Assembly Language Succinctly

by Chris Rose

Syncfusion | Technology Resource Portal

Assembly Language Succinctly

By

Christopher Rose

Foreword by Daniel Jebaraj

Technical Reviewer: Jarred Capellman
Copy Editor: Ben Ball
Acquisitions Coordinator: Jessica Rightmer, senior marketing strategist, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Chris Rose is an Australian software engineer. His background is mainly in data mining and charting software for medical research. He has also developed desktop and mobile apps and a series of programming videos for an educational channel on YouTube. He is a musician and can often be found accompanying silent films at the Pomona Majestic Theatre in Queensland.

Introduction

Assembly Language

This book is an introduction to x64 assembly language. This is the language used by almost all modern desktop and laptop computers. x64 is a generic term for the newest generation of the x86 CPU used by AMD, Intel, VIA, and other CPU manufacturers. x64 assembly has a steep learning curve and very few concepts from high-level languages are applicable. It is the most powerful language available to x64 CPU programmers, but it is not often the most practical language.

An assembly language is the language of a CPU, but the numbers of the machine code are replaced by easy-to-remember mnemonics. Instead of programming using pure hexadecimal, such as **`83 C4 04`**, programmers can use something easier to remember and read, such as **`ADD ESP, 4`**, which adds 4 to ESP. The human readable version is read by a program called an assembler, and then it is translated into machine code by a process called assembling (analogous to compiling in high-level languages). A modern assembly language is the result of both the physical CPU and the assembler. Modern assembly languages also have high-level features such as macros and user-defined data types.

Why Learn Assembly?

Many high-level languages (Java, C#, Python, etc.) share common characteristics. If a programmer is familiar with any one of them, then he or she will have no trouble picking up one of the others after a few weeks of study. Assembly language is very different; it shares almost nothing with high-level languages. Assembly languages for different CPU architectures often have little in common. For instance, the MIPS R4400 assembly language is very different from the x86 language. There are no compound statements. There are no if statements, and the goto instruction (**`JMP`**) is used all the time. There are no objects, and there is no type safety. Programmers have to build their own looping structures, and there is no difference between a float and an int. There is nothing to assist programmers in preventing logical errors, and there is no difference between execute instructions and data. There are many differences between assembly languages.

I could go on forever listing the useful features that x64 assembly language is missing when compared to high-level languages, but in a sense, this means that assembly language has fewer obstacles. Type safety, predefined calling conventions, and separating code from data are all restrictions. These restrictions do not exist in assembly; the only restrictions are those imposed by the hardware itself. If the machine is capable of doing something, it can be told to do so using its own assembly language.

A French person might know English as their second language and they could be instructed to do a task in English, but if the task is too complicated, some concepts may be lost in translation. The best way to explain how to perform a complex task to a French person is to explain it in French. Likewise, C++ and other high-level languages are not the CPU's native language. The computer is very good at taking instructions in C++, but when you need to explain exactly how to do something very complicated, the CPU's native language is the only option.

Another important reason to learn an assembly language is simply to understand the CPU. A CPU is not distinct from its assembly language. The language is etched into the silicon of the CPU itself.

Intended Audience

This book is aimed at developers using Microsoft's Visual Studio. This is a versatile and very powerful assembly language IDE. This book is targeted at programmers with a good foundation in C++ and a desire to program native assembly using the Visual Studio IDE (professional versions and the express editions). The examples have been tested using Visual Studio and the assembler that comes bundled with it, ML64.exe (the 64-bit version of MASM, Microsoft's Macro Assembler).

Having knowledge of assembly language programming also helps programmers understand high-level languages like Java and C#. These languages are compiled to virtual machine code (Java Byte Code for Java and CIL or Common Intermediate Language for .NET languages). The virtual machine code can be disassembled and examined from .NET executables or DLL files using the ILDasm.exe tool, which comes with Visual Studio. When a .NET application is executed by another tool, ILAsm.exe, it translates the CIL machine code into native x86 machine code, which is then executed by the CPU. CIL is similar to an assembly language, and a thorough knowledge of x86 assembly makes most of CIL readable, even though they are different languages. This book is focused on C++, but this information is similarly applicable to programming high-level languages.

This book is about the assembly language of most desktop and laptop PCs. Almost all modern desktop PCs have a 64-bit CPU based on the x86 architecture. The legacy 32-bit and 16-bit CPUs and their assembly languages will not be covered in any great detail.

MASM uses Intel syntax, and the code in this book is not compatible with AT&T assemblers. Most of the instructions are the same in other popular Intel syntax assemblers, such as YASM and NASM, but the directive syntax for each assembler is different.

Chapter 1 Assembly in Visual Studio

There would be little point in describing x64 assembly language without having examined a few methods for coding assembly. There are a number of ways to code assembly in both 32-bit and 64-bit applications. This book will mostly concentrate on 64-bit assembly, but first let us examine some ways of coding 32-bit assembly, since 32-bit x86 assembly shares many characteristics with 64-bit x86.

Inline Assembly in 32-Bit Applications

Visual C++ Express and Visual Studio Professional allow what is called inline assembly in 32-bit applications. I have used Visual Studio 2010 for the code in this book, but the steps are identical for newer versions of the IDE. All of this information is applicable to users of Visual Studio 2010, 2012, and 2013, both Express and Professional editions. Inline assembly is where assembly code is embedded into otherwise normal C++ in either single lines or code blocks marked with the **__asm** keyword.

Note: You can also use _asm with a single underscore at the start. This is an older directive maintained for backwards compatibility. Initially the keyword was asm with no leading underscores, but this is no longer accepted by Visual Studio.

You can inject a single line of assembly code into C++ code by using the **__asm** keyword without opening a code block. Anything to the right of this keyword will be treated by the C++ compiler as native assembly code.

```
	int i = 0;
	_asm mov i, 25		// Inline assembly for i = 25
	cout<<"The value of i is: "<<i<<endl;
```

You can inject multiple lines of assembly code into regular C++. This is achieved by placing the **__asm** keyword and opening a code block directly after it.

```
float Sqrt(float f) {
	__asm {
		fld f		// Push f to x87 stack
		fsqrt		// Calculate sqrt
		}
	}
```

There are several benefits to using inline assembly instead of a native 32-bit assembly file. Passing parameters to procedures is handled entirely by the C++ compiler, and the programmer can refer to local and global variables by name. In native assembly, the stack must be manipulated manually. Parameters passed to procedures, as well as local variables, must be referred to as offsets from the **`RSP`** (stack pointer) or the **`RBP`** (base pointer). This requires some background knowledge.

There is absolutely no overhead for using inline assembly. The C++ compiler will inject the exact machine code the inline assembly generates into the machine code it is generating from the C++ source. Some things are simply easier to describe in assembly, and it is sometimes not convenient to add an entire native assembly file to a project.

Another benefit of inline assembly is that it uses the same commenting syntax as C++ since we have not actually left the C++ code file. Not having to add separate assembly source code files to a project may make navigating the project easier and enable better maintainability.

The downside to using inline assembly is that programmers lose some of the control they would have otherwise. They lose the ability to manually manipulate the stack and define their own calling convention, as well as the ability to describe segments in detail. The most important compromise is in Visual Studio's lack of support for x64 inline assembly. Visual Studio does not support inline assembly for 64-bit applications, so any programs with inline assembly will already be obsolete because they are confined to the legacy 32-bit x86. This may not be a problem, since applications that require the larger addressing space and registers provided by x64 are rare.

Native Assembly Files in C++

Inline assembly offers a good deal of flexibility, but there are some things that programmers cannot access with inline assembly. For this reason, it is common to add a separate, native assembly code file to your project.

Visual Studio Professional installs all the components to easily change a project's target CPU from 32-bit to 64-bit, but the express versions of Visual C++ require the additional installation of the Windows 7 SDK.

Note: If you are using Visual C++ Express, download and install the latest Windows 7 SDK (version 7.1 or higher for .NET 4).

You will now go through a guide on how to add a native assembly to a simple C++ project.

1. Create a new **Empty C++** project. I have created an empty project for this example, but adding assembly files to Windows applications is the same.

2. Add a C++ file to your project called **main.cpp**. As mentioned previously, this book is not about making entire applications in assembly. For this reason, we shall make a basic C++ front end that calls upon assembly whenever it requires more performance.

3. Right-click on your project name in the Solution Explorer and choose **Build Customizations...**. The build customizations are important because they contain the rules for how Visual Studio deals with assembly files. We do not want the C++ compiler to compile .asm files, we wish for Visual Studio to give these files to MASM for assembling. MASM assembles the .asm files, and they are linked with the C++ files after compilation to form the final executable.

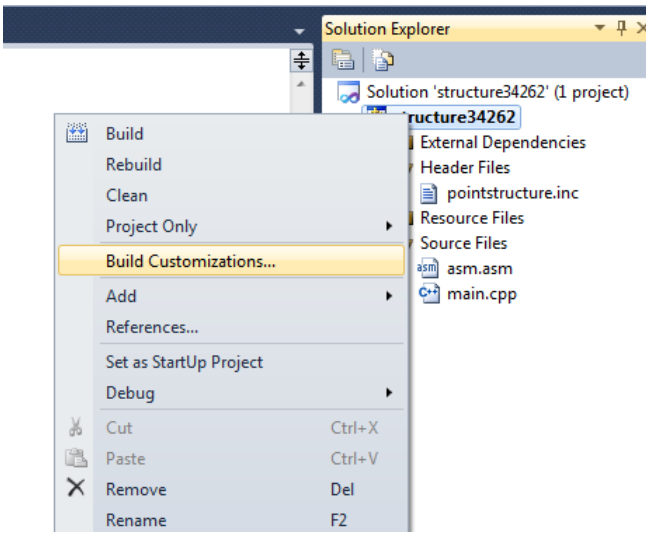
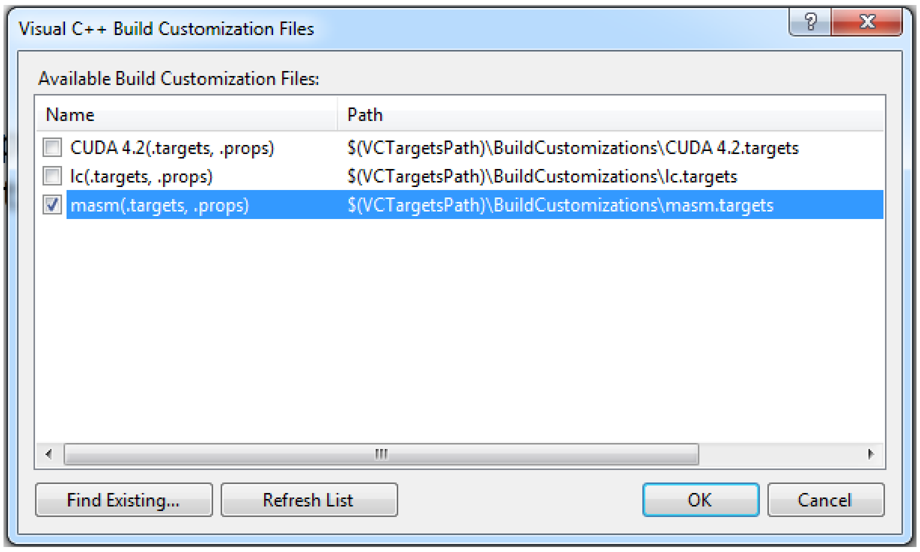


Figure 1

4. Select the box named **masm (.targets, .props)**. It is important to do this step prior to actually adding an assembly code file, because Visual Studio assigns what is to be done with a file when the file is created, not when the project is built.



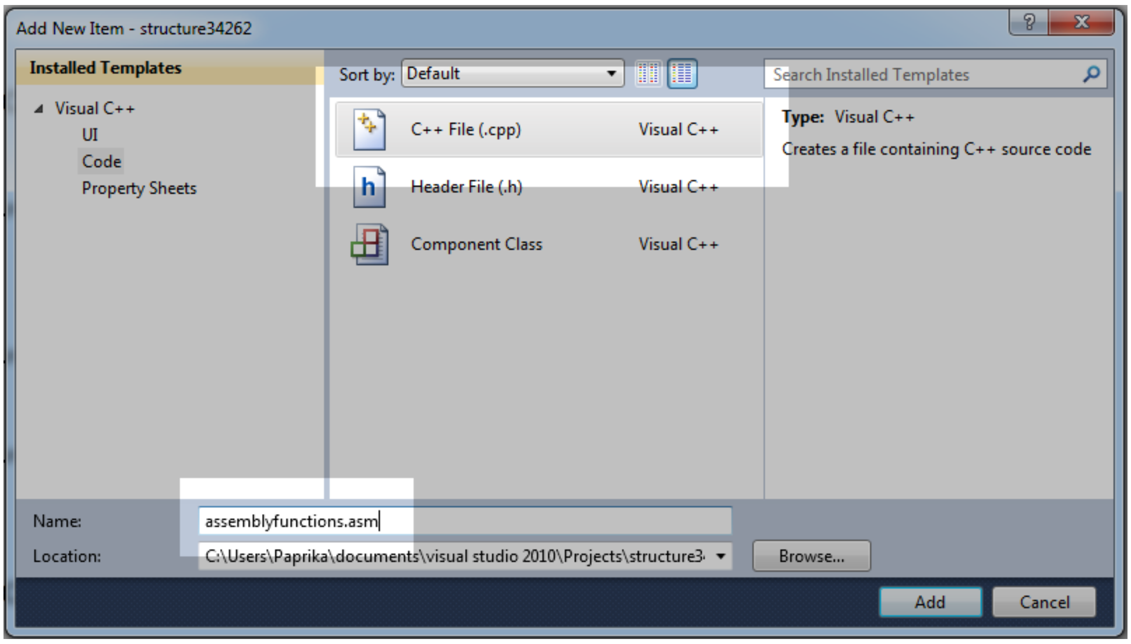
Visual C++ Build Customization Files

Available Build Customization Files:

Name	Path
☐ CUDA 4.2(.targets, .props)	$(VCTargetsPath)\BuildCustomizations\CUDA 4.2.targets
☐ lc(.targets, .props)	$(VCTargetsPath)\BuildCustomizations\lc.targets
☑ masm(.targets, .props)	$(VCTargetsPath)\BuildCustomizations\masm.targets

Figure 2

5. Add another C++ code file, this time with an .asm extension. I have used asmfunctions.asm for my second file name in the sample code). The file name can be anything other than the name you selected for your main program file. Do not name your assembly file main.asm because the compiler may have trouble identifying where your main method is.


Add New Item - structure34262

Installed Templates

Visual C++ > Code

Sort by: Default

C++ File (.cpp)	Visual C++
Header File (.h)	Visual C++
Component Class	Visual C++

Type: Visual C++

Creates a file containing C++ source code

Name: assemblyfunctions.asm

Location: C:\Users\Paprika\documents\visual studio 2010\Projects\structure3

Figure 3

Note: If your project is 32-bit, then you should be able to compile the following 32-bit test program (the code is presented in step six). This small application passes a list of integers from C++ to assembly. It uses a native assembly procedure to find the smallest integer of the array.

Note: If you are compiling to 64-bit, then this program will not work with 32-bit MASM, since 64-bit MASM requires different code. For more information on using 64-bit MASM, please read the [Additional Steps for x64](#) section where setting up a 64-bit application for use with native assembly is explained.

6. Type the 32-bit sample code into each of the source code files you have created. The first listing is for the C++ file and the second is for assembly.

```
// Listing: Main.cpp
#include <iostream>


using namespace std;


// External procedure defined in asmfunctions.asm
extern "C" int FindSmallest(int* i, int count);


int main() {
	int arr[] = { 4, 2, 6, 4, 5, 1, 8, 9, 5, -5 };
```

```
       cout<<"Smallest is "<<FindSmallest(arr, 10)<<endl;


       cin.get();


       return 0;
}
```

```
; asmfunctions.asm
.xmm
.model flat, c


.data


.code
FindSmallest proc export
       mov edx, dword ptr [esp+4] ; edx = *int
       mov ecx, dword ptr [esp+8] ; ecx = Count


       mov eax, 7fffffffh  ; eax will be our answer


       cmp ecx, 0          ; Are there 0 items?
       jle Finished ; If so we're done


MainLoop:
       cmp dword ptr [edx], eax   ; Is *edx < eax?
       cmovl eax, dword ptr [edx]; If so, eax = edx


       add edx, 4          ; Move *edx to next int
```

```
      dec ecx                   ; Decrement counter

      jnz MainLoop ; Loop if there's more


Finished:

      ret          ; Return with lowest in eax

FindSmallest endp

end
```

Additional Steps for x64

Visual Studio 2010, 2012, and 2013 Professional come with all the tools needed to quickly add native assembly code files to your C++ projects. These steps provide one method of adding native assembly code to a C++ project. The screenshots are taken from Visual Studio 2010, but 2012 is almost identical in these aspects. Steps one through six for creating this project are identical to those described for 32-bit applications. After you have completed these steps, the project must be changed to compile for the x64 architecture.

7. Open the **Build** menu and select **Configuration Manager**.

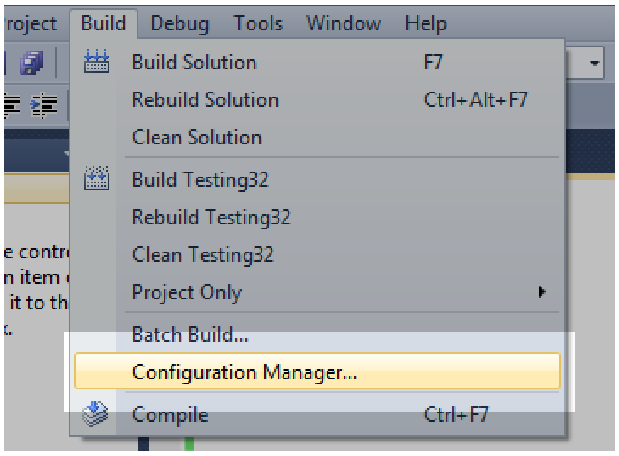


Figure 4

8. In the configuration manager window, select **<New...>** from the Platform column.

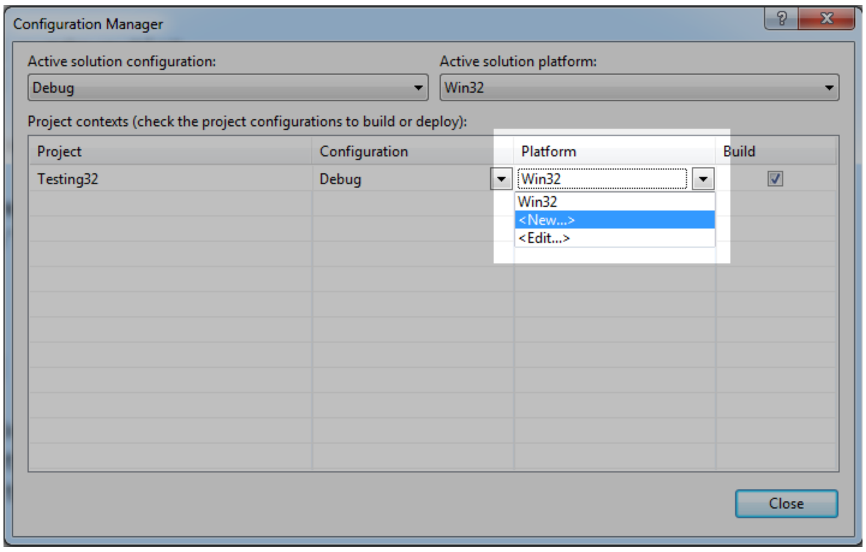

Configuration Manager
Active solution configuration:
Debug
Active solution platform:
Win32
Project contexts (check the project configurations to build or deploy):
Project
Configuration
Platform
Build
Testing32
Debug
Win32
Win32
<New...>
<Edit...>
Close

Figure 5

9. In the **New Project Platform** window, select **x64** from the **New Platform** drop-down list. Ensure that **Copy Settings from** is set to **Win32**, and that the **Create new solution platforms** box is selected. This will make Visual Studio do almost all the work in changing our paths from 32-bit libraries to 64-bit. The compiler will change from ML.exe (the 32-bit version of MASM) to ML64.exe (the 64-bit version) only if the **create new solutions platforms** is selected, and only if the Windows 7 SDK is installed.

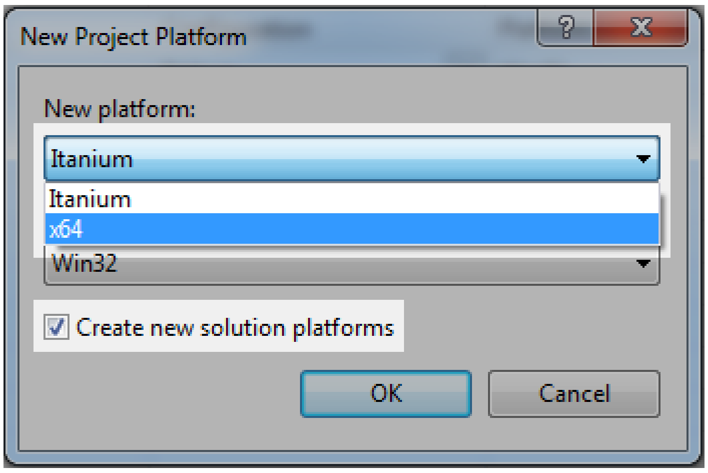


Figure 6

If you are using Visual Studio Professional edition, you should now be able to compile the example at the end of this section. If you are using Visual C++ Express edition, then there is one more thing to do.

The Windows 7 SDK does not set up the library directories properly for x64 compilation. If you try to run a program with a native assembly file, then you will get an error saying the compiler needs **kernel32.lib**, the main Windows kernel library.

```
LINK : fatal error LNK1104: cannot open file 'kernel32.lib'
```

You can easily add the library by telling your project to search for the x64 libraries in the directory that the Windows SDK was installed to.

10. Right-click on your solution and select **Properties**.

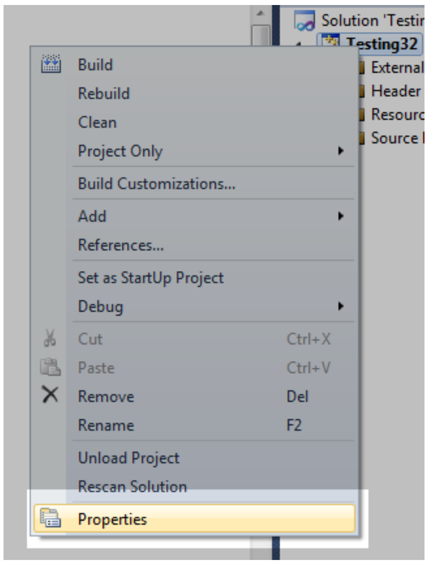
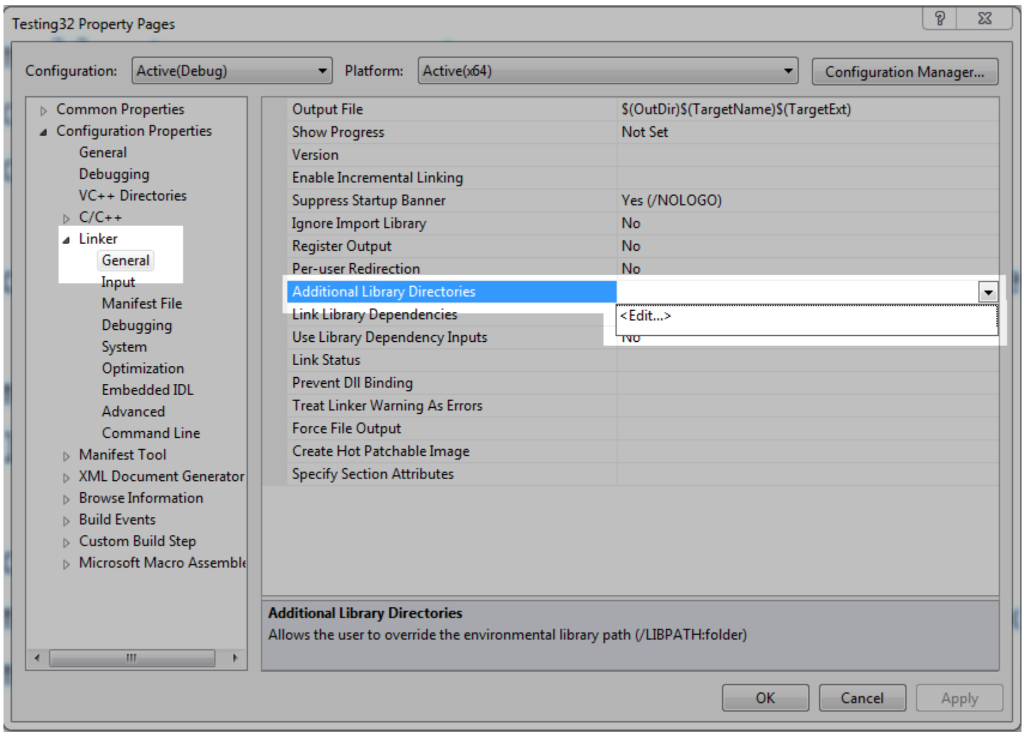


Figure 7

11. Select **Linker**, and then select **General**. Click **Additional Library Directories** and choose **<Edit…>**.



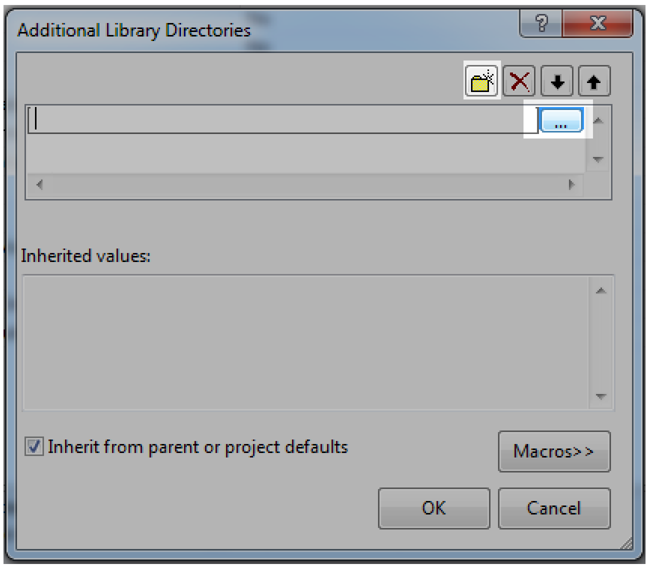
Configuration: Active(Debug) Platform: Active(x64)

Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	<Edit...>
Link Library Dependencies	
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	
Force File Output	
Create Hot Patchable Image	
Specify Section Attributes	

Additional Library Directories
Allows the user to override the environmental library path (/LIBPATH:folder)

Figure 8

12. Click the **New Folder** icon in the top-right corner of the window. This will add a new line in the box below it. To the right of the box is a button with an ellipsis in it. Click the ellipsis box and you will be presented with a standard folder browser used to locate the directory with **kernel32.lib**.



Additional Library Directories
...
Inherited values:
Inherit from parent or project defaults
Macros>>
OK
Cancel

Figure 9

The **C:\Program Files\Microsoft SDKs\Windows\v7.1\Lib\x64** directory shown in the following figure is the directory where Windows 7 SDK installs the **kernel32.lib** library by default. Once this directory is opened, click **Select Folder**. In the **Additional Library Directories** window, click **OK**. This will take you back to the **Project Properties** page. Click **Apply** and close the properties window.

You should now be able to compile x64 and successfully link to a native assembly file.

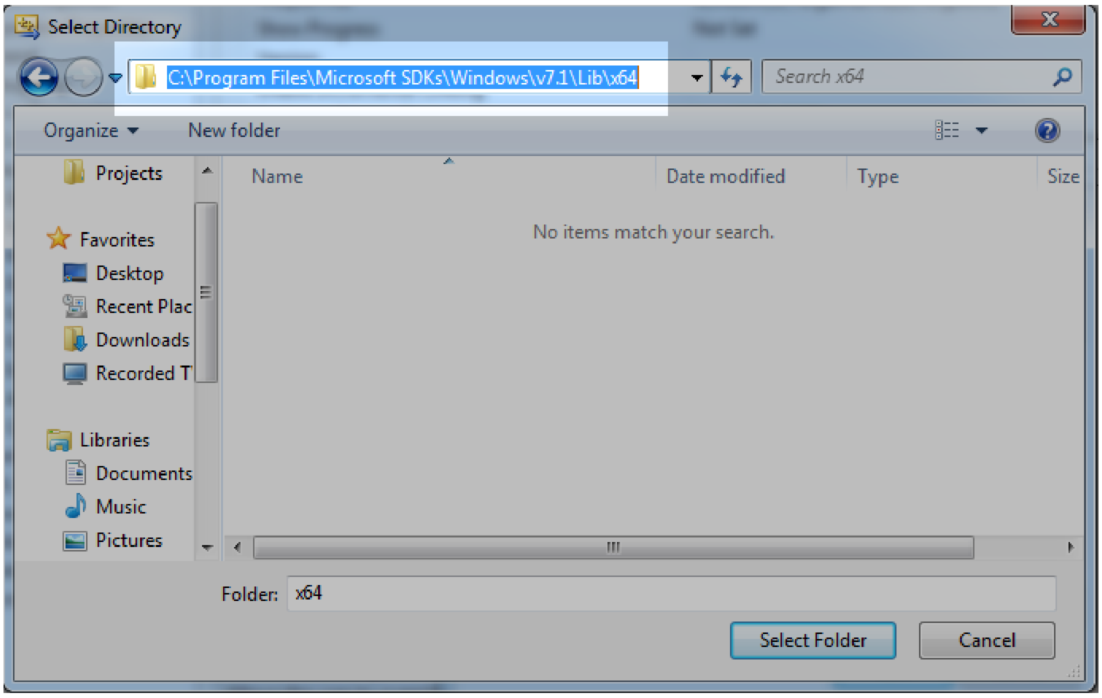


Figure 10

Note: There is a kernel32.lib for 32-bit applications and a kernel32.lib for x64. They are named exactly the same but they are not the same libraries. Make sure the kernel32.lib file you are trying to link to is in an x64 directory, not an x86 directory.

64-bit Code Example

Add the following two code listings to the C++ source and assembly files we added to the project.

```
// Listing: Main.cpp

#include <iostream>


using namespace std;


// External procedure defined in asmfunctions.asm

extern "C" int FindSmallest(int* i, int count);


int main() {

	int arr[] = { 4, 2, 6, 4, 5, 1, 8, 9, 5, -5 };


	cout<<"Smallest is "<<FindSmallest(arr, 10)<<endl;


	cin.get();


	return 0;

}
```

```
; Listing: asmfunctions.asm

.code

; int FindSmallest(int* arr, int count)

FindSmallest proc          ; Start of the procedure

	mov eax, 7fffffffh  ; Assume the smallest is maximum int


	cmp edx, 0          ; Is the count <= 0?
```

```
      jle Finished        ; If yes get out of here


MainLoop:

      cmp dword ptr [rcx], eax   ; Compare an int with our smallest so far

      cmovl eax, dword ptr [rcx]; If the new int is smaller update our smallest

      add rcx, 4                 ; Move RCX to point to the next int


      dec edx                    ; Decrement the counter

      jnz MainLoop        ; Loop if there's more


Finished:

      ret          ; Return whatever is in EAX

FindSmallest endp   ; End of the procedure

end   ; Required at the end of x64 ASM files, closes the segments
```

Chapter 2 Fundamentals

Now that we have some methods for coding assembly, we can begin to examine the language itself. Assembly code is written into a plain text document that is assembled by MASM and linked to our program at compile time or stored in a library for later use. The assembling and linking is mostly done automatically in the background by Visual Studio.

Note: Assembly language files are not said to be compiled, but are said to be assembled. The program that assembles assembly code files is called an assembler, not a compiler (MASM in our case).

Blank lines and other white space is completely ignored in the assembly code file, except within a string. As in all programming, intelligent use of white space can make code much more readable.

MASM is not case sensitive. All register names, instruction mnemonics, directives, and other keywords need not match any particular case. In this document, they will invariably be written as lowercase in any code examples. Instructions and registers will be written in upper case when referred to by name (this convention has been adopted from the AMD programmer's manuals, and it makes register names easier to read).

Note: If you would like MASM to treat variable names and labels in a case sensitive way, you can include the following option at the top of your assembly code file: "option casemap: none."

Statements in assembly are called instructions; they are usually very simple and do some tiny, almost insignificant tasks. They map directly to an actual operation the CPU knows how to perform. The CPU uses only machine code. The instructions you type when programming assembly are memory aids so that you don't need to remember machine code. For this reason, the words used for instructions (**`MOV`**, **`ADD`**, **`XOR`**, etc.) are often called mnemonics.

Assembly code consists of a list of these instructions one after the other, each on a new line. There are no compound instructions. In this way, assembly is very different from high-level languages where programmers are free to create complex conditional statements or mathematical expressions from simpler forms and parentheses. MASM is actually a high-level assembler, and complex statements can be formed by using its macro facilities, but that is not covered in detail in this book. In addition, MASM often allows mathematical expressions in place of constants, so long as the expressions evaluate to a constant (for instance, **`MOV AX, 5`** is the same as **`MOV AX, 2+3`**).

Skeleton of an x64 Assembly File

The most basic native x64 assembly file of all would consist of just **`End`** written at the top of the file. This sample file is slightly more useful; it contains a .data and a .code segment, although no segments are actually necessary.

```
.data

            ; Define variables here
```

```
.code

             ; Define procedures here

End
```

Skeleton of an x32 Assembly File

The skeleton of a basic 32-bit assembly file is slightly more verbose than the 64-bit version.

```


.xmm

.model flat, c


.data


.code

Function1 proc export

      push ebp

      mov ebp, esp


      ; Place your code here


      pop ebp

      ret

Function1 endp

End
```

The very first line describes the CPU the program is meant to run on. I have used .xmm, which means that the program requires a CPU with SSE instruction sets. This instruction set will be discussed in detail in Chapter 8). Almost all CPUs used nowadays have these instruction sets to some degree.

Note: Some other possible CPU values are .MMX, .586, .286. It is best to use the best possible CPU you wish your program to run on, since selecting an old CPU will enable backwards compatibility but at the expense of modern, powerful instruction sets.

I have included a procedure called **Function1** in this skeleton. Sometimes the push, mov, and pop lines are not required, but I have included them here as a reminder that in 32-bit assembly, parameters are always passed on the stack and accessing them is very different in 32-bit assembly compared to 64-bit.

Comments

Anything to the right of a semicolon (;) is a comment. Comments can be placed on a line by themselves or they can be placed after an instruction.

```
; This is a comment on a line by itself

mov eax, 24  ; This comment is after an instruction
```

Note: It is a good idea to comment almost every line of assembly. Debugging uncommented assembly is extremely time consuming, even more so than uncommented high-level language code.

You can also use multiline or block comments with the comment directive shown in the sample code. The comment directive is followed by a single character; this character is selected by the programmer. MASM will treat all text until the next occurrence of this same character as a comment. Often the carat (^) or the tilde (**~**) characters are used, as they are uncommon in regular assembly code. Any character is fine as long as it does not appear within the text of the comment.

```
CalculateDistance proc

      comment ~

      movapd xmm0, xmmword ptr [rcx]

      subpd xmm0, xmmword ptr [rdx]

      mulpd xmm0, xmm0

      haddpd xmm0, xmm0

      ~

      sqrtpd xmm0, xmm0


      ret

CalculateDistance endp
```

In the sample code, the comment directive appears with the tilde. This would comment out the four lines of code that are surrounded by the tilde. Only the final two lines would actually be assembled by MASM.

Destination and Source Operands

Throughout this reference, parameters to instructions will be called parameters, operands, or destination and source.

Destination: This is almost always the first operand; it is the operand to which the answer is written. In most two-operand instructions, the destination also acts as a source operand.

Source: This is almost always the second operand. The source of a computation can be either of the two operands, but in this book I have used the term source to exclusively mean the second parameter.

For instance, consider the following.

```
add rbx, rcx
```

`RBX` is the destination; it is the place that the answer is to be stored. `RCX` is the source; it is the value being added to the destination.

Segments

Assembly programs consist of a number of sections called segments; each segment is usually for a particular purpose. The code segment holds the instructions to be executed, which is the actual code for the CPU to run. The data segment holds the program's global data, variables, structure, and other data type definitions. Each segment resides in a different page in RAM when the program is executed.

In high-level languages, you can usually mix data and code together. Although this is possible in assembly, it is very messy and not recommended. Segments are usually defined by one of the following quick directives:

Table 1: Common Segment Directives

Directive	Segment	Characteristics
.code	Code Segment	Read, Execute
.data	Data Segment	Read, Write
.const	Constant Data Segment	Read
.data?	Uninitialized Data Segment	Read, Write

Note: .code, .data, and the other segment directives mentioned in the previous table are predefined segment types. If you require more flexibility with your segment's characteristics, then look up the segment directive for MASM from Microsoft.

The constant data segment holds data that is read only. The uninitialized data segment holds data that is initialized to 0 (even if the data is defined as having some other value, it is set to 0). The uninitialized data segment is useful when a programmer does not care what value data should have when the application first starts.

Note: Instead of using the uninitialized data segment, it is also common to simply use a regular .data segment and initialize the data elements with “?”.

The characteristics column in the sample table indicates what can be done with the data in the segment. For instance, the code segment is read only and executable, whereas the data segment can be read and written.

Segments can be named by placing the name after the segment directive.

```
.code MainCodeSegment
```

This is useful for defining sections of the same segment in different files, or mixing data and code together.

Note: Each segment becomes a part of the compiled .exe file. If you create a 5-MB array in your data segment your .exe will be 5 MB larger. The data defined in the data segment is not dynamic.

Labels

Labels are positions in the code segment to which the IP can jump using the **`JMP`** instructions.

`[LabelName]:`

Where **`[LabelName]`** is any valid variable name. To jump to a defined label you can use the **`JMP`**, **`Jcc`** (conditional jumps), or the **`CALL`** instruction.

```
SomeLabel:

        ; Some code

        jmp SomeLabel; Immediately moves the IP to SomeLabel
```

You can store a label in a register and jump to it indirectly. This is essentially using the register as a pointer to some spot in the code segment.

```
SomeLabel:

        mov rax, SomeLabel

        jmp rax      ; Moves the IP to the address specified in RAX, SomeLabel
```

Anonymous Labels

Sometimes it is not convenient to think of names for all the labels in a block of code. You can use the anonymous label syntax instead of naming labels. An anonymous label is specified by **`@@:`**. MASM will give it a unique name.

You can jump forward to an address higher than the current instruction pointer (IP) by using **`@F`** as the parameter to a **`JMP`** instruction. You can jump backwards to an address lower than the current IP by using **`@B`** as the parameter to a **`JMP`** instruction.

```
@@:       ; An anonymous label

jmp @F    ; Instruction to jump forwards to the nearest anonymous label

jmp @b    ; Instruction to jump backwards to the nearest anonymous label
```

Anonymous labels tend to become confusing and difficult to maintain, unless there is only a small number of them. It is usually better to define label names yourself.

Data Types

Most of the familiar fundamental data items from any high-level language are also inherent to assembly, but they all have different names.

The following table lists the data types referred to by assembly and C++. The sizes of the data types are extremely important in assembly because pointer arithmetic is not automatic. If you add 1 to an integer (dword) pointer it will move to the next byte, not the next integer as in C++.

Some of the data types do not have standardized names; for example, the XMM word and the REAL10 are just groups of 128 bits and 80 bits. They are referred to as XMM words or REAL10 in this book, despite that not being their name but a description of their size.

Some of the data types in the ASM column have a short version in parentheses. When defining data in the data segment, you can use either the long name or the short one. The short names are abbreviations. For example, "define byte" becomes “db”.

Note: Throughout this book, I will always refer to double words as dwords, and double-precision floats as doubles.

Table 2: Fundamental Data Types

Type	ASM	C++	Bits	Bytes
Byte	byte (db)	char	8	1
Signed byte	sbyte	char	8	1
Word	word (dw)	unsigned short	16	2
Signed word	sword	short	16	2
Double word	dword (dd)	unsigned int	32	4
Signed double word	sdword	int	32	4
Quad word	qword (dq)	unsigned long long	64	8
Signed quad word	sqword	long long	64	8
XMM word (dqword)	xmmword		128	16
YMM word	ymmword		128	16
Single	real4	float	32	4

Type	ASM	C++	Bits	Bytes
Double	real8	double	64	8
Ten byte float	real10 (tbyte, dt)		80	10

Data is usually drawn with the most significant bit to the left and the least significant to the right. There is no real direction in memory, but this book will refer to data in this manner. All data types are a collection of bytes, and all data types except the REAL10 occupy a number of bytes that is some power of two.

There is no difference between data types of the same size to the CPU. A REAL4 is exactly the same as a dword; both are simply 4-byte chunks of RAM. The CPU can treat a 4-byte block of code as a REAL4, and then treat the same block as a dword in the very next instruction. It is the instructions that define whether the CPU is to use a particular chunk of RAM as a dword or a REAL4. The variable types are not defined for the CPU; they are defined for the programmer. It is best to define data correctly in your data segment because Visual Studio's debugging windows display data as signed or unsigned and integer or floating point based on their declarations.

There are several data types which have no native equivalent in C++. The XMM and YMM word types are for Single Instruction Multiple Data (SIMD), and the rather oddball REAL10 is from the old x87 floating point unit.

Note: This book will not cover the x87 floating point unit's instructions, but it is worth noting that this unit, although legacy, is actually capable of performing tasks the modern SSE instructions cannot. The REAL10 type adds a large degree of precision to floating point calculations by using an additional 2 bytes of precision above a C++ double.

Little and Big Endian

x86 and x64 processors use little endian (as opposed to big endian) byte order to represent data. So the byte at the lowest address of a multiple byte data type (words, dwords, etc.) is the least significant, and the byte at the highest address is the most significant. Imagine RAM as a single long array of bytes from left to right.

If there is a word or 2-byte integer at some address (let us use 0x00f08480, although in reality a quad word would be used to store this pointer so it would be twice as long) with the values 153 in the upper byte and 34 in the lower, then the 34 would be at the exact address of the word (0x00f08480). The upper byte would have 153 and would be at the next byte address (0x00f08481), one byte higher. The number the word is storing in this example is the combination of these bytes as a base 256 number (34+153×256).

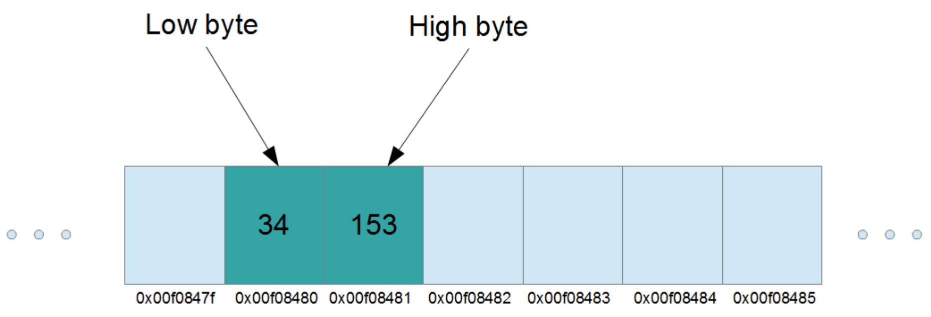


Figure 11

This word would actually be holding the integer 39,202. It can be thought of as a number in base 256 where the 34 is the first digit and the 153 is the second, or 39202 = 34+153×(256^1).

Two’s and One’s Complement

In addition to being little endian, x86 and x64 processors use two’s complement to represent signed, negative numbers. In this system, the most significant bit (usually drawn as the leftmost) is the sign bit. When this bit is 0, the number being represented is positive and when this bit is 1, the number is negative. In addition, when a number is negative, the number it represents is the same as flipping all the bits and adding 1 to this result. So for example, the bit pattern 10110101 in a signed byte is negative since the left bit is 1. To find the actual value of the number, flip all the bits and add 1.

Flipping each bit of 10110101 gives you 01001010.

01001010 + 1 = 01001011

01001011 in binary is the number 75 in decimal.

So the bit pattern 10110101 in a signed byte on a system that represents signed numbers with two's complement is representing the value -75.

Note: Flipping the bits is called the one's complement, bitwise complement, or the complement. Flipping the bits and adding one is called the two's complement or the negative. Computers use two's complement, as it enables the same circuitry used for addition to be used for subtraction. Using two's complement means there is a single representation of 0 instead of -0 and +0.

Chapter 3 Memory Spaces

Computers are made of many components, some of which have memory or spaces to store information. The speed of these various memory spaces and the amount of memory each is capable of holding are quite different. Generally, the closer to the CPU the memory space, the faster the data can be read and written.

There are countless possible memory spaces inside a computer: the graphics card, USB sticks, and even printers and other external devices all add memory spaces to the system. Usually the memory of a peripheral device is accessed by the drivers that come with the devices. The following table lists just a few standard memory spaces.

Table 3: Memory Spaces

Memory Space	Speed	Capacity
Human input	Unknown	Unknown
Hard drives and external storage	Extremely slow	Massive, > 100 gigabytes
RAM	Fast	Large, gigabytes
CPU caches	Very fast	Small, megabytes
CPU registers	Fastest	Tiny, < 1 kilobyte

The two most important memory spaces to an assembly program are the RAM and the CPU memories. RAM is the system memory; it is large and quite fast. In the 32-bit days, RAM was segmented, but nowadays we use a flat memory model where the entire system RAM is one massive array of bytes. RAM is fairly close to the CPU, as there are special buses designed to traffic data to and from the RAM hundreds of times quicker than a hard drive.

There are small areas of memory on the CPU. These include the caches, which store copies of data read from external RAM so that it can be quickly accessed if required. There are usually different levels of cache on a modern CPU, perhaps up to 3. Level 1 (abbreviated to L1 cache) is the smallest but quickest, and level 3 (abbreviated to L3 cache) is the slowest cache but may be megabytes in size. The operation of the caches is almost entirely automatic. The CPU handles its own caches based on the data coming into it and being written to RAM, but there are a few instructions that deal specifically with how data should or should not be cached.

It is important to be aware of the caches, even though in x86 programmers are not granted direct control over them. When some value from an address in RAM is already in the L1 cache, reading or writing to it is almost as fast as reading and writing to the registers. Generally, if data is read or written, the CPU will expect two things:

- The same data will probably be required again in the near future (temporal locality).
- The neighboring data will probably also be required (spatial locality).

As a result of these two expectations, the CPU will store both the values requested by an instruction from RAM and its cache. It will also fetch and store the neighboring values.

More important than the CPU caches are the registers. The CPU cannot perform calculations on data in RAM; data must be loaded to the CPU before it can be used. Once loaded from RAM, the data is stored in the CPU registers. These registers are the fastest memory in the entire computer. They are not just close to the CPU, they are the CPU. The registers are just a handful of variables that reside on the CPU, and they have some very strange characteristics.

Registers

The registers are variables residing on the CPU. The registers have no data type. Specifically, they are all data types, bytes, words, dwords, and qwords. They have no address because they do not reside in RAM. They cannot be accessed by pointers or dereferenced like data segment variables.

The present register set (x64) comes from earlier x86 CPUs. It is easiest to understand why you have these registers when you examine the older CPU register sets. This small trip through history is not just for general knowledge, as most of the registers from 1970s CPUs are still with us.

Note: There is no actual definition for what makes a CPU 64-bit, 32-bit, or 16-bit, but one of the main defining characteristics is the size of the general purpose registers. x64 CPUs have 16 general purpose registers and they are all 64 bits wide.

16-Bit Register Set

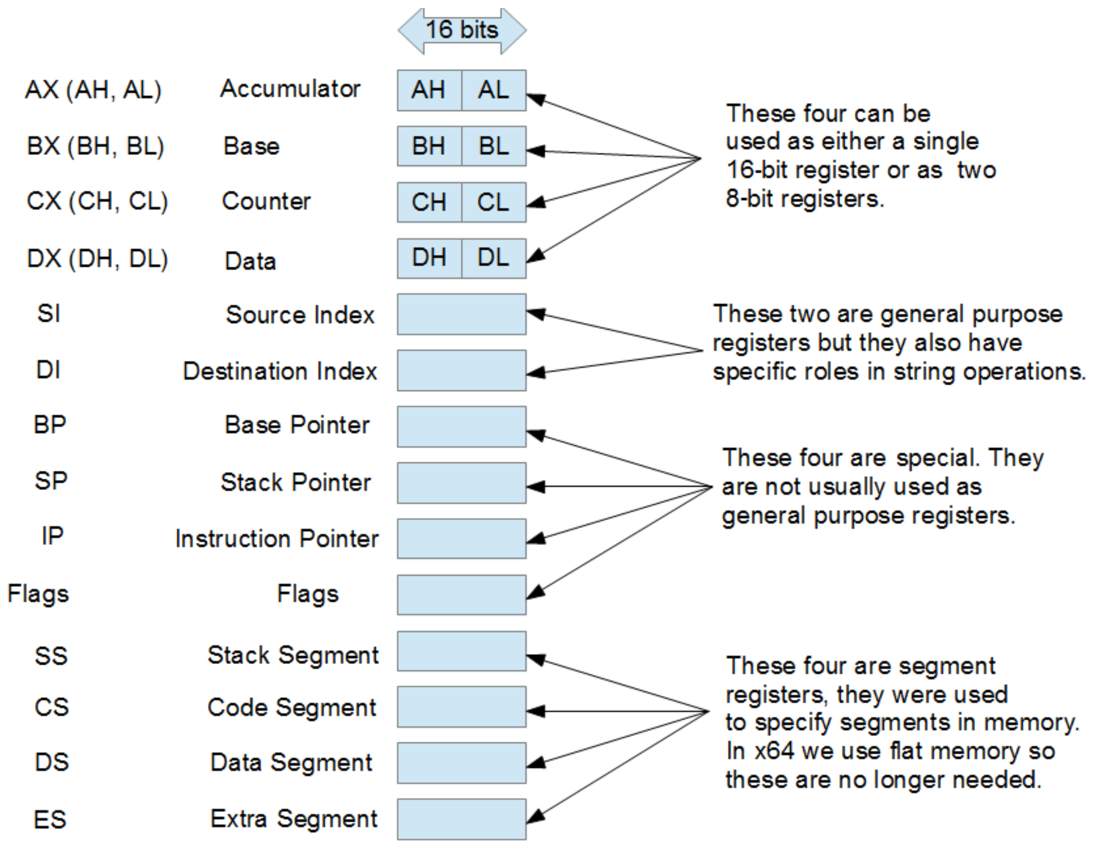

16 bits
AX (AH, AL)
Accumulator
AH
AL
BX (BH, BL)
Base
BH
BL
CX (CH, CL)
Counter
CH
CL
DX (DH, DL)
Data
DH
DL
These four can be used as either a single 16-bit register or as two 8-bit registers.
SI
Source Index
DI
Destination Index
These two are general purpose registers but they also have specific roles in string operations.
BP
Base Pointer
SP
Stack Pointer
IP
Instruction Pointer
Flags
Flags
These four are special. They are not usually used as general purpose registers.
SS
Stack Segment
CS
Code Segment
DS
Data Segment
ES
Extra Segment
These four are segment registers, they were used to specify segments in memory. In x64 we use flat memory so these are no longer needed.

Figure 12

Let us begin by examining the original 16-bit 8086 register set from the 1970s. Each of the original 8086 registers had a name indicating what the register was mainly used for. The first important thing to note is that AX, BX, CX, and DX can each be used as a single 16-bit register or as two 8-bit registers.

AX, BX, CX, and DX: The register AL (which means A Low) is the low byte of AX, and the register AH (which means A High) is the upper byte. The same is true for BX, CX, and DX; each 16-bit register has two 8-bit versions. This means that changing one of the low bytes (AL, BL, CL, or DL) will change the value in the word-sized version (AX, BX, CX, or DX). The same is true of changing the high bytes (AH, BH, CH, and DH). This also means that programmers can perform arithmetic on bytes or words. The four 16-bit registers can be used as eight 8-bit registers, four 16-bit registers, or any other combination.

SI and DI: These are the source and destination index registers. They are used for string instructions where SI points to the source of the instruction and DI points to the destination. They were originally only available in 16-bit versions, but there were no byte versions of these registers like there are for AX, BX, CX, and DX.

BP: This is the base pointer; it is used in conjunction with the SP to assist in maintaining a stack frame when calling procedures.

SP: This is the stack pointer; it points to the address of the first item that will be popped from the stack upon executing the **`POP`** instructions.

IP: This is the instruction pointer (called PC for Program Counter in some assembly languages); it points to the spot in RAM that is to be read for the next machine code bytes. The IP register is not a general purpose register, and IP cannot be referenced in instructions that allow the general purpose registers as parameters. Instead, the IP is manipulated implicitly by calling the jump instructions (**`JMP`**, **`JE`**, **`JL`**, etc.). Usually the IP simply counts up one instruction at a time. As the code is executed, instructions are fetched from RAM at the address the IP indicates, and they are fed into the CPU's arithmetic units and executed. Jumping instructions and procedure calls cause the IP to move to some other spot in RAM and continue reading code from the new address.

Flags: This is another special register; it cannot be referenced as a general purpose register. It holds information about various aspects of the state of the CPU. It is used to perform conditional statements, such as jumps and conditional moves. The flags register is a set of 16 bits that each tell something about the recent events that have occurred in the CPU. Many arithmetic and compare instructions set the bits in the flags register, and with subsequent conditional jumps and moves performs the instructions based on the status of the bits of this register. There are many more flag bits in the flags register, but the following table lists the important ones for general application programming.

Table 4: Flags Register

Flag Name	Bit	Abbrev.	Description
Carry	0	CF	Last arithmetic instruction resulted in carry or borrow.
Parity	2	PF	1 if lowest byte of last operation has even 1 count.
Auxiliary Carry	4	AF	Carry for BCD (not used any more).
Zero	6	ZF	Last result equaled zero.
Sign	7	SF	Sign of last operation, 1 for – and 0 for +.

Flag Name	Bit	Abbrev.	Description
Direction	10	DF	Direction for string operations to proceed.
Overflow	11	OF	Carry flag for signed operations.

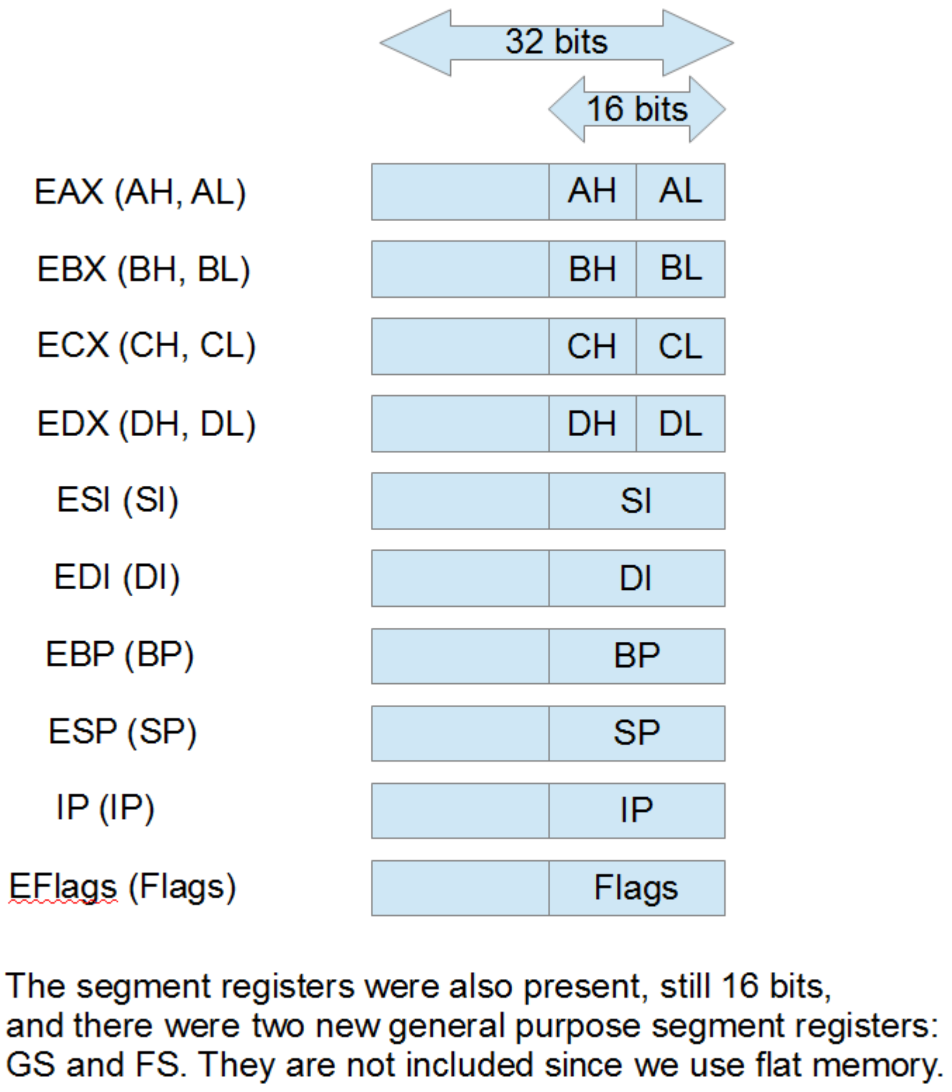
The individual flag bits of the flags register are not only used for what they were originally named. The names of the flags also reflect the most general use for each. For instance, CF is used to indicate whether the last addition or subtraction resulted in a final carry or borrow, but it is also set by the rotating instructions.

The parity flag was originally used in error checking, but it is now almost completely useless. It is set based on the count of bits set to 1 in the lowest byte of the last operation's result. If there is an even number of 1 bits set by the last result, the parity flag will be set to 1. If not, it will be cleared to 0. The auxiliary carry flag was used in Binary Coded Decimal (BCD) operations, but most of the BCD instructions are no longer available in x64.

The final four registers in the 8086 list (SS, CS, DS, and ES) are the segment pointers. They were used to point to segments in RAM. A 16-bit pointer can point to at most 64 kilobytes of different RAM addresses. Some systems at the time had more than 64 kilobytes of RAM. In order to access more than this 64-KB limit, RAM was segmented and the segment pointers specified a segment of the total installed RAM, while another pointer register held a 16-bit offset into the segment. In this way, a segment pointer in conjunction with an offset pointer could be thought of as a single 32-bit pointer. This is a simplification, but we no longer use segmented memory.

32-Bit Register Set

When 32-bit CPUs came about, backwards compatibility was a driving force in the register set. All previous registers were kept but were also extended to allow for 32-bit operations.



32 bits
16 bits
EAX (AH, AL)
AH
AL
EBX (BH, BL)
BH
BL
ECX (CH, CL)
CH
CL
EDX (DH, DL)
DH
DL
ESI (SI)
SI
EDI (DI)
DI
EBP (BP)
BP
ESP (SP)
SP
IP (IP)
IP
EFlags (Flags)
Flags
The segment registers were also present, still 16 bits, and there were two new general purpose segment registers: GS and FS. They are not included since we use flat memory.

Figure 13

The original registers can all still be referenced as the low 16 bits of the new 32-bit versions. For example, AX is the lowest word of EAX, and AL is still the lowest byte of AX, while AH is the upper byte of AX. The same is true for EBX, ECX, and EDX. As a result of this expansion to the register set, the 386 and 486 CPUs could perform arithmetic on bytes, words, and dwords.

The SI, DI, BP, and SP registers also added a 32-bit version and the original 16-bit registers were the low word of this. There was no byte form of these registers at that point.

The segment registers were also present and another two were added (GS and FS). Again, the segment registers are no longer as useful as they were, since modern Windows systems use a flat memory model.

Note: It is perfectly acceptable to use the different parts of a single register as two different operands to an instruction. For instance, "mov al, ah" moves the data from AH to AL. This is possible because the CPU has internal temporary registers to which it copies the values prior to performing arithmetic.

64-bit Register Set

Finally, we arrive at our present register set. This was a massive change, but once again, almost all backwards compatibility was maintained. In addition to increasing all general purpose registers to 64 bits wide by adding another 32 bits to the left of the 32-bit versions (EAX, EBX, etc.), eight new general purpose registers were added (R8 to R15). BP, SP, DI, and SI could also now have their lowest bytes referenced, as well as the lowest word or lowest dword.

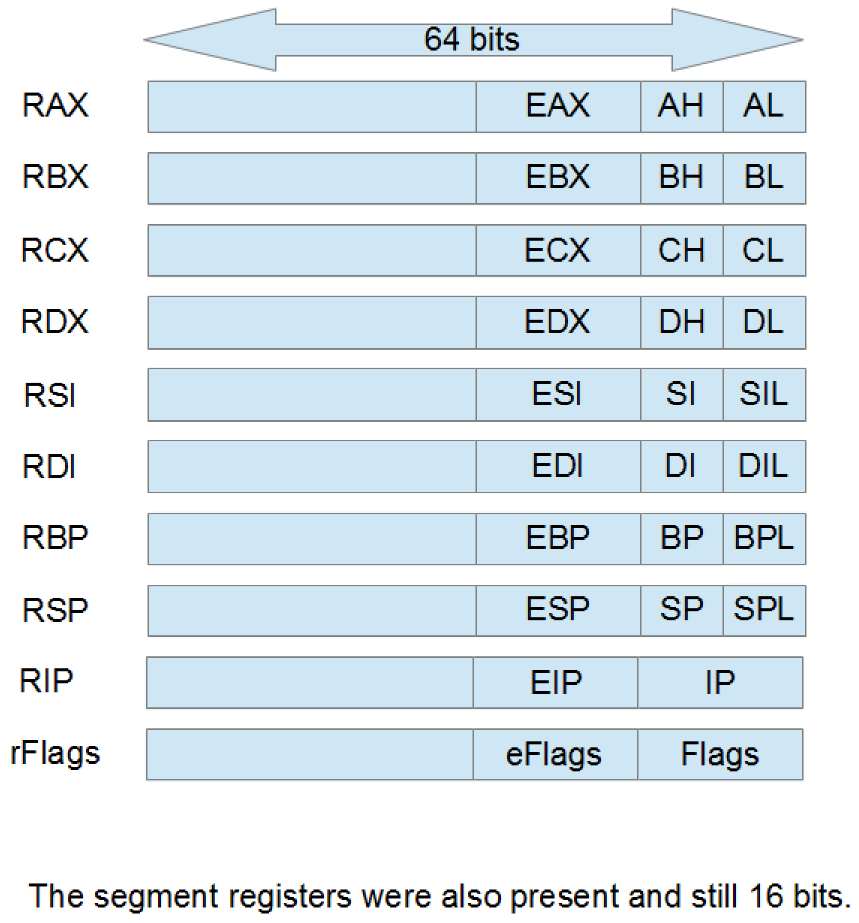


Figure 14

The general purpose registers AX, BX, CX, and DX still have high bytes (AH, BH, CH, and DH), but none of the other registers have their second byte addressable (there is no RDH, a high byte version of RDI). The high bytes of RAX, RBX, RCX, or RDX cannot be used with the low bytes of the other registers in a single instruction. For example, **`mov al, r8b`** is legal, but **`mov ah, r8b`** is not.

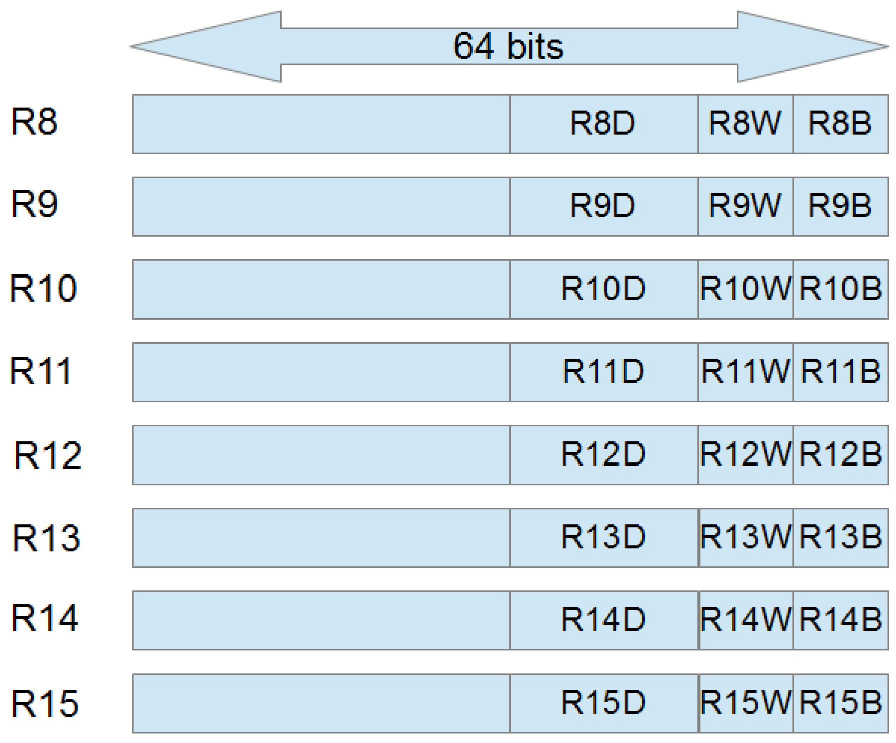


Figure 15

These are the new 64-bit general purpose registers R8 to R15. They can be used for anything the original RAX, RBX, RCX, or RDX registers can be used for. It is not clear in the diagram, but the lowest 32 bits of the new registers are addressable as R8D. The lowest 16 bits of R8 are called R8W and the lowest byte is called R8B. Although the image seems to depict R8D adjacent to R8W and R8B, R8W is actually the low 16 bits, exactly the same as RAX, EAX, AX, and AL.

Chapter 4 Addressing Modes

The different types of parameters an instruction can take are called addressing modes. This term is not to be confused with addresses in memory. The addressing modes include methods for addressing memory as well as the registers. Addressing modes are defined both by the CPU and the assembler. They are methods by which a programmer can address operands.

Registers Addressing Mode

The registers addressing mode is fairly self-explanatory. Any of the x86 registers can be used.

```
mov eax, ebx      ; EAX and EBX are both registers

add rcx, rdx      ; RCX and RDX are 64-bit registers

sub al, bl        ; AL and BL are the low 8-bit registers of RAX and RBX
```

Immediate Addressing Mode

The immediate or literal addressing mode is where a literal number appears as a parameter to an instruction, such as **`mov eax, 128`** where **`128`** would be the literal or immediate value. MASM understands literal numbers in several different bases.

Table 5: Common Bases

Base	Name	Suffix	Digits	Example
2	Binary	b	0 and 1	1001b
8	Octal	o	0 to 7	77723o
10	Decimal	d or none	0 to 9	1893 or 235d
16	Hexadecimal	h	0 to F	783ffh or 0fch

Note: When describing numbers in hexadecimal, if they begin with a letter digit (leftmost digit is A, B, C, D, E, or F), then an additional zero must be placed before it; "ffh" must be "0ffh". This does not change the size of the operand.

In addition to using a number, you can also use mathematical expressions, so long as they evaluate to a constant. The mathematical expressions will not be evaluated by the CPU at run time, but MASM will translate them to their constant values prior to assembling.

```
mov rax, 29+23      ; This is fine, will become mov rax, 52

mov rcx, 32/(19-4); Evaluates to 2, so MASM will translate to mov rax, 2

mov rdx, rbx*82     ; rbx*82 is not constant, this statement will not work
```

Implied Addressing Mode

Many instructions manipulate a register or some part of memory pointed to by a register, even though the register or memory address does not appear as a parameter. For instance, the string instructions (**MOVSxx**, **SCASxx**, **LODSxx**, etc.) reference memory, RAX, RCX, RSI, and RDI even though they take no parameters. This usage is called the implied addressing mode; parameters are implied by the instructions themselves and do not appear in the code.

```
REP SCASB     ; Scan string at [RDI] for AL and scan the number of bytes in RCX

CPUID         ; CPUID takes EAX as input and outputs to EAX, EBX, ECX, and EDX
```

Memory Addressing Mode

There is a multitude of ways to reference memory in MASM. They all do essentially the same thing; they read or write data from some address in RAM. The most basic usage of the memory addressing mode is using a variable defined in the data segment by name.

```
.data

xyzVar db ?    ; Define some variable in the data segment



.code

SomeFunction proc

      mov al, xyzVar      ; Move *xyzVar, the value of xyzVar, into AL

      .

      . Code continues

      .
```

Note: Because a label defined in the data segment is actually a pointer, some people tend not to call them variables but rather pointers or labels. The usage of "xyzVar" in the sample code is actually something like "mov al, byte ptr [xyzVar]" where xyzVar is a literal address.

It is often necessary to tell MASM what size the memory operand is, so that it knows what machine code to generate. For instance, there are many **MOV** instructions: there is one that moves bytes, one for words, and another for dwords. The same **MOV** mnemonic is used for all of them, but they generate completely different machine code and the CPU does different things for each of them.

These prefixes can be placed to the left of the square braces. The size prefixes are as follows.

Table 6: Pointer Size Prefixes

Size in Bytes	Prefix
1	byte ptr
2	word ptr
4	dword ptr
8	qword ptr
10	real10 ptr
16	xmmword ptr
32	ymmword ptr

Note: Signed, unsigned, or float versus integer is irrelevant here. A signed word is two bytes long, just as an unsigned word is two bytes long. These prefixes are only telling MASM the amount of data in bytes; they do not need to specify with any more clarity. For instance, to move a double (64-bit float) you can use the qword ptr, since 8 bytes is a quad word and it does not matter that the data happens to be a real8. You can also use real8 to move this amount of data.

In addition to using simple variables defined in the data segment, you can use registers as pointers.

```
mov eax, dword ptr [rcx]; Move 4 bytes starting where RCX is pointing

mov bl, byte ptr [r8]     ; Move a byte from *R8 into BL

add dx, word ptr [rax]    ; Add the word at *RAX to the value in DX
```

You can also add two registers together in the square braces. This allows a single base register to point to the first element of an array and a second offset pointer to step through the array. You can also use a register and add or subtract some literal value from it.

Note: Values being added or subtracted from a register can be complex expressions so long as they evaluate to a constant. MASM will calculate the expression prior to assembling the file.

```
sub rbx, qword ptr [rcx+rax]     ; Perhaps the base is RCX and RAX is an offset

add dword ptr [r8+68], r9d; Here we have added a constant to r8

add dword ptr [r8-51], r9d; Here we have subtracted a constant from r8
```

Note: Whenever values are being subtracted or added to addresses, either by using literal numbers or by using registers, the amount being added or subtracted always represents a number of bytes. Assembly is not like C++ with its pointer arithmetic. All pointers in assembly increment and decrement a single byte at a time, whereas in C++ an integer pointer will increment and decrement 4 bytes at a time automatically.

The most flexible of all memory addressing modes is perhaps the SIB (Scale, Index, Base) memory addressing mode. This involves a base register pointing to the start of some array, an index register that is being used as an offset into the array, and a scale multiplier that can be 1, 2, 4, or 8, and is used to multiply the value the index holds to properly access elements of different sizes.

```
mov bx, byte ptr [rcx+rdx*2]     ; RCX is the base, RDX is an offset and we
                             ; are using words so the scale is 2
add qword ptr [rax+rcx*8], r12   ; RAX is the base, RCX is the index
                                   ; and we are referencing qwords so the
                                   ; scale is 8
```

This addressing mode is useful for stepping through arrays in a manner similar to C++. Set the base register to the first element of the array, and then increment the index register and set the scale to the data size of the elements of the array.

```
mov rax, qword ptr [rcx+rbx*8]; Traverse a qword array at *RCX with RBX
mov cx, word ptr [rax+r8*2]   ; Traverse a word array at *RAX with R8
```

Chapter 5 Data Segment

The data segment is the place in RAM that a program stores its global and static data. This data is defined at compile time. The data segment does not hold variables that are allocated at run time (the heap is used for this purpose) or variables that are local to subprocedures (the stack is used to hold these). Most of the information presented here is to be used in any segment. For instance, variables can be declared in the uninitialized data segment (.data) or the constant data segment (.constant).

Note: All variables declared in your data segment will become bytes in your actual .exe file. They are not generated when the program is run; they are read from the .exe file. Creating a data segment with 150-MB of variables will generate a 150-MB .exe file and will take a very long time to compile.

Scalar Data

Scalar data defined in the data segment is given a name, size, and optional initial value. To declare a variable in the data segment is to name an offset from the start of the data segment. All the variable names are actually pointers; they are a number referring to an offset into the data segment in RAM, so programmers do not have to remember the offsets as numbers.

Note: Variables in assembly are sometimes referred to as labels, but to avoid confusion with labels in the code segment, I will refer to data segment labels as variables.

To define a variable in the data segment, the general layout is as follows:

```
[VarName]   [Type]            [Initial Value]
```

Where **`[VarName]`** is any legal variable name and will be the name of the point in data that you wish to use to as a reference.

Note: The rules for variable names are the same as those for C++. They cannot begin with a digit, and they can contain letters, digits, and underscores. You can also use some additional symbols that are illegal in C++, such as @ and ?.

`[Type]` is the data type and can be any one of the data types or short versions in the ASM column of the *Fundamental Data Types* table.

The initial value can be either a literal value or it can be “?”. The question mark means the data is not given an initial value. In actuality, data will be given a value even if it is uninitialized. The point of the “?” is to declare that the programmer does not care what value the data is set to initially and presumably the program will set some other initial value prior to using the data.

Here are some examples of defining simple scalar data types in a data segment.

```
.data
myByte db 0              ; Defines a byte set to 0 called myByte
patientID dw ?            ; Defines a word, uninitialized called patientID
averageSpeed dt 0.0     ; Defines 10-byte real, reals must have a decimal
                        ; point if initialized
totalCost sdword 5000   ; Defines signed dword set to 5000, called totalCost
```

Note: The first variable is placed at the DS:0 (it is placed at the first byte of the data segment) and the second straight after that (there is no padding paced between variables). If the first variable was 1 byte then the second would be at DS:1. If the first variable was a word then the second would be at DS:2. The way consecutive variables are stored in RAM is called alignment and it is important for performance as some of the fastest data processing instructions require data to be aligned to 16 bytes.

Arrays

After scalar data types, the next most fundamental data type is probably the array. An array is a list of elements of the same data type in contiguous memory. In assembly, an array is just a block of memory and the first element of the array is given a name.

Arrays Declared with Commas

You can declare the elements of an array separated by commas.

```
MyWord dw 1, 2, 3, 4      ; Makes a 4 word array with 1, 2, 3, and 4 as elements
```

If you need to use more than one line, finish the line with a comma and continue on the next.

```
MyWord dw 1, 2, 3, 4,     ; Four words in an array
             5, 6, 7, 8   ; Another four words in the same array!
```

This is legal because you actually do not need the label at all. The **MyWord** name of the variable is completely optional.

Duplicate Syntax for Larger Arrays

You can create larger arrays in your data segment using the duplicate syntax, but remember that every byte in your data segment is a byte in your final file.

To create larger arrays you can declare an array of values with the following pattern (the duplicate syntax):

`[Name]      [type] [n] [dup (?)]`

Where **`[Name]`** is the array name, any legal variable name. **`[Type]`** is one of the data types from the *Fundamental Data Types* table, and **`[n]`** is the number of items in the array. **`DUP`** is short for duplicate, and the data that it duplicates is in the parentheses. To make 50 words all set to 25 in an array called **MyArray**, the array declaration using the duplicate syntax would be the following:

```
MyArray word 50 dup (25)
```

You can combine the simple comma separated array definition syntax with the duplicate syntax and produce arrays of repeating patterns.

```
MyArray      byte   50 dup (1, 6, 8)
```

This will define an array 150 bytes long (50×3) with the repeating pattern 1, 6, 8, 1, 6, 8, 1, 6, 8....

You can nest the duplicate directive to create multidimensional arrays. For example, to create a 10×25 dimensional byte array and to set all elements to A, you could use the following:

```
MyMArray     byte 10 dup (25 dup ('A'))
```

Note: RAM is linear. Whether the sample code actually defines a 10×25 or a 25×10 array must be decided by the programmer. To the CPU, it is just a block of linear RAM and there is no such thing as a multidimensional array.

For a three-dimensional array, you could use something like this:

```
My3dArray    byte 10 dup (25 dup (100 dup (0)))
```

This will create a 10×25×100 3-D array of bytes all set to 0. From the CPU's point of view, this 3-D array is exactly the same as the following:

```
My3dArray byte 25000 dup (0)
```

Getting Information about an Array

Once defined, MASM has some directives to retrieve information about the array:

`lengthof`: Returns the length of the array in elements.

`sizeof`: Returns the length of the array in bytes.

`type`: Returns the element size of the array in bytes.

For example, if you have an array called **myArray** and you want to move information about it into AX, you would do the following:

```
mov ax, lengthof myArray  ; Move length in elements of the array

mov ax, sizeof myArray                ; Move the size in bytes of the array

mov ax, type myArray                  ; Move the element size into AX
```

Note: These directives are translated to immediate values prior to assembling the file; therefore, "mov lengthof myArray, 200" is actually translated to "mov 16, 200". Moving a value into a literal constant means nothing (we cannot change the meaning of 16, even in assembly), so the line is illegal.

Defining Strings

In MASM, single and double quotes are exactly the same. They are used for both strings and single characters.

Note: A string in C and C++ is a byte array often with a null, 0, at the end. These types of strings are called zero delimited strings. Many C++ functions are designed to work with zero delimited strings.

To define a string of text characters, you can use this string syntax:

```
errMess db 'You do not have permission to do this thing, lol', 0
```

This is equivalent to defining a byte array with the values set to the ASCII numbers of the characters in the string. The Y at the start will be the first (least significant) byte of the array.

Note: The comma and zero at the end are the final null. This makes the string a null-terminated string as understood by cout and other C++ functions. Cout stops writing a string when it reaches the 0 character. In C++, the 0 is added automatically when we use the double quotes; in assembly, it must be explicit.

To define a string that does not fit on a single line, you can use the following:

```
myLongString db "This is a ",

       "string that does not ",

       "fit on a single lion!", 0
```

In each of the previous examples, the single quote could also have been used to the same effect:

```
myLongString db 'This is a ',
       'string that does not',
       'fit on a single lion!', 0
```

If you need to use either a single quote in a single quote array or a double quote in a double quote array, place two of them together:

```
myArr1 db "This ""IS"" Tom's array!", 0     ; This "IS" Tom's array!
myArr2 db 'That''s good, who''s Tom?', 0    ; That's good, who's Tom?
```

Typedef

You can declare your own names for data types with the type definition (**typedef**) directive.

```
integer         typedef     sword ; Defines "integer" to mean sword
MyInteger   integer         ?     ; Defines a new sword called MyInteger
```

You cannot use reserved words for your typedefs, so trying to make a signed dword type called "int" will not work, since "int" is the x86 instruction to call an interrupt.

Note: You can use typedef to define new names for user-defined types, fundamental types, structures, unions, and records.

Structures and Unions

To define a structure (analogous to a C++ struct), you can use the struc (or struct) directive.

```
ExampleStructure struct    ; Structure name followed by "struct" or "struc"
       X word 0
       Y word 0
       Z word 0
       ID byte 0
ExampleStructure ends  ; The name followed by "ends" closes the definition
```

This would create a structure with four variables; three are words set to 0 and called X, Y, and Z, and the final variable is a byte called ID, also set to 0 by default.

The previous example was the prototype. To create an instance of the previous structure prototype, you can use the following:

```
person1 ExampleStructure { }     ; Declares person1 with default values

person2 ExampleStructure { 10, 25, 8, ? }     ; Declares person2 with

                                       ; specific values

                                       ; and ID of ?, or 0 probably
```

Each field of the instance of the structure can be initialized with the value supplied in respective order in curly brackets. Use “?” to initialize to MASM's default value (0). You can initialize less than the amount of values the structure and the rest are automatically given. These are their default values as per the structure's prototype.

```
person2 ExampleStructure { 10 }  ; Declares person2 with 10 for x

                                       ; but the rest are as per the

                                       ; structure's prototype
```

With a prototype declaration, you can create an instance of this structure with some of the values initialized, and others with their defaults, by not including any value. Just place whitespace with a comma to indicate where the value would have been.

```
MyStructure struct

      x word 5

      y word 7

MyStructure ends


InstanceOfStruct MyStructure { 9, }     ; Change x to 9 but keep y

                                      ; as 5 as per prototype
```

To change the values of a previously instantiated structure from code, you can use a period in a similar manner to accessing structure elements in C++.

```
mov person1.X, 25    ; Moves 25 into person1's X

mov person2.ID, 90 ; Moves 90 into person2's ID
```

Note: When structures are passed to functions from C++, they are not passed by reference. They are copied to the registers and stack depending on the size of the structure. If a structure has two integers, then the whole instance of the structure will be copied to RCX (since two 32-bit dwords fit into the 64-bit RCX). This is awkward because you cannot

reference the separate elements of the structure when they are in a register. For instance, there is no way to reference the top dword of RCX. For this reason, it may be easier to pass structures from C++ as pointers.

You can load the effective address of a previously instantiated structure with the LEA instruction (load effective address). To use a register (RCX in this example) as a pointer to an instance of a structure, you must tell MASM the address, type of structure being pointed to, and the field.

```
lea rcx, person1    ; Loads the address of person1 into RCX

mov [rcx].ExampleStructure.X, 200      ; Moves 200 into person1.X using

                                       ; RCX as a pointer
```

The CPU does not check to make sure RCX is actually pointing to an ExampleStructure instance. RCX could be pointing to anything. **`[RCX].ExampleStructure.X`** simply means find what RCX is pointing to and add the amount that X was offset in the ExampleStructure prototype to this address. In other words, **`[RCX].ExampleStructure.X`** translates to **`RCX+0`**, since X was at byte number 0 in the prototype of ExampleStructure. **`[RCX].ExampleStructure.Y`** translates to **`RCX+2`**, since Y was the second element after the two byte word X.

To pass an instance of a structure as a parameter to a function, it is usual to pass its address and manipulate it as per the previous point. This is passing by reference, and the initial object will be changed, but it is much faster than copying the data of the structure to the registers and stack in the manner of C++.

```
; This is the function that is initially called

Function1 proc

       lea rcx, person2    ; Load *person2 into RCX to be passed to Fiddle

       call Fiddle  ; Call Fiddle with RCX param 1

       ret

Function1 endp



; Fiddle, RCX = *ExampleStructure

Fiddle proc

       mov [rcx].ExampleStructure.Y, 89 ; Change something

       ret

Fiddle endp
```

Structures of Structures

To define a structure that has a smaller substructure as its member variables, declare the smaller one first. Then place the instances of the substructure inside the declaration of the larger structure.

```
; This is the smaller sub-structure

Point struct

      X word 0

      y word 0

Point ends



; This is the larger structure that owns a Point as one of its parameters:

Square struct

      cnr1 Point { 7, 4 } ; This one uses 7 and 4

      cnr2 Point { }            ; Use default parameters!

Square ends
```

To declare an instance of a structure that contains substructures in the data segment, you can use nested curly braces.

```
MySquare Square { { 9, 8 }, { ?, 7 } }
```

Note: If you do not want to set any of the values of a struct, you can use {} to mean defaults for all values, even if the structure has substructures within it.

To set the value of a structure's substructure, append a period to specify which variable you wish to change.

```
mov MySquare.cnr1.Y, 5
```

You can use a register as a pointer and reference the nested structure's elements as follows:

```
mov word ptr [rcx].Square.cnr1.X, 10
```

Unions

A union is similar to a structure, except the actual memory used for each of the elements in the union is physically at the same place in RAM. Unions are a way to reference the same address in RAM as more than one data type.

```
MyUnion union

      w1 word 0

      d1 dword 0

MyUnion ends ; Note that it is ends, not endu!
```

Here, **MyUnion.w1** has exactly the same address as **MyUnion.w2**. The dword version is 4 bytes long and the word is only 2 bytes, but the least significant byte of both has the same address.

Records

Records are another complex data type of MASM. They are like structures, but they work on and are defined at the bit level. The syntax for definition is as follows:

`[name] RECORD [fldName:sz], [fldName:sz]...`

Where **`[name]`** is the name of the record, **`[fldName]`** is the name of a field, and **`[sz]`** is the size of the field in bits.

```
color RECORD blBit:1, hueNib:4
```

The sample code in the data segment is the prototype to a record called **color**. The record can then be accessed by the following:

```
mov cl, blBit
```

This would move 4 into CL, since blBit was defined as bit number 4 in the record. hueNib takes bits 0, 1, 2, and 3, and blBit comes after this.

You cannot use a record to access bits directly.

```
mov [rax].color.blBit, 1  ; Won't change the 4th bit from RAX to 1
```

A record is just a form of directive; it defines a set of constants to be used with bitwise operations. The constants are bit indices. You can use a record for rotating.

```
mov cl, blBit; Move the index of the record's blBit into cl

rol rax, cl  ; Use this to rotate the bits in RAX
```

You can define records in your data segment and initialize the values of the bit fields just as you can with a structure. This is the only time you can set each element of a record without using bitwise operations.

```
.data

color RECORD qlBit:3, blBit:1, hueNib:4; Defines a record



; Following defines a new byte with the bits set as specified

; by the record declaration:

; qlBit gets 0, blBit gets 1 and the hueNib gets 2

; So MyColor will actuall be a byte set to  00010010b

MyColor color { 0, 1, 2 } ; Declare a color record with initializers



.code _text

Function1 proc

       mov cl, MyColor     ; Moves 000:1:0010b, or 18 in decimal

       ret

Function1 endp
```

Note: The qlBit, blBit and hueBit from the previous record become constants of their bit indices: hueBit = 0, blBit = 4, qlBit = 5.

You can get the width in bits of a field in a record by using MASM's **WIDTH** directive.

```
mov ax, WIDTH color.hueNib
```

You can get a bit mask of the record's field by using MASM's **MASK** directive.

```
and al, mask myCol.blBit; AND contents of AL with bit mask of defined color

                        ; record
```

You can specify **NOT** prior to the **MASK** directive to flip the bit mask.

```
and al, NOT MASK myCol.blBit
```

Constants Using Equates To

You can define a numerical constant using the **=** symbol, and you can define numerical and text constants using the **equ** directive. This is short for “equates to.”

```
Somevar = 25           ; Somevar becomes a constant immediate value 25

name equu 237       ; "name" is the symbol for the constant

mov eax, name       ; Translates to “mov eax, 237”

moc ecx, SomeVar        ; Sets ECX to 25
```

You can also use the **EQU** directive to define text constants by surrounding the value with triangle braces.

```
quickMove equ <mov eax, 23>

quickMove        ; Translates to “mov eax, 23”
```

You can use the equates directive to define machine code by using a **db** (define byte) in the value.

```
NoOperation equ <db 90h>     ; 90h is machine code for the NOP instruction

NoOperation                   ; Translates to NOP or 90h
```

This usage of **db** in the code segment is fine because **db** does nothing more than place the exact byte values you specify at the position in the file. Using **db** in the code segment effectively enables us to program in pure machine code.

```
; This procedure returns 1 if ECX is odd

; otherwise it returns 0, it is programmed

; in pure machine code using db.

IsOdd proc

db     83h, 0E1h, 01h,           ; and ecx, 1
```

```
      8Bh, 0C1h,                ; mov eax, ecx

      0C3h                      ; ret

IsOdd endp
```

The point of using pure machine code is that sometimes an assembler may not understand some instructions that the CPU can understand. An older assembler may not understand the SSE instructions. By using **EQU** and **db** in the manner described previously, a programmer can define his or her own way of specifying SSE instructions, whether the assembler understands them naturally or not.

Macros

You can define macro functions using the macro directive.

```
[name] MACRO [p1], [p2]...

      ; Macro body

ENDM
```

Where **[name]** is the symbol associated with the macro, **MACRO** and **ENDM** are keywords, and **[p1]**, **[p2]**, and any other symbols are the parameters as they are referred to in the body of the macro.

```
Halve macro dest, input   ; dest and input are the parameters

      mov dest, input     ;; Refer to parameters in body

      shr dest, 1

endm                      ; endm with no macro name preceding


; And later in your code:

Halve ecx, 50; Moves 25 into ecx

Halve eax, ecx      ; Moves 12 into eax

Halve ecx, ecx      ; Moves 12 into ecx

Halve 25, ecx; Error, ecx/2 cannot be stored in 25!
```

The symbol name is swapped for the corresponding code each time MASM finds the macro name when assembling. This means that if there are labels in the macro code (if the code has jumps to other points within its code), MASM will write the labels again and again. Each time the macro is used, the labels will appear. Since MASM cannot allow duplicate labels and still know where to jump, you can define labels as local in the macro definition. Labels defined as local will actually be replaced by an automatically generated, unique label.

```
SomeMacro macro dest, input

local label1, label2

      test dest, 1

      jnz label1

      jz label2

label1:                    ;; Automatically renamed ??0000

      mov eax, 3

label2:                    ;; Automatically renamed ??0001

      mov ecx, 12         ;; Each label each time SomeMacro is

                           ;; called will increment the counter,

                           ;; next will be ??0002 the ??0003 etc.

Endm
```

Note: You may have noticed the ";;" comments in the body of the macros; these are macro comments. They are useful when generating listing files, since these comments will not appear every time a macro function is referenced in code, only once at the macro's definition. If you use the single ";" comments the same comments will appear over and over throughout the generated listing file.

In your macro definition you can specify default values for any parameters, allowing you to call the macro without specifying every parameter (place **`:=`** and then the default value after the parameter's name, **`somevariable:=<eax>`**). You can also indicate that particular parameters are required (place a colon followed by **`req, somevariable:req`**).

Note: When specifying the default values, the syntax is similar to the "equ" directive; instead of "eax" you must use "<eax>".

```
SomeMacro macro p1:=<eax>, p2:req, p3:=<49>

      ;; Macro body

Endm
```

The macro definition in the sample code would allow us to omit values for both first and third parameters. Only the second is required, and the others can be left to defaults.

```
; Specify all parameters:

SomeMacro ecx, 389, 12     ; p1 = ecx

                           ; p2 = 389

                           ; p3 = 12



; Just specify parameter 2:

SomeMacro , ebx,           ; p1 = eax from default

                           ; p2 = ebx

                           ; p3 = 49 from default
```

Chapter 6 C Calling Convention

A calling convention is a set of steps that must be undertaken by a caller (the code calling the procedure) and a callee (the procedure being called). High-level languages take care of all the calling convention intricacies, and one can simply pass parameters to and from functions without caring about how they are being passed. When programming in assembly, the callee needs to know where or how the caller has passed the function's parameters, and the caller needs to know how the callee will return the answer. At the assembly level, the calling convention is not restricted at all, and programmers are free to define their own. The C++ compilers that ship with Visual Studio use the C calling convention, so it is usually advantageous to adopt this when programming assembly routines, especially if the routines are called from C++ or if they themselves call procedures written in C++.

The Stack

The stack is a portion of memory that is used as a semiautomatic last-in-first-out data structure for passing parameters to functions. It allows function calls to be recursive, handles parameter passing, return addresses, and is used to save registers or other values temporarily. Values are added to the stack using the **PUSH** and **CALL** instructions, and they are removed from the stack using the **POP** and **RET** instructions in the opposite order they were pushed. The stack is used to save the address in the .code segment of the caller of the function, such that when the subroutine is finished, the return address can be popped from the stack (using the **RET** instruction) and control can resume from the caller's position in code.

The stack is pointed to by a special pointer, the **RSP** (stack pointer). The instructions **PUSH** and **POP** both **MOV** data to the point **RSP** points to and the decrement (**PUSH**) or increment (**POP**) the stack pointer, such that the next value to be pushed will be done at the next address in the stack segment.

In the past, passing parameters and saving the return addresses was exclusively the task of the stack, but in x64 some parameters are passed via the registers. It is common to avoid the **PUSH** and **POP** instructions in favor of incrementing and decrementing the stack pointer manually and using **MOV** instructions. Manually manipulating the stack is common in x64, since the **PUSH** and **POP** instructions do not allow operands of any size. It is often faster to set the position of the **RSP** using **ADD** and **SUB** and using **MOV** instead of repeatedly calling **PUSH**. The stack is simply another segment in RAM that has been marked as read/write. The only difference between the stack and any other segment in the program is that the stack pointer (**RSP**) happens to point to it.

Scratch versus Non-Scratch Registers

In the C calling convention used by Visual Studio, some of the registers are expected to maintain the same values across function calls. Functions should not change the value of these registers in their code without restoring the original values prior to returning. These registers are called non-scratch.

Table 7: Register's Scratch/Non-Scratch Status

Register	Scratch/Non-Scratch
RAX	Scratch

Register	Scratch/Non-Scratch
RBX	Non-Scratch
RCX	Scratch
RDX	Scratch
RSI	Non-Scratch
RDI	Non-Scratch
RBP	Non-Scratch
RSP	Non-Scratch
R8 to R11	Scratch
R12 to R15	Non-Scratch
XMM0 to XMM5	Scratch
XMM6 to XMM15	Non-Scratch
ST(0) to ST(7)	Scratch
MM0 to MM7	Scratch
YMM0 to YMM5	Scratch
YMM6 to YMM15	Non-Scratch

Some of the registers can be modified at will by a subprocedure or function, and the caller does not expect that the subprocedure will maintain any particular values. These registers are called scratch.

There is nothing wrong with using a non-scratch register in your code. The following example uses RBX and RSI to sum the values from 100 to 1 together (both RBX and RSI are non-scratch). The important thing to note is that the non-scratch registers are pushed to the stack at the start of the procedure and popped just prior to returning.

```
Sum100 proc

      push rbx     ; Save RBX

      push rsi     ; Save RSI

      xor rsi, rsi

      mov rbx, 100

MyLoop:

      add rsi, rbx

      dec rbx

      jnz MyLoop
```

```
        mov rax, rsi


        pop rsi     ; Restore RSI

        pop rbx     ; Restore RBX

        ret

Sum100 endp
```

The push instruction saves the value of the register to the stack, and the pop instruction pops it back into the register again. By the time the subprocedure returns, all of the non-scratch registers will have exactly the same values they had when the subprocedure was called.

It is often better to use scratch registers instead of pushing and popping non-scratch registers. Pushing and popping requires reading and writing to RAM, which is always slower than using the registers.

Passing Parameters

When we specify a procedure as using the C calling convention in x64 applications, Microsoft's C++ compiler uses fastcall, which means that some parameters are passed via the registers instead of using the stack. Only the first four parameters are passed via registers. Any additional parameters are passed via the stack.

Table 8: Integer and Float Parameters

Parameter Number	If integer	If float
1	RCX	XMM0
2	RDX	XMM1
3	R8	XMM2
4	R9	XMM3

Integer parameters are passed in RCX, RDX, R8, and R9 while floating point parameters use the first four SSE registers (XMM0 to XMM3). The appropriate size of the register is used such that if you are passing integers (32-bit values), then ECX, EDX, R8D, and R9D will be used. If you are passing bytes, then CL, DL, R8B, and R9B will be used. Likewise, if a floating point parameter is 32 bits (float in C++), it will occupy the lowest 32 bits of the appropriate SSE register, and if it is 64 bits (a C++ double), then it will occupy the lowest 64 bits of the SSE register.

Note: The first parameter is always passed in RCX or XMM0; the second is always passed in RDX or XMM2. If the first parameter is an integer and the second is a float, then the second will be passed in XMM1 and XMM0 will go unused. If the first parameter is a floating point value and the second is an integer, then the second will be passed in RDX and RCX will go unused.

As an example, consider the following C++ function prototype:

`int SomeProc(int a, int b, float c, int d);`

This procedure takes four parameters, which are floating point or integer values, so all of them are going to be passed via the registers (only the 5th and subsequent parameters require the stack).

The following is how the C++ compiler will pass the parameters, or how it will expect you to pass them if you are calling a C++ procedure from assembly:

- a will be passed in ECX
- b will be passed in EDX
- c will be passed in the lowest dword of XMM2
- d will be passed in R9D

Integer values are always returned in RAX and floating point values are returned in XMM0. Pointers or references are also always returned in RAX.

The following example takes two integer parameters from a caller and adds them together, returning the result in RAX:

```
; First parameter is passed in ECX, second is passed in EDX

; The prototype would be something like: int AddInts(int a, int b);

AddInts proc

      add ecx, edx ; Add the second parameter's value to the first

      mov eax, ecx ; Place this result into EAX for return

      ret          ; Caller will read EAX for the return value

AddInts endp
```

Shadow Space

In the past, all parameters were passed to procedures via the stack. In the C calling convention, the caller still has to allocate blank stack space as if parameters were being passed on the stack, even though the values are being passed in the registers. The space you create on the stack in place of passing parameters when calling a function or subprocedure is called shadow space. It is the space where the parameters would have been passed had they not been placed into registers instead.

The amount of shadow space is supposed to be no less than 32 bytes, regardless of the number of parameters being passed. Even if you are passing a single byte, you reserve 32 bytes on the stack.

Note: This wasteful use of the stack is possibly due to it being easier to program the C++ compiler. Many things on this level of programming have little to no clear documentation or explanation available. The exact reasons for the Microsoft C calling convention using shadow space the way it does are not clear.

To call a function with the following prototype, use the following:

`void Uppercase(char a);`

The C++ compiler would use something like the following:

```
sub rsp, 20h ; Make 32 bytes of shadow space

mov cl, 'a'         ; Move parameter in to cl

call Uppercase      ; Call the function

add rsp, 20h ; Deallocate the shadow space from the stack
```

To call a function with six parameters, use the following:

`void Sum(int a, int b, int c, int d, int e, int f);`

Some parameters must be passed on the stack; only the first four will be passed using the registers.

```
sub rsp 20h   ; Allocate 32 bytes of shadow space

mov ecx, a    ; Move the four register parameters into their registers

mov edx, b

mov r8d, c

mov r9d, d

push f ; Push the remaining parameters onto the stack

push e

call  Sum     ; Call the function



add rsp, 28h; Delete shadow space and the parameters we passed via the stack
```

Note: Parameters passed via the stack are not actually removed from memory when the subroutine returns. The stack pointer is simply incremented such that newly pushed parameters will overwrite the old values.

To call a function written in C++ from an external assembly file, both C++ and assembly must have an **`extern`** keyword to say the function is available externally.

```
// C++ File:

#include <iostream>
```

```
using namespace std;

extern "C" void SubProc();

extern "C" int SumIntegers(int a, int b, int c, int d, int e, int f)
{
	return a + b + c + d + e + f;
}

int main()
{
	SubProc();

	return 0;
}

; Assembly file in the same project
extern SumIntegers: proc

.code
SubProc proc
	push 60		; Push two params that don't
	push 50		; fit int regs. Opposite order!

	sub rsp, 20h ; Allocate shadow space

	mov ecx, 10	; Move the first four params
	mov edx, 20	; into their regs in any order
	mov r8d, 30
	mov r9d, 40

	call SumIntegers

	add rsp, 30h ; Deallocate shadow space
	             ; and space from params
	             ; this is 6x8=48 bytes.
	ret
SubProc endp
End
```

The stack is decreased as parameters are pushed onto it. Parameters are pushed from right to left (reverse order to that of a function's C++ prototype).

Bytes and dwords cannot be pushed onto the stack, as the **PUSH** instruction only takes a word or qword for its operand. For this reason, the stack pointer can be decremented to its final position (this is the number of operands multiplied by 8) in the instruction where shadow space is allocated. Parameters can then be moved into their appropriate positions in the stack segment with **MOV** instructions in place of the pushes.

The first parameter is moved into RCX, then the second into RDX, the third into R8, and the fourth into R9. The subsequent parameters are moved into RAM starting at RSP+20h, then RSP+28h, RSP+30h, and so on, leaving 8 bytes of space for each parameter on the stack whether they are qwords or bytes. Each additional parameter is RSP+xxh where xx is 8 multiplied by the parameter index.

Note: As an alternate to hexadecimal, it may be more natural to use octal. In octal, the fourth parameter is passed at RSP+40o, the fifth is RSP+50o, and the sixth is RSP+60o. This pattern continues until RSP+100o, which is the 8th parameter.

```
; Assembly file alternate version without PUSH
extern SumIntegers: proc
.code
SubProc proc
	sub rsp, 30h	; Sub enough for 6 parameters from RSP
	mov ecx, 10	; Move the first four params
	mov edx, 20	; into their regs in any order
```

```
      mov r8d, 30

      mov r9d, 40

      ; And we can use MOV to move dwords

      ; bytes or whatever we need to the stack

      ; as if we'd pushed them!

      mov dword ptr [rsp+20h], 50

      mov dword ptr [rsp+28h], 60

      call SumIntegers

      add rsp, 30h ; Deallocate shadow space

                   ; and space from params

                   ; this is 6x8=48 bytes.

      ret

SubProc endp

End
```

Chapter 7 Instruction Reference

The following instruction reference is intended to summarize some of the information in the Intel and AMD programmer's manuals, and provide an easy-to-reference but detailed description of the most common and useful instructions. Full details of all instructions can be found in the Intel and AMD manuals (see the Recommended Reading section for a link to these documents).

This reference covers only application programming instruction sets. System programming instructions (or privileged instructions) are not included, nor are instructions that are now obsolete and have been removed from x64 assembly. Instructions are not included even where they are still supported in compatibility mode (for instance the Binary Coded Decimal (BCD) instructions, etc.). Only the most common and useful instructions have been included, but there are many hundreds more.

CISC Instruction Sets

Modern x64 CPUs are CISC (Complex Instruction Set Computing), as opposed to RISC (Reduced Instruction Set Computing). This means there are a very large number of specialized instructions, which are almost useless for general purpose programming, but have been added to the instruction sets for particular purposes such as 3-D graphics algorithms, encryption, and others.

There is almost no consistent logic to the naming of the instructions because they have been added over several decades and belong to different instruction sets.

Many instructions require hardware support from the CPU, such as each of the SIMD instruction sets and the conditional moves. Please refer to the **CPUID** instruction for details on how to detect if hardware is capable of particular instructions.

Parameter Format

The following table lists the shorthand for instruction parameter types I have used in this reference:

Table 9: Shorthand Instruction Parameters

Shorthand	Meaning
reg	x86 register eax, ebx etc.
mmx	64-bit MMX register
xmm	128-bit SSE register
ymm	256-bit AVX register
mem	Memory operand, data segment variable
imm	Immediate value, literal or constant
st	Floating point unity register

It is very important to note that you can never use two memory operands as parameters for an instruction, despite what the parameter shorthand appears to state. For instance, one of the parameters to the **MOV** instruction can be a memory operand, but both of them cannot; one must be an immediate or a register. There is only one address generation unit per arithmetic logic unit. By the time the arithmetic logic unit has the instruction to execute, it can only generate at most one memory address.

Unless expressly stated otherwise, all parameters to an instruction must match in size. You cannot move a word into a dword nor a dword into a word. There is no notion of implicit casting in assembly. There are some instructions (for instance, the move and sign/zero extend instructions) that are designed to convert one data size to another and must necessarily take operands of differing sizes, but almost all other instructions adhere to this rule.

The possible sizes of the operands have been included in the shorthand for the instructions. Some instructions do not work for all sized operands. For example, the mnemonic and parameters for the conditional move instructions might look like this:

CMOVcc [reg16/32/64], [reg16/32/64/mem16/32/64]

This means the instructions take two operands, and each operand is in square braces, though in the code they are not surrounded by square braces unless they are a pointer. The first can be an x86 register of sizes 16 bits, 32 bits, or 64 bits, and the second can be another x86 register of the same size or a memory operand.

```
CMOVE ax, bx; This would be fine, CMOVcc [reg16], [reg16]

CMOVE al, bl; This will not work because AL and BL are 8 bit registers
```

Note: As mentioned previously, the high byte forms of the original x86 registers cannot be used with the low byte forms of the new x64 registers. Something like "MOV AH, R8B" will not compile, as it uses the high byte form AH along with a new byte form R8B. The high byte forms are included in x64 only for backwards compatibility. The CPU knows no machine code to achieve "MOV AH, R8B" in a single instruction.

Flags Register

Many of the x86 instructions alter the state of the flags register so that subsequent conditional jumps or moves can be used based on the results of the previous instructions. The flags registers abbreviations appear differently in Visual Studio compared to almost all other sources. The meaning of the flags and their abbreviations in this manual and Visual Studio's Register Window are as follows:

Table 10: Flags Register Abbreviations

Flag Name	Abbreviation	Visual Studio
Carry Flag	CF	CY
Parity Flag	PF	PE
Zero Flag	ZF	ZR
Sign Flag	SF	PL

Flag Name	Abbreviation	Visual Studio
Direction Flag	DF	UP
Overflow Flag	OF	OV

A flags field of **`carry, zero`** would mean that both the carry flag and the zero flag are altered by the instruction in some way. This means that all other flags are either not altered or undefined. It is not safe to trust that an instruction will not modify a flag when the flag is undefined. Where it is not obvious how an instruction would alter the flags, see the instruction's description for details. If more information is required on whether flags are modified or left undefined, see the programmer's manuals of your CPU manufacturer.

If an instruction does not affect the flags register (such as the **`MOV`** instruction), the flags field will appear as **`Flags: (None)`**. If the flags field to an instruction is **`(None)`**, then the instruction will not alter the flags register at all.

Almost all of the SIMD instructions do not modify the x86 flags register, so the flags field has been left out for their descriptions.

Prefixes

Some instructions allow prefixes that alter the way the instructions work. If an instruction allows prefixes, it will have a prefix field in its description.

Repeat Prefixes

The repeat prefixes are used for the string instructions to enable blocks of memory to be searched. They are set to a particular value or copied. They are not designed to be used with any other instructions, even where the compiler allows. The results of using the repeat prefixes are undefined when they are used with non-string instructions.

- **REP**: Repeats the following instruction the number of times in RCX. **`REP`** is used in conjunction with store string (**`STOS`**) and move string (**`MOVS`**) instructions. Although this prefix can also be used with **`LODS`**, there is no point in doing this. Each repetition of the instruction following the **`REP`** prefix decrements RCX.

- **REPZ, REPE**: Repeat while zero or repeat while equal are two different prefixes for exactly the same thing. This means repeat the following instruction while the zero flag is set to 1 and while RCX is not zero. As in the **`REP`** prefix, this prefix also decrements RCX at each repetition of the instruction following it. This prefix is used to scan arrays (**`SCAS`** instruction) and compare arrays (**`CMPS`** instruction)

- **REPNZ, REPNE**: Repeat while not zero or repeat while not equal are the opposites of **`REPZ`** or **`REPE`**. This prefix will repeatedly execute the instruction while the zero flag is set to 0 and RCX is not 0. Like the other repeat prefixes it decrements RCX at each repetition. This prefix is used with the same instructions as the **`REPZ`** and **`REPE`** prefixes.

Lock Prefix

Assembly instructions are not atomic (they do not happen in a single uninterruptible move by the CPU) by default.

```
add dword ptr [rcx], 2
```

This sample code will result in what is called a read-modify-write operation. The original value in RAM that RCX is pointing to will be read, 2 will added, and the result will be written. There are three steps to this operation (read-modify-write). In multithreaded applications, while one thread is in the middle of this three step operation, another thread may begin reading, writing, or modifying exactly the same address. This could lead to the second thread reading the same value as the first and only one of the threads successfully writing the actual result of the value +2.

This is known as a race condition; threads are racing to read-modify-write the same value. The problem is that the programmer is no longer in control of which threads will successfully complete their instructions and which will overlap and produce some other results. If there are race conditions in a multithreaded application, then by definition the output of the code cannot be ascertained and is potentially any one of a number of scenarios.

The **LOCK** prefix makes the following instruction atomic; it guarantees that only one thread is able to operate on some particular point in RAM at a time. While only valid for instructions that reference memory, it prevents another thread from accessing the memory until the current thread has finished the operation. This assures no race conditions occur, but at the cost of speed. Adding the **LOCK** prefix to every line of code will make any threads that try to access the same RAM work in sequence, not parallel, thus negating the performance increase that would otherwise be gained through multithreading.

```
lock add dword ptr [rcx], 2
```

In the example, the **LOCK** prefix has been placed beside the instruction. Now no matter how many threads try to access this dword, whether they are running this exact code or any other code that references this exact point in RAM, they will be queued and their accesses will become sequential. This **ADD** instruction is atomic; it is guaranteed not to be interrupted.

The **LOCK** prefix is useful for creating multithreading synchronization primitives such as mutexes and semaphores. There are no such primitives inherent to assembly and programmers must create their own or use a library.

x86 Data Movement Instructions

Move

MOV [reg8/16/32/64/mem8/16/32/64], [reg8/16/32/64/mem8/16/32/64/imm8/16/32/64]

The **MOV** instruction copies data from the second operand to the first. Both operands must be the same size. Although its name suggests that data will be moved, the data is actually copied; it will remain in the second operand after the instruction.

The **MOV** instruction is the standard assignment operator.

// C++ assignment

```
rax = 25


; Assembly equivalent

mov rax, 25
```

Note: When the first operand is a 32-bit register, this instruction clears the top 32 bits of the 64-bit version of the register to 0. This leads to a special use for MOV in x64. When you wish to clear the top 32 bits of an x86 register (for example, RDX), you can use the 32-bit version of the register as both operands:

mov edx, edx   ; Clears the top 32 bits of RDX to 0

Flags: (None)

Conditional Moves

`CMOVcc [reg16/32/64], [reg16/32/64mem16/32/64]`

This moves data from the second operand to the first operand, but only if the condition specified is true. This instruction reads the flags register to determine whether to perform a **`MOV`** or not. The condition code is placed in the mnemonic where the **`cc`** is; some common condition codes are listed in the following table. Simply replace the **`cc`** with the appropriate condition code to find the mnemonic you require.

Table 11: Some Useful Conditions for CMOVcc

Condition Code	Meaning
O	Overflow, signed overflow
NO	Not overflow, no signed overflow
Z or E	Zero or equal to, signed and unsigned
NZ or NE	Not zero or not equal to, signed and unsigned
B	Below, unsigned less than
A	Above, unsigned greater than
AE	Above or equal, unsigned
BE	Below or equal, unsigned
L	Less, signed less than
G	Greater, signed greater than
GE	Greater or equal, signed
LE	Less or equal, signed
C	Carry, unsigned overflow
NC	Not carry, no unsigned overflow

Condition Code	Meaning
S	Sign, answer was negative
NS	Not sign, answer was positive
PE	Parity even, 1's count in low byte is even
PO	Parity odd, 1's count in low byte is odd

If the second is a memory location, it must be readable whether the instruction's condition is true or not. These instructions cannot be used with 8-bit operands, only 16 bits and above.

It is often better to use conditional moves in place of conditional jumps. Conditional moves are much faster than branching (using `Jcc` instructions). A modern CPU reads ahead of the code it is actually executing, so that it can be sure the next instructions have been fetched from RAM when it requires them. When it finds a conditional branch, it guesses which of the two is most likely using a manufacturer-specific algorithm called a branch predictor. If it guesses incorrectly, there is a large performance penalty. All the machine code it had read and attempted to execute needs to be flushed from the CPU, and it has to read the code from the actual branch. It is for this reason that **`CMOVcc`** instructions were invented and why they are often so much faster than `Jcc` instructions.

```
; To move data from AX to CX only if the signed value in DX is

; equal to the value in R8W:

cmp dx, r8w

cmove cx, ax ; Only moves if dx = rw8



; To move data from AX to CX only if the unsigned value in DX is

; above (unsigned greater than) the value in R8W:

cmp dx, r8w

cmova cx, ax ; Only moves if dx > r8w?
```

Note: With a similar behavior to that of the MOV instruction, this instruction clears the top 32 bits of the 64-bit version of first operand when the operands are 32-bit registers. Even if the condition is false, the top will be cleared to 0, while the low 32 bits will be unchanged. If the condition is true, the top will be cleared to 0 and the value of the second operand will be moved into the low 32 bits.

Flags: (None)

CPUID: Function 1; read bit 15 of EDX to ensure a CPU is capable of conditional moves.

Nontemporal Move

`MOVNTI [mem32/64], [reg32/64]`

The nontemporal move instruction moves a value from a register into memory and lets the CPU know that the value will not be needed in cache. Different CPUs will do different things based on this. The CPU may ignore the nontemporal hint completely, placing the value in cache regardless of your instruction. Some CPUs will use the nontemporal hint to ensure the data is not cached, thus allowing more space in cache for data that will be needed again in the near future.

Flags: (None)

CPUID: Function1; read bit 26 (SSE2) of EDX to ensure a CPU is capable of the MOVNTI instruction.

Move and Zero Extend

`MOVZX [reg16/32/64], [reg8/16/mem8/16]`

This moves the value from the second operand into the first operand, but extends it to the second operand's size by adding zeros to the left. The source operand can only be 8 bits or 16 bits wide and it can be extended to 16 bits, 32 bits, or 64 bits.

There is no limitation on the difference between the operands. This means you can use a byte as the second and extend it to a 64-bit qword.

Flags: (None)

Move and Sign Extend

`MOVSX [reg16/32/64], [reg8/16/mem8/16]`

This converts a smaller signed integer to a larger type by copying the smaller source value to the destination's low half, and then copying the sign of the source across the upper half of the destination. This instruction cannot sign extend from a 32-bit source to a 64-bit destination, which requires using the **`MOVSXD`** instruction instead.

There is no limitation on the difference between the operands, meaning one can use a byte as the second and extend it to a 64-bit qword.

Flags: (None)

Move and Sign Extend Dword to Qword

`MOVSXD [reg64], [reg32/mem32]`

Converts a 32-bit signed integer to a 64-bit signed integer. The source is moved into the low half of the destination and the sign bit of the source is copied across all bits of the destination.

Flags: (None)

Exchange

`XCHG [reg8/16/32/64/mem8/16/32/64], [reg8/16/32/64/mem8/16/32/64]`

This swaps or exchanges the values of the two operands. This instruction can be used in place of **BSWAP** for the 16-bit registers since **BSWAP** does not allow 8-bit operands; instead of **bswap ax** you can use **xchg al, ah**.

This instruction is automatically atomic (applies the **LOCK** prefix automatically) if a memory operand is used.

Flags: (None)

Prefix: LOCK

Translate Table

XLAT [mem8]

XLATB

This instruction translates the value in AL to that of the table pointed to by RBX. Point RBX to a byte array of up to 256 different values and set AL to the index in the array you want to translate.

This instruction does not affect the top 7 bytes of RAX; only AL is altered. The instruction accomplishes something like the otherwise illegal address calculation of adding RBX to AL.

```
mov al, byte ptr [rbx+al]
```

The memory operand version does exactly the same thing, and the memory operand is completely ignored. The only purpose of the memory operand is to document where RBX may be pointing. Do not be misled; no matter what the memory operand, the table is specified by RBX as a pointer.

```
XLAT myTable ; myTable is completely ignored, [RBX] is always used!
```

Flags: (None)

Sign Extend AL, AX, and EAX

CBW

CWDE

CDQE

These instructions sign extend the various versions of RAX to larger versions. The operand is implied, and it is always AL for **CBW**, AX for **CWDE**, and EAX for **CDQE**.

CBW copies the sign of AL across AH, effectively making AX the sign extended version of what was in AL. **CWDE** copies the sign of AX across the upper half of EAX, effectively sign extending from AX to EAX. **CDQE** sign extends EAX to RAX by copying the sign of EAX across the upper half of RAX, and sign extending EAX to RAX.

Flags: (None)

Copy Sign of RAX across RDX

`CWD`

`CDQ`

`CQO`

These instructions create the signed combination of RDX:RAX used by the division instructions **`IDIV`** and **`DIV`**. They copy the sign of AX, EAX, or RAX across the same sized version of the RDX register.

`CWD` copies the sign of AX across DX such that DX is **`FFFFh`** if AX was negative, or **`0000h`** if AX was positive. This creates the 32-bit composite register DX:AX. **`CDQ`** copies the sign of EAX across EDX such that EDX becomes **`FFFFFFFFh`** if EAX is negative, and **`0`** if AX is positive. This creates the composite register EDX:EAX. **`CQO`** copies the sign of RAX across all bits of RDX. This creates the 128-bit composite register RDX:RAX.

Flags: (None)

Push to Data to Stack

`PUSH [reg16/32/64/mem16/32/64/imm16/32/64/seg16]`

This pushes a value to the stack. This results in the stack pointer being decremented by the size of the value in bytes and the value being moved into RAM. This is used to pass parameters between procedures, and also to save the return address prior to calling a procedure.

In addition to its role as the backbone to passing parameters, the stack is also used to save temporary values to free the register for some other use.

8-bit operands cannot be pushed to the stack, but you can push the segment registers FS and ES. Pushing an odd number of 16-bit values results in a misaligned stack pointer (one that is not on an address divisible by 32). You should always push an even number of values, as a misaligned stack pointer will result in a crash.

Flags: (None)

Pop Data from Stack

`POP [reg16/32/64/mem16/32/64/seg16]`

This pops data previously pushed onto the stack. This results in incrementing the stack pointer to point to the next data to be popped, and the last pushed data item being read from memory.

Flags: (None)

Push Flags Register

PUSHF

PUSHFQ

This pushes the flags register to the stack. You have the option of pushing only the low 16 bits (**PUSHF**) or the entire 64-bit rflags register (**PUSHFQ**). This instruction is useful for saving the exact state of the flags register prior to calling procedures, since the procedures will most likely alter its state. This instruction and the pop flags register instructions can also be used to set the bits of the flags register:

```
PUSHF        ; Push the state of the flags register

POP AX       ; Pop the flags register into ax

OR AX, 64    ; Set the bits using OR, BTS, BTR, or AND

PUSH AX      ; Push the altered flags

POPF         ; Pop the altered flags back into the real flags register
```

There are instructions to easily set and clear the carry and direction flags. See **CLD**, **CLC**, **STC**, and **STD**. Pushing and popping the flags register need not be used to set or clear these particular flags.

Flags: (None)

Pop Flags Register

POPF

POPFQ

This pops the values from the stack into the flags register. The flags register cannot be directly manipulated like the general purpose registers (excepting the **CLD**, **CLC**, and **STC** instructions). If you need to set particular bits of rflags, follow the example in the push flags register instruction.

Flags: Carry, Parity, Zero, Sign, Direction, Overflow

Load Effective Address

LEA [reg16/32/64], [mem]

This loads the effective address of the memory location into the source. If the source is 16 bits, only the lowest 16 bits of the address are loaded; if the source is 32 bits, then only the low 32 bits of the address are loaded into the source. Usually the source is 64 bits and the **LEA** instruction loads the entire effective address of the memory operand.

This instruction actually calculates an address and moves this into the source. It is similar to the **MOV** instruction but **LEA** does not actually read memory. It just calculates an address.

```
.data
myVar dq 23         ; Define a variable and set it to 23

.code
MyProc proc
       mov rax, myVar; MOV will read the contents of myVar into RAX
       lea rax, myVar; LEA loads the address of myVar to RAX

       ; From the LEA instruction RAX has the address of myVar
       mov qword ptr [rax], 0    ; So we could set myVar to 0 like this
       ret
MyProc endp
End
```

Note: Because this instruction actually just calculates an address and complex addressing modes (e.g. [RBX+RCX*8]), the instruction can be used, but it does not make any attempt to read from the address, and it can be used to perform fast arithmetic.

For example, to set RAX to 5 * RCX you could use the following:

`LEA RAX, [RCX+RCX*4]`

To set RBX to R9+12 you could use the following:

`LEA RBX, [R9+12]`

This optimization technique and a multitude more are detailed in Michael Abrash's *Black Book of Graphics Programming* (see the Recommended Reading section for a link).

Flags: (None)

Byte Swap

`BSWAP [reg32/64]`

This reverses the order of the bytes in the source. This instruction was designed to swap the endianness of values. That is, it is used to change from little endian to big endian and vice versa. With the dominance of x86-based CPUs (x86 uses little endian), the desire to change endianness is almost gone, so the instruction is also useful in reversing character strings.

Note: If you need to "BSWAP reg16" you should use XCHG instruction. The BSWAP instruction does not allow for 16-bit parameters, so instead of "BSWAP AX" you can use "XCHG AL, AH".

Flags: (None)

x86 Arithmetic Instructions

Addition and Subtraction

```
ADD [reg8/16/32/64/mem8/16/32/64],
[reg8/16/32/64/mem8/16/32/64/imm8/16/32]
```

```
SUB [reg8/16/32/64/mem8/16/32/64],
[reg8/16/32/64/mem8/16/32/64/imm8/16/32]
```

This adds or subtracts the second operand from the first and stores the result in the first operand. For addition it does not matter, but when using `SUB` it is important to note that the second operand is subtracted from the first, not the other way round.

These instructions can be used for both signed and unsigned arithmetic; it is important to know how to read the flags, since the flags reflect both the signed and unsigned result.

If you are doing unsigned arithmetic, you should read the carry flag. The carry flag will be 0 if there was no final overflow. If there was a final overflow (indicating a carry or borrow on the final bit of the operation), it will be set to 1.

If you are doing signed arithmetic, or if there was no final carry or borrow on the second to last bit of the operation (since the final bit is the sign bit), the overflow flag will be 0. If there was a final carry or borrow, the overflow flag will be set to 1.

If the result of the addition or subtraction is exactly 0, then the zero flag will be set.

If the result was a negative number (this can be ignored if unsigned arithmetic is being done), then the sign flag will be set.

Flags: Carry, Parity, Zero, Sign, Overflow

Prefix: LOCK

Add with Carry and Subtract with Borrow

```
ADC [reg8/16/32/64/mem8/16/32/64],
[reg8/16/32/64/mem8/16/32/64/imm8/16/32]
```

```
SBB [reg8/16/32/64/mem8/16/32/64],
[reg8/16/32/64/mem8/16/32/64/imm8/16/32]
```

These instructions do the same as the **ADD** and **SUB** instructions, except that they also add or subtract the carry flag (they add or subtract an additional 1 or 0 depending on the state of the carry flag). They are useful for performing arbitrarily large integer additions or subtractions where the number being worked on does not fit inside a 64-bit register, but is broken into multiple 64-bit digits.

You can also use these instructions to set a register to the carry flag. To set EAX to the carry flag you can use the following:

```
MOV EAX, 0          ; Clear EAX to 0.

                    ; You can't use XOR here because that would clear

                    ; the carry flag to 0.

ADC EAX, EAX ; Sets EAX to 1 or 0 based on the carry flag
```

Flags: Carry, Parity, Zero, Sign, Overflow

Prefix: LOCK

Increment and Decrement

INC [reg8/16/32/64/mem8/16/32/64]

DEC [reg8/16/32/64/mem8/16/32/64]

These instructions increment (add 1 to) or decrement (subtract 1 from) a register or memory variable. They are often used in conjunction with a register to create looping structures. A common pattern is something like the following which will loop 100 times:

```
mov cx, 100  ; Number of times to loop


LoopHead:    ; Start of the loop


             ; Body of the loop


      dec cx        ; Decrement counter

      jnz LoopHead ; Loop if there's more, i.e. 100 times
```

Note: INC and DEC do not set the carry flag. If you need to perform INC or DEC but also set the carry flag, it is recommended to use ADD or SUB with 1 as the second operand.

Flags: Parity, Zero, Sign, Overflow

Prefix: Lock

Negate

`NEG [reg8/16/32/64/mem8/16/32/64]`

This negates a signed number so that negative values become their positive counterparts and vice versa. This is the equivalent to flipping each of the bits and adding 1 to the result. This is called two's complement of a number, as opposed to the one's complement, which can be obtained with the **`NOT`** instruction.

Note: x86 and x64 CPUs perform multiplication slowly compared to many of the bit manipulation instructions; if you need to multiply a number by -1 it is always quicker to use NEG than IMUL.

Flags: Carry, Parity, Zero, Sign, Overflow

Prefix: LOCK

Compare

`CMP [reg8/16/32/64/mem8/16/32/64], [reg8/16/32/64/mem8/16/32/64/imm8/16/32]`

This compares the two operands and sets the flags register to indicate the relationship between the two operands.

This instruction actually does exactly the same as the **`SUB`** instruction, but it does not store the result, it just sets the flags. The second operand is subtracted from the first operand and the flags are set accordingly, but the destination operand is not altered. Usually the compare instruction is followed by conditional jumps or conditional moves.

This instruction is used to set the flags and subsequently perform some conditional operation based on the results. It is very important to note how the operands are being compared by the **`CMP`** instruction, since comparisons such as **`>`**, **`>=`**, **`<`**, and **`<=`** are important to the order of the operands.

```
cmp dx, ax

jg SomeLabel ; Jump if DX > AX
```

Note: CMP op1, op2 is the same as asking, "what relation does the first operand have to the second," not the other way round. The second operand is subtracted from the first.

Flags: Carry, Parity, Zero, Sign, Overflow

Multiply

`MUL [reg8/16/32/64]`

`IMUL [reg8/16/32/64]`

`IMUL [reg8/16/32/64], [reg8/16/32/64/mem8/16/32/64/imm]`

`IMUL [reg8/16/32/64], [reg8/16/32/64/mem8/16/32/64], [imm8]`

`MUL` performs unsigned integer multiplication and **`IMUL`** performs signed integer multiplication.

There is only a single-operand version of **`MUL`**, whereas **`IMUL`** has three versions. In the single-operand version of **`IMUL`** or **`MUL`**, the second operand is implied and the answer is stored in predefined implied registers. The implied second operand is the appropriate size of the RAX register, so if the operand is 8 bits, then the second implied operand is AL. If the source operand is 64 bits then the implied second operand is RAX.

The answer to the multiplication is stored in AX for 8-bit multiplications. For the other data sizes (16-bit, 32-bit, and 64-bit operands), the answer is stored with the upper half in the appropriate size of RDX and the lower half in the appropriate size of RAX. This is because the original 16-bit CPUs did not possess registers large enough to store the possible 32-bit result from a 16-bit multiplication, so the composite 32-bit of DX:AX was used. When 32-bit CPUs came about, exactly the same thing happened. The 64-bit answer from a 32-bit multiplication could not be stored in a 32-bit register, so the composite of EDX:EAX is used. And now with our 64-bit CPUs, the 128-bit answer is stored in the composite of RDX:RAX.

If anything ends up in the top half of the answer (AH, DX, EDX, or RDX), then the carry and overflow flags are set to 1, otherwise they are 0.

Table 12

Operand 1	Implied Operand 2	Answer
8 bits	AL	AX
16 bits	AX	DX:AX
32 bits	EAX	EDX:EAX
64 bits	RAX	RDX:RAX

The two-operand version of **`IMUL`** simply multiplies the second operand by the first and stores the result in the first. The overflow (any bits from the result that do not fit into the first operand) are lost and the carry and overflow flags are set to 1. If there is no overflow, the entire answer fits into the first operand and the carry and overflow flags are set to 0.

In the three-operand version of **`IMUL`**, the second operand is multiplied by the third operand (an immediate value) and the result is stored in the first operand. Once again, if the result overflows, both the carry and overflow flags are set to 1, otherwise they are cleared.

Note: These instructions are quite slow, so if it is possible it may be quicker to swap a multiplication for a shift (SHL) or use the LEA instruction.

Flags: Carry, Overflow

Signed and Unsigned Division

`DIV [reg8/16/32/64/mem8/16/32/64]`

`IDIV [reg8/16/32/64/mem8/16/32/64]`

Unlike **`IMUL`**, there are only single-operand versions of the division instructions. **`DIV`** divides unsigned integers and **`IDIV`** divides signed integers. These instructions return both the quotient and remainder of the division.

The single operand given to the instruction is the divisor (the y in x/y of the division). The dividend (the x in the x/y division) is implied. See the examples in Table 13 for the location of the implied dividend. The quotient ends up in the appropriate size of RAX and the remainder goes in RDX.

Note: Division has always been one of the slowest instructions (perhaps 30–40 times slower than addition). This is still the case today. If possible, division should be avoided completely in tight loops. If a number is to be divided by a power of 2, use the SAR (Arithmetic Shift Right) instead of signed division and SHR instead of unsigned. If there are many divisions to be performed, consider using SSE.

Note: Be very careful about what is in RDX. If the number being divided is small enough to fit entirely in the appropriate size of RAX, you must remember RDX! Either clear RDX using XOR for unsigned division or copy RAX's sign across it using CWD, CDQ, or CQO.

For example, if we wanted to calculate 100/43 using signed dwords (this code would work for -100/43 as well), use something like the following:

```
mov eax, 100 ; Move implied dividend into EAX

mov ecx, 43  ; Move divisor into ECX

cdq          ; Copy sign of EAX across EDX

idiv ecx     ; Perform division, EAX gets quotient, EDX gets remainder!
```

Table 13: Summary of Divide Instruction Operands and Results

Operand 1 (Divisor)	Implied Dividend	Quotient	Remainder
8 bits	AX	AL	AH
16 bits	DX:AX	AX	DX
32 bits	EDX:EAX	EAX	EDX
64 bits	RDX:RAX	RAX	RDX

Flags: None (All flags are undefined after a divide!)

x86 Boolean Instructions

Boolean And, Or, Xor

`AND [reg8/16/32/64/mem8/16/32/64], [reg8/16/32/64/mem8/16/32/64/imm8/16/32]`

`OR [reg8/16/32/64/mem8/16/32/64], [reg8/16/32/64/mem8/16/32/64/imm8/16/32]`

`XOR [reg8/16/32/64/mem8/16/32/64], [reg8/16/32/64/mem8/16/32/64/imm8/16/32]`

These instructions **`AND`**, **`OR`**, or **`XOR`** the operands and store the result in the first operand. Each pair of bits (one from the source and the corresponding one from the destination) has the operation applied and the answer stored exactly the same as C++ Boolean operations.

Table 14: AND Truth Table

Bit 1	Bit 2	Result
0	0	0
0	1	0
0	0	0
1	1	1

Table 15: OR Truth Table

Bit 1	Bit 2	Result
0	0	0
0	1	1
0	0	1
1	1	1

Table 16: XOR Truth Table

Bit 1	Bit 2	Result
0	0	0
0	1	1
0	0	1
1	1	0

The overflow and carry flags are cleared to 0 while the sign and zero flags indicate the result.

Note: Traditionally the XOR instruction is faster than MOV so programmers usually use XOR to clear a register to 0. If both operands to an XOR have exactly the same value, then XOR returns 0, so to clear RAX to 0 "XOR RAX, RAX" is more common than "MOV RAX, 0" even though today's CPUs probably perform the XOR no faster.

Flags: Carry, Parity, Zero, Sign, Overflow

Prefix: LOCK

Boolean Not (Flip Every Bit)

`NOT [reg8/16/32/64/mem8/16/32/64]`

This instruction flips every bit in the operand given such that zeroes become ones and ones become zeroes. It is the bitwise or Boolean **`NOT`** and is sometimes called the one's complement, as opposed to the **`NEG`** instruction, which returns the two's complement.

Flags: (None)

Prefix: LOCK

Test Bits

`TEST [reg8/16/32/64/mem8/16/32/64], [reg8/16/32/64/mem8/16/32/64/imm8/16/32]`

This instruction is to bitwise tests as **`CMP`** is to arithmetic tests. It performs a Boolean **`AND`** instruction between the source and destination, but does not set the result in the destination. Instead it just alters the flags.

The carry flag is always reset to 0, the parity flag is set, and the zero and sign flags reflect the result of the Boolean **`AND`**.

For example, if you wish to know if any bits in the third byte of EAX are set to 1, you could use **`TEST`** as follows:

```
test eax, 00ff0000h  ; 00ff0000h is only 1's in the 3rd byte

jnz ThereAreOnes     ; If zero flag isn't set, EAX has something in 3rd byte

jz ThirdByteIsClear  ; If zero flag is set then EAX has nothing in 3rd byte
```

If you wish to test whether RDX contains an even number, you can employ the **`TEST`** instruction as follows:

```
test rdx, 1    ; Is the first bit 1?

jz EvenNumber  ; If it is not, the number is even

jnz OddNumber  ; Otherwise the number in RDX is odd
```

Flags: Carry, Parity, Zero, Sign, Overflow

Shift Right and Left

SHL [reg8/16/32/64/mem8/16/32/64], [CL/imm8]

SHR [reg8/16/32/64/mem8/16/32/64], [CL/imm8]

SAR [reg8/16/32/64/mem8/16/32/64], [CL/imm8]

This shifts the bits in the first operand by the amount specified in the second operand. These instructions shift the bits left (**SHL**), right (**SHR**), or arithmetically right (**SAR**). The second operand can be the CL register or an immediate 8-bit value (there is also a special version of this instruction when this operand is the immediate value 1).

SHL can be used to multiply a signed or unsigned number by a power of 2. **SHR** can be used to divide an unsigned number by a power of 2.

```
shl rax, 5   ; RAX = RAX * (2 ^ 5)


shr rdx, 3   ; RDX = RDX / (2 ^ 3) where RDX is unsigned, use SAR for signed
```

With the **SHL** and **SHR** instructions, the vacated bits on the right and left side are filled with 0 just as the shift operations in C++. The arithmetic right shift (**SAR**) shifts the bits right, but fills the vacant bits on the left with the sign bit, so it can be used to divide a signed number by a power of 2.

If the second operand is 0 (whether it is immediate or CL) the flags will not be set.

If the shift is not zero, then the flags are affected. The carry flag holds the final bit that was shifted out of the destination.

Flags: Carry, Parity, Zero, Sign, Overflow

Rotate Left and Right

ROL [reg8/16/32/64/mem8/16/32/64], [CL/imm8]

ROR [reg8/16/32/64/mem8/16/32/64], [CL/imm8]

This rotates the first operand by the number of bits specified in the source. The rotate operation is the same as bit shifting, only as bits are shifted out on the right (**ROR**) they reenter on the left, or as bits are shifted out on the left (**ROL**) they reenter on the right.

Note: There are special versions of these rotate and shift instructions. If the immediate operand is used and it is exactly 1, the overflow flag is set. This indicates whether the sign of the first operand has changed. If the overflow flag is 1 after the instruction then the sign of the destination operand has been changed, otherwise it has stayed the same.

Flags: Carry, Overflow

Rotate Left and Right Through the Carry Flag

`RCL [reg8/16/32/64/mem8/16/32/64], [CL/imm8]`

`RCR [reg8/16/32/64/mem8/16/32/64], [CL/imm8]`

This rotates the destination left (**`RCL`**) or right (**`RCR`**) through the carry flag. These instructions are the same as the **`ROL`** and **`ROR`** rotate instructions, only they also rotate the carry flag from the flags register as if it was part of the destination.

For **`RCL`** (rotate left through the carry flag), the register being rotated can be thought of as having the carry flag as its ninth bit (most significant). For **`RCR`** (rotate right through the carry flag), the register being rotated has the carry flag as the first bit (least significant).

Flags: Carry, Overflow

Shift Double Left or Right

`SHLD [reg/mem16/32/64], [reg16/32/64], [CL/imm8]`

`SHRD [reg/mem16/32/64], [reg16/32/64], [CL/imm8]`

This shifts the first operand left (**`SHLD`**) or right (**`SHRD`**), and shifts in the bits of the second operand from the left (**`SHRD`**) or right (**`SHLD`**). The number of bits to shift is specified in the third operand. This instruction lets you shift the contents of a register into another register or memory location. The instruction does not alter the second operand.

Note: There is no version of these instructions that take 8-bit operands; if an 8-bit SHLD or SHRD is required, you should use one of the 16-bit x86 registers. For example, you can use AX to shift the bits from AL to and from the bits in AH.

Flags: Overflow, Sign, Zero, Parity, Carry

Bit Test

`BT [reg16/32/64/mem16/32/64], [reg16/32/64/imm8]`

`BTC [reg16/32/64/mem16/32/64], [reg16/32/64/imm8]`

`BTR [reg16/32/64/mem16/32/64], [reg16/32/64/imm8]`

`BTS [reg16/32/64/mem16/32/64], [reg16/32/64/imm8]`

This copies the bit at the zero-based index specified by the second operand from the first operand into the carry flag.

```
bt eax, 4    ; Copy the 4th bit of EAX into the Carry Flag
```

A special version of these instructions is used when the first operand is memory and the second is a register. In this case, the entirety of RAM becomes a bit array instead of the regular byte array! The parameter passed becomes the base of the bit array (its zero bit, the rightmost, is the start of the bit array whose expanse is the remainder of RAM). All the rules for accessing memory still apply and segmentation faults will be generated.

```
mov eax, 27873      ; We wish to know what the 27873th bit is.

bt MyVariable, eax  ; Beginning from rightmost bit in MyVariable.
```

BT tests the bit and copies its value to the carry flag. **BTC** tests the bit and then complements it in the first operand. **BTR** tests the bit and then resets it to 0 in the first operand. **BTS** tests the bit and then sets it to 1 in the first operand.

Flags: Carry (all others except for direction are undefined)

Prefix: LOCK (But not on BT since it cannot write to memory)

Bit Scan Forward and Reverse

BSF [reg16/32/64], [reg16/32/64/mem16/32/64]

BSR [reg16/32/64], [reg16/32/64/mem16/32/64]

This searches the second operand right to left (forward, **BSF**) or left to right (reverse, **BSR**) for the first bit set to 1. If a bit is found set to 1, the first operand is set to its index and the zero flag is cleared. If there is no bit set to 1 at all in the second operand, the zero flag is set to 1.

The bit indices do not change regardless of the scan's direction. If there is only one bit set in the operand, both **BSF** and **BSR** will return exactly the same value. If there is more than one bit set, they will return different values.

```
mov ax, 2

bsf bx, ax  ; Places 1 into bx

bsr bx, ax  ; Places 1 into bx
```

Flags: Zero (all the rest except for direction are undefined)

Conditional Byte Set

SETO [reg8/mem8]	**Overflow OF = 1**
SETNO [reg8/mem8]	**Overflow OF = 0**
SETB, SETC, SETNAE [reg8/mem8]	**Below, carry CF = 1**
SETNB, SETNC, SETAE [reg8/mem8]	**Above or equal, carry CF = 0**

SETZ, SETE [reg8/mem8]	**Equal, zero ZF = 1**
SETNZ, SETNE [reg8/mem8]	**Not equal, zero ZF = 0**
SETBE, SETNA [reg8/mem8]	**Below or equal, CF = 1 or ZF = 1**
SETNBE, SETA [reg8/mem8]	**Above, CF = 0 and ZF = 0**
SETS [reg8/mem8]	**Sign SF = 1**
SETNS [reg8/mem8]	**Sign SF = 0**
SETP, SETPE [reg8/mem8]	**Parity is even PF = 1**
SETNP, SETPO [reg8/mem8]	**Parity is odd PF = 0**
SETL, SETNGE [reg8/mem8]	**Less than SF <> OF**
SETNL, SETGE [reg8/mem8]	**Not less than SF = OF**
SETLE, SETNG [reg8/mem8]	**Less or equal ZF = 1 or SF <> OF**
SETNLE, SETG [reg8/mem8]	**Greater than ZF = 0 and SF <> OF**

These instructions set the operand to 0 or 1 based on whether the flags meet the specified condition. The destination becomes 1 if the condition is met, otherwise it becomes 0. The conditions all reference the flags so this instruction is usually placed after a **CMP** or **TEST**; it is similar to the **CMOVcc** instructions, only it moves 0 or 1 instead of moving the second operand into the first like the **CMOVcc** instructions.

Flags: (None)

Set and Clear the Carry or Direction Flags

STC	**Set carry flag CF = 1**
CLC	**Clears the carry flag to 0**
STD	**Set direction flag DF = 1**
CLD	**Clears the direction flag**

STC sets the carry flag to 1 while **CLC** clears it to 0. Likewise, **STD** sets the direction flag to 1 while **CLD** clears it to 0. Setting or clearing the direction flag is useful for setting the direction the string instructions move their automatic pointers.

Flags: Carry (STC and CLC), Direction (STD and CLD)

Jumps

`JMP`	`Unconditionally jump`
`JO`	`Jump on overflow`
`JNO`	`Jump on no overflow`
`JB,JC,JNAE`	`Jump if below, CF = 1, not above or equal`
`JNB,JNC,JAE`	`Jump if not below, CF = 0, above or equal`
`JZ,JE`	`ZF = 1, jump if equal`
`JNZ,JNE`	`ZF = 0, jump if not equal`
`JBE,JNA`	`Jump if below or equal, not above, CF or ZF = 1`
`JNBE,JA`	`Jump if not below or equal, above, CF and ZF = 0`
`JS`	`Jump on sign, SF = 1`
`JNS`	`Jump on no sign, SF = 0`
`JP,JPE`	`Jump on parity, parity even, PF = 1`
`JNP,JPO`	`Jump on no parity, parity odd, PF = 0`
`JL,JNGE`	`Jump if less, not greater or equal, SF != OF`
`JNL,JGE`	`Jump if not less, greater, or equal, SF = OF`
`JLE,JNG`	`Jump if less or equal, not greater than, ZF = 1 or SF != OF`
`JNLE,JG`	`Jump if not less or equal, greater than, ZF = 0 and SF = OF`
`JCXZ`	`Jump if CX = 0`
`JECXZ`	`Jump if ECX = 0`
`JRCXZ`	`Jump if RCX = 0`

Each of the jump instructions takes a single operand. This operand is usually a label defined somewhere in the code but it is actually fairly flexible. The addressing modes available to the **`Jxx`** instructions are as follows:

```
JMP [reg/mem/imm]

Jcc [imm8/16/32]

JcCX [imm/8/16/32]
```

The instructions are sometimes called branching; the RIP register will fall through to the operand if the condition is true, otherwise the RIP will fall through to the next line of code.

Usually the operand is a label.

```
      cmp edx, eax

      jg SomeLabel ; Jump if greater



             ; Some code to skip



SomeLabel:
```

Flags: (None)

Call a Function

`CALL [reg16/32/64/mem16/32/64]`

`CALL [imm16/32]`

This calls a procedure. This instruction pushes the offset of the next instruction to the stack and jumps the RIP register to the procedure or label given as the first operand. It is essentially exactly the same as a jump instruction, only the return address is pushed to the stack so the RIP can return and resume execution of the calling function using a **`RET`** instruction from within the body of the subprocedure.

Note: There used to be a distinction between near and far calls. Far calls ended up in another code segment. However, since x64 uses a flat memory model, all calls are near calls.

Flags: (None)

Return from Function

`RET`

This instruction returns from a function called with the **`CALL`** instruction. This is achieved by popping from the return address into the RIP.

Flags: (None)

x86 String Instructions

Load String

`LODS [mem8/16/32/64]`

`LODSB`	**`Load byte`**
`LODSW`	**`Load word`**
`LODSD`	**`Load dword`**

`LODSQ`	`Load qword`

These instructions load a byte, word, dword, or qword into the appropriate size of the RAX register, and then they increment (or decrement depending on the direction flag) RSI to point to the next byte, word, dword, or qword. They read whatever RSI (the source index register) is pointing to in RAX and then move RSI to point to the next data of the same size.

The **`REP`** prefix can be used, but it is pointless since no operation can be performed on the consecutive values being stored in RAX; the loop will simply run through the string and leave you with only the final value in RAX.

Note: Even the versions with a memory operand read only from whatever RSI is pointing to. The memory operand is almost completely ignored. Its only purpose is to indicate both what size data should be read and into what version of RAX it should be placed.

Note: If the direction flag, DF, is 1 as set by the STD instruction, the string instructions will decrement RDI and RSI instead of incrementing. Otherwise the instruction will increment.

Flags: (None)

Prefix: REP

Store String

`STOS [mem8/16/32/64]`

`STOSB`	`Store byte`
`STOSW`	`Store word`
`STOSD`	`Store dword`
`STOSQ`	`Store qword`

This stores AL, AX, EAX, or RAX to the memory pointed to by RDI and increments (or decrements depending on the direction flag) RDI. This instruction can be used to quickly set a large number of values to the same thing. RDI is incremented or decremented by the size of the data type each repetition.

To set 100 words to 56, make sure RDI is pointing to the start of the 100 words in memory.

```
lea rdi, someArray ; Point RDI to the start of the array

mov rcx, 100

mov ax, 56

rep stosw
```

Note: Even the versions with a memory operand only store to RDI. The memory operand is almost completely ignored. The memory operand's only purpose is to indicate which of AL, AX, EAX, or RAX should be stored and how much to increment RDI.

Note: If the direction flag, DF, is 1 as set by the STD instruction, the string instructions will decrement RDI and RSI instead of incrementing. Otherwise the instruction will increment.

Flags: (None)

Prefix: REP

Move String

MOVS [mem8/16/32/64], [mem8/16/32/64]

MOVSB

MOVSW

MOVSD

MOVSQ

This moves the byte, word, dword, or qword pointed to by RSI to that pointed to by RDI and increments both RSI and RDI to point to the next (or decrements depending on the direction flag). Both RSI and RDI are incremented by the size of the data type each repetition. This instruction can be used to quickly move data from one array to another. Set RSI at the start of the source array and RDI to the start of the destination and place the number of elements to copy into RCX.

```
lea rsi, SomeArray

lea rdi, SomeOtherArray

mov rcx, 1000


rep movsq    ; Copy 8000 byes
```

Note: Even the versions with memory operands copy data from RSI to RDI; the memory operand's only use is to specify the size of the data to copy.

Note: If the direction flag, DF, is 1, as set by the STD instruction, the string instructions will decrement RDI and RSI instead of incrementing. Otherwise the instruction will increment.

Prefix: REP

Scan String

SCAS [mem8/16/32/64], [mem8/16/32/64]

SCASB

SCASW

SCASD

SCASQ

This compares the byte, word, dword, or qword pointed to by RDI to the appropriate size of RAX and sets the flags accordingly. It then increments (or decrements depending on the direction flag) RDI to point to the next element of the same size in RAM. This instruction is meant to be used with the **REPE**, **Z**, **NE**, and **NZ** prefixes, and it scans a string for the element in RAX or until the count in RCX reaches 0.

To scan an array of bytes up to 100 bytes to find the first occurrence of the character “a,” use the following:

```
      lea rdi, arr     ; Point RDI to some array
      mov rcx, 100     ; Load max into RCX
      mov al, 'a'      ; Load value to seek into AL
repnz scasb             ; Search for AL in *RDI
      jnz NotFound     ; If the zero flag is not set after the
                       ; scan AL is not in arr
      lea rax, [arr+1] ; Otherwise we can find the index of the
                       ; first occurrence of AL
      sub rdi, rax     ; By subtracting arr+1 from the address where we found AL
```

Note: Even the versions with a memory operand scan only whatever RDI is pointing to. The memory operand is almost completely ignored. The memory operand’s only purpose is to indicate which of AL, AX, EAX, or RAX should be compared to RDI and how much to increment RDI.

Note: If the direction flag, DF, is 1 as set by the STD instruction, the string instructions will decrement RDI and RSI instead of incrementing. Otherwise the instruction will increment.

Flags: Overflow, Sign, Zero, Parity, Carry

Prefix: REPE, REPZ, REPNE, REPNZ

Compare String

CMPS [mem8/16/32/64], [mem8/16/32/64]

CMPSB

CMPSW

CMPSD

CMPSQ

These instructions compare the data pointed to by RSI to the data pointed to by RDI, and set the flags accordingly. They increment (or decrement depending on the direction flag) RSI and RDI depending on the operand size. They can be used to scan *RSI and *RDI for the first byte, word, dword, or qword that is different or the first that is the same between the two arrays.

Note: Even the versions with a memory operand compare only RSI to RDI. The memory operand is almost completely ignored. The memory operand's only purpose is to indicate how much RDI and RSI should be incremented or decremented each round.

Note: If the direction flag, DF, is 1 as set by the STD instruction, the string instructions will decrement RDI and RSI instead of incrementing. Otherwise the instruction will increment.

Prefix: REPE, REPZ, REPNE, REPNZ

x86 Miscellaneous Instructions

No Operation

`NOP`

This instruction does nothing but wait for a clock cycle. However, it is useful for optimizing pipeline usage and patching code.

Flags: (None)

Pause

`Pause`

This instruction is similar to **`NOP`**, but it also indicates to the CPU that the thread is in a spin loop so that the CPU can use any power-saving features it has.

Flags: (None)

Read Time Stamp Counter

`RDTSC`

This instruction loads the time stamp counter into EDX:EAX. The time stamp counter is the number of clock cycles that have elapsed since the CPU was reset. This is useful for getting extremely fine grained timing readings.

The following could be a small function to read the time stamp counter:

```
ReadTimeStamp proc

        rdtsc

        shl rdx, 32
```

```
      or rax, rdx

      ret

ReadTimeStamp endp
```

Getting performance readings at the level of single clock cycles is difficult, since Windows is constantly switching between the running applications and multitasking. The best thing to do is run tests repeatedly. You should test how long the **`ReadTimeStamp`** procedure takes, and subtract this from subsequent tests, and then take the average or best clock cycle readings as the benchmark.

Flags: (None)

Loop

`LOOP [Label]`

`LOOPE [Label]`

`LOOPNE [Label]`

`LOOPZ [Label]`

`LOOPNZ [Label]`

These will decrement the RCX counter and jump to the specified label if a condition is met. For example, the **`LOOP`** instruction decrements RCX and repeats from the label until it is 0. Then it does not branch, but falls through to execute the code following the loop. The **`LOOP`** instructions are almost never used, because the manual decrement and jump is faster.

```
dec rcx

jnz LoopTop
```

In addition to being faster, the manual two-line version allows the programmer to specify which register is used as the counter where **`LOOPxx`** makes use of RCX.

The **`LOOP`** instructions are interesting, but they do not set the flags register at all where the manual **`DEC`** and **`JNZ`** does. When RCX reaches 0 in the **`LOOP`**, the RIP register will fall through, but the zero flag will not be altered from the last setting it had in the body of the loop.

If it is important for a loop’s structural components not to alter the flags register, then using the **`LOOP`** instruction in place of the manual two-line loops may be worth investigating. With **`LOOPE`** and **`LOOPZ`**, if the zero flag is 1, the loop falls through. With **`LOOPNE`** and **`LOOPNZ`**, if the zero flag is 0, the loop falls through.

The loops can be broken either by RCX becoming 0 or on the condition of the zero flag. This may lead to some confusion. If the zero flag happens to be set during the first iteration of a long loop, then the **`LOOPE`** instruction will not decrement RCX and repeat the loop. The loop will break on the condition of the zero flag.

As mentioned previously, the **LOOP** instructions are often not used. They are slower than the two-line manual loop tail in the last sample because they do more than simply **DEC** the counter and jump. If the **LOOP** instructions happen to perform exactly what you need, they may give a good performance increase as opposed to checking the zero flag. However, in the vast majority of cases a simple **DEC** and **JNZ** will be faster.

CPUID

MOV EAX, [function]   ; Move the function number into EAX first

CPUID

This instruction returns information on the executing CPU, including data on the CPU vendor, cache size, number of cores, and the available instruction sets.

The **CPUID** instruction itself may not be available on older CPUs. The recommended method for testing if the **CPUID** instruction can be executed is to toggle the 21st bit of the flags register. If this bit can be set to 1 by the program, then the CPU understands the **CPUID** instruction. Otherwise, the CPU does not understand the **CPUID** instruction.

The following is an example of testing for the availability of the **CPUID** instruction:

```
      pushfq       ; Save the flags register

      push 200000h ; Push nothing but bit 21

      popfq        ; Pop this into the flags

      pushfq       ; Push the flags again

      pop rax      ; This time popping it back into RAX

      popfq        ; Restore the original flag's state

      cmp rax, 0   ; Check if our bit 21 was changed or stuck

      je No_CPUID  ; If it reverted back to 0, there's no CPUID
```

Not all CPUs are able to execute all instructions. Modern CPUs are usually capable of executing more instruction sets than older ones. In order to know if the CPU executing your code is aware of any particular instruction set, you can call the special **CPUID** instruction.

The **CPUID** instruction takes no operands, but EAX is implied. The value in EAX when the instruction is called is read by the CPU as the function number.

There is a great number of different functions, and each CPU manufacturer is able to define its own. Manufacturer-specific functions usually have the top 16 bits of EAX set to 8000, for example. The functions for determining many instruction sets are standard across Intel, AMD, and VIA.

To call a particular function, first **MOV** the function number into EAX and then use the **CPUID** instruction.

```
mov eax, 1   ; Function number 1

cpuid        ; No formal parameters but EAX is implied!
```

`CPUID` function 1 (calling **`CPUID`** when EAX is set to 1) lists the feature identifiers; feature identifiers are the instruction sets that the CPU knows. It lists the possible instruction sets by storing a series of bit flags in ECX and EDX. Bits are set to 1 to indicate that the CPU is capable of a particular feature, and 0 to indicate that it is not. In the following table, some of the most useful features have been listed with the register (ECX or EDX) and the bit number to check for the feature. There are many more features with additional features added with each new generation of CPU.

Table 17: Abridged Feature Identifiers

Function Number (EAX)	Register (ECX/EDX)	Bit Index in ECX/EDX	Feature
1	ECX	28	AVX
1	ECX	25	AES
1	ECX	23	Pop Count, POPCNT
1	ECX	20	SSE4.2
1	ECX	19	SSE4.1
1	ECX	9	SSSE3
1	ECX	0	SSE3
1	EDX	26	SSE2
1	EDX	25	SSE
1	EDX	23	MMX
1	EDX	15	Conditional Moves
1	EDX	4	RDTSC
1	EDX	0	x87 FPU

The following example tests for MMX and SSE4.2. In the assembly file, the register (ECX or EDX) and the bit number can be changed to test for any feature. For a full list of what **`CPUID`** can do on AMD chips, consult *CPUID Specification* by AMD. For a full list of what **`CPUID`** can do on Intel chips, consult *Intel Processor Identification and the CPUID Instruction* by Intel. Links to the manuals and other recommended reading can be found at the end of this book.

```
// This is the C++ file

#include <iostream>

using namespace std;

extern "C" bool MMXCapable();

extern "C" bool SSE42Capable();


int main()

{

      if(MMXCapable()) cout<<"This CPU is MMX capable!"<<endl;

      else cout<<"This CPU does not have the MMX instruction set :("<<endl;


      if(SSE42Capable()) cout<<"This CPU is SSE4.2 capable!"<<endl;

      else cout<<"This CPU does not have the SSE4.2 instruction set"<<endl;

      cin.get();

      return 0;

}


; This is the assembly file

.code

; bool MMXCapable()

; Returns true if the current CPU knows MMX

; else returns false

MMXCapable proc

      mov eax, 1             ; Move function 1 into EAX

      cpuid                  ; Call CPUID
      shr edx, 23          ; Shift the MMX bit to position 0

      and edx, 1           ; Mask out all but this bit in EDX

      mov eax, edx         ; Move this answer, 1 or 0, into EAX

      ret                  ; And return it

MMXCapable endp


; bool SSE42Capable()

; Returns true if the current CPU knows SSE4.2

; else returns false

SSE42Capable proc

      mov eax, 1           ; Move function 1 into EAX

      cpuid                ; Call CPUID

      shr ecx, 20          ; Shift SSE4.2 bit to position 0

      and ecx, 1           ; Mask out all but this bit

      mov eax, ecx         ; Move this bit into EAX

      ret                  ; And return it

SSE42Capable endp

end
```

Note: It was common in the past to simply call an instruction and let an exception be thrown by the CPU if the instruction set was not supported. There is a slight possibility that a given machine code will execute on an older CPU without throwing an exception, but it will actually execute some other instruction. For this reason, the CPUID instruction is the recommended method for testing if instruction sets are available.

Chapter 8 SIMD Instruction Sets

SIMD stands for Single Instruction, Multiple Data. It is a type of parallel programming. The idea of SIMD is to perform the same instructions on multiple pieces of data at once. The SIMD instructions added to the x86 architecture were originally used to speed up multimedia processing. It is common in multimedia programming to perform the same operation for every pixel on the screen or perhaps every byte in a stream of sound data.

Since its introduction, hundreds of new instructions have been added to x86 CPUs. It is a very different way to program. It requires a new set of eyes, debugging skills, new algorithms, and data structures. The SIMD instruction sets have their own new registers and instructions. These new registers are often larger than the original x86 registers, and they can be thought of as holding a small array of values. For instance, a 64-bit SIMD register can hold an array of 8 bytes, 4 words, or 2 dwords.

Many of the SIMD style instructions operate on corresponding elements in two registers. Addition of bytes in SIMD involves adding the two lowest bytes from both operands and storing the answer in the lowest byte, adding the two second-to-lowest and storing this in the second lowest byte of the answer—this goes on until the two top bytes are added together, and their result is stored in the top element of the answer. Although many instructions work in this manner, just as many do something completely different.

As an example, consider a loop in scalar programming which adds the elements from two arrays. Perhaps the arrays contain 8 elements. The elements might be added with C++ as follows:

```
for(int i = 0; i < 8; i++)

{

      arr1[i] += arr2[i];

}
```

This results in around twenty-four lines of assembly being generated when optimizations are switched on in Visual Studio. The C++ compiler unrolls the entire loop and uses eight **ADD** instructions. Unrolling loops is a good optimization technique, and modern compilers are experts at exactly this kind of optimization.

However, using even the most basic SIMD instructions (MMX is used in this example), the same addition of 8-byte arrays can be accomplished in just four instructions.

```
movq mm0, qword ptr arr1   ; Load 8 bytes from arr1 into MM0

paddb mm0, qword ptr arr2  ; Add the 8 bytes from arr2

movq qword ptr arr1, mm0   ; Store the answer back in arr1

emms                       ; Close multimedia state
```

The most important thing to note is that there is no loop in the SIMD version. All eight iterations of the loop can be carried out in a single instruction (**PADDB**, the add packed bytes instruction). The addition instruction is a single step for the CPU; it takes one or two clock cycles (depending on the CPU and pipeline issues, and assuming the arrays are in cache). MMX can perform eight operations at once, but it will not perform eight times the speed of regular scalar code. While it won't perform eight times faster, you will see a performance increase, particularly with compute-intensive functions. Compute-intensive functions are those that load data from RAM, perform many operations on the data, and then store the results.

SIMD Concepts

Saturating Arithmetic versus Wraparound Arithmetic

In the regular x86 instructions, when a result is too large to fit into the destination, the top of the result is lost and only the bits that do fit into the destination are actually written. This effectively wraps the results around at the top and bottom, such as the following:

```
mov al, 255

inc al        ; AL will wrap around to 0 since 256 is too large



mov al, 0

dec al        ; AL will wrap to 255 if unsigned and -1 if signed
```

This is called wraparound arithmetic, or modular arithmetic. It is often all that is needed, but there are some problems. Many operations on multimedia do not benefit from this type of arithmetic. For example, consider an algorithm to increase the brightness of an image by adding some value to each pixel. Some of the pixels may already be almost white, having a value near 255, 255, 255, which is a white pixel in the standard RGB24 color modes. If the extra brightness is added to these pixels, they will suddenly go very dark. Our original pixel has a value of 252 for its red component, we add 12 to the brightness to make 264, but due to wrapping round at 256, we will end up with 8. Our pixel will go from very light to very dark, making the adjust brightness algorithm appear incorrect.

For this exact reason, saturating arithmetic was incorporated into many instructions in the SIMD instruction sets. Saturating arithmetic sets a maximum and minimum value for each of the data types, and instead of wrapping around, the answer will be capped at these values. The 252 + 12 in our example would be saturated to 255.

Each data size and type (unsigned or signed) has two saturating values: one is the minimum and the other is the maximum.

Table 18: Saturating Values

Data Type	Minimum Saturate	Maximum Saturate
Unsigned Byte	0	255

Data Type	Minimum Saturate	Maximum Saturate
Signed Byte	-128	127
Unsigned Word	0	65535
Signed Word	-32768	32767
Unsigned Dword	0	4294967295
Signed Dword	-2147483648	2147483647

Packed/SIMD versus Scalar

Operations on more than one data element at once are called SIMD or packed operations. Operations on single elements, as in the x86 instructions, are called scalar. Some of the SIMD instruction sets perform scalar operations as well as SIMD.

MMX

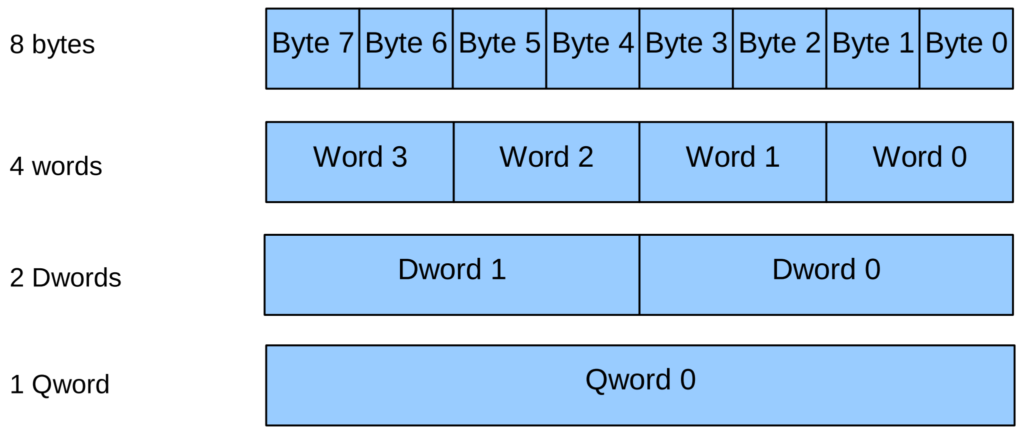
MMX was the first SIMD instruction set added to the x86 architecture. It was added by Intel in 1996 (other x86 CPU manufacturers also include these SIMD instruction sets such as AMD and VIA). At the time it was added, Intel did not wish to add new registers to the CPUs. Instead, they used the same registers as the x87 floating point unit (the MMX registers are said to be aliased to the x87 registers and vice versa). The x87 floating point unit has a dual role; it can perform its regular x87 floating point arithmetic or it can be in its multimedia mode, performing MMX instructions. The x87 unit will change to MMX mode simply by executing an MMX instruction, but once the program is finished with MMX, you have to call the EMMS (exit multimedia state) instruction to have the x87 return to its regular floating point mode.

It is important to note that the MMX instructions were also added to the SSE2 instruction set. When using these instructions with the SSE registers, they perform double the workload, working with 128-bit SSE registers instead of the 64-bit MMX registers. Most of the SSE instruction sets work with floating point, but you can also use any of the MMX instructions in conjunction with 128-bit SSE registers to perform operations on integers if the CPU is SSE2 capable. Almost all CPUs today are SSE2 capable. Windows 8 and Office 2013 require SSE2 capable CPUs, which indicates how widespread SSE2 CPUs are. Where the difference between the MMX and SSE2 versions of the instruction is important, I have included the instruction in both the MMX and the SSE reference sections.

Registers

There are eight MMX registers aliased to the x87 registers (ST0 to ST7). We will not examine x87 in this book, since SSE has made it mostly legacy. The MMX registers are named MM0, MM2, MM3, MM4, MM5, MM6, and MM7. Each register is 64 bits wide. An MMX register can be used as 8 bytes, 4 words, 2 dwords, or occasionally a single qword. The size of data that an MMX register is being used for is dictated by the instruction. The data size a particular register is holding is not fixed; you are free to change it whenever you need to. Traditionally, the registers are drawn as arrays with the first byte on the far right and the last one on the left. The bytes in each element are also an array of bits, with the least significant bits on the right of each element and the most significant bits on the left.

The MMX registers can be used in the following ways:



Most MMX instructions take two operands; the first is the destination and the second is the source. Like the x86 instructions, the destination often also acts as a source. For instance, with the add instructions, the destination becomes the destination plus the source. Since this might be misleading, I will use the terms operand 1 and 2 or parameter 1 and 2.

Referencing Memory

To reference memory in an MMX instruction, use the following:

```
movq mm5, qword ptr [A]
```

A is an array of at least 64 bits. You cannot use **A** by itself (without the pointer prefix); MMX needs to know there is a qword of data there. You can also use the x86 complex addressing modes.

```
mov mm0, qword ptr [rax+rdx*2]

mov qword ptr [rdx], mm3
```

Most MMX (and SSE as well) instructions do not allow for the first operand to be memory. Usually, the first operand must be an MMX register and the second can be either a MMX register or a memory operand.

Exit Multimedia State

EMMS

The eight MMX registers physically use the floating point unit's ST(x) registers. As soon as an MMX instruction is executed, the CPU enters multimedia state, or MMX mode. To resume regular floating point use for these registers, you must call EMMS.

```
movq mm0, mm1; Call some MMX instruction to begin MMX mode

emms   ; After you are done in MMX, call emms to restore floating point mode
```

Note: Almost all floating point arithmetic is performed by using the SSE scalar instructions instead of the rather lonesome and increasingly neglected x87 unit. In x64, not calling EMMS will usually not cause a problem. EMMS is always a good idea though, since it may be difficult to track the bugs that this type of heinous neglect of good practices would instantiate.

Do not call EMMS in the middle of a loop, since (particularly on older Intel CPUs) the instruction is quite slow to execute. It is better to do large chunks of MMX processing together and call this instruction only once all MMX processing is complete.

Moving Data into MMX Registers

There are two data movement instructions in MMX; one moves 64 bits of data and the other moves 32 bits. When moving 32 bits, if the destination operand is an MMX register, the top is cleared to 0 and the data is only moved into the bottom half.

Move Quad-Word

MOVQ [mmx], [mmx/mem64]

MOVQ [mmx/mem64], [mmx]

This instruction copies 64 bits of data from the second operand into the first.

Move Dword

MOVD [mmx], [mmx/mem32/reg32]

MOVD [mmx/mem32/re32], [mmx]

This instruction moves 32 bits from the second operand into the bottom of the MMX register first operand; it can also be used to move 32 bits of data from a general purpose register or memory location to an MMX register.

Note: When the first operand is an MMX register, this instruction clears the top 32 bits to 0.

```
mov eax, 12
mov mm0, eax      ; MM0 would have the following dwords: [0, 12]
```

Boolean Instructions

```
PAND [mmx], [mmx/mem64]

POR [mmx], [mmx/mem64]

PXOR [mmx], [mmx/mem64]

PANDN [mmx], [mmx/mem64]
```

These instructions apply the Boolean operation between the bits of the two operands and store the result in the first operand.

Packed **AND NOT** (**PANDN**) has no x86 equivalent; it performs a bitwise **AND** with the source and the inverse (bitwise **NOT**) of the destination (first operand). The first operand is complemented, then has the **AND** performed with the second operand. For truth tables of the other bitwise instructions, please consult the entries for **OR**, **AND**, and **XOR** in the x86 instruction set.

Table 19: Truth Table for PANDN

Operand 1	Operand 2	Result in Operand 1
0	0	0
0	1	1
1	0	0
1	1	0

Note: To clear an MMX register to 0, you can use PXOR with the register to clear as both operands.

```
pxor mm0, mm0     ; Clear MM0 to 0

pandn mm0, mm0    ; The new ANDN instruction will clear a register to 0
```

There is no difference between the data types when using the Boolean instructions. Using **XOR** on data in words produces exactly the same result as using **XOR** on data in bytes. Unlike many other instructions, the Boolean instructions do not have any specified data types.

Shifting Bits

```
PSLLW [mmx], [mmx/mem64/imm8] ; Left, Logical, Words

PSLLD [mmx], [mmx/mem64/imm8] ; Left, Logical, dwords

PSLLQ [mmx], [mmx/mem64/imm8] ; Left, Logical, qword

PSRLW [mmx], [mmx/mem64/imm8] ; Right, Logical, Words
```

PSRLD [mmx], [mmx/mem64/imm8] ; Right, Logical, dwords

PSRLQ [mmx], [mmx/mem64/imm8] ; Right, Logical, qword

PSRAW [mmx], [mmx/mem64/imm8] ; Right, Arithmetic, Words

PSRAD [mmx], [mmx/mem64/imm8] ; Right, Arithmetic, dwords

These instructions shift the elements in the destination right or left by the number of bits specified in the second operand. Left shifting effectively multiplies data by a power of two, while right shifting shifts the bits right, and divides by a power of two.

Logical shifts move 0 into the vacant spots, while the arithmetic right shifts duplicate the sign bit and can be used for signed division of integers by powers of two.

Note: If the second operand is a 64-bit memory location or an MMX register, it is not read SIMD style. It is read as a single 64-bit number. You cannot shift the individual elements in an MMX register by different amounts; they are all shifted the same number of bits.

The shifting instructions do not wrap data around, so you can clear a register to 0 by shifting left more bits than the data size, and shifting left 64 bits or more clears the register to 0.

You can also copy the sign of elements across the entire element by using the arithmetic right shifts and a value greater or equal to the data element size.

```
psraw mm0, 16; Sets each of MM0's words to 1s if negative, otherwise 0
```

Arithmetic Instructions

```
PADDB [mmx], [mmx/mem64]      ; Add unsigned/signed bytes, wrap around

PADDSB [mmx], [mmx/mem64]     ; Add signed bytes, saturate

PADDUSB [mmx], [mmx/mem64]    ; Add unsigned bytes, saturate

PADDW [mmx], [mmx/mem64]      ; Add unsigned/signed words, wrap around

PADDSW [mmx], [mmx/mem64]     ; Add signed words, saturate

PADDUSW [mmx], [mmx/mem64]    ; Add unsigned words, saturate

PADDD [mmx], [mmx/mem64]      ; Add unsigned/signed double-words, wrap
around

PSUBB [mmx], [mmx/mem64]      ; Subtract signed/unsigned bytes, wrap
around

PSUBSB [mmx], [mmx/mem64]     ; Subtract signed words, saturate

PSUBUSB [mmx], [mmx/mem64]    ; Subtract unsigned bytes, saturate
```

```
PSUBW [mmx], [mmx/mem64]      ; Subtract unsigned/signed words, wrap
around

PSUBSW [mmx], [mmx/mem64]     ; Subtract signed words, saturate

PSUBUSW [mmx], [mmx/mem64]    ; Subtract unsigned words, saturate

PSUBD [mmx], [mmx/mem64]      ; Subtract signed/unsigned doubles
```

These instructions add and subtract bytes, words, or dwords. Each element in the second operand is subtracted from or added to the corresponding element in the first operand, and the results from the packed additions and subtractions are stored in the first operand. These instructions have the option to use wrap around or saturation arithmetic. There are no saturation or unsigned instructions that operate on dwords in MMX; if this functionality is required, then use SSE.

There is an instruction for each data type where it does not matter if the data type is signed or unsigned, since the result is the same. **PADDB**, **PADDW**, **PADDD**, **PSUBB**, **PSUBW**, and **PSUBD** all give the same answer if the data is signed or unsigned. None of them use saturation.

Note: When you need to detect what elements overflowed in SIMD operations, you can perform both the saturating version of the operation and the wrap around version, and compare the results. Any elements that are not identical in both answers have overflowed. This is useful because there is no carry flag or overflow available in SIMD.

The instruction **PADDW mm0, mm1** is illustrated as follows:

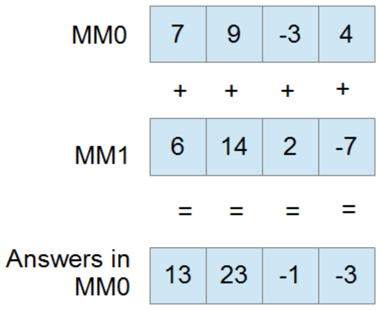


Figure 16

The next example is of **PSUBW mm0, mm1**. Note the second operand is subtracted from the first.

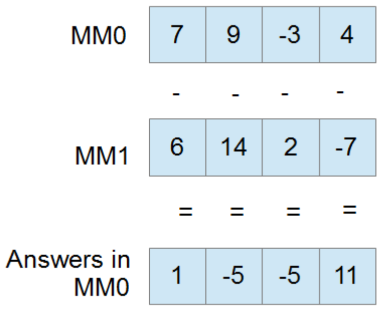


Figure 17

Multiplication

```
PMULHW [mmx], [mmx/mem64]     ; Multiply and keep high 16 bits of 32 bit
result

PMULLW [mmx], [mmx/mem64]     ; Multiply and keep low 16 bits of 32 bit
result

PMADDWD [mm0], [mmx/mem64]    ; Multiply and add words to double-words
```

You can multiply two MMX registers or an MMX register with a memory location, but all these instructions only operate on signed words. For multiplication of other integer data types, use SSE2. When multiplying, you can choose to keep either the top 16 bits (**`PMULHW`**) or the bottom 16 bits (**`PMULLW`**) of the multiplication.

There is also a fused multiply add instruction (**`PMADDWD`**, assumed words are signed) that allows you to multiply and add in a single operation with added precision. Each word in the destination is multiplied by the corresponding word in the destination to give four results: A, B, C, and D. A is then added to B, C is added to D, and both these answers form the final two dwords in the destination. The initial multiplication is able to result in 32 bits since a larger temporary register is used.

Note: If one of your operands for the PMADDWD is { 1, 1, 1, 1 } (four words all set to 1), the operation performs a horizontal add. It adds each adjacent pair of words in the destination and stores the two results as two dwords.

Note: If every second word in both operands to a PMADDWD is 0, then the instruction will result in two word multiplication, but the entire 32-bit result will be kept.

Comparisons

```
pcmpeqb [mmx], [mmx/mem64]    ; Compare bytes for equality

pcmpgtb [mmx], [mmx/mem64]    ; Compare signed bytes for greater than

pcmpeqw [mmx], [mmx/mem64]    ; Compare words for equality

pcmpgtw [mmx], [mmx/mem64]    ; Compare signed words for greater than

pcmpeqd [mmx], [mmx/mem64]    ; Compare double-words for equality

pcmpgtd [mmx], [mmx/mem64]    ; Compare signed double-words for greater
than
```

MMX has a comparison instruction for testing equality and greater-than only. All other comparisons must be built from these and the Boolean instructions.

Comparisons in MMX (and the other SIMD instruction sets) result in setting all of the bits of each element to either 1 or 0. Elements are set to a mask of all 1s where they pass the condition and all 0s where they do not. All “greater than” comparisons are signed in MMX.

For instance, if mm0 has { 9, 14, 21, 40 } as four words, and MM1 has { 9, 4, 21, 4 }, then the following code will result in MM0 having { ffffh, 0000h, ffffh, 0000h }:

```
pcmpeqw mm0, mm1        ; Packed compare words for equality
```

This is because there are two words that are the same in MM0 and MM1, and two that are not the same. MM0 effectively becomes a mask of words that are the same between MM0 and MM1.

Note: The PCMPEQB instruction (and the other equality comparison instructions) can be used to set an MMX register to all 1s or ffffffffffffffffh by using the same register as both operands.

```
pcmpeqb mm0, mm0        ; Set MM0 to all 1s
```

Creating the Remaining Comparison Operators

There are a number of ways to create the other comparisons (`!=`, `>=`, etc.). To perform the not-equal operation (`!=`) between MM0 and MM1 using MM7 as a temporary register, use the following:

```
pcmpeqb mm0, mm1

pcmpeqb mm7, mm7

pxor mm0, mm7
```

To perform the greater than or equal to operation (`>=`) between MM0 and MM1 using MM7 as a temporary register, use the following:

```
movq mm7, mm0    ; Backup parameter to mm7

pcmpeqb mm0, mm1 ; Find where they are equal

pcmpgb mm7, mm1  ; Find where they are greater

por mm0, mm7     ; OR these two results
```

To perform the less-than operation (`<`) between MM0 and MM1 using MM7 as a temporary register, use the following:

```
movq mm7, mm0             ; Backup parameter to mm7

pcmpeqd mm0, mm1          ; Does mm0 = mm1 ?

pcmpgd mm7, mm1           ; Is mm7 > mm1 ?

por mm0, mm7              ; mm0 is now >= mm1

pcmpeqd mm7, mm7          ; Get mm7 ready to complement mm0

pxor mm0, mm7             ; mm0 is now < mm1
```

Packing

```
PACKSSDW [mmx], [mmx/mem64]   ; Pack signed double-words to words and
saturate

PACKSSWB [mmx], [mmx/mem64]   ; Pack signed words to bytes and saturate

PACKUSWB [mmx], [mmx/mem64]   ; Pack unsigned words to bytes and saturate
```

Packing instructions convert large data types into smaller ones, and then take the elements of two operands and resize them so that they fit into a single operand. You can convert dwords to words or words to bytes. You can use signed or unsigned words when converting bytes. All the MMX packing instructions use saturation.

`PACKSSDW` converts dwords to words by saturating the dwords and storing the two converted dwords from the first operand in the lower half of the answer, and storing the two converted words from the second operand in the upper half of the answer.

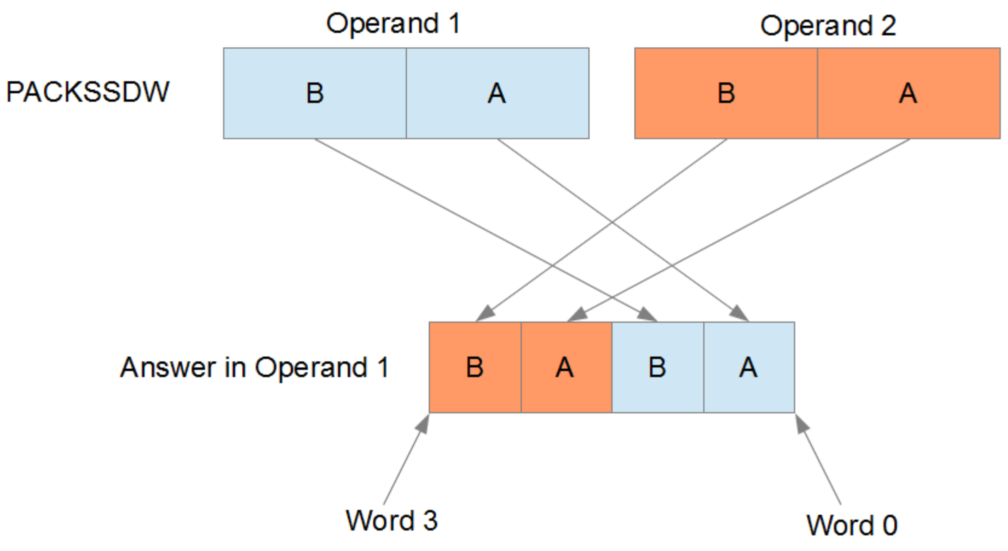


Figure 18

`PACKSSWB` and **`PACKUSWB`** convert words to bytes by first applying saturation. The **`PACKSSWB`** range is -128 to 127, and the **`PACKUSWB`** range is 0 to 255. The four words from the first operand are converted to bytes and stored in the lower half of the answer, and the four words from the second operand are converted to bytes and stored in the upper half of the answer.

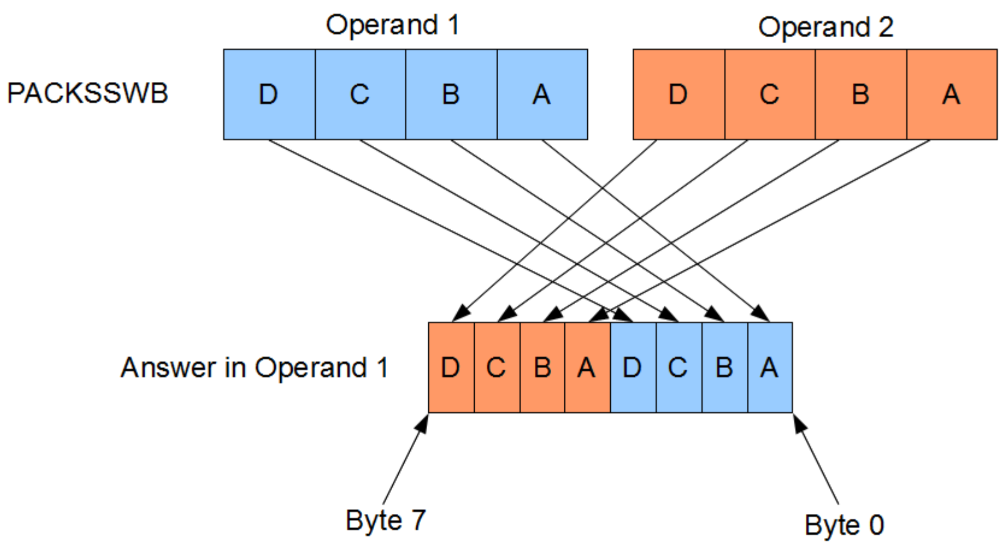


Figure 19

Unpacking

```
PUNPCKLBW [mmx], [mmx/mem64]  ; Unpack low bytes to words

PUNPCKHBW [mmx], [mmx/mem64]  ; Unpack high bytes to words and interleave

PUNPCKLWD [mmx], [mmx/mem64]  ; Unpack low words to double-words

PUNPCKHWD [mmx], [mmx/mem64]  ; Unpack high words to double-words

PUNPCKLDQ [mmx], [mmx/mem64]  ; Unpack low double-words to quad-words

PUNPCKHDQ [mmx], [mmx/mem64]  ; Unpack high dwords to qword and interleave
```

The unpacking instructions convert from smaller data types to larger ones. They are **MOV** instructions, as there is actually no data conversion at all. For example, converting the eight bytes in the source to eight words would require more space than can fit in an MMX register, so you must choose to convert either the lower or upper half.

The conversion is performed using what is called interleaving. Elements from the first operand are interleaved with those from the second operand.

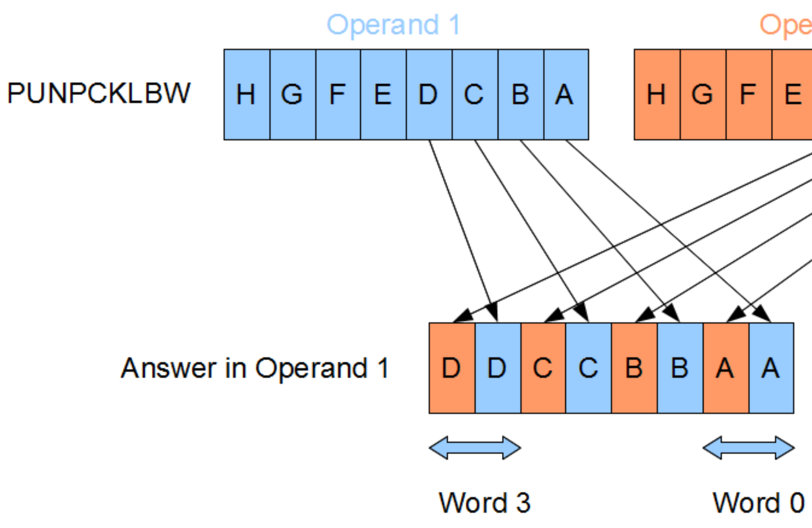


Figure 20

The interleaving system was invented to allow the greatest flexibility while adding the fewest number of instructions to the CPU. If the second operand contains all zeros, then these instructions have the effect of zero-extending (unsigned extending) the data in the first operand.

Initially, these instructions were also used to broadcast values (duplicate elements across the whole register). However, the SSE shuffle instructions make broadcasting a lot easier.

Some examples of using the packing and unpacking instructions are as follows:

```
; To duplicate the bottom 32 bits of mm0 into the top:

punpckldq mm0, mm0



; To fill MM0 with 8 copies of a byte from AL (9 in this example):

xor eax, eax               ; Make sure the top of EAX is 0

mov al, 9                  ; Copy the byte into eax, 9 in this example

movd mm0, eax              ; Copy eax to the low dword of mm0

punpckldq mm0, mm0         ; Duplicate the bottom into the topof MM0

packssdw mm0, mm0          ; Copy these two 9s into 4

packuswb mm0, mm0         ; Copy those four 9s into 8



; Sign extending: If the second operand contains the signs of the

; elements in the first operand (use the PSRAW instruction to get this)

; then these instructions have the effect of sign extending the data
; in the first operand:

movq mm1, mm0       ; Copy words

psraw mm1, 16       ; Fill mm1 with signs of elements in mm0

punpcklwd mm0, mm1  ; Unpack interleaving with signs, sign extend to dwords!
```

SSE Instruction Sets

Introduction

The SSE (Streaming SIMD Extensions) instruction sets perform SIMD operation on new 128-bit registers. Over the years, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, and SSE4a instruction sets have been created. The original SSE instruction set's purpose was to allow SIMD floating point instructions and the ability to work on four single-precision floating point values at once. The additional instruction sets added a multitude of instructions for doing almost anything in both SIMD and scalar. For instance, the SSE2 instruction set added integer instructions as well as many double-precision floating point instructions.

Where MMX registers were aliased to the x87 unit's registers, the SSE registers occupy a new register space on the CPU. There is no need to switch from multimedia mode to floating point when using SSE. Some of the SSE instructions do use the MMX registers in conjunction with the SSE registers, and these instructions still cause a switch in the x87 unit. Just as in the MMX instruction set, whenever an SSE instruction references the MMX registers, **`EMMS`** must be called to restore the x87 unit to floating point mode.

Originally there were eight SSE registers named from XMM0 to XMM7. In x64 applications, this has been increased to sixteen registers named from XMM0 to XMM15. Each SSE register is 128 bits wide and can be used for the data sizes shown in Figure 21.

In the instruction listings that follow, I have included the instruction sets (beside the mnemonics). This is where the instructions come from. Be sure to check that the CPU is capable of an instruction with the **`CPUID`** Function 1 prior to using it.

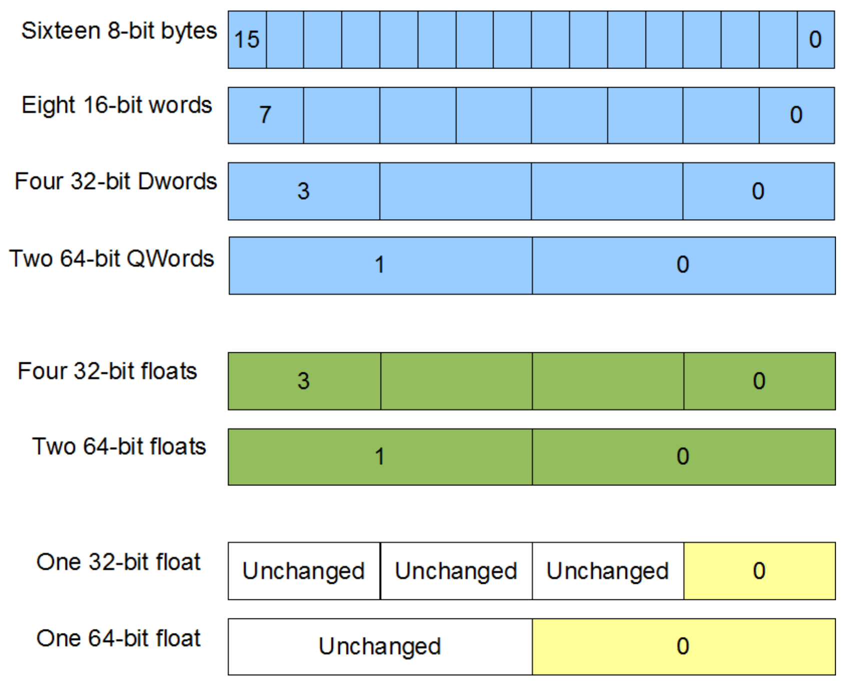


Figure 21

Collectively, the SSE instruction sets present a programmer with a staggering array of instructions numbering in the hundreds. Many instructions have scalar as well as SIMD versions.

Note: The Microsoft compiler users scalar SSE instructions to perform floating point operations, instead of using the x87 FPU.

In x64, the C calling convention used by the C++ compiler passes floating point parameters using the XMM0 register. Only the lowest single or double precision element of XMM0 is used to return values, but the lowest elements of XMM0, XMM1, XMM2, and XMM3 are used to pass the first four floating point values to functions.

Data alignment is very important for many of the SSE instructions that reference memory. Data must be aligned to 16 bytes where memory is read or written, or else a segmentation fault will occur.

AVX

Included in this reference are some of the new AVX instructions. These instructions are only available on the new CPUs. The AVX instruction set is the largest addition of instructions since the original SSE in 1999. The instruction set consists of new AVX versions of many of the SSE instructions and new 256-bit SIMD registers.

Unlike the earlier instruction sets, AVX requires the support of the operating system. Windows 7 does not support AVX by default and must be upgraded by installing the Service Pack 1 (SP1). AVX is natively supported by Windows 8 and later.

There are sixteen AVX registers named from YMM0 to YMM15. They are all 256 bits wide and are aliased to the sixteen SSE registers, so the low 128 bits of YMM0 is the SSE register XMM0.

Each AVX register can be broken up exactly as the SSE registers, only there are twice the number of elements available in the AVX registers. The AVX instructions begin with "V" and the mnemonic is otherwise similar to the SSE versions of the instructions.

In addition to larger registers, the new AVX instructions often offer nondestructive versions of many instructions.

```
ADDPD XMM0, XMM2        ; Destructive, XMM0 is overwritten

VADDPD XMM0, XMM2, XMM7 ; Non destructive, XMM0 = XMM2 + XMM7
```

This instruction adds corresponding packed doubles in a similar way to the SSE version; only operands 2 and 3 are added and the answers are stored in operand 1. This allows for the destination to be a different operand to both sources.

Data Moving Instructions

Move Aligned Packed Doubles/Singles

```
MOVAPD [xmm/mem128], [xmm/mem128]  - SSE2

VMOVAPD [xmm/mem128], [xmm/mem128] - AVX

VMOVAPD [ymm/mem256], [ymm/mem256] - AVX

MOVAPS [xmm/mem128], [xmm/mem128]  - SSE2

VMOVAPS [xmm/mem128], [xmm/mem128] - AVX

VMOVAPS [ymm/mem256], [ymm/mem256] - AVX
```

The move aligned instructions move 128 bits or 256 bits of data from the second operand into the first. If either of the operands is a memory location, then it must be aligned to 16 bytes.

Data can be aligned in C++ to 16 bytes using the **_declspec(align(16))** directive prior to the data type of the variable in its declaration.

Data can be aligned in the .data segment in assembly by using **align 16** on the line prior to the declaration of the variable.

The CPU performs operations faster on aligned data, although many instructions in SSE and AVX require aligned data or else they will generate a segmentation fault.

Move Unaligned Packed Doubles/Singles

```
MOVUPD [xmm/mem128], [xmm/mem128]  - SSE2

VMOVUPD [xmm/mem128], [xmm/mem128] - AVX

VMOVUPD [ymm/mem256], [ymm/mem256] - AVX

MOVUPS [xmm/mem128], [xmm/mem128]  - SSE
```

VMOVUPS [xmm/mem128], [xmm/mem128] - AVX

VMOVUPS [ymm/mem256], [ymm/mem256] - AVX

The move unaligned packed doubles and singles instructions move 128 bits or 256 bits of data from the second operand into the first. Unlike the aligned move instructions, if one of the operands is a memory operand, then it need not be aligned to 16 bytes.

Arithmetic Instructions

Arithmetic on integer types can be performed using the SSE registers and the same instruction mnemonics as MMX instructions. The MMX instruction mnemonics can be used with the SSE registers only if the SSE2 instruction set is available.

Adding Floating Point Values

Table 20

Mnemonic	Meaning	Operands	Instruction Set
ADDPD	Add packed doubles	[xmm], [xmm/mem128]	SSE2
VADDPD	Add packed doubles	[xmm], [xmm], [xmm/mem128]	AVX
VADDPD	Add packed doubles	[ymm], [ymm], [ymm/mem256]	AVX
ADDPS	Add packed singles	[xmm], [xmm/mem128]	SSE
VADDPS	Add packed singles	[xmm]/[xmm], [xmm/mem128]	AVX
VADDPS	Add packed singles	[ymm]/[ymm], [ymm/mem256]	AVX
ADDSD	Add scalar double	[xmm], [xmm/mem64]	SSE2
VADDSD	Add scalar double	[xmm], [xmm], [xmm/mem64]	AVX
ADDSS	Add scalar single	[xmm], [xmm/mem32]	SSE
VADDSS	Add scalar single	[xmm], [xmm], [xmm/mem32]	AVX

These instructions are used to add elements from one SSE or AVX register to another. The two-operand SSE and AVX versions add elements from the second operand to the corresponding elements in the first, and store the answers in the first operand. The three-operand AVX versions add the elements in the second operand and third operand together, and then store the answers in the first operand.

The following example illustrates the way the add instructions operate. Here, the **ADDPS** (add packed singles) instruction is used to add the values in XMM1 to those in XMM0:

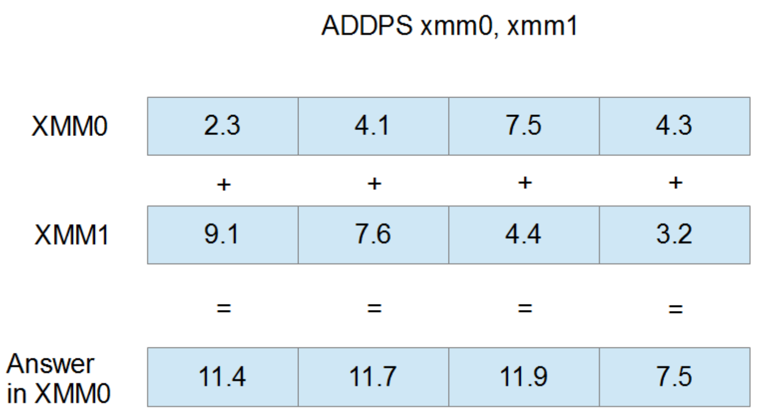


Figure 22

Subtracting Floating Point Values

Table 21

Mnemonic	Meaning	Operands	Instruction Set
SUBPD	Subtract packed doubles	[xmm], [xmm/mem128]	SSE2
VSUBPD	Subtract packed doubles	[xmm], [xmm], [xmm/mem128]	AVX
VSUBPD	Subtract packed doubles	[ymm], [ymm], [ymm/mem256]	AVX
SUBPS	Subtract packed singles	[xmm], [xmm/mem128]	SSE
VSUBPS	Subtract packed singles	[xmm], [xmm], [xmm/mem128]	AVX
VSUBPS	Subtract packed singles	[ymm], [ymm], [ymm/mem256]	AVX
SUBSD	Subtract scalar double	[xmm], [xmm/mem64]	SSE2
VSUBSD	Subtract scalar double	[xmm], [xmm], [xmm/mem64]	AVX
SUBSS	Subtract scalar single	[xmm], [xmm/mem32]	SSE
VSUBSS	Subtract scalar single	[xmm], [xmm], [xmm/mem32]	AVX

The subtraction instructions subtract elements in one register or memory from the corresponding elements in another register. The two-operand versions of the instructions subtract the elements of the second operand from the corresponding elements in the first, and store the answer in the first operand. The three-operand AVX versions subtract the elements of the third operand from those in the second, and store the result in the first operand.

The following example illustrates a **SUBPS** instruction using XMM0 and XMM1 as operands. The four single-precision floats in XMM1 are subtracted from those in XMM0, and the result is placed into XMM0.

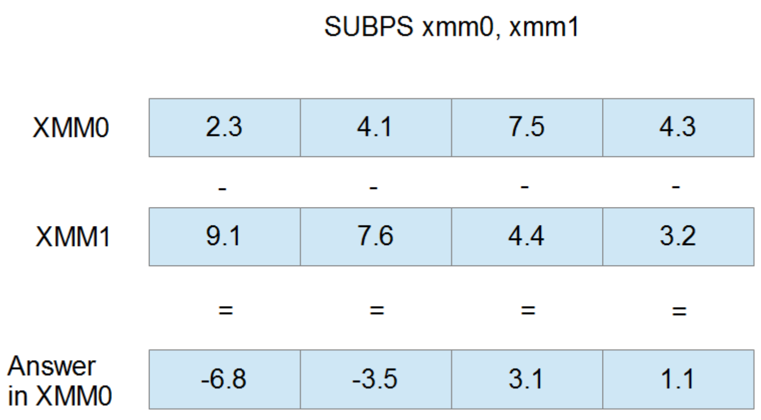


Figure 23

Dividing Floating Point Values

Table 22

Mnemonic	Meaning	Operands	Instruction Set
DIVPD	Divide packed doubles	[xmm], [xmm/mem128]	SSE2
VDIVPD	Divide packed doubles	[xmm], [xmm], [xmm/mem128]	AVX
VDIVPD	Divide packed doubles	[ymm], [ymm], [ymm/mem256]	AVX
DIVPS	Divide packed singles	[xmm], [xmm/mem128]	SSE
VDIVPS	Divide packed singles	[xmm], [xmm], [xmm/mem128]	AVX
VDIVPS	Divide packed singles	[ymm], [ymm], [ymm/mem256]	AVX
DIVSD	Divide scalar double	[xmm], [xmm/mem64]	SSE2
VDIVSD	Divide scalar double	[xmm], [xmm], [xmm/mem64]	AVX
DIVSS	Divide scalar single	[xmm], [xmm/mem32]	SSE
VDIVSS	Divide scalar single	[xmm], [xmm], [xmm/mem32]	AVX

The division instructions divide elements in one register or memory by the corresponding elements in another. The two-operand versions divide the values in the first operand by the corresponding values in the second, and store the results in the first. The three-operand versions divide the elements in the second operand by those in the third and store the results in the first operand.

The sample illustration is of the `DIVPD` instruction with XMM0 and XMM1 as operands. The elements in XMM0 are divided by those in XMM1 and the resulting doubles are stored in XMM0.

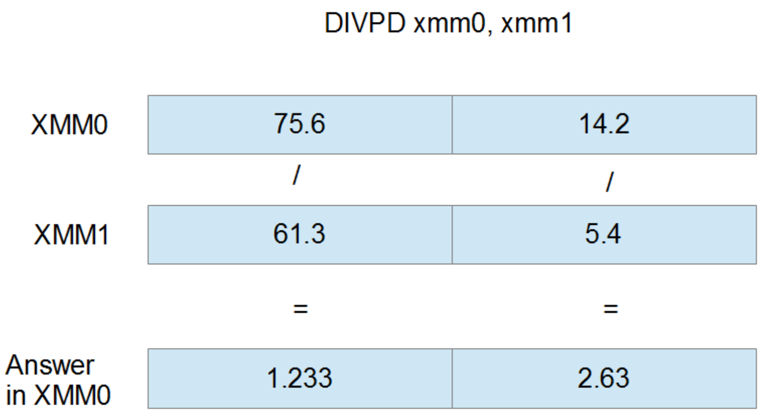


Figure 24

Multiplying Floating Point Values

Table 23

Mnemonic	Meaning	Operands	Instruction Set
MULPD	Multiply packed doubles	[xmm], [xmm/mem128]	SSE2
VMULPD	Multiply packed doubles	[xmm], [xmm], [xmm/mem128]	AVX
VMULPD	Multiply packed doubles	[ymm], [ymm], [ymm/mem256]	AVX
MULPS	Multiply packed singles	[xmm], [xmm/mem128]	SSE
VMULPS	Multiply packed singles	[xmm], [xmm], [xmm/mem128]	AVX
VMULPS	Multiply packed singles	[ymm], [ymm], [ymm/mem256]	AVX
MULSD	Multiply scalar double	[xmm], [xmm/mem64]	SSE2
VMULSD	Multiply scalar double	[xmm], [xmm], [xmm/mem64]	AVX
MULSS	Multiply scalar single	[xmm], [xmm/mem32]	SSE
VMULSS	Multiply scalar single	[xmm], [xmm], [xmm/mem32]	AVX

The multiplication instructions multiply the elements in one register by the corresponding elements in another register or memory. The two-operand versions multiply the values in the first operand by those in the second and store the results in the first operand. The three-operand versions multiply the values in the third operand by those in the second, and store the results in the first operand.

The following figure is the **`MULPD`** instruction using XMM0 and XMM1 as operands. The doubles in XMM0 are multiplied by those in XMM1 and the result is stored in XMM0.

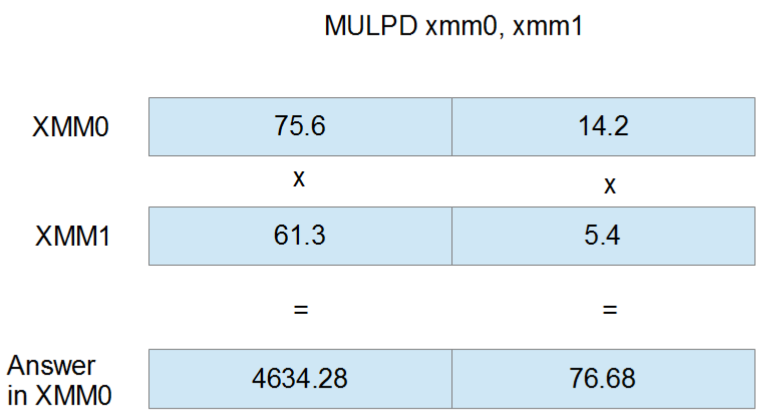


Figure 25

Square Root of Floating Point Values

Table 24

Mnemonic	Meaning	Operands	Instruction Set
SQRTPD	Square root packed doubles	[xmm], [xmm/mem128]	SSE2
VSQRTPD	Square root packed doubles	[xmm], [xmm/mem128]	AVX
VSQRTPD	Square root packed doubles	[ymm], [ymm/mem256]	AVX
SQRTPS	Square root packed singles	[xmm], [xmm/mem128]	SEE
VSQRTPS	Square root packed singles	[xmm], [xmm/mem128]	AVX
VSQRTPS	Square root packed singles	[ymm], [ymm/mem256]	AVX
SQRTSD	Square root scalar double	[xmm], [xmm/mem64]	SSE2
VSQRTSD	Square root scalar double	[xmm], [xmm/mem64]	AVX
SQRTSS	Square root scalar single	[xmm], [xmm/mem32]	SSE
VSQRTSS	Square root scalar single	[xmm], [xmm/mem32]	AVX

The square root instructions calculate the square root of the elements in the second operand and store the answers in the first operand.

The following figure is the **`SQRTPD`** instruction using the registers XMM0 and XMM1 as operands. The first operand (XMM0) is ignored in the calculation; its elements have been grayed out. The final results in the elements of XMM0 are the square root of the doubles in XMM1.

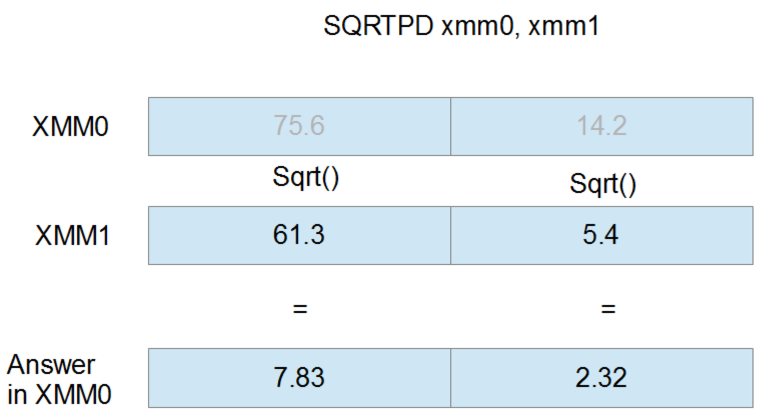


Figure 26

Reciprocal of Single-Precision Floats

Table 25

Mnemonic	Meaning	Operands	Instruction Set
RCPPS	Reciprocal packed singles	[xmm], pxmm/mem128]	SSE
VRCPPS	Reciprocal packed singles	[xmm], [xmm/mem128]	AVX
VRCPPS	Reciprocal packed singles	[ymm], [ymm/mem128]	AVX
RCPSS	Reciprocal scalar single	[xmm], [xmm/mem32]	SSE
VRCPSS	Reciprocal scalar single	[xmm], [xmm/mem32]	AVX
VRCPSS	Reciprocal scalar single	[ymm], [ymm/mem32]	AVX

The reciprocal instructions calculate the reciprocal (1/*x*, where *x* is the element) of the elements in the second operand and store the result in the elements of the first operand. The elements of the first operand are ignored for the calculation. The result of dividing by zero in these instructions is infinity. These instructions only give a quick approximation of the real reciprocal; they are intended to be used when exact precision is not required.

The following figure shows the **RCPPS** instruction with XMM0 and XMM1 as operands. The initial values in XMM0 are ignored by the instruction and are overwritten by the reciprocal of the elements in XMM1. They are grayed out in Figure 27.

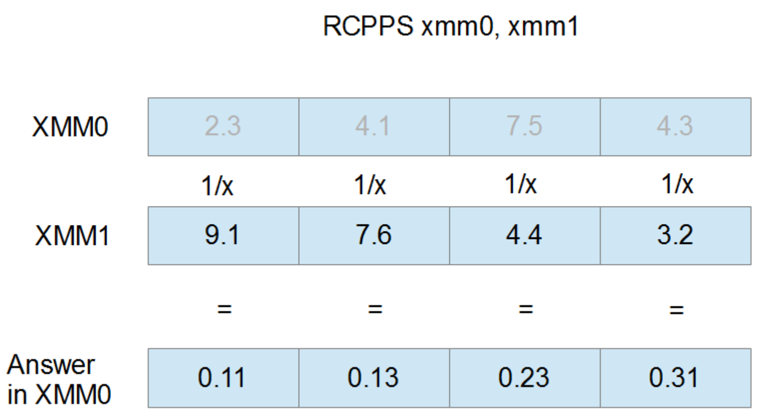


Figure 27

Reciprocal of Square Root of Single-Precision Floats

RSQRTPS [xmm], [xmm/mem128] – SSE

VRSQRTPS [xmm], [xmm/mem128] – AVX

VRSQRTPS [ymm], [ymm/mem256] – AVX

RSQRTSS [xmm], [xmm/mem32] – SSE

VRSQRTSS [xmm], [xmm], [xmm/mem32] – AVX

These instructions calculate the reciprocal of the square root (1/sqrt(*x*) or sqrt(*x*)/*x*, where *x* is the element) of the elements in the second operand and store the results in the first operand. In other words, they divide one by the square root of the elements in the second operand and store the results in the first operand. The answers are not precise and the instruction is intended for use only when a quick approximation is required.

The following figure shows the **RSQRTPS** instruction using XMM0 and XMM1 as operands. The values in XMM0 are ignored for the calculation and have been grayed out. The resulting reciprocal square roots are only those from the second operand.

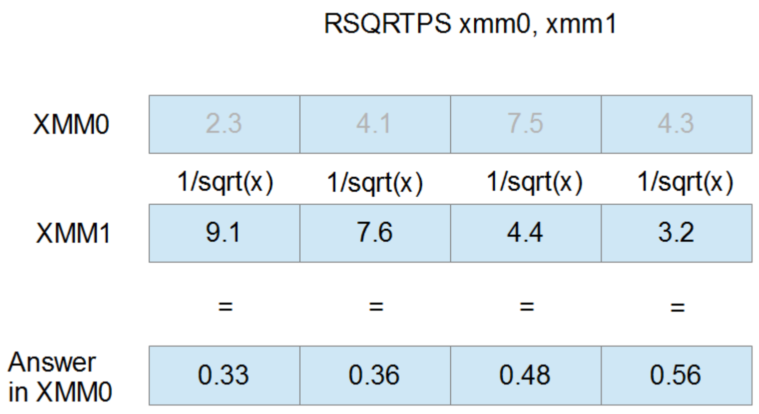


Figure 28

Boolean Operations

All of these operations essentially do exactly the same thing despite the data types in the registers, since a bitwise operation on packed doubles has the same effect on a SIMD register as a bitwise operation on packed singles. Some CPUs suffer a minor performance penalty when data in SIMD registers is not treated as the same size. For this reason, it is safest to use the bitwise or Boolean instructions designed for the particular data type you are working with.

AND NOT Packed Doubles/Singles

ANDNPD [xmm], [xmm/mem128] – SSE2

PANDN [xmm], [xmm/mem128] – SSE2

VANDNPD [xmm], [xmm], [xmm/mem128] - AVX

VANDNPD [ymm], [ymm], [ymm/mem256] – AVX

ANDNPS [xmm], [xmm/mem128] – SSE2

VANDNPS [xmm], [xmm], [xmm/mem128] – AVX

VPANDN [xmm], [xmm], [xmm/mem128] - AVX

VANDNPS [ymm], [ymm], [ymm/mem256] – AVX

The **AND NOT** instructions first complement an operand, then perform a bitwise **AND** between two operands. They store the result in the destination (see the MMX Boolean instructions for the truth table for this instruction). For the two-operand versions of these instructions, the first operand is complemented and then a bitwise **AND** is performed between the bits of both operands. The result is stored in the first operand.

For the three-operand versions of these instructions, the second operand is complemented and then a bitwise **AND** is performed between the second and third operands. The result is stored in the first operand.

This instruction is useful for creating bit masks that represent negative comparisons.

AND Packed Doubles/Singles

ANDPD [xmm], [xmm/mem128] – SSE2

PAND [xmm], [xmm/mem128] – SSE2

VANDPD [xmm], [xmm], [xmm/mem128] - AVX

VPAND [xmm], [xmm], [xmm/mem128] - AVX

VANDPD [ymm], [ymm], [ymm/mem256] – AVX

ANDPS [xmm], [xmm/mem128] – SSE

VANDPS [xmm], [xmm], [xmm/mem128] - AVX

VANDPS [ymm], [ymm], [ymm/mem256] – AVX

These instructions perform a bitwise **AND** between two operands. The two-operand versions of this instruction perform a bitwise **AND** between the first and second operands, storing the result in the first operand. The three-operand versions perform a bitwise **AND** between the second and third operand and store the results in the first operand.

OR Packed Doubles/Singles

ORPD [xmm], [xmm/mem128] – SSE2

POR [xmm], [xmm/mem128] – SSE2

VORPD [xmm], [xmm], [xmm/mem128] - AVX

VPOR [xmm], [xmm], [xmm/mem128] – AVX

VORPD [ymm], [ymm], [ymm/mem256] – AVX

ORPS [xmm], [xmm/mem128] – SSE

VORPS [xmm], [xmm], [xmm/mem128] - AVX

VORPS [ymm], [ymm], [ymm/mem256] – AVX

The **OR** instructions perform a bitwise **OR** between two operands. The two-operand versions perform a bitwise **OR** between the first and second operand and store the results in the first operand. The three-operand AVX versions perform a bitwise **OR** between the second and third operands, storing the results in the first operand.

XOR Packed Doubles/Singles

XORPD [xmm], [xmm/mem128] – SSE2

PXOR [xmm], [xmm/mem128] – SSE2

VXORPD [xmm], [xmm], [xmm/mem128] - AVX

VXORPD [ymm], [ymm], [ymm/mem256] – AVX

VPXOR [xmm], [xmm], [xmm/mem128] – AVX

XORPS [xmm], [xmm/mem128] – SSE

VXORPS [xmm], [xmm], [xmm/mem128] - AVX

VXORPS [ymm], [ymm], [ymm/mem256] – AVX

The **XOR** instructions perform a bitwise **XOR** operation between two operands. The two-operand versions of these instructions perform a bitwise **XOR** between the first and second operands and store the results in the first operand. The three-operand versions perform a bitwise **XOR** between the second and third operands and store the result in the first operand.

Comparison Instructions

Comparing Packed Doubles and Singles

CMPxxPtt [xmm], [xmm/mem128]	**- SSE and SSE2 versions**
VCMPxxPtt [xmm], [xmm], [xmm/mem128]	**- AVX versions**
VCMPxxPtt [ymm], [ymm], [ymm/mem256]	**- AVX versions**

There are many comparison instructions; the mnemonics follow the outline (**CMPxxPtt** or **VCMPxxPtt**) where the **xx** is replaced by the operator abbreviation (from the **Comparison Operators** table that follows) and the **tt** is replaced by the data type (**D** for packed double-precision floats and **S** for packed single-precision floats).

Table 26: Comparison Operators

Abbreviation	Meaning
EQ	Equal to
LT	Less than
LE	Less than or equal to
UNORD	Unordered (NaN or Undefined)
ORD	Ordered (not NaN or Undefined)
NEQ	Not equal to
NLT	Greater than or equal to, not less than
NLE	Greater than, not less or equal to

They perform the comparison operator between corresponding elements of two operands. All bits of any elements that the operator is true are set to 1. All bits of any elements where the operator is false are set to 0.

In the SSE and SSE2 versions, the comparison is performed between operands one and two and the resulting bit masks are stored in the first operand.

In the AVX versions, the comparison is performed between operands two and three, and the resulting bit masks are placed into the first operand.

The **UNORD** and **ORD** comparison operators are used to determine where various NaN (not a number) elements are. NaN values in doubles or floats are unordered and will return true if the **UNORD** comparison is used and false if **ORD** is used. All numerically orderable values (those that are not NaN or #IND) return true when the **ORD** operator is used and false when the **UNORD** operator is used.

Comparing Scalar Doubles and Singles

CMPxxStt [xmm], [xmm/mem64/mem32] - SSE and SSE2 versions

VCMPxxStt [xmm], [xmm], [xmm/mem64/mem32] - AVX versions

The scalar versions of the comparison instructions are the same as their packed counterparts, only they perform the comparison on the lowest double or single. They have an **S** for scalar in their mnemonic instead of the **P** for packed.

Comparing and Setting rFlags

COMISD [xmm], [xmm/mem64] - SSE2

VCOMISD [xmm], [xmm/mem64] - AVX

COMISS [xmm], [xmm/mem32] - SSE

VCOMISS [xmm], [xmm/mem32] – AVX

These interesting instructions bridge the gap between the SIMD instruction sets and the regular x86 instruction sets by comparing SSE or AVX registers, but setting the rFlags register. The instructions are scalar, and compare either the lowest single-precision floats (**COMISS** and **VCOMISS**) or the lowest doubles (**COMISD** and **VCOMISD**). They set the flags register in the following manner:

Table 27: x86 Flags after xCOMISxx

Condition	Zero Flag	Parity Flag	Carry Flag
NaN	1	1	1
Parameter 1 > Parameter 2	0	0	0
Parameter 1 < Parameter 2	0	0	1
Parameter 1 = Parameter 2	1	0	0

Converting Data Types/Casting

Conversion Instructions

Converting to Doubles

```
CVTDQ2PD [xmm], [xmm/mem64]   ; Converts two dwords to doubles using SSE2

VCVTDQ2PD [xmm], [xmm/mem64]  ; Converts two dwords to doubles using AVX

VCVTDQ2PD [ymm], [ymm/mem128] ; Converts two dwords to doubles using AVX

CVTPS2PD [xmm], [xmm/mem64]   ; Converts packed singles to packed doubles
using SSE2

VCVTPS2PD [xmm], [xmm/mem64]  ; Converts packed singles to packed doubles
using AVX

VCVTPS2PD [ymm], [ymm/mem128] ; Converts packed singles to packed doubles
using AVX

CVTSI2SD [xmm], [reg32/64]    ; Converts from x86 register to double using
SSE2

VCVTSI2SD [ymm], [ymm], [reg32/64]  ; Converts from x86 register to double
using AVX

CVTSS2SD [xmm], [xmm/mem64]   ; Converts a scalar single to a scalar
double using SSE2

VCVTSS2SD [ymm], [ymm], [ymm/mem64] ; Converts a scalar single to a scalar
double using AVX
```

Converting to Singles

```
CVTDQ2PS [xmm], [xmm/mem128]  ; Converts packed dwords to singles using
SSE2

VCVTDQ2PS [xmm], [xmm/mem128] ; Converts packed dwords to singles using
AVX

VCVTDQ2PS [ymm], [ymm/mem256] ; Converts packed dwords to singles using
AVX

CVTPD2PS [xmm], [xmm/mem128]  ; Converts packed doubles to singles using
SSE2

VCVTPD2PS [xmm], [xmm/mem128] ; Converts packed doubles to singles using
AVX

VCVTPD2PS [ymm], [ymm/mem256] ; Converts packed doubles to singles using
AVX

CVTSD2SS [xmm], [xmm/mem64]   ; Converts scalar double to scalar single
using SSE2

VCVTSD2SS [ymm], [ymm], [ymm/mem64] ; Converts scalar double to scalar
single using AVX

CVTSI2SS [xmm], [reg32/64]    ; Converts from x86 registers to a scalar
single using SSE2

VCVTSI2SS [ymm], [ymm], [reg32/64]  ; Converts from x86 registers to a
scalar single using AVX
```

Converting to Integers

```
CVT(T)PD2DQ [xmm], [xmm/mem128]  ; Converts packed doubles to dwords using
SSE2

VCVT(T)PD2DQ [xmm], [xmm/mem128] ; Converts packed doubles to dwords using
AVX

VCVT(T)PD2DQ [ymm], [ymm/mem256] ; Converts packed doubles to dwords using
AVX

CVT(T)PS2DQ [xmm], [xmm/mem128]  ; Converts singles to dwords using SSE2

VCVT(T)PS2DQ [xmm], [xmm/mem128] ; Converts singles to dwords using AVX

VCVT(T)PS2DQ [ymm], [ymm/mem256] ; Converts singles to dwords using AVX

CVT(T)SD2SI [reg32/64], [xmm/mem32/64]  ; Converts double to x86 register
using SSE2
```

`VCVT(T)SD2SI [reg32/64], [ymm/mem32/64] ; Converts double to x86 register using AVX`

`CVT(T)SS2SI [reg32/64], [mem32/64]      ; Converts scalar single to scalar integer using SSE2`

`VCVT(T)SS2SI [reg32/64], [mem32/64]     ; Converts scalar single to scalar integer using AVX`

The data conversion instructions convert between doubles, singles, and integer data. They convert the elements (or element) in the second operand to some other data type and store the converted results in the elements of the first operand. The conversion is analogous to a C++ type cast.

The versions that convert floating point values to x86 registers only work on scalar values, since the x86 registers are essentially scalar in nature. They are useful because they allow answers calculated using SSE floating point to be quickly and easily cast to integers in the x86 registers.

When converting to integers, you have the option of either using truncation (by placing the additional **`T`** in the middle of the mnemonic, indicated in the previous list by the **`(T)`**) or using the rounding function specified in the MXCSR register.

Selecting the Rounding Function

`STMXCSR [mem32]   – Store MXCSR`

`LDMXCSR [mem32]   – Load MXCSR`

The conversion instructions that convert from a floating point value to an integer perform rounding based on the rounding function (bits 13 and 14) of the MXCSR register.

Table 28: Rounding Function Bits in MXCSR

Bit 14	Bit 13	Rounding Function
0	0	Round to nearest integer
0	1	Round down to nearest integer
1	0	Round up to nearest integer
1	1	Truncate (round toward 0)

To set the rounding function in MXCSR, the register must first be copied to RAM using the store MXCSR instruction, **`STMCXSR`**. Then bits 13 and 14 can be set in RAM using the bit test instructions, **`BTS`** and **`BTR`** (or the Boolean instructions). Finally, this altered value of MXCSR can be loaded from RAM back into the real MXCSR using the load MXCSR instruction, **`LDMXCSR`**.

The **`LDMXCSR`** and **`STMCXSR`** instructions both take a single 32-bit memory operand. In **`STMXCSR`**, this operand is a variable to store the MXCSR in RAM. In **`LDMXCSR`**, the operand is the memory location from which to copy the new values of the MXCSR.

```
; Example of selecting Round to Nearest, 00

STMXCSR mxcsrState  ; Copy MXCSR to the 32-bit memory operand

btr mxcsrState, 13  ; Unset both bits, set it to 0

btr mxcsrState, 14

LDMXCSR mxcsrState  ; Load the altered value back into MXCSR


; Example of selecting Round Down, 01

STMXCSR mxcsrState  ; Copy MXCSR to the 32-bit memory operand

bts mxcsrState, 13  ; Set bit 13 to 1

btr mxcsrState, 14  ; And bit 14 to 0

LDMXCSR mxcsrState  ; Load the altered value back into MXCSR
```

Conclusion

This book has been a brief introduction to a massive topic, including the most basic aspects of modern x64 assembly programming. The SSE instruction sets have not been covered to any degree of detail, and there are simply far too many instructions to do these instruction sets any justice within a hundred pages.

Assembly programming is very different from high-level languages. It requires practice and a very detailed knowledge of the hardware with which the application is executing. If the CPU's native assembly language is used, there is an excellent opportunity to speed up data processing. This language and the CPUs it runs on are at the very cutting edge of technology. CPUs and their assembly languages are the result of thousands of very intelligent people working very hard over generations. It is complex but extraordinarily powerful.

I recommend having the AMD, Intel, and VIA manufacturer's manuals as references. They are available from the manufacturer's websites (there is a link in the Recommended Reading section). These books have many thousands of pages and are a testament to just how large the topic of modern x64 CPUs and their assembly language can be. They are excellent references and should be kept within reach at all times when programming these CPUs. They are the most complete descriptions available.

To output the assembly code from a C++ project, click **Properties**, and then click **C/C++ Output Files** and change the **Assembler Output** option. This outputs the assembly code produced by the C++ compiler. Modern C++ compilers generate extremely efficient assembly code, and this output is an excellent place to study how to use this language.

Thank you very much for reading this book, and I hope it has been a helpful introduction to a fascinating topic.

Recommended Reading

Intel Programmer's Manuals: There is really no better reference on how Intel CPUs should be programmed and how they operate than the company's own programmer's manuals. Even if you are coding for AMD hardware, it is recommended that you keep the Intel manuals as well, because they offer a second explanation of every instruction. The manuals are available from the company's website:

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

AMD Programmer's Manuals: The AMD programmer's manuals are worth keeping as a reference, even if you are developing specifically for Intel hardware, as they offer a second explanation of each instruction. Intel and AMD CPUs are functionally identical in many respects. They are available from the company's website:

http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/

Agner Fog's Optimization Manuals: After the Intel and AMD programmer's manuals, the most comprehensive work on low-level modern CPU programming is the *Optimization Manuals* by Agner Fog. These manuals contain an extraordinary amount of information, including instruction latency and throughput for many modern CPUs. A large amount of this information is not even published by Intel or AMD.

http://www.agner.org/optimize/#manuals

Graphics Programming Black Book by Michael Abrash: Michael Abrash is responsible (with John Carmack and others) for programming and optimizing early 3-D engines, such as those used for *Doom II* and *Quake*. These engines allowed gamers to play 3-D games on very primitive hardware, such as 486s and the original Pentium processors. Hardware has changed considerably since this book was written, but most of the information is still applicable to some degree on modern hardware.

http://www.nondot.org/sabre/Mirrored/GraphicsProgrammingBlackBook/

Art of Assembly by Randall Hyde: *The Art of Assembly* has now moved its focus away from MASM and towards Randall's new HLA (High Level Assembler). It is a very good read whether you use HLA or MASM. Early chapters focus on basic concepts like number systems, data types, Boolean arithmetic, and memory. All of which Hyde explains in great detail, and yet the book is very accessible and easy to understand.

http://www.plantation-productions.com/Webster/

The Art of Computer Programming by Donald Knuth: This is a series of books on the study of computer algorithms. It is written in MMIX assembler, which is the language of a hypothetical CPU designed to illustrate machine-level programming. It is easily the most comprehensive and important text on computer algorithms ever written. The set is available from Amazon.

http://www.amazon.com/Computer-Programming-Volumes-1-4A-Boxed/dp/0321751043

C++ Compiler Assembly Listings: Possibly the best way to study assembly language is to examine the output of the C++ compiler. It is difficult to be more optimized than the compiler without studying how the compiler optimizes. Assembly code can be generated from any C++ projects.

Open the **File** menu and select the **Project** option, and from that select the **Project Options**. Then click **C/C++**, **Output Files**, and finally click **Assembler Output.**

You can also open the **Disassembly** window at runtime while debugging a program by opening the **Debug** menu, selecting **Windows**, and then clicking **Disassembly** when the program halts at a breakpoint. This is very helpful as you can step through the code one instruction at a time, and view the results on the registers and memory.