Modern Parallel Programming with C++ and Assembly Language

X86 SIMD Development Using AVX, AVX2, and AVX-512

Daniel Kusswurm

Apress®

Modern Parallel Programming with C++ and Assembly Language

X86 SIMD Development Using AVX, AVX2, and AVX-512

Daniel Kusswurm

Apress®

Modern Parallel Programming with C++ and Assembly Language: X86 SIMD Development Using AVX, AVX2, and AVX-512

Daniel Kusswurm
Geneva, IL, USA

ISBN-13 (pbk): 978-1-4842-7917-5 ISBN-13 (electronic): 978-1-4842-7918-2
https://doi.org/10.1007/978-1-4842-7918-2

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

Cover designed by eStudioCalamar

Cover image by Viktor Forgacs on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Daniel Kusswurm has over 35 years of professional experience as a software developer, computer scientist, and author. 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 C++ intrinsic functions, x86 assembly language, and SIMD programming techniques to significantly improve the performance of computationally intense algorithms or 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. Daniel Kusswurm is also the author of *Modern X86 Assembly Language Programming* (ISBN: 978-1484200650), *Modern X86 Assembly Language Programming, Second Edition* (ISBN: 978-1484240625), and *Modern Arm Assembly Language Programming* (ISBN: 978-1484262665), all published by Apress.

About the Technical Reviewer

Mike Kinsner is a principal engineer at Intel developing languages and parallel programming models for a variety of computer architectures. He has recently been one of the architects of Data Parallel C++. He started his career at Altera working on high-level synthesis for field-programmable gate arrays and still contributes to spatial programming models and compilers. Mike is a representative within the Khronos Group standards organization, where he works on the SYCL and OpenCL open industry standards for parallel programming. Mike holds a PhD in computer engineering from McMaster University and recently coauthored the industry's first book on SYCL and Data Parallel C++.

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 including Steve Anglin, Mark Powers, and Jim Markham for their efforts and contributions. I would also like to thank the entire production staff at Apress. Michael Kinsner warrants applause and a thank you for his comprehensive technical review and constructive comments. Ed Kusswurm merits kudos for reviewing each chapter and offering helpful suggestions. I accept full responsibility for any remaining imperfections.

Thanks to 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

SIMD (single instruction multiple data) is a parallel computing technology that simultaneously executes the same processor operation using multiple data items. For example, a SIMD-capable processor can carry out an arithmetic operation using several elements of a floating-point array concurrently. Programs often use SIMD operations to accelerate the performance of computationally intense algorithms in machine learning, image processing, audio/video encoding and decoding, data mining, and computer graphics.

Since the late 1990s, both AMD and Intel have incorporated various SIMD instruction set extensions into their respective x86 processors. The most recent x86 SIMD instruction set extensions are called AVX (Advanced Vector Extensions), AVX2, and AVX-512. These SIMD resources facilitate arithmetic and other data processing operations using multiple elements in a 128-, 256-, or 512-bit wide processor register (most standard x86 arithmetic operations are carried out using scalar values in an 8-, 16-, 32-, or 64-bit wide register).

Despite the incorporation of advanced SIMD capabilities in x86 modern processors, high-level language compilers are sometimes unable to fully exploit these resources. To optimally utilize the SIMD capabilities of a modern x86 processor, a software developer must occasionally write SIMD code that explicitly employs the AVX, AVX2, or AVX-512 instruction sets. A software developer can use either C++ SIMD intrinsic functions or assembly language programming to accomplish this. A C++ SIMD intrinsic function is code that looks like an ordinary C++ function but is handled differently by the compiler. More specifically, the compiler directly translates a C++ SIMD intrinsic function into one or more assembly language instructions without the overhead of a normal function (or subroutine) call.

Before continuing, a couple of caveats are warranted. First, the SIMD programming techniques described in this book are not appropriate for every "slow" algorithm or function. Both C++ SIMD intrinsic function use and assembly language code development should be regarded as specialized programming tools that can significantly accelerate the performance of an algorithm or function when judiciously employed. However, it is important to note that explicit SIMD coding usually requires extra effort during initial development and possibly when performing future maintenance. Second, it should be noted that SIMD parallelism is different than other types of parallel computing you may have encountered. For example, the task-level parallelism of an application that exploits multiple processor cores or threads to accelerate the performance of an algorithm is different than SIMD parallelism. Task-level parallelism and SIMD parallelism are not mutually exclusive; they are frequently utilized together. The focus of this book is x86 SIMD parallelism and software development, specifically the computational resources of AVX, AVX2, and AVX-512.

Modern Parallel Programming with C++ and Assembly Language

Modern Parallel Programming with C++ and Assembly Language is an instructional text that explains x86 SIMD programming using both C++ intrinsic functions and assembly language. The content and organization of this book are designed to help you quickly understand and exploit the computational

resources of AVX, AVX2, and AVX-512. This book also contains an abundance of source code that is structured to accelerate learning and comprehension of essential SIMD programming concepts and algorithms. After reading this book, you will be able to code performance-enhanced AVX, AVX2, and AVX-512 functions and algorithms using either C++ SIMD intrinsic functions or x86-64 assembly language.

Target Audience

The target audience for *Modern Parallel Programming with C++ and Assembly Language* is software developers including

- Software developers who are creating new programs for x86 platforms and want to learn how to code performance-enhancing SIMD algorithms using AVX, AVX2, or AVX-512
- Software developers who need to learn how to write x86 SIMD functions to accelerate the performance of existing code using C++ SIMD intrinsic functions or x86-64 assembly language functions
- Software developers, computer science/engineering students, or hobbyists who want to learn about or need to gain a better understanding of x86 SIMD architectures and the AVX, AVX2, and AVX-512 instruction sets

Readers of this book should have some previous programming experience with modern C++ (i.e., ISO C++11 or later). Some familiarity with Microsoft's Visual Studio and/or the GNU toolchain will also be helpful.

Content Overview

The primary objective of this book is to help you learn x86 SIMD programming using C++ SIMD intrinsic functions and x86-64 assembly language. The book's chapters and content are structured to achieve this goal. Here is a brief overview of what you can expect to learn.

Chapter 1 discusses SIMD fundamentals including data types, basic arithmetic, and common data manipulation operations. It also includes a brief historical overview of x86 SIMD technologies including AVX, AVX2, and AVX-512.

Chapters 2 and 3 explain AVX arithmetic and other essential operations using C++ SIMD intrinsic functions. These chapters cover both integer and floating-point operands. 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.

Chapters 4, 5, and 6 cover AVX2 using C++ SIMD intrinsic functions. In these chapters, you will learn how to code practical SIMD algorithms including image processing functions, matrix operations, and signal processing algorithms. You will also learn how to perform SIMD fused-multiply-add (FMA) arithmetic.

Chapters 7 and 8 describe AVX-512 integer and floating-point operations using C++ SIMD intrinsic functions. These chapters also highlight how to take advantage of AVX-512's wider operands to improve algorithm performance.

Chapter 9 covers supplemental x86 SIMD programming techniques. This chapter explains how to programmatically detect whether the target processor and its operating system support the AVX, AVX2, or AVX-512 instruction sets. It also describes how to utilize SIMD versions of common C++ library functions.

Chapter 10 explains x86-64 processor architecture including data types, register sets, memory addressing modes, and condition codes. The purpose of this chapter is to provide you with a solid foundation for the subsequent chapters on x86-64 SIMD assembly language programming.

Chapters 11 and 12 cover the basics of x86-64 assembly language programming. In these chapters, you will learn how to perform scalar integer and floating-point arithmetic. You will also learn about other essential assembly language programming topics including for-loops, compare operations, data conversions, and function calling conventions.

Chapter 13 and 14 explain AVX arithmetic and other operations using x86-64 assembly language. These chapters also illustrate how to code x86-64 assembly language functions that perform operations using arrays and matrices.

Chapters 15 and 16 demonstrate AVX2 and x86-64 assembly language programming. In these chapters, you will learn how to code x86-64 assembly language functions that perform image processing operations, matrix calculations, and signal processing algorithms using the AVX2 instruction set.

Chapters 17 and 18 focus on developing x86-64 assembly language code using the AVX-512 instruction set.

Chapter 19 discusses some usage guidelines and optimization techniques for both C++ SIMD intrinsic functions and assembly language code development.

Appendix A describes how to download and set up the source code. It also includes some basic instructions for using Visual Studio and the GNU toolchain. Appendix B contains a list of references and resources that you can consult for additional information about x86 SIMD programming and the AVX, AVX2, and AVX-512 instruction sets.

Source Code

The source code published in this book is available on GitHub at `https://github.com/Apress/modern-parallel-programming-cpp-assembly`.

■ **Caution** The sole purpose of the source code is to elucidate programming topics that are directly related to the content of 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 concerns should you decide to use any of the source code in your own programs.

The C++ SIMD source code examples (Chapters 2–9) can be built using either Visual Studio (version 2019 or later, any edition) on Windows or GNU C++ (version 8.3 or later) on Linux. The x86-64 assembly language source code examples (Chapters 11–18) require Visual Studio and Windows. If you are contemplating the use of x86-64 assembly language with Linux, you can still benefit from this book since most of the x86-AVX instruction explanations are OS independent (developing assembly language code that runs on both Windows and Linux is challenging due to differences between the various development tools and runtime calling conventions). To execute the source code, you must use a computer with a processor that supports AVX, AVX2, or AVX-512. You must also use a recent 64-bit operating system that supports these instruction sets. Compatible 64-bit operating systems include (but not limited to) Windows 10 (version 1903 or later), Windows 11, Debian (version 9 or later), and Ubuntu (version 18.04 LTS or later). Appendix A contains additional information about the source code and software development tools.

Additional Resources

An extensive set of x86-related SIMD programming documentation is available from both AMD and Intel. Appendix B lists several important resources that both aspiring and experienced SIMD programmers will find useful. Of all the resources listed in Appendix B, two stand out.

The Intel Intrinsics Guide website (https://software.intel.com/sites/landingpage/IntrinsicsGuide) is an indispensable online reference for information regarding x86 C++ SIMD intrinsic functions and data types. This site documents the C++ SIMD intrinsic functions that are supported by the Intel C++ compiler. Most of these functions can also be used in programs that are developed using either Visual C++ or GNU C++. Another valuable programming resource is Volume 2 of the reference manual entitled *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4* (www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html). Volume 2 contains comprehensive information for every AVX, AVX2, and AVX-512 processor instruction including detailed operational descriptions, lists of valid operands, affected status flags, and potential exceptions. You are strongly encouraged to consult this reference manual when developing your own x86 SIMD code to verify correct instruction usage.

CHAPTER 1

SIMD Fundamentals

Chapter 1 introduces x86 SIMD fundamentals and essential concepts. It begins with a section that defines SIMD. This section also introduces SIMD arithmetic using a concise source code example. The next section presents a brief historical overview of x86 SIMD instruction set extensions. The principal sections of Chapter 1 are next. These highlight x86 SIMD concepts and programming constructs including data types, arithmetic calculations, and data manipulation operations. These sections also describe important particulars related to AVX, AVX2, and AVX-512. It is important for you to understand the material presented in this chapter since it provides the necessary foundation to successfully comprehend the topics and source code discussed in subsequent chapters.

Before proceeding, a few words about terminology are warranted. In all ensuing discussions, I will use the official acronyms AVX, AVX2, and AVX-512 when explaining specific features or instructions of these x86 SIMD instruction set extensions. I will use the term x86-AVX as an umbrella expression for x86 SIMD instructions or computational resources that pertain to more than one of the aforementioned x86 SIMD extensions. The terms x86-32 and x86-64 are used to signify x86 32-bit and 64-bit processors and execution environments. This book focuses exclusively on the latter, but the former is occasionally mentioned for historical context or comparison purposes.

What Is SIMD?

SIMD (single instruction multiple data) is a parallel computing technique whereby a CPU (or processing element incorporated within a CPU) performs a single operation using multiple data items concurrently. For example, a SIMD-capable CPU can carry out a single arithmetic operation using several elements of a floating-point array simultaneously. SIMD operations are frequently employed to accelerate the performance of computationally intense algorithms and functions in machine learning, image processing, audio/video encoding and decoding, data mining, and computer graphics.

The underlying concepts behind a SIMD arithmetic calculation are probably best illustrated by a simple source code example. Listing 1-1 shows the source code for three different calculating functions that perform the same arithmetic operation using single-precision floating-point arrays.

Listing 1-1. Example Ch01_01

```
//-----------------------------------------------
//              Ch01_01_fcpp.cpp
//-----------------------------------------------

#include <immintrin.h>
#include "Ch01_01.h"
```


D. Kusswurm, *Modern Parallel Programming with C++ and Assembly Language*,
https://doi.org/10.1007/978-1-4842-7918-2_1

```
void CalcZ_Cpp(float* z, const float* x, const float* y, size_t n)
{
    for (size_t i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}

void CalcZ_Iavx(float* z, const float* x, const float* y, size_t n)
{
    size_t i = 0;
    const size_t num_simd_elements = 8;

    for (; n - i >= num_simd_elements; i += num_simd_elements)
    {
        // Calculate z[i:i+7] = x[i:i+7] + y[i:i+7]
        __m256 x_vals = _mm256_loadu_ps(&x[i]);
        __m256 y_vals = _mm256_loadu_ps(&y[i]);
        __m256 z_vals = _mm256_add_ps(x_vals, y_vals);

        _mm256_storeu_ps(&z[i], z_vals);
    }

    // Calculate z[i] = x[i] + y[i] for any remaining elements
    for (; i < n; i += 1)
        z[i] = x[i] + y[i];
}

;-----------------------------------------------
;               Ch01_01_fasm.asm
;-----------------------------------------------

;-------------------------------------------------------------------------------
; extern "C" void CalcZ_Aavx(float* z, const float* x, const float* x, size_t n);
;-------------------------------------------------------------------------------

NSE     equ 8                                   ;num_simd_elements

        .code
CalcZ_Aavx proc
        xor rax,rax                             ;i = 0;

Loop1:  mov r10,r9                              ;r10 = n
        sub r10,rax                             ;r10 = n - i
        cmp r10,NSE                             ;is n - i < NSE?
        jb Loop2                                ;jump if yes

; Calculate z[i:i+7] = x[i:i+7] + y[i:i+7]
        vmovups ymm0,ymmword ptr [rdx+rax*4]    ;ymm0 = x[i:i+7]
        vmovups ymm1,ymmword ptr [r8+rax*4]     ;ymm1 = y[i:i+7]
        vaddps ymm2,ymm0,ymm1                   ;z[i:i+7] = x[i:i+7] + y[i:i+7]
        vmovups ymmword ptr [rcx+rax*4],ymm2    ;save z[i:i+7]
```

```
        add rax,NSE                             ;i += NSE
        jmp Loop1                               ;repeat Loop1 until done

Loop2:  cmp rax,r9                              ;is i >= n?
        jae Done                                ;jump if yes

; Calculate z[i] = x[i] + y[i] for remaining elements
        vmovss xmm0,real4 ptr [rdx+rax*4]       ;xmm0 = x[i]
        vmovss xmm1,real4 ptr [r8+rax*4]        ;xmm1 = y[i]
        vaddss xmm2,xmm0,xmm1                   ;z[i] = x[i] + y[i]
        vmovss real4 ptr [rcx+rax*4],xmm2       ;save z[i]

        inc rax                                 ;i += 1
        jmp Loop2                               ;repeat Loop2 until done

Done:   vzeroupper                              ;clear upper bits of ymm regs
        ret                                     ;return to caller
CalcZ_Aavx endp
        end
```

The function `CalcZ_Cpp()`, shown at the beginning of Listing 1-1, is a straightforward non-SIMD C++ function that calculates `z[i] = x[i] + y[i]`. However, a modern C++ compiler may generate SIMD code for this function as explained later in this section.

The next function in Listing 1-1, `CalcZ_Iavx()`, calculates the same result as `CalcZ_Cpp()` but employs C++ SIMD intrinsic functions to accelerate the computations. In `CalcZ_Iavx()`, the first for-loop uses the C++ SIMD intrinsic function `_mm256_loadu_ps()` to load eight consecutive elements from array `x` (i.e., elements `x[i:i+7]`) and temporarily saves these elements in an `__m256` object named `x_vals`. An `__m256` object is a generic container that holds eight values of type `float`. The ensuing `_mm256_load_ps()` call performs the same operation using array `y`. This is followed by a call to `_mm256_add_ps()` that calculates `z[i:i+7] = x[i:i+7] + y[i:i+7]`. What makes this code different from the code in the non-SIMD function `CalcZ_Cpp()` is that `_mm256_add_ps()` performs all eight array element additions concurrently. The final C++ intrinsic function in the first for-loop, `_mm256_storeu_ps()`, saves the resulting array element sums to `z[i:i+7]`.

It is important to note that since the first for-loop in `CalcZ_Iavx()` processes eight array elements per iteration, it must terminate if there are fewer than eight elements remaining to process. The second for-loop handles any remaining (or residual) elements and only executes if `n` is not an integral multiple of eight. It is also important to mention that the C++ compiler treats C++ SIMD intrinsic function calls differently than normal C++ function calls. In the current example, the C++ compiler directly translates each `__mm256` function into its corresponding AVX assembly language instruction. The overhead associated with a normal C++ function call is eliminated.

The final function in Listing 1-1 is named `CalcZ_Aavx()`. This is an x86-64 assembly language function that performs the same array calculation as `CalcZ_Cpp()` and `CalcZ_Iavx()`. What is noteworthy about this function is that the AVX instructions `vmovps` and `vaddps` contained in the code block are the same instructions that the C++ compiler emits for the C++ SIMD intrinsic functions `_mm256_loadu_ps()` and `_mm256_add_ps()`, respectively. The remaining code in `CalcZ_Aavx()` implements the two for-loops that are also implemented in function `CalcZ_Cpp()`.

Do not worry if you are somewhat perplexed by the source code in Listing 1-1. The primary purpose of this book is to teach you how to develop and code SIMD algorithms like this using either C++ SIMD intrinsic functions or x86-64 assembly language. There are two takeaway points from this section. First, the CPU executes most SIMD arithmetic operations on the specified data elements concurrently. Second, similar design patterns are often employed when coding a SIMD algorithm regardless of whether C++ or assembly language is used.

One final note regarding the code in Listing 1-1. Recent versions of mainstream C++ compilers such as Visual C++ and GNU C++ are sometimes capable of automatically generating efficient x86 SIMD code for trivial arithmetic functions like `CalcZ_Cpp()`. However, these compilers still have difficulty generating efficient SIMD code for more complicated functions, especially ones that employ nested for-loops or nontrivial decision logic. In these cases, functions written using C++ SIMD intrinsic functions or x86-64 assembly language code can often outperform the SIMD code generated by a modern C++ compiler. However, employing C++ SIMD intrinsic functions does not improve performance in all cases. Many programmers will often code computationally intensive algorithms using standard C++ first, benchmark the code, and then recode bottleneck functions using C++ SIMD intrinsic functions or assembly language.

Historical Overview of x86 SIMD

For aspiring x86 SIMD programmers, having a basic understanding about the history of x86 SIMD and its various extensions is extremely beneficial. This section presents a brief overview that focuses on noteworthy x86 SIMD instruction set extensions. It does not discuss x86 SIMD extensions incorporated in special-use processors (e.g., Intel Xeon Phi) or x86 SIMD extensions that were never widely used. If you are interested in a more comprehensive chronicle of x86 SIMD architectures and instruction set extensions, you can consult the references listed in Appendix B.

Intel introduced the first x86 SIMD instruction set extension, called MMX, in 1997. This extension added instructions that facilitated simple SIMD operations using 64-bit wide packed integer operands. The MMX extension did not add any new registers to the x86 platform; it simply repurposed the registers in the x87 floating-point unit for SIMD integer arithmetic and other operations. In 1998, AMD launched an x86 SIMD extension called 3DNow, which facilitated vector operations using single-precision floating-point values. It also added a few new integer SIMD instructions. Like MMX, 3DNow uses x87 FPU registers to hold instruction operands. Both MMX and 3DNow have been superseded by newer x86 SIMD technologies and should not be used to develop new code.

In 1999, Intel launched a new SIMD technology called Streaming SIMD extensions (SSE). SSE adds 128-bit wide registers to the x86 platform and instructions that perform packed single-precision (32-bit) floating-point arithmetic. SSE also includes a few packed integer instructions. In 2000, SSE2 was launched and extends the floating-point capabilities of SSE to cover packed double-precision (64 bit) values. SSE2 also significantly expands the packed integer capabilities of SSE. Unlike x86-32 processors, all x86-64-compatible processors from both AMD and Intel support the SSE2 instruction set extension. The SIMD extensions that followed SSE2 include SSE3 (2004), SSSE3 (2006), SSE4.1 (2008), and SSE4.2 (2008). These extensions incorporated additional SIMD instructions that perform operations using either packed integer or floating-point operands, but no new registers or data types.

In 2011, Intel introduced processors that supported 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 assembly language instruction syntax, which helps reduce the number of register-to-register data transfers that a software function must perform. In 2013, Intel unveiled AVX2, which extends AVX to support packed-integer operations using 256-bit wide registers. AVX2 also adds enhanced data transfer capabilities with its broadcast, gather, and permute instructions. Processors that support AVX or AVX2 may also support fused-multiply-add (FMA) operations. FMA enables software algorithms to perform sum-of-product (e.g., dot product) calculations using a single floating-point rounding operation, which can improve both performance and accuracy.

Beginning in 2017, high-end desktop and server-oriented processors marketed by Intel included 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 SIMD extensions that facilitate instruction-level conditional data merging, floating-point rounding control, and embedded broadcast operations.

In addition to the abovementioned SIMD extensions, numerous non-SIMD instructions have been added to the x86 platform during the past 25 years. This ongoing evolution of the x86 platform presents some challenges to software developers who want to exploit the latest instruction sets and computational resources. Fortunately, there are techniques that you can use to determine which x86 SIMD and non-SIMD instruction set extensions are available during program execution. You will learn about these methods in Chapter 9. To ensure software compatibility with future processors, a software developer should *never* assume that a particular x86 SIMD or non-SIMD instruction set extension is available based on processor manufacturer, brand name, model number, or underlying microarchitecture.

SIMD Data Types

An x86 SIMD data type is a contiguous collection of bytes that is used by the processor to perform an arithmetic calculation or data manipulation operation using multiple values. An x86 SIMD data type can be regarded as a generic container object that holds multiple instances of the same fundamental data type (e.g., 8-, 16-, 32-, or 64-bit integers, single-precision or double-precision floating-point values, etc.). The bits of an x86 SIMD data type are numbered from right to left with *0* and *size – 1* denoting the least and most significant bits, respectively. X86 SIMD data types are stored in memory using little-endian byte ordering. In this ordering scheme, the least significant byte of an x86 SIMD data type is stored at the lowest memory address as illustrated in Figure 1-1. In this figure, the terms `xmmword`, `ymmword`, and `zmmword` are x86 assembly language expressions for 128-, 256-, and 512-bit wide SIMD data types and operands.

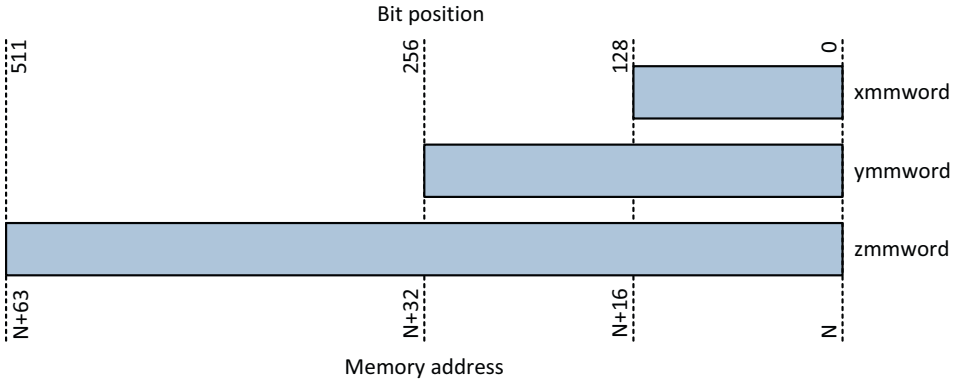

Figure 1-1. *X86 SIMD data types*

A program can use x86 SIMD (also called packed) data types to perform simultaneous calculations using either integer or floating-point values. For example, a 256-bit wide packed operand can hold thirty-two 8-bit integers, sixteen 16-bit integers, eight 32-bit integers, or four 64-bit integers. It can also be used to hold eight single-precision or four double-precision floating-point values. Table 1-1 contains a complete list of x86 SIMD data types and the maximum number of elements for various integer and floating-point types.

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

The width of an x86 SIMD instruction operand varies depending on the x86 SIMD extension and the underlying fundamental data type. AVX supports packed integer operations using 128-bit wide operands. It also supports packed floating-point operations using either 128- or 256-bit wide operands. AVX2 also supports these same operand sizes and adds support for 256-bit wide packed integer operands. Figure 1-2 illustrates the AVX and AVX2 operand types in greater detail. In this figure, the terms byte, word, doubleword, and quadword signify 8-, 16-, 32-, and 64-bit wide integers; SPFP and DPFP denote single-precision and double-precision floating-point values, respectively.

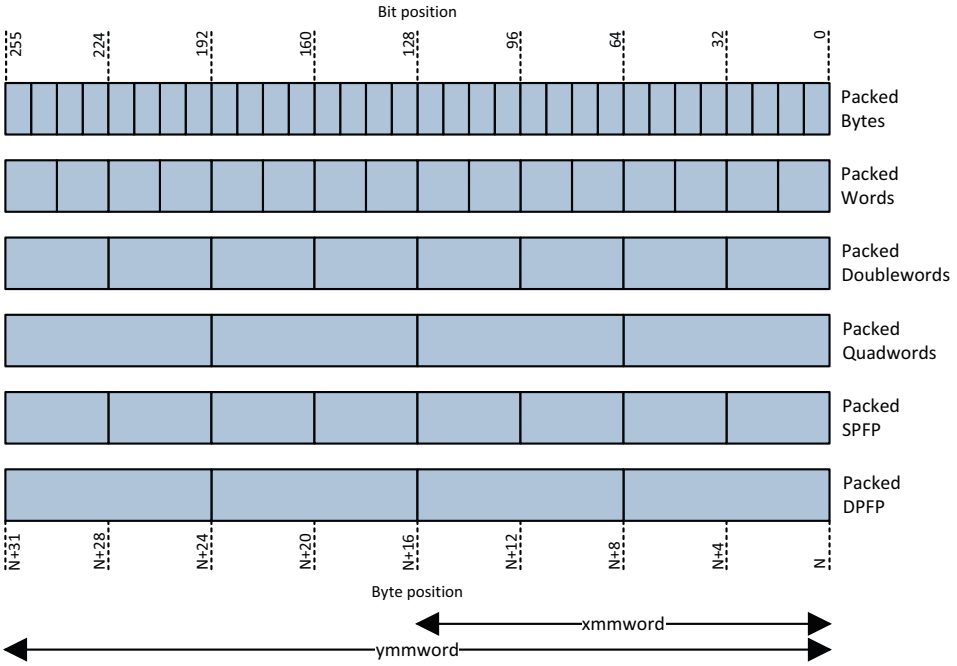

Figure 1-2. *AVX and AVX2 operands*

AVX-512 extends maximum width of an x86 SIMD operand from 256 to 512 bits. Many AVX-512 instructions can also be used with 128- and 256-bit wide SIMD operands. However, it should be noted at this point that unlike AVX and AVX2, AVX-512 is not a cohesive x86 SIMD instruction set extension. Rather, it is a collection of interrelated but distinct instruction set extensions. An AVX-512-compliant processor must

minimally support 512-bit wide operands of packed floating-point (single-precision or double-precision) and packed integer (32- or 64-bit wide) elements. The AVX-512 instructions that exercise 128- and 256-bit wide operands are a distinct x86 SIMD extension as are the instructions that support packed 8- and 16-bit wide integers. You will learn more about this in the chapters that explain AVX-512 programming. AVX-512 also adds eight opmask registers that a function can use to perform masked moves or masked zeroing.

SIMD Arithmetic

Source code example Ch01_01 introduced simple SIMD addition using single-precision floating-point elements. In this section, you will learn more about SIMD arithmetic operations that perform their calculations using either integer or floating-point elements.

SIMD Integer Arithmetic

Figure 1-3 exemplifies integer addition using 128-bit wide SIMD operands. In this figure, integer addition is illustrated using eight 16-bit integers, four 32-bit integers, or two 64-bit integers. Like the floating-point example that you saw earlier, faster processing is possible when using SIMD arithmetic since the processor can perform the required calculations in parallel. For example, when 16-bit integer elements are used in a SIMD operand, the processor performs all eight 16-bit additions simultaneously.

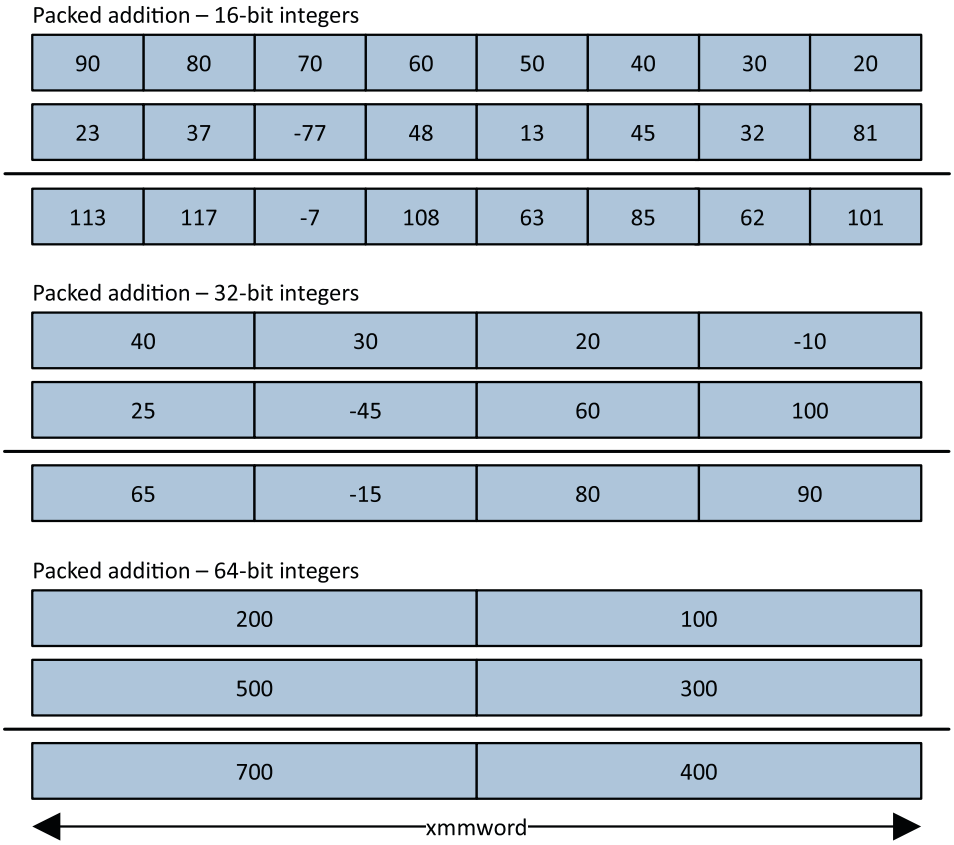

Figure 1-3. *SIMD integer addition using various element sizes*

Besides packed integer addition, x86-AVX includes instructions that perform other common arithmetic calculations with packed integers including subtraction, multiplication, shifts, and bitwise logical operations. Figure 1-4 illustrates various packed shift operations using 32-bit wide integer elements.

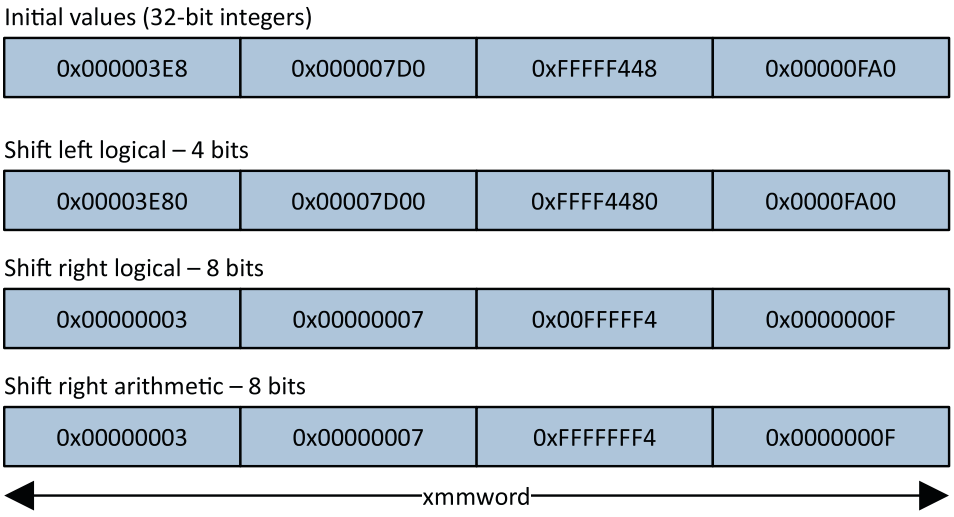

Figure 1-4. *SIMD logical and arithmetic shift operations*

Figure 1-5 demonstrates bitwise logical operations using packed 32-bit integers. Note that when performing SIMD bitwise logical operations, distinct elements are irrelevant since the logical operation is carried out using the corresponding bit positions of each SIMD operand.

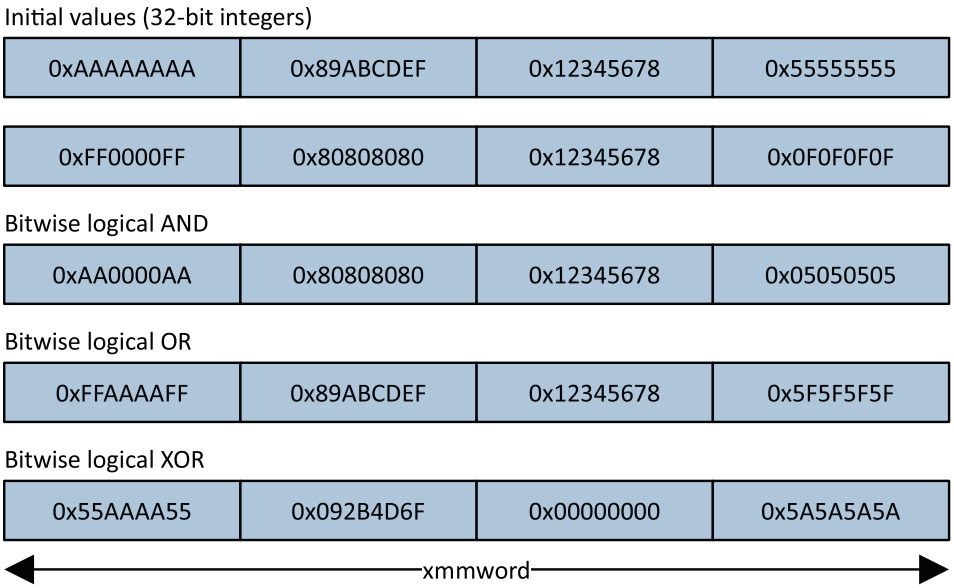

Figure 1-5. *SIMD bitwise logical operations*

Wraparound vs. Saturated Arithmetic

One notable feature of x86-AVX is its support for saturated integer arithmetic. When performing saturated integer arithmetic, the processor automatically clips the elements of a SIMD operand to prevent an arithmetic overflow or underflow condition from occurring. This is different from normal (or wraparound) integer arithmetic where an overflow or underflow result is retained. Saturated arithmetic is extremely useful when working with pixel values since it eliminates the need to explicitly check each pixel value for an overflow or underflow. X86-AVX includes instructions that perform packed saturated addition and subtraction using 8- or 16-bit wide integer elements, both signed and unsigned.

Figure 1-6 shows an example of packed 16-bit signed integer addition using both wraparound and saturated arithmetic. An overflow condition occurs when the two 16-bit signed integers are summed using wraparound arithmetic. With saturated arithmetic, however, the result is clipped to the largest possible 16-bit signed integer value. Figure 1-7 illustrates a similar example using 8-bit unsigned integers. Besides addition, x86-AVX also supports saturated packed integer subtraction as shown in Figure 1-8. Table 1-2 summarizes the saturated addition and subtraction range limits for all supported integer sizes and sign types.

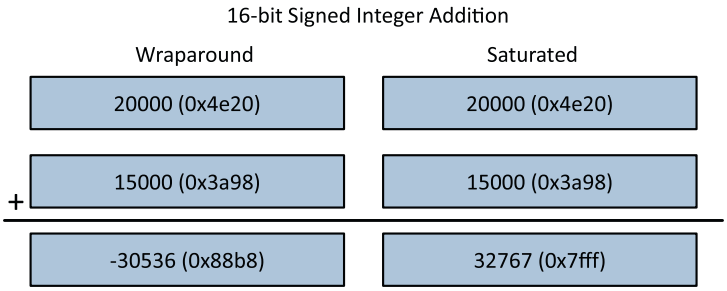

Figure 1-6. *16-bit signed integer addition using wraparound and saturated arithmetic*

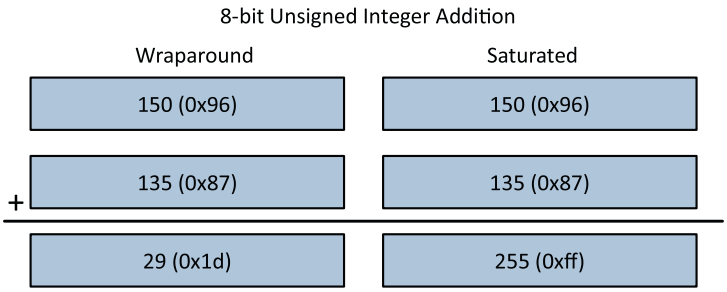

Figure 1-7. *8-bit unsigned integer addition using wraparound and saturated arithmetic*

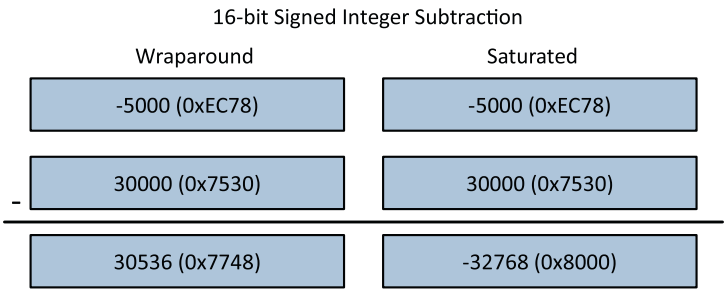

Figure 1-8. *16-bit signed integer subtraction using wraparound and saturated arithmetic*

Table 1-2. *Range Limits for Saturated Arithmetic*

Integer Type	Lower Limit	Upper Limit
8-bit signed	-128	127
8-bit unsigned	0	255
16-bit signed	-32768	32767
16-bit unsigned	0	65535

SIMD Floating-Point Arithmetic

X86-AVX supports arithmetic operations using packed SIMD operands containing single-precision or double-precision floating-point elements. This includes addition, subtraction, multiplication, division, and square roots. Figures 1-9 and 1-10 illustrate a few common SIMD floating-point arithmetic operations.

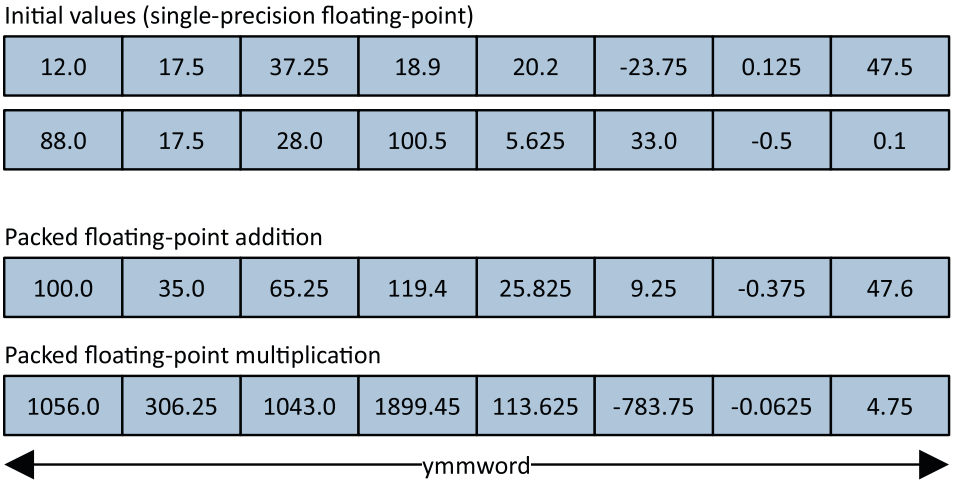

Figure 1-9. *SIMD single-precision floating-point addition and multiplication*

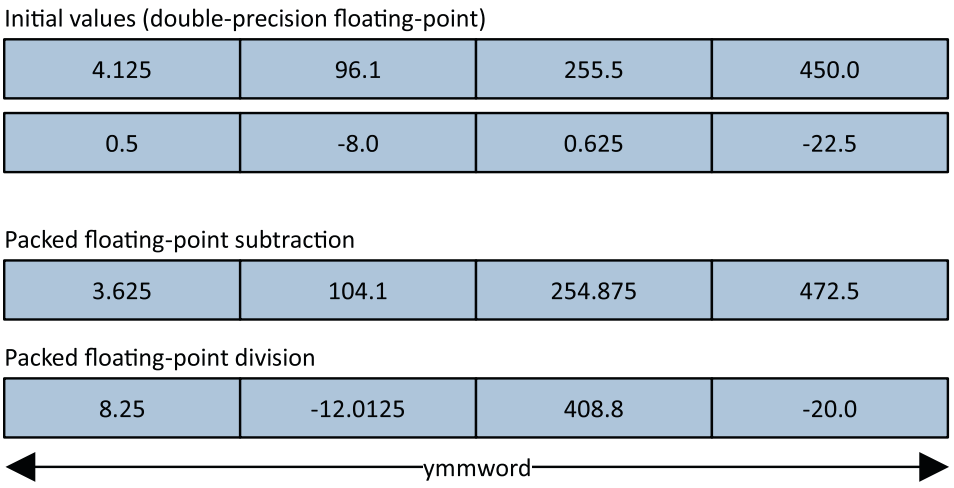

Figure 1-10. *SIMD double-precision floating-point subtraction and division*

The SIMD arithmetic operations that you have seen thus far perform their calculations using corresponding elements of the two source operands. These types of operations are usually called vertical arithmetic. X86-AVX also includes arithmetic instructions that carry out operations using the adjacent elements of a SIMD operand. Adjacent element calculations are termed horizonal arithmetic. Horizontal arithmetic is frequently used to reduce the elements of a SIMD operand to a single scalar value. Figure 1-11 illustrates horizontal addition using packed single-precision floating-point elements and horizontal subtraction using packed double-precision floating-point elements. X86-AVX also supports integer horizontal addition and subtraction using packed 16- or 32-bit wide integers.

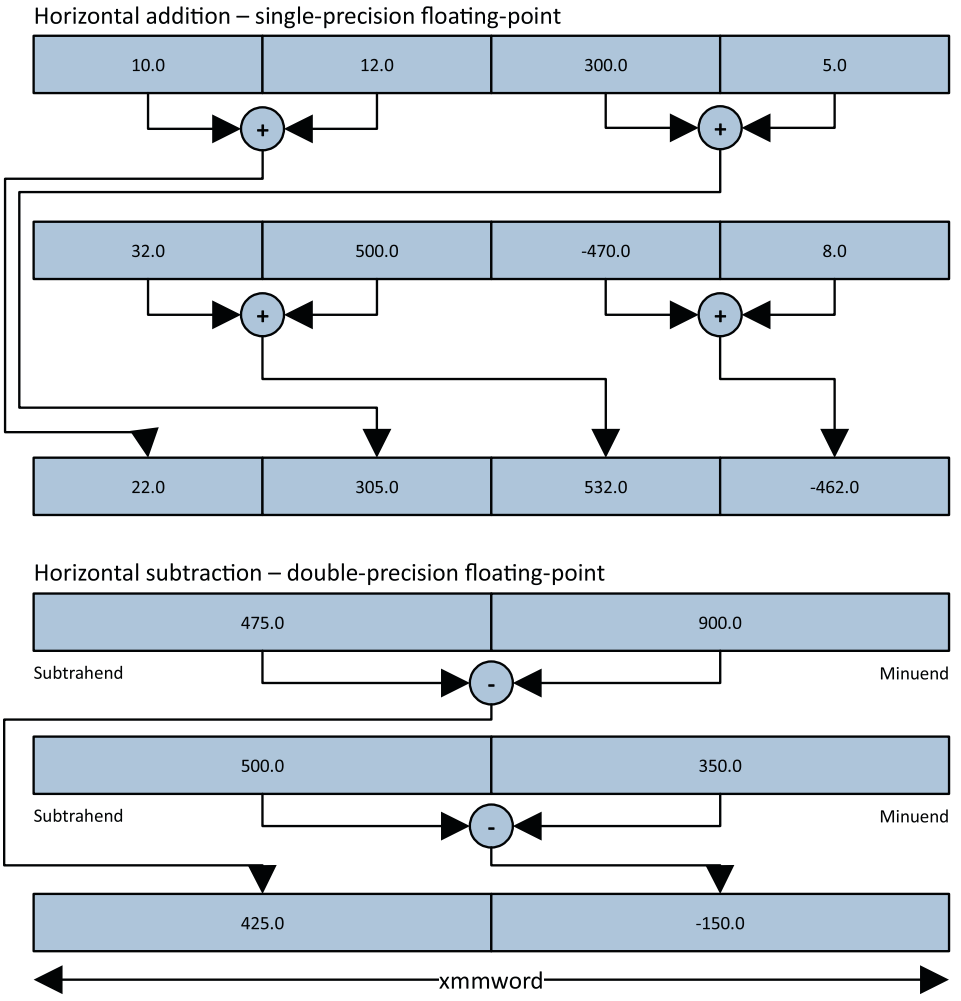

Figure 1-11. *Floating-point horizontal addition and subtraction*

SIMD Data Manipulation Operations

Besides arithmetic calculations, many algorithms frequently employ SIMD data manipulation operations. X86-AVX SIMD data manipulation operations include element compares, shuffles, permutations, blends, conditional moves, broadcasts, size promotions/reductions, and type conversions. You will learn more about these operations in the programming chapters of this book. A few common SIMD data manipulation operations are, however, employed regularly to warrant a few preliminary comments in this chapter.

One indispensable SIMD data manipulation operation is a data compare. Like a SIMD arithmetic calculation, the operations performed during a SIMD compare are carried out simultaneously using all operand elements. However, the results generated by a SIMD compare are different than those produced by an ordinary scalar compare. When performing a scalar compare such as `a > b`, the processor conveys the result using status bits in a flags register (on x86-64 processors, this flags register is named RFLAGS). A SIMD compare is different in that it needs to report the results of multiple compare operations, which means a