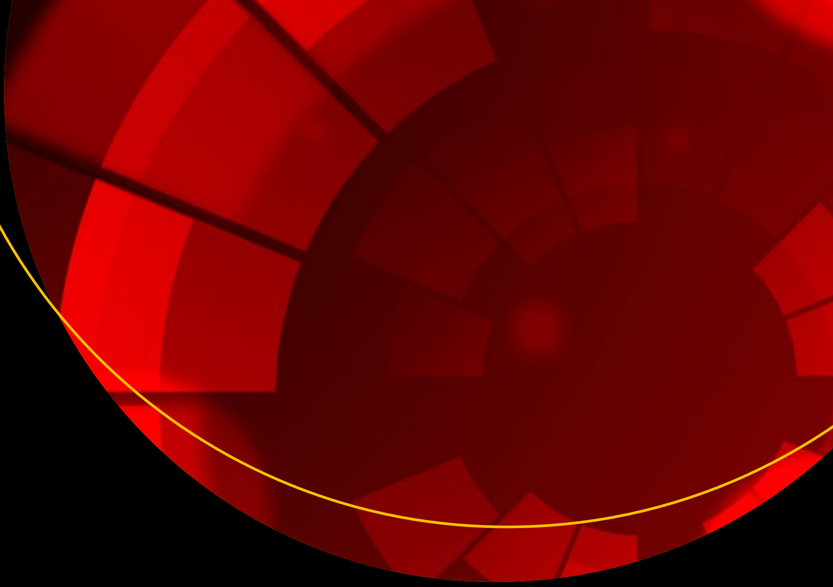

Modern X86 Assembly Language Programming

Covers x86 64-bit, AVX, AVX2, and AVX-512

Second Edition

Daniel Kusswurm

Apress®

Modern X86 Assembly Language Programming

Covers x86 64-bit, AVX, AVX2, and AVX-512

Second Edition

Daniel Kusswurm

Apress®

Modern X86 Assembly Language Programming: Covers x86 64-bit, AVX, AVX2, and AVX-512

Daniel Kusswurm
Geneva, IL, USA

ISBN-13 (pbk): 978-1-4842-4062-5 ISBN-13 (electronic): 978-1-4842-4063-2
https://doi.org/10.1007/978-1-4842-4063-2

Library of Congress Control Number: 2018964262

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

This book is dedicated to those individuals who suffer the ravages of Alzheimer’s disease and their unsung compassionate caregivers.

Contents

About the Author

Daniel Kusswurm has over 30 years of professional experience as a software developer and computer scientist. During his career, he has developed innovative software for medical devices, scientific instruments, and image processing applications. On many of these projects, he successfully employed x86 assembly language to significantly improve the performance of computationally-intense algorithms and solve unique programming challenges. His educational background includes a BS in electrical engineering technology from Northern Illinois University along with an MS and PhD in computer science from DePaul University.

About the Technical Reviewer

Paul Cohen joined Intel Corporation during the very early days of the x86 architecture, starting with the 8086, and retired from Intel after 26 years in sales/marketing/management. He is currently partnered with Douglas Technology Group, focusing on the creation of technology books on behalf of Intel and other corporations. Paul also teaches a class that transforms middle and high school students into real, confident entrepreneurs, in conjunction with the Young Entrepreneurs Academy (YEA). He is also a Traffic Commissioner for the City of Beaverton, Oregon and on the Board of Directors of multiple non-profit organizations.

Acknowledgments

The production of a motion picture and the publication of a book are somewhat analogous. Movie trailers extol the performances of the lead actors. The front cover of a book trumpets the authors' names. Actors and authors ultimately receive public acclamation for their efforts. It is, however, impossible to produce a movie or publish a book without the dedication, expertise, and creativity of a professional behind-the-scenes team. This book is no exception.

I would like to thank the talented editorial team at Apress for their efforts especially Steve Anglin, Mark Powers, and Matthew Moodie. Paul Cohen deserves kudos for his meticulous technical review and practical suggestions. Proofreader Ed Kusswurm merits applause and recognition for his hard work and constructive feedback. I accept full responsibility for any remaining imperfections.

I would also like to thank Nirmal Selvaraj, Dulcy Nirmala, Kezia Endsley, Dhaneesh Kumar and the entire production staff at Apress for their contributions, and my professional colleagues for their support and encouragement. Finally, I would like to recognize parental nodes Armin (RIP) and Mary along with sibling nodes Mary, Tom, Ed, and John for their inspiration during the writing of this book.

Introduction

Since the invention of the personal computer, software developers have used x86 assembly language to create innovative solutions for a wide variety of algorithmic challenges. During the early days of the PC era, it was common practice to code large portions of a program or complete applications using x86 assembly language. Given the 21st Century prevalence of high-level languages such as C++, C#, Java, and Python, it may be surprising to learn that many software developers still employ assembly language to code performance-critical sections of their programs. And while compilers have improved remarkably over the years in terms of generating machine code that is both spatially and temporally efficient, situations still exist where it makes sense for a software developer to exploit the benefits of assembly language programming.

The single-instruction multiple-data (SIMD) architectures of modern x86 processors provide another explanation for the continued interest in assembly language programming. A SIMD-capable processor contains computational resources that facilitate simultaneous calculations using multiple data values, which can significantly improve the performance of applications that must deliver real-time responsiveness. SIMD architectures are also well-suited for computationally-intense problem domains, such as image processing, audio and video encoding, computer-aided design, computer graphics, and data mining. Unfortunately, many high-level languages and development tools are still unable to fully or even partially exploit the SIMD capabilities of a modern x86 processor. Assembly language, on the other hand, enables the software developer to take full advantage of a processor's SIMD resources.

Modern X86 Assembly Language Programming

Modern X86 Assembly Language Programming, Second Edition is an edifying text about x86 64-bit (x86-64) assembly language programming. The book's content and organization are designed to help you quickly understand x86-64 assembly language programming and the computational resources of Advanced Vector Extensions (AVX). It also contains an abundance of source code that is structured to accelerate learning and comprehension of essential x86-64 assembly language constructs and SIMD programming concepts. After reading and using this book, you'll be able to code performance-enhancing functions and algorithms using x86-64 assembly language and the AVX, AVX2, and AVX-512 instruction sets.

Before proceeding I should explicitly mention that this book does not cover x86-32 assembly language programming. It also doesn't discuss legacy x86 technologies such as the x87 floating-point unit, MMX, and Streaming SIMD Extensions. The first edition remains relevant if you're interested in learning about these topics. This book does not explain x86 architectural features or privileged instructions that are used in operating systems. However, you will need to thoroughly understand the material that's presented in this book to develop x86 assembly language code for use in an operating system.

While it is still theoretically possible to write an entire application program using assembly language, the demanding requirements of contemporary software development make such an approach impractical and ill advised. Instead, this book concentrates on coding x86-64 assembly language functions that are callable from C++. Each source code example was created using Microsoft Visual Studio C++ and Microsoft Macro Assembler (MASM).

Target Audience

The target audience for this book is software developers, including:

- Software developers who are creating application programs for Windows-based platforms and want to learn how to write performance-enhancing algorithms and functions using x86-64 assembly language
- Software developers who are creating application programs for non-Windows environments and want to learn x86-64 assembly language programming
- Software developers who want to learn how to create SIMD calculating functions using the AVX, AVX2, and AVX-512 instruction sets
- Software developers and computer science students who want or need to gain a better understanding of the x86-64 platform and its SIMD architecture

The principal audience for *Modern X86 Assembly Language Programming, Second Edition* is Windows software developers, since the source code examples were developed using Visual Studio C++ and MASM. Software developers who are targeting non-Windows platforms can also benefit from this book since most of the informative content is organized and communicated independent of any specific operating system. It is assumed that readers of this book will have previous high-level language programming experience and a basic understanding of C++. Familiarity with Visual Studio or Windows programming is not necessary.

Content Overview

The primary objective of this book is to help you learn x86 64-bit assembly language programming along with AVX, AVX2, and AVX-512. The book's chapters and content are structured to achieve this goal. Here's a brief overview of what you can expect to learn.

Chapter 1 covers the core architecture of the x86-64 platform. It includes a discussion of the platform's fundamental data types, internal architecture, register sets, instruction operands, and memory addressing modes. This chapter also describes the core x86-64 instruction set. Chapters 2 and 3 explain the fundamentals of x86-64 assembly language programming using the core instruction set and common programming constructs, including arrays and structures. The source code examples presented in these (and subsequent) chapters are packaged as working programs, which means that you can run, modify, or otherwise experiment with the code to enhance your learning experience.

Chapter 4 focuses on the architectural resources of AVX including its register sets, data types, and instruction set. Chapter 5 explains how to use the AVX instruction set to perform scalar floating-point arithmetic using both single-precision and double-precision values. Chapters 6 and 7 illustrate AVX SIMD programming using packed floating-point and packed integer operands.

Chapter 8 introduces AVX2 and explores its enhanced capabilities including data broadcasts, gathers, and permutes. It also explains fused-multiply-add (FMA) operations. Chapters 9 and 10 contain source code examples that exemplify a variety of computational algorithms using AVX2 with packed floating-point and packed integer operands. Chapter 11 includes source code examples that demonstrate FMA programming. This chapter also covers examples that explicate recent x86 platform extensions using the general-purpose registers.

Chapter 12 delves into the architectural details of AVX-512. This chapter describes AVX-512's register sets and data types. It also elucidates pivotal AVX-512 enhancements including conditional execution and merging, embedded broadcast operations, and instruction-level rounding. Chapters 13 and 14 contain numerous source code examples that demonstrate how to exploit these advanced features.

Chapter 15 presents an overview of a modern x86 multi-core processor and its underlying microarchitecture. This chapter also outlines specific coding strategies and techniques that can be used to boost the performance of x86 assembly language code. Chapter 16 reviews several source code examples that illustrate advanced x86 assembly language programming techniques including processor feature detection, accelerated memory accesses, and multithreaded computations.

Appendix A describes how to execute the source code examples using Visual Studio and MASM. It also includes a list of references and resources that you can consult for more information about x86 assembly language programming.

Source Code

Source code download information for this book is available on the Apress website at `https://www.apress.com/us/book/9781484240625`. For each chapter, there is a ZIP file that contains the C++ and assembly language source code files along with the Visual Studio project files. There is no setup or install program to run. You can simply extract the contents of a chapter ZIP file into a folder of your own choosing.

■ **Caution** The sole purpose of the source code is to elucidate programming examples that are directly related to the topics discussed in this book. Minimal attention is given to essential software engineering concerns such as robust error handling, security risks, numerical stability, rounding errors, or ill-conditioned functions. You are responsible for addressing these issues should you decide to use any of the source code in your own programs.

The source code examples were created using Visual Studio Professional 2017 (version 15.7.1) on a PC running Windows 10 Pro 64-bit. The Visual Studio website (`https://visualstudio.microsoft.com`) contains more information about this and the other editions of Visual Studio. Technical details regarding Visual Studio installation, configuration, and application program development are available at `https://docs.microsoft.com/en-us/visualstudio/?view=vs-2017`.

The recommended hardware platform for running the source code examples is an x86-based PC with Windows 10 64-bit and a processor that supports AVX. An AVX2 or AVX-512 compatible processor is required to run the source code examples that employ these instruction sets. You can use one of freely available utilities listed in Appendix A to determine which x86-AVX instruction set extensions your PC supports.

Additional Resources

An extensive set of x86-related programming documentation is available from both AMD and Intel. Appendix A lists several important resources that both aspiring and experienced x86 assembly language programmers will find useful. Of all the resources listed Appendix A, the most valuable reference is Volume 2 of *Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4* (`https://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`). This tome contains comprehensive programming information for every x86 processor instruction including detailed operational descriptions, lists of valid operands, affected status flags, and potential exceptions. You are strongly encouraged to consult this indispensable resource when developing your own x86 assembly language code to verify correct instruction usage.

CHAPTER 1

X86-64 Core Architecture

Chapter 1 examines the x86-64's core architecture from the perspective of an application program. It opens with a brief historical overview of the x86 platform in order to provide a frame of reference for subsequent content. This is followed by a review of fundamental, numeric, and SIMD data types. X86-64 core architecture is examined next, which includes explanations of processor register sets, status flags, instruction operands, and memory addressing modes. The chapter concludes with an overview of the core x86-64 instruction set.

Unlike high-level languages such as C and C++, assembly language programming requires the software developer to comprehend specific architectural features of the target processor before attempting to write any code. The topics discussed in this chapter will fulfill this requirement and provide a foundation for understanding the sample code that's presented later in this book. This chapter also provides the base material that's necessary to understand the x86-64's SIMD enhancements.

Historical Overview

Before examining the technical details of the x86-64's core architecture, it can be beneficial to understand how the architecture has evolved over the years. The short review that follows focuses on noteworthy processors and instruction set enhancements that have affected how software developers use x86 assembly language. Readers who are interested in a more comprehensive chronicle of the x86's lineage should consult the resources listed in Appendix A.

The x86-64 processor platform is an extension of the original x86-32 platform. The first silicon embodiment of the x86-32 platform was the Intel 80386 microprocessor, which was introduced in 1985. The 80386 extended the architecture of the 16-bit 80286 to include 32-bit wide registers and data types, flat memory model options, a 4 GB logical address space, and paged virtual memory. The 80486 processor improved the performance of the 80386 with the inclusion of on-chip memory caches and optimized instructions. Unlike the 80386 with its separate 80387 floating-point unit (FPU), most versions of the 80486 CPU also included an integrated x87 FPU.

Expansion of the x86-32 platform continued with the introduction of the first Pentium brand processor in 1993. Known as the P5 microarchitecture, performance enhancements included a dual-instruction execution pipeline, 64-bit external data bus, and separate on-chip memory caches for both code and data. Later versions (1997) of the P5 microarchitecture incorporated a new computational resource called MMX technology, which supports single-instruction multiple-data (SIMD) operations on packed integers using 64-bit wide registers. A packed integer is a collection of multiple integer values that are processed simultaneously.

The P6 microarchitecture, first used on the Pentium Pro (1995) and later on the Pentium II (1997), extended the x86-32 platform using a three-way superscalar design. This means that that the processor is able (on average) to decode, dispatch, and execute three distinct instructions during each clock cycle. Other P6 augmentations included out-of-order instruction executions, improved branch prediction algorithms,

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_1

and speculative executions. The Pentium III, also based on the P6 microarchitecture, was launched in 1999 and included a new SIMD technology called Streaming SIMD extensions (SSE). SSE adds eight 128-bit wide registers to the x86-32 platform and instructions that perform packed single-precision floating-point arithmetic.

In 2000 Intel introduced a new microarchitecture called Netburst that included SSE2, which extended the floating-point capabilities of SSE to cover packed double-precision values. SSE2 also incorporated additional instructions that enabled the 128-bit SSE registers to be used for packed integer calculations and scalar floating-point operations. Processors based on the Netburst architecture included several variations of the Pentium 4. In 2004 the Netburst microarchitecture was upgraded to include SSE3 and hyper-threading technology. SSE3 adds new packed integer and packed floating-point instructions to the x86 platform, while hyper-threading technology parallelizes the processor's front-end instruction pipelines in order to improve performance. SSE3 capable processors include 90 nm (and smaller) versions of the Pentium 4 and Xeon product lines.

In 2006 Intel launched a new microarchitecture called Core. The Core microarchitecture included redesigns of many Netburst front-end pipelines and execution units in order to improve performance and reduce power consumption. It also incorporated a number of SIMD enhancements including SSSE3 and SSE4.1. These extensions added new packed integer and packed floating-point instructions to the platform but no new registers or data types. Processors based on the Core microarchitecture include CPUs from the Core 2 Duo and Core 2 Quad series and Xeon 3000/5000 series.

A microarchitecture called Nehalem followed Core in late 2008. This microarchitecture re-introduced hyper-threading to the x86 platform, which had been excluded from the Core microarchitecture. The Nehalem microarchitecture also incorporates SSE4.2. This final x86-SSE enhancement adds several application-specific accelerator instructions to the x86-SSE instruction set. SSE4.2 also includes new instructions that facilitate text string processing using the 128-bit wide x86-SSE registers. Processors based on the Nehalem microarchitecture include first generation Core i3, i5, and i7 CPUs. It also includes CPUs from the Xeon 3000, 5000, and 7000 series.

In 2011 Intel launched a new microarchitecture called Sandy Bridge. The Sandy Bridge microarchitecture introduced a new x86 SIMD technology called Advanced Vector Extensions (AVX). AVX adds packed floating-point operations (both single-precision and double-precision) using 256-bit wide registers. AVX also supports a new three-operand instruction syntax, which improves code efficiency by reducing the number of register-to-register data transfers that a software function must perform. Processors based on the Sandy Bridge microarchitecture include second and third generation Core i3, i5, and i7 CPUs along with Xeon V2 series CPUs.

In 2013 Intel unveiled its Haswell microarchitecture. Haswell includes AVX2, which extends AVX to support packed-integer operations using 256-bit wide registers. AVX2 also supports enhanced data transfer capabilities with its broadcast, gather, and permute instructions. (Broadcast instructions replicate a single value to multiple locations; data gather instructions load multiple elements from non-contiguous memory locations; permute instructions rearrange the elements of a packed operand.) Another feature of the Haswell microarchitecture is its inclusion of fused-multiply-add (FMA) operations. FMA enables software algorithms to perform product-sum (or dot product) calculations using a single floating-point rounding operation, which can improve both performance and accuracy. The Haswell microarchitecture also encompasses several new general-purpose register instructions. Processors based on the Haswell microarchitecture include fourth generation Core i3, i5, and i7 CPUs. AVX2 is also included later generations of Core family CPUs, and in Xeon V3, V4, and V5 series CPUs.

X86 platform extensions over the past several years have not been limited to SIMD enhancements. In 2003 AMD introduced its Opteron processor, which extended the x86's execution platform from 32 bits to 64 bits. Intel followed suit in 2004 by adding essentially the same 64-bit extensions to its processors starting with certain versions of the Pentium 4. All Intel processors based on the Core, Nehalem, Sandy Bridge, Haswell, and Skylake microarchitectures support the x86-64 execution environment.

Processors from AMD have also evolved over the past few years. In 2003 AMD introduced a series of processors based on its K8 microarchitecture. Original versions of the K8 included support for MMX, SSE, and SSE2 while later versions added SSE3. In 2007 the K10 microarchitecture was launched and included a

SIMD enhancement called SSE4a. SSE4a contains several mask shift and streaming store instructions that are not available on processors from Intel. Following the K10, AMD introduced a new microarchitecture called Bulldozer in 2011. The Bulldozer microarchitecture includes SSSE3, SSE4.1, SSE4.2, SSE4a, and AVX. It also includes FMA4, which is a four-operand version of fused-multiply-add. Like SSE4a, processors marketed by Intel do not support FMA4 instructions. A 2012 update to the Bulldozer microarchitecture called Piledriver includes support for both FMA4 and the three-operand version of FMA, which is called FMA3 by some CPU feature-detection utilities and third-party documentation sources. The most recent AMD microarchitecture, introduced during 2017, is called Zen. This microarchitecture includes the AVX2 instruction set enhancements and is used in the Ryzen series of processors.

High-end desktop and server-oriented processors based on Intel's Skylake-X microarchitecture, also first marketed during 2017, include a new SIMD extension called AVX-512. This architectural enhancement supports packed integer and floating-point operations using 512-bit wide registers. AVX-512 also includes architectural additions that facilitate instruction-level conditional data merging, floating-point rounding control, and broadcast operations. Over the next few years, it is expected that both AMD and Intel will incorporate AVX-512 into their mainstream processors for desktop and notebook PCs.

Data Types

Programs written using x86 assembly language can use a wide variety of data types. Most program data types originate from a small set of fundamental data types that are intrinsic to the x86 platform. These fundamental data types enable the processor to perform numerical and logical operations using signed and unsigned integers, single-precision (32-bit) and double-precision (64-bit) floating-point values, text strings, and SIMD values. In this section, you'll learn about the fundamental data types along with a few miscellaneous data types supported by the x86.

Fundamental Data Types

A fundamental data type is an elementary unit of data that is manipulated by the processor during program execution. The x86 platform supports fundamental data types ranging in size from 8 bits (1 byte) to 128 bits (16 bytes). Table 1-1 shows these types along with typical use patterns.

Table 1-1. *Fundamental Data Types*

Data Type	Size (Bits)	Typical Use
Byte	8	Characters, small integers
Word	16	Characters, integers
Doubleword	32	Integers, single-precision floating-point
Quadword	64	Integers, double-precision floating-point
Double Quadword	128	Packed integers, packed floating-point

Unsurprisingly, the fundamental data types are sized using integer powers of two. The bits of a fundamental data type are numbered from right to left with zero and size – 1 used to identify the least and most significant bits, respectively. Fundamental data types larger than a single byte are stored in consecutive memory locations starting with the least-significant byte at the lowest memory address. This type of in-memory byte ordering is called little endian. Figure 1-1 illustrates the bit numbering and byte ordering schemes that are used by the fundamental data types.

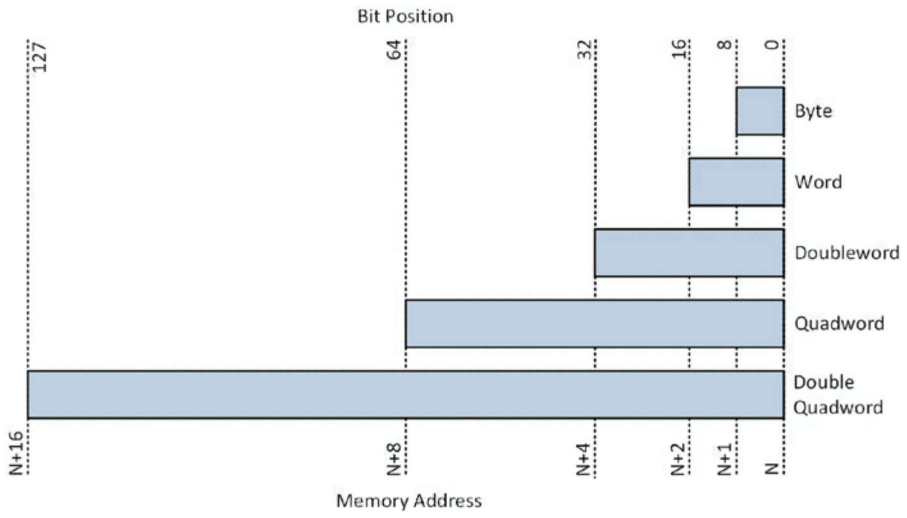

Figure 1-1. *Bit-numbering and byte-ordering for fundamental data types*

A properly-aligned fundamental data type is one whose address is evenly divisible by its size in bytes. For example, a doubleword is properly aligned when it's stored at a memory location with an address that is evenly divisible by four. Similarly, quadwords are properly aligned at addresses evenly divisible by eight. Unless specifically enabled by the operating system, an x86 processor does not require proper alignment of multi-byte fundamental data types in memory. However, it is standard practice to properly align all multi-byte values whenever possible in order to avoid potential performance penalties that can occur if the processor is required to access misaligned data in memory.

Numerical Data Types

A numerical data type is an elementary scalar value such as an integer or floating-point number. All numerical data types recognized by the CPU are represented using one of the fundamental data types discussed in the previous section. Table 1-2 contains a list of x86 numerical data types along with corresponding C/C++ types. This table also includes the fixed-size types that are defined in the C++ header file `<cstdint>` (see `http://www.cplusplus.com/reference/cstdint/` for more information about this header file). The x86-64 instruction set intrinsically supports arithmetic and logical operations using 8-, 16-, 32-, and 64-bit integers, both signed and unsigned. It also supports arithmetic calculations and data manipulation operations using single-precision and double-precision floating-point values.

Table 1-2. *X86 Numerical Data Types*

Type	Size (Bits)	C/C++ Type	<cstdint>
Signed integers	8	`char`	`int8_t`
	16	`short`	`int16_t`
	32	`int, long`	`int32_t`
	64	`long long`	`int64_t`
Unsigned integers	8	`unsigned char`	`uint8_t`
	16	`unsigned short`	`uint16_t`
	32	`unsigned int, unsigned long`	`uint32_t`
	64	`unsigned long long`	`uint64_t`
Floating-point	32	`float`	Not applicable
	64	`double`	Not applicable

SIMD Data Types

A SIMD data type is contiguous collection of bytes that's used by the processor to perform an operation or calculation using multiple values. A SIMD data type can be regarded as a container object that holds several instances of the same fundamental data type (e.g., bytes, words, double words, or quadwords). Like fundamental data types, the bits of a SIMD data type are numbered from right to left with zero and size – 1 denoting the least and most significant bits, respectively. Little-endian ordering is also used when SIMD values are stored in memory, as illustrated in Figure 1-2.

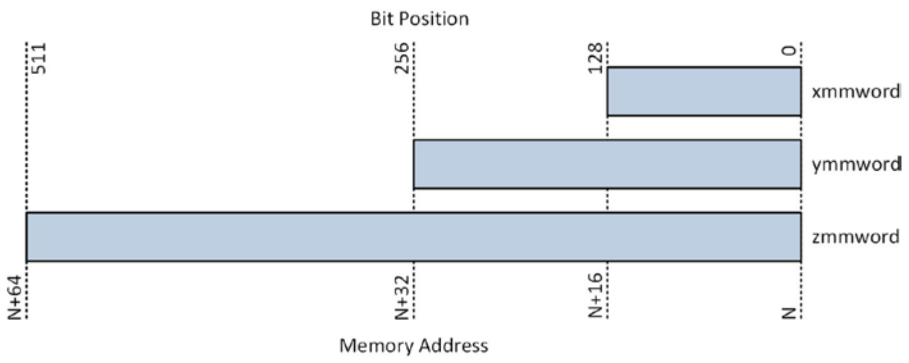

Figure 1-2. *SIMD data types*

Programmers can use SIMD (or packed) data types to perform simultaneous calculations using either integers or floating-point values. For example, a 128-bit wide packed data type can be used to hold sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers, or two 64-bit integers. A 256-bit wide packed data type can hold a variety of data elements including eight single-precision floating-point values or four double-precision floating-point values. Table 1-3 contains a complete list of the SIMD data types and the maximum number of elements for various numerical data types.

Table 1-3. *SIMD Data Types and Maximum Number of Data Elements*

Numerical Type	xmmword	ymmword	zmmword
8-bit integer	16	32	64
16-bit integer	8	16	32
32-bit integer	4	8	16
64-bit integer	2	4	8
Single-precision floating-point	4	8	16
Double-precision floating-point	2	4	8

As discussed earlier in this chapter, SIMD enhancements have been regularly added to the x86 platform starting in 1997 with MMX technology and most recently with the addition of AVX-512. This presents some challenges to the software developer who wants to exploit these technologies in that the packed data types described in Table 1-3 and their associated instruction sets are not universally supported by all processors. Fortunately, methods are available to determine at runtime the specific SIMD features and instruction sets that a processor supports. You'll learn how to use some of these methods in Chapter 16.

Miscellaneous Data Types

The x86 platform also supports a number of miscellaneous data types including strings, bit fields, and bit strings. An x86 string is contiguous block of bytes, words, doublewords, or quadwords. X86 strings are used to support text-based data types and processing operations. For example, the C/C++ data types `char` and `wchar_t` are usually implemented using an x86 byte or word, respectively. X86 strings can also be employed to perform processing operations on arrays, bitmaps, and similar contiguous-block data structures. The x86 instruction set includes instructions that can carry out compare, load, move, scan, and store operations using strings.

Other miscellaneous data types include bit fields and bit strings. A bit field is a contiguous sequence of bits and is used as a mask value by some instructions. A bit field can start at any bit position within a byte and contain up to 32 bits. A bit string is a contiguous sequence of bits containing up to $2^{32} - 1$ bits. The x86 instruction set includes instructions that can clear, set, scan, and test individual bits within a bit string.

Internal Architecture

From the perspective of an executing program, the internal architecture of an x86-64 processor can be logically partitioned into several distinct units. These include the general-purpose registers, status and control flags (RFLAGS register), instruction pointer (RIP register), XMM registers, and floating-point control and status (MXCSR). By definition, an executing program uses the general-purpose registers, the RFLAGS register, and the RIP register. Program utilization of the XMM, YMM, ZMM, or MXCSR registers is optional. Figure 1-3 illustrates the internal architecture of an x86-64 processor.

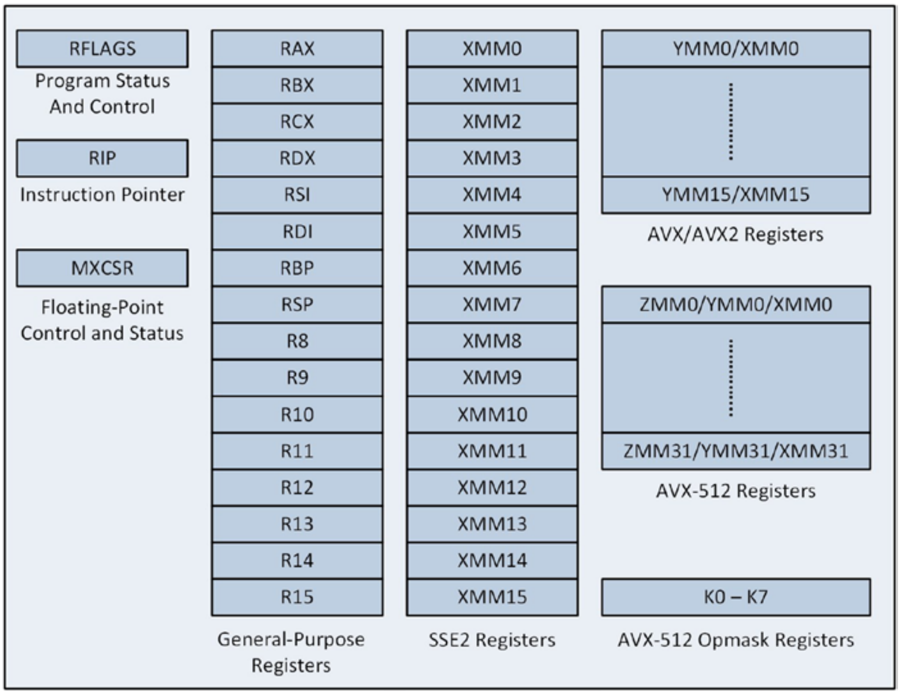

Figure 1-3. *X86-64 processor internal architecture*

All x86-64 compatible processors support SSE2 and include 16 128-bit XMM registers that programmers can use to perform scalar floating-point computations. These registers can also be employed to carry out SIMD operations using packed integers or packed floating-point values (both single precision and double precision). You'll learn how to use the XMM registers, the MXCSR register, and the AVX instruction set to perform floating-point calculations in Chapter 4 and 5. This chapter also discusses the YMM register set and other AVX architectural concepts in greater detail. You'll learn about AVX2 and AVX-512 in Chapters 8 and 12, respectively.

General-Purpose Registers

The x86-64 execution unit contains 16 64-bit general-purpose registers, which are used to perform arithmetic, logical, compare, data transfer, and address calculation operations. They also can be used as temporary storage locations for constant values, intermediate results, and pointers to data values stored in memory. Figure 1-4 shows the complete set of x86-64 general-purpose registers along with their instruction operand names.

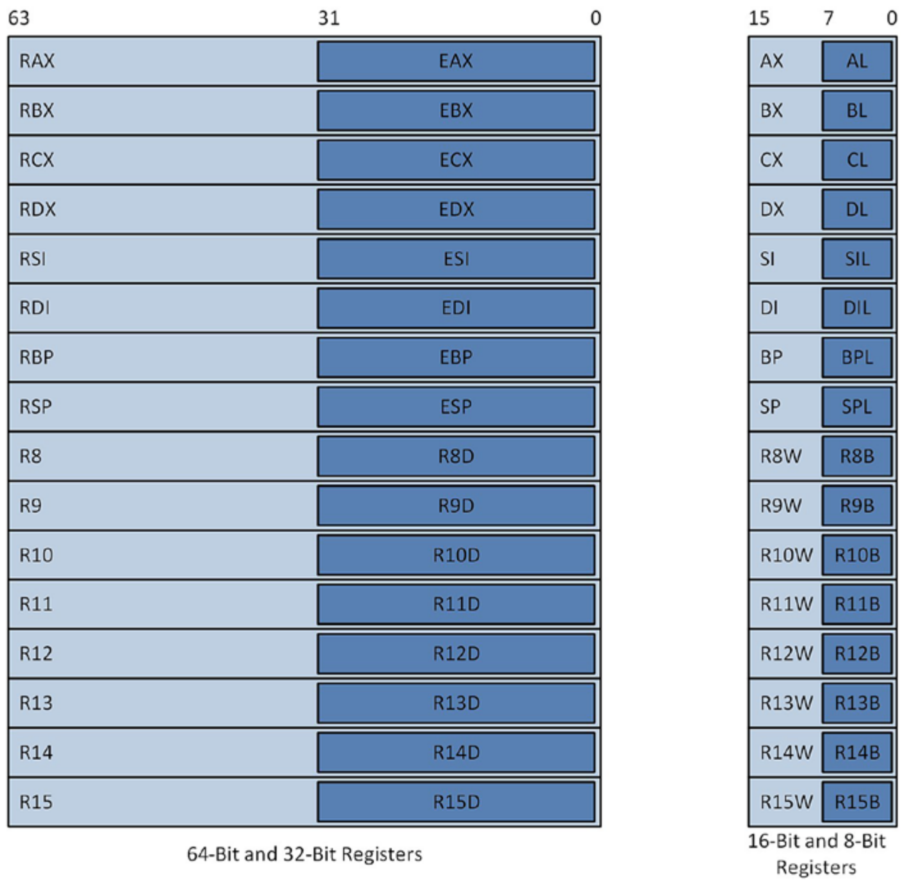

Figure 1-4. *X86-64 general-purpose registers*

The low-order doubleword, word, and byte of each 64-bit register are independently accessible and can be used to manipulate 32-bit, 16-bit, and 8-bit wide operands. For example, a function can use registers EAX, EBX, ECX, and EDX to perform 32-bit calculations in the low-order doublewords of registers RAX, RBX, RCX, and RDX, respectively. Similarly, registers AL, BL, CL, and DL can be used to carry out 8-bit calculations in the low-order bytes. It should be noted that a discrepancy exists regarding the names of some byte registers. The Microsoft 64-bit assembler uses the names shown in Figure 1-4, while the Intel documentation uses the names R8L – R15L. This book uses the Microsoft register names in order to maintain consistency between the text and the sample code. Not shown in Figure 1-4 are the legacy byte registers AH, BH, CH, and DH. These registers are aliased to the high-order bytes of registers AX, BX, CX, and DX, respectively. The legacy byte registers can be used in x86-64 programs, albeit with some restrictions, as described later in this chapter.

Despite their designation as general-purpose registers, the x86-64 instruction set imposes some notable restrictions on how they can be used. Some instructions either require or implicitly use specific registers as operands. This is a legacy design pattern that dates back to the 8086 ostensibly to improve code density. For example, some variations of the `imul` (Signed Integer Multiplication) instruction save the calculated integer product to RDX:RAX, EDX:EAX, DX:AX, or AX (the colon notation signifies that the final product is contained in two registers, with the first register holding the high-order bits). The `idiv` (Signed Integer Division) instruction requires the integer dividend to be loaded in RDX:RAX, EDX:EAX, DX:AX, or AX. The x86 string instructions require that the addresses of the source and destination operands be placed in registers RSI and RDI, respectively. String instructions that include a repeat prefix must use RCX as the count register, while variable-bit shift and rotate instructions must load the count value into register CL.

The processor uses register RSP to support stack-related operations such as function calls and returns. The stack itself is simply a contiguous block of memory that is assigned to a process or thread by the operating system. Application programs can also use the stack to pass function arguments and store temporary data. The RSP register always points to the stack's top most item. Stack push and pop operations are performed using 64-bit wide operands. This means that the location of the stack in memory is usually aligned to an 8-byte boundary. Some runtime environments (e.g., 64-bit Visual C++ programs running on Windows) align stack memory and RSP to a 16-byte boundary in order to avoid improperly-aligned memory transfers between the XMM registers and 128-bit wide operands stored on the stack.

While it is technically possible to use the RSP register as a general-purpose register, such use is impractical and strongly discouraged. Register RBP is typically used as a base pointer to access data items that are stored on the stack. RSP can also be used as a base pointer to access data items on the stack. When not employed as a base pointer, programs can use RBP as a general-purpose register.

RFLAGS Register

The RFLAGS register contains a series of status bits (or flags) that the processor uses to signify the results of an arithmetic, logical, or compare operation. It also contains a number of control bits that are primarily used by operating systems. Table 1-4 shows the organization of the bits in the RFLAGS register.

Table 1-4. *RFLAGS Register*

Bit Position	Name	Symbol	Use
0	Carry Flag	CF	Status
1	Reserved		1
2	Parity Flag	PF	Status
3	Reserved		0
4	Auxiliary Carry Flag	AF	Status
5	Reserved		0
6	Zero Flag	ZF	Status
7	Sign Flag	SF	Status
8	Trap Flag	TF	System
9	Interrupt Enable Flag	IF	System
10	Direction Flag	DF	Control
11	Overflow Flag	OF	Status
12	I/O Privilege Level Bit 0	IOPL	System
13	I/O Privilege Level Bit 1	IOPL	System
14	Nested Task	NT	System
15	Reserved		0

(*continued*)

Table 1-4. *(continued)*

Bit Position	Name	Symbol	Use
16	Resume Flag	RF	System
17	Virtual 8086 Mode	VM	System
18	Alignment Check	AC	System
19	Virtual Interrupt Flag	VIF	System
20	Virtual Interrupt Pending	VIP	System
21	ID Flag	ID	System
22 - 63	Reserved		0

For application programs, the most important bits in the RFLAGS register are the following status flags: carry flag (CF), overflow flag (OF), parity flag (PF) , sign flag (SF) , and zero flag (ZF). The carry flag is set by the processor to signify an overflow condition when performing unsigned integer arithmetic. It is also used by some register rotate and shift instructions. The overflow flag signals that the result of a signed integer operation is too small or too large. The processor sets the parity flag to indicate whether the least-significant byte of an arithmetic, compare, or logical operation contains an even number of 1 bits (parity bits are used by some communication protocols to detect transmission errors). The sign and zero flags are set by arithmetic and logical instructions to signify a negative, zero, or positive result.

The RFLAGS register contains control bit called the direction flag (DF). An application program can set or reset the direction flag, which defines the auto increment direction (0 = low to high addresses, 1 = high to low addresses) of the RDI and RSI registers during execution of string instructions. The remaining bits in the RFLAGS register are used exclusively by the operating system to manage interrupts, restrict I/O operations, support program debugging, and handle virtual operations. They should never be modified by an application program. Reserved bits also should never be modified, and no assumptions should ever be made regarding the state of any reserved bit.

Instruction Pointer

The instruction pointer register (RIP) contains the logical address of the next instruction to be executed. The value in register RIP updates automatically during execution of each instruction. It is also implicitly altered during execution of control-transfer instructions. For example, the `call` (Call Procedure) instruction pushes the contents of the RIP register onto the stack and transfers program control to the address designated by the specified operand. The `ret` (Return from Procedure) instruction transfers program control by popping the top-most eight bytes off the stack and loading them into the RIP register.

The `jmp` (Jump) and `jcc` (Jump if Condition is Met) instructions also transfer program control by modifying the contents of the RIP register. Unlike the `call` and `ret` instructions, all x86-64 jump instructions are executed independent of the stack. The RIP register is also used for displacement-based operand memory addressing as explained in the next section. It is not possible for an executing task to directly access the contents of the RIP register.

Instruction Operands

All x86-64 instructions use operands, which designate the specific values that an instruction will act upon. Nearly all instructions require one or more source operands along with a single destination operand. Most instructions also require the programmer to explicitly specify the source and destination operands. There are, however, a number of instructions where the register operands are either implicitly specified or required by an instruction, as discussed in the previous section.

There are three basic types of operands: immediate, register, and memory. An immediate operand is a constant value that is encoded as part of the instruction. These are typically used to specify constant values. Only source operands can specify an immediate value. Register operands are contained in a general-purpose or SIMD register. A memory operand specifies a location in memory, which can contain any of the data types described earlier in this chapter. An instruction can specify either the source or destination operand as a memory operand but not both. Table 1-5 contains several examples of instructions that employ the various operand types.

***Table 1-5.** Examples of Basic Operand Types*

Type	Example	Analogous C/C++ Statement
Immediate	`mov rax,42`	`rax = 42`
	`imul r12,-47`	`r12 *= -47`
	`shl r15,8`	`r15 <<= 8`
	`xor ecx,80000000h`	`ecx ^= 0x80000000`
	`sub r9b,14`	`r9b -= 14`
Register	`mov rax,rbx`	`rax = rbx`
	`add rbx,r10`	`rbx += r10`
	`mul rbx`	`rdx:rax = rax * rbx`
	`and r8w,0ff00h`	`r8w &= 0xff00`
Memory	`mov rax,[r13]`	`rax = *r13`
	`or rcx,[rbx+rsi*8]`	`rcx |= *(rbx+rsi*8)`
	`sub qword ptr [r8],17`	`*(long long*)r8 -= 17`
	`shl word ptr [r12],2`	`*(short*)r12 <<= 2`

The `mul rbx` (Unsigned Multiply) instruction that is shown in Table 1-5 is an example of implicit operand usage. In this example, implicit register RAX and explicit register RBX are used as the source operands, and implicit register pair RDX:RAX is the destination operand. The multiplicative product's high-order and low-order quadwords are stored in RDX and RAX, respectively.

In Table 1-5's penultimate example, the text `qword ptr` is an assembler operator that acts like a C/C++ cast operator. In this instance, the value 17 is subtracted from a 64-bit value whose memory location is specified by the contents of register R8. Without the `qword ptr` operator, the assembly language statement is ambiguous since the assembler can't ascertain the size of the operand pointed to by R8. In this example, the destination could also an 8-bit, 16-bit, or 32-bit sized operand. The final example in Table 1-5 uses the `word ptr` operator in a similar manner. You'll learn more about assembler operators and directives in the programming chapters of this book.

Memory Addressing

An x86-64 instruction requires up to four separate components in order to specify the location of an operand in memory. The four components include a constant displacement value, a base register, an index register, and a scale factor. Using these components, the processor calculates an effective address for a memory operand as follows:

```
EffectiveAddress = BaseReg + IndexReg * ScaleFactor + Disp
```

The base register (`BaseReg`) can be any general-purpose register. The index register (`IndexReg`) can be any general-purpose register except RSP. Valid scale factors (`ScaleFactor`) include 2, 4, and 8. Finally, the displacement (`Disp`) is a constant 8-bit, 16-bit, or 32-bit signed offset that's encoded within the instruction. Table 1-6 illustrates x86-64 memory addressing using different forms of the `mov` (Move) instruction. In these examples, register RAX (the destination operand) is loaded with the quadword value that's specified by the source operand. Note that it is not necessary for an instruction to explicitly specify all of the components required for an effective address. For example, a default value of zero is used for the displacement if an explicit value is not specified. The final size of an effective address calculation is always 64 bits.

Table 1-6. *Memory Operand Addressing*

Addressing Form	Example
`RIP + Disp`	`mov rax,[Val]`
`BaseReg`	`mov rax,[rbx]`
`BaseReg + Disp`	`mov rax,[rbx+16]`
`IndexReg * SF + Disp`	`mov rax,[r15*8+48]`
`BaseReg + IndexReg`	`mov rax,[rbx+r15]`
`BaseReg + IndexReg + Disp`	`mov rax,[rbx+r15+32]`
`BaseReg + IndexReg * SF`	`mov rax,[rbx+r15*8]`
`BaseReg + IndexReg * SF + Disp`	`mov rax,[rbx+r15*8+64]`

The memory addressing forms shown in Table 1-6 are used to directly reference program variables and data structures. For example, the simple displacement form is often used to access a simple global or static variable. The base register form is analogous to a C/C++ pointer and is used to indirectly reference a single value. Individual fields within a data structure can be retrieved using a base register and a displacement. The index register forms are useful for accessing individual elements within an array. Scale factors can reduce the amount code needed to access the elements of an array that contains integer or floating-point values. Elements in more elaborate data structures can be referenced by using a base register together with an index register, scale factor, and displacement.

The `mov rax,[Val]` instruction that's shown in the first row of Table 1-6 is an example of RIP-relative (or instruction pointer relative) addressing. With RIP-relative addressing, the processor calculates an effective address using the contents of the RIP register and a signed 32-bit displacement value that's encoded within the instruction. Figure 1-5 illustrates this calculation in greater detail. Note the little endian ordering of the displacement value that's embedded in the `mov rax,[Val]` instruction. RIP-relative addressing allows the processor to reference global or static operands using a 32-bit displacement instead of a 64-bit displacement, which reduces required code space. It also facilitates position-independent code.