (lowest word), R8B (lowest byte MASM style, Intel style R8L). Note there is no R8H.

There are odd limitations accessing the byte registers due to coding issues in the REX opcode prefix used for the new registers: an instruction cannot reference a legacy high byte (AH, BH, CH, DH) and one of the new byte registers at the same time (such as R11B), but it can use legacy low bytes (AL, BL, CL, DL). This is enforced by changing (AH, BH, CH, DH) to (BPL, SPL, DIL, SIL) for instructions using a REX prefix.

The 64-bit instruction pointer RIP points to the next instruction to be executed, and supports a 64-bit flat memory model. Memory address layout in current operating systems is covered later.

The stack pointer RSP points to the last item pushed onto the stack, which grows toward lower addresses. The stack is used to store return addresses for subroutines, for passing parameters in higher level languages such as C/C++, and for storing "shadow space" covered in calling conventions.

The RFLAGS register stores flags used for results of operations and for controlling the processor. This is formed from the x86 32-bit register EFLAGS by adding a higher 32 bits which are reserved and currently unused. Table 1 lists the most useful flags. Most of the other flags are used for operating system level tasks and should always be set to the value previously read.

**Table 1 – Common Flags**

| Symbol | Bit | Name | Set if…. |
|---|---|---|---|
| CF | 0 | Carry | Operation generated a carry or borrow |
| PF | 2 | Parity | Last byte has even number of 1's, else 0 |
| AF | 4 | Adjust | Denotes Binary Coded Decimal in-byte carry |
| ZF | 6 | Zero | Result was 0 |
| SF | 7 | Sign | Most significant bit of result is 1 |
| OF | 11 | Overflow | Overflow on signed operation |
| | | | |
| DF | 10 | Direction | Direction string instructions operate (increment or decrement) |
| ID | 21 | Identification | Changeability denotes presence of CPUID instruction |

The floating point unit (FPU) contains eight registers FPR0-FPR7, status and control registers, and a few other specialized registers. FPR0-7 can each store one value of the types shown in Table 2. Floating point operations conform to IEEE 754. Note that most C/C++ compilers support the 32 and 64 bit types as float and double, but not the 80-bit one available from assembly. These registers share space with the eight 64-bit MMX registers.

**Table 2 – Floating Point Types**

| Data Type | Length | Precision (bits) | Decimal digits Precision | Decimal Range |
|---|---|---|---|---|
| Single Precision | 32 | 24 | 7 | 1.18*10^-38 to 3.40*10^38 |
| Double Precision | 64 | 53 | 15 | 2.23 *10^-308 to 1.79*10^308 |
| Extended Precision | 80 | 64 | 19 | 3.37*10^-4932 to 1.18*10^4932 |

Binary Coded Decimal (BCD) is supported by a few 8-bit instructions, and an oddball format supported on the floating point registers gives an 80 bit, 17 digit BCD type.

The sixteen 128-bit XMM registers (eight more than x86) are covered in more detail.

Final registers include segment registers (mostly unused in x64), control registers, memory management registers, debug registers, virtualization registers, performance registers tracking all sorts of internal parameters (cache hits/misses, branch hits/misses, micro-ops executed, timing,

and much more). The most notable performance opcode is RDTSC, which is used to count processor cycles for profiling small pieces of code.

Full details are available in the five-volume set "Intel® 64 and IA-32 Architectures Software Developer's Manuals" at http://www.intel.com/products/processor/manuals/. They are available for free download as PDF, order on CD, and often can be ordered for free as a hardcover set when listed.

### *SIMD Architecture*

Single Instruction Multiple Data (SIMD) instructions execute a single command on multiple pieces of data in parallel and are a common usage for assembly routines. MMX and SSE commands (using the MMX and XMM registers respectively) support SIMD operations, which perform an instruction on up to eight pieces of data in parallel. For example, eight bytes can be added to eight bytes in one instruction using MMX.

The eight 64-bit MMX registers MMX0-MMX7 are aliased on top of FPR0-7, which means any code mixing FP and MMX operations must be careful not to overwrite required values. The MMX instructions operate on integer types, allowing byte, word, and doubleword operations to be performed on values in the MMX registers in parallel. Most MMX instructions begin with 'P' for "packed". Arithmetic, shift/rotate, comparison, e.g.: PCMPGTB "Compare packed signed byte integers for greater than".

The sixteen 128-bit XMM registers allow parallel operations on four single or two double precision values per instruction. Some instructions also work on packed byte, word, doubleword, and quadword integers. These instructions, called the Streaming SIMD Extensions (SSE), come in many flavors: SSE, SSE2, SSE3, SSSE3, SSE4, and perhaps more by the time this prints. Intel has announced more extensions along these lines called Intel® Advanced Vector Extensions (Intel® AVX), with a new 256-bit-wide datapath. SSE instructions contain move, arithmetic, comparison, shuffling and unpacking, and bitwise operations on both floating point and integer types. Instruction names include such beauties as PMULHUW and RSQRTPS. Finally, SSE introduced some instructions for memory pre-fetching (for performance) and memory fences (for multi-threaded safety).

Table 3 lists some command sets, the register types operated on, the number of items manipulated in parallel, and the item type. For example, using SSE3 and the 128-bit XMM registers, you can operate on 2 (must be 64-bit) floating point values in parallel, or even 16 (must be byte sized) integer values in parallel.

To find which technologies a given chip supports, there is a CPUID instruction that returns processor-specific information.

**Table 3**

| Technology | Register size/type | Item type | Items in Parallel |
|---|---|---|---|
| MMX | 64 MMX | Integer | 8, 4, 2, 1 |
| SSE | 64 MMX | Integer | 8,4,2,1 |
| SSE | 128 XMM | Float | 4 |
| SSE2/SSE3/SSSE3… | 64 MMX | Integer | 2,1 |
| SSE2/SSE3/SSSE3… | 128 XMM | Float | 2 |
| SSE2/SSE3/SSSE3… | 128 XMM | Integer | 16,8,4,2,1 |

# Tools

## *Assemblers*

An Internet search reveals x64-capable assemblers such as the Netwide Assembler NASM , a NASM rewrite called YASM, the fast Flat Assembler FASM , and the traditional Microsoft MASM. There is even a free IDE for x86 and x64 assembly called WinASM. Each assembler has varying support for other assemblers' macros and syntax, but assembly code is not source-compatible across assemblers like C++ or Java* are.

For the examples below, I use the 64-bit version of MASM, ML64.EXE, freely available in the platform SDK. For the examples below note that MASM syntax is of the form

```
Instruction   Destination,   Source
```

Some assemblers reverse source and destination, so read your documentation carefully.

## *C/C++ Compilers*

C/C++ compilers often allow embedding assembly in the code using inline assembly, but Microsoft Visual Studio* C/C++ removed this for x64 code, likely to simplify the task of the code optimizer. This leaves two options: use separate assembly files and an external assembler, or use intrinsics from the header file "intrn.h" (see Birtolo and MSDN). Other compilers feature similar options.

Some reasons to use intrinsics:

- Inline asm not supported in x64.
- Ease of use: you can use variable names instead of having to juggle register allocation manually.
- More cross-platform than assembly: the compiler maker can port the intrinsics to various architectures.
- The optimizer works better with intrinsics.

For example, Microsoft Visual Studio* 2008 has an intrinsic

```
unsigned short _rot16(unsigned short a, unsigned char b)
```

which rotates the bits in a 16-bit value right b bits and returns the answer. Doing this in C gives

```
unsigned short a1 = (b>>c)|(b<<(16-c));
```

which expands to fifteen assembly instructions (in debug builds - in release builds whole program optimization made it harder to separate, but it was of a similar length), while using the equivalent intrinsic

```
unsigned short a2 = _rotr16(b,c);
```

expands to four instructions. For more information read the header file and documentation.

# Instruction Basics

## *Addressing Modes*

Before covering some basic instructions, you need to understand addressing modes, which are ways an instruction can access registers or memory. The following are common addressing modes with examples:

- Immediate: the value is stored in the instruction.
  ```
  ADD EAX, 14 ; add 14 into 32-bit EAX
  ```

- Register to register
  ```
  ADD R8L, AL  ; add 8 bit AL into R8L
  ```

- Indirect: this allows using an 8, 16, or 32 bit displacement, any general purpose registers for base and index, and a scale of 1, 2, 4, or 8 to multiply the index. Technically, these can also be prefixed with segment FS: or GS: but this is rarely required.
  ```
  MOV R8W, 1234[8*RAX+RCX] ; move word at address 8*RAX+RCX+1234 into R8W
  ```

  There are many legal ways to write this. The following are equivalent
  ```
  MOV     ECX, dword ptr table[RBX][RDI]
  MOV     ECX, dword ptr table[RDI][RBX]
  MOV     ECX, dword ptr table[RBX+RDI]
  MOV     ECX, dword ptr [table+RBX+RDI]
  ```
  The `dword ptr` tells the assembler how to encode the `MOV` instruction.

- RIP-relative addressing: this is new for x64 and allows accessing data tables and such in the code relative to the current instruction pointer, making position independent code easier to implement.
  ```
  MOV AL, [RIP] ; RIP points to the next instruction aka NOP
  NOP
  ```

  Unfortunately, MASM does not allow this form of opcode, but other assemblers like

FASM and YASM do. Instead, MASM embeds RIP-relative addressing implicitly.

```
MOV EAX, TABLE ; uses RIP- relative addressing to get table address
```

- Specialized cases: some opcodes use registers in unique ways based on the opcode. For example, signed integer division **IDIV** on a 64 bit operand value divides the 128-bit value in **RDX:RAX** by the value, storing the result in **RAX** and the remainder in **RDX**.

## *Instruction Set*

Table 4 lists some common instructions. * denotes this entry is multiple opcodes where the * denotes a suffix.

**Table 4 – Common Opcodes**

| Opcode | Meaning | Opcode | Meaning |
|---|---|---|---|
| MOV | Move to/from/between memory and registers | AND/OR/XOR/NOT | Bitwise operations |
| CMOV* | Various conditional moves | SHR/SAR | Shift right logical/arithmetic |
| XCHG | Exchange | SHL/SAL | Shift left logical/arithmetic |
| BSWAP | Byte swap | ROR/ROL | Rotate right/left |
| PUSH/POP | Stack usage | RCR/RCL | Rotate right/left through carry bit |
| ADD/ADC | Add/with carry | BT/BTS/BTR | Bit test/and set/and reset |
| SUB/SBC | Subtract/with carry | JMP | Unconditional jump |
| MUL/IMUL | Multiply/unsigned | JE/JNE/JC/JNC/J* | Jump if equal/not equal/carry/not carry/ many others |
| DIV/IDIV | Divide/unsigned | LOOP/LOOPE/LOOPNE | Loop with ECX |
| INC/DEC | Increment/Decrement | CALL/RET | Call subroutine/return |
| NEG | Negate | NOP | No operation |
| CMP | Compare | CPUID | CPU information |

A common instruction is the LOOP instruction, which decrements RCX, ECX, or CX depending on usage, and then jumps if the result is not 0. For example,

```
   XOR      EAX, EAX     ; zero out eax
   MOV      ECX, 10      ; loop 10 times
Label:                   ; this is a label in assembly
   INX      EAX          ; increment eax
   LOOP     Label        ; decrement ECX, loop if not 0
```

Less common opcodes implement string operations, repeat instruction prefixes, port I/O instructions, flag set/clear/test, floating point operations (begin usually with a F, and support move, to/from integer, arithmetic, comparison, transcendental, algebraic, and control functions), cache and memory opcodes for multithreading and performance issues, and more. The Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2, in two parts, covers each opcode in detail.

## Operating Systems

64-bit systems allow addressing $2^{64}$ bytes of data in theory, but no current chips allow accessing all 16 exabytes (18,446,744,073,709,551,616 bytes). For example, AMD architecture uses only the lower 48 bits of an address, and bits 48 through 63 must be a copy of bit 47 or the processor raises an exception. Thus addresses are 0 through 00007FFF`FFFFFFFF, and from FFFF8000`00000000 through FFFFFFFF`FFFFFFFF, for a total of 256 TB (281,474,976,710,656 bytes) of usable virtual address space. Another downside is that addressing all 64 bits of memory requires a lot more paging tables for the OS to store, using valuable memory for systems with less than all 16 exabytes installed. Note these are virtual addresses, not physical addresses.

As a result, many operating systems use the higher half of this space for the OS, starting at the top and growing down, while user programs use the lower half, starting at the bottom and growing upwards. Current Windows* versions use 44 bits of addressing (16 terabytes = 17,592,186,044,416 bytes). The resulting addressing is shown in Figure 2. The resulting addresses are not too important for user programs since addresses are assigned by the OS, but the distinction between user addresses and kernel addresses are useful for debugging.

A final OS-related item relates to multithreaded programming, but this topic is too large to cover here. The only mention is that there are memory barrier opcodes for helping to keep shared resources uncorrupted.

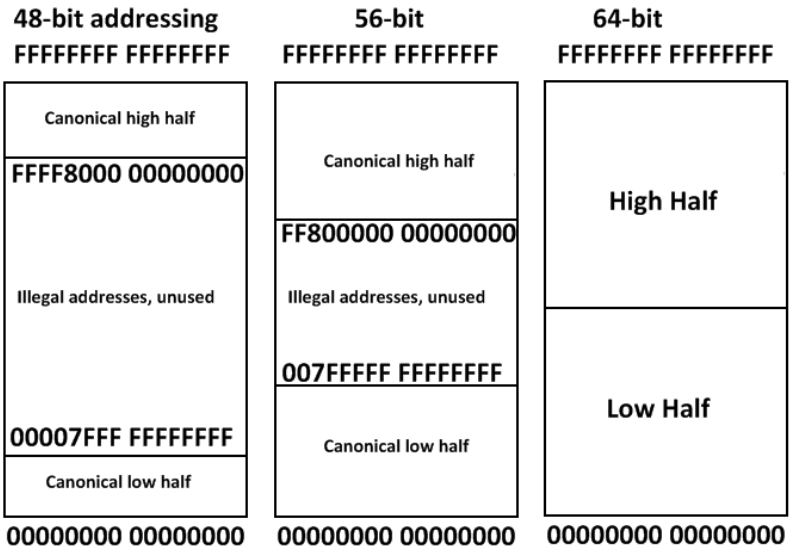


**Figure 2 – Memory Addressing**

## Calling Conventions

Interfacing with operating system libraries requires knowing how to pass parameters and manage the stack. These details on a platform are called a calling convention.

A common x64 calling convention is the Microsoft 64 calling convention used for C style function calling (see MSDN, Chen, and Pietrek). Under Linux* this would be called an Application Binary Interface (ABI). Note the calling convention covered here is different than the one used on x64 Linux* systems.

For the Microsoft* x64 calling convention, the additional register space let fastcall be the only calling convention (under x86 there were many: stdcall, thiscall, fastcall, cdecl, etc.). The rules for interfacing with C/C++ style functions:

- RCX, RDX, R8, R9 are used for integer and pointer arguments in that order left to right.
- XMM0, 1, 2, and 3 are used for floating point arguments.
- Additional arguments are pushed on the stack left to right.
- Parameters less than 64 bits long are not zero extended; the high bits contain garbage.
- It is the caller's responsibility to allocate 32 bytes of "shadow space" (for storing RCX, RDX, R8, and R9 if needed) before calling the function.
- It is the caller's responsibility to clean the stack after the call.
- Integer return values (similar to x86) are returned in RAX if 64 bits or less.
- Floating point return values are returned in XMM0.
- Larger return values (structs) have space allocated on the stack by the caller, and RCX then contains a pointer to the return space when the callee is called. Register usage for integer parameters is then pushed one to the right. RAX returns this address to the caller.
- The stack is 16-byte aligned. The "call" instruction pushes an 8-byte return value, so the all non-leaf functions must adjust the stack by a value of the form 16n+8 when allocating stack space.
- Registers RAX, RCX, RDX, R8, R9, R10, and R11 are considered volatile and must be considered destroyed on function calls.
- RBX, RBP, RDI, RSI, R12, R14, R14, and R15 must be saved in any function using them.
- Note there is no calling convention for the floating point (and thus MMX) registers.
- Further details (varargs, exception handling, stack unwinding) are at Microsoft's site.

## Examples

Armed with the above, here are a few examples showing x64 usage. The first is a simple x64 standalone assembly program that pops up a Windows MessageBox.

```
; Sample x64 Assembly Program
; Chris Lomont 2009 www.lomont.org
extrn ExitProcess: PROC   ; external functions in system libraries
extrn MessageBoxA: PROC
```

```
.data
caption db '64-bit hello!', 0
message db 'Hello World!', 0

.code
Start PROC
  sub    rsp,28h      ; shadow space, aligns stack
  mov    rcx, 0       ; hWnd = HWND_DESKTOP
  lea    rdx, message ; LPCSTR lpText
  lea    r8,  caption ; LPCSTR lpCaption
  mov    r9d, 0       ; uType = MB_OK
  call   MessageBoxA  ; call MessageBox API function
  mov    ecx, eax     ; uExitCode = MessageBox(...)
  call ExitProcess
Start ENDP
End
```

Save this as hello.asm, compile this with ML64, available in the Microsoft Windows* x64 SDK as follows:

```
ml64 hello.asm /link /subsystem:windows /defaultlib:kernel32.lib
/defaultlib:user32.lib /entry:Start
```

which makes a windows executable and links with appropriate libraries. Run the resulting executable hello.exe and you should get the message box to pop up.

The second example links an assembly file with a C/C++ file under Microsoft Visual Studio* 2008. Other compiler systems are similar. First make sure your compiler is an x64-capable version. Then

1. Create a new empty C++ console project. Create a function you'd like to port to assembly, and call it from main.
2. To change the default 32-bit build, select Build/Configuration Manager.
3. Under Active Platform, select New…
4. Under Platform, select x64. If it does not appear figure out how to add the 64-bit SDK tools and repeat.
5. Compile and step into the code. Look under Debug/Windows/Disassembly to see the resulting code and interface needed for your assembly function.
6. Create an assembly file, and add it to the project. It defaults to a 32 bit assembler which is fine.
7. Open the assembly file properties, select all configurations, and edit the custom build step.
8. Put command line
   ```
   ml64.exe /DWIN_X64 /Zi /c /Cp /Fl /Fo $(IntDir)\$(InputName).obj
   $(InputName).asm
   ```

and set outputs to
**$(IntDir)\$(InputName).obj**

9. Build and run.

For example, in main.cpp we put a function **CombineC** that does some simple math on five integer parameters and one double parameter, and returns a double answer. We duplicate that functionality in assembly in a separate file CombineA.asm in a function called **CombineA**. The C++ file is:

```
// C++ code to demonstrate x64 assembly file linking
#include <iostream>
using namespace std;
double CombineC(int a, int b, int c, int d, int e, double f)
{
     return (a+b+c+d+e)/(f+1.5);
}

// NOTE: extern "C" needed to prevent C++ name mangling
extern "C" double CombineA(int a, int b, int c, int d, int e, double
f);

int main(void)
{
     cout << "CombineC: " << CombineC(1,2,3,4, 5, 6.1) << endl;
     cout << "CombineA: " << CombineA(1,2,3,4, 5, 6.1) << endl;
     return 0;
}
```

Be sure to make functions extern "C" linkage to prevent C++ name mangling. Assembly file CombineA.asm contains

```
; Sample x64 Assembly Program
.data
realVal REAL8 +1.5 ; this stores a real number in 8 bytes

.code
PUBLIC CombineA
CombineA PROC
   ADD    ECX, DWORD PTR [RSP+28H] ; add overflow parameter to first
parameter
   ADD    ECX, R9D                 ; add other three register parameters
   ADD    ECX, R8D                 ;
   ADD    ECX, EDX                 ;
   MOVD   XMM0, ECX                ; move doubleword ECX into XMM0
   CVTDQ2PD  XMM0, XMM0            ; convert doubleword to floating point
   MOVSD  XMM1, realVal            ; load 1.5
   ADDSD  XMM1, MMWORD PTR [RSP+30H]  ; add parameter
   DIVSD  XMM0, XMM1               ; do division, answer in xmm0
```

```
   RET                                 ; return
CombineA ENDP

End
```

Running this should result in the value 1.97368 being output twice.

## Conclusion

This has been a necessarily brief introduction to x64 assembly programming. The next step is to browse the Intel® 64 and IA-32 Architectures Software Developer's Manuals. Volume 1 contains the architecture details and is a good start if you know assembly. Other places are assembly books or online assembly tutorials. To get an understanding of how your code executes, it is instructive to step through code in debugger, looking at the disassembly, until you can read assembly code as well as your favorite language. For C/C++ compilers, debug builds are much easier to read than release builds so be sure to start there. Finally, read the forums at masm32.com for a lot of material.

## References


"AMD64 Architecture Tech Docs," available online at http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_875_7044,00.html

NASM: http://www.nasm.us/

YASM: http://www.tortall.net/projects/yasm/

Flat Assembler (FASM): http://www.flatassembler.net/

Dylan Birtolo, "New Intrinsic Support in Visual Studio 2008", available online at http://blogs.msdn.com/vcblog/archive/2007/10/18/new-intrinsic-support-in-visual-studio-2008.aspx

Raymond Chen, "The history of calling conventions, part 5: amd64," available online at http://blogs.msdn.com/oldnewthing/archive/2004/01/14/58579.aspx

"Intel® 64 and IA-32 Architectures Software Developer's Manuals," available online at http://www.intel.com/products/processor/manuals/

"Compiler Intrinsics", available online at http://msdn.microsoft.com/en-us/library/26td21ds.aspx

"Calling Convention", available online at http://msdn.microsoft.com/en-us/library/9b372w95.aspx

Matt Pietrek, “Everything You Need To Know To Start Programming 64-Bit Windows Systems”, available online at http://msdn.microsoft.com/en-us/magazine/cc300794.aspx, 2009.