# Modern X86 Assembly Language Programming

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

—

*Second Edition*

—


Daniel Kusswurm

# 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

Copyright © 2018 by Daniel Kusswurm

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. 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 editorial@apress.com; for reprint, paperback, or audio rights, please email 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 via the book's product page, located at www.apress.com/9781484240625. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

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

# Contents

About the Author .......... xiii

About the Technical Reviewer .......... xv

Acknowledgments .......... xvii

Introduction .......... xix

■ Chapter 1: X86-64 Core Architecture .......... 1

Historical Overview .......... 1

Data Types .......... 3

Fundamental Data Types .......... 3

Numerical Data Types .......... 4

SIMD Data Types .......... 5

Miscellaneous Data Types .......... 6

Internal Architecture .......... 6

General-Purpose Registers .......... 7

RFLAGS Register .......... 9

Instruction Pointer .......... 10

Instruction Operands .......... 10

Memory Addressing .......... 11

Differences Between x86-64 and x86-32 Programming .......... 13

Invalid Instructions .......... 15

Deprecated Instructions .......... 15

Instruction Set Overview .......... 15

Summary .......... 18

■ Chapter 2: X86-64 Core Programming – Part 1 ........................................ 21
Simple Integer Arithmetic ........................................ 21
Addition and Subtraction ........................................ 22
Logical Operations ........................................ 24
Shift Operations ........................................ 27
Advanced Integer Arithmetic ........................................ 30
Multiplication and Division ........................................ 31
Calculations Using Mixed Types ........................................ 35
Memory Addressing and Condition Codes ........................................ 40
Memory Addressing Modes ........................................ 40
Condition Codes ........................................ 44
Summary ........................................ 49
■ Chapter 3: X86-64 Core Programming – Part 2 ........................................ 51
Arrays ........................................ 51
One-Dimensional Arrays ........................................ 51
Two-Dimensional Arrays ........................................ 58
Structures ........................................ 68
Strings ........................................ 71
Counting Characters ........................................ 71
String Concatenation ........................................ 74
Comparing Arrays ........................................ 79
Array Reversal ........................................ 82
Summary ........................................ 86
■ Chapter 4: Advanced Vector Extensions ........................................ 87
AVX Overview ........................................ 87
SIMD Programming Concepts ........................................ 88
Wraparound vs. Saturated Arithmetic ........................................ 90
AVX Execution Environment ........................................ 91
Register Set ........................................ 91

Data Types ............ 92
Instruction Syntax ............ 93
AVX Scalar Floating-Point ............ 94
Floating-Point Programming Concepts ............ 94
Scalar Floating-Point Register Set ............ 97
Control-Status Register ............ 97
Instruction Set Overview ............ 98
AVX Packed Floating-Point ............ 100
Instruction Set Overview ............ 101
AVX Packed Integer ............ 103
Instruction Set Overview ............ 104
Differences Between x86-AVX and x86-SSE ............ 105
Summary ............ 107
**Chapter 5: AVX Programming – Scalar Floating-Point ............ 109**
Scalar Floating-Point Arithmetic ............ 109
Single-Precision Floating-Point ............ 110
Double-Precision Floating-Point ............ 112
Scalar Floating-Point Compares and Conversions ............ 118
Floating-Point Compares ............ 118
Floating-Point Conversions ............ 128
Scalar Floating-Point Arrays and Matrices ............ 135
Floating-Point Arrays ............ 135
Floating-Point Matrices ............ 138
Calling Convention ............ 143
Basic Stack Frames ............ 144
Using Non-Volatile General-Purpose Registers ............ 148
Using Non-Volatile XMM Registers ............ 153
Macros for Prologs and Epilogs ............ 159
Summary ............ 166

**Chapter 6: AVX Programming – Packed Floating-Point ..................................... 167**

Packed Floating-Point Arithmetic ..................................................................... 167

Packed Floating-Point Compares ...................................................................... 173

Packed Floating-Point Conversions ................................................................... 179

Packed Floating-Point Arrays .......................................................................... 183

Packed Floating-Point Square Roots ............................................................... 184

Packed Floating-Point Array Min-Max ............................................................ 188

Packed Floating-Point Least Squares ............................................................... 193

Packed Floating-Point Matrices ....................................................................... 199

Matrix Transposition ................................................................................... 199

Matrix Multiplication ................................................................................... 207

Summary ....................................................................................................... 214

**Chapter 7: AVX Programming – Packed Integers ............................................ 215**

Packed Integer Addition and Subtraction .......................................................... 215

Packed Integer Shifts ..................................................................................... 221

Packed Integer Multiplication .......................................................................... 226

Packed Integer Image Processing .................................................................... 232

Pixel Minimum-Maximum Values ................................................................... 232

Pixel Mean Intensity .................................................................................... 240

Pixel Conversions ........................................................................................ 246

Image Histograms ........................................................................................ 255

Image Thresholding ..................................................................................... 262

Summary ....................................................................................................... 274

**Chapter 8: Advanced Vector Extensions 2 ....................................................... 277**

AVX2 Execution Environment ......................................................................... 277

AVX2 Packed Floating-Point ........................................................................... 278

AVX2 Packed Integer ..................................................................................... 279

X86 Instruction Set Extensions ........................................................................ 280

Half-Precision Floating-Point ......................................................................... 280

Fused-Multiply-Add (FMA) ... 281
General-Purpose Register Instruction Set Extensions ... 282
Summary ... 283
Chapter 9: AVX2 Programming – Packed Floating-Point ... 285
Packed Floating-Point Arithmetic ... 285
Packed Floating-Point Arrays ... 292
Simple Calculations ... 292
Column Means ... 298
Correlation Coefficient ... 305
Matrix Multiplication and Transposition ... 312
Matrix Inversion ... 320
Blend and Permute Instructions ... 333
Data Gather Instructions ... 339
Summary ... 346
Chapter 10: AVX2 Programming – Packed Integers ... 347
Packed Integer Fundamentals ... 347
Basic Arithmetic ... 347
Pack and Unpack ... 352
Size Promotions ... 358
Packed Integer Image Processing ... 363
Pixel Clipping ... 363
RGB Pixel Min-Max Values ... 369
RGB to Grayscale Conversion ... 376
Summary ... 384
Chapter 11: AVX2 Programming – Extended Instructions ... 385
FMA Programming ... 385
Convolutions ... 385
Scalar FMA ... 388
Packed FMA ... 398

General-Purpose Register Instructions ........ 406
Flagless Multiplication and Shifts ........ 406
Enhanced Bit Manipulation ........ 412
Half-Precision Floating-Point Conversions ........ 415
Summary ........ 419
**Chapter 12: Advanced Vector Extensions 512 ........ 421**
AVX-512 Overview ........ 421
AVX-512 Execution Environment ........ 422
Register Sets ........ 422
Data Types ........ 423
Instruction Syntax ........ 424
Instruction Set Overview ........ 427
AVX512F ........ 427
AVX512CD ........ 430
AVX512BW ........ 430
AVX512DQ ........ 431
Opmask Registers ........ 431
Summary ........ 432
**Chapter 13: AVX-512 Programming – Floating-Point ........ 433**
Scalar Floating-Point ........ 433
Merge Masking ........ 433
Zero Masking ........ 437
Instruction-Level Rounding ........ 440
Packed Floating-Point ........ 444
Packed Floating-Point Arithmetic ........ 445
Packed Floating-Point Compares ........ 452
Packed Floating-Point Column Means ........ 457
Vector Cross Products ........ 466

Matrix-Vector Multiplication ....... 476
Convolutions ....... 485
Summary ....... 489
**Chapter 14: AVX-512 Programming – Packed Integers ....... 491**
Basic Arithmetic ....... 491
Image Processing ....... 497
Pixel Conversions ....... 497
Image Thresholding ....... 504
Image Statistics ....... 510
RGB to Grayscale Conversion ....... 520
Summary ....... 527
**Chapter 15: Optimization Strategies and Techniques ....... 529**
Processor Microarchitecture ....... 529
Processor Architecture Overview ....... 530
Microarchitecture Pipeline Functionality ....... 531
Execution Engine ....... 532
Optimizing Assembly Language Code ....... 534
Basic Techniques ....... 534
Floating-Point Arithmetic ....... 536
Program Branches ....... 536
Data Alignment ....... 538
SIMD Techniques ....... 539
Summary ....... 540
**Chapter 16: Advanced Programming ....... 541**
CPUID Instruction ....... 541
Non-Temporal Memory Stores ....... 557
Data Prefetch ....... 562
Multiple Threads ....... 570
Summary ....... 584

■ Appendix A ............................................................ 585
Software Utilities for x86 Processors ............................................................ 585
Visual Studio ............................................................ 585
Running a Source Code Example ............................................................ 586
Creating a Visual Studio C++ Project ............................................................ 586
References ............................................................ 594
X86 Programming Reference Manuals ............................................................ 594
X86 Programming and Microarchitecture References ............................................................ 595
Ancillary Resources ............................................................ 596
Algorithm References ............................................................ 597
C++ References ............................................................ 598
Index ............................................................ 599

# 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.

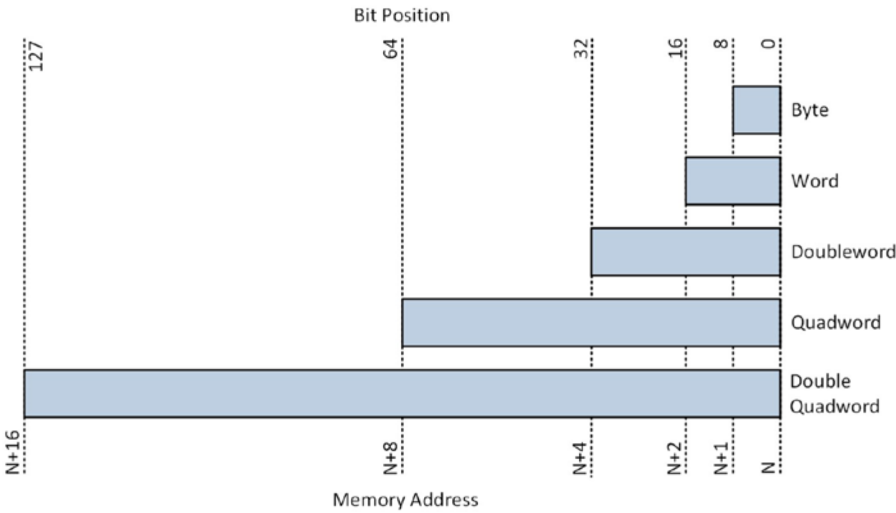



***Figure 1-1.*** *Bit-numbering and byte-ordering for fundamental data types*

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.

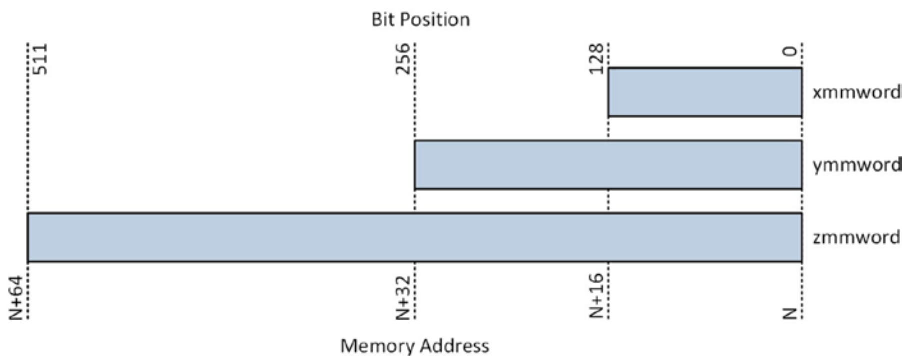



***Figure 1-2.*** *SIMD data types*

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.

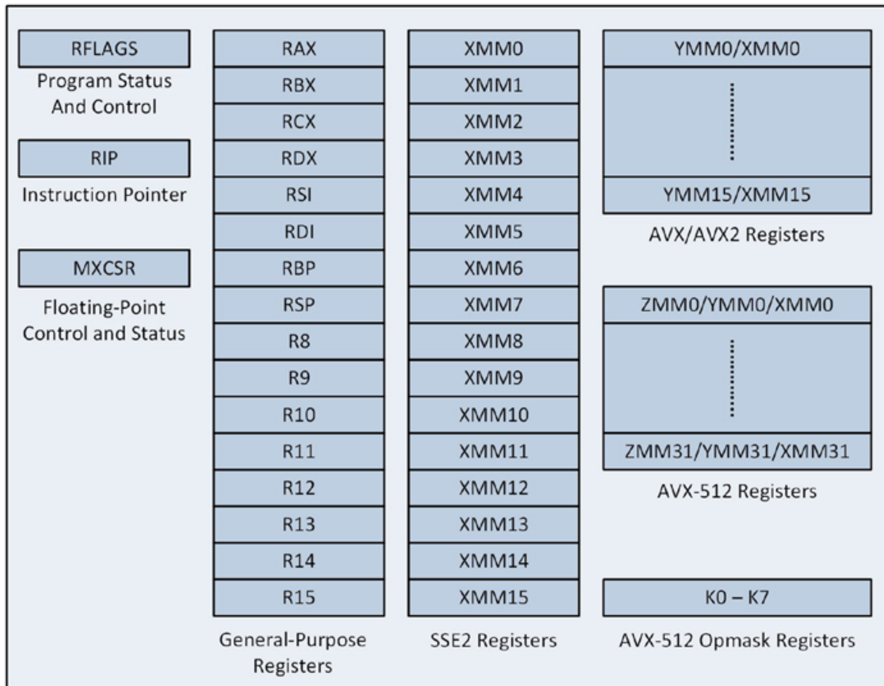



***Figure 1-3.*** *X86-64 processor internal architecture*

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.

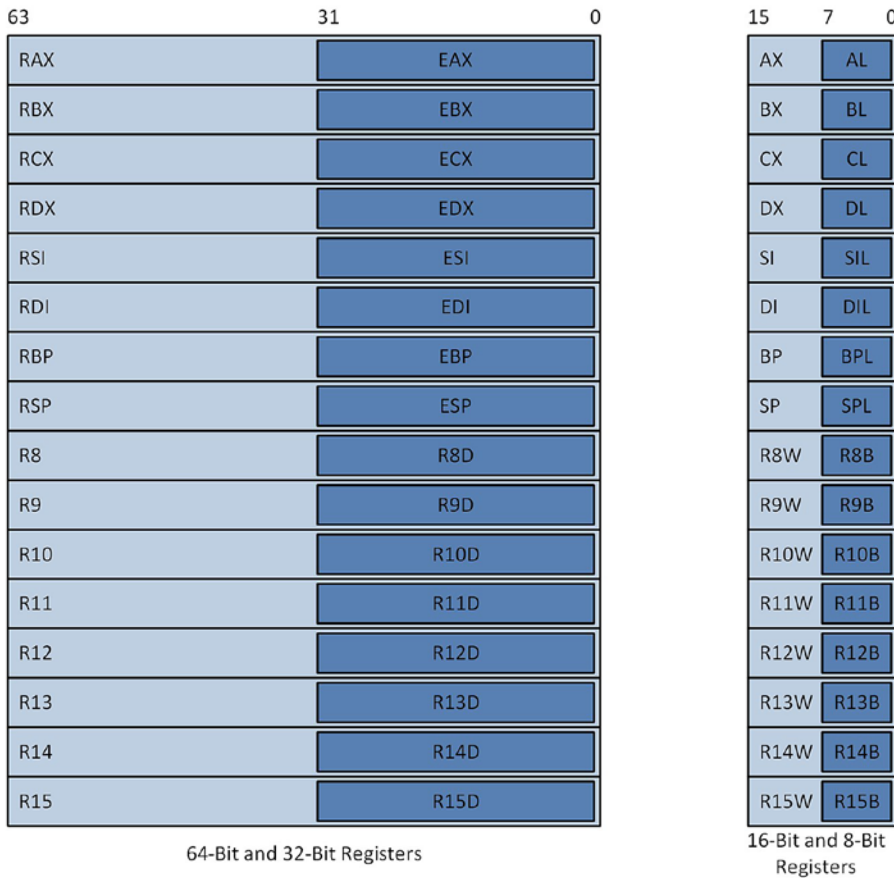



***Figure 1-4.*** *X86-64 general-purpose registers*

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.

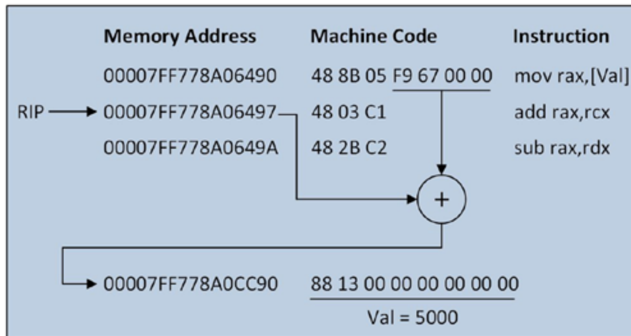



***Figure 1-5.*** *RIP-relative effective address calculation*

One minor constraint of RIP-relative addressing is that the target operand must reside with a ± 2 GB address window of the value that's contained in register RIP. For most programs, this limitation is rarely a concern. The calculation of a RIP-relative displacement value is automatically determined by the assembler during code generation. This means that you can use a `mov rax,[Val]` or similar instructions without having to worry about the details of the displacement value calculation.

# Differences Between x86-64 and x86-32 Programming

There are some important differences between x86-64 and x86-32 assembly language programming. If you are learning x86 assembly language programming for the first time, you can either skim or skip this section since it discusses concepts that aren't fully explained until later in this book.

Most existing x86-32 instructions have an x86-64 equivalent instruction that enables a function to exploit 64-bit wide addresses and operands. X86-64 functions can also perform calculations using instructions that manipulate 8-bit, 16-bit, or 32-bit registers and operands. Except for the `mov` instruction, the maximum size of an x86-64 mode immediate value is 32 bits. If an instruction manipulates a 64-bit wide register or memory operand, any specified 32-bit immediate value is signed-extended to 64 bits prior to its use.

Table 1-7 contains some examples of x86-64 instructions using various operand sizes. Note that the memory operands in these example instructions are referenced using 64-bit registers, which is required in order to access the entire 64-bit linear address space. While it is possible in x86-64 mode to reference a memory operand using a 32-bit register (e.g., `mov r10,[eax]`), the location of the operand must reside in the low 4 GB portion of the 64-bit effective address space. Using 32-bit registers to access memory operands in x86-64 mode is not recommended since it introduces unnecessary and potentially dangerous code obfuscations. It also complicates software testing and debugging.

***Table 1-7.*** *Examples of x86-64 Instructions Using Various Operand Sizes*

| 8-Bit | 16-Bit | 32-Bit | 64-Bit |
|---|---|---|---|
| `add al,bl` | `add ax,bx` | `add eax,ebx` | `add rax,rbx` |
| `cmp dl,[r15]` | `cmp dx,[r15]` | `cmp edx,[r15]` | `cmp rdx,[r15]` |
| `mul r10b` | `mul r10w` | `mul r10d` | `mul r10` |
| `or [r8+rdi],al` | `or [r8+rdi],ax` | `or [r8+rdi],eax` | `or [r8+rdi],rax` |
| `shl r9b,cl` | `shl r9w,cl` | `shl r9d,cl` | `shl r9,cl` |

The aforementioned immediate value size limitation warrants some extra discussion since it sometimes affects the instruction sequences that a program must use to carry out certain operations. Figure 1-6 contains a few examples of instructions that use a 64-bit register with an immediate operand. In the first example, the `mov rax,100` instruction loads an immediate value into the RAX register. Note that the machine code uses only 32 bits to encode the immediate value `100`, which is underlined. This value is signed extended to 64 bits and saved in RAX. The `add rax,200` instruction that follows also sign extends its immediate value prior to performing the addition. The next example opens with a `mov rcx,-2000` instruction that loads a negative immediate value into RCX. The machine code for this instruction also uses 32 bits to encode the immediate value `-2000`, which is signed extended to 64 bits and saved in RCX. The subsequent `add rcx,1000` instruction yields a 64-bit result of `-1000`.

| Machine Code | Instruction | DesOp Result |
|---|---|---|
| 48 C7 C0 64 00 00 00 | mov rax,100 | 0000000000000064h |
| 48 05 C0 C8 00 00 00 | add rax,200 | 000000000000012Ch |
| 48 C7 C1 30 F8 FF FF | mov rcx,-2000 | FFFFFFFFFFFFF830h |
| 48 81 C1 E8 03 00 00 | add rcx,1000 | FFFFFFFFFFFFFC18h |
| 48 C7 C2 FF 00 00 00 | mov rdx,0ffh | 00000000000000FFh |
| 48 81 CA 00 00 00 80 | or rdx,80000000h | FFFFFFFF800000FFh |
| 48 C7 C2 FF 00 00 00 | mov rdx,0ffh | 00000000000000FFh |
| 49 B8 00 00 00 80 00 00 00 00 | mov r8,80000000h | 0000000080000000h |
| 49 0B D0 | or rdx,r8 | 00000000800000FFh |

***Figure 1-6.*** *Using 64-bit registers with immediate operands*

The third example employs a `mov rdx,0ffh` instruction to initialize register RDX. This is followed by an `or rdx,80000000h` instruction that sign extends the immediate value `0x80000000` to `0xFFFFFFFF80000000`, and then performs a bitwise inclusive OR operation. The value that's shown for RDX is almost certainly not the intended result. The final example illustrates how to carry out an operation that requires a 64-bit immediate value. A `mov r8,80000000h` instruction loads the 64-bit value `0x0000000080000000` into R8. As mentioned earlier in this section, the `mov` instruction is the only instruction that supports 64-bit immediate operands. Execution of the ensuing `or rdx,r8` instruction yields the expected value.

The 32-bit size limitation for immediate values also applies to `jmp` and `call` instructions that specify relative-displacement targets. In these cases, the target (or location) of a `jmp` or `call` instruction must reside with a ± 2 GB address window of the current RIP register. Targets whose relative displacements exceed this window can only be accessed using a `jmp` or `call` instruction that employs an indirect operand (e.g., `jmp qword ptr [FuncPtr]` or `call rax`). Like RIP-relative addressing, the size limitations described in this paragraph are unlikely to present significant obstacles for most assembly language functions.

Another difference between x86-32 and x86-64 assembly language programming is the effect that some instructions have on the upper 32 bits of a 64-bit general-purpose register. When using instructions that manipulate 32-bit registers and operands, the high-order 32 bits of the corresponding 64-bit general-purpose register are zeroed during execution. For example, assume that register RAX contains the value `0x8000000000000000`. Execution of the instruction `add eax,10` generates a result of `0x000000000000000A` in RAX. However, when working with 8-bit or 16-bit registers and operands, the upper 56 or 48 bits of the

corresponding 64-bit general-purpose register are not modified. Assuming again that if RAX contains `0x8000000000000000`, execution of the instructions `add al,20` or `add ax,40` would yield RAX values of `0x8000000000000014` or `0x8000000000000028`, respectively.

The x86-64 platform imposes some restrictions on the use of legacy registers AH, BH, CH, and DH. These registers cannot be used with instructions that also reference one of the new 8-bit registers (i.e., SIL, DIL, BPL, SPL, and R8B - 15B). Existing x86-32 instructions such as `mov ah,bl` and `add dh,bl` are still allowed in x86-64 programs. However, the instructions `mov ah,r8b` and `add dh,r8b` are invalid.

## Invalid Instructions

A handful of rarely used x86-32 instructions are cannot be used in x86-64 programs. Table 1-8 lists these instructions. Somewhat surprisingly, early-generation x86-64 processors did not support the `lahf` and `sahf` instructions in x86-64 mode (they still worked in x86-32 mode). Fortunately, these instructions were reinstated, and should be available in most AMD and Intel processors marketed since 2006. A program can confirm processor support for the `lahf` and `sahf` instructions in x86-64 mode by testing the `cupid` feature flag LAHF-SAHF.

***Table 1-8.*** *X86-64 Mode Invalid Instructions*

| Mnemonic | Name |
|---|---|
| `aaa` | ASCII Adjust After Addition |
| `aad` | ASCII Adjust After Division |
| `aam` | ASCII Adjust After Multiplication |
| `aas` | ASCII Adjust After Subtraction |
| `bound` | Check Array Index Against Bounds |
| `daa` | Decimal Adjust After Addition |
| `das` | Decimal Adjust After Subtraction |
| `into` | Generate interrupt if RFLAGS.OF Equals 1 |
| `pop[a\|ad]` | Pop all General-Purpose Registers |
| `push[a\|ad]` | Push all General-Purpose Registers |

## Deprecated Instructions

Processors that support the x86-64 instruction set also include the computational resources of SSE2. This means that x86-64 programs can safely use the packed integer instructions of SSE2 instead of MMX. It also means that x86-64 programs can use SSE2's (or AVX's if available) scalar floating-point instructions instead of x87 FPU instructions. X86-64 programs can still take advantage of the MMX and x87 FPU instruction sets, and such use might be warranted when migrating x86-32 legacy code to the x64-64 platform. For new x86-64 software development, however, using the MMX and x87 FPU instruction sets is not recommended.

# Instruction Set Overview

Table 1-9 lists in alphabetical order the core x86-64 instructions that are frequently used in assembly language functions. For each instruction mnemonic, there is a deliberately succinct description since comprehensive details of each instruction including execution particulars, valid operands, affected flags,

and exceptions are readily available in reference manuals published by AMD and Intel. Appendix A contains a list of these manuals. The programming examples in Chapters 2 and 3 also contain additional information regarding proper use of these instructions.

Note that Table 1-9 uses brackets in the mnemonics column to represent distinct variations of a common instruction. For example, `bs[f|r]` denotes the distinct instructions `bsf` (Bit Scan Forward) and `bsr` (Bit Scan Reverse).

***Table 1-9.*** *Overview of Core X86-64 Instructions*

| Mnemonic | Instruction Name |
|---|---|
| adc | Integer addition with carry |
| add | Integer addition |
| and | Bitwise AND |
| bs[f\|r] | Bit scan forward, bit scan reverse |
| b[t\|tr\|ts] | Bit test; Bit test and reset; Bit test and set |
| call | Call procedure |
| cld | Clear direction flag (RFLAGS.DF) |
| cmovcc | Conditional move |
| cmp | Compare operands |
| cmps[b\|w\|d\|q] | Compare string operands |
| cupid | Query CPU identification and feature information |
| c[wd\|dq\|do] | Convert operand |
| dec | Decrement operand by 1 |
| div | Unsigned integer division |
| idiv | Signed integer division |
| imul | Signed integer multiplication |
| inc | Increment operand by 1 |
| jcc | Conditional jump |
| jmp | Unconditional jump |
| lahf | Load status flags into register AH |
| lea | Load effective address |
| lods[b\|w\|d\|q] | Load string operand |
| mov | Move data |
| mov[sx\|sxd] | Move integer with sign extension |
| movzx | Move integer with zero extension |
| mul | Unsigned integer multiplication |
| neg | Two's complement negation |
| not | One's complement negation |
| or | Bitwise inclusive OR |

(*continued*)

***Table 1-9.*** (*continued*)

| Mnemonic | Instruction Name |
|---|---|
| `pop` | Pop top-of-stack value to operand |
| `popfq` | Pop top-of-stack value to RFLAGS |
| `push` | Push operand onto stack |
| `pushfq` | Push RFLAGS onto stack |
| `rc[l\|r]` | Rotate left with RFLAGS.CF; Rotate right with RFLAGS.CF |
| `ret` | Return from procedure |
| `re[p\|pe\|pz\|pne\|pnz]` | Repeat string operation (instruction prefix) |
| `ro[l\|r]` | Rotate left; Rotate right |
| `sahf` | Store AH into status flags |
| `sar` | Shift arithmetic right |
| `setcc` | Set byte on condition |
| `sh[l\|r]` | Shift logical left; Shift logical right |
| `sbb` | Integer subtraction with borrow |
| `std` | Set direction flag (RFLAGS.DF) |
| `stos[b\|w\|d\|q]` | Store string value |
| `test` | Test operand (sets status flags) |
| `xchg` | Exchange source and destination operand values |
| `xor` | Bitwise exclusive OR |

Most arithmetic and logical instructions update one or more of the status flags in the RFLAGS register. As discussed earlier in this chapter, the status flags provide additional information about the results of an operation. The `jcc`, `cmovcc`, and `setcc` instructions use what are called condition codes to test the status flags either individually or in multiple-flag combinations. Table 1-10 lists the condition codes, mnemonic suffixes, and the corresponding RFLAGS tested by these instructions.

***Table 1-10.*** *Condition Codes, Mnemonic Suffixes, and Test Conditions*

| Condition Code | Mnemonic Suffix | RFLAGS Test Condition |
|---|---|---|
| Above<br>Neither below nor equal | A<br>NBE | `CF == 0 && ZF == 0` |
| Above or equal<br>Not below | AE<br>NB | `CF == 0` |
| Below<br>Neither above nor equal | B<br>NAE | `CF == 1` |
| Below or equal<br>Not above | BE<br>NA | `CF == 1 \|\| ZF == 1` |

(*continued*)

***Table 1-10.*** (*continued*)

| Condition Code | Mnemonic Suffix | RFLAGS Test Condition |
|---|---|---|
| Equal<br>Zero | E<br>Z | `ZF == 1` |
| Not equal<br>Not zero | NE<br>NZ | `ZF == 0` |
| Greater<br>Neither less nor equal | G<br>NLE | `ZF == 0 && SF == OF` |
| Greater or equal<br>Not less | GE<br>NL | `SF == OF` |
| Less<br>Neither greater nor equal | L<br>NGE | `SF != OF` |
| Less or equal<br>Not greater | LE<br>NG | `ZF == 1 \|\| SF != OF` |
| Sign | S | `SF == 1` |
| Not sign | NS | `SF == 0` |
| Carry | C | `CF == 1` |
| Not carry | NC | `CF == 0` |
| Overflow | O | `OF == 1` |
| Not overflow | NO | `OF == 0` |
| Parity<br>Parity even | P<br>PE | `PF == 1` |
| Not parity<br>Parity odd | NP<br>PO | `PF == 0` |

The alternate forms of many Table 1-10 mnemonics are defined to provide algorithmic flexibility or improve program readability. When using one of the aforementioned conditional instructions in source code, condition-codes containing the words “above” and “below” are employed for unsigned-integer operands while the words “greater” and “less” are used for signed-integer operands. If the contents of Table 1-10 seem a little confusing or abstract, don’t worry. You'll see a plethora of condition code examples in subsequent chapters of this book.

# Summary

Here are the key learning points of Chapter 1:

- The fundamental data types of the x86-64 platform include bytes, words, doublewords, quadwords, and double quadwords. Intrinsic programming language data types such as characters, text strings, integers, and floating-point values are derived from the fundamental data types.
- The x86-64 execution unit includes 16 64-bit general-purpose registers that are used to perform arithmetic, logical, and data transfer operations using 8-bit, 16-bit, 32-bit and 64-bit operands.

- The x86-64 execution unit includes 16 128-bit XMM registers that can be used to perform scalar floating-point arithmetic using single-precision or double-precision values. These registers can also be employed to carry out SIMD operations using packed integers or packed floating-point values.
- Most x86-64 assembly language instructions can be used with the following explicit operand types: immediate, register, and memory. Some instructions employ implicit registers as their operands.
- An operand in memory can be referenced using a variety of addressing modes that include one or more of the following components: fixed displacement, base register, index register, and/or scale factor.
- Most arithmetic and logical instructions update one or more of the status flags in the RFLAGS register. These flags can be tested to alter program flow or conditionally assign values to variables.

CHAPTER 2

# X86-64 Core Programming – Part 1

In the previous chapter, you learned about the fundamentals of the x86-64 platform including its data types, register sets, memory addressing modes, and the core instruction set. In this chapter, you learn how to code basic x86-64 assembly language functions that are callable from C++. You also learn about the semantics and syntax of an x86-64 assembly language source code file. The sample source code and accompanying remarks of this chapter are intended to complement the instructive material presented in Chapter 1.

The content of Chapter 2 is organized as follows. The first section describes how to code functions that perform simple integer arithmetic such as addition and subtraction. You also learn the basics of passing arguments and return values between functions written in C++ and x86-64 assembly language. The next section highlights additional arithmetic instructions including integer multiplication and division. In the final section, you learn how to reference operands in memory and use conditional jumps and conditional moves.

It should be noted that the primary purpose of the sample code presented in this chapter is to elucidate proper use of the x86-64 instruction set and basic assembly language programming techniques. All of the assembly language code is straightforward, but not necessarily optimal since understanding optimized assembly language code can be challenging especially for beginners. The sample code that's discussed in later chapters places more emphasis on efficient coding techniques. Chapter 15 also examines techniques that you can use to improve the efficiency of your assembly language code.

## Simple Integer Arithmetic

In this section, you learn the basics of x86-64 assembly language programming. It begins with a simple program that demonstrates how to perform integer addition and subtraction. This is followed by an example program that illustrates use of the logical instructions `and`, `or`, and `xor`. The final program describes how to execute shift operations. All three programs illustrate passing argument and return values between a C++ and assembly language function. They also show how to employ commonly-used assembler directives.

As mentioned in the Introduction, all of the sample code discussed in this book was created using Microsoft's Visual C++ and Macro Assembler (MASM), which are included with Visual Studio. Before taking a look at the first code example, a few instructive comments about these development tools may be helpful. Visual Studio uses entities called solutions and projects to help simplify application development. A solution is a collection of one or more projects that are used to build an application. Projects are container objects that help organize an application's files including (but not limited to) source code, resources, icons, bitmaps, HTML, and XML. A Visual Studio project is usually created for each buildable component (e.g., executable file, dynamic-linked library, static library, etc.) of an application. You can open and load a chapter's sample programs into the Visual Studio development environment by double-clicking on its solution (.sln) file. Appendix A contain additional information regarding the use of Visual C++ and MASM.

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_2

■ **Note** All of the source code examples in this book include one or more functions written in x86-64 assembly language plus some C++ code that demonstrates how to invoke the assembly language code. The C++ code also contains ancillary functions that perform required initializations and display results. For each source code example, a single listing that includes both the C++ and assembly language source code is used in order to minimize the number of listing references in the main text. The actual source code uses separate files for the C++ (.cpp) and assembly language (.asm) code.

## Addition and Subtraction

The first source code example of this chapter is called Ch02_01. This example demonstrates how to use the x86-64 assembly language instructions add (Integer Add) and sub (Integer Subtract). It also illustrates some basic assembly language programming concepts including argument passing, returning values, and how to use a few MASM assembler directives. Listing 2-1 shows the source code for example Ch02_01.

***Listing 2-1.*** Example Ch02_01

```
//------------------------------------------------
//               Ch02_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>

using namespace std;

extern "C" int IntegerAddSub_(int a, int b, int c, int d);

static void PrintResult(const char* msg, int a, int b, int c, int d, int result)
{
    const char nl = '\n';

    cout << msg << nl;
    cout << "a = " << a << nl;
    cout << "b = " << b << nl;
    cout << "c = " << c << nl;
    cout << "d = " << d << nl;
    cout << "result = " << result << nl;
    cout << nl;
}

int main()
{
    int a, b, c, d, result;

    a = 10; b = 20; c = 30; d = 18;
    result = IntegerAddSub_(a, b, c, d);
    PrintResult("Test 1", a, b, c, d, result);
```

```
    a = 101; b = 34; c = -190; d = 25;
    result = IntegerAddSub_(a, b, c, d);
    PrintResult("Test 2", a, b, c, d, result);

    return 0;
}

;-------------------------------------------------
;               Ch02_01.asm
;-------------------------------------------------

; extern "C" int IntegerAddSub_(int a, int b, int c, int d);

        .code
IntegerAddSub_ proc

; Calculate a + b + c - d
        mov eax,ecx                         ;eax = a
        add eax,edx                         ;eax = a + b
        add eax,r8d                         ;eax = a + b + c
        sub eax,r9d                         ;eax = a + b + c - d

        ret                                 ;return result to caller
IntegerAddSub_ endp
        end
```

The C++ code in Listing 2-1 is mostly straightforward but includes a few lines that warrant some explanatory comments. The `#include "stdafx.h"` statement specifies a project-specific header file that contains references to frequently used system items. Visual Studio automatically generates this file whenever a new C++ project is created. The line `extern "C" int IntegerAddSub_(int a, int b, int c, int d)` is a declaration statement that defines the parameters and return value for the x86-64 assembly language function `IntegerAddSub_` (all assembly language function names and public variables used in this book include a trailing underscore for easier recognition). The declaration statement's `"C"` modifier instructs the C++ compiler to use C-style naming for function `IntegerAddSub_` instead of a C++ decorated name (a C++ decorated name includes extra characters that help support function overloading). It also notifies the compiler to use C-style linkage for the specified function.

The C++ function `main` contains the code that calls the assembly language function `IntegerAddSub_`. This function requires four arguments of type `int` and returns a single `int` value. Like many programming languages, Visual C++ uses a combination of processor registers and the stack to pass argument values to a function. In the current example, the C++ compiler generates code that loads the values of `a`, `b`, `c`, and `d` into registers ECX, EDX, R8D, and R9D, respectively, prior to calling function `IntegerAddSub_`.

In Listing 2-1 the x86-64 assembly language code for example `Ch02_01` is shown immediately after the C++ function `main`. The first thing to notice are the lines that begin with a semicolon. These are comments lines. MASM treats any text that follows a semicolon as comment text. The `.code` statement is a MASM directive that defines the start of an assembly language code section. A MASM directive is a statement that instructs the assembler how to perform certain actions. You'll learn how to use additional directives throughout this book.

The `IntegerAddSub_ proc` statement defines the start of the assembly language function. Toward the end of Listing 2-1, the `IntegerAddSub_ endp` statement marks the end of the function. Like the `.code` line, the `proc` and `endp` statements are not executable instructions but assembler directives that signify the start and end of an assembly language function. The final `end` statement is a required assembler directive that indicates the completion of statements for the assembly language file. The assembler ignores any text that appears after the `end` directive.

The assembly language function IntegerAddSub_ calculates a + b + c - d and returns this value to the calling C++ function. It begins with a mov eax,ecx (Move) instruction that copies the value a from ECX into EAX. Note that the contents of ECX are not altered by the mov instruction. Following execution of this mov instruction, registers EAX and ECX both contain the value a. The add eax,edx instruction adds the values in registers EAX and EDX. It then saves the sum (or a + b) in register EAX. Like the previous mov instruction, the contents of register EDX are not modified by the add instruction. The next instruction, add eax,r8d computes a + b + c. This is followed by a sub eax,r9d instruction that calculates the final value a + b + c - d.

An x86-64 assembly language function must use register EAX to return a single 32-bit integer (or C++ int) value to its calling function. In the current example, no additional instructions are necessary to achieve this requirement since EAX already contains the correct return value. The final ret (Return from Procedure) instruction transfers control back to the calling function main, which displays the result. Here's the output for example Ch02_01.

```
Test 1
a = 10
b = 20
c = 30
d = 18
result = 42

Test 2
a = 101
b = 34
c = -190
d = 25
result = -80
```

## Logical Operations

The next source code example is called Ch02_02. This example illustrates use of the x86-64 instructions and (Logical AND), or (Logical Inclusive OR), and xor (Logical Exclusive OR). It also shows how to access a C++ global variable from an assembly language function. Listing 2-2 shows the source code for Example Ch02_02.

***Listing 2-2.*** Example Ch02_02

```
//------------------------------------------------
//               Ch02_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" unsigned int IntegerLogical_(unsigned int a, unsigned int b, unsigned int c,
unsigned int d);

extern "C" unsigned int g_Val1 = 0;
```

```
unsigned int IntegerLogicalCpp(unsigned int a, unsigned int b, unsigned int c, unsigned int d)
{
    // Calculate (((a & b) | c ) ^ d) + g_Val1
    unsigned int t1 = a & b;
    unsigned int t2 = t1 | c;
    unsigned int t3 = t2 ^ d;
    unsigned int result = t3 + g_Val1;

    return result;
}

void PrintResult(const char* s, unsigned int a, unsigned int b, unsigned int c, unsigned int
d, unsigned val1, unsigned int r1, unsigned int r2)
{
    const int w = 8;
    const char nl = '\n';

    cout << s << nl;
    cout << setfill('0');
    cout << "a =    0x" << hex << setw(w) << a << " (" << dec << a << ")" << nl;
    cout << "b =    0x" << hex << setw(w) << b << " (" << dec << b << ")" << nl;
    cout << "c =    0x" << hex << setw(w) << c << " (" << dec << c << ")" << nl;
    cout << "d =    0x" << hex << setw(w) << d << " (" << dec << d << ")" << nl;
    cout << "val1 = 0x" << hex << setw(w) << val1 << " (" << dec << val1<< ")" << nl;
    cout << "r1 =   0x" << hex << setw(w) << r1 << " (" << dec << r1 << ")" << nl;
    cout << "r2 =   0x" << hex << setw(w) << r2 << " (" << dec << r2 << ")" << nl;
    cout << nl;

    if (r1 != r2)
        cout << "Compare failed" << nl;
}

int main()
{
    unsigned int a, b, c, d, r1, r2 = 0;

    a = 0x00223344;
    b = 0x00775544;
    c = 0x00555555;
    d = 0x00998877;
    g_Val1 = 7;
    r1 = IntegerLogicalCpp(a, b, c, d);
    r2 = IntegerLogical_(a, b, c, d);
    PrintResult("Test 1", a, b, c, d, g_Val1, r1, r2);

    a = 0x70987655;
    b = 0x55555555;
    c = 0xAAAAAAAA;
    d = 0x12345678;
    g_Val1 = 23;
    r1 = IntegerLogicalCpp(a, b, c, d);
```

```
    r2 = IntegerLogical_(a, b, c, d);
    PrintResult("Test 2", a, b, c, d, g_Val1, r1, r2);

    return 0;
}

;-------------------------------------------------
;               Ch02_02.asm
;-------------------------------------------------

; extern "C" unsigned int IntegerLogical_(unsigned int a, unsigned int b, unsigned int c,
unsigned int d);

        extern g_Val1:dword                 ;external doubleword (32-bit) value

        .code
IntegerLogical_ proc

; Calculate (((a & b) | c ) ^ d) + g_Val1
        and ecx,edx                         ;ecx = a & b
        or ecx,r8d                          ;ecx = (a & b) | c
        xor ecx,r9d                         ;ecx = ((a & b) | c) ^ d
        add ecx,[g_Val1]                    ;ecx = (((a & b) | c) ^ d) + g_Val1

        mov eax,ecx                         ;eax = final result
        ret                                 ;return to caller
IntegerLogical_ endp
        end
```

Similar to what you saw in the first example, the declaration of assembly language function IntegerLogical_ uses the "C" modifier to instruct the C++ compiler not to generate a decorated name for this function. Omitting this modifier would result in a link error during program build. (If the "C" modifier is omitted from the current example, Visual C++ 2017 uses the decorated function name ?IntegerLogical_@@YAIIIII@Z instead of IntegerLogical_. Decorated names are derived using the function's argument types, and these names are compiler specific.) Function IntegerLogical_ requires four unsigned int arguments and returns a single unsigned int result. Immediately following the declaration of function IntegerLogical_ is the definition of a global unsigned int variable named g_Val1. This variable is defined to demonstrate how to access a global value from an assembly language function. Like function declarations, use of the "C" modifier for g_Val1 instructs the compiler to use C-style naming instead of a decorated C++ name.

The definition of function IntegerLogicalCpp follows next in the C++ source code. The reason for defining this function is to provide a simple method for determining whether or not the corresponding x86-64 assembly language function IntegerLogical_ calculates the correct result. While overkill for this particular example, coding complex functions using both C++ and assembly language is often helpful for software test and debugging purposes. The function main in Listing 2-2 includes code that calls both IntegerLogicalCpp and IntegerLogical_. It also calls the function PrintResult to display the results.

In Listing 2-2 the x86-64 assembly language code for example Ch02_02 follows the C++ function main. The first assembly language source code statement, extern g_Val1:dword, is the MASM equivalent of the corresponding declaration for g_Val1 that's used in the C++ code. In this instance, the extern directive notifies the assembler that storage space for the variable g_Val1 is defined in another module, and the dword directive indicates that g_Val1 is a doubleword (or 32-bit) unsigned value.

Similar to the example in the previous section, the arguments a, b, c, and d are passed to function IntegerLogical_ using registers ECX, EDX, R8D, and R9D. The and ecx,edx instruction performs a bitwise AND operation using the values in registers ECX and EDX, and saves the result to register ECX. The or ecx,r8d and xor ecx,r9d instructions carry out bitwise inclusive OR and exclusive OR operations, respectively. The add ecx,[g_Val1] instruction adds the contents of register ECX and the value of global variable g_Val1, and saves the resultant sum to register ECX. A mov eax,ecx copies the final result to register EAX so that it can be passed back to the calling function. Here's the output for example Ch02_02.

```
Test 1
a =    0x00223344 (2241348)
b =    0x00775544 (7820612)
c =    0x00555555 (5592405)
d =    0x00998877 (10061943)
val1 = 0x00000007 (7)
r1 =   0x00eedd29 (15654185)
r2 =   0x00eedd29 (15654185)

Test 2
a =    0x70987655 (1889039957)
b =    0x55555555 (1431655765)
c =    0xaaaaaaaa (2863311530)
d =    0x12345678 (305419896)
val1 = 0x00000017 (23)
r1 =   0xe88ea89e (3901663390)
r2 =   0xe88ea89e (3901663390)
```

## Shift Operations

The last source code example of this section, which is similar in form to the previous two examples, demonstrates use of the shl (Shift Logical Left) and shr (Shift Logical Right) instructions. It also illustrates use of a few more frequently used instructions including cmp (Compare), ja (Jump if Above), and xchg (Exchange). Listing 2-3 shows the C++ and assembly language source code for example Ch02_03.

***Listing 2-3.*** Example Ch02_03

```
//------------------------------------------------
//               Ch02_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <bitset>

using namespace std;

extern "C" int IntegerShift_(unsigned int a, unsigned int count, unsigned int* a_shl,
unsigned int* a_shr);
```

```
static void PrintResult(const char* s, int rc, unsigned int a, unsigned int count, unsigned
int a_shl, unsigned int a_shr)
{
    bitset<32> a_bs(a);
    bitset<32> a_shl_bs(a_shl);
    bitset<32> a_shr_bs(a_shr);
    const int w = 10;
    const char nl = '\n';

    cout << s << '\n';
    cout << "count =" << setw(w) << count << nl;
    cout << "a =    " << setw(w) << a << " (0b" << a_bs << ")" << nl;

    if (rc == 0)
        cout << "Invalid shift count" << nl;
    else
    {
        cout << "shl =  " << setw(w) << a_shl << " (0b" << a_shl_bs << ")" << nl;
        cout << "shr =  " << setw(w) << a_shr << " (0b" << a_shr_bs << ")" << nl;
    }

    cout << nl;
}

int main()
{
    int rc;
    unsigned int a, count, a_shl, a_shr;

    a = 3119;
    count = 6;
    rc = IntegerShift_(a, count, &a_shl, &a_shr);
    PrintResult("Test 1", rc, a, count, a_shl, a_shr);

    a = 0x00800080;
    count = 4;
    rc = IntegerShift_(a, count, &a_shl, &a_shr);
    PrintResult("Test 2", rc, a, count, a_shl, a_shr);

    a = 0x80000001;
    count = 31;
    rc = IntegerShift_(a, count, &a_shl, &a_shr);
    PrintResult("Test 3", rc, a, count, a_shl, a_shr);

    a = 0x55555555;
    count = 32;
    rc = IntegerShift_(a, count, &a_shl, &a_shr);
    PrintResult("Test 4", rc, a, count, a_shl, a_shr);

    return 0;
}
```

```
;-------------------------------------------------
;               Ch02_03.asm
;-------------------------------------------------

;
; extern "C" int IntegerShift_(unsigned int a, unsigned int count, unsigned int* a_shl,
  unsigned int* a_shr);
;
; Returns:      0 = error (count >= 32), 1 = success
;

        .code
IntegerShift_ proc
        xor eax,eax                         ;set return code in case of error
        cmp edx,31                          ;compare count against 31
        ja InvalidCount                     ;jump if count > 31

        xchg ecx,edx                        ;exchange contents of ecx & edx
        mov eax,edx                         ;eax = a
        shl eax,cl                          ;eax = a << count;
        mov [r8],eax                        ;save result

        shr edx,cl                          ;edx = a >> count
        mov [r9],edx                        ;save result

        mov eax,1                           ;set success return code

InvalidCount:
        ret                                 ;return to caller

IntegerShift_ endp
        end
```

Near the top of the C++ code, the declaration of the x86 assembly language function `IntegerShift_` is somewhat different than the previous examples in that it defines two pointer arguments. Pointers are used by this function since it needs to return more than one result to its calling function. The other minor difference is that the `int` return value from `IntegerShift_` is used to indicate whether or not the value of `count` is valid. The remaining C++ code in Listing 2-3 exercises the assembly language function `IntegerShift_` using a few test cases and displays results.

The assembly language code of function `IntegerShift_` starts with an `xor eax,eax` instruction that sets register EAX to zero. This is done to ensure that register EAX contains the correct return code should an invalid value for argument `count` be detected. The next instruction, `cmp edx,31`, compares the contents of register EDX, which contains `count`, to the constant value 31. When the processor performs a compare operation, it subtracts the second operand from the first operand, sets the status flags based on the results of this operation, and discards the result. If the value of `count` is above 31, the `ja InvalidCount` instruction performs a jump to the program location specified by the destination operand. If you look ahead a few lines, you will notice a statement with the text `InvalidCount:`. This text is called a label. If `count > 31` is true, the `ja InvalidCount` instruction transfers program control to the first assembly language instruction immediately following the label `InvalidCount`. Note that this instruction can be on same line or a different line, as shown in Listing 2-3.

The xchg ecx,edx instruction swaps the values in registers ECX and EDX. The reason for doing this is that the shl and shr instructions must use register CL for the shift count. The mov eax,edx copies the value a into register EAX, and the subsequent shl eax,cl instruction shifts this value left by the number of bits that's specified in register CL. The 64-bit pointer value a_shr is passed to function IntegerShift_ using register R8 (in 64-bit programming, all pointers are 64 bits). The mov [r8],eax instruction saves the result of the shift operation to the memory location that's specified by the contents of register R8.

The subsequent shr edx,cl instruction shifts the value in register EDX (which contains argument value a) right by the number of bits specified in register CL. This result is then saved to the memory location pointed to by register R9, which contains a pointer to the memory location specified by a_shr. The shr instruction is used in function IntegerShift_ since argument a is declared as an unsigned int. If a were declared an int, the sar (Shift Arithmetic Right) instruction could be used to preserve the sign bit of the source operand. The mov eax,1 instruction loads EAX with the constant one to indicate that the value of the count argument was valid. It should be noted that the testing of count for a value above 31 was implemented to illustrate argument checking in an assembly language function. For shift instructions that use an immediate or variable bit shift count, the processor performs a masking operation that limits the shift count to a value between 0 and 31 when the target operand is 32-bits (the limits 0 and 63 are used for 64-bit operands). Here's the output for source code example Ch02_03.

```
Test 1
count =        6
a =         3119 (0b00000000000000000000110000101111)
shl =     199616 (0b00000000000000110000101111000000)
shr =         48 (0b00000000000000000000000000110000)

Test 2
count =        4
a =      8388736 (0b00000000100000000000000010000000)
shl =  134219776 (0b00001000000000000000100000000000)
shr =     524296 (0b00000000000010000000000000001000)

Test 3
count =       31
a =   2147483649 (0b10000000000000000000000000000001)
shl = 2147483648 (0b10000000000000000000000000000000)
shr =          1 (0b00000000000000000000000000000001)

Test 4
count =       32
a =   1431655765 (0b01010101010101010101010101010101)
Invalid shift count
```

# Advanced Integer Arithmetic

In this section, you'll learn how to perform integer multiplication and division. You'll also learn how to use the x86-64 assembly language instruction set to carry out integer arithmetic using operands of different sizes. In addition to these topics, this section introduces important programming concepts and a few particulars related to Visual C++ calling convention.

■ **Note** The Visual C++ calling convention requirements that are described in this section and in subsequent chapters may be different for other high-level programming languages and operating systems. If you're reading this book to learn x86-64 assembly language and plan on using it with a different high-level programming language or operating system, you should consult the appropriate documentation for more information regarding the target platform's calling convention requirements.

## Multiplication and Division

Listing 2-4 contains the source code for example Ch02_04. In this example, the function IntegerMulDiv_ computes the product, quotient, and remainder of two integers using the imul (Integer Multiplication) and idiv (Integer Division) instructions. Note that the C++ declaration of function IntegerMulDiv_ includes five parameters. Up to this point you've only seen function declarations with a maximum of four parameters, and the arguments values for these parameters were passed using registers RCX, RDX, R8, and R9 or the low-order portion of these registers. The reason for using these registers is that they are required by the Visual C++ calling convention.

***Listing 2-4.*** Example Ch02_04

```
//------------------------------------------------
//               Ch02_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>

using namespace std;

extern "C" int IntegerMulDiv_(int a, int b, int* prod, int* quo, int* rem);

void PrintResult(const char* s, int rc, int a, int b, int p, int q, int r)
{
    const char nl = '\n';

    cout << s << nl;
    cout << "a = " << a << ", b = " << b << ", rc = " << rc << nl;

    if (rc != 0)
        cout << "prod = " << p << ", quo = " << q << ", rem = " << r << nl;
    else
        cout << "prod = " << p << ", quo = undefined" << ", rem = undefined" << nl;

    cout << nl;
}

int main()
{
    int rc;
    int a, b;
    int prod, quo, rem;
```

```
    a = 47;
    b = 13;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    PrintResult("Test 1", rc, a, b, prod, quo, rem);

    a = -291;
    b = 7;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    PrintResult("Test 2", rc, a, b, prod, quo, rem);

    a = 19;
    b = 0;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    PrintResult("Test 3", rc, a, b, prod, quo, rem);

    a = 247;
    b = 85;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    PrintResult("Test 4", rc, a, b, prod, quo, rem);

    return 0;
}
```

```
;-------------------------------------------------
;               Ch02_04.asm
;-------------------------------------------------

;
; extern "C" int IntegerMulDiv_(int a, int b, int* prod, int* quo, int* rem);
;
; Returns:      0 = error (divisor equals zero), 1 = success
;

        .code
IntegerMulDiv_ proc

; Make sure the divisor is not zero
        mov eax,edx                         ;eax = b
        or eax,eax                          ;logical OR sets status flags
        jz InvalidDivisor                   ;jump if b is zero

; Calculate product and save result
        imul eax,ecx                        ;eax = a * b
        mov [r8],eax                        ;save product

; Calculate quotient and remainder, save results
        mov r10d,edx                        ;r10d = b
        mov eax,ecx                         ;eax = a
```

```
        cdq                                 ;edx:eax contains 64-bit dividend
        idiv r10d                           ;eax = quotient, edx = remainder

        mov [r9],eax                        ;save quotient
        mov rax,[rsp+40]                    ;rax = 'rem'
        mov [rax],edx                       ;save remainder
        mov eax,1                           ;set success return code

InvalidDivisor:
        ret                                 ;return to caller

IntegerMulDiv_ endp
        end
```

A calling convention is a binary protocol that describes how arguments and return values are exchanged between two functions. As you have already seen, the Visual C++ calling convention for x86-64 programs on Windows requires a calling function to pass the first four integer (or pointer) arguments using registers RCX, RDX, R8, and R9. The low-order portions of these registers are used for argument values smaller than 64 bits (e.g., ECX, CX, or CL for a 32-, 16-, or 8-bit integer). Any additional arguments are passed using the stack. The calling convention also defines additional requirements including rules for passing floating-point values, general-purpose and XMM register use, and stack frames. You'll learn about these additional requirements in Chapter 5.

The C++ code in Listing 2-4 is similar to the other examples that you've already seen. It simply exercises some test cases and displays results. Upon entry to function `IntegerMulDiv_`, registers ECX, EDX, R8, and R9 contain the argument values `a`, `b`, `prod`, and `quo`, respectively. The fifth argument `rem` is passed on the stack, as shown in Figure 2-1. Note that since `prod`, `quo`, and `rem` are pointers, they are passed to `IntegerMulDiv_` as 64-bit values.

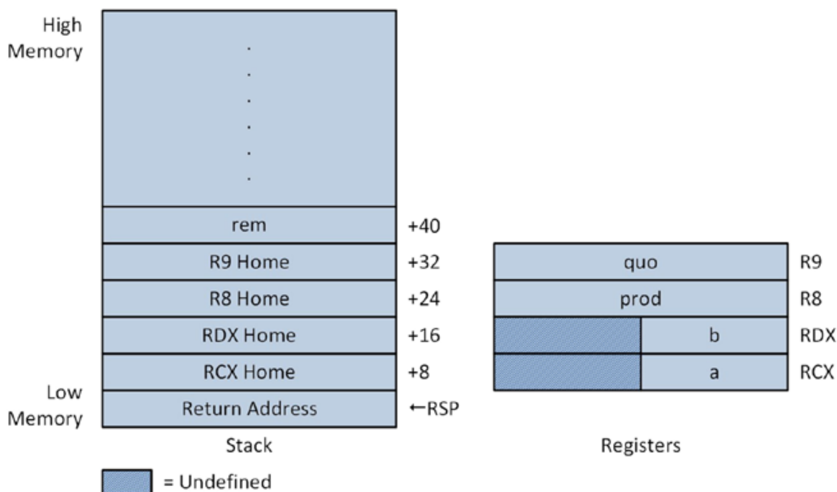



***Figure 2-1.*** *Argument registers and stack at entry to function `IntegerMulDiv_`*

Figure 2-1 illustrates the state of the stack and the argument registers upon entry to IntegerMulDiv_ but prior to the execution of its first instruction. Note that the location of the fifth argument value rem is at memory address RSP + 40. As simple mov instruction can be used to load rem, which is a pointer, into a general-purpose register when it's needed. Also note in Figure 2-1 that register RSP points to the caller's return address on the stack. During execution of a ret instruction, the processor copies this value from the stack and ultimately stores it in register RIP. The ret instruction also removes the caller's return address from the stack by adding 8 to the value in RSP. The stack locations labeled RCX Home, RDX Home, R8 Home, and R9 Home are storage areas that can be used to temporarily save the corresponding argument registers. These areas can also be used to store other transient data. You'll learn more about the home area in Chapter 5.

The function IntegerMulDiv_ computes and saves the product a * b. It also calculates and saves the quotient and remainder of a / b. Since IntegerMulDiv_ performs division using b, it makes sense to test the value of b to confirm that it's not equal to zero. In Listing 2-4, the mov eax,edx instruction copies b into register EAX. The next instruction, or eax,eax, performs a bitwise OR operation to set the status flags. If b is zero, the jz InvalidDivisor (Jump if Zero) instruction skips over the code that performs the division. Like the previous example, the function IntegerMulDiv_ uses a return value of zero to indicate an error condition. Since EAX already contains zero, no additional instructions are necessary.

The next instruction imul eax,ecx computes a * b and saves the product to the memory location specified by R8, which contains the pointer prod. The x86-64 instruction set supports several different forms of the imul instruction. The two-operand form that's used here actually computes a 64-bit result (recall that the product of two 32-bit integers is always a 64-bit result) but saves only the lower 32 bits in the destination operand. The single-operand form of imul can be used when a non-truncated result is required.

Integer division occurs next. The mov r10d,rdx and mov eax,ecx instructions load registers R10D and EAX with argument values b and a, respectively. Before performing the division operation, the 32-bit dividend in EAX must be sign-extended to 64 bits and this is carried out by the cdq (Convert Doubleword to Quadword) instruction. Following execution of cdq, register pair EDX:EAX contains the 64-bit dividend and register R10D contains the 32-bit divisor. The idiv r10d instruction divides the contents of register pair EDX:EAX by the value in R10D. After execution of the idiv instruction, the 32-bit quotient and 32-bit remainder reside in registers EAX and EDX, respectively. The subsequent mov [r9],eax saves the quotient to the memory location specified by quo. In order to save the remainder, the pointer rem must be obtained from the stack and this is achieved using a mov rax,[rsp+40] instruction. The mov [rax],edx instruction saves the remainder to the memory location specified by rem. The output for example Ch02_04 is the following:

```
Test 1
a = 47, b = 13, rc = 1
prod = 611, quo = 3, rem = 8

Test 2
a = -291, b = 7, rc = 1
prod = -2037, quo = -41, rem = -4

Test 3
a = 19, b = 0, rc = 0
prod = 0, quo = undefined, rem = undefined

Test 4
a = 247, b = 85, rc = 1
prod = 20995, quo = 2, rem = 77
```

## Calculations Using Mixed Types

In many programs, it is often necessary to perform arithmetic calculations using multiple integer types. Consider the C++ expression `a = b * c * d * e`, where `a`, `b`, `c`, `d`, and `e` are declared as `long long`, `long long`, `int`, `short`, and `char`. Calculating the correct result requires proper promotion of the smaller-sized integers into large ones. In the next example, you'll learn few techniques that can be used to carry out integer promotions in an assembly language function. You'll also learn how to access integer argument values of various sizes that are stored on the stack. Listing 2-5 contains the source code for example `Ch02_05`.

***Listing 2-5.*** Example Ch02_05

```
//------------------------------------------------
//              Ch02_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cstdint>

using namespace std;

extern "C" int64_t IntegerMul_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e, int16_t
f, int32_t g, int64_t h);

extern "C" int UnsignedIntegerDiv_(uint8_t a, uint16_t b, uint32_t c, uint64_t d, uint8_t e,
uint16_t f, uint32_t g, uint64_t h, uint64_t* quo, uint64_t* rem);

void IntegerMul(void)
{
    int8_t a = 2;
    int16_t b = -3;
    int32_t c = 8;
    int64_t d = 4;
    int8_t e = 3;
    int16_t f = -7;
    int32_t g = -5;
    int64_t h = 10;

    // Calculate a * b * c * d * e * f * g * h
    int64_t prod1 = a * b * c * d * e * f * g * h;
    int64_t prod2 = IntegerMul_(a, b, c, d, e, f, g, h);

    cout << "\nResults for IntegerMul\n";
    cout << "a = " << (int)a << ", b = " << b << ", c = " << c << ' ';
    cout << "d = " << d << ", e = " << (int)e << ", f = " << f << ' ';
    cout << "g = " << g << ", h = " << h << '\n';
    cout << "prod1 = " << prod1 << '\n';
    cout << "prod2 = " << prod2 << '\n';
}
```

```
void UnsignedIntegerDiv(void)
{
    uint8_t a = 12;
    uint16_t b = 17;
    uint32_t c = 71000000;
    uint64_t d = 90000000000;
    uint8_t e = 101;
    uint16_t f = 37;
    uint32_t g = 25;
    uint64_t h = 5;
    uint64_t quo1, rem1;
    uint64_t quo2, rem2;

    quo1 = (a + b + c + d) / (e + f + g + h);
    rem1 = (a + b + c + d) % (e + f + g + h);
    UnsignedIntegerDiv_(a, b, c, d, e, f, g, h, &quo2, &rem2);

    cout << "\nResults for UnsignedIntegerDiv\n";
    cout << "a = " << (unsigned)a << ", b = " << b << ", c = " << c << ' ';
    cout << "d = " << d << ", e = " << (unsigned)e << ", f = " << f << ' ';
    cout << "g = " << g << ", h = " << h << '\n';
    cout << "quo1 = " << quo1 << ", rem1 = " << rem1 << '\n';
    cout << "quo2 = " << quo2 << ", rem2 = " << rem2 << '\n';
}

int main()
{
    IntegerMul();
    UnsignedIntegerDiv();
    return 0;
}
```

```
;-------------------------------------------------
;               Ch02_05.asm
;-------------------------------------------------

; extern "C" int64_t IntegerMul_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e,
 int16_t f, int32_t g, int64_t h);

        .code
IntegerMul_ proc

; Calculate a * b * c * d
        movsx rax,cl                        ;rax = sign_extend(a)
        movsx rdx,dx                        ;rdx = sign_extend(b)
        imul rax,rdx                        ;rax = a * b
        movsxd rcx,r8d                      ;rcx = sign_extend(c)
        imul rcx,r9                         ;rcx = c * d
        imul rax,rcx                        ;rax = a * b * c * d
```

```
; Calculate e * f * g * h
        movsx rcx,byte ptr [rsp+40]         ;rcx = sign_extend(e)
        movsx rdx,word ptr [rsp+48]         ;rdx = sign_extend(f)
        imul rcx,rdx                        ;rcx = e * f
        movsxd rdx,dword ptr [rsp+56]       ;rdx = sign_extend(g)
        imul rdx,qword ptr [rsp+64]         ;rdx = g * h
        imul rcx,rdx                        ;rcx = e * f * g * h

; Compute the final product
        imul rax,rcx                        ;rax = final product

        ret
IntegerMul_ endp

; extern "C" int UnsignedIntegerDiv_(uint8_t a, uint16_t b, uint32_t c, uint64_t d, uint8_t e,
uint16_t f, uint32_t g, uint64_t h, uint64_t* quo, uint64_t* rem);

UnsignedIntegerDiv_ proc

; Calculate a + b + c + d
        movzx rax,cl                        ;rax = zero_extend(a)
        movzx rdx,dx                        ;rdx = zero_extend(b)
        add rax,rdx                         ;rax = a + b
        mov r8d,r8d                         ;r8 = zero_extend(c)
        add r8,r9                           ;r8 = c + d
        add rax,r8                          ;rax = a + b + c + d
        xor rdx,rdx                         ;rdx:rax = a + b + c + d

; Calculate e + f + g + h
        movzx r8,byte ptr [rsp+40]          ;r8 = zero_extend(e)
        movzx r9,word ptr [rsp+48]          ;r9 = zero_extend(f)
        add r8,r9                           ;r8 = e + f
        mov r10d,[rsp+56]                   ;r10 = zero_extend(g)
        add r10,[rsp+64]                    ;r10 = g + h;
        add r8,r10                          ;r8 = e + f + g + h
        jnz DivOK                           ;jump if divisor is not zero

        xor eax,eax                         ;set error return code
        jmp done

; Calculate (a + b + c + d) / (e + f + g + h)

DivOK:  div r8                              ;unsigned divide rdx:rax / r8
        mov rcx,[rsp+72]
        mov [rcx],rax                       ;save quotient
        mov rcx,[rsp+80]
        mov [rcx],rdx                       ;save remainder

        mov eax,1                           ;set success return code

Done:   ret
UnsignedIntegerDiv_ endp
        end
```

The assembly language function `IntegerMul_` calculates the product of eight signed integers ranging in size from 8 bits to 64 bits. The C++ declaration for this function uses the fixed-sized integer types that are declared in the header file `<cstdint>` instead of the normal `long long`, `int`, `short`, and `char`. Some assembly language programmers (including me) prefer to use fixed-sized integer types for assembly language function declarations since it emphasizes the exact size of the argument. The declaration of function `UnsignedIntegerDiv_`, which demonstrates how to perform unsigned integer division, also uses fixed-size integer types. Figure 2-2 illustrates the contents of the stack at entry to `IntegerMul_`.

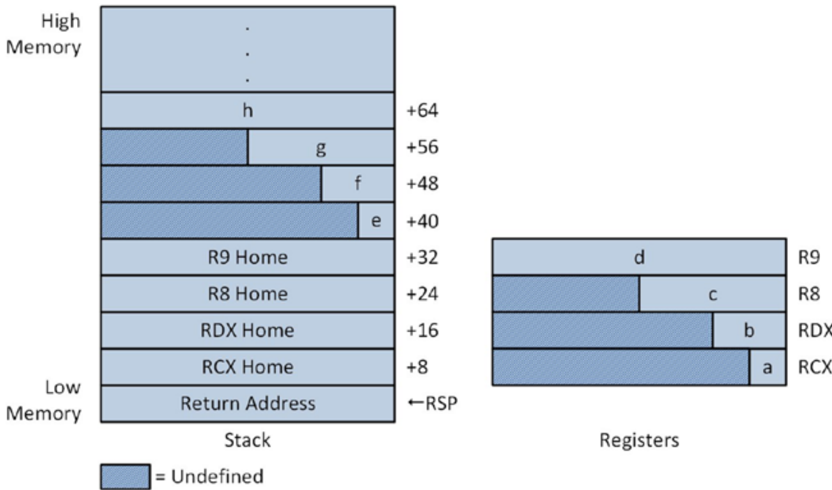



***Figure 2-2.*** *Argument registers and stack at entry to function `IntegerMul_`*

The first instruction of `IntegerMul_`, `movsx rax,cl` (Move with Sign Extension), sign-extends a copy of the 8-bit integer value `a` that's in register CL to 64 bits and saves this value in register RAX. Note that the original value in register CL is unaltered by this operation. Another `movsx` instruction follows that saves a 64-bit sign-extend copy of the 16-bit value `d` to RDX. Like the previous `movsx` instruction, the source operand is not modified by this operation. An `imul rax,rdx` instruction computes the product of `a` and `b`. The two-operand form of the `imul` instruction that's used here saves only the lower 64 bits of the 128-bit product in the destination operand RAX. The next instruction `movsxd rcx,r8d` sign-extends the 32-bit operand `c` to 64 bits. Note that a different instruction mnemonic is required when sign extending a 32-bit integer to 64 bits. The next two `imul` instructions compute the intermediate product `a * b * c * d`.

Calculation of the second intermediate product `e * f * g * h` is carried out next. All of these argument values were passed using the stack as shown in Figure 2-2. The `movsx rcx,byte ptr [rsp+40]` sign extends a copy of the 8-bit argument value `e` that's located on the stack and saves the result to register RCX. The text `byte ptr` is a MASM directive that acts like a C++ cast operator and conveys to the assembler the size of the source operand. Without the `byte ptr` directive, the `movsx` instruction is ambiguous since several different sizes are possible for the source operand. The argument value `f` is loaded next using a `movsx rdx,word ptr [rsp+48]` instruction. Following calculation of the intermediate product `e * f` using an `imul` instruction, a `movsxd rdx,dword ptr[rsp+56]` instruction loads a sign-extended copy of `g` into RDX. This is followed by an `imul rdx,qword ptr[rsp+64]` instruction that calculates the intermediate product `g * h`. Use of the `qword ptr` directive is optional here; size directives are often used in this manner to improve program readability. The final two `imul` instructions calculate the final product.

Figure 2-3 illustrates the contents of the stack at entry to the function `UnsignedIntegerDiv_`. This function calculates the quotient and remainder of the expression `(a + b + c + d) / (e + f + g + g)`. As implied by its name, `UnsignedIntegerDiv_` uses unsigned integer arguments of different sizes and performs unsigned integer division. In order to calculate the correct results, the smaller-sized arguments must be zero-extended prior to any arithmetic operations. The `movzx rax,cl` and `movzx rdx,dx` instructions load zero-extended copies of argument values `a` and `b` into their respective destination registers. The `add rax,rdx` instruction that follows next calculates the intermediate sum `a + b`. At first glance, the `mov r8d,r8d` instruction that follows seems superfluous, but it's actually performing a necessary operation. When an x86 processor is running in 64-bit mode, instructions that employ 32-bit operands produce 32-bit results. If the destination operand is a 32-bit register, the high-order 32 bits (i.e., bits 63 – 32) of the corresponding 64-bit register are set to zero. The `mov r8d,r8d` instruction is used here to zero-extend the 32-bit value `c` that's already loaded in register R8D to a 64-bit value in R8. The next two `add` instructions calculate the intermediate sum `a + b + c + d` and save the result to RAX. The ensuing `xor rdx,rdx` instruction yields a 128-bit zero-extended dividend value that's stored in register pair RDX:RAX.

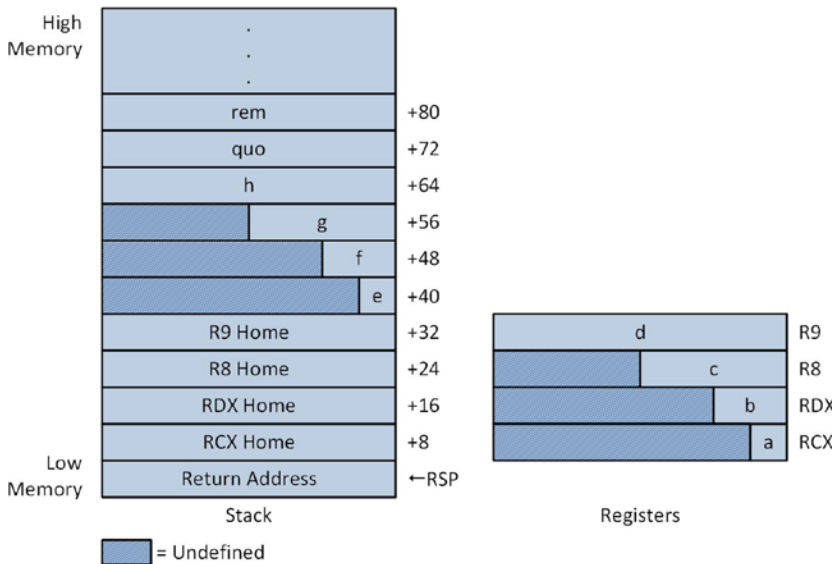



***Figure 2-3.*** *Argument registers and stack at entry to function* `UnsignedIntegerDiv_`

A similar sequence of instructions is used to calculate the intermediate sum `e + f + g + h`, with the main difference being that these arguments are loaded from the stack. This value is then tested to see if it's equal to zero since it will be used as the divisor. If the divisor is not zero, a `div r8` instruction performs unsigned integer division using register pair RDX:RAX as the dividend and register R8 as the divisor. The resulting quotient (RAX) and remainder (RDX) are then saved to the memory locations specified by the pointers `quo` and `rem`, which were passed on the stack. Here's the output for example `Ch02_05`.

```
Results for IntegerMul
a = 2, b = -3, c = 8 d = 4, e = 3, f = -7 g = -5, h = 10
prod1 = -201600
prod2 = -201600
```

```
Results for UnsignedIntegerDiv
a = 12, b = 17, c = 71000000 d = 90000000000, e = 101, f = 37 g = 25, h = 5
quo1 = 536136904, rem1 = 157
quo2 = 536136904, rem2 = 157
```

# Memory Addressing and Condition Codes

Thus far the source code examples of this chapter have primarily illustrated how to use basic arithmetic and logical instructions. In this section, you'll learn more about the x86's memory addressing modes. You'll also examine sample code that demonstrates how to exploit some of the x86's condition-code based instructions.

## Memory Addressing Modes

You learned in Chapter 1 that the x86-64 instruction set supports a variety of addressing modes that can be used to reference an operand in memory. In this section, you'll examine an assembly language function that illustrates how to use some of these modes. You'll also learn how to initialize an assembly language lookup table and use assembly language global variables in a C++ function. Listing 2-6 shows the source code for example Ch02_06.

***Listing 2-6.*** Example Ch02_06

```
//------------------------------------------------
//               Ch02_06.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" int NumFibVals_, FibValsSum_;
extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int* v4);

int main()
{
    const int w = 5;
    const char nl = '\n';
    const char* delim = ", ";

    FibValsSum_ = 0;

    for (int i = -1; i < NumFibVals_ + 1; i++)
    {
        int v1 = -1, v2 = -1, v3 = -1, v4 = -1;
        int rc = MemoryAddressing_(i, &v1, &v2, &v3, &v4);

        cout << "i = " << setw(w - 1) << i << delim;
        cout << "rc = " << setw(w - 1) << rc << delim;
        cout << "v1 = " << setw(w) << v1 << delim;
```

```
        cout << "v2 = " << setw(w) << v2 << delim;
        cout << "v3 = " << setw(w) << v3 << delim;
        cout << "v4 = " << setw(w) << v4 << delim;
        cout << nl;
    }

    cout << "FibValsSum_ = " << FibValsSum_ << nl;
    return 0;
}

;-------------------------------------------------
;               Ch02_06.asm
;-------------------------------------------------

; Simple lookup table (.const section data is read only)

            .const
FibVals     dword 0, 1, 1, 2, 3, 5, 8, 13
            dword 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597

NumFibVals_ dword ($ - FibVals) / sizeof dword
            public NumFibVals_

; Data section (data is read/write)

            .data
FibValsSum_ dword ?             ;value to demo RIP-relative addressing
            public FibValsSum_

;
; extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int* v4);
;
; Returns:      0 = error (invalid table index), 1 = success
;

           .code
MemoryAddressing_ proc

; Make sure 'i' is valid
        cmp ecx,0
        jl InvalidIndex                     ;jump if i < 0
        cmp ecx,[NumFibVals_]
        jge InvalidIndex                    ;jump if i >= NumFibVals_

; Sign extend i for use in address calculations
        movsxd rcx,ecx                      ;sign extend i
        mov [rsp+8],rcx                     ;save copy of i (in rcx home area)

; Example #1 - base register
        mov r11,offset FibVals              ;r11 = FibVals
        shl rcx,2                           ;rcx = i * 4
```

```
        add r11,rcx                         ;r11 = FibVals + i * 4
        mov eax,[r11]                       ;eax = FibVals[i]
        mov [rdx],eax                       ;save to v1

; Example #2 - base register + index register
        mov r11,offset FibVals              ;r11 = FibVals
        mov rcx,[rsp+8]                     ;rcx = i
        shl rcx,2                           ;rcx = i * 4
        mov eax,[r11+rcx]                   ;eax = FibVals[i]
        mov [r8],eax                        ;save to v2

; Example #3 - base register + index register * scale factor
        mov r11,offset FibVals              ;r11 = FibVals
        mov rcx,[rsp+8]                     ;rcx = i
        mov eax,[r11+rcx*4]                 ;eax = FibVals[i]
        mov [r9],eax                        ;save to v3

; Example #4 - base register + index register * scale factor + disp
        mov r11,offset FibVals-42           ;r11 = FibVals - 42
        mov rcx,[rsp+8]                     ;rcx = i
        mov eax,[r11+rcx*4+42]              ;eax = FibVals[i]
        mov r10,[rsp+40]                    ;r10 = ptr to v4
        mov [r10],eax                       ;save to v4

; Example #5 - RIP relative
        add [FibValsSum_],eax               ;update sum

        mov eax,1                           ;set success return code
        ret

InvalidIndex:
        xor eax,eax                         ;set error return code
        ret

MemoryAddressing_ endp
        end
```

Toward the top of the C++ code are the requisite declaration statements for this example. Earlier in this chapter you learned how to reference a C++ global variable in an assembly language function. In this example, the opposite is illustrated. Storage space for the variables `NumFibVals_` and `FibValsSum_` is defined in the assembly language code, and these variables are referenced in the function `main`.

In the assembly language function `MemoryOperands_`, argument `i` is employed as an index into an array (or lookup table) of constant integers, while the four pointer arguments are used to save values loaded from the lookup table using different addressing modes. Near the top of Listing 2-6 is a `.const` directive, which defines a memory block that contains read-only data. Immediately following the `.const` directive, a lookup table named `FibVals` is defined. This table contains 16 doubleword integer values. The text `dword` is an assembler directive that is used to allocate storage space and optionally initialize a doubleword value (the text `dd` can also be used as a synonym for `dword`).

The line `NumFibVals_ dword ($ - FibVals) / sizeof dword` allocates storage space for a single doubleword value and initializes it with the number of doubleword elements in the lookup table `FibVals`. The `$` character is an assembler symbol that equals the current value of the location counter (or offset from

the beginning of the current memory block). Subtracting the offset of FibVals from $ yields the size of the table in bytes. Dividing this result by the size in bytes of a doubleword value generates the correct number of elements. These statements emulate a commonly-used technique in C++ to define and initialize a variable with the number of elements in an array:

```
const int Values[] = {10, 20, 30, 40, 50};
const int NumValues = sizeof(Values) / sizeof(int);
```

The final line of the .const section declares NumFibVals_ as a public symbol in order to enable its use in main. The .data directive denotes the start of a memory block that contains modifiable data. The FibValsSum_ dword ? statement defines an uninitialized doubleword value, and the subsequent public statement makes it globally accessible.

Let's now look at the assembly language code for MemoryAddressing_. Upon entry into the function, the argument i is checked for validity since it will be used as an index into the lookup table FibVals. The cmp ecx,0 instruction compares the contents of ECX, which contains i, to the immediate value 0. As discussed earlier in this chapter, the processor carries out this comparison by subtracting the source operand from the destination operand. It then sets the status flags based on the result of the subtraction (the result is not saved to the destination operand). If the condition ecx < 0 is true, program control will be transferred to the location specified by the jl (Jump if Less) instruction. A similar sequence of instructions is used to determine if the value of i is too large. The cmp ecx,[NumFibVals_] instruction compares ECX against the number of elements in the lookup table. If ecx >= [NumFibVals] is true, a jump is performed to the target location specified by the jge (Jump if Greater or Equal) instruction.

Immediately following the validation of i, a movsxd rcx,ecx sign-extends the table index value to 64 bits. Sign-extending or zero-extending a 32-bit integer to a 64-bit integer is often necessary when using an addressing mode that employs an index register as you'll soon see. The subsequent mov [rsp+8],rcx saves a copy of the signed-extended table index value to the RCX home area on the stack and is done primarily to exemplify use of the stack home area.

The remaining instructions of MemoryAddressing_ illustrate accessing items in the lookup table using various memory addressing modes. The first example uses a single base register to read an item from the table. In order to use a single base register, the function must explicitly calculate the address of the *i-th* table element, which is achieved by adding the offset (or starting address) of FibVals and the value i * 4. The mov r11,offset FibVals instruction loads R11 with the correct table offset value. This is followed by a shl rcx,2 instruction that determines the offset of the *i-th* item relative to the start of the lookup table. An add r11,rcx instruction calculates the final address. Once this is complete, the specified table value is read using a mov eax,[r11] instruction. It is then saved to the memory location specified by the argument v1.

In the second example, the table value is read using BaseReg+IndexReg memory addressing. This example is similar to the first one except that the processor computes the final effective address during execution of the mov eax,[r11+rcx] instruction. Note that recalculation of the lookup table element offset using the mov rcx,[rsp+8] and shl rcx,2 instructions is unnecessary here but included to illustrate use of the stack home area.

The third example demonstrates use of BaseReg+IndexReg*ScaleFactor memory addressing. In this example, the offset of FibVals and the value i are loaded into registers R11 and RCX, respectively. The correct table value is loaded into EAX using a mov eax,[r11+rcx*4] instruction. In the fourth (and somewhat contrived) example, BaseReg+IndexReg*ScaleFactor+Disp memory addressing is demonstrated. The fifth and final memory address mode example uses an add[FibValsSum_],eax instruction to demonstrate RIP-relative addressing. This instruction, which uses a memory location as a destination operand, updates a running sum that is ultimately displayed by the C++ code.

The function main that's shown in Listing 2-6 contains a simple looping construct that exercises the function MemoryOperands_ including test cases with an invalid index. Note that the for loop uses the variable NumFibVals_, which was defined as a public symbol in the assembly language file. The output for the sample program Ch02_06 is shown here.

```
i =   -1, rc =    0, v1 =    -1, v2 =    -1, v3 =    -1, v4 =    -1,
i =    0, rc =    1, v1 =     0, v2 =     0, v3 =     0, v4 =     0,
i =    1, rc =    1, v1 =     1, v2 =     1, v3 =     1, v4 =     1,
i =    2, rc =    1, v1 =     1, v2 =     1, v3 =     1, v4 =     1,
i =    3, rc =    1, v1 =     2, v2 =     2, v3 =     2, v4 =     2,
i =    4, rc =    1, v1 =     3, v2 =     3, v3 =     3, v4 =     3,
i =    5, rc =    1, v1 =     5, v2 =     5, v3 =     5, v4 =     5,
i =    6, rc =    1, v1 =     8, v2 =     8, v3 =     8, v4 =     8,
i =    7, rc =    1, v1 =    13, v2 =    13, v3 =    13, v4 =    13,
i =    8, rc =    1, v1 =    21, v2 =    21, v3 =    21, v4 =    21,
i =    9, rc =    1, v1 =    34, v2 =    34, v3 =    34, v4 =    34,
i =   10, rc =    1, v1 =    55, v2 =    55, v3 =    55, v4 =    55,
i =   11, rc =    1, v1 =    89, v2 =    89, v3 =    89, v4 =    89,
i =   12, rc =    1, v1 =   144, v2 =   144, v3 =   144, v4 =   144,
i =   13, rc =    1, v1 =   233, v2 =   233, v3 =   233, v4 =   233,
i =   14, rc =    1, v1 =   377, v2 =   377, v3 =   377, v4 =   377,
i =   15, rc =    1, v1 =   610, v2 =   610, v3 =   610, v4 =   610,
i =   16, rc =    1, v1 =   987, v2 =   987, v3 =   987, v4 =   987,
i =   17, rc =    1, v1 =  1597, v2 =  1597, v3 =  1597, v4 =  1597,
i =   18, rc =    0, v1 =    -1, v2 =    -1, v3 =    -1, v4 =    -1,
FibValsSum_ = 4180
```

Given the multiple addressing modes that are available on an x86 processor, you might wonder which mode should be used. The answer to this question depends on a number of factors, including register availability, the number of times an instruction (or sequence of instructions) is expected to execute, instruction ordering, and memory space vs. execution time tradeoffs. Hardware features such as the processor's underlying microarchitecture and cache sizes also need to be considered.

When coding an x86 assembly language function, one suggested guideline is to favor simple (a single base register or displacement) rather than complex (multiple registers) memory addressing. The drawback of this approach is that the simpler forms generally require the programmer to code longer instruction sequences and may consume more code space. The use of a simple form also may be imprudent if extra instructions are needed to preserve non-volatile registers on the stack (non-volatile registers are explained in Chapter 3). Chapter 15 considers in greater detail some of the issues and tradeoffs that can affect the efficiency of assembly language code.

## Condition Codes

The final sample program of this chapter expounds on how to use the x86's conditional instructions jcc (Conditional Jump) and cmovcc (Conditional Move). As you have already seen in a few of this chapter's source code examples, the execution of a conditional instruction is contingent on its specified condition code and the state of one or more status flags. The source code example Ch02_07, which is shown in Listing 2-7, demonstrates a few more use cases for the previously-mentioned instructions.

***Listing 2-7.*** Example Ch02_07

```
//------------------------------------------------
//               Ch02_07.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" int SignedMinA_(int a, int b, int c);
extern "C" int SignedMaxA_(int a, int b, int c);
extern "C" int SignedMinB_(int a, int b, int c);
extern "C" int SignedMaxB_(int a, int b, int c);

void PrintResult(const char* s1, int a, int b, int c, int result)
{
    const int w = 4;

    cout << s1 << "(";
    cout << setw(w) << a << ", ";
    cout << setw(w) << b << ", ";
    cout << setw(w) << c << ") = ";
    cout << setw(w) << result << '\n';
}

int main()
{
    int a, b, c;
    int smin_a, smax_a, smin_b, smax_b;

    // SignedMin examples
    a = 2; b = 15; c = 8;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    PrintResult("SignedMinA", a, b, c, smin_a);
    PrintResult("SignedMinB", a, b, c, smin_b);
    cout << '\n';

    a = -3; b = -22; c = 28;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    PrintResult("SignedMinA", a, b, c, smin_a);
    PrintResult("SignedMinB", a, b, c, smin_b);
    cout << '\n';

    a = 17; b = 37; c = -11;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    PrintResult("SignedMinA", a, b, c, smin_a);
```

```
    PrintResult("SignedMinB", a, b, c, smin_b);
    cout << '\n';

    // SignedMax examples
    a = 10; b = 5; c = 3;
    smax_a = SignedMaxA_(a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    PrintResult("SignedMaxA", a, b, c, smax_a);
    PrintResult("SignedMaxB", a, b, c, smax_b);
    cout << '\n';

    a = -3; b = 28; c = 15;
    smax_a = SignedMaxA_(a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    PrintResult("SignedMaxA", a, b, c, smax_a);
    PrintResult("SignedMaxB", a, b, c, smax_b);
    cout << '\n';

    a = -25; b = -37; c = -17;
    smax_a = SignedMaxA_(a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    PrintResult("SignedMaxA", a, b, c, smax_a);
    PrintResult("SignedMaxB", a, b, c, smax_b);
    cout << '\n';
}

;-------------------------------------------------
;               Ch02_07.asm
;-------------------------------------------------

; extern "C" int SignedMinA_(int a, int b, int c);
;
; Returns:      min(a, b, c)

        .code
SignedMinA_ proc
        mov eax,ecx
        cmp eax,edx                         ;compare a and b
        jle @F                              ;jump if a <= b
        mov eax,edx                         ;eax = b

@@:     cmp eax,r8d                         ;compare min(a, b) and c
        jle @F
        mov eax,r8d                         ;eax = min(a, b, c)

@@:     ret
SignedMinA_ endp

; extern "C" int SignedMaxA_(int a, int b, int c);
;
; Returns:      max(a, b, c)
```

```
SignedMaxA_ proc
        mov eax,ecx
        cmp eax,edx                         ;compare a and b
        jge @F                              ;jump if a >= b
        mov eax,edx                         ;eax = b

@@:     cmp eax,r8d                         ;compare max(a, b) and c
        jge @F
        mov eax,r8d                         ;eax = max(a, b, c)

@@:     ret
SignedMaxA_ endp

; extern "C" int SignedMinB_(int a, int b, int c);
;
; Returns:      min(a, b, c)

SignedMinB_ proc
        cmp ecx,edx
        cmovg ecx,edx                       ;ecx = min(a, b)
        cmp ecx,r8d
        cmovg ecx,r8d                       ;ecx = min(a, b, c)
        mov eax,ecx
        ret
SignedMinB_ endp

; extern "C" int SignedMaxB_(int a, int b, int c);
;
; Returns:      max(a, b, c)

SignedMaxB_ proc
        cmp ecx,edx
        cmovl ecx,edx                       ;ecx = max(a, b)
        cmp ecx,r8d
        cmovl ecx,r8d                       ;ecx = max(a, b, c)
        mov eax,ecx
        ret
SignedMaxB_ endp
        end
```

When developing code to implement a particular algorithm, it is often necessary to determine the minimum or maximum value of two numbers. The standard C++ library defines two template functions named `std::min()` and `std::max()` to perform these operations. The assembly language code that's shown in Listing 2-7 contains several three-argument versions of signed-integer minimum and maximum functions. The purpose of these functions is to illustrate proper use of the `jcc` and `cmovcc` instructions. The first function, called `SignedMinA_`, finds the minimum value of three signed integers. The first code block determines `min(a, b)` using two instructions: `cmp eax,ecx` and `jle @F`. The `cmp` instruction, which you saw earlier in this chapter, subtracts the source operand from the destination operand and sets the status flags based on the result (the result is not saved). The operand of the `jle` (Jump if Less or Equal) instruction, `@F`, is an assembler symbol that designates nearest forward `@@` label as the target of the conditional jump

(the symbol @B can be used for backward jumps). Following calculation of min(a, b), the next code block determines min(min(a, b), c) using the same technique. With the result already present in register EAX, SignedMinA_ can return to the caller.

The function SignedMaxA_ uses the same approach to find the maximum of three signed integers. The only difference between SignedMaxA_ and SignedMinA_ is the use of a jge (Jump if Greater or Equal) instead of a jle instruction. Versions of SignedMinA_ and SignedMaxA_ that operate on unsigned integers can be easily created by changing the jle and jge instructions to jbe (Jump if Below or Equal) and jae (Jump if Above or Equal), respectively. Recall from the discussion in Chapter 1 that condition codes using the words "greater" and "less" are intended for signed integer operands, while "above" and "below" are used with unsigned integer operands.

The assembly language code also contains the functions SignedMinB_ and SignedMaxB_. These functions determine the minimum and maximum of three signed integers using conditional move instructions instead of conditional jumps. The cmovcc instruction tests the specified condition and if it's true, the source operand is copied to the destination operand. If the specified condition is false, the destination operand is not altered.

If you examine the function SignedMinB_, you will notice that following the cmp ecx,edx instruction is a cmovg ecx,edx instruction. The cmovg (Move if Greater) instruction copies the contents of EDX to ECX if ECX is greater than EDX. In this example, registers ECX and EDX contain argument values a and b. Following execution of the cmovg instruction, register ECX contains min(a, b). Another cmp and cmovg instruction sequence follows which yields min(a, b, c). The same technique is used in SignedMaxB_, which employs cmovl instead of cmovg to save the largest signed integer. Unsigned versions of these functions can be easily created by using cmova and cmovb instead of cmovg and cmovl, respectively. Here's the output for Ch02_07.

```
SignedMinA(   2,   15,    8) =   2
SignedMinB(   2,   15,    8) =   2

SignedMinA(  -3,  -22,   28) = -22
SignedMinB(  -3,  -22,   28) = -22

SignedMinA(  17,   37,  -11) = -11
SignedMinB(  17,   37,  -11) = -11

SignedMaxA(  10,    5,    3) =  10
SignedMaxB(  10,    5,    3) =  10

SignedMaxA(  -3,   28,   15) =  28
SignedMaxB(  -3,   28,   15) =  28

SignedMaxA( -25,  -37,  -17) = -17
SignedMaxB( -25,  -37,  -17) = -17
```

The use of a conditional move instruction to eliminate one or more conditional jump statements frequently results in faster code, especially in situations where the processor is unable to accurately predict whether the jump will be performed. You'll learn more about some of issues related to optimal use of the conditional jump and conditional move instructions in Chapter 15.

# Summary

Here are the key learning points of Chapter 2:

- The `add` and `sub` instructions perform integer (signed and unsigned) addition and subtraction.
- The `imul` and `idiv` instructions carry out signed integer multiplication and division. The corresponding instructions for unsigned integers are `mul` and `div`. The `idiv` and `div` instructions usually require the dividend to be sign- or zero-extended prior to use.
- The `and`, `or`, and `xor` instructions are used to perform bitwise AND, inclusive OR, and exclusive OR operations. The `shl` and `shr` instructions execute logical left and right shifts; `sar` is used for arithmetic right shifts.
- Nearly all arithmetic, logical, and shift instructions set the status flags to indicate the results of an operation. The `cmp` instruction also sets the status flags. The `jcc` and `cmovcc` instructions can be used to alter program flow or perform conditional data moves based on the state of one or more status flags.
- The x86-64 instruction set supports a variety of different address modes for accessing operands stored in memory.
- MASM uses the `.code`, `.data`, and `.const` directives to designate code, data, and constant data sections. The directives `proc` and `endp` denote the beginning and end of an assembly language function.
- The Visual C++ calling convention requires a calling function to use registers RCX, RDX, R8, and R9 (or the low-order portions of these registers for values smaller than 64 bits) for the first four integer or pointer arguments. Additional arguments are passed on the stack.
- To disable the creation of decorated names by the C++ compiler, assembly language functions must be declared using the `extern "C"` modifier. Global variables shared between C++ and assembly language code must also use the `extern "C"` modifier.

CHAPTER 3

# X86-64 Core Programming – Part 2

The previous chapter introduced the fundamentals of x86-64 assembly language programming. You learned how to use the x86-64 instruction set to perform integer addition, subtraction, multiplication, and division. You also examined source code that illustrated use of logical instructions, shift operations, memory addressing modes, and conditional jumps and moves. In addition to learning about frequently used instructions, your initiation to x86-64 assembly language programming has also covered important practical details including assembler directives and calling convention requirements.

In this chapter, your exploration of x86-64 assembly language programming fundamentals continues. You'll learn how to use additional x86-64 instructions and assembler directives. You'll also study source code that elucidates how to manipulate common programming constructs including arrays and data structures. This chapter concludes with several examples that demonstrate use of the x86's string instructions.

## Arrays

Arrays are an indispensable data construct in virtually all programming languages. In C++ there is an inherent connection between arrays and pointers since the name of an array is essentially a pointer to its first element. Moreover, whenever an array is used as a C++ function parameter, a pointer is passed instead of duplicating the array on the stack. Pointers are also employed for arrays that are dynamically allocated at runtime. This section examines x86-64 assembly language code that processes arrays. The first two sample programs demonstrate how to perform simple operations using one-dimensional arrays. This is followed by two examples that explain the techniques necessary to access the elements of a two-dimensional array.

### One-Dimensional Arrays

In C++ one-dimensional arrays are stored in a contiguous block of memory that can be statically allocated at compile time or dynamically during program execution. The elements of a C++ array are accessed using zero-based indexing, which means that valid indices for an array of size N range from 0 to N-1. The sample code of this section includes examples that carry out basic operations with one-dimensional arrays using the x86-64 instruction set.

#### Accessing Elements

Listing 3-1 shows the source code for example `Ch03_01`. In this example, the function `CalcArraySum_` sums the elements of an integer array. Near the top of the C++ code is the now familiar declaration for the assembly-language function `CalcArraySum_`. The summing calculation that's performed by this function is duplicated in the C++ function `CalcArraySumCpp` for comparison purposes.

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_3

***Listing 3-1.*** Example Ch03_01

```
//------------------------------------------------
//               Ch03_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" int CalcArraySum_(const int* x, int n);

int CalcArraySumCpp(const int* x, int n)
{
    int sum = 0;

    for (int i = 0; i < n; i++)
        sum += *x++;

    return sum;
}

int main()
{
    int x[] {3, 17, -13, 25, -2, 9, -6, 12, 88, -19};
    int n = sizeof(x) / sizeof(int);

    cout << "Elements of array x" << '\n';

    for (int i = 0; i < n; i++)
        cout << "x[" << i << "] = " << x[i] << '\n';
    cout << '\n';

    int sum1 = CalcArraySumCpp(x, n);
    int sum2 = CalcArraySum_(x, n);

    cout << "sum1 = " << sum1 << '\n';
    cout << "sum2 = " << sum2 << '\n';
    return 0;
}

;------------------------------------------------
;               Ch03_01.asm
;------------------------------------------------

; extern "C" int CalcArraySum_(const int* x, int n)
;
; Returns:      Sum of elements in array x
```

```
        .code
CalcArraySum_ proc

; Initialize sum to zero
        xor eax,eax                     ;sum = 0

; Make sure 'n' is greater than zero
        cmp edx,0
        jle InvalidCount                ;jump if n <= 0

; Sum the elements of the array
@@:     add eax,[rcx]                   ;add next element to total (sum += *x)
        add rcx,4                       ;set pointer to next element (x++)
        dec edx                         ;adjust counter (n -= 1)
        jnz @B                          ;repeat if not done

InvalidCount:
        ret

CalcArraySum_ endp
        end
```

The function `CalcArraySum_` begins with an `xor eax,eax` instruction that initializes the running sum to zero. The `cmp edx,0` and `jle InvalidCount` instructions prevent execution of the summing loop if `n <= 0` is true. Sweeping through the array to sum the elements requires only four instructions. The `add eax,[rcx]` instruction adds the current array element to the running sum in register EAX. Four is then added to register RCX so that it points to the next element in the array. The constant four is used here since the size of each integer in array `x` is four bytes. A `dec edx` (Decrement by 1) instruction subtracts 1 from the counter and updates the state of RFLAGS.ZF. This enables the `jnz` instruction to terminate the loop after all `n` elements have been summed. The instruction sequence employed here to calculate the array element sum is the assembly language equivalent of the `for` loop that's used in function `CalcArraySumCpp`. Here's the output for `Ch03_01`:

```
Elements of array x
x[0] = 3
x[1] = 17
x[2] = -13
x[3] = 25
x[4] = -2
x[5] = 9
x[6] = -6
x[7] = 12
x[8] = 88
x[9] = -19

sum1 = 114
sum2 = 114
```

## Using Elements in Calculations

When working with arrays, it is frequently necessary to define functions that perform element-by-element transformations. The next source code example, named Ch03_02, illustrates an array transformation operation using separate source and destination arrays. It also introduces function prologs and epilogs, and a few new instructions. Listing 3-2 shows the source code for example Ch03_02.

***Listing 3-2.*** Example Ch03_02

```
//------------------------------------------------
//               Ch03_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cassert>

using namespace std;

extern "C" long long CalcArrayValues_(long long* y, const int* x, int a, short b, int n);

long long CalcArrayValuesCpp(long long* y, const int* x, int a, short b,  int n)
{
    long long sum = 0;

    for (int i = 0; i < n; i++)
    {
        y[i] = (long long)x[i] * a + b;
        sum += y[i];
    }

    return sum;
}

int main()
{
    const int a = -6;
    const short b = -13;
    const int x[] {26, 12, -53, 19, 14, 21, 31, -4, 12, -9, 41, 7};
    const int n = sizeof(x) / sizeof(int);

    long long y1[n];
    long long y2[n];

    long long sum_y1 = CalcArrayValuesCpp(y1, x, a, b, n);
    long long sum_y2 = CalcArrayValues_(y2, x, a, b, n);

    cout << "a = " << a << '\n';
    cout << "b = " << b << '\n';
    cout << "n = " << n << "\n\n";
```

```
    for (int i = 0; i < n; i++)
    {
        cout << "i: " << setw(2) << i << "  ";
        cout << "x: " << setw(6) << x[i] << "  ";
        cout << "y1: " << setw(6) << y1[i] << "  ";
        cout << "y2: " << setw(6) << y2[i] << '\n';
    }

    cout << '\n';
    cout << "sum_y1 = " << sum_y1 << '\n';
    cout << "sum_y2 = " << sum_y2 << '\n';

    return 0;
}
```

```
;-------------------------------------------------
;               Ch03_02.asm
;-------------------------------------------------

; extern "C" long long CalcArrayValues_(long long* y, const int* x, int a, short b, int n);
;
; Calculation:  y[i] = x[i] * a + b
;
; Returns:      Sum of the elements in array y.

        .code
CalcArrayValues_ proc frame

; Function prolog
        push rsi                            ;save volatile register rsi
        .pushreg rsi
        push rdi                            ;save volatile register rdi
        .pushreg rdi
        .endprolog

; Initialize sum to zero and make sure 'n' is valid
        xor rax,rax                         ;sum = 0
        mov r11d,[rsp+56]                   ;r11d = n
        cmp r11d,0
        jle InvalidCount                    ;jump if n <= 0

; Initialize source and destination pointers
        mov rsi,rdx                         ;rsi = ptr to array x
        mov rdi,rcx                         ;rdi = ptr to array y

; Load expression constants and array index
        movsxd r8,r8d                       ;r8 = a (sign extended)
        movsx r9,r9w                        ;r9 = b (sign extended)
        xor edx,edx                         ;edx = array index i
```

```
; Repeat until done
@@:     movsxd rcx,dword ptr [rsi+rdx*4]    ;rcx = x[i] (sign extended)
        imul rcx,r8                         ;rcx = x[i] * a
        add rcx,r9                          ;rcx = x[i] * a + b
        mov qword ptr [rdi+rdx*8],rcx       ;y[i] = rcx

        add rax,rcx                         ;update running sum

        inc edx                             ;edx = i + i
        cmp edx,r11d                        ;is i >= n?
        jl @B                               ;jump if i < n

InvalidCount:

; Function epilog
        pop rdi                             ;restore caller's rdi
        pop rsi                             ;restore caller's rsi
        ret
CalcArrayValues_ endp
        end
```

The x86-64 assembly language function `CalcArrayValues_` computes `y[i] = x[i] * a + b`. If you examine the declaration for this function in the C++ code, you will notice that the source array `x` is declared as an `int` while the destination array `y` is declared as `long long`. The other function arguments `a`, `b`, and `n` are declared as `int`, `short`, and `int` respectively. The remainder of the C++ code includes the function `CalcArrayValuesCpp` that also computes the specified array transformation for comparison purposes. It also includes code to display the results.

You may have noticed that in all of the sample source code presented thus far, only a subset of the general-purpose registers have been used. The reason for this is that the Visual C++ calling convention designates each general-purpose register as either volatile or non-volatile. Functions are permitted to use and alter the contents of any volatile register but cannot use a non-volatile register unless it preserves the caller's original value. The Visual C++ calling convention designates registers RAX, RCX, RDX, R8, R9, R10, and R11 as volatile and the remaining general-purpose registers as non-volatile.

The function `CalcArrayValues_` uses non-volatile registers RSI and RDI, which means that their values must be preserved. A function typically saves the values of any non-volatile registers it uses on the stack in a section of code called the prolog. A function epilog contains code that restores the values of any saved non-volatile registers. Function prologs and epilogs are also used to perform other calling-convention initialization tasks and you'll learn about these in Chapter 5.

In the assembly language code for `Ch03_02`, the statement `CalcArrayValues_ proc frame` denotes the start of function `CalcArrayValues_`. Note the `frame` attribute on the `proc` directive. This attribute indicates that `CalcArrayValues_` uses a formal function prolog. It also enables additional directives that must be used whenever a general-purpose register is saved on the stack or whenever a function employs a stack frame pointer. Chapter 5 discusses the `frame` attribute and stack frame pointers in greater detail.

The first x86-64 assembly language instruction of `CalcArrayValues_` is `push rsi` (Push Value onto Stack), which saves the current value in register RSI on the stack. Immediately following this is a `.pushreg rsi` directive. This directive instructs the assembler to save information about `push rsi` instruction in an assembler-maintained table that is used to unwind the stack during exception processing. Using exceptions with assembly language code is not discussed in this book but the calling convention requirements for saving registers on the stack must still be observed. Register RDI is then saved on the stack using a `push rdi` instruction. The required `.pushreg rdi` directive follows next and the subsequent `.endprolog` directive signifies the end of the prolog for `CalcArrayValues_`.

Figure 3-1 illustrates the contents of the stack after execution of the `push rsi` and `push rdi` instructions. Following the function prolog, argument `n` is tested for validity. A `mov r11d,[rsp+56]` loads the value of `n` into register R11D. It is important to note that the displacement used in this instruction to load `n` from the stack is different than in previous examples due to the `push` instructions that were used in the prolog. If the value of `n` is valid, registers RSI and RDI are initialized as pointers to the arrays `x` and `y`. The `movsxd r8,r8d` and `movsx r9,r9w` instructions load argument values `a` and `b` into registers R8 and R9 while the `xor edx,edx` instructions initializes array index `i` to zero.

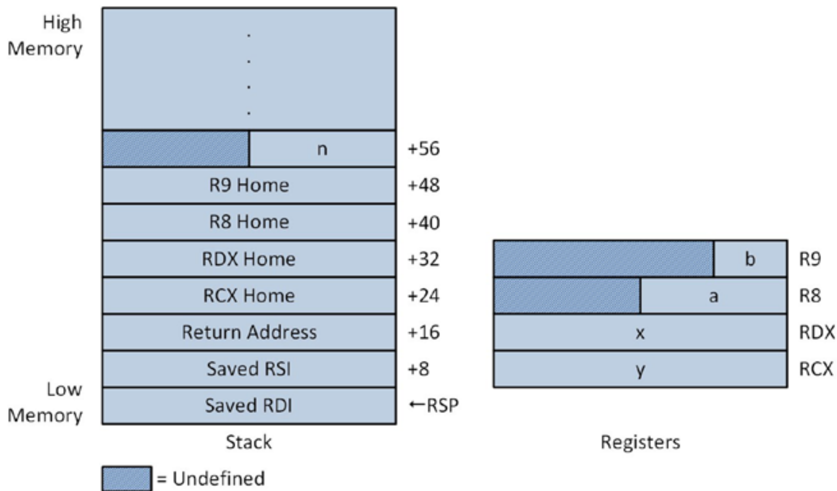



***Figure 3-1.*** *Stack and register contents after prolog in* `CalcArrayValues_`

The processing loop of `CalcArrayValues_` uses a `movsxd rcx,dword ptr [rsi+rdx*4]` instruction to load a sign-extended copy of `x[i]` into register RCX. The ensuing `imul rcx,r8` and `add rcx,r9` instructions calculate `x[i] * a + b` and the `mov qword ptr [rdi+rdx*8]` instruction saves the final result to `y[i]`. Note that in the processing loop, the two move instructions use different scale factors. This is because array `x` and array `y` are declared as `int` and `long long`. The `add rax,rcx` instruction updates a running sum that will be used as the return value. The `inc edx` (Increment by 1) instruction adds 1 to the value that's in register EDX. It also zeros bits 63:32 of register RDX. The reason for using an `inc edx` instruction instead of an `inc rdx` instruction is that the machine language encoding of the former requires less code space. More importantly, it is okay to use an `inc edx` instruction here since the maximum number of elements to be processed is specified by a 32-bit signed integer (`n`) that's already been validated as being greater than zero. The following `cmp edx,r11d` instruction compares the contents of EDX (which is `i`) to `n`, and the processing loop repeats until `i` equals `n`.

After the main processing loop is the epilog for function `CalcArrayValues_`. Recall that in the prolog, the caller's RSI and RDI registers were saved on the stack using two `push` instructions. In the epilog, the instructions `pop rdi` and `pop rsi` (Pop Value from Stack) are used to restore the caller's RDI and RSI registers. The order in which a caller's non-volatile register are popped from the stack in an epilog *must* be the reverse of how they were saved in the prolog. Following non-volatile register restoration is a `ret` instruction that transfers program control back to the calling function. Given the stack operations that occur in a function's prolog and epilog, it should be readily apparent that failure to properly save or restore a non-volatile register is likely to cause a program crash (if the return address is incorrect) or a subtle software bug that may be difficult to pinpoint. Here are the results for example `Ch03_02`.

```
a = -6
b = -13
n = 12

i:  0  x:    26  y1:   -169  y2:   -169
i:  1  x:    12  y1:    -85  y2:    -85
i:  2  x:   -53  y1:    305  y2:    305
i:  3  x:    19  y1:   -127  y2:   -127
i:  4  x:    14  y1:    -97  y2:    -97
i:  5  x:    21  y1:   -139  y2:   -139
i:  6  x:    31  y1:   -199  y2:   -199
i:  7  x:    -4  y1:     11  y2:     11
i:  8  x:    12  y1:    -85  y2:    -85
i:  9  x:    -9  y1:     41  y2:     41
i: 10  x:    41  y1:   -259  y2:   -259
i: 11  x:     7  y1:    -55  y2:    -55

sum_y1 = -858
sum_y2 = -858
```

# Two-Dimensional Arrays

C++ also utilizes a contiguous block of memory to implement a two-dimensional array or matrix. The elements of a C++ matrix in memory are organized using row-major ordering. Row-major ordering arranges the elements of a matrix first by row and then by column. For example, elements of the matrix `int x[3][2]` are stored in memory as follows: `x[0][0]`, `x[0][1]`, `x[1][0]`, `x[1][1]`, `x[2][0]`, and `x[2][1]`. In order to access a specific element in the matrix, a function (or a compiler) must know the starting address of the matrix (i.e., the address of its first element), the row and column indices, the total number of columns, and the size in bytes of each element. Using this information, a function can use simple arithmetic to access a specific element in a matrix as exemplified by the sample code in this section.

## Accessing Elements

Listing 3-3 shows the source code for example `Ch03_03`, which demonstrates how to use x86-64 assembly language to access the elements of a matrix. In this example, the functions `CalcMatrixSquaresCpp` and `CalcMatrixSquares_` perform the following matrix calculation: `y[i][j] = x[j][i] * x[j][i]`. Note that in this expression, the indices `i` and `j` for matrix `x` are intentionally reversed in order to make the code for this example a little more interesting.

***Listing 3-3.*** Example Ch03_03

```
//------------------------------------------------
//               Ch03_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
```

```
using namespace std;

extern "C" void CalcMatrixSquares_(int* y, const int* x, int nrows, int ncols);

void CalcMatrixSquaresCpp(int* y, const int* x, int nrows, int ncols)
{
    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
        {
            int kx = j * ncols + i;
            int ky = i * ncols + j;
            y[ky] = x[kx] * x[kx];
        }
    }
}

int main()
{
    const int nrows = 6;
    const int ncols = 3;
    int y2[nrows][ncols];
    int y1[nrows][ncols];
    int x[nrows][ncols] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 },
                          { 10, 11, 12 }, {13, 14, 15}, {16, 17, 18} };

    CalcMatrixSquaresCpp(&y1[0][0], &x[0][0], nrows, ncols);
    CalcMatrixSquares_(&y2[0][0], &x[0][0], nrows, ncols);

    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
        {
            cout << "y1[" << setw(2) << i << "][" << setw(2) << j << "] = ";
            cout << setw(6) << y1[i][j] << ' ' ;

            cout << "y2[" << setw(2) << i << "][" << setw(2) << j << "] = ";
            cout << setw(6) << y2[i][j] << ' ';

            cout << "x[" << setw(2) << j << "][" << setw(2) << i << "] = ";
            cout << setw(6) <<  x[j][i] << '\n';

            if (y1[i][j] != y2[i][j])
                cout << "Compare failed\n";
        }
    }

    return 0;
}
```

```
;-------------------------------------------------
;               Ch03_03.asm
;-------------------------------------------------

; void CalcMatrixSquares_(int* y, const int* x, int nrows, int ncols);
;
; Calculates:     y[i][j] = x[j][i] * x[j][i]

        .code
CalcMatrixSquares_ proc frame

; Function prolog
        push rsi                                ;save caller's rsi
        .pushreg rsi
        push rdi                                ;save caller's rdi
        .pushreg rdi
        .endprolog

; Make sure nrows and ncols are valid
        cmp r8d,0
        jle InvalidCount                        ;jump if nrows <= 0
        cmp r9d,0
        jle InvalidCount                        ;jump if ncols <= 0

; Initialize pointers to source and destination arrays
        mov rsi,rdx                             ;rsi = x
        mov rdi,rcx                             ;rdi = y
        xor rcx,rcx                             ;rcx = i
        movsxd r8,r8d                           ;r8 = nrows sign extended
        movsxd r9,r9d                           ;r9 = ncols sign extended

; Perform the required calculations
Loop1:
        xor rdx,rdx                             ;rdx = j
Loop2:
        mov rax,rdx                             ;rax = j
        imul rax,r9                             ;rax = j * ncols
        add rax,rcx                             ;rax = j * ncols + i
        mov r10d,dword ptr [rsi+rax*4]          ;r10d = x[j][i]
        imul r10d,r10d                          ;r10d = x[j][i] * x[j][i]

        mov rax,rcx                             ;rax = i
        imul rax,r9                             ;rax = i * ncols
        add rax,rdx                             ;rax = i * ncols + j;
        mov dword ptr [rdi+rax*4],r10d          ;y[i][j] = r10d

        inc rdx                                 ;j += 1
        cmp rdx,r9
        jl Loop2                                ;jump if j < ncols

        inc rcx                                 ;i += 1
        cmp rcx,r8
        jl Loop1                                ;jump if i < nrows
```

```
InvalidCount:

; Function epilog
        pop rdi                                 ;restore caller's rdi
        pop rsi                                 ;restore caller's rsi
        ret

CalcMatrixSquares_ endp
        end
```

The C++ function CalcMatrixSquaresCpp illustrates how to access the elements of a matrix. The first thing to note is that arguments x and y point to the memory blocks that contain their respective matrices. Inside the second for loop, the expression kx = j * ncols + i calculates the offset necessary to access element x[j][i]. Similarly, the expression ky = i * ncols + j calculates the offset for element y[i][j].

The assembly language function CalcMatrixSquares_ implements the same calculations as the C++ code to access elements in matrices x and y. This function begins with a prolog that saves non-volatile registers RSI and RDI using the same instructions and directives as the previous source code example. Next, argument values nrows and ncols are checked to ensure that they're greater than zero. Prior to the start of the nested processing loops, registers RSI and RDI are initialized as pointers to x and y. Registers RCX and RDX are also primed as the loop index variables and perform the same functions as variables i and j in the C++ code. This is followed by two movsxd instructions that load sign-extended copies of nrows and ncols into registers R8 and R9.

The section of code that accesses element x[j][i] begins with a mov rax,rdx instruction that copies j into register RAX. This is followed by the instructions imul rax,r9 and add rax,rcx, which compute the value j * ncols + i. The ensuing mov r10d,dword ptr [rsi+rax*4] instruction loads register R10D with x[j][i] and the imul r10d,r10d instruction squares this value. A similar sequence of instructions is used to calculate the offset i * ncols + j that's needed for y[i][j]. The mov dword ptr [rdi+rax*4],r10d instruction completes execution of the expression y[i][j] = x[j][i] * x[j][i]. Like the corresponding C++ code, the nested processing loops in CalcMatixSquares_ continue executing until the index counters j and i (registers RDX and RCX) reach their respective termination values. The final two pop instructions restore registers RDI and RSI from the stack prior to execution of the ret instruction. The output for example Ch03_03 is shown here.

```
y1[ 0][ 0] =      1 y2[ 0][ 0] =      1 x[ 0][ 0] =      1
y1[ 0][ 1] =     16 y2[ 0][ 1] =     16 x[ 1][ 0] =      4
y1[ 0][ 2] =     49 y2[ 0][ 2] =     49 x[ 2][ 0] =      7
y1[ 1][ 0] =      4 y2[ 1][ 0] =      4 x[ 0][ 1] =      2
y1[ 1][ 1] =     25 y2[ 1][ 1] =     25 x[ 1][ 1] =      5
y1[ 1][ 2] =     64 y2[ 1][ 2] =     64 x[ 2][ 1] =      8
y1[ 2][ 0] =      9 y2[ 2][ 0] =      9 x[ 0][ 2] =      3
y1[ 2][ 1] =     36 y2[ 2][ 1] =     36 x[ 1][ 2] =      6
y1[ 2][ 2] =     81 y2[ 2][ 2] =     81 x[ 2][ 2] =      9
y1[ 3][ 0] =     16 y2[ 3][ 0] =     16 x[ 0][ 3] =      4
y1[ 3][ 1] =     49 y2[ 3][ 1] =     49 x[ 1][ 3] =      7
y1[ 3][ 2] =    100 y2[ 3][ 2] =    100 x[ 2][ 3] =     10
y1[ 4][ 0] =     25 y2[ 4][ 0] =     25 x[ 0][ 4] =      5
y1[ 4][ 1] =     64 y2[ 4][ 1] =     64 x[ 1][ 4] =      8
y1[ 4][ 2] =    121 y2[ 4][ 2] =    121 x[ 2][ 4] =     11
y1[ 5][ 0] =     36 y2[ 5][ 0] =     36 x[ 0][ 5] =      6
y1[ 5][ 1] =     81 y2[ 5][ 1] =     81 x[ 1][ 5] =      9
y1[ 5][ 2] =    144 y2[ 5][ 2] =    144 x[ 2][ 5] =     12
```

## Row-Column Calculations

Listing 3-4 shows the source code for example Ch03_04, which demonstrates how to sum the rows and columns of a matrix. The C++ code in Listing 3-4 includes a couple of ancillary functions named Init and PrintResult that perform matrix initialization and display results. The function CalcMatrixRowColSumsCpp illustrates the summing algorithm. This function sweeps through matrix x using a set of nested for loops. During each iteration, it adds the matrix element x[i][j] to the appropriate entries in the arrays row_sums and col_sums. Function CalcMatrixRowColSumsCpp also uses the same arithmetic that you saw in the previous example to determine the offset of each matrix element.

***Listing 3-4.*** Example Ch03_04

```
//------------------------------------------------
//              Ch03_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>

using namespace std;

extern "C" int CalcMatrixRowColSums_(int* row_sums, int* col_sums, const int* x, int nrows,
int ncols);

void Init(int* x, int nrows, int ncols)
{
    unsigned int seed = 13;
    uniform_int_distribution<> d {1, 200};
    default_random_engine rng {seed};

    for (int i = 0; i < nrows * ncols; i++)
        x[i] = d(rng);
}

void PrintResult(const char* msg, const int* row_sums, const int* col_sums, const int* x,
int nrows, int ncols)
{
    const int w = 6;
    const char nl = '\n';

    cout << msg;
    cout << "-----------------------------------------\n";

    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
            cout << setw(w) << x[i* ncols + j];
        cout << "  " << setw(w) << row_sums[i] << nl;
    }
```

```
    cout << nl;

    for (int i = 0; i < ncols; i++)
        cout << setw(w) << col_sums[i];
    cout << nl;
}

int CalcMatrixRowColSumsCpp(int* row_sums, int* col_sums, const int* x, int nrows, int ncols)
{
    int rc = 0;

    if (nrows > 0 && ncols > 0)
    {
        for (int j = 0; j < ncols; j++)
            col_sums[j] = 0;

        for (int i = 0; i < nrows; i++)
        {
            row_sums[i] = 0;
            int k = i * ncols;

            for (int j = 0; j < ncols; j++)
            {
                int temp = x[k + j];
                row_sums[i] += temp;
                col_sums[j] += temp;
            }
        }

        rc = 1;
    }

    return rc;
}

int main()
{
    const int nrows = 7;
    const int ncols = 5;
    int x[nrows][ncols];

    Init((int*)x, nrows, ncols);

    int row_sums1[nrows], col_sums1[ncols];
    int row_sums2[nrows], col_sums2[ncols];
```

```
    const char* msg1 = "\nResults using CalcMatrixRowColSumsCpp\n";
    const char* msg2 = "\nResults using CalcMatrixRowColSums_\n";

    int rc1 = CalcMatrixRowColSumsCpp(row_sums1, col_sums1, (int*)x, nrows, ncols);
    int rc2 = CalcMatrixRowColSums_(row_sums2, col_sums2, (int*)x, nrows, ncols);

    if (rc1 == 0)
        cout << "CalcMatrixRowSumsCpp failed\n";
    else
        PrintResult(msg1, row_sums1, col_sums1, (int*)x, nrows, ncols);

    if (rc2 == 0)
        cout << "CalcMatrixRowSums_ failed\n";
    else
        PrintResult(msg2, row_sums2, col_sums2, (int*)x, nrows, ncols);

    return 0;
}

;-------------------------------------------------
;               Ch03_04.asm
;-------------------------------------------------

; extern "C" int CalcMatrixRowColSums_(int* row_sums, int* col_sums, const int* x, int
nrows, int ncols)
;
; Returns:      0 = nrows <= 0 or ncols <= 0, 1 = success

        .code
CalcMatrixRowColSums_ proc frame

; Function prolog
        push rbx                            ;save caller's rbx
        .pushreg rbx
        push rsi                            ;save caller's rsi
        .pushreg rsi
        push rdi                            ;save caller's rdi
        .pushreg rdi
        .endprolog

; Make sure nrows and ncols are valid
        xor eax,eax                         ;set error return code

        cmp r9d,0
        jle InvalidArg                      ;jump if nrows <= 0

        mov r10d,[rsp+64]                   ;r10d = ncols
        cmp r10d,0

        jle InvalidArg                      ;jump if ncols <= 0
```

```
; Initialize elements of col_sums array to zero
        mov rbx,rcx                         ;temp save of row_sums
        mov rdi,rdx                         ;rdi = col_sums
        mov ecx,r10d                        ;rcx = ncols
        xor eax,eax                         ;eax = fill value
        rep stosd                           ;fill array with zeros

; The code below uses the following registers:
;   rcx = row_sums          rdx = col_sums
;   r9d = nrows             r10d = ncols
;   eax = i                 ebx = j
;   edi = i * ncols         esi = i * ncols + j
;   r8 = x                  r11d = x[i][j]

; Initialize outer loop variables.
        mov rcx,rbx                         ;rcx = row_sums
        xor eax,eax                         ;i = 0

Lp1:    mov dword ptr [rcx+rax*4],0         ;row_sums[i] = 0
        xor ebx,ebx                         ;j = 0
        mov edi,eax                         ;edi = i
        imul edi,r10d                       ;edi = i * ncols

; Inner loop
Lp2:    mov esi,edi                         ;esi = i * ncols
        add esi,ebx                         ;esi = i * ncols + j
        mov r11d,[r8+rsi*4]                 ;r11d = x[i * ncols + j]
        add [rcx+rax*4],r11d                ;row_sums[i] += x[i * ncols + j]
        add [rdx+rbx*4],r11d                ;col_sums[j] += x[i * ncols + j]

; Is the inner loop finished?
        inc ebx                             ;j += 1
        cmp ebx,r10d
        jl Lp2                              ;jump if j < ncols

; Is the outer loop finished?
        inc eax                             ;i += 1
        cmp eax,r9d
        jl Lp1                              ;jump if i < nrows

        mov eax,1                           ;set success return code

; Function epilog
InvalidArg:
        pop rdi                             ;restore NV registers and return
        pop rsi
        pop rbx
        ret
CalcMatrixRowColSums_ endp
        end
```

The assembly language function CalcMatrixRowColSums_ implements the same algorithm as the C++ code. Following the function prolog, the arguments nrows and ncols are tested for validity. Note that the argument ncols was passed on the stack, as illustrated in Figure 3-2. The elements of col_sums are then initialized to zero using a rep stosd (Repeat Store String Doubleword) instruction. This instruction stores the contents of EAX, which was initialized to zero, to the memory location specified by RDI; it then adds four to RDI so that it points to the next array element. The rep mnemonic is an instruction prefix that tells the processor to repeat execution of the stosd instruction. Specifically, this prefix instructs the CPU to decrement RCX by 1 following each store action and repeat execution of the stosd instruction until RCX equals zero. You'll take a closer look at the x86-64 string processing instructions later in this chapter.

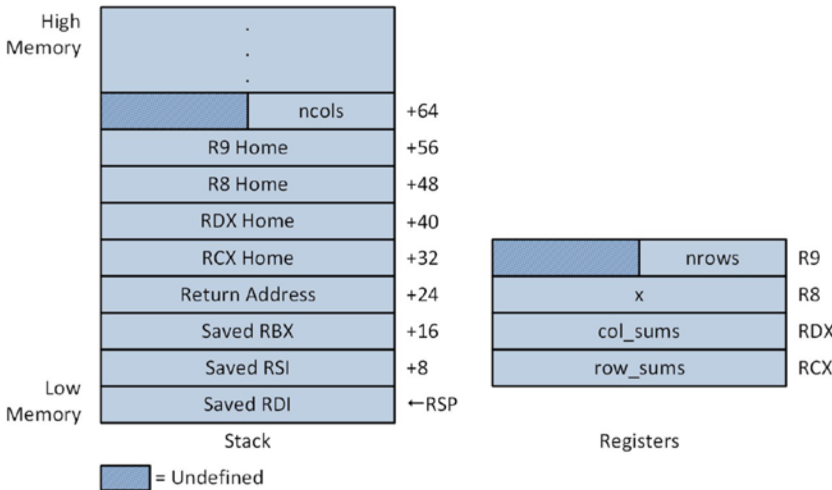



***Figure 3-2.*** *Stack and register contents after prolog in CalcMatrixRowColSums_*

In function CalcMatrixRowColSums_, R8 holds the base address of the matrix x. Registers EAX and EBX contain the row and column indices i and j, respectively. Each outer loop starts by initializing row_sums[i] (RCX points to row_sums) to zero and calculating the intermediate value i * ncols (R10D contains ncols). Within the inner loop, the final offset of matrix element x[i][j] is calculated. A mov r11d,[r8+rsi*4] instruction loads x[i][j] into R11D. The instructions add [rcx+rax*4],r11d and add[rdx+rbx*4],r11d update the totals for row_sums[i] and col_sums[j]. Note that these two instructions use destination operands in memory instead of registers. Figure 3-3 illustrates the memory addressing that's used to reference elements in x, row_sums, and col_sums.

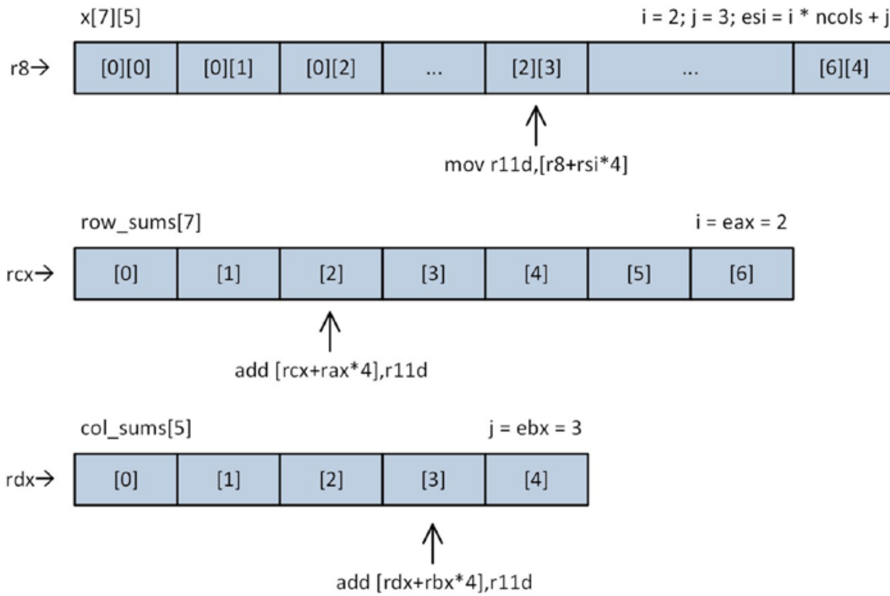



***Figure 3-3.*** *Memory addressing used in function* `CalcMatrixRowColSums_`

The nested processing loops in `CalcMatrixRowColSums_` repeat until all of the elements in matrix `x` have been added to the correct elements in both `row_sums` and `col_sums`. Note that this function uses 32-bit registers for its counters and indices. Using 32-bit registers often requires less code space than 64-bit registers, as discussed earlier in this chapter. The code in `CalcMatrixRowColSums_` also exploits `BaseReg+IndexReg*ScaleFactor` memory addressing, which simplifies the loading of elements from matrix `x` and the updating of elements in both `row_sums` and `col_sums`. Here are the results for example `Ch03_04`.

```
Results using CalcMatrixRowColSumsCpp
-----------------------------------------
   19   153   155   177   119     623
   27    37   130   165    99     458
   68    27    61     7   195     358
  127   143   110    86    43     509
  114    84   109   179    17     503
  140   126    28    52    55     401
  126   100   186   115   145     672

  621   670   779   781   673

Results using CalcMatrixRowColSums_
-----------------------------------------
   19   153   155   177   119     623
   27    37   130   165    99     458
   68    27    61     7   195     358
  127   143   110    86    43     509
```

```
   114    84   109   179    17      503
   140   126    28    52    55      401
   126   100   186   115   145      672

   621   670   779   781   673
```

# Structures

A structure is a programming language construct that facilitates the definition of new data types using one or more existing data types. In this section, you'll learn how to define and use a common structure in both a C++ and x86-64 assembly language function. You'll also learn how to deal with potential semantic issues that can arise when working with a common structure that's manipulated by software functions written using different programming languages.

In C++ a structure is equivalent to a class. When a data type is defined using the keyword `struct` instead of `class`, all members are public by default. A C++ `struct` that's declared sans any member functions or operators is equivalent to a C-style structure such as `typedef struct { ... } MyStruct;`. C++ structure declarations are usually placed in a header (.h) file so they can be easily referenced by multiple C++ files. The same technique also can be employed to declare and reference structures that are used in assembly language code. Unfortunately, it is not possible to declare a single structure in a header file and include this file in both C++ and assembly-language source code files. If you want to use the "same" structure in both C++ and assembly language code, it must be declared twice and both declarations must be semantically equivalent.

Listing 3-5 shows the C++ and x86 assembly language source code for example `Ch03_05`. In the C++ code, a simple structure named `TestStruct` is declared. This structure uses sized integer types instead of the more common C++ types to highlight the exact size of each member. The other noteworthy detail regarding `TestStruct` is the inclusion of the structure member `Pad8`. While not explicitly required, the presence of this member helps document the fact that the C++ compiler defaults to aligning structure members to their natural boundaries. The assembly language version of `TestStruct` looks similar to its C++ counterpart. The biggest difference between the two is that the assembler *does not* automatically align structure members to their natural boundaries. Here the definition of `Pad8` is required; without the member `Pad8`, the C++ and assembly language versions would be semantically different. The `?` symbol that's included with each data element declaration notifies the assembler to perform storage allocation only and is customarily used to remind the programmer that structure members are always uninitialized.

***Listing 3-5.*** Example Ch03_05

```
//------------------------------------------------
//               Ch03_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>

using namespace std;

struct TestStruct
{
    int8_t  Val8;
    int8_t  Pad8;
```

```
    int16_t Val16;
    int32_t Val32;
    int64_t Val64;
};

extern "C" int64_t CalcTestStructSum_(const TestStruct* ts);

int64_t CalcTestStructSumCpp(const TestStruct* ts)
{
    return ts->Val8 + ts->Val16 + ts->Val32 + ts->Val64;
}

int main()
{
    TestStruct ts;

    ts.Val8 = -100;
    ts.Val16 = 2000;
    ts.Val32 = -300000;
    ts.Val64 = 40000000000;

    int64_t sum1 = CalcTestStructSumCpp(&ts);
    int64_t sum2 = CalcTestStructSum_(&ts);

    cout << "ts1.Val8 =  " << (int)ts.Val8 << '\n';
    cout << "ts1.Val16 = " << ts.Val16 << '\n';
    cout << "ts1.Val32 = " << ts.Val32 << '\n';
    cout << "ts1.Val16 = " << ts.Val64 << '\n';
    cout << '\n';
    cout << "sum1 = " << sum1 << '\n';
    cout << "sum2 = " << sum2 << '\n';

    return 0;
}

;-------------------------------------------------
;               Ch03_05.asm
;-------------------------------------------------

TestStruct struct
Val8    byte ?
Pad8    byte ?
Val16   word ?
Val32   dword ?
Val64   qword ?
TestStruct ends

; extern "C" int64_t CalcTestStructSum_(const TestStruct* ts);
;
; Returns:      Sum of structure's values as a 64-bit integer.
```

```
        .code
CalcTestStructSum_ proc

; Compute ts->Val8 + ts->Val16, note sign extension to 32-bits
        movsx eax,byte ptr [rcx+TestStruct.Val8]
        movsx edx,word ptr [rcx+TestStruct.Val16]
        add eax,edx

; Sign extend previous result to 64 bits
        movsxd rax,eax

; Add ts->Val32 to sum
        movsxd rdx,[rcx+TestStruct.Val32]
        add rax,rdx

; Add ts->Val64 to sum
        add rax,[rcx+TestStruct.Val64]
        ret

CalcTestStructSum_ endp
        end
```

The C++ function `CalcTestStructSumCpp` sums the members of the `TestStruct` instance that's passed to it. The x86 assembly language function `CalcTestStructSum_` performs the same operation. The `movsx eax,byte ptr [rcx+TestStruct.Val8]` and `movsx edx,word ptr [rcx+TestStruct.Val16]` instructions load sign-extended copies of structure members `TestStruct.Val8` and `TestStruct.Val16` into registers EAX and EDX, respectively. These instructions also illustrate the syntax that is required to reference a structure member in an assembly language instruction. From the perspective of the assembler, the `movsx` instructions are instances of `BaseReg+Disp` memory addressing since the assembler ultimately converts structure members `TestStruct.Val8` and `TestStruct.Val16` into constant displacement values.

Next, the function `CalcTestStructSum_` uses an `add eax,edx` instruction to sum structure members `TestStruct.Val8` and `TestStruct.Val16`. It then sign-extends this sum to 64 bits using a `movsxd rax,eax` instruction. The next instruction, `movsxd rdx,[rcx+TestStruct.Val32]`, loads a sign-extended copy `TestStruct.Val32` into RDX and adds this value to intermediate sum that's in RAX. The instruction `add rax,[rcx+TestStruct.Val64]` adds the value structure member `TestStruct.Val64` to the running sum in RAX, which generates the final result. The Visual C++ calling convention requires 64-bit return values to be placed in register RAX. Since the final result is already in the required register, no additional `mov` instructions are necessary. Here's the output for example `Ch03_05`.

---

```
ts1.Val8 =  -100
ts1.Val16 = 2000
ts1.Val32 = -300000
ts1.Val16 = 40000000000

sum1 = 39999701900
sum2 = 39999701900
```

---

# Strings

The x86-64 instruction set includes several useful instructions that process and manipulate strings. In x86 parlance, a string is a contiguous sequence of bytes, words, doublewords, or quadwords. Programs can use the x86 string instructions to process conventional text strings such as "Hello, World." They also can be employed to perform operations using the elements of an array or similarly-ordered data in memory. In this section, you'll examine some sample code that demonstrates how to use the x86-64 string instructions with text strings and integer arrays.

## Counting Characters

Listing 3-6 shows the C++ and assembly language code, for example Ch03_06. This example explains how to use the lodsb (Load String Byte) instruction to count the number of character occurrences in a text string.

***Listing 3-6.*** Example Ch03_06

```
//------------------------------------------------
//               Ch03_06.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>

using namespace std;

extern "C" unsigned long long CountChars_(const char* s, char c);

int main()
{
    const char nl = '\n';
    const char* s0 = "Test string: ";
    const char* s1 = "  SearchChar: ";
    const char* s2 = " Count: ";

    char c;
    const char* s;

    s = "Four score and seven seconds ago, ...";
    cout << nl << s0 << s << nl;

    c = 's';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'o';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'z';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'F';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = '.';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
```

```
    s = "Red Green Blue Cyan Magenta Yellow";
    cout << nl << s0 << s << nl;

    c = 'e';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'w';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'l';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'Q';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'n';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;

    return 0;
}

;-------------------------------------------------
;               Ch03_06.asm
;-------------------------------------------------

; extern "C" unsigned long long CountChars_(const char* s, char c);
;
; Description:  This function counts the number of occurrences
;               of a character in a string.
;
; Returns:      Number of occurrences found.

        .code
CountChars_ proc frame

; Save non-volatile registers
        push rsi                            ;save caller's rsi
        .pushreg rsi
        .endprolog

; Load parameters and initialize count registers
        mov rsi,rcx                         ;rsi = s
        mov cl,dl                           ;cl = c
        xor edx,edx                         ;rdx = Number of occurrences
        xor r8d,r8d                         ;r8 = 0 (required for add below)

; Repeat loop until the entire string has been scanned
@@:     lodsb                               ;load next char into register al
        or al,al                            ;test for end-of-string
        jz @F                               ;jump if end-of-string found
        cmp al,cl                           ;test current char
        sete r8b                            ;r8b = 1 if match, 0 otherwise
        add rdx,r8                          ;update occurrence count
        jmp @B

@@:     mov rax,rdx                        ;rax = number of occurrences
```

```
; Restore non-volatile registers and return
        pop rsi
        ret
CountChars_ endp
        end
```

The assembly language function CountChars_ accepts two arguments: a text string pointer s and a search character c. Both arguments are of type char, which means that each text string character and the search character require one byte of storage. The function CountChars_ starts with a function prolog that saves the caller's RSI on the stack. It then loads the text string pointer s into RSI and the search character c into register CL. An xor edx,edx instruction initializes register RDX to zero for use as a character occurrence counter. The processing loop uses the lodsb instruction to read each text string character. This instruction loads register AL with the contents of the memory pointed to by RSI; it then increments RSI by one so that it points to the next character.

Next, the function CountChars_ uses an or al,al instruction to test for the end-of-string character ('\0'). This instruction sets the zero flag (RFLAGS.ZF) if register AL is equal to zero. If the end-of-string character is not found, a cmp al,cl instruction compares the current text string character to the search character. The subsequent sete r8b (Set Byte if Equal) instructions loads register R8B with a value of one if a character match is found; otherwise R8B is set to zero. One important item that should be noted here is that the sete instruction does not modify the upper 56 bits of register R8. Whenever the destination operand of an instruction is an 8-bit or 16-bit register, the upper 56 or 48 bits of the corresponding 64-bit register are unaffected by the specified operation. Following the sete instruction is an add rdx,r8 instruction that updates the occurrence counter. This process is repeated until the end-of-string character is found. Following completion of the text string scan, the final occurrence count is moved into register RAX and returned to the caller. The output for example Ch03_06 is as follows:

```
Test string: Four score and seven seconds ago, ...
  SearchChar: s Count: 4
  SearchChar: o Count: 4
  SearchChar: z Count: 0
  SearchChar: F Count: 1
  SearchChar: . Count: 3

Test string: Red Green Blue Cyan Magenta Yellow
  SearchChar: e Count: 6
  SearchChar: w Count: 1
  SearchChar: l Count: 3
  SearchChar: Q Count: 0
  SearchChar: n Count: 3
```

A version of CountChars_ that processes strings of type wchar_t instead of char can be easily created by changing the lodsb instruction to a lodsw (Load String Word) instruction. 16-bit registers would also need to be used instead of 8-bit registers for the character matching instructions. The last character of an x86 string instruction mnemonic indicates the size of the operand that is processed.

## String Concatenation

The concatenation of two text strings is a common operation that is performed by many programs. C++ programs can use the library functions `strcat`, `strcat_s`, `wcscat`, and `wcscat_s` to concatenate two strings. One limitation of these functions is that they can process only a single source string. Multiple calls are necessary to concatenate several strings together. The next example, named `Ch03_07`, demonstrates how to use the `scas` (Scan String) and `movs` (Move String) instructions to concatenate multiple strings. Listing 3-7 shows the C++ and x86-assembly language source code.

***Listing 3-7.*** Example Ch03_07

```
//------------------------------------------------
//               Ch03_07.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;

extern "C" size_t ConcatStrings_(char* des, size_t des_size, const char* const* src, size_t
src_n);

void PrintResult(const char* msg, const char* des, size_t des_len, const char* const* src,
size_t src_n)
{
    string s_test;
    const char nl = '\n';

    cout << nl << "Test case: " << msg << nl;
    cout << "  Original Strings" << nl;

    for (size_t i = 0; i < src_n; i++)
    {
        const char* s1 = (strlen(src[i]) == 0) ? "<empty string>" : src[i];
        cout << "    i:" << i << " " << s1 << nl;

        s_test += src[i];
    }

    const char* s2 = (strlen(des) == 0) ? "<empty string>" : des;

    cout << "  Concatenated Result" << nl;
    cout << "    " << s2 << nl;

    if (s_test != des)
        cout << "  Error - test string compare failed" << nl;
}
```

```
int main()
{
    // Destination buffer size OK
    const char* src1[] = { "One ", "Two ", "Three ", "Four" };
    size_t src1_n = sizeof(src1) / sizeof(char*);
    const size_t des1_size = 64;
    char des1[des1_size];

    size_t des1_len = ConcatStrings_(des1, des1_size, src1, src1_n);
    PrintResult("destination buffer size OK", des1, des1_len, src1, src1_n);

    // Destination buffer too small
    const char* src2[] = { "Red ", "Green ", "Blue ", "Yellow " };
    size_t src2_n = sizeof(src2) / sizeof(char*);
    const size_t des2_size = 16;
    char des2[des2_size];

    size_t des2_len = ConcatStrings_(des2, des2_size, src2, src2_n);
    PrintResult("destination buffer too small", des2, des2_len, src2, src2_n);

    // Empty source string
    const char* src3[] = { "Plane ", "Car ", "", "Truck ", "Boat ", "Train ", "Bicycle " };
    size_t src3_n = sizeof(src3) / sizeof(char*);
    const size_t des3_size = 128;
    char des3[des3_size];

    size_t des3_len = ConcatStrings_(des3, des3_size, src3, src3_n);
    PrintResult("empty source string", des3, des3_len, src3, src3_n);

    // All strings empty
    const char* src4[] = { "", "", "", "" };
    size_t src4_n = sizeof(src4) / sizeof(char*);
    const size_t des4_size = 42;
    char des4[des4_size];

    size_t des4_len = ConcatStrings_(des4, des4_size, src4, src4_n);
    PrintResult("all strings empty", des4, des4_len, src4, src4_n);

    // Minimum des_size
    const char* src5[] = { "1", "22", "333", "4444" };
    size_t src5_n = sizeof(src5) / sizeof(char*);
    const size_t des5_size = 11;
    char des5[des5_size];

    size_t des5_len = ConcatStrings_(des5, des5_size, src5, src5_n);
    PrintResult("minimum des_size", des5, des5_len, src5, src5_n);

    return 0;
}
```

```
;-------------------------------------------------
;               Ch03_07.asm
;-------------------------------------------------

; extern "C" size_t ConcatStrings_(char* des, size_t des_size, const char* const* src,
size_t src_n);
;
; Returns:      -1      Invalid 'des_size'
;               n >= 0  Length of concatenated string

        .code
ConcatStrings_ proc frame

; Save non-volatile registers
        push rbx
        .pushreg rbx
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
        .endprolog

; Make sure des_size and src_n are valid
        mov rax,-1                          ;set error code

        test rdx,rdx                        ;test des_size
        jz InvalidArg                       ;jump if des_size is 0

        test r9,r9                          ;test src_n
        jz InvalidArg                       ;jump if src_n is 0

; Registers used processing loop below
;   rbx = des               rdx = des_size
;   r8 = src                r9 = src_n
;   r10 = des_index         r11 = i
;   rcx = string length
;   rsi, rdi = pointers for scasb & movsb instructions

; Perform required initializations
        xor r10,r10                         ;des_index = 0
        xor r11,r11                         ;i = 0
        mov rbx,rcx                         ;rbx = des
        mov byte ptr [rbx],0                ;*des = '\0'

; Repeat loop until concatenation is finished
Loop1:  mov rax,r8                          ;rax = 'src'
        mov rdi,[rax+r11*8]                 ;rdi = src[i]
        mov rsi,rdi                         ;rsi = src[i]

; Compute length of s[i]
        xor eax,eax
        mov rcx,-1
```

```
        repne scasb                         ;find '\0'
        not rcx
        dec rcx                             ;rcx = len(src[i])

; Compute des_index + src_len
        mov rax,r10                         ;rax = des_index
        add rax,rcx                         ;des_index + len(src[i])
        cmp rax,rdx                         ;is des_index + src_len >= des_size?
        jge Done                            ;jump if des is too small

; Update des_index
        mov rax,r10                         ;des_index_old = des_index
        add r10,rcx                         ;des_index += len(src[i])

; Copy src[i] to &des[des_index] (rsi already contains src[i])
        inc rcx                             ;rcx = len(src[i]) + 1
        lea rdi,[rbx+rax]                   ;rdi = &des[des_index_old]
        rep movsb                           ;perform string move

; Update i and repeat if not done
        inc r11                             ;i += 1
        cmp r11,r9
        jl Loop1                            ;jump if i < src_n

; Return length of concatenated string

Done:   mov rax,r10                        ;rax = des_index (final length)

; Restore non-volatile registers and return

InvalidArg:
        pop rdi
        pop rsi
        pop rbx
        ret
ConcatStrings_ endp
        end
```

Let's begin by examining the C++ code in Listing 3-7. It starts with a declaration statement for the assembly language function `ConcatStrings_`, which includes four parameters: `des` is the destination buffer for the final string; `des_size` is the size of `des` in characters; and parameter `src` points to an array that contains pointers to `src_n` text strings. In 64-bit Visual C++ programs, the type `size_t` is equivalent to a 64-bit unsigned integer. The function `ConcatStrings_` returns the length of `des` or -1 if the supplied value for `des_size` is less than or equal to zero.

The test cases presented in `main` illustrate use of `ConcatStrings_`. If, for example, `src` points to a text string array consisting of "Red", "Green", "Blue", the final string in `des` is "RedGreenBlue" provided `des` is large enough to contain the result. If `des_size` is insufficient, `ConcatStrings_` produces a partially concatenated string. For example, a `des_size` equal to 10 would yield "RedGreen" as the final string.

Following its prolog, the function `ConcatStrings_` checks argument value `des_size` for validity using a `test rdx,rdx` instruction. This instruction performs a bitwise AND of its two operands and sets the parity (RFLAGS.PF), sign (RFLAGS.SF), and zero (RFLAGS.ZF) flags based on the result (the carry (RFLAGS.CF) and overflow (RFLAGS.OF) are set to zero). The result of the bitwise AND operation is

not saved. The `test` instruction is often used as an alternative to the `cmp` instruction, especially when a function needs to ascertain if a value is less than, equal to, or greater than zero. Using a `test` instruction may also be more efficient in terms of code space. In this instance, the `test rdx,rdx` instruction requires fewer opcode bytes than a `cmp rdx,0` instruction. Register initialization is carried out next prior to the start of the concatenation processing loop.

The subsequent block of instructions marks the top of the concatenation loop that begins by loading registers RSI and RDI with a pointer to string `src[i]`. The length of `src[i]` is determined next using a `repne scasb` instruction in conjunction with several support instructions. The `repne` (Repeat String Operation While not Equal) is an instruction prefix that repeats execution of a string instruction while the condition `RCX != 0 && RFLAGS.ZF == 0` is true. The exact operation of the `repne scasb` (Scan String Byte) combination is as follows: If RCX is not zero, the `scasb` instruction compares the string character pointed to by RDI to the contents of register AL and sets the status flags according to the results. Register RDI is then automatically incremented by one so that it points to the next character and a count of one is subtracted from RCX. This string-processing operation is repeated as long as the aforementioned test conditions remain true; otherwise, the repeat string operation terminates.

Prior to use of the `repne scasb` instruction, register RCX was loaded with -1. Upon completion of `repne scasb`, register RCX contains -(L + 2), where L denotes the actual length of string `src[i]`. The value L is calculated using a `not rcx` (One's Complement Negation) instruction followed by a `dec rcx` (Decrement by 1) instruction, which is equal to subtracting 2 from the two's complement negation of -(L + 2). It should be noted that the instruction sequence used here to calculate the length of a text string is a well-known technique that dates back to the 8086 CPU.

Following the computation of `len(src[i])`, a check is made to verify that the string `src[i]` will fit into the destination buffer. If the sum `des_index + len(src[i])` is greater than or equal to `des_size`, the function terminates. Otherwise, `len(src[i])` is added to `des_index` and string `src[i]` is copied to the correct position in `des` using a `rep movsb` (Repeat Move String Byte) instruction.

The `rep movsb` instruction copies the string pointed to by RSI to the memory location pointed to by RDI using the length specified in RCX. An `inc rcx` instruction is executed before the string copy to ensure that the end-of-string terminator `'\0'` is also transferred to `des`. Register RDI is initialized to the correct offset in `des` using a `lea rdi,[rbx+rax]` (Load Effective Address) instruction, which computes the address of the specified source operand (i.e., `lea` calculates RDI = RBX + RAX). The concatenation loop can use a `lea` instruction since register RBX points to the start of `des` and RAX contains the value of `des_index` prior to its addition with `len(src[i])`. Subsequent to the string copy operation, the value of `i` is updated and if it's less than `src_n`, the concatenation loop is repeated. Following completion of the concatenation operation, register RAX is loaded with `des_index`, which is the length of the final string in `des`. Here's the output of example `Ch03_07`.

```
Test case: destination buffer size OK
  Original Strings
    i:0 One
    i:1 Two
    i:2 Three
    i:3 Four
  Concatenated Result
    One Two Three Four

Test case: destination buffer too small
  Original Strings
    i:0 Red
    i:1 Green
    i:2 Blue
    i:3 Yellow
```

```
  Concatenated Result
    Red Green Blue
  Error - test string compare failed

Test case: empty source string
  Original Strings
    i:0 Plane
    i:1 Car
    i:2 <empty string>
    i:3 Truck
    i:4 Boat
    i:5 Train
    i:6 Bicycle
  Concatenated Result
    Plane Car Truck Boat Train Bicycle

Test case: all strings empty
  Original Strings
    i:0 <empty string>
    i:1 <empty string>
    i:2 <empty string>
    i:3 <empty string>
  Concatenated Result
    <empty string>

Test case: minimum des_size
  Original Strings
    i:0 1
    i:1 22
    i:2 333
    i:3 4444
  Concatenated Result
    1223334444
```

## Comparing Arrays

Besides text strings, the x86 string instructions also can be used to perform operations on other contiguously-ordered data elements. The next source code example demonstrates how to use the cmps (Compare String Operands) instruction to compare the elements of two arrays. Listing 3-8 contains the C++ and x86-64 assembly language source code for example Ch03_08.

***Listing 3-8.*** Example Ch03_08

```
//------------------------------------------------
//               Ch03_08.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <memory>
```

```
using namespace std;

extern "C" long long CompareArrays_(const int* x, const int* y, long long n);

void Init(int* x, int* y, long long n, unsigned int seed)
{
    uniform_int_distribution<> d {1, 10000};
    default_random_engine rng {seed};

    for (long long i = 0; i < n; i++)
        x[i] = y[i] = d(rng);
}

void PrintResult(const char* msg, long long result1, long long result2)
{
    cout << msg << '\n';
    cout << "  expected = " << result1;
    cout << "  actual = " << result2 << "\n\n";
}

int main()
{
    // Allocate and initialize the test arrays
    const long long n = 10000;
    unique_ptr<int[]> x_array {new int[n]};
    unique_ptr<int[]> y_array {new int[n]};
    int* x = x_array.get();
    int* y = y_array.get();

    Init(x, y, n, 11);

    cout << "Results for CompareArrays_ - array_size = " << n << "\n\n";

    long long result;

    // Test using invalid array size
    result = CompareArrays_(x, y, -n);
    PrintResult("Test using invalid array size", -1, result);

    // Test using first element mismatch
    x[0] += 1;
    result = CompareArrays_(x, y, n);
    x[0] -= 1;
    PrintResult("Test using first element mismatch", 0, result);

    // Test using middle element mismatch
    y[n / 2] -= 2;
    result = CompareArrays_(x, y, n);
    y[n / 2] += 2;
    PrintResult("Test using middle element mismatch", n / 2, result);
```

```
    // Test using last element mismatch
    x[n - 1] *= 3;
    result = CompareArrays_(x, y, n);
    x[n - 1] /= 3;
    PrintResult("Test using last element mismatch", n - 1, result);

    // Test with identical elements in each array
    result = CompareArrays_(x, y, n);
    PrintResult("Test with identical elements in each array", n, result);
    return 0;
}
```

```
;-------------------------------------------------
;               Ch03_08.asm
;-------------------------------------------------

; extern "C" long long CompareArrays_(const int* x, const int* y, long long n)
;
; Returns       -1          Value of 'n' is invalid
;               0 <= i < n  Index of first non-matching element
;               n           All elements match

        .code
CompareArrays_ proc frame

; Save non-volatile registers
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
        .endprolog

; Load arguments and validate 'n'
        mov rax,-1                          ;rax = return code for invalid n
        test r8,r8
        jle @F                              ;jump if n <= 0

; Compare the arrays for equality
        mov rsi,rcx                         ;rsi = x
        mov rdi,rdx                         ;rdi = y
        mov rcx,r8                          ;rcx = n
        mov rax,r8                          ;rax = n
        repe cmpsd
        je @F                               ;arrays are equal

; Calculate index of first non-match
        sub rax,rcx                         ;rax = index of mismatch + 1
        dec rax                             ;rax = index of mismatch
```

```
; Restore non-volatile registers and return
@@:     pop rdi
        pop rsi
        ret
CompareArrays_ endp
        end
```

The assembly language function `CompareArrays_` compares the elements of two integer arrays and returns the index of the first non-matching element. If the arrays are identical, the number of elements is returned. Otherwise, -1 is returned to indicate an error. Following the function prolog, a `test r8,r8` instruction checks argument value `n` to see if it's less than or equal to zero. As you learned in the previous section, this instruction performs a bitwise AND of the two operands and sets the status flags RFLAGS.PF, RFLAGS.SF, and RFLAGS.ZF based on the result (RFLAGS.CF and RFLAGS.OF are cleared). The result of the AND operation is discarded. If argument value `n` is invalid, the `jle @F` instruction skips over the compare code.

The actual compare code begins by loading register RSI with a pointer to `x` and RDI with pointer to `y`. The number of elements is then loaded into register RCX. The arrays are compared using a `repe cmpsd` (Compare String Doubleword) instruction. This instruction compares the two doublewords pointed to by RSI and RDI and sets the status flags according to the results. Registers RSI and RDI are incremented by four after each compare operation (the value 4 is used since that's the size of a doubleword in bytes). The `repe` (Repeat While Equal) prefix instructs the processor to repeat the `cmpsd` instruction as long as the condition `RCX != 0 && RFLAGS.ZF == 1` is true. Upon completion of the `cmpsd` instruction, a conditional jump is performed if the arrays are equal (RAX already contains the correct return value) or the index of the first non-matching elements is calculated. Here's the output for example `Ch03_08`.

```
Results for CompareArrays_ - array_size = 10000

Test using invalid array size
  expected = -1  actual = -1

Test using first element mismatch
  expected = 0  actual = 0

Test using middle element mismatch
  expected = 5000  actual = 5000

Test using last element mismatch
  expected = 9999  actual = 9999

Test with identical elements in each array
  expected = 10000  actual = 10000
```

## Array Reversal

The final source code example of this section demonstrates use of the `lods` (Load String) instruction to reverse the elements of an array. Unlike this section's previous source code examples, the processing loop of example `Ch03_09` traverses the source array starting from the last element and ending at the first element. Executing a reverse array traversal requires the direction flag (RFLAGS.DF) to be modified in a manner that is compatible with the Visual C++ runtime environment as elucidated in this example. Listing 3-9 shows the C++ and x86-64 assembly language source code for example `Ch03_09`.

***Listing 3-9.*** Example Ch03_09

```
//------------------------------------------------
//               Ch03_09.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>

using namespace std;

extern "C" int ReverseArray_(int* y, const int* x, int n);

void Init(int* x, int n)
{
    unsigned int seed = 17;
    uniform_int_distribution<> d {1, 1000};
    default_random_engine rng {seed};

    for (int i = 0; i < n; i++)
        x[i] = d(rng);
}

int main()
{
    const int n = 25;
    int x[n], y[n];

    Init(x, n);
    int rc = ReverseArray_(y, x, n);

    if (rc != 0)
    {
        cout << "\nResults for ReverseArray\n";

        const int w = 5;
        bool compare_error = false;

        for (int i = 0; i < n && !compare_error; i++)
        {
            cout << "  i: " << setw(w) << i;
            cout << "  y: " << setw(w) << y[i];
            cout << "  x: " << setw(w) << x[i] << '\n';

            if (x[i] != y[n - 1 - i])
                compare_error = true;
        }
```

```
        if (compare_error)
            cout << "ReverseArray compare error\n";
        else
            cout << "ReverseArray compare OK\n";
    }
    else
        cout << "ReverseArray_() failed\n";

    return 0;
}
```

```
;-------------------------------------------------
;               Ch03_09.asm
;-------------------------------------------------

; extern "C" int ReverseArray_(int* y, const int* x, int n);
;
; Returns       0 = invalid n, 1 = success

        .code
ReverseArray_ proc frame

; Save non-volatile registers
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
        .endprolog

; Make sure n is valid
        xor eax,eax                         ;error return code
        test r8d,r8d                        ;is n <= 0?
        jle InvalidArg                      ;jump if n <= 0

; Initialize registers for reversal operation
        mov rsi,rdx                         ;rsi = x
        mov rdi,rcx                         ;rdi = y
        mov ecx,r8d                         ;rcx = n
        lea rsi,[rsi+rcx*4-4]               ;rsi = &x[n - 1]

; Save caller's RFLAGS.DF, then set RFLAGS.DF to 1
        pushfq                              ;save caller's RFLAGS.DF
        std                                 ;RFLAGS.DF = 1

; Repeat loop until array reversal is complete
@@:     lodsd                               ;eax = *x--
        mov [rdi],eax                       ;*y = eax
        add rdi,4                           ;y++
        dec rcx                             ;n--
        jnz @B
```

```
; Restore caller's RFLAGS.DF and set return code
        popfq                               ;restore caller's RFLAGS.DF
        mov eax,1                           ;set success return code

; Restore non-volatile registers and return
InvalidArg:
        pop rdi
        pop rsi
        ret
ReverseArray_ endp
        end
```

The function ReverseArray_ copies the elements of a source array to a destination array in reverse order. This function requires three parameters: a pointer to a destination array named y, a pointer to a source array named x, and the number of elements n. Following validation of n, registers RSI and RDI are initialized with pointers to the arrays x and y. A mov ecx,r8d instruction loads the number of elements into register RCX. In order to reverse the elements of the source array, the address of the last array element x[n - 1] needs to be calculated. This is accomplished using a lea rsi,[rsi+rcx*4-4] instruction, which computes the effective address of the source memory operand (i.e., it performs the arithmetic operation specified between the brackets and saves the result to register RSI).

The Visual C++ runtime environment assumes that the direction flag (RFLAGS.DF) is always cleared. If an assembly language function sets RFLAGS.DF to perform auto-decrementing with a string instruction, the flag must be cleared before returning to the caller or using any library functions. The function ReverseArray_ partially fulfills this requirement by saving the current state of RFLAGS.DF on the stack using the pushfq (Push RFLAGS Register onto Stack) instruction. It then uses the std (Set Direction Flag) instruction to set RFLAGS.DF to 1. The duplication of array elements from x to y is straightforward. A lodsd (Load String Doubleword) instruction loads an element from x into EAX and subtracts four from register RSI. The next instruction, mov [rdi],eax, saves this value to the element in y that is pointed to by RDI. An add rdi,4 instruction points EDI to the next element in y. Register RCX is then decremented and the loop is repeated until the array reversal is complete.

Following the reverse array loop, a popfq (Pop Stack into RFLAGS Register) instruction restores the original state of RFLAGS.DF. One question that might be asked at this point is if the Visual C++ runtime environment assumes that RFLAGS.DF is always cleared, why doesn't the function ReverseArray_ use a cld (Clear Direction Flag) instruction to restore RFLAGS.DF instead of a pushfq/popfq sequence? Yes, the Visual C++ runtime environment assumes that RFLAGS.DF is always cleared, but it cannot enforce this policy during program execution. If ReverseArray_ were to be included in a DLL, it could conceivably be called by a function written in a language that uses a different default state for the direction flag. Using pushfq and popfq ensures that the state of the caller's direction is always properly restored. Here is the output example Ch03_09.

```
Results for ReverseArray
  i:     0  y:   583  x:   560
  i:     1  y:   904  x:   586
  i:     2  y:   924  x:   752
  i:     3  y:   635  x:   743
  i:     4  y:   347  x:   511
  i:     5  y:   313  x:   370
  i:     6  y:   738  x:   809
  i:     7  y:   810  x:   214
  i:     8  y:   935  x:   823
  i:     9  y:   354  x:   456
  i:    10  y:   592  x:    13
  i:    11  y:   613  x:   240
```

```
  i:    12  y:   413  x:   413
  i:    13  y:   240  x:   613
  i:    14  y:    13  x:   592
  i:    15  y:   456  x:   354
  i:    16  y:   823  x:   935
  i:    17  y:   214  x:   810
  i:    18  y:   809  x:   738
  i:    19  y:   370  x:   313
  i:    20  y:   511  x:   347
  i:    21  y:   743  x:   635
  i:    22  y:   752  x:   924
  i:    23  y:   586  x:   904
  i:    24  y:   560  x:   583
ReverseArray compare OK
```

# Summary

Here are the key learning points for Chapter 3:

- The address of an element in a one-dimensional array can be calculated using the base address (i.e., the address of the first element) of the array, the index of the element, and the size in bytes of each element. The address of an element in a two-dimensional array can be calculated using the base address of the array, the row and column indices, the number of columns, and the size in bytes of each element.
- The Visual C++ calling convention designates each general-purpose register as volatile or non-volatile. A function must preserve the contents of any non-volatile general-purpose register it uses. A function should use the `push` instruction in its prolog to save the contents of a non-volatile register on the stack. A function should use the `pop` instruction in its epilog to restore the contents of any previously-saved non-volatile register.
- X86-64 assembly language code can define and use structures similar to the way they are used in C++. An assembly language structure may require extra padding elements to ensure that it's semantically equivalent to a C++ structure.
- The upper 32 bits of a 64-bit general-purpose register are set to zero in instructions that specify the corresponding 32-bit register as a destination operand. The upper 56 or 48 bits of a 64-bit general-purpose register are not affected when the destination operand of an instruction is an 8-bit or 16-bit register.
- The x86 string instructions `cmps`, `lods`, `movs`, `scas`, and `stos` can be used to compare, load, copy, scan, or initialize text strings. They also can be used to perform operations on arrays and other similarly-ordered data structures.
- The prefixes `rep`, `repe`, `repz`, `repne`, and `repnz` can be used with a string instruction to repeat a string operation multiple times (RCX contains the count value) or until the specified zero flag (RFLAGS.ZF) condition occurs.
- The state of the direction flag (RFLAGS.DF) must be preserved across function boundaries.
- The `test` instruction is often used as an alternative to the `cmp` instruction, especially when testing a value to ascertain if it's less than, equal to, or greater than zero.
- The `lea` instruction can be used to simplify effective address calculations.

# CHAPTER 4

# Advanced Vector Extensions

In the first three chapters of this book, you learned about the core x86-64 platform including its data types, general-purpose registers, and memory addressing modes. You also examined a cornucopia of sample code that illustrated the fundamentals of x86-64 assembly language programming, including basic operands, integer arithmetic, compare operations, conditional jumps, and manipulation of common data structures.

This chapter introduces Advanced Vector Extensions (AVX). It begins with a brief overview of AVX technologies and SIMD (Single Instruction Multiple Data) processing concepts. This is followed by an examination of the AVX execution environment that covers register sets, data types, and instruction syntax. The chapter also includes discussions of AVX's scalar floating-point capabilities and its SIMD computational resources. The material presented in this chapter is relevant not only to AVX but also provides the necessary background information to understand AVX2 and AVX-512, which are explained in later chapters.

In the discussions that follow in this and subsequent chapters, the term x86-AVX is used to describe general characteristics and computing resources of Advanced Vector Extensions. The acronyms AVX, AXV2, and AVX-512 are employed when examining attributes or instructions related to a specific x86 feature set enhancement.

## AVX Overview

AMD and Intel first incorporated AVX into their CPUs starting in 2011. AVX extends the packed single-precision and double-precision floating-point capabilities of x86-SSE from 128 bits to 256 bits. Unlike general-purpose register instructions, AVX instructions use a three-operand syntax that employs non-destructive source operands, which simplifies assembly language programming considerably. Programmers can use this new instruction syntax with packed 128-bit integer, packed 128-bit floating-point, and packed 256-bit floating-point operands. The three-operand instruction syntax can also be exploited to perform scalar single-precision and double-precision floating-point arithmetic.

In 2013 Intel launched processors that included AVX2. This architectural enhancement extends the packed integer capabilities of AVX from 128 bits to 256 bits. AVX2 adds new data broadcast, blend, and permute instructions to the x86 platform. It also introduces a new vector-index addressing mode that facilitates memory loads (or gathers) of data elements from non-contiguous locations. The most recent x86-AVX extension is called AVX-512, which expands the SIMD capabilities of AVX and AVX2 from 256 bits to 512 bits. AVX-512 also adds eight new opmask registers named K0–K7 to the x86 platform. These registers facilitate conditional instruction execution and data merging operations using per-element granularity. Table 4-1 summarizes current x86-AVX technologies. In this table (and subsequent tables), the acronyms SPFP and DPFP are used to signify single-precision floating-point and double-precision floating-point, respectively.

© Daniel Kusswurm 2018

D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_4

***Table 4-1.** Summary of x86-AVX Technologies*

| Feature | AVX | AVX2 | AVX-512 |
|---|---|---|---|
| Three-operand syntax; non-destructive source operands | Yes | Yes | Yes |
| SIMD operations using 128-bit packed integers | Yes | Yes | Yes |
| SIMD operations using 256-bit packed integers | No | Yes | Yes |
| SIMD operations using 512-bit packed integers | No | No | Yes |
| SIMD operations using 128-bit packed SPFP, DPFP | Yes | Yes | Yes |
| SIMD operations using 256-bit packed SPFP, DPFP | Yes | Yes | Yes |
| SIMD operations using 512-bit packed SPFP, DPFP | No | No | Yes |
| Scalar SPFP, DPFP arithmetic | Yes | Yes | Yes |
| Enhanced SPFP, DPFP compare operations | Yes | Yes | Yes |
| Basic SPFP, DPFP broadcast and permute | Yes | Yes | Yes |
| Enhanced SPFP, DPFP broadcast and permute | No | Yes | Yes |
| Packed integer broadcast | No | Yes | Yes |
| Enhanced packed integer broadcast, compare, permute, conversions | No | No | Yes |
| Instruction-level broadcast and rounding control | No | No | Yes |
| Fused-multiply-add | No | Yes | Yes |
| Data gather | No | Yes | Yes |
| Data scatter | No | No | Yes |
| Conditional execution and data merging using opmask registers | No | No | Yes |

It should be noted that fuse-multiply-add is a distinct x86 platform feature extension that was introduced in tandem with AVX2. A program must confirm the presence of this feature extension by testing the CPUID FMA feature flag before using any of the corresponding instructions. You'll learn how to do this in Chapter 16. The remainder of this chapter focuses primarily on AVX. Chapters 8 and 12 discuss the particulars of AVX2 and AVX-512 in greater detail.

## SIMD Programming Concepts

As implied by the words of the acronym, a SIMD computing element executes the same operation on multiple data items simultaneously. Universal SIMD operations include basic arithmetic such as addition, subtraction, multiplication, and division. SIMD processing techniques can also be applied to a variety of other computational tasks including data compares, conversions, Boolean calculations, permutations, and bit shifts. Processors facilitate SIMD operations by reinterpreting the bits of an operand in a register or memory location. For example, a 128-bit wide operand can hold two independent 64-bit integer values. It is also capable of accommodating four 32-bit integers, eight 16-bit integers, or sixteen 8-bit integers, as illustrated in Figure 4-1.

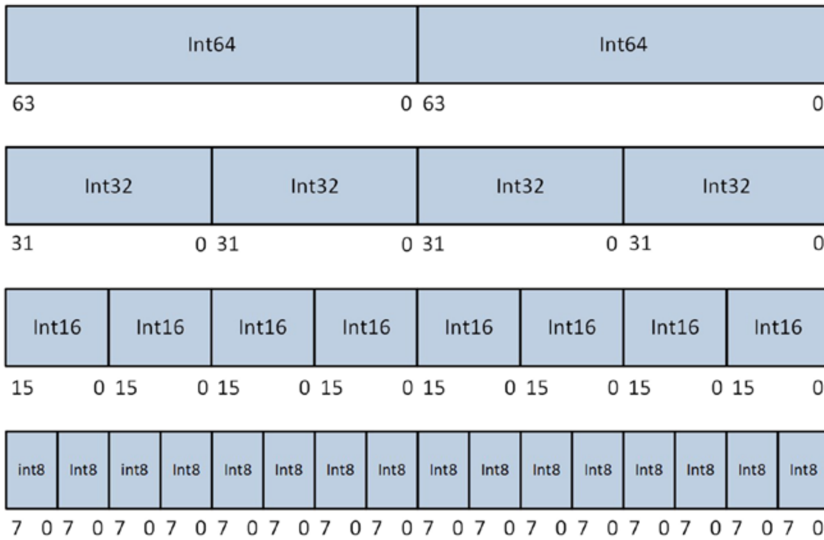



***Figure 4-1.*** *128-bit wide operand using distinct integers*

Figure 4-2 exemplifies a few SIMD arithmetic operations greater detail. In this figure, integer addition is illustrated using two 64-bit integers, four 32-bit integers, or eight 16-bit integers. Faster algorithmic processing takes place when multiple data items are exercised, since the CPU can carry out the necessary operations in parallel. For example, when 16-bit integer operands are specified by an instruction, the CPU performs all eight 16-bit integer additions simultaneously.

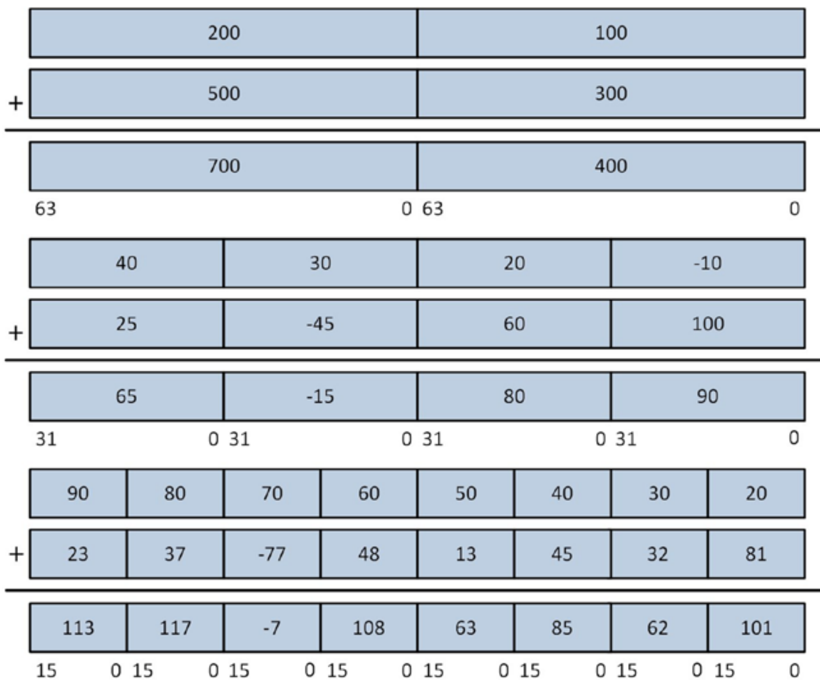



***Figure 4-2.*** *SIMD integer addition*

# Wraparound vs. Saturated Arithmetic

One extremely useful feature of x86-AVX technology is its support for saturated integer arithmetic. In saturated integer arithmetic, computational results are automatically clipped by the processor to prevent overflow and underflow conditions. This differs from normal wraparound integer arithmetic where an overflow or underflow result is retained (as you'll soon see). Saturated arithmetic is handy when working with pixel values since it automatically clips values and eliminates the need to explicitly check the result of each pixel calculation for an overflow or underflow condition. X86-AVX includes instructions that perform saturated arithmetic using 8-bit and 16-bit integers, both signed and unsigned.

Let's take a closer look at some examples of both wraparound and saturated arithmetic. Figure 4-3 shows an example of 16-bit signed integer addition using wraparound and saturated arithmetic. An overflow condition occurs if the two 16-bit signed integers are added using wraparound arithmetic. With saturated arithmetic, however, the result is clipped to the largest possible 16-bit signed integer value. Figure 4-4 illustrates a similar example using 8-bit unsigned integers. Besides addition, x86-AVX also supports saturated integer subtraction, as shown in Figure 4-5. Table 4-2 summarizes the saturated arithmetic range limits for all supported integer sizes and sign types.

16-bit Signed Integer Addition

| | Wraparound | Saturated |
|---|---|---|
| | 20000 (0x4e20) | 20000 (0x4e20) |
| + | 15000 (0x3a98) | 15000 (0x3a98) |
| | -30536 (0x88b8) | 32767 (0x7fff) |

***Figure 4-3.*** *16-bit signed integer addition using wraparound and saturated arithmetic*

8-bit Unsigned Integer Addition

| | Wraparound | Saturated |
|---|---|---|
| | 150 (0x96) | 150 (0x96) |
| + | 135 (0x87) | 135 (0x87) |
| | 29 (0x1d) | 255 (0xff) |

***Figure 4-4.*** *8-bit unsigned integer addition using wraparound and saturated arithmetic*

***Figure 4-5.*** *16-bit signed integer subtraction using wraparound and saturated arithmetic*

***Table 4-2.*** *Range Limits for Saturated Arithmetic*

| Integer Type | Lower Limit | Upper Limit |
| --- | --- | --- |
| 8-bit signed | -128 (0x80) | +127 (0x7f) |
| 8-bit unsigned | 0 | +255 (0xff) |
| 16-bit signed | -32768 (0x8000) | +32767 (0x7fff) |
| 16-bit unsigned | 0 | +65535 (0xffff) |

# AVX Execution Environment

In this section you'll learn about the x86-AVX execution environment. Included are explanations of the AVX register set, its data types, and instruction syntax. As mentioned earlier, x86-AVX is an architectural enhancement that extends x86-SSE technology to support SIMD operations using either 256-bit or 128-bit wide operands. The material that's presented in this section assumes no previous knowledge or experience with x86-SSE.

## Register Set

X86-64 processors that support AVX incorporate 16 256-wide registers named YMM0 – YMM15. The low-order 128 bits of each YMM register are aliased to a corresponding XMM register, as illustrated in Figure 4-6. Most AVX instructions can use any of the XMM or YMM registers as SIMD operands. The XMM registers can also be employed to carry out scalar floating-point calculations using either single-precision or double-precision values similar to x86-SSE. Programmers with assembly language experience using x86-SSE need to be aware of some minor execution differences between this earlier instruction set extension and x86-AVX. These differences are explained later in this chapter.

Bit Position

| 255 — 128 | 127 — 0 |
| --- | --- |
| YMM0 | XMM0 |
| YMM1 | XMM1 |
| YMM2 | XMM2 |
| YMM3 | XMM3 |
| YMM4 | XMM4 |
| YMM5 | XMM5 |
| YMM6 | XMM6 |
| YMM7 | XMM7 |
| YMM8 | XMM8 |
| YMM9 | XMM9 |
| YMM10 | XMM10 |
| YMM11 | XMM11 |
| YMM12 | XMM12 |
| YMM13 | XMM13 |
| YMM14 | XMM14 |
| YMM15 | XMM15 |

***Figure 4-6.*** *AVX register set*

The x86-AVX execution environment also includes a control-status register named MXCSR. This register contains status flags that facilitate the detection of error conditions caused by floating-point arithmetic operations. It also includes control bits that programs can use to enable or disable floating-point exceptions and specify rounding options. You'll learn more about MXCSR register later in this chapter.

## Data Types

As previously mentioned, AVX supports SIMD operations using 256-bit and 128-bit wide packed single-precision or packed double-precision floating-point operands. A 256-bit wide YMM register or memory location can hold eight single-precision or four double-precision values, as shown in Figure 4-7. When used with a 128-bit wide XMM register or memory location, an AVX instruction can process four single-precision or two double-precision values. Like SSE and SSE2, AVX instructions use the low-order doubleword or quadword of an XMM register to carry out scalar single-precision or double-precision floating-point arithmetic.

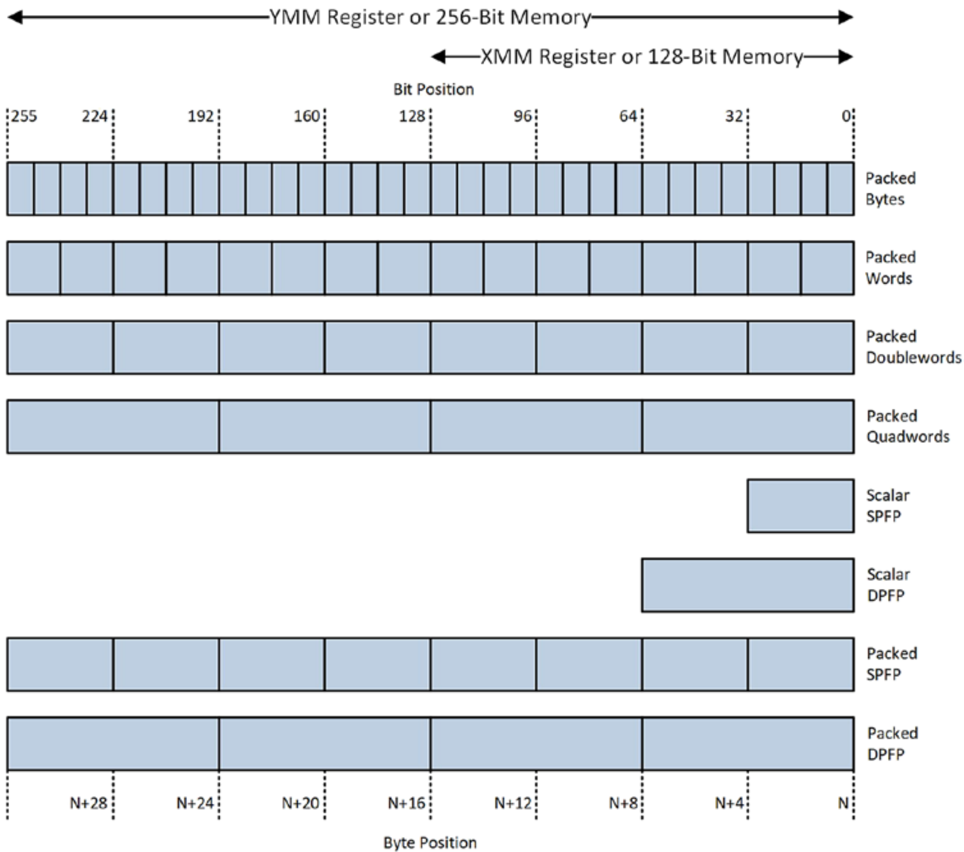



***Figure 4-7.*** *AVX and AVX2 data types*

AVX also includes instructions that use the XMM registers to perform SIMD operations using a variety of packed integer operands including bytes, words, doublewords, and quadwords. AVX2 extends the packed integer processing capabilities of AVX to the YMM registers and 256-bit wide operands in memory. Figure 4-7 also shows these data types.

## Instruction Syntax

Perhaps the most noteworthy programming facet of x86-AVX is its use of a contemporary assembly language instruction syntax. Most x86-AVX instructions use a three-operand format that consists of two source operands and one destination operand. The general syntax that's employed for x86-AVX instructions is `InstrMnemonic DesOp,SrcOp1,SrcOp2`. Here, `InstrMnemonic` signifies the instruction mnemonic, `DesOp` represents the destination operand, and `SrcOp1` and `SrcOp2` denote the source operands. A small subset of x86-AVX instructions employ one or three source operands along with a destination operand. Nearly all x86-AVX instruction source operands are non-destructive. This means source operands are not modified during instruction execution, except in cases where the destination operand register is the same as one of the source operand registers. The use of non-destructive source operands often results in simpler and slightly faster code since the number of register-to-register data transfers that a function must perform is reduced.

X86-AVX's ability to support a three-operand instruction syntax is due to a new instruction-encoding prefix. The vector extension (VEX) prefix enables x86-AVX instructions to be encoded using a more efficient format than the prefixes used for x86-SSE instructions. The VEX prefix has also been used to add new general-purpose register instructions to the x86 platform. You'll learn about these instructions in Chapter 8.

# AVX Scalar Floating-Point

This section examines the scalar floating-point capabilities of AVX. It begins with a short explanation of some important floating-point concepts including data types, bit encodings, and special values. Software developers who understand these concepts are often able to improve the performance of algorithms that make heavy use of floating-point arithmetic and minimize potential floating-point errors. The AVX scalar floating-point registers are also explained in this section and this includes descriptions the XMM registers and the MXCSR control-status register. The section concludes with an overview of the AVX scalar floating-point instruction set.

## Floating-Point Programming Concepts

In mathematics a real-number system depicts an infinite continuum of all possible positive and negative numbers including integers, rational numbers, and irrational numbers. Given their finite resources, modern computing architectures typically employ a floating-point system to approximate a real-number system. Like many other computing platforms, the x86's floating-point system is based on the IEEE 754 standard for binary floating-point arithmetic. This standard includes specifications that define bit encodings, range limits, and precisions for scalar floating-point values. The IEEE 754 standard also specifies important details related to floating-point arithmetic operations, rounding rules, and numerical exceptions.

The AVX instruction set supports common floating-point operations using single precision (32-bit) and double precision (64-bit) values. Many C++ compilers including Visual C++ use the x86's intrinsic single-precision and double-precision types to implement the C++ types `float` and `double`. Figure 4-8 illustrates the memory organization of both single-precision and double-precision floating-point values. This figure also includes common integer types for comparison purposes.

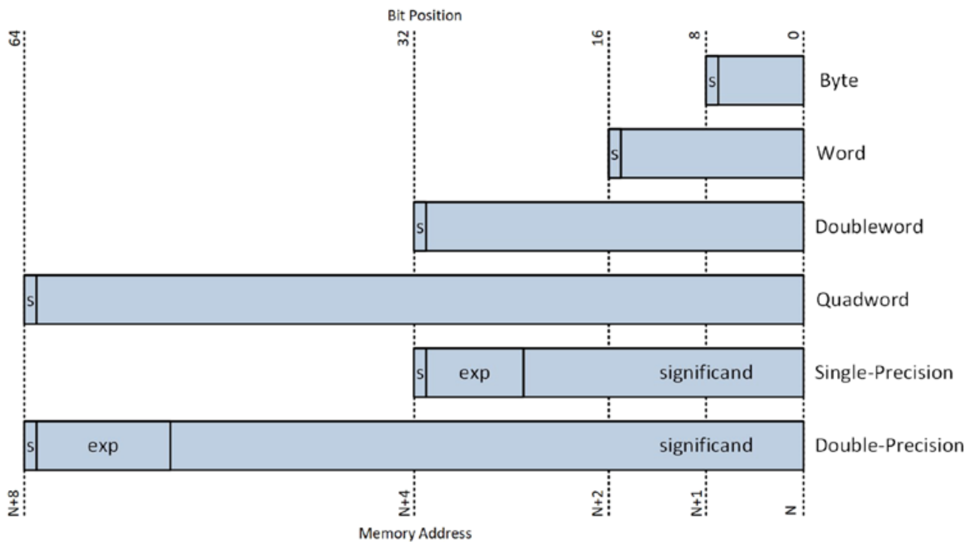



***Figure 4-8.*** *Memory organization of floating-point values*

The binary encoding of a floating-point value requires three distinct fields: a significand, an exponent, and a sign bit. The significand field represents a number's significant digits (or fractional part). The exponent specifies the location of the binary "decimal" point in the significand, which determines the magnitude. The sign bit indicates whether the number is positive (s = 0) or negative (s = 1). Table 4-3 lists the various size parameters that are used to encode single-precision and double-precision floating-point values.

***Table 4-3.*** *Floating-Point Size Parameters*

| Parameter | Single-Precision | Double-Precision |
| --- | --- | --- |
| Total width | 32 bits | 64 bits |
| Significand width | 23 bits | 52 bits |
| Exponent width | 8 bits | 11 bits |
| Sign width | 1 bit | 1 bit |
| Exponent bias | +127 | +1023 |

Figure 4-9 illustrates how to convert a decimal number into an x86 compatible floating-point encoded value. In this example, the number 237.8125 is transformed from a decimal number to its single-precision floating-point encoding. The process starts by converting the number from base 10 to base 2. Next, the base 2 value is transformed to a binary scientific value. The value to the right of the $E_2$ symbol is the binary exponent. A properly encoded floating-point value uses a biased exponent instead of the true exponent since this expedites floating-point compare operations. For a single-precision floating-point number, the bias value is +127. Adding the exponent bias value to the true exponent generates a binary scientific number with a biased exponent value. In the example that's shown in Figure 4-9, adding 111b (+7) to 1111111b (+127) yields a binary scientific with a biased exponent value of 10000110b (+134).

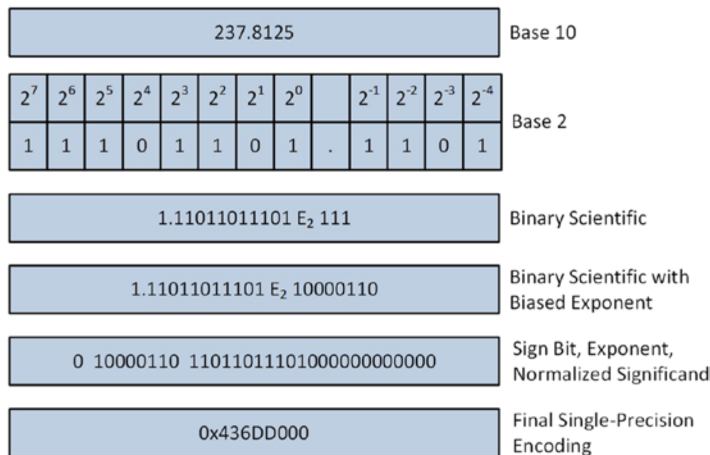



***Figure 4-9.*** *Single-precision floating-point encoding process*

When encoding a single-precision or double-precision floating-point value, the leading 1 digit of the significand is implied and not included in the final binary representation. Dropping the leading 1 digit forms a normalized significand. The three fields required for an IEEE 754 complaint encoding are now available, as shown in Table 4-4. A reading of the bit fields in this table from left to right yields the 32-bit value 0x436DD000, which is the final single-precision floating-point encoding of 237.8125.

***Table 4-4.*** *Bit Fields for IEEE 754 Compliant Encoding of 237.8125*

| Sign | Biased Exponent | Normalized Significand |
|---|---|---|
| 1 | 10000110 | 11011011101000000000000 |

The IEEE 754 floating-point encoding scheme reserves a small set of bit patterns for special values that are used to handle certain processing conditions. The first group of special values includes denormalized numbers (or denormal). As shown in the earlier encoding example, the standard encoding of a floating-point number assumes that the leading digit of the significand is always a 1. One limitation of IEEE 754 floating-point encoding scheme is its inability to accurately represent numbers very close to zero. In these cases, values get encoded using a non-normalized format, which enables tiny numbers close to zero (both positive and negative) to be encoded using less precision. Denormals rarely occur but when they do, the CPU can still process them. In algorithms where the use of a denormal is problematic, a function can test a floating-point value in order to ascertain its denormal state or the processor can be configured to generate an underflow or denormal exception.

Another application of special values involves the encodings that are used for floating-point zero. The IEEE 754 standard supports two different representations of floating-point zero: positive zero (+0.0) and negative zero (–0.0). A negative zero can be generated either algorithmically or as a side effect of the floating-point rounding mode. Computationally, the processor treats positive and negative zero the same and the programmer typically does not need to be concerned.

The IEEE 754 encoding scheme also supports positive and negative representations of infinity. Infinities are produced by certain numerical algorithms, overflow conditions, or division by zero. As discussed later in this chapter, the processor can be configured to generate an exception whenever a floating-point overflow occurs or if a program attempts to divide a number by zero.

The final special value type is called Not a Number (NaN). NaNs are floating-point encodings that represent invalid numbers. The IEEE 754 standard defines two types of NaNs: signaling NaN (SNaN) and quiet NaN (QNaN). SNaNs are created by software; an x86-64 CPU will not create a SNaN during any arithmetic operation. Any attempt by an instruction to use a SNaN will cause an invalid operation exception, unless the exception is masked. SNaNs are useful for testing exception handlers. They can also be exploited by an application program for proprietary numerical-processing purposes. An x86 CPU uses QNaNs as a default response to certain invalid arithmetic operations whose exceptions are masked. For example, one unique encoding of a QNaN, called an indefinite, is substituted for a result whenever a function uses one of the scalar square root instructions with a negative value. QNaNs also can be used by programs to signify algorithm-specific errors or other unusual numerical conditions. When QNaNs are used as operands, they enable continued processing without generating an exception.

When developing software that performs floating-point calculations, it is important to keep in mind that the employed encoding scheme is simply an approximation of a real-number system. It is impossible for any floating-point encoding system to represent an infinite number of values using a finite number of bits. This leads to floating-point rounding errors that can affect the accuracy of a calculation. Also, some mathematical properties that hold true for integers and real numbers are not necessarily true for floating-point numbers. For example, floating-point multiplication is not necessarily associative; `(a * b) * c` may not equal `a * (b * c)` depending on the values of `a`, `b`, and `c`. Developers of algorithms that require high levels of floating-point accuracy must be aware of these issues. Appendix A contains a list of references that explain this and other potential pitfalls of floating-point arithmetic in greater detail. Chapter 9 also includes a source code example that exemplifies floating-point non-associativity.

## Scalar Floating-Point Register Set

As previously shown in Figure 4-6, all x86-64 compatible processors include 16 128-bit registers named XMM0 – XMM15. A program can use any of the XMM registers to perform scalar floating-point operations including common arithmetic calculations, data transfers, comparisons, and type conversions. The CPU uses the low-order 32 bits of an XMM register to carry out single-precision floating-point calculations. Double-precision floating-point operations employ the low-order 64 bits. Figure 4-10 illustrates these register locations in greater detail. Programs cannot use the high-order bits of an XMM register to perform scalar floating-point calculations. However, when used as a destination operand, the values of these bits might be modified during the execution of an AVX scalar floating-point instruction as explained later in this section.

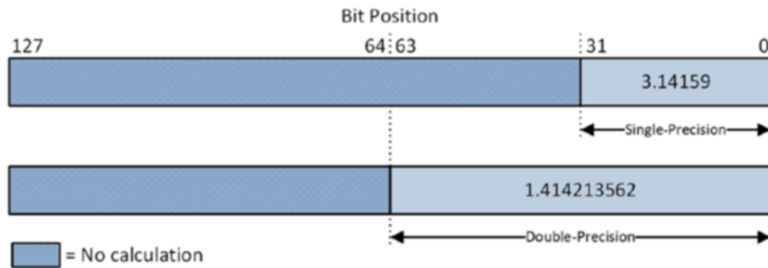



***Figure 4-10.*** *Scalar floating-point values when loaded in an XMM register*

## Control-Status Register

In addition to the XMM registers, x86-64 processors include a 32-bit control-status register named MXCSR. This register contains a series of control flags that enable a program to specify options for floating-point calculations and exceptions. It also includes a set of status flags that can be tested to detect floating-point error conditions. Figure 4-11 shows the organization of the bits in MXCSR; Table 4-5 describes the purpose of each bit field.

***Figure 4-11.*** *MXCSR control and status register*

***Table 4-5.*** *Description of MXCSR Register Bit Fields*

| Bit | Field Name | Description |
|---|---|---|
| IE | Invalid operation flag | Floating-point invalid operation error flag. |
| DE | Denormal flag | Floating-point denormal error flag. |
| ZE | Divide-by-zero flag | Floating-point division-by-zero error flag. |
| OE | Overflow flag | Floating-point overflow error flag. |
| UE | Underflow flag | Floating-point underflow error flag. |
| PE | Precision flag | Floating-point precision error flag. |
| DAZ | Denormals are zero | When set to 1, forcibly converts a denormal source operand to zero prior to its use in a calculation. |
| IM | Invalid operation mask | Floating-point invalid operation error exception mask. |
| DM | Denormal mask | Floating-point denormal error exception mask. |
| ZM | Divide-by-zero mask | Floating-point divide-by-zero error exception mask. |
| OM | Overflow mask | Floating-point overflow error exception mask. |
| UM | Underflow mask | Floating-point underflow error exception mask. |
| PM | Precision mask | Floating-point precision error exception mask. |
| RC | Rounding control | Specifies the method for rounding floating-point results. Valid options include round to nearest (00b), round down toward +∞ (01b), round up toward +∞ (10b), and round toward zero or truncate (11b). |
| FTZ | Flush to zero | When set to 1, forces a zero result if the underflow exception is masked and a floating-point underflow error occurs. |

An application program can modify any of the MXCSR's control flags or status bits to accommodate its specific SIMD floating-point processing requirements. Any attempt to write a non-zero value to a reserved bit position will cause the processor to generate an exception. The processor sets an MXCSR error flag to 1 following the occurrence of an error condition. MXCSR error flags are not automatically cleared by the processor after an error is detected; they must be manually reset. The control flags and status bits of the MXCSR register can be modified using the `vldmxcsr` (Load MXCSR Register) instruction. Setting a mask bit to 1 disables the corresponding exception. The `vstmxcsr` (Store MXCSR Register) instruction can be used to save the current MXCSR state. An application program cannot directly access the internal processor tables that specify floating-point exception handlers. However, most C++ compilers provide a library function that allows an application program to designate a callback function that gets invoked whenever a floating-point exception occurs.

The MXCSR includes two control flags that can be used to speed up certain floating-point calculations. Setting the MXCSR.DAZ control flag to 1 can improve the performance of algorithms where the rounding of a denormal value to zero is acceptable. Similarly, the MXCSR.FTZ control flag can be used to accelerate computations where floating-point underflows are common. The downside of enabling either of these options is non-compliance with the IEEE 754 floating-point standard.

## Instruction Set Overview

Table 4-6 lists in alphabetical order commonly used AVX scalar floating-point instructions. In this table, mnemonic text `[d|s]` signifies that an instruction can be used with either double-precision floating-point or single-precision floating-point operands. You'll learn how to use many of these instructions in Chapter 5.

***Table 4-6.*** *Overview of Commonly-Used AVX Scalar Floating-Point Instructions*

| Mnemonic | Description |
|---|---|
| `vadds[d|s]` | Scalar floating-point addition |
| `vbroadcasts[d|s]` | Broadcast scalar floating-point value |
| `vcmps[d|s]` | Scalar floating-point compare |
| `vcomis[d|s]` | Ordered scalar floating-point compare and set RFLAGS |
| `vcvts[d|s]2si` | Convert scalar floating-point to doubleword signed integer |
| `vcvtsd2ss` | Convert scalar DPFP to scalar SPFP |
| `vcvtsi2s[d|s]` | Convert signed doubleword integer to scalar floating-point |
| `vcvtss2sd` | Convert scalar SPFP to DPFP |
| `vcvtts[d|s]2si` | Convert with truncation scalar floating-point to signed integer |
| `vdivs[d|s]` | Scalar floating-point division |
| `vmaxs[d|s]` | Scalar floating-point maximum |
| `vmins[d|s]` | Scalar floating-point minimum |
| `vmovs[d|s]` | Move scalar floating-point value |
| `vmuls[d|s]` | Scalar floating-point multiplication |
| `vrounds[d|s]` | Round scalar floating-point value |
| `vsqrts[d|s]` | Scalar floating-point square root |
| `vsubs[d|s]` | Scalar floating-point subtraction |
| `vucomis[d|s]` | Unordered scalar floating-point compare and set RFLAGS |

Table 4-7 illustrates operation of the AVX scalar floating-point instructions `vadds[d|s]` and `vsqrts[d|s]`. In these examples, the colon notation designates bit position ranges within a register (e.g., 31:0 designates bits positions 31 through 0 inclusive). Note that execution of an AVX scalar floating-point instruction also copies the unused bits of the first source operand to the destination operand. Also note that the upper 128 bits of the corresponding YMM register are set to zero.

***Table 4-7.*** *AVX Scalar Floating-Point Instruction Examples*

| Instruction | Operation |
|---|---|
| `vaddss xmm0,xmm1,xmm2` | `xmm0[31:0] = xmm1[31:0] + xmm2[31:0]`<br>`xmm0[127:32] = xmm1[127:32]`<br>`ymm0[255:128] = 0` |
| `vaddsd xmm0,xmm1,xmm2` | `xmm0[63:0] = xmm1[63:0] + xmm2[63:0]`<br>`xmm0[127:64] = xmm1[127:64]`<br>`ymm0[255:128] = 0` |
| `vsqrtss xmm0,xmm1,xmm2` | `xmm0[31:0] = sqrt(xmm2[31:0])`<br>`xmm0[127:32] = xmm1[127:32]`<br>`ymm0[255:128] = 0` |
| `vsqrtsd xmm0,xmm1,xmm2` | `xmm0[63:0] = sqrt(xmm2[63:0])`<br>`xmm0[127:64] = xmm1[127:64]`<br>`ymm0[255:128] = 0` |

# AVX Packed Floating-Point

AVX supports packed floating-point operations using either 128-bit wide or 256-bit wide operands. Figures 4-12 and 4-13 illustrate common packed floating-point arithmetic operations using 256-bit wide operands with single-precision and double-precision elements. Similar to AVX scalar floating-point, rounding for AVX packed floating-point arithmetic operations is specified by the MXCSR's rounding control field, as defined in Table 4-5. The processor also uses the MXCSR's status flags to signal the occurrence of a packed floating-point error condition.

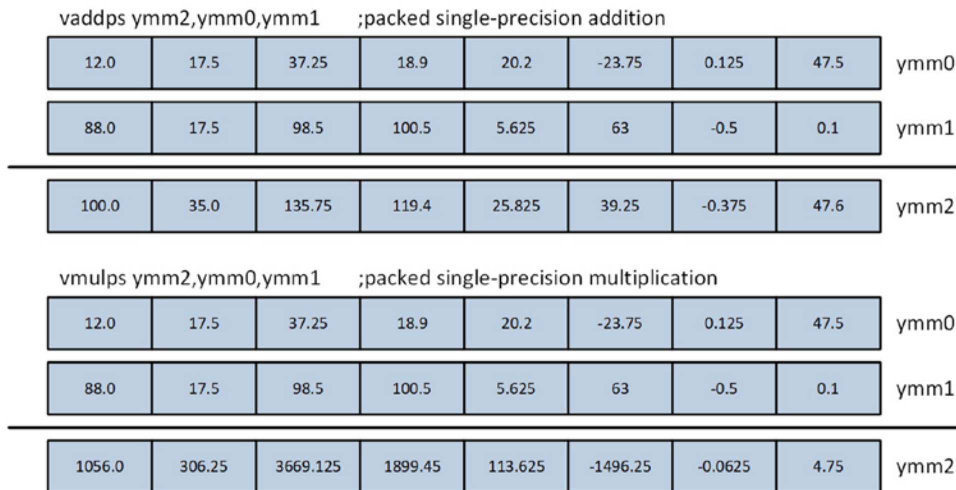



***Figure 4-12.*** *AVX packed single-precision floating-point addition*

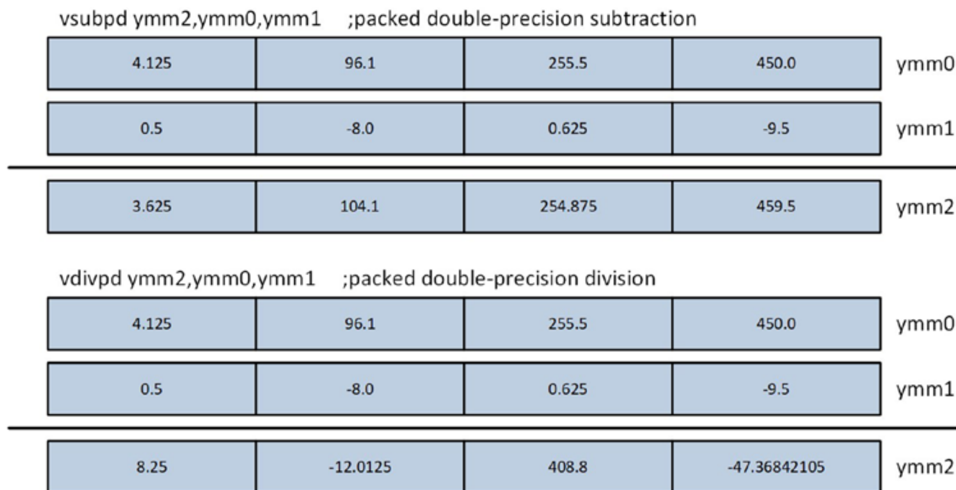



***Figure 4-13.*** *AVX packed double-precision floating-point multiplication*

Most AVX arithmetic instructions perform their operations using the corresponding element positions of the two source operands. AVX also supports horizontal arithmetic operations using either packed floating-point or packed integer operands. A horizontal arithmetic operation carries out its computations using the adjacent elements of a packed data type. Figure 4-14 illustrates horizontal addition using single-precision floating-point and horizontal subtraction using double-precision floating-point operands. The AVX instruction set also supports integer horizontal addition and subtraction using packed words and doublewords. Horizontal operations are typically used to reduce a packed data operand that contains multiple intermediate values to a single final result.

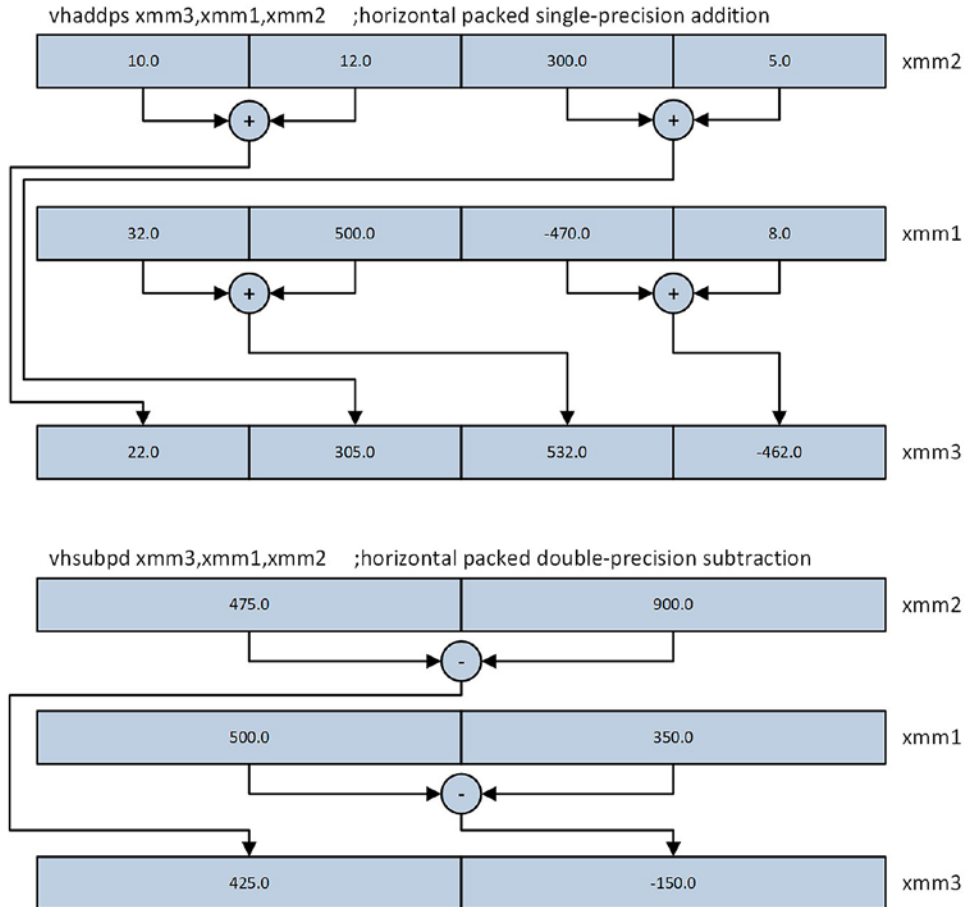



***Figure 4-14.*** *AVX horizontal addition and subtraction using single-precision and double-precision elements*

## Instruction Set Overview

Table 4-8 lists in alphabetical order commonly used AVX packed floating-point instructions. Similar to the scalar floating-point table that you saw in the previous section, the mnemonic text `[d|s]` signifies that an instruction can be used with either packed double-precision floating-point or packed single-precision floating-point operands. You'll learn how to use many of these instructions in Chapter 6.

*Table 4-8. Overview of Commonly-Used AVX Packed Floating-Point Instructions*

| Instruction | Description |
|---|---|
| vaddp[d\|s] | Packed floating-point addition |
| vaddsubp[d\|s] | Packed floating-point add-subtract |
| vandp[d\|s] | Packed floating-point bitwise AND |
| vandnp[d\|s] | Packed floating-point bitwise AND NOT |
| vblendp[d\|s] | Packed floating-point blend |
| vblendvp[d\|s] | Variable packed floating-point blend |
| vcmpp[d\|s] | Packed floating-point compare |
| vcvtdq2p[d\|s] | Convert packed signed doubleword integers to floating-point |
| vcvtp[d\|s]2dq | Convert packed floating-point to signed doublewords |
| vcvtpd2ps | Convert packed DPFP to packed SPFP |
| vcvtps2pd | Convert packed SPFP to packed DPFP |
| vdivp[d\|s] | Packed floating-point division |
| vdpp[d\|s] | Packed dot product |
| vhaddp[d\|s] | Horizontal packed floating-point addition |
| vhsubp[d\|s] | Horizontal packed floating-point subtraction |
| vmaskmovp[d\|s] | Packed floating-point conditional load and store |
| vmaxp[d\|s] | Packed floating-point maximum |
| vminp[d\|s] | Packed floating-point minimum |
| vmovap[d\|s] | Move aligned packed floating-point values |
| vmovmskp[d\|s] | Extract packed floating-point sign bitmask |
| vmovup[d\|s] | Move unaligned packed floating-point values |
| vmulp[d\|s] | Packed floating-point multiplication |
| vorp[d\|s] | Packed floating-point bitwise inclusive OR |
| vpermilp[d\|s] | Permute in-lane packed floating-point elements |
| vroundp[d\|s] | Round packed floating-point values |
| vshufp[d\|s] | Shuffle packed floating-point values |
| vsqrtp[d\|s] | Packed floating-point square root |
| vsubp[d\|s] | Packed floating-point subtraction |
| vunpckhp[d\|s] | Unpack and interleave high packed floating-point values |
| vunpcklp[d\|s] | Unpack and interleave low packed floating-point values |
| vxorp[d\|s] | Packed floating-point bitwise exclusive OR |

## AVX Packed Integer

AVX supports packed integer operations using 128-bit wide operands. A 128-bit wide operand facilitates packed integer operations using two quadwords, four doublewords, eight words, or sixteen bytes, as shown in Figure 4-15. In this figure, the `vpaddb` (Add Packed Integers) instruction illustrates packed 8-bit integer addition. The `vpmaxsw` (Packed Signed Integer Maximums) saves the maximum signed word value of each element pair to the specified destination operand. The `vpmulld` (Multiply Packed Integers and Store Low Result) carries out packed signed doubleword multiplication and saves the low-order 32 bits of each result. Finally, the `vpsllq` (Shift Packed Data Left Logical) performs a logical left shift of each quadword element using the bit count that's specified by the immediate operand. Note that this instruction supports the use of an immediate operand to specify the bit count.

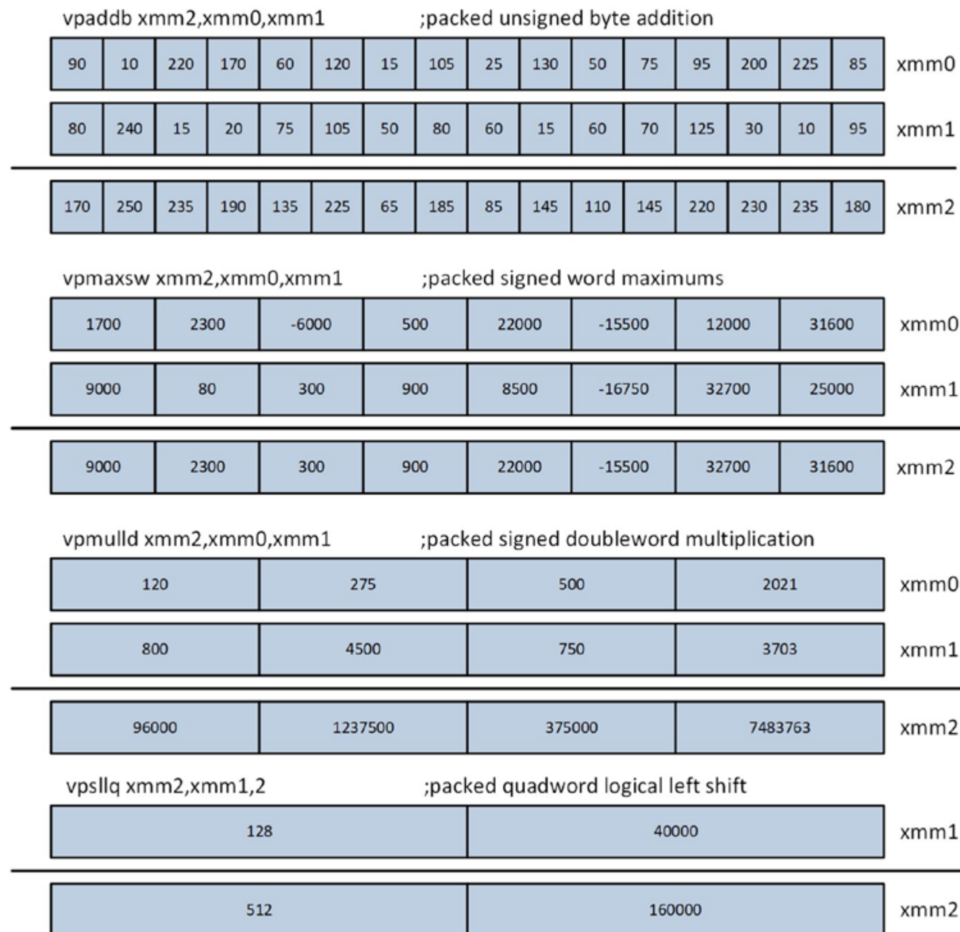



***Figure 4-15.*** *Example AVX packed integer operations*

Most AVX packed integer instructions *do not* update the status flags in the RFLAGS register. This means that error conditions such as arithmetic overflow and underflow are not reported. It also means that the results of a packed integer operation do not directly affect execution of the conditional instructions `cmovcc`, `jcc`, and `setb`. However, programs can employ SIMD-specific techniques to make logical decisions based on the outcome of a packed integer operation. You'll see examples of these techniques in Chapter 7.

## Instruction Set Overview

Table 4-9 lists in alphabetical order commonly-used AVX packed integer instructions. In this table, the mnemonic text `[b|w|d|q]` signifies the size (byte, word, doubleword, or quadword) of the elements that are processed. You'll learn how to use many of these instructions in Chapter 7.

***Table 4-9.*** *Overview of Commonly-Used AVX Packed Integer Instructions*

| Instruction | Description |
| --- | --- |
| `vmov[d|q]` | Move to/from XMM register |
| `vmovdqa` | Move aligned packed integer values |
| `vmovdqu` | Move unaligned packed integer values |
| `vpabs[b|w|d]` | Packed integer absolute value |
| `vpackss[dw|wb]` | Pack with signed saturation |
| `vpackus[dw|wb]` | Pack with unsigned saturation |
| `vpadd[b|w|d|q]` | Packed integer addition |
| `vpadds[b|w]` | Packed integer addition with signed saturation |
| `vpaddus[b|w]` | Packed integer addition with unsigned saturation |
| `vpand` | Packed bitwise AND |
| `vpandn` | Packed bitwise AND NOT |
| `vpcmpeq[b|w|d|q]` | Pack integer compare for equality |
| `vpcmpgt[b|w|d|q]` | Packed signed integer compare for greater than |
| `vpextr[b|w|d|q]` | Extract integer from XMM register |
| `vphadd[w|d]` | Horizontal packed addition |
| `vphsub[w|d]` | Horizontal packed subtraction |
| `vpinsr[b|w|d|q]` | Insert integer into XMM register |
| `vpmaxs[b|w|d]` | Packed signed integer maximum |
| `vpmaxu[b|w|d]` | Packed unsigned integer maximum |
| `vpmins[b|w|d]` | Packed signed integer minimum |
| `vpminu[b|w|d]` | Packed unsigned integer minimum |
| `vpmovsx` | Packed integer move with sign extend |
| `vpmovzx` | Packed integer move with zero extend |
| `vpmuldq` | Packed signed doubleword multiplication |
| `vpmulhuw` | Packed unsigned word multiplication, save high result |

(*continued*)

***Table 4-9.*** (*continued*)

| Instruction | Description |
|---|---|
| `vpmul[h|l]w` | Packed signed word multiplication, save [high \| low] result |
| `vpmull[d|w]` | Packed signed multiplication (save low result) |
| `vpmuludq` | Packed unsigned doubleword multiplication |
| `vpshuf[b|d]` | Shuffle packed integers |
| `vpshuf[h|l]w` | Shuffle [high \| low] packed words |
| `vpslldq` | Shift logical left double quadword |
| `vpsll[w|d|q]` | Packed logical shift left |
| `vpsra[w|d]` | Packed arithmetic shift right |
| `vpsrldq` | Shift logical right double quadword |
| `vpsrl[w|d|q]` | Packed logical shift right |
| `vpsub[b|w|d|q]` | Packed integer subtraction |
| `vpsubs[b|w]` | Packed integer subtraction with signed saturation |
| `vpsubus[b|w]` | Packed integer subtraction with unsigned saturation |
| `vpunpckh[bw|wd|dq]` | Unpack high data |
| `vpunpckl[bw|wd|dq]` | Unpack low data |

# Differences Between x86-AVX and x86-SSE

If you have any previous experience with x86-SSE assembly language programming, you have undoubtedly noticed that a high degree of symmetry exists between this execution environment and x86-AVX. Most x86-SSE instructions have an x86-AVX equivalent that can use either 256-bit or 128-bit wide operands. There are, however, a few important differences between the x86-SSE and x86-AVX execution environments. The remainder of this section explains these differences. Even if you don't have any previous experience with x86-SSE, I still recommend reading this section since it elucidates important details that you need to be aware of when writing code that uses the x86-AVX instruction set.

Within an x86-64 processor that supports x86-AVX, each 256-bit YMM register is partitioned into an upper and lower 128-bit lane. Many x86-AVX instructions carry out their operations using same-lane source and destination operand elements. This independent lane execution tends to be inconspicuous when using x86-AVX instructions that perform arithmetic calculations. However, when using instructions that re-order the data elements of a packed quantity, the effect of separate execution lanes is more evident. For example, the `vshufps` (Packed Interleave Shuffle of Single-Precision Values) instruction rearranges the elements of its source operands according to a control mask that's specified as an immediate operand. The `vpunpcklwd` (Unpack Low Data) instruction interleaves the low-order elements in its two source operands. Figure 4-16 illustrates the in-lane effect of these instructions in greater detail. Note that the floating-point shuffle and unpack operations are carried out independently in both the upper (bits 255:128) and lower (bits 127:0) double quadwords. You'll learn more about the `vshufps` and `vpunpcklwd` instructions in Chapters 6 and 7.

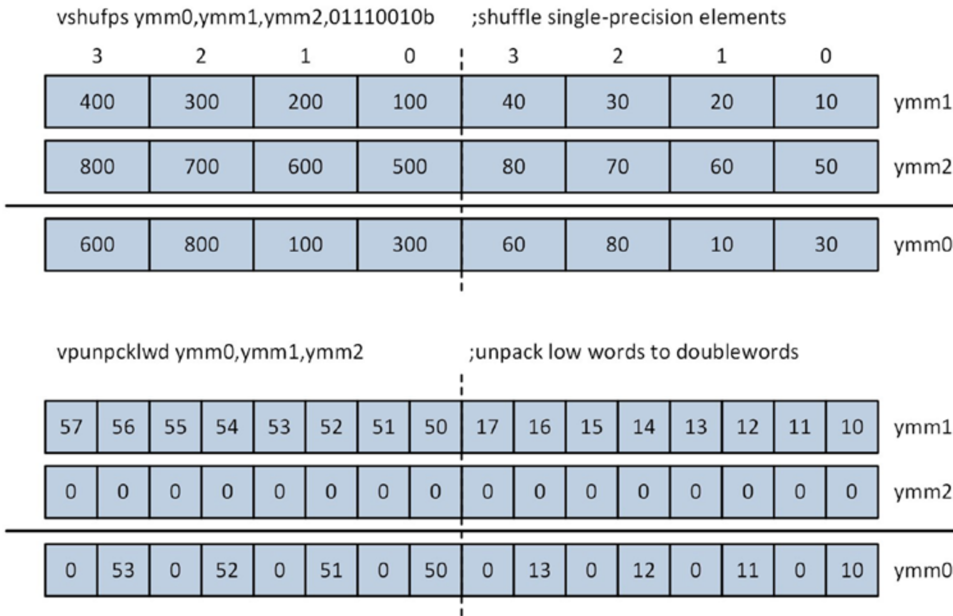



***Figure 4-16.*** *Examples of x86-AVX instruction execution using independent lanes*

The aliasing of the XMM and YMM register sets introduces a few programming issues that software developers need to keep in mind. The first issue relates to the processor's handling of a YMM register's high-order 128 bits when the corresponding XMM register is used as a destination operand. When executing on a processor that supports x86-AVX technology, an x86-SSE instruction that uses an XMM register as a destination operand will never modify the upper 128 bits of the corresponding YMM register. However, the equivalent x86-AVX instruction will zero the upper 128 bits of the respective YMM register. Consider, for example, the following instances of the `(v)cvtps2pd` (Convert Packed Single-Precision to Packed Double-Precision) instruction:

```
cvtps2pd xmm0,xmm1
vcvtps2pd xmm0,xmm1
vcvtps2pd ymm0,ymm1
```

The x86-SSE `cvtps2pd` instruction converts the two packed single-precision floating-point values in the low-order quadword of XMM1 to double-precision floating-point and saves the result in register XMM0. This instruction does not modify the high-order 128 bits of register YMM0. The first `vcvtps2pd` instruction performs the same packed single-precision to packed double-precision conversion operation; it also zeros the high-order 128 bits of YMM0. The second `vcvtps2pd` instruction converts the four packed single-precision floating-point values in the low-order 128 bits of YMM1 to packed double-precision floating-point values and saves the result to YMM0.

X86-AVX relaxes the alignment requirements of x86-SSE for packed operands in memory. Except for instructions that explicitly specify an aligned operand (e.g., `vmovaps`, `vmovdqa`, etc.), proper alignment of a 128-bit or 256-bit wide operand in memory is not mandatory. However, 128-bit and 256-bit wide operands should always be properly aligned whenever possible in order to prevent processing delays that can occur when the processor accesses unaligned operands in memory.

The last issue that programmers need to be aware of involves the intermixing of x86-AVX and x86-SSE code. Programs are allowed to intermix x86-AVX and x86-SSE instructions, but any intermixing should be kept to a minimum in order avoid internal processor state transition penalties that can affect performance. These penalties can occur if the processor is required to preserve the upper 128 bits of each YMM register during a transition from executing x86-AVX to executing x86-SSE instructions. State transition penalties can be completely avoided by using the `vzeroupper` (Zero Upper Bits of YMM Registers) instruction, which zeroes the upper 128 bits of all YMM registers. This instruction should be used prior to any transition from 256-bit x86-AVX code (i.e., any x86-AVX code that uses a YMM register) to x86-SSE code.

One common use of the `vzeroupper` instruction is by a public function that uses 256-bit x86-AVX instructions. These types of functions should include a `vzeroupper` instruction prior to the execution of any `ret` instruction since this prevents processor state transition penalties from occurring in any high-level language code that uses x86-SSE instructions. The `vzeroupper` instruction should also be employed before calling any library functions that might contain x86-SSE code. Later in this book, you'll see several source code examples that demonstrate proper use of the `vzeroupper` instruction. Functions can also use the `vzeroall` (Zero All YMM Registers) instruction instead of `vzeroupper` to avoid potential x86-AVX/x86-SSE state transition penalties.

# Summary

Here are the key learning points for Chapter 4:

- AVX technology is an x86 platform architectural enhancement that facilitates SIMD operations using 128-bit and 256-bit wide packed floating-point operands, both single-precision and double-precision.
- AVX also supports SIMD operations using 128-bit wide packed integer and scalar floating-point operands. AVX2 extends the AVX instruction set to support SIMD operations using 256-bit wide packed integer operands.
- AVX adds 16 YMM (256-bit) and XMM (128-bit) registers to the x86-64 platform. Each XMM register is aliased with the low-order 128 bits of its corresponding YMM register.
- Most AVX instructions use a three-operand syntax that includes two non-destructive source operands.
- AVX floating-point operations conform to the IEEE 754 standard for floating-point arithmetic.
- Programs can use the control and status flags in the MXCSR register to enable floating-point exceptions, detect floating-point error conditions, and configure floating-point rounding.
- Except for instructions that explicitly specify aligned operands, 128-bit and 256-bit wide operands in memory need not be properly aligned. However, SIMD operands in memory should always be properly aligned whenever possible to avoid delays that can occur when the processor accesses an unaligned operand in memory.
- A `vzeroupper` or `vzeroall` instruction should be used in any function that uses a YMM register as an operand in order to avoid potential x86-AVX to x86-SSE state transition performance penalties.

CHAPTER 5

# AVX Programming – Scalar Floating-Point

In the previous chapter, you learned about the architecture and computing capabilities of AVX. In this chapter, you'll learn how to use the AVX instruction set to perform scalar floating-point calculations. The first section includes a couple of sample programs that illustrate basic scalar floating-point arithmetic including addition, subtraction, multiplication, and division. The next section contains code that explains use of the scalar floating-point compare and conversion instructions. This is followed by two examples that demonstrate scalar floating-point operations using arrays and matrices. The final section of this chapter formally describes the Visual C++ calling convention.

All of the sample code in this chapter requires a processor and operating system that support AVX. You can use one of the freely-available tools listed in Appendix A to determine whether or not your computer fulfills this requirement. In Chapter 16, you learn how to programmatically detect the presence of AVX and other x86 processor feature extensions.

---

**Note** Developing software that employs floating-point arithmetic always entails a few caveats. The purpose of the sample code presented in this and subsequent chapters is to illustrate the use of various x86 floating-point instructions. The sample code does not address important floating-point concerns such as rounding errors, numerical stability, or ill-conditioned functions. Software developers must always be cognizant of these issues during the design and implementation of any algorithm that employs floating-point arithmetic. If you're interested in learning more about the potential pitfalls of floating-point arithmetic, you should consult the references listed in Appendix A.

---

## Scalar Floating-Point Arithmetic

The scalar floating-point capabilities of AVX provides programmers with a modern alternative to the floating-point resources of SSE2 and the legacy x87 floating-point unit. The ability to exploit addressable registers means that performing elementary scalar floating-point operations such as addition, subtraction, multiplication, and division is similar to performing integer arithmetic using the general-purpose registers. In this section you learn how to code functions that perform basic floating-point arithmetic using the AVX instruction set. The source code examples demonstrate how to perform fundamental operations using both single-precision and double-precision values. You also learn about floating-point argument passing, return values, and MASM directives.

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_5

## Single-Precision Floating-Point

Listing 5-1 (example Ch05_01) shows the C++ and assembly language source code for a simple program that performs Fahrenheit to Celsius temperature conversions using single-precision floating-point arithmetic. The C++ code begins with a declaration for the assembly language function ConvertFtoC_. Note that this function requires one argument of type float and returns a value of type float. A similar declaration is also used for the assembly language function ConvertCtoF_. The remaining C++ code exercises the two temperature conversion functions using several test values and displays the results.

***Listing 5-1.*** Example Ch05_01

```
//------------------------------------------------
//               Ch05_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" float ConvertFtoC_(float deg_f);
extern "C" float ConvertCtoF_(float deg_c);

int main()
{
    const int w = 10;
    float deg_fvals[] = {-459.67f, -40.0f, 0.0f, 32.0f, 72.0f, 98.6f, 212.0f};
    size_t nf = sizeof(deg_fvals) / sizeof(float);

    cout << setprecision(6);

    cout << "\n-------- ConvertFtoC Results --------\n";

    for (size_t i = 0; i < nf; i++)
    {
        float deg_c = ConvertFtoC_(deg_fvals[i]);

        cout << "  i: " << i << "  ";
        cout << "f: " << setw(w) << deg_fvals[i] << "  ";
        cout << "c: " << setw(w) << deg_c << '\n';
    }

    cout << "\n-------- ConvertCtoF Results --------\n";

    float deg_cvals[] = {-273.15f, -40.0f, -17.777778f, 0.0f, 25.0f, 37.0f, 100.0f};
    size_t nc = sizeof(deg_cvals) / sizeof(float);

    for (size_t i = 0; i < nc; i++)
    {
        float deg_f = ConvertCtoF_(deg_cvals[i]);
```

```
        cout << "  i: " << i << "  ";
        cout << "c: " << setw(w) << deg_cvals[i] << "  ";
        cout << "f: " << setw(w) << deg_f << '\n';
    }

    return 0;
}

;-------------------------------------------------
;               Ch05_01.asm
;-------------------------------------------------

        .const
r4_ScaleFtoC   real4 0.55555556              ; 5 / 9
r4_ScaleCtoF   real4 1.8                     ; 9 / 5
r4_32p0        real4 32.0

; extern "C" float ConvertFtoC_(float deg_f)
;
; Returns: xmm0[31:0] = temperature in Celsius.

        .code
ConvertFtoC_ proc
        vmovss xmm1,[r4_32p0]               ;xmm1 = 32
        vsubss xmm2,xmm0,xmm1               ;xmm2 = f - 32

        vmovss xmm1,[r4_ScaleFtoC]          ;xmm1 = 5 / 9
        vmulss xmm0,xmm2,xmm1               ;xmm0 = (f - 32) * 5 / 9
        ret
ConvertFtoC_ endp

; extern "C" float CtoF_(float deg_c)
;
; Returns: xmm0[31:0] = temperature in Fahrenheit.

ConvertCtoF_ proc
        vmulss xmm0,xmm0,[r4_ScaleCtoF]     ;xmm0 = c * 9 / 5
        vaddss xmm0,xmm0,[r4_32p0]          ;xmm0 = c * 9 / 5 + 32
        ret
ConvertCtoF_ endp
        end
```

The assembly language code starts with a `.const` section that defines the constants needed to convert a temperature value from Fahrenheit to Celsius and vice versa. The text `real4` is a MASM directive that allocates storage space for a single-precision floating-point value. Following the `.const` section is the code for function `ConvertFtoC_`. The first instruction of this function, `vmovss xmm1,[r4_32p0]`, loads the single-precision floating-point value 32.0 from memory into register XMM1 (or more precisely into XMM1[31:0]). A memory operand is used here since, unlike the general-purpose registers, floating-point values cannot be used as immediate operands.

Per the Visual C++ calling convention, the first four floating-point argument values are passed to a function using registers XMM0, XMM1, XMM2, and XMM3. This means that upon entry to function `ConvertFtoC_`, register XMM0 contains the argument value `deg_f`. Following execution of the `vmovss` instruction, the `vsubss xmm2,xmm0,xmm1` instruction calculates `deg_f - 32.0` and saves the result to XMM2. Execution of the `vsubss` instruction does not modify the contents of the source operands XMM0 and XMM1. This instruction also copies bits XMM0[127:32] to XMM2[127:32]. The ensuing `vmovss xmm1,[r4_ScaleFtoC]` loads the constant value 0.55555556 (or 5 / 9) into register XMM1. This is followed by a `vmulss xmm0,xmm2,xmm1` instruction that computes `(deg_f - 32.0) * 0.55555556` and saves the multiplicative result (i.e., the converted temperature in Celsius) in XMM0. The Visual C++ calling convention designates register XMM0 for floating-point return values. Since the return value is already in XMM0, no additional `vmovss` instructions are necessary.

The assembly language function `ConvertCtoF_` follows next. The code for this function differs slightly from `ConvertFtoC_` in that the floating-point arithmetic instructions use memory operands to reference the required conversion constants. At entry to `ConvertCtoF_`, register XMM0 contains argument value `deg_c`. The instruction `vmulss xmm0,xmm0,[r4_ScaleCtoF]` calculates `deg_c * 1.8`. This is followed by an `vaddss xmm0,xmm0,[r4_32p0]` instruction that calculates `deg_c * 1.8 + 32.0`. At this point it would be scientifically remiss for me not to mention that neither `ConvertFtoC_` nor `ConvertCtoF_` perform any validity checks for argument values that are physically impossible (e.g., -1000 degrees Fahrenheit). Such checks require floating-point compare instructions and you'll learn how to use these instructions later in this chapter. Here are the results for source code example `Ch05_01`.

```
-------- ConvertFtoC Results --------
  i: 0  f:    -459.67  c:    -273.15
  i: 1  f:        -40  c:        -40
  i: 2  f:          0  c:   -17.7778
  i: 3  f:         32  c:          0
  i: 4  f:         72  c:    22.2222
  i: 5  f:       98.6  c:         37
  i: 6  f:        212  c:        100

-------- ConvertCtoF Results --------
  i: 0  c:    -273.15  f:    -459.67
  i: 1  c:        -40  f:        -40
  i: 2  c:   -17.7778  f:          0
  i: 3  c:          0  f:         32
  i: 4  c:         25  f:         77
  i: 5  c:         37  f:       98.6
  i: 6  c:        100  f:        212
```

## Double-Precision Floating-Point

The source code examples presented in this section illustrate simple floating-point arithmetic using double-precision values. Listing 5-2 shows the source code for example `Ch05_02`. In this example, the assembly language function `CalcSphereAreaVolume_` calculates the surface area and volume of a sphere using the supplied radius value.

***Listing 5-2.*** Example Ch05_02

```
//------------------------------------------------
//               Ch05_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void CalcSphereAreaVolume_(double r, double* sa, double* vol);

int _tmain(int argc, _TCHAR* argv[])
{
    double r[] = { 0.0, 1.0, 2.0, 3.0, 5.0, 10.0, 20.0, 32.0 };
    size_t num_r = sizeof(r) / sizeof(double);

    cout << setprecision(8);
    cout << "\n--------- Results for CalcSphereAreaVol -----------\n";

    for (size_t i = 0; i < num_r; i++)
    {
        double sa = -1, vol = -1;

        CalcSphereAreaVolume_(r[i], &sa, &vol);

        cout << "i: " << i << "  ";
        cout << "r: " << setw(6) << r[i] << "  ";
        cout << "sa: " << setw(11) << sa << "  ";
        cout << "vol: " << setw(11) << vol << '\n';
    }

    return 0;
}

;-------------------------------------------------
;               Ch05_02.asm
;-------------------------------------------------

        .const
r8_PI   real8 3.14159265358979323846
r8_4p0  real8 4.0
r8_3p0  real8 3.0

; extern "C" void CalcSphereAreaVolume_(double r, double* sa, double* vol);

        .code
CalcSphereAreaVolume_ proc

; Calculate surface area = 4 * PI * r * r
```

```
        vmulsd xmm1,xmm0,xmm0               ;xmm1 = r * r
        vmulsd xmm2,xmm1,[r8_PI]            ;xmm2 = r * r * PI
        vmulsd xmm3,xmm2,[r8_4p0]           ;xmm3 = r * r * PI * 4

; Calculate volume = sa * r / 3
        vmulsd xmm4,xmm3,xmm0               ;xmm4 = r * r * r * PI * 4
        vdivsd xmm5,xmm4,[r8_3p0]           ;xmm5 = r * r * r * PI * 4 / 3

; Save results
        vmovsd real8 ptr [rdx],xmm3         ;save surface area
        vmovsd real8 ptr [r8],xmm5          ;save volume
        ret
CalcSphereAreaVolume_ endp
        end
```

The declaration of function `CalcSphereAreaVolume_` includes an argument value of type `double` for the radius and two `double*` pointers to return the computed surface area and volume. The surface area and volume of a sphere can be calculated using the following formulas:

$$sa = 4\pi^2$$

$$v = 4\pi r^3 / 3 = (sa)r / 3$$

Similar to the previous example, the assembly language code begins with a `.const` section that defines several constants. The text `real8` is a MASM directive that defines storage space for a double-precision floating-point value. At entry to `CalcSphereAreaVolume_`, XMM0 contains the sphere radius. The `vmulsd xmm1,xmm0,xmm0` instruction squares the radius and saves the result to XMM1. Execution of this instruction also copies the upper 64 bits of XMM0 to the same positions in XMM1 (i.e., XMM0[127:64] is copied to XMM1[127:64]). The ensuing `vmulsd xmm2,xmm1,[r8_PI]` and `vmulsd xmm3,xmm2,[r8_4p0]` instructions calculate `r * r * PI * 4`, which yields the surface area of the sphere.

The next two instructions, `vmulsd xmm4,xmm3,xmm0` and `vdivsd xmm5,xmm4,[r8_3p0]`, calculate the sphere volume. The `vmovsd real8 ptr [rdx],xmm3` and `vmovsd real8 ptr [r8],xmm5` instructions save the calculated surface area and volume values to the specified buffers. Note that the pointer arguments `sa` and `vol` were passed to `CalcSphereAreaVolume_` in registers RDX and R8. When a function uses a mixture of integer (or pointer) and floating-point arguments, the position of the argument in the function declaration determines which general-purpose or XMM registers get used. You'll learn more about this aspect of the Visual C++ calling convention later in this chapter. Here is the output for example `Ch05_02`.

```
--------- Results for CalcSphereAreaVol -----------
i: 0  r:      0  sa:          0  vol:          0
i: 1  r:      1  sa:  12.566371  vol:  4.1887902
i: 2  r:      2  sa:  50.265482  vol:  33.510322
i: 3  r:      3  sa:  113.09734  vol:  113.09734
i: 4  r:      5  sa:  314.15927  vol:  523.59878
i: 5  r:     10  sa:  1256.6371  vol:  4188.7902
i: 6  r:     20  sa:  5026.5482  vol:  33510.322
i: 7  r:     32  sa:  12867.964  vol:  137258.28
```

Listing 5-3 (example Ch05_03) contains the code for the next source code example, which also illustrates how to carry out calculations using double-precision floating-point arithmetic. In this example, the assembly language function CalcDistance_ calculates the Euclidian distance between two points in 3D space using the following equation:

$$dist = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

***Listing 5-3.*** Example Ch05_03

```
//------------------------------------------------
//               Ch05_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <cmath>

using namespace std;

extern "C" double CalcDistance_(double x1, double y1, double z1, double x2, double y2,
double z2);

void Init(double* x, double* y, double* z, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 100};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
    {
        x[i] = ui_dist(rng);
        y[i] = ui_dist(rng);
        z[i] = ui_dist(rng);
    }
}

double CalcDistanceCpp(double x1, double y1, double z1, double x2, double y2, double z2)
{
    double tx = (x2 - x1) * (x2 - x1);
    double ty = (y2 - y1) * (y2 - y1);
    double tz = (z2 - z1) * (z2 - z1);
    double dist = sqrt(tx + ty + tz);

    return dist;
}

int main()
{
    const size_t n = 20;
    double x1[n], y1[n], z1[n];
    double x2[n], y2[n], z2[n];
```

```
    double dist1[n];
    double dist2[n];

    Init(x1, y1, z1, n, 29);
    Init(x2, y2, z2, n, 37);

    for (size_t i = 0; i < n; i++)
    {
        dist1[i] = CalcDistanceCpp(x1[i], y1[i], z1[i], x2[i], y2[i], z2[i]);
        dist2[i] = CalcDistance_(x1[i], y1[i], z1[i], x2[i], y2[i], z2[i]);
    }

    cout << fixed;

    for (size_t i = 0; i < n; i++)
    {
        cout << "i: " << setw(2) << i << "  ";

        cout << setprecision(0);

        cout << "p1(";
        cout << setw(3) << x1[i] << ",";
        cout << setw(3) << y1[i] << ",";
        cout << setw(3) << z1[i] << ") | ";

        cout << "p2(";
        cout << setw(3) << x2[i] << ",";
        cout << setw(3) << y2[i] << ",";
        cout << setw(3) << z2[i] << ") | ";

        cout << setprecision(4);
        cout << "dist1: " << setw(8) << dist1[i] << " | ";
        cout << "dist2: " << setw(8) << dist2[i] << '\n';
    }

    return 0;
}
```

```
;-------------------------------------------------
;               Ch05_03.asm
;-------------------------------------------------

; extern "C" double CalcDistance_(double x1, double y1, double z1, double x2, double y2,
double z2)

        .code
CalcDistance_ proc
; Load arguments from stack
        vmovsd xmm4,real8 ptr [rsp+40]      ;xmm4 = y2
        vmovsd xmm5,real8 ptr [rsp+48]      ;xmm5 = z2
```

```
; Calculate squares of coordinate distances
        vsubsd xmm0,xmm3,xmm0               ;xmm0 = x2 - x1
        vmulsd xmm0,xmm0,xmm0               ;xmm0 = (x2 - x1) * (x2 - x1)

        vsubsd xmm1,xmm4,xmm1               ;xmm1 = y2 - y1
        vmulsd xmm1,xmm1,xmm1               ;xmm1 = (y2 - y1) * (y2 - y1)

        vsubsd xmm2,xmm5,xmm2               ;xmm2 = z2 - z1
        vmulsd xmm2,xmm2,xmm2               ;xmm2 = (z2 - z1) * (z2 - z1)

; Calculate final distance
        vaddsd xmm3,xmm0,xmm1
        vaddsd xmm4,xmm2,xmm3               ;xmm4 = sum of squares
        vsqrtsd xmm0,xmm0,xmm4              ;xmm0 = final distance value
        ret
CalcDistance_ endp
        end
```

If you examine the declaration of function `CalcDistance_`, you will notice that it specifies six double precision argument values. The argument values `x1`, `y1`, `z1`, and `x2` are passed in registers XMM0, XMM1, XMM2, and XMM3, respectively. The final two argument values, `y2` and `z2`, are passed on the stack, as illustrated in Figure 5-1. Note that this figure shows only the low-order quadword of each XMM register; the high-order quadwords are not used to pass argument values and are undefined.

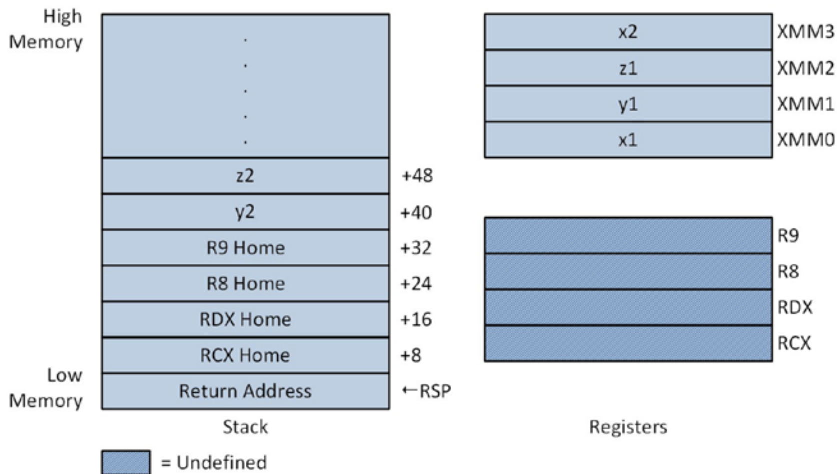



***Figure 5-1.*** *Stack layout and argument registers at entry to `CalcDistance_`*

The function `CalcDistance_` begins with a `vmovsd xmm4,real8 ptr [rsp+40]` instruction that loads argument value `y2` from the stack into register XMM4. This is followed by a `vmovsd xmm5,real8 ptr [rsp+48]` instruction that loads argument value `z2` into register XMM5. The next two instructions, `vsubsd xmm0,xmm3,xmm0` and `vmulsd xmm0,xmm0,xmm0`, calculate `(x2 - x1) * (x2 - x1)`. A similar sequence of instructions is then used to calculate `(y2 - y1) * (y2 - y1)` and `(z2 - z1) * (z2 - z1)`. This is followed by two `vaddsd` instructions that sum the three coordinate squares. A `vsqrtsd xmm0,xmm0,xmm4` instruction computes the final distance. Note that the `vsqrtsd` instruction computes the square root of its second source

operand. Similar to other scalar double-precision floating-point arithmetic instructions, vsqrtsd also copies bits 127:64 of the first source operand to the same bit positions of the destination operand. Here is the output for example Ch05_03:

```
i:  0  p1( 86, 84,  5) | p2( 32,  8, 77) | dist1: 117.7964 | dist2: 117.7964
i:  1  p1( 38, 63, 77) | p2( 28, 49, 86) | dist1:  19.4165 | dist2:  19.4165
i:  2  p1( 17, 18, 54) | p2( 79, 51, 80) | dist1:  74.8933 | dist2:  74.8933
i:  3  p1( 85, 50, 28) | p2( 40, 87, 90) | dist1:  85.0764 | dist2:  85.0764
i:  4  p1( 98, 47, 79) | p2( 28, 85, 38) | dist1:  89.5824 | dist2:  89.5824
i:  5  p1( 21, 78, 36) | p2( 92, 12, 47) | dist1:  97.5602 | dist2:  97.5602
i:  6  p1( 16, 50, 97) | p2( 61, 13, 40) | dist1:  81.5046 | dist2:  81.5046
i:  7  p1( 31, 96, 49) | p2( 31, 37, 45) | dist1:  59.1354 | dist2:  59.1354
i:  8  p1( 13, 87, 40) | p2( 95, 41, 87) | dist1: 105.1142 | dist2: 105.1142
i:  9  p1( 35, 48,  4) | p2( 26, 13, 43) | dist1:  53.1695 | dist2:  53.1695
i: 10  p1( 43, 56, 85) | p2( 88, 17, 45) | dist1:  71.7356 | dist2:  71.7356
i: 11  p1( 59, 88, 77) | p2( 26, 11, 72) | dist1:  83.9226 | dist2:  83.9226
i: 12  p1( 56, 48, 71) | p2(  3, 56, 81) | dist1:  54.5252 | dist2:  54.5252
i: 13  p1( 97, 19, 11) | p2( 36, 35, 58) | dist1:  78.6511 | dist2:  78.6511
i: 14  p1( 50, 79, 74) | p2( 60,  7, 32) | dist1:  83.9524 | dist2:  83.9524
i: 15  p1( 84, 16, 29) | p2( 91,  4, 91) | dist1:  63.5374 | dist2:  63.5374
i: 16  p1( 67, 77, 65) | p2( 86, 47, 59) | dist1:  36.0139 | dist2:  36.0139
i: 17  p1( 67,  1,  3) | p2( 34, 19, 64) | dist1:  71.6519 | dist2:  71.6519
i: 18  p1( 41, 79, 73) | p2( 17,  2, 68) | dist1:  80.8084 | dist2:  80.8084
i: 19  p1( 86, 40, 66) | p2( 76, 12, 61) | dist1:  30.1496 | dist2:  30.1496
```

# Scalar Floating-Point Compares and Conversions

Any function that carries out basic floating-point arithmetic is also likely to perform floating-point compare operations and conversions between integer and floating-point values. The sample source code of this section illustrates how to perform scalar floating-point compares and data conversions. It begins with a couple of examples that demonstrate methods for comparing two floating-point values and making a logical decision based on the result. This is followed by an example that shows floating-point conversion operations using values of different types.

## Floating-Point Compares

Listing 5-4 shows the source code for example Ch05_04, which demonstrates the use of the floating-point compare instructions vcomis[d|s]. Similar to the AVX scalar floating-point arithmetic instructions, the final letter of these mnemonics denotes the operand type (d = double-precision, s = single-precision). The vcomis[d|s] instructions compare two floating-point operands and set status flags in RFLAGS to signify a result of less than, equal, greater than, or unordered. An unordered floating-point compare is true when one or both of the instruction operands is a NaN or erroneously encoded. The assembly language functions CompareVCOMISD_ and CompareVCOMISS_ illustrate the use of the vcomisd and vcomiss instructions, respectively. In the discussions that follow, I'll describe the workings of CompareVCOMISS_; any comments made about this function also apply to CompareVCOMISD_.

***Listing 5-4.*** Example Ch05_04

```
//------------------------------------------------
//               Ch05_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <string>
#include <iostream>
#include <iomanip>
#include <limits>

using namespace std;

extern "C" void CompareVCOMISS_(float a, float b, bool* results);
extern "C" void CompareVCOMISD_(double a, double b, bool* results);

const char* c_OpStrings[] = {"UO", "LT", "LE", "EQ", "NE", "GT", "GE"};
const size_t c_NumOpStrings = sizeof(c_OpStrings) / sizeof(char*);

const string g_Dashes(72, '-');

template <typename T> void PrintResults(T a, T b, const bool* cmp_results)
{
    cout << "a = " << a << ", ";
    cout << "b = " << b << '\n';

    for (size_t i = 0; i < c_NumOpStrings; i++)
    {
        cout << c_OpStrings[i] << '=';
        cout << boolalpha << left << setw(6) << cmp_results[i] << ' ';
    }

    cout << "\n\n";
}

void CompareVCOMISS()
{
    const size_t n = 6;
    float a[n] {120.0, 250.0, 300.0, -18.0, -81.0, 42.0};
    float b[n] {130.0, 240.0, 300.0, 32.0, -100.0, 0.0};

    // Set NAN test value
    b[n - 1] = numeric_limits<float>::quiet_NaN();

    cout << "\nResults for CompareVCOMISS\n";
    cout << g_Dashes << '\n';

    for (size_t i = 0; i < n; i++)
    {
        bool cmp_results[c_NumOpStrings];
```

```
        CompareVCOMISS_(a[i], b[i], cmp_results);
        PrintResults(a[i], b[i], cmp_results);
    }
}

void CompareVCOMISD(void)
{
    const size_t n = 6;
    double a[n] {120.0, 250.0, 300.0, -18.0, -81.0, 42.0};
    double b[n] {130.0, 240.0, 300.0, 32.0, -100.0, 0.0};

    // Set NAN test value
    b[n - 1] = numeric_limits<double>::quiet_NaN();

    cout << "\nResults for CompareVCOMISD\n";
    cout << g_Dashes << '\n';

    for (size_t i = 0; i < n; i++)
    {
        bool cmp_results[c_NumOpStrings];

        CompareVCOMISD_(a[i], b[i], cmp_results);
        PrintResults(a[i], b[i], cmp_results);
    }
}

int main()
{
    CompareVCOMISS();
    CompareVCOMISD();
    return 0;
}

;-------------------------------------------------
;               Ch05_04.asm
;-------------------------------------------------

; extern "C" void CompareVCOMISS_(float a, float b, bool* results);

        .code
CompareVCOMISS_ proc

; Set result flags based on compare status
        vcomiss xmm0,xmm1
        setp byte ptr [r8]                  ;RFLAGS.PF = 1 if unordered
        jnp @F
        xor al,al
        mov byte ptr [r8+1],al              ;Use default result values
        mov byte ptr [r8+2],al
        mov byte ptr [r8+3],al
        mov byte ptr [r8+4],al
```

```
        mov byte ptr [r8+5],al
        mov byte ptr [r8+6],al
        jmp Done

@@:     setb byte ptr [r8+1]                ;set byte if a < b
        setbe byte ptr [r8+2]               ;set byte if a <= b
        sete byte ptr [r8+3]                ;set byte if a == b
        setne byte ptr [r8+4]               ;set byte if a != b
        seta byte ptr [r8+5]                ;set byte if a > b
        setae byte ptr [r8+6]               ;set byte if a >= b

Done:   ret
CompareVCOMISS_ endp

; extern "C" void CompareVCOMISD_(double a, double b, bool* results);

CompareVCOMISD_ proc

; Set result flags based on compare status
        vcomisd xmm0,xmm1
        setp byte ptr [r8]                  ;RFLAGS.PF = 1 if unordered
        jnp @F
        xor al,al
        mov byte ptr [r8+1],al              ;Use default result values
        mov byte ptr [r8+2],al
        mov byte ptr [r8+3],al
        mov byte ptr [r8+4],al
        mov byte ptr [r8+5],al
        mov byte ptr [r8+6],al
        jmp Done

@@:     setb byte ptr [r8+1]                ;set byte if a < b
        setbe byte ptr [r8+2]               ;set byte if a <= b
        sete byte ptr [r8+3]                ;set byte if a == b
        setne byte ptr [r8+4]               ;set byte if a != b
        seta byte ptr [r8+5]                ;set byte if a > b
        setae byte ptr [r8+6]               ;set byte if a >= b

Done:   ret
CompareVCOMISD_ endp
        end
```

The function `CompareVCOMISS_` accepts two argument values of type `float` and a pointer to an array of `bool`s for the compare results. The first instruction of `CompareVCOMISS_`, `vcomiss xmm0,xmm1`, performs a single-precision floating-point compare of argument values a and b. Note that these values were passed to `CompareVCOMISS_` in registers XMM0 and XMM1. Execution of `vcomiss` sets RFLAGS.ZF, RFLAGS.PF, and RFLAGS.ZF, as shown Table 5-1. The setting of these status flags facilitates the use of the conditional instructions `cmovcc`, `jcc`, and `setcc`, as shown in Table 5-2.

***Table 5-1.*** *Status Flags Set by the* `vcomis[d|s]` *Instructions*

| Condition | RFLAGS.ZF | RFLAGS.PF | RFLAGS.CF |
|---|---|---|---|
| `XMM0 > XMM1` | 0 | 0 | 0 |
| `XMM0 == XMM1` | 1 | 0 | 0 |
| `XMM0 < XMM1` | 0 | 0 | 1 |
| Unordered | 1 | 1 | 1 |

***Table 5-2.*** *Condition Codes Following Execution of* `vcomis[d|s]`

| Relational Operator | Condition Code | RFLAGS Test Condition |
|---|---|---|
| `XMM0 < XMM1` | Below (b) | `CF == 1` |
| `XMM0 <= XMM1` | Below or equal (be) | `CF == 1 \|\| ZF == 1` |
| `XMM0 == XMM1` | Equal (e or z) | `ZF == 1` |
| `XMM0 != XMM1` | Not Equal (ne or nz) | `ZF == 0` |
| `XMM0 > XMM1` | Above (a) | `CF == 0 && ZF == 0` |
| `XMM0 >= XMM1` | Above or Equal (ae) | `CF == 0` |
| Unordered | Parity (p) | `PF == 1` |

It should be noted that the status flags shown in Table 5-1 are set only if floating-point exceptions are masked (the default state for Visual C++) and neither `vcomis[d|s]` operand is a QNaN, SNaN, or denormal. If floating-point invalid operation or denormal exceptions are unmasked (MXCSR.IM = 0 or MXCSR.DM = 0) and one of the compare operands is a QNaN, SNaN, or denormal, the processor will generate an exception without updating the status flags in RFLAGS. Chapter 4 contains additional information regarding use of the MXCSR register, QNaNs, SNaNs, and denormals.

Following execution of the `vcomiss xmm0,xmm1` instruction, a series of `setcc` (Set Byte on Condition) instructions are used to highlight the relational operators shown in Table 5-2. The `setp byte ptr [r8]` instruction sets the destination operand byte to 1 if RFLAGS.PF is set (i.e., one of the operands is a QNaN or SNaN); otherwise, the destination operand byte is set to 0. If the compare was ordered, the remaining `setcc` instructions in `CompareVCOMISS_` save all possible compare outcomes by setting each entry in array `results` to 0 or 1. As previously mentioned, functions can also use the `jcc` and `cmovcc` instructions following execution of a `vcomis[d|s]` instruction to perform program jumps or conditional data moves based on the outcome of a floating-point compare. Here is the output for source code example `Ch05_04`:

```
Results for CompareVCOMISS
------------------------------------------------------------------------
a = 120, b = 130
UO=false  LT=true   LE=true   EQ=false  NE=true   GT=false  GE=false

a = 250, b = 240
UO=false  LT=false  LE=false  EQ=false  NE=true   GT=true   GE=true

a = 300, b = 300
UO=false  LT=false  LE=true   EQ=true   NE=false  GT=false  GE=true
```

```
a = -18, b = 32
UO=false  LT=true   LE=true   EQ=false  NE=true   GT=false  GE=false

a = -81, b = -100
UO=false  LT=false  LE=false  EQ=false  NE=true   GT=true   GE=true

a = 42, b = nan
UO=true   LT=false  LE=false  EQ=false  NE=false  GT=false  GE=false

Results for CompareVCOMISD
------------------------------------------------------------------------
a = 120, b = 130
UO=false  LT=true   LE=true   EQ=false  NE=true   GT=false  GE=false

a = 250, b = 240
UO=false  LT=false  LE=false  EQ=false  NE=true   GT=true   GE=true

a = 300, b = 300
UO=false  LT=false  LE=true   EQ=true   NE=false  GT=false  GE=true

a = -18, b = 32
UO=false  LT=true   LE=true   EQ=false  NE=true   GT=false  GE=false

a = -81, b = -100
UO=false  LT=false  LE=false  EQ=false  NE=true   GT=true   GE=true

a = 42, b = nan
UO=true   LT=false  LE=false  EQ=false  NE=false  GT=false  GE=false
```

Listing 5-5 contains the source code for example Ch05_05. This example illustrates the use of the vcmpsd instruction, which compares two double-precision floating-point values using a compare predicate that's specified as an immediate operand. The vcmpsd instruction does not use any of the status bits in RFLAGS to indicate compare results. Instead, it returns a quadword mask of all ones or all zeros to signify a true or false result. The AVX instruction set also includes vcmpss, which can be used to perform single-precision floating-point compares. This instruction is equivalent to the vcmpsd instruction except that it returns a doubleword mask.

***Listing 5-5.*** Example Ch05_05

```
//------------------------------------------------
//               Ch05_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <limits>
#include <string>
```

```
using namespace std;

extern "C" void CompareVCMPSD_(double a, double b, bool* results);

const string g_Dashes(40, '-');

int main()
{
    const char* cmp_names[] =
    {
        "cmp_eq", "cmp_neq", "cmp_lt", "cmp_le",
        "cmp_gt", "cmp_ge", "cmp_ord", "cmp_unord"
    };

    const size_t num_cmp_names = sizeof(cmp_names) / sizeof(char*);

    const size_t n = 6;
    double a[n] = {120.0, 250.0, 300.0, -18.0, -81.0, 42.0};
    double b[n] = {130.0, 240.0, 300.0, 32.0, -100.0, 0.0};

    b[n - 1] = numeric_limits<double>::quiet_NaN();

    cout << "Results for CompareVCMPSD\n";
    cout << g_Dashes << '\n';

    for (size_t i = 0; i < n; i++)
    {
        bool cmp_results[num_cmp_names];

        CompareVCMPSD_(a[i], b[i], cmp_results);

        cout << "a = " << a[i] << "   ";
        cout << "b = " << b[i] << '\n';

        for (size_t j = 0; j < num_cmp_names; j++)
        {
            string s1 = cmp_names[j] + string(":");
            string s2 = ((j & 1) != 0) ? "\n" : "  ";

            cout << left << setw(12) << s1;
            cout << boolalpha << setw(6) << cmp_results[j] << s2;
        }

        cout << "\n";
    }

    return 0;
}
```

```
;-------------------------------------------------
;               cmpequ.asmh
;-------------------------------------------------

; Basic compare predicates
CMP_EQ          equ 00h
CMP_LT          equ 01h
CMP_LE          equ 02h
CMP_UNORD       equ 03h
CMP_NEQ         equ 04h
CMP_NLT         equ 05h
CMP_NLE         equ 06h
CMP_ORD         equ 07h

; Extended compare predicates for AVX
CMP_EQU_UQ      equ 08h
CMP_NGE         equ 09h
CMP_NGT         equ 0Ah
CMP_FALSE       equ 0Bh
CMP_NEQ_OQ      equ 0Ch
CMP_GE          equ 0Dh
CMP_GT          equ 0Eh
CMP_TRUE        equ 0Fh
CMP_EQ_OS       equ 10h
CMP_LT_OQ       equ 11h
CMP_LE_OQ       equ 12h
CMP_UNORD_S     equ 13h
CMP_NEQ_US      equ 14h
CMP_NLT_UQ      equ 15h
CMP_NLE_UQ      equ 16h
CMP_ORD_S       equ 17h
CMP_EQ_US       equ 18h
CMP_NGE_UQ      equ 19h
CMP_NGT_UQ      equ 1Ah
CMP_FALSE_OS    equ 1Bh
CMP_NEQ_OS      equ 1Ch
CMP_GE_OQ       equ 1Dh
CMP_GT_OQ       equ 1Eh
CMP_TRUE_US     equ 1Fh

;-------------------------------------------------
;               Ch05_05.asm
;-------------------------------------------------

        include <cmpequ.asmh>

; extern "C" void CompareVCMPSD_(double a, double b, bool* results)

        .code
CompareVCMPSD_ proc
```

```
; Perform compare for equality
        vcmpsd xmm2,xmm0,xmm1,CMP_EQ        ;perform compare operation
        vmovq rax,xmm2                      ;rax = compare result (all 1s or 0s)
        and al,1                            ;mask out unneeded bits
        mov byte ptr [r8],al                ;save result as C++ bool

; Perform compare for inequality
        vcmpsd xmm2,xmm0,xmm1,CMP_NEQ
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+1],al

; Perform compare for less than
        vcmpsd xmm2,xmm0,xmm1,CMP_LT
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+2],al

; Perform compare for less than or equal
        vcmpsd xmm2,xmm0,xmm1,CMP_LE
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+3],al

; Perform compare for greater than
        vcmpsd xmm2,xmm0,xmm1,CMP_GT
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+4],al

; Perform compare for greater than or equal
        vcmpsd xmm2,xmm0,xmm1,CMP_GE
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+5],al

; Perform compare for ordered
        vcmpsd xmm2,xmm0,xmm1,CMP_ORD
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+6],al

; Perform compare for unordered
        vcmpsd xmm2,xmm0,xmm1,CMP_UNORD
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+7],al

        ret
CompareVCMPSD_ endp
        end
```

Similar to the previous example, the C++ code for example Ch05_05 contains some test cases that exercise the assembly language function CompareVCMPSD_. Following the C++ code in Listing 5-5 is the assembly language header file cmpequ.asmh. This file contains a collection of equate directives, which are used to assign symbolic names to numeric values. The equate directives in cmpequ.asmh define symbolic names for the compare predicates that are used by a number of x86-AVX scalar and packed compare instructions including vcmpsd. You'll shortly see how this works. There is no standard file extension for an x86 assembly language header file; I use .asmh but .inc is also frequently used.

Using an assembly language header file is similar to using a C++ header file. In the current example, the statement include <cmpequ.asmh> incorporates the contents of cmpequ.asmh into the file Ch05_05_.asm during assembly. The angled brackets surrounding the filename can be omitted if the filename doesn't contain any backslashes or MASM special characters, but it's usually simpler and more consistent to just always use them. Besides equate statements, assembly language header files are often used for macro definitions. You'll learn about macros later in this chapter.

The first instruction of function CompareVCMPSD_, vcmpsd xmm2,xmm0,xmm1,CMP_EQ, compares the contents of registers XMM0 and XMM1 for equality. These registers contain argument values a and b. If a and b are equal, the low-order quadword of XMM2 is set to all ones; otherwise, the low-order quadword is set to all zeros. Note that the vcmpsd instruction requires four operands: an immediate operand that specifies the compare predicate, two source operands (the first source operand must be an XMM register while the second source operand can be an XMM register or an operand in memory), and a destination operand that must be an XMM register. The ensuing vmovq rax,xmm2 instruction copies the low-order quadword of XMM2 (which contains all zeros or all ones) to register RAX. This is followed by an and al,1 instruction that sets register AL to 1 if the compare predicate is true; otherwise AL is set to 0. The final instruction of the sequence, mov byte ptr [r8],al, saves the compare outcome to the array results. The function CompareVCMPSD_ then uses similar instruction sequences to demonstrate other frequently-used compare predicates. Here are the results for example Ch05_05:

```
Results for CompareVCMPSD
----------------------------------------
a = 120   b = 130
cmp_eq:     false   cmp_neq:    true
cmp_lt:     true    cmp_le:     true
cmp_gt:     false   cmp_ge:     false
cmp_ord:    true    cmp_unord:  false

a = 250   b = 240
cmp_eq:     false   cmp_neq:    true
cmp_lt:     false   cmp_le:     false
cmp_gt:     true    cmp_ge:     true
cmp_ord:    true    cmp_unord:  false

a = 300   b = 300
cmp_eq:     true    cmp_neq:    false
cmp_lt:     false   cmp_le:     true
cmp_gt:     false   cmp_ge:     true
cmp_ord:    true    cmp_unord:  false

a = -18   b = 32
cmp_eq:     false   cmp_neq:    true
cmp_lt:     true    cmp_le:     true
cmp_gt:     false   cmp_ge:     false
cmp_ord:    true    cmp_unord:  false
```

```
a = -81   b = -100
cmp_eq:     false   cmp_neq:    true
cmp_lt:     false   cmp_le:     false
cmp_gt:     true    cmp_ge:     true
cmp_ord:    true    cmp_unord:  false

a = 42   b = nan
cmp_eq:     false   cmp_neq:    true
cmp_lt:     false   cmp_le:     false
cmp_gt:     false   cmp_ge:     false
cmp_ord:    false   cmp_unord:  true
```

Many x86 assemblers including MASM support pseudo-op forms of the `vcmpsd` instruction and its single-precision counterpart `vcmpss`. Pseudo-ops are simulated instruction mnemonics with the compare predicate embedded within the mnemonic text. In function `CompareVCMPSD_`, for example, the pseudo-op `vcmpeqsd xmm2,xmm0,xmm1` could have been used instead of the instruction `vcmpsd xmm2,xmm0,xmm1,CMP_EQ`. Personally, I find the standard reference manual mnemonics easier to read since the compare predicate is explicitly specified as an operand instead being buried within the pseudo-op, especially when using one of the more esoteric compare predicates.

In this section, you learned how to perform compare operations using the `vcomi[d|s]` and `vcmps[d|s]` instructions. You might be wondering at this point which compare instructions should be used. For basic scalar floating-point compare operations (e.g., equal, not equal, less than, less than or equal, greater than, and greater than or equal), the `vcomis[d|s]` instructions are slightly simpler to use since they directly set the status flags in RFLAGS. The `vcmps[d|s]` instructions must be used to take advantage of the extended compare predicates that AVX supports. Another reason for using the `vcmps[d|s]` instructions is the similarity between these instructions and the corresponding `vcmpp[d|s]` instructions for packed floating-point operands. You'll learn how to use the packed floating-point compare instructions in Chapter 6.

## Floating-Point Conversions

A common operation in many C++ programs is to cast a single-precision or double-precision floating-point value to an integer or vice versa. Other frequent operations include the promotion of a floating-point value from single-precision to double-precision and the narrowing of a double-precision value to single-precision. AVX includes a number of instructions that perform these types of conversions. Listing 5-6 shows the code for a sample program that demonstrates how to use some of the AVX conversion instructions. It also illustrates how to modify the rounding control field of the MXCSR register in order to change the AVX floating-point rounding mode.

***Listing 5-6.*** Example Ch05_06

```
//------------------------------------------------
//              Ch05_06.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>
#include <string>
```

```
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

// Simple union for data exchange
union Uval
{
    int32_t m_I32;
    int32_t m_I64;
    float m_F32;
    double m_F64;
};

// The order of values below must match the jump table
// that's defined in the .asm file.
enum CvtOp : unsigned int
{
    I32_F32,       // int32_t to float
    F32_I32,       // float to int32_t
    I32_F64,       // int32_t to double
    F64_I32,       // double to int32_t
    I64_F32,       // int64_t to float
    F32_I64,       // float to int64_t
    I64_F64,       // int64_t to double
    F64_I64,       // double to int64_t
    F32_F64,       // float to double
    F64_F32,       // double to float
};

// Enumerated type for rounding mode
enum RoundingMode : unsigned int
{
    Nearest, Down, Up, Truncate
};

const string c_RoundingModeStrings[] = {"Nearest", "Down", "Up", "Truncate"};
const RoundingMode c_RoundingModeVals[] = {RoundingMode::Nearest, RoundingMode::Down,
RoundingMode::Up, RoundingMode::Truncate};
const size_t c_NumRoundingModes = sizeof(c_RoundingModeVals) / sizeof (RoundingMode);

extern "C" RoundingMode GetMxcsrRoundingMode_(void);
extern "C" void SetMxcsrRoundingMode_(RoundingMode rm);
extern "C" bool ConvertScalar_(Uval* a, Uval* b, CvtOp cvt_op);

int main()
{
    Uval src1, src2, src3, src4, src5;

    src1.m_F32 = (float)M_PI;
    src2.m_F32 = (float)-M_E;
```

```
    src3.m_F64 = M_SQRT2;
    src4.m_F64 = M_SQRT1_2;
    src5.m_F64 = 1.0 + DBL_EPSILON;

    for (size_t i = 0; i < c_NumRoundingModes; i++)
    {
        Uval des1, des2, des3, des4, des5;
        RoundingMode rm_save = GetMxcsrRoundingMode_();
        RoundingMode rm_test = c_RoundingModeVals[i];

        SetMxcsrRoundingMode_(rm_test);

        ConvertScalar_(&des1, &src1, CvtOp::F32_I32);
        ConvertScalar_(&des2, &src2, CvtOp::F32_I64);
        ConvertScalar_(&des3, &src3, CvtOp::F64_I32);
        ConvertScalar_(&des4, &src4, CvtOp::F64_I64);
        ConvertScalar_(&des5, &src5, CvtOp::F64_F32);

        SetMxcsrRoundingMode_(rm_save);

        cout << fixed;
        cout << "\nRounding mode = " << c_RoundingModeStrings[rm_test] << '\n';

        cout << "  F32_I32: " << setprecision(8);
        cout << src1.m_F32 << " --> " << des1.m_I32 << '\n';

        cout << "  F32_I64: " << setprecision(8);
        cout << src2.m_F32 << " --> " << des2.m_I64 << '\n';

        cout << "  F64_I32: " << setprecision(8);
        cout << src3.m_F64 << " --> " << des3.m_I32 << '\n';

        cout << "  F64_I64: " << setprecision(8);
        cout << src4.m_F64 << " --> " << des4.m_I64 << '\n';

        cout << "  F64_F32: ";
        cout << setprecision(16) << src5.m_F64 << " --> ";
        cout << setprecision(8) << des5.m_F32 << '\n';
    }

    return 0;

;-------------------------------------------------
;               Ch05_06.asm
;-------------------------------------------------

MxcsrRcMask equ 9fffh                       ;bit pattern for MXCSR.RC
MxcsrRcShift equ 13                         ;shift count for MXCSR.RC

; extern "C" RoundingMode GetMxcsrRoundingMode_(void);
;
```

```
; Description:  The following function obtains the current
;               floating-point rounding mode from MXCSR.RC.
;
; Returns:      Current MXCSR.RC rounding mode.

        .code
GetMxcsrRoundingMode_ proc
        vstmxcsr dword ptr [rsp+8]          ;save mxcsr register
        mov eax,[rsp+8]
        shr eax,MxcsrRcShift                ;eax[1:0] = MXCSR.RC bits
        and eax,3                           ;masked out unwanted bits
        ret
GetMxcsrRoundingMode_ endp

;extern "C" void SetMxcsrRoundingMode_(RoundingMode rm);
;
; Description:  The following function updates the rounding mode
;               value in MXCSR.RC.

SetMxcsrRoundingMode_ proc
        and ecx,3                           ;masked out unwanted bits
        shl ecx,MxcsrRcShift                ;ecx[14:13] = rm

        vstmxcsr dword ptr [rsp+8]          ;save current MXCSR
        mov eax,[rsp+8]
        and eax,MxcsrRcMask                 ;masked out old MXCSR.RC bits
        or eax,ecx                          ;insert new MXCSR.RC bits
        mov [rsp+8],eax
        vldmxcsr dword ptr [rsp+8]          ;load updated MXCSR
        ret
SetMxcsrRoundingMode_ endp

; extern "C" bool ConvertScalar_(Uval* des, const Uval* src, CvtOp cvt_op)
;
; Note:         This function requires linker option /LARGEADDRESSAWARE:NO
;               to be explicitly set.

ConvertScalar_ proc

; Make sure cvt_op is valid, then jump to target conversion code
        mov eax,r8d                         ;eax = CvtOp
        cmp eax,CvtOpTableCount
        jae BadCvtOp                        ;jump if cvt_op is invalid
        jmp [CvtOpTable+rax*8]              ;jump to specified conversion

; Conversions between int32_t and float/double

I32_F32:
        mov eax,[rdx]                       ;load integer value
        vcvtsi2ss xmm0,xmm0,eax             ;convert to float
        vmovss real4 ptr [rcx],xmm0         ;save result
```

```
        mov eax,1
        ret

F32_I32:
        vmovss xmm0,real4 ptr [rdx]         ;load float value
        vcvtss2si eax,xmm0                  ;convert to integer
        mov [rcx],eax                       ;save result
        mov eax,1
        ret

I32_F64:
        mov eax,[rdx]                       ;load integer value
        vcvtsi2sd xmm0,xmm0,eax             ;convert to double
        vmovsd real8 ptr [rcx],xmm0         ;save result
        mov eax,1
        ret

F64_I32:
        vmovsd xmm0,real8 ptr [rdx]         ;load double value
        vcvtsd2si eax,xmm0                  ;convert to integer
        mov [rcx],eax                       ;save result
        mov eax,1
        ret

; Conversions between int64_t and float/double

I64_F32:
        mov rax,[rdx]                       ;load integer value
        vcvtsi2ss xmm0,xmm0,rax             ;convert to float
        vmovss real4 ptr [rcx],xmm0         ;save result
        mov eax,1
        ret

F32_I64:
        vmovss xmm0,real4 ptr [rdx]         ;load float value
        vcvtss2si rax,xmm0                  ;convert to integer
        mov [rcx],rax                       ;save result
        mov eax,1
        ret

I64_F64:
        mov rax,[rdx]                       ;load integer value
        vcvtsi2sd xmm0,xmm0,rax             ;convert to double
        vmovsd real8 ptr [rcx],xmm0         ;save result
        mov eax,1
        ret

F64_I64:
        vmovsd xmm0,real8 ptr [rdx]         ;load double value
        vcvtsd2si rax,xmm0                  ;convert to integer
        mov [rcx],rax                       ;save result
```

```
        mov eax,1
        ret

; Conversions between float and double

F32_F64:
        vmovss xmm0,real4 ptr [rdx]         ;load float value
        vcvtss2sd xmm1,xmm1,xmm0            ;convert to double
        vmovsd real8 ptr [rcx],xmm1         ;save result
        mov eax,1
        ret

F64_F32:
        vmovsd xmm0,real8 ptr [rdx]         ;load double value
        vcvtsd2ss xmm1,xmm1,xmm0            ;convert to float
        vmovss real4 ptr [rcx],xmm1         ;save result
        mov eax,1
        ret

BadCvtOp:
        xor eax,eax                         ;set error return code
        ret

; The order of values in following table must match the enum CvtOp
; that's defined in the .cpp file.

        align 8
CvtOpTable equ $
        qword I32_F32, F32_I32
        qword I32_F64, F64_I32
        qword I64_F32, F32_I64
        qword I64_F64, F64_I64
        qword F32_F64, F64_F32
CvtOpTableCount equ ($ - CvtOpTable) / size qword

ConvertScalar_ endp
        end
```

Near the top of the C++ code is a declaration for `union Uval`, which is used for data exchange purposes. This is followed by two enumerations: one to select a floating-point conversion type (`CvtOp`) and another to specify a floating-point rounding mode (`RoundingMode`). The C++ function `main` initializes a couple of `Uval` instances as test cases and invokes the assembly language function `ConvertScalar_` to perform various conversions using different rounding modes. The result of each conversion operation is then displayed for verification and comparison purposes.

The AVX floating-point rounding mode is determined by the rounding control field (bits 14 and 13) of the MXCSR register, as discussed in Chapter 4. The default rounding mode for Visual C++ programs is round to nearest. According to the Visual C++ calling convention, the values in MXCSR[15:6] (i.e., MXCSR register bits 15 through 6) must be preserved across most function boundaries. The code in `main` fulfills this requirement by calling the function `GetMxcsrRoundingMode_` to save the current rounding mode prior to performing any conversion operations using `ConvertScalar_`. The original rounding mode is ultimately restored using the function `SetMxcsrRoundingMode_`. Note that the original rounding mode is restored

prior to the cout statements in main. Also note that I've simplified the rounding mode save and restore code somewhat by not preserving the rounding mode prior to each use ConvertScalar_ and restoring it immediately afterward.

Listing 5-6 also shows the rounding mode control functions. The function GetMxcsrRoundingMode_ uses a vstmxcsr dword ptr [rsp+8] instruction (Store MXCSR RegisterState) to save the contents of MXCSR to the RCX home area on the stack. Recall that a function can use its home area on the stack for any transient storage purpose. The sole operand of the vstmxcsr instruction must be a doubleword in memory; it cannot be a general-purpose register. The ensuing mov eax,[rsp+8] instruction copies the current MXCSR value into register EAX. This is followed a shift and bitwise AND operation that extracts the rounding control bits. The corresponding SetMxcsrRoundingMode_ function uses the vldmxcsr instruction (Load MXCSR Register) to set a rounding mode. The vldmxcsr instruction also requires its sole operand to be a doubleword in memory. Note that the function SetMxcsrRoundingMode_ also uses the vstmxcsr instruction and some masking operations to ensure that only the MXCSR's rounding control bits are modified when setting a new rounding mode.

The function ConvertScalar_ performs floating-point conversions using the specified numerical arguments and conversion operator. Following validation of the argument cvt_op, a jmp [CvtOpTable+rax*8] instruction transfers control to the appropriate section in the code that performs the actual conversion. Note that this instruction exploits a jump table. Here, register RAX (which contains cvt_op) specifies an index into the table CvtOpTable. The table CvtOpTable is defined immediately after the ret instruction and contains offsets to the various conversion code blocks. You'll learn more about jump tables in Chapter 6.

It is also important to note that the same instruction mnemonic is sometimes used when converting an integer to floating-point and vice versa. For example, the instruction vcvtsi2ss xmm0,xmm0,eax (located near the label I32_F32) converts a 32-bit signed integer to single-precision floating-point, and the instruction vcvtsi2ss xmm0,xmm0,rax (located near the label I64_F32) converts a 64-bit signed integer to single-precision floating-point.

Conversions between two different numerical data types are not always possible. For example, the vcvtss2si instruction cannot convert large floating-point values to signed 32-bit integers. If a particular conversion is impossible and invalid operation exceptions (MXCSR.IM) are masked (the default for Visual C++), the processor sets MXCSR.IE (Invalid Operation Error Flag) and the value 0x80000000 is copied to the destination operand. The output for example Ch05_06 is the following:

---

```
Rounding mode = Nearest
  F32_I32: 3.14159274 --> 3
  F32_I64: -2.71828175 --> -3
  F64_I32: 1.41421356 --> 1
  F64_I64: 0.70710678 --> 1
  F64_F32: 1.0000000000000002 --> 1.00000000

Rounding mode = Down
  F32_I32: 3.14159274 --> 3
  F32_I64: -2.71828175 --> -3
  F64_I32: 1.41421356 --> 1
  F64_I64: 0.70710678 --> 0
  F64_F32: 1.0000000000000002 --> 1.00000000

Rounding mode = Up
  F32_I32: 3.14159274 --> 4
  F32_I64: -2.71828175 --> -2
  F64_I32: 1.41421356 --> 2
  F64_I64: 0.70710678 --> 1
```

```
  F64_F32: 1.0000000000000002 --> 1.00000012

Rounding mode = Truncate
  F32_I32: 3.14159274 --> 3
  F32_I64: -2.71828175 --> -2
  F64_I32: 1.41421356 --> 1
  F64_I64: 0.70710678 --> 0
  F64_F32: 1.0000000000000002 --> 1.00000000
```

## Scalar Floating-Point Arrays and Matrices

In Chapter 3 you learned how to access individual elements and carry out calculations using integer arrays and matrices. In this section, you learn how to perform similar operations using floating-point array and matrices. As you'll soon see, the same assembly language coding techniques are often used for both integer and floating-point arrays and matrices.

### Floating-Point Arrays

Listing 5-7 shows the code for example Ch05_07. This example illustrates how to calculate the sample mean and sample standard deviation of an array of double-precision floating-point values.

***Listing 5-7.*** Example Ch05_07

```
//------------------------------------------------
//               Ch05_07.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

extern "C" bool CalcMeanStdev_(double* mean, double* stdev, const double* x, int n);

bool CalcMeanStdevCpp(double* mean, double* stdev, const double* x, int n)
{
    if (n < 2)
        return false;

    double sum = 0.0;
    for (int i = 0; i < n; i++)
        sum += x[i];

    *mean = sum / n;

    double sum2 = 0.0;
    for (int i = 0; i < n; i++)
```

```
    {
        double temp = x[i] - *mean;
        sum2 += temp * temp;
    }

    *stdev = sqrt(sum2 / (n - 1));
    return true;
}

int main()
{
    double x[] = { 10, 2, 33, 19, 41, 24, 75, 37, 18, 97, 14, 71, 88, 92, 7};
    const int n = sizeof(x) / sizeof(double);

    double mean1 = 0.0, stdev1 = 0.0;
    double mean2 = 0.0, stdev2 = 0.0;

    bool rc1 = CalcMeanStdevCpp(&mean1, &stdev1, x, n);
    bool rc2 = CalcMeanStdev_(&mean2, &stdev2, x, n);

    cout << fixed << setprecision(2);

    for (int i = 0; i < n; i++)
    {
        cout << "x[" << setw(2) << i << "] = ";
        cout << setw(6) << x[i] << '\n';
    }

    cout << setprecision(6);

    cout << '\n';
    cout << "rc1 = " << boolalpha << rc1;
    cout << "  mean1 = " << mean1 << "  stdev1 = " << stdev1 << '\n';
    cout << "rc2 = " << boolalpha << rc2;
    cout << "  mean2 = " << mean2 << "  stdev2 = " << stdev2 << '\n';
}

;-------------------------------------------------
;               Ch05_07.asm
;-------------------------------------------------

; extern "C" bool CalcMeanStdev(double* mean, double* stdev, const double* a, int n);
;
; Returns:      0 = invalid n, 1 = valid n

        .code
CalcMeanStdev_  proc

; Make sure 'n' is valid
        xor eax,eax                         ;set error return code (also i = 0)
        cmp r9d,2
        jl InvalidArg                       ;jump if n < 2
```

```
; Compute sample mean
        vxorpd xmm0,xmm0,xmm0               ;sum = 0.0

@@:     vaddsd xmm0,xmm0,real8 ptr [r8+rax*8]   ;sum += x[i]
        inc eax                                 ;i += 1
        cmp eax,r9d
        jl @B                                  ;jump if i < n

        vcvtsi2sd xmm1,xmm1,r9d             ;convert n to DPFP
        vdivsd xmm3,xmm0,xmm1               ;xmm3 = mean (sum / n)
        vmovsd real8 ptr [rcx],xmm3         ;save mean

; Compute sample stdev
        xor eax,eax                         ;i = 0
        vxorpd xmm0,xmm0,xmm0               ;sum2 = 0.0

@@:     vmovsd xmm1,real8 ptr [r8+rax*8]    ;xmm1 = x[i]
        vsubsd xmm2,xmm1,xmm3               ;xmm2 = x[i] - mean
        vmulsd xmm2,xmm2,xmm2               ;xmm2 = (x[i] - mean) ** 2
        vaddsd xmm0,xmm0,xmm2               ;sum2 += (x[i] - mean) ** 2
        inc eax                             ;i += 1
        cmp eax,r9d
        jl @B                               ;jump if i < n

        dec r9d                             ;r9d = n - 1
        vcvtsi2sd xmm1,xmm1,r9d             ;convert n - 1 to DPFP
        vdivsd xmm0,xmm0,xmm1               ;xmm0 = sum2 / (n - 1)
        vsqrtsd xmm0,xmm0,xmm0              ;xmm0 = stdev
        vmovsd real8 ptr [rdx],xmm0         ;save stdev

        mov eax,1                           ;set success return code

InvalidArg:
        ret
CalcMeanStdev_ endp
        end
```

Here are the formulas that example Ch05_07 uses to calculate the sample mean and sample standard deviation:

$$\bar{x} = \frac{1}{n}\sum_i x_i$$

$$s = \sqrt{\frac{1}{n-1}\sum_i \left(x_i - \bar{x}\right)^2}$$

The C++ code for example Ch05_07 is straightforward. It includes a function named CalcMeanStdevCpp that calculates the sample mean and sample standard deviation of an array of double-precision floating-point values. Note that this function and its assembly language equivalent return the calculated mean and standard deviation using pointers. The remaining C++ code initializes a test array and exercises both calculating functions.

Upon entry to the assembly language function CalcMeanStdev_, the number of array elements n is checked for validity. Note that the number of array elements must be greater than one in order to calculate a sample standard deviation. Following validation of n, the vxorpd,xmm0,xmm0,xmm0 instruction (Bitwise XOR of Packed Double-Precision Floating-Point Values) initializes sum to 0.0. This instruction performs a bitwise XOR operation using all 128 bits of both source operands. A vxorpd instruction is used here to initialize sum to 0.0 since AVX does not include an explicit XOR instruction for scalar floating-point operands.

The code block that calculates the sample mean requires only seven instructions. The first instruction of the summing loop, vaddsd xmm0,xmm0,real8 ptr [r8+rax*8], adds x[i] to sum. The inc eax instruction that follows updates i and the summing loop repeats until i reaches n. Following the summing loop, the instruction vcvtsi2sd xmm1,xmm1,r9d promotes a copy of n to double-precision floating-point, and the ensuing vdivsd xmm3,xmm0,xmm1 instruction calculates the final sample mean. The mean is then saved to the memory location pointed to by RCX.

Calculation of the sample standard deviation begins with two instructions, xor eax,eax and vxorpd xmm0,xmm0,xmm0, that initialize i to 0 and sum2 to 0.0. The ensuing vsubsd, vmulsd, and vaddsd instructions calculate sum2 += (x[i] - mean) ** 2 and the summing loop repeats until all array elements have been processed. Execution of the dec r9d instruction yields the value n - 1. This value is then promoted to double-precision floating-point by the vcvtsi2sd xmm1,xmm1,r9d instruction. The final two arithmetic instructions, vdivsd xmm0,xmm0, xmm1 and vsqrtsd xmm0,xmm0,xmm0, compute the sample standard deviation, and this value is saved to the memory location pointed to by RDX. Here's the output for example Ch05_07:

```
x[ 0] =  10.00
x[ 1] =   2.00
x[ 2] =  33.00
x[ 3] =  19.00
x[ 4] =  41.00
x[ 5] =  24.00
x[ 6] =  75.00
x[ 7] =  37.00
x[ 8] =  18.00
x[ 9] =  97.00
x[10] =  14.00
x[11] =  71.00
x[12] =  88.00
x[13] =  92.00
x[14] =   7.00

rc1 = true  mean1 = 41.866667  stdev1 = 33.530086
rc2 = true  mean2 = 41.866667  stdev2 = 33.530086
```

## Floating-Point Matrices

Chapter 3 presented an example program (see Ch03_03) that carried out calculations using the elements of an integer matrix. In this section, you'll learn how to perform similar calculations using the elements of a single-precision floating-point matrix. Listing 5-8 shows the source code for example Ch05_08.

***Listing 5-8.*** Example Ch05_08

```
//------------------------------------------------
//               Ch05_08.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void CalcMatrixSquaresF32_(float* y, const float* x, float offset, int nrows, int
ncols);

void CalcMatrixSquaresF32Cpp(float* y, const float* x, float offset, int nrows, int ncols)
{
    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
        {
            int kx = j * ncols + i;
            int ky = i * ncols + j;
            y[ky] = x[kx] * x[kx] + offset;
        }
    }
}

int main()
{
    const int nrows = 6;
    const int ncols = 3;
    const float offset = 0.5;
    float y2[nrows][ncols];
    float y1[nrows][ncols];
    float x[nrows][ncols] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 },
                          { 10, 11, 12 }, {13, 14, 15}, {16, 17, 18} };

    CalcMatrixSquaresF32Cpp(&y1[0][0], &x[0][0], offset, nrows, ncols);
    CalcMatrixSquaresF32_(&y2[0][0], &x[0][0], offset, nrows, ncols);

    cout << fixed << setprecision(2);

    cout << "offset = " << setw(2) << offset << '\n';

    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
        {
            cout << "y1[" << setw(2) << i << "][" << setw(2) << j << "] = ";
            cout << setw(6) << y1[i][j] << "   " ;
```

```
            cout << "y2[" << setw(2) << i << "][" << setw(2) << j << "] = ";
            cout << setw(6) << y2[i][j] << "   ";

            cout << "x[" << setw(2) << j << "][" << setw(2) << i << "] = ";
            cout << setw(6) <<  x[j][i] << '\n';

            if (y1[i][j] != y2[i][j])
                cout << "Compare failed\n";
        }
    }

    return 0;
}
```

```
;-------------------------------------------------
;               Ch05_08.asm
;-------------------------------------------------

; void CalcMatrixSquaresF32_(float* y, const float* x, float offset, int nrows, int ncols);
;
; Calculates:     y[i][j] = x[j][i] * x[j][i] + offset

        .code
CalcMatrixSquaresF32_ proc frame

; Function prolog
        push rsi                                ;save caller's rsi
        .pushreg rsi
        push rdi                                ;save caller's rdi
        .pushreg rdi
        .endprolog

; Make sure nrows and ncols are valid
        movsxd r9,r9d                           ;r9 = nrows
        test r9,r9
        jle InvalidCount                        ;jump if nrows <= 0

        movsxd r10,dword ptr [rsp+56]           ;r10 = ncols
        test r10,r10
        jle InvalidCount                        ;jump if ncols <= 0

; Initialize pointers to source and destination arrays
        mov rsi,rdx                             ;rsi = x
        mov rdi,rcx                             ;rdi = y
        xor rcx,rcx                             ;rcx = i

; Perform the required calculations
Loop1:  xor rdx,rdx                             ;rdx = j

Loop2:  mov rax,rdx                             ;rax = j
        imul rax,r10                            ;rax = j * ncols
```

```
        add rax,rcx                         ;rax = j * ncols + i
        vmovss xmm0,real4 ptr [rsi+rax*4]   ;xmm0 = x[j][i]
        vmulss xmm1,xmm0,xmm0               ;xmm1 = x[j][i] * x[j][i]
        vaddss xmm3,xmm1,xmm2               ;xmm2 = x[j][i] * x[j][i] + offset

        mov rax,rcx                         ;rax = i
        imul rax,r10                        ;rax = i * ncols
        add rax,rdx                         ;rax = i * ncols + j;
        vmovss real4 ptr [rdi+rax*4],xmm3   ;y[i][j] = x[j][i] * x[j][i] + offset

        inc rdx                             ;j += 1
        cmp rdx,r10
        jl Loop2                            ;jump if j < ncols

        inc rcx                             ;i += 1
        cmp rcx,r9
        jl Loop1                            ;jump if i < nrows

InvalidCount:

; Function epilog
        pop rdi                             ;restore caller's rdi
        pop rsi                             ;restore caller's rsi
        ret

CalcMatrixSquaresF32_ endp
        end
```

The C++ source code that's shown in Listing 5-8 is similar to what you saw in Chapter 3. The techniques used to calculate the matrix element offsets are identical. The biggest modification made to the C++ code was changing the appropriate matrix type declarations from `int` to `float`. Another difference between this example and the one you saw in Chapter 3 is the addition of the argument `offset` to the declarations of `CalcMatrixSquaresF32Cpp` and `CalcMatrixSquaresF32_`. Both of these functions now calculate `y[i][j] = x[j][i] * x[j][i] + offset`.

Figure 5-2 shows the stack layout and argument registers immediately following execution of the `push rdi` instruction in function `CalcMatrixSquaresF32_`. This figure illustrates argument passing to a function that uses a mixture of integer (or pointer) and floating-point arguments. Per the Visual C++ calling convention, the first four arguments are passed using either a general-purpose or XMM register depending on argument type and position. More specifically, the first argument value is passed using either register RCX or XMM0. The second, third, and fourth arguments are passed using RDX/XMM1, R8/XMM2, or R9/XMM3. Any remaining arguments are passed on the stack.

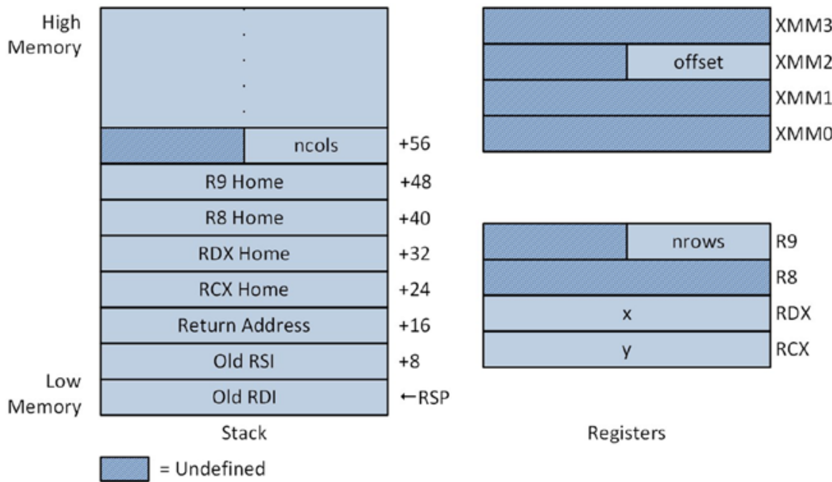



***Figure 5-2.*** *Stack layout and argument registers after execution of* `push rdi` *in* `CalcMatrixSquaresF32_`

The assembly language code for function `CalcMatrixSquaresF32_` is similar to what you studied in Chapter 3. Like the C++ code, the methods used to calculate matrix element offsets are the same. The original matrix element calculating code used integer arithmetic and these instructions have been replaced with analogous AVX scalar single-precision floating-point instructions. Following calculation of the correct matrix element offset, the instruction `vmovss xmm0,real4 ptr [rsi+rax*4]` loads register XMM0 with matrix element `x[j][i]`. The ensuing `vmulss xmm1,xmm0,xmm0` and `vaddss xmm3,xmm1,xmm2` instructions calculate the required result, and a `vmovss real4 ptr [rdi+rax*4],xmm3` instruction saves the result to `y[i][j]`. Here is the output for example `Ch05_08`.

```
offset = 0.50
y1[ 0][ 0] =   1.50   y2[ 0][ 0] =   1.50   x[ 0][ 0] =   1.00
y1[ 0][ 1] =  16.50   y2[ 0][ 1] =  16.50   x[ 1][ 0] =   4.00
y1[ 0][ 2] =  49.50   y2[ 0][ 2] =  49.50   x[ 2][ 0] =   7.00
y1[ 1][ 0] =   4.50   y2[ 1][ 0] =   4.50   x[ 0][ 1] =   2.00
y1[ 1][ 1] =  25.50   y2[ 1][ 1] =  25.50   x[ 1][ 1] =   5.00
y1[ 1][ 2] =  64.50   y2[ 1][ 2] =  64.50   x[ 2][ 1] =   8.00
y1[ 2][ 0] =   9.50   y2[ 2][ 0] =   9.50   x[ 0][ 2] =   3.00
y1[ 2][ 1] =  36.50   y2[ 2][ 1] =  36.50   x[ 1][ 2] =   6.00
y1[ 2][ 2] =  81.50   y2[ 2][ 2] =  81.50   x[ 2][ 2] =   9.00
y1[ 3][ 0] =  16.50   y2[ 3][ 0] =  16.50   x[ 0][ 3] =   4.00
y1[ 3][ 1] =  49.50   y2[ 3][ 1] =  49.50   x[ 1][ 3] =   7.00
y1[ 3][ 2] = 100.50   y2[ 3][ 2] = 100.50   x[ 2][ 3] =  10.00
y1[ 4][ 0] =  25.50   y2[ 4][ 0] =  25.50   x[ 0][ 4] =   5.00
y1[ 4][ 1] =  64.50   y2[ 4][ 1] =  64.50   x[ 1][ 4] =   8.00
y1[ 4][ 2] = 121.50   y2[ 4][ 2] = 121.50   x[ 2][ 4] =  11.00
y1[ 5][ 0] =  36.50   y2[ 5][ 0] =  36.50   x[ 0][ 5] =   6.00
y1[ 5][ 1] =  81.50   y2[ 5][ 1] =  81.50   x[ 1][ 5] =   9.00
y1[ 5][ 2] = 144.50   y2[ 5][ 2] = 144.50   x[ 2][ 5] =  12.00
```

Based on the source code examples in this section, it should be readily apparent that when working with arrays or matrices, techniques independent of the actual data type can be employed to reference specific elements. For-loop constructs can also be coded using methods that are detached from the actual data type.

## Calling Convention

The sample source code presented thus far in this book has informally discussed various aspects of the Visual C++ calling convention. In this section, the calling convention is formally explained. It reiterates some earlier elucidations and also introduces new requirements and features that haven't been discussed. A basic understanding of the calling convention is necessary since it's used extensively in the sample code of subsequent chapters. As a reminder, if you're reading this book to learn x86-64 assembly language programming and plan on using it with a different operating system or high-level language, you should consult the appropriate documentation for information regarding the particulars of that calling convention.

The Visual C++ calling convention designates each x86-64 CPU general-purpose register as volatile or non-volatile. It also applies a volatile or non-volatile classification to each XMM register. An x86-64 assembly language function can modify the contents of any volatile register, but *must* preserve the contents of any non-volatile register it uses. Table 5-3 lists the volatile and non-volatile general-purpose and XMM registers.

On systems that support AVX or AVX2, the high-order 128 bits of each YMM register are classified as volatile. Similarly, the high-order 384 bits of registers ZMM0–ZMM15 are classified as volatile on systems that support AVX-512. Registers ZMM16–ZMM31 and the corresponding YMM and XMM registers are also designated as volatile and need not be preserved. 64-bit Visual C++ programs normally don't use the x87 FPU. Assembly language functions that use this resource are not required to preserve the contents of the x87 FPU register stack, which means that the entire register stack is classified as volatile.

***Table 5-3.*** *Visual C++ 64-Bit Volatile and Non-Volatile Registers*

| Register Group | Volatile Registers | Non-Volatile Registers |
|---|---|---|
| General-purpose | RAX, RCX, RDX, R8, R9, R10, R11 | RBX, RSI, RDI, RBP, RSP, R12, R13, R14, R15 |
| XMM | XMM0 – XMM5 | XMM6 – XMM15 |

The programming requirements imposed on an x86-64 assembly language function by the Visual C++ calling convention vary depending on whether the function is a leaf or non-leaf function. Leaf functions are functions that:

- Do not call any other functions.
- Do not modify the contents of the RSP register.
- Do not allocate any local stack space.
- Do not modify any of the non-volatile general-purpose or XMM registers.
- Do not use exception handling.

64-bit assembly language leaf functions are easier to code, but they're only suitable for relatively simple computational tasks. A non-leaf function can use the entire x86-64 register set, create a stack frame, allocate local stack space, or call other functions provided it complies with the calling convention's precise requirements for prologs and epilogs. The sample code of this section exemplifies these requirements.

In the remainder of this section, you'll examine four source code examples. The first three examples illustrate how to code non-leaf functions using explicit instructions and assembler directives. These programs also convey critical programming information regarding the organization of a non-leaf function stack frame. The fourth example demonstrates how to use several prolog and epilog macros. These macros help automate most of the programming labor that's associated with non-leaf functions.

## Basic Stack Frames

Listing 5-9 shows the source code for example Ch05_09. This program demonstrates how to initialize a stack frame pointer in an assembly language function. Stack frame pointers are used to reference argument values and local variables on the stack. Example Ch05_09 also illustrates some of the programming protocols that an assembly language function prolog and epilog must observe.

***Listing 5-9.*** Example Ch05_09

```
//------------------------------------------------
//              Ch05_09.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cstdint>

using namespace std;

extern "C" int64_t Cc1_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e, int16_t f,
int32_t g, int64_t h);

int main()
{
    int8_t a = 10, e = -20;
    int16_t b = -200, f = 400;
    int32_t c = 300, g = -600;
    int64_t d = 4000, h = -8000;

    int64_t sum = Cc1_(a, b, c, d, e, f, g, h);

    const char nl = '\n';

    cout << "Results for Cc1\n\n";

    cout << "a = " << (int)a << nl;
    cout << "b = " << b << nl;
    cout << "c = " << c << nl;
    cout << "d = " << d << nl;
    cout << "e = " << (int)e << nl;
    cout << "f = " << f << nl;
    cout << "g = " << g << nl;
    cout << "h = " << h << nl;
```

```
    cout << "sum = " << sum << nl;

    return 0;
}

;-------------------------------------------------
;               Ch05_09.asm
;-------------------------------------------------

; extern "C" Int64 Cc1_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e, int16_t f,
int32_t g, int64_t h);

        .code
Cc1_    proc frame

; Function prolog
        push rbp                            ;save caller's rbp register
        .pushreg rbp

        sub rsp,16                          ;allocate local stack space
        .allocstack 16

        mov rbp,rsp                         ;set frame pointer
        .setframe rbp,0

RBP_RA = 24                                 ;offset from rbp to return addr
        .endprolog                          ;mark end of prolog

; Save argument registers to home area (optional)
        mov [rbp+RBP_RA+8],rcx
        mov [rbp+RBP_RA+16],rdx
        mov [rbp+RBP_RA+24],r8
        mov [rbp+RBP_RA+32],r9

; Sum the argument values a, b, c, and d
        movsx rcx,cl                        ;rcx = a
        movsx rdx,dx                        ;rdx = b
        movsxd r8,r8d                       ;r8 = c;
        add rcx,rdx                         ;rcx = a + b
        add r8,r9                           ;r8 = c + d
        add r8,rcx                          ;r8 = a + b + c + d
        mov [rbp],r8                        ;save a + b + c + d

; Sum the argument values e, f, g, and h
        movsx rcx,byte ptr [rbp+RBP_RA+40]  ;rcx = e
        movsx rdx,word ptr [rbp+RBP_RA+48]  ;rdx = f
        movsxd r8,dword ptr [rbp+RBP_RA+56] ;r8 = g
        add rcx,rdx                         ;rcx = e + f
        add r8,qword ptr [rbp+RBP_RA+64]    ;r8 = g + h
        add r8,rcx                          ;r8 = e + f + g + h
```

```
; Compute the final sum
        mov rax,[rbp]                       ;rax = a + b + c + d
        add rax,r8                          ;rax = final sum

; Function epilog
        add rsp,16                          ;release local stack space
        pop rbp                             ;restore caller's rbp register
        ret

Cc1_ endp
        end
```

The purpose of the C++ code in Listing 5-9 is to initialize a test case for the assembly language function `Cc1_`. This function calculates and returns the sum of its eight signed-integer argument values. The results are then displayed using a series stream writes to `cout`.

In the assembly language code, the `Cc1_ proc fame` statement marks the beginning of function `Cc1_`. The `frame` attribute notifies the assembler that the function `Cc1_` uses a stack frame pointer. It also instructs the assembler to generate static table data that the Visual C++ runtime environment uses to process exceptions. The ensuing `push rbp` instruction saves the caller's RBP register on the stack since function `Cc1_` uses this register as its stack frame pointer. The `.pushreg rbp` statement that follows is an assembler directive that saves offset information about the `push rbp` instruction in the exception handling tables. Keep in mind that assembler directives are not executable instructions; they are directions to the assembler on how to perform specific actions during assembly of the source code.

A `sub rsp,16` instruction allocates 16 bytes of stack space for local variables. The function `Cc1_` only uses eight bytes of this space, but the Visual C++ calling convention requires non-leaf functions to maintain 16-byte alignment of the stack pointer outside of the prolog. You'll learn more about stack pointer alignment requirements later in this section. The next statement, `.allocstack 16`, is an assembler directive that saves local stack size allocation information in the runtime exception handling tables.

The `mov rbp,rsp` instruction initializes register RBP as the stack frame pointer, and the `.setframe rbp,0` directive notifies the assembler of this action. The offset value 0 that's included in the `.setframe` directive is the difference in bytes between RSP and RBP. In function `Cc1_`, registers RSP and RBP are the same so the offset value is zero. Later in this section, you learn more about the `.setframe` directive. It should be noted that assembly language functions can use any non-volatile register as a stack frame pointer. Using RBP provides consistency between x86-64 and legacy x86 assembly language code. The final assembler directive, `.endprolog`, signifies the end of the prolog for function `Cc1_`. Figure 5-3 shows the stack layout and argument registers following completion of the prolog.

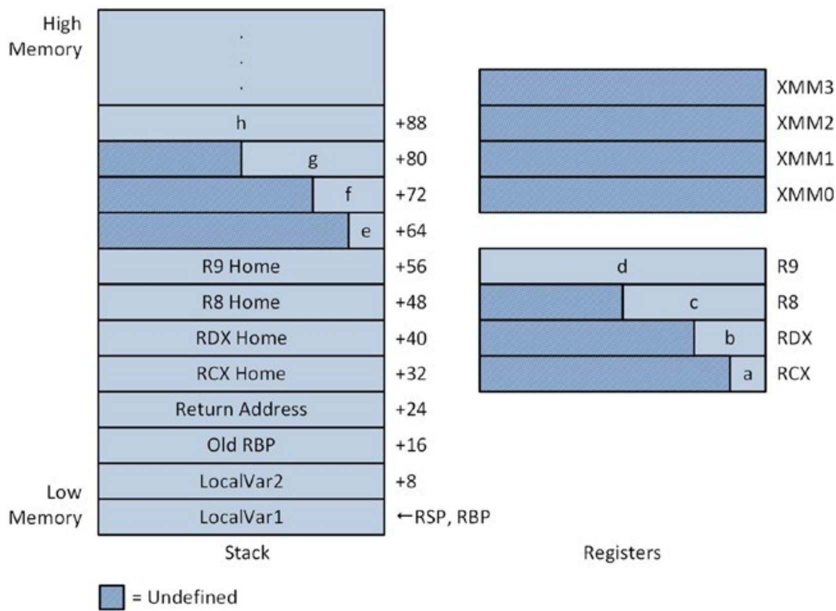



***Figure 5-3.** Stack layout and argument registers of function `Cc1_` following completion of prolog*

The `RBP_RA = 24` statement is a directive similar to an equate that assigns the value 24 to the symbol named `RBP_RA`. This represents the extra offset bytes (compared to a standard leaf function) needed to correctly reference the home area of `Cc1_`, as shown in Figure 5-3. The next block of instructions saves registers RCX, RDX, R8, and R9 to their respective home areas on this stack. This step is optional and included in `Cc1_` for illustrative purposes. Note that the offset of each `mov` instruction includes the symbolic constant `RBP_RA`. Another option allowed by the Visual C++ calling convention is to save an argument register to its corresponding home area prior to the `push rbp` instruction using RSP as a base register (e.g., `mov [rsp+8],rcx`, `mov [rsp+16],rdx`, and so on). Also keep in mind that a function can use its home area to store other temporary values. When used for alternative storage purposes, the home area should not be referenced by an assembly language instruction until after the `.endprolog` directive.

Following the home area save operation, the function `Cc1_` sums argument values `a`, `b`, `c`, and `d`. It then saves this intermediate sum to `LocalVar1` on the stack using a `mov [rbp],r8` instruction. Note that the summation calculation sign-extends argument values `a`, `b`, and `c` using a `movsx` or `movsxd` instruction. A similar sequence of instructions is used to sum argument values `e`, `f`, `g`, and `h`, which are located on the stack and referenced using the stack frame pointer RBP and a constant offset. The symbolic constant `RBP_RA` is also used here to account for the extra stack space needed to reference argument values on the stack. The two intermediate sums are then added to produce the final result in register RAX.

A function epilog must release any local stack storage space that was allocated in the prolog, restore any non-volatile registers that were saved on the stack, and execute a function return. The `add rsp,16` instruction releases the 16 bytes of stack space that `Cc1_` allocated in its prolog. This is followed by a `pop rbp`

instruction, which restores the caller's RBP register. The obligatory `ret` instruction is next. Here is the output for example Ch05_09:

```
Results for Cc1

a = 10
b = -200
c = 300
d = 4000
e = -20
f = 400
g = -600
h = -8000
sum = -4110
```

## Using Non-Volatile General-Purpose Registers

The next sample program is named Ch05_10 and demonstrates how to use the non-volatile general-purpose registers in a 64-bit assembly language function. It also provides additional programming details regarding stack frames and the use of local variables. Listing 5-10 shows the C++ and assembly language source code for sample program Ch05_10.

***Listing 5-10.*** Example Ch05_10

```
//------------------------------------------------
//               Ch05_10.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>

using namespace std;

extern "C" bool Cc2_(const int64_t* a, const int64_t* b, int32_t n, int64_t * sum_a,
int64_t* sum_b, int64_t* prod_a, int64_t* prod_b);

int main()
{
    const int n = 6;
    int64_t a[n] = { 2, -2, -6, 7, 12, 5 };
    int64_t b[n] = { 3, 5, -7, 8, 4, 9 };
    int64_t sum_a, sum_b;
    int64_t prod_a, prod_b;

    bool rc = Cc2_(a, b, n, &sum_a, &sum_b, &prod_a, &prod_b);

    cout << "Results for Cc2\n\n";
```

```
    if (rc)
    {
        const int w = 6;
        const char nl = '\n';
        const char* ws = "   ";

        for (int i = 0; i < n; i++)
        {
            cout << "i: " << setw(w) << i << ws;
            cout << "a: " << setw(w) << a[i] << ws;
            cout << "b: " << setw(w) << b[i] << nl;
        }

        cout <<  nl;
        cout << "sum_a =  " << setw(w) << sum_a << ws;
        cout << "sum_b =  " << setw(w) << sum_b << nl;
        cout << "prod_a = " << setw(w) << prod_a << ws;
        cout << "prod_b = " << setw(w) << prod_b << nl;
    }
    else
        cout << "Invalid return code\n";

    return 0;
}

;-------------------------------------------------
;               Ch05_10.asm
;-------------------------------------------------

; extern "C" void Cc2_(const int64_t* a, const int64_t* b, int32_t n, int64_t* sum_a,
int64_t* sum_b, int64_t* prod_a, int64_t* prod_b)

; Named expressions for constant values:
;
; NUM_PUSHREG   = number of prolog non-volatile register pushes
; STK_LOCAL1    = size in bytes of STK_LOCAL1 area (see figure in text)
; STK_LOCAL2    = size in bytes of STK_LOCAL2 area (see figure in text)
; STK_PAD       = extra bytes (0 or 8) needed to 16-byte align RSP
; STK_TOTAL     = total size in bytes of local stack
; RBP_RA        = number of bytes between RBP and ret addr on stack

NUM_PUSHREG     = 4
STK_LOCAL1      = 32
STK_LOCAL2      = 16
STK_PAD         = ((NUM_PUSHREG AND 1) XOR 1) * 8
STK_TOTAL       = STK_LOCAL1 + STK_LOCAL2 + STK_PAD
RBP_RA          = NUM_PUSHREG * 8 + STK_LOCAL1 + STK_PAD

        .const
TestVal db 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
```

```
        .code
Cc2_    proc frame

; Save non-volatile GP registers on the stack
        push rbp
        .pushreg rbp
        push rbx
        .pushreg rbx
        push r12
        .pushreg r12
        push r13
        .pushreg r13

; Allocate local stack space and set frame pointer
        sub rsp,STK_TOTAL                   ;allocate local stack space
        .allocstack STK_TOTAL

        lea rbp,[rsp+STK_LOCAL2]            ;set frame pointer
        .setframe rbp,STK_LOCAL2

        .endprolog                          ;end of prolog

; Initialize local variables on the stack (demonstration only)
        vmovdqu xmm5, xmmword ptr [TestVal]
        vmovdqa xmmword ptr [rbp-16],xmm5   ;save xmm5 to LocalVar2A/2B
        mov qword ptr [rbp],0aah            ;save 0xaa to LocalVar1A
        mov qword ptr [rbp+8],0bbh          ;save 0xbb to LocalVar1B
        mov qword ptr [rbp+16],0cch         ;save 0xcc to LocalVar1C
        mov qword ptr [rbp+24],0ddh         ;save 0xdd to LocalVar1D

; Save argument values to home area (optional)
        mov qword ptr [rbp+RBP_RA+8],rcx
        mov qword ptr [rbp+RBP_RA+16],rdx
        mov qword ptr [rbp+RBP_RA+24],r8
        mov qword ptr [rbp+RBP_RA+32],r9

; Perform required initializations for processing loop
        test r8d,r8d                        ;is n <= 0?
        jle Error                           ;jump if n <= 0

        xor rbx,rbx                         ;rbx = current element offset
        xor r10,r10                         ;r10 = sum_a
        xor r11,r11                         ;r11 = sum_b
        mov r12,1                           ;r12 = prod_a
        mov r13,1                           ;r13 = prod_b

; Compute the array sums and products
@@:     mov rax,[rcx+rbx]                   ;rax = a[i]
        add r10,rax                         ;update sum_a
        imul r12,rax                        ;update prod_a
        mov rax,[rdx+rbx]                   ;rax = b[i]
```

```
        add r11,rax                         ;update sum_b
        imul r13,rax                        ;update prod_b

        add rbx,8                           ;set ebx to next element
        dec r8d                             ;adjust count
        jnz @B                              ;repeat until done

; Save the final results
        mov [r9],r10                        ;save sum_a
        mov rax,[rbp+RBP_RA+40]             ;rax = ptr to sum_b
        mov [rax],r11                       ;save sum_b
        mov rax,[rbp+RBP_RA+48]             ;rax = ptr to prod_a
        mov [rax],r12                       ;save prod_a
        mov rax,[rbp+RBP_RA+56]             ;rax = ptr to prod_b
        mov [rax],r13                       ;save prod_b
        mov eax,1                           ;set return code to true

; Function epilog
Done:   lea rsp,[rbp+STK_LOCAL1+STK_PAD]    ;restore rsp
        pop r13                             ;restore non-volatile GP registers
        pop r12
        pop rbx
        pop rbp
        ret

Error:  xor eax,eax                         ;set return code to false
        jmp Done
Cc2_    endp
        end
```

Similar to the previous example of this section, the purpose of the code C++ in Listing 5-10 is to prepare a simple test case in order to exercise the assembly language function `Cc2_`. In this example, the function `Cc2_` calculates the sums and products of two 64-bit signed integer arrays. The results are then streamed to `cout`.

Toward the top of the assembly language code is a series of named constants that control how much stack space is allocated in the prolog of function `Cc2_`. Like the previous example, the function `Cc2_` includes the `frame` attribute as part of its `proc` statement to indicate that it uses a stack frame pointer. A series of `push` instructions saves non-volatile registers RBP, RBX, R12, and R13 on the stack. Note that a `.pushreg` directive is used following each `push` instruction, which instructs the assembler to add information about each `push` instruction to the Visual C++ runtime exception handling tables.

A `sub rsp,STK_TOTAL` instruction allocates space on the stack for local variables, and the required `.allocstack STK_TOTAL` directive follows next. Register RBP is then initialized as the function's stack frame pointer using an `lea rbp,[rsp+STK_LOCAL2]` instruction, which sets RBP equal to `rsp + STK_LOCAL2`. Figure 5-4 illustrates the layout of the stack following execution of the `lea` instruction. Positioning RBP so that it "splits" the local stack area into two sections enables the assembler to generate machine code that's slightly more efficient since a larger portion of the local stack area can be referenced using signed 8-bit instead of signed 32-bit displacements. It also simplifies saving and restoring the non-volatile XMM registers, which is discussed later in this section. Following the `lea` instruction is a `.setframe rbp,STK_LOCAL2` directive that enables the assembler to properly configure the runtime exception handling tables. Note that the size parameter of this directive must be an even multiple of 16 and less than or equal to 240. The `.endprolog` directive signifies the end of the prolog for function `Cc2_`.

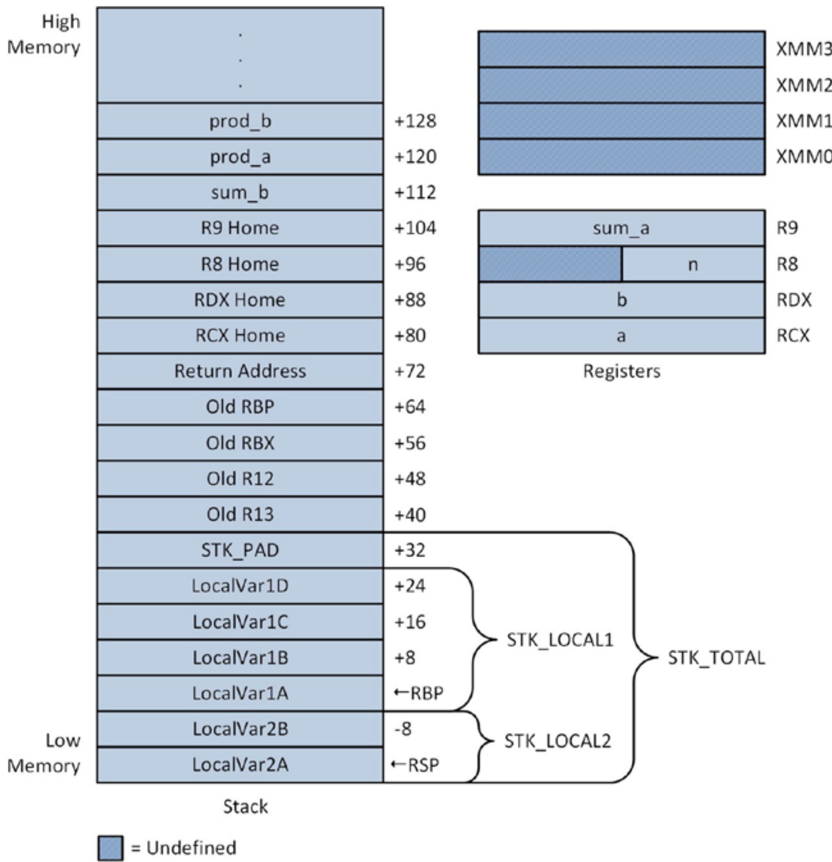



***Figure 5-4.*** *Stack layout and argument registers following execution of the* `lea rbp,[rsp+STK_LOCAL2]` *instruction in function* `Cc2_`

The next code block contains instructions that initialize the local variables on the stack. These instructions are for demonstration purposes only. Note that this block uses a `vmovdqa [rbp-16],xmm5` instruction (Move Aligned Packed Integer Values), which requires its destination operand to be aligned on a 16-byte boundary. This instruction embodies the calling convention's mandatory alignment of the RSP register to a 16-byte boundary. Following initialization of the local variables, the argument registers are saved to their home locations, also merely for demonstration purposes.

The logic of the main processing loop is straightforward. Following validation of argument value `n`, the function `Cc2_` initializes the intermediate values `sum_a` (R10) and `sum_b` (R11) to `0`, and `prod_a` (R12) and `prod_b` (R13) to 1. It then calculates the sum and product of the input arrays `a` and `b`. The final results are saved to the memory locations specified by the caller. Note that the pointers for `sum_b`, `prod_a`, and `prod_b` were passed to `Cc2_` using the stack.

The epilog of function `Cc2_` begins with a `lea rsp,[rbp+STK_LOCAL1+STK_PAD]` instruction that restores register RSP to the value it had just after the `push r13` instruction in the prolog. When restoring RSP in an epilog, the Visual C++ calling convention specifies that either a `lea rsp,[RFP+X]` or `add rsp,X` instruction must be used, where `RFP` denotes the frame pointer register and `X` is a constant value. This limits the number of instruction patterns that the runtime exception handler must identify. The subsequent `pop` instructions restore the non-volatile general-purpose registers prior to execution of the `ret` instruction. According to the Visual C++ calling convention, function epilogs *must* be void of any processing logic including the setting

of a return value since this simplifies the amount of processing that's needed within the Visual C++ runtime exception handler. You'll learn more about the requirements for function epilogs later in this chapter. The output for example Ch05_10 is the following:

```
Results for Cc2

i:      0   a:      2   b:      3
i:      1   a:     -2   b:      5
i:      2   a:     -6   b:     -7
i:      3   a:      7   b:      8
i:      4   a:     12   b:      4
i:      5   a:      5   b:      9

sum_a =      18   sum_b =      22
prod_a =  10080   prod_b = -30240
```

## Using Non-Volatile XMM Registers

Earlier in this chapter, you learned how to use the volatile XMM registers to perform scalar floating-point arithmetic. The next source code example, Ch05_11, illustrates the prolog and epilog conventions that must be observed in order to use the non-volatile XMM registers. Listing 5-11 shows the C++ and assembly language source code for example Ch05_11.

***Listing 5-11.*** Example Ch05_11

```
//------------------------------------------------
//               Ch05_11.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

extern "C" bool Cc3_(const double* r, const double* h, int n, double* sa_cone, double* vol_
cone);

int main()
{
    const int n = 7;
    double r[n] = { 1, 1, 2, 2, 3, 3, 4.25 };
    double h[n] = { 1, 2, 3, 4, 5, 10, 12.5 };

    double sa_cone1[n], sa_cone2[n];
    double vol_cone1[n], vol_cone2[n];
```

```
    // Calculate surface area and volume of right-circular cones
    for (int i = 0; i < n; i++)
    {
        sa_cone1[i] = M_PI * r[i] * (r[i] + sqrt(r[i] * r[i] + h[i] * h[i]));
        vol_cone1[i] = M_PI * r[i] * r[i] * h[i] / 3.0;
    }

    Cc3_(r, h, n, sa_cone2, vol_cone2);

    cout << fixed;
    cout << "Results for Cc3\n\n";

    const int w = 14;
    const char nl = '\n';
    const char sp = ' ';

    for (int i = 0; i < n; i++)
    {
        cout << setprecision(2);
        cout << "r/h: " << setw(w) << r[i] << sp;
        cout << setw(w) << h[i] << nl;

        cout << setprecision(6);
        cout << "sa:  " << setw(w) << sa_cone1[i] << sp;
        cout << setw(w) << sa_cone2[i] << nl;
        cout << "vol: " << setw(w) << vol_cone1[i] << sp;
        cout << setw(w) << vol_cone2[i] << nl;
        cout << nl;
    }

    return 0;
}

;-------------------------------------------------
;               Ch05_11.asm
;-------------------------------------------------

; extern "C" bool Cc3_(const double* r, const double* h, int n, double* sa_cone, double*
vol_cone)

; Named expressions for constant values
;
; NUM_PUSHREG   = number of prolog non-volatile register pushes
; STK_LOCAL1    = size in bytes of STK_LOCAL1 area (see figure in text)
; STK_LOCAL2    = size in bytes of STK_LOCAL2 area (see figure in text)
; STK_PAD       = extra bytes (0 or 8) needed to 16-byte align RSP
; STK_TOTAL     = total size in bytes of local stack
; RBP_RA        = number of bytes between RBP and ret addr on stack

NUM_PUSHREG     = 7
STK_LOCAL1      = 16
```

```
STK_LOCAL2      = 64
STK_PAD         = ((NUM_PUSHREG AND 1) XOR 1) * 8
STK_TOTAL       = STK_LOCAL1 + STK_LOCAL2 + STK_PAD
RBP_RA          = NUM_PUSHREG * 8 + STK_LOCAL1 + STK_PAD

            .const
r8_3p0      real8 3.0
r8_pi       real8 3.14159265358979323846

        .code
Cc3_    proc frame

; Save non-volatile registers on the stack.
        push rbp
        .pushreg rbp
        push rbx
        .pushreg rbx
        push rsi
        .pushreg rsi
        push r12
        .pushreg r12
        push r13
        .pushreg r13
        push r14
        .pushreg r14
        push r15
        .pushreg r15

; Allocate local stack space and initialize frame pointer
        sub rsp,STK_TOTAL                   ;allocate local stack space
        .allocstack STK_TOTAL
        lea rbp,[rsp+STK_LOCAL2]            ;rbp = stack frame pointer
        .setframe rbp,STK_LOCAL2

; Save non-volatile registers XMM12 - XMM15.  Note that STK_LOCAL2 must
; be greater than or equal to the number of XMM register saves times 16.
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+48],xmm12
       .savexmm128 xmm12,48
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+32],xmm13
       .savexmm128 xmm13,32
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+16],xmm14
       .savexmm128 xmm14,16
        vmovdqa xmmword ptr [rbp-STK_LOCAL2],xmm15
       .savexmm128 xmm15,0
        .endprolog

; Access local variables on the stack (demonstration only)
        mov qword ptr [rbp],-1              ;LocalVar1A = -1
        mov qword ptr [rbp+8],-2            ;LocalVar1B = -2

; Initialize the processing loop variables. Note that many of the
```

```
; register initializations below are performed merely to illustrate
; use of the non-volatile GP and XMM registers.
        mov esi,r8d                         ;esi = n
        test esi,esi                        ;is n > 0?
        jg @F                               ;jump if n > 0

        xor eax,eax                         ;set error return code
        jmp done

@@:     xor rbx,rbx                         ;rbx = array element offset
        mov r12,rcx                         ;r12 = ptr to r
        mov r13,rdx                         ;r13 = ptr to h
        mov r14,r9                          ;r14 = ptr to sa_cone
        mov r15,[rbp+RBP_RA+40]             ;r15 = ptr to vol_cone
        vmovsd xmm14,real8 ptr [r8_pi]      ;xmm14 = pi
        vmovsd xmm15,real8 ptr [r8_3p0]     ;xmm15 = 3.0

; Calculate cone surface areas and volumes
; sa = pi * r * (r + sqrt(r * r + h * h))
; vol = pi * r * r * h / 3
@@:     vmovsd xmm0,real8 ptr [r12+rbx]      ;xmm0 = r
        vmovsd xmm1,real8 ptr [r13+rbx]      ;xmm1 = h
        vmovsd xmm12,xmm12,xmm0              ;xmm12 = r
        vmovsd xmm13,xmm13,xmm1              ;xmm13 = h

        vmulsd xmm0,xmm0,xmm0         ;xmm0 = r * r
        vmulsd xmm1,xmm1,xmm1         ;xmm1 = h * h
        vaddsd xmm0,xmm0,xmm1         ;xmm0 = r * r + h * h

        vsqrtsd xmm0,xmm0,xmm0        ;xmm0 = sqrt(r * r + h * h)
        vaddsd xmm0,xmm0,xmm12        ;xmm0 = r + sqrt(r * r + h * h)
        vmulsd xmm0,xmm0,xmm12        ;xmm0 = r * (r + sqrt(r * r + h * h))
        vmulsd xmm0,xmm0,xmm14        ;xmm0 = pi * r * (r + sqrt(r * r + h * h))

        vmulsd xmm12,xmm12,xmm12       ;xmm12 = r * r
        vmulsd xmm13,xmm13,xmm14       ;xmm13 = h * pi
        vmulsd xmm13,xmm13,xmm12       ;xmm13 = pi * r * r * h
        vdivsd xmm13,xmm13,xmm15       ;xmm13 = pi * r * r * h / 3

        vmovsd real8 ptr [r14+rbx],xmm0      ;save surface area
        vmovsd real8 ptr [r15+rbx],xmm13     ;save volume

        add rbx,8                           ;set rbx to next element
        dec esi                             ;update counter
        jnz @B                              ;repeat until done

        mov eax,1                           ;set success return code

; Restore non-volatile XMM registers
Done:   vmovdqa xmm12,xmmword ptr [rbp-STK_LOCAL2+48]
        vmovdqa xmm13,xmmword ptr [rbp-STK_LOCAL2+32]
```

```
        vmovdqa xmm14,xmmword ptr [rbp-STK_LOCAL2+16]
        vmovdqa xmm15,xmmword ptr [rbp-STK_LOCAL2]

; Function epilog
        lea rsp,[rbp+STK_LOCAL1+STK_PAD]    ;restore rsp
        pop r15                             ;restore NV GP registers
        pop r14
        pop r13
        pop r12
        pop rsi
        pop rbx
        pop rbp
        ret

Cc3_    endp
        end
```

The C++ code of example Ch05_11 contains code that calculates the surface area and volume of right-circular cones. It also exercises an assembly language function named Cc3_, which performs the same surface area and volume calculations. The following formulas are used to calculate a cone's surface area and volume:

$$sa = \pi r\left(r + \sqrt{r^2 + h^2}\right)$$

$$vol = \pi r^2 h / 3$$

The function Cc3_ begins by saving the non-volatile general-purpose registers that it uses on the stack. It then allocates the specified amount of local stack space and initializes RBP as the stack frame pointer. The next code block saves non-volatile registers XMM12-XMM15 on the stack using a series of vmovdqa instructions. A .savexmm128 directive must be used after each vmovdqa instruction. Like the other prolog directives, the .savexmm128 directive instructs the assembler to store information regarding an XMM register save operation in its exception handling tables. The offset argument of a .savexmm128 directive represents the displacement of the saved XMM register on the stack relative to the RSP register. Note that the size of STK_LOCAL2 must be greater than or equal to the number of saved XMM registers multiplied by 16. Figure 5-5 illustrates the layout of the stack following execution of the vmovdqa xmmword ptr [rbp-STK_LOCAL2],xmm15 instruction.

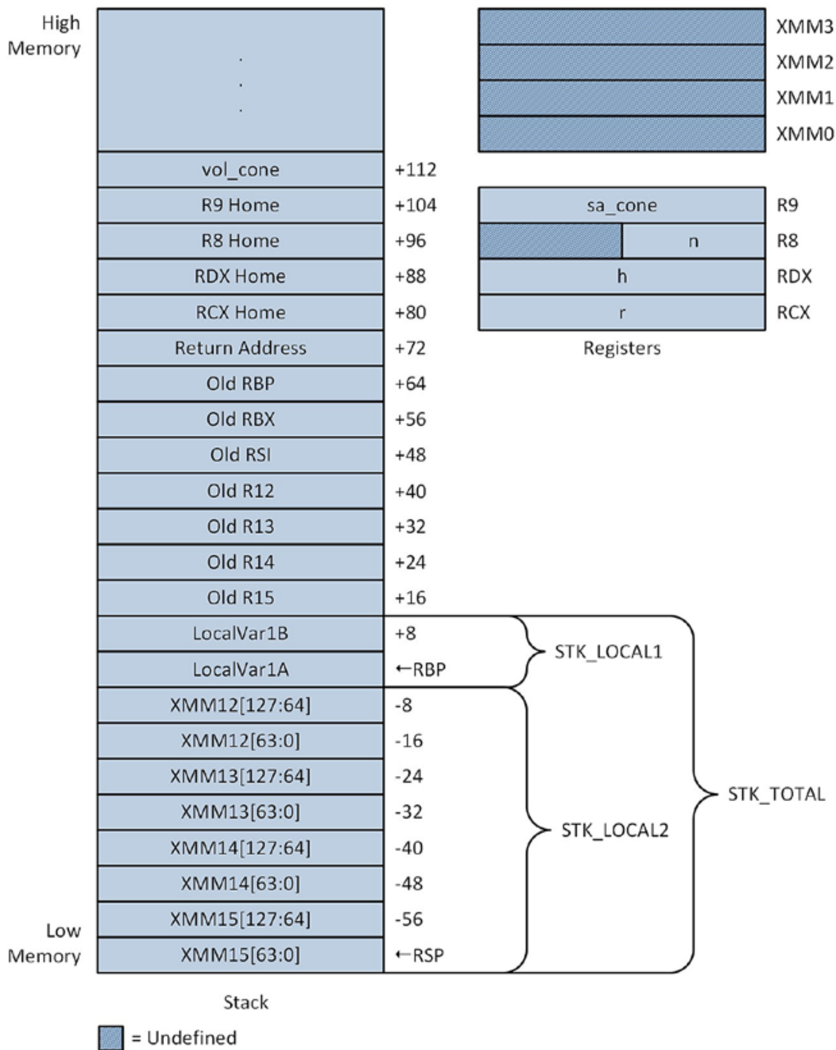



***Figure 5-5.*** *Stack layout and argument registers following execution of the `vmovdqa xmmword ptr [rbp-STK_LOCAL2],xmm15` instruction in function `Cc3_`*

Following the prolog, local variables `LocalVar1A` and `LocalVar1B` are accessed for demonstration purposes only. Initialization of the registers used by the main processing loop occurs next. Note that many of these initializations are either suboptimal or superfluous; they are performed merely to elucidate use of the non-volatile and general-purpose and XMM registers. Calculation of the cone surface areas and volumes is then carried out using AVX double-precision floating-point arithmetic.

Subsequent to the completion of the processing loop, the non-volatile XMM registers are restored using a series of `vmovdqa` instructions. The function `Cc3_` then releases its local stack space and restores the previously saved non-volatile general-purpose registers that it used. Here is the output for example `Ch05_11`.

```
Results for Cc3

r/h:          1.00          1.00
sa:       7.584476      7.584476
vol:      1.047198      1.047198

r/h:          1.00          2.00
sa:      10.166407     10.166407
vol:      2.094395      2.094395

r/h:          2.00          3.00
sa:      35.220717     35.220717
vol:     12.566371     12.566371

r/h:          2.00          4.00
sa:      40.665630     40.665630
vol:     16.755161     16.755161

r/h:          3.00          5.00
sa:      83.229761     83.229761
vol:     47.123890     47.123890

r/h:          3.00         10.00
sa:     126.671905    126.671905
vol:     94.247780     94.247780

r/h:          4.25         12.50
sa:     233.025028    233.025028
vol:    236.437572    236.437572
```

## Macros for Prologs and Epilogs

The purpose of the previous three source code examples was to elucidate use of the Visual C++ calling convention for 64-bit non-leaf functions. The calling convention's rigid requirements for function prologs and epilogs are somewhat lengthy and a potential source of programming errors. It is important to recognize that the stack layout of a non-leaf function is primarily determined by the number of non-volatile (both general-purpose and XMM) registers that must be preserved and the amount of local stack storage space that's needed. A method is needed to automate most of the coding drudgery associated with the calling convention.

Listing 5-12 shows the C++ and assembly language source code for example Ch05_12. This source code example demonstrates how to use several macros that I've written to simplify the coding of a prolog and epilog in a non-leaf function. It also illustrates how to call a C++ library function.

***Listing 5-12.*** Example Ch05_12

```
//------------------------------------------------
//               Ch05_12.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

extern "C" bool Cc4_(const double* ht, const double* wt, int n, double* bsa1, double* bsa2,
double* bsa3);

int main()
{
    const int n = 6;
    const double ht[n] = { 150, 160, 170, 180, 190, 200 };
    const double wt[n] = { 50.0, 60.0, 70.0, 80.0, 90.0, 100.0 };
    double bsa1_a[n], bsa1_b[n];
    double bsa2_a[n], bsa2_b[n];
    double bsa3_a[n], bsa3_b[n];

    for (int i = 0; i < n; i++)
    {
        bsa1_a[i] = 0.007184 * pow(ht[i], 0.725) * pow(wt[i], 0.425);
        bsa2_a[i] = 0.0235 * pow(ht[i], 0.42246) * pow(wt[i], 0.51456);
        bsa3_a[i] = sqrt(ht[i] * wt[i] / 3600.0);
    }

    Cc4_(ht, wt, n, bsa1_b, bsa2_b, bsa3_b);

    cout << "Results for Cc4_\n\n";
    cout << fixed;

    const char sp = ' ';

    for (int i = 0; i < n; i++)
    {
        cout << setprecision(1);
        cout << "height: " << setw(6) << ht[i] << " cm\n";
        cout << "weight: " << setw(6) << wt[i] << " kg\n";

        cout << setprecision(6);

        cout << "BSA (C++):    ";
        cout << setw(10) << bsa1_a[i]  << sp;
        cout << setw(10) << bsa2_a[i]  << sp;
        cout << setw(10) << bsa3_a[i]  << " (sq. m)\n";
```

```
        cout << "BSA (X86-64): ";
        cout << setw(10) << bsa1_b[i]  << sp;
        cout << setw(10) << bsa2_b[i]  << sp;
        cout << setw(10) << bsa3_b[i]  << " (sq. m)\n\n";
    }
    return 0;
}

;-------------------------------------------------
;               Ch05_12.asm
;-------------------------------------------------

; extern "C" bool Cc4_(const double* ht, const double* wt, int n, double* bsa1, double*
bsa2, double* bsa3);

        include <MacrosX86-64-AVX.asmh>

                .const
r8_0p007184     real8 0.007184
r8_0p725        real8 0.725
r8_0p425        real8 0.425
r8_0p0235       real8 0.0235
r8_0p42246      real8 0.42246
r8_0p51456      real8 0.51456
r8_3600p0       real8 3600.0

        .code
        extern pow:proc

Cc4_ proc frame
        _CreateFrame Cc4_,16,64,rbx,rsi,r12,r13,r14,r15
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Save argument registers to home area (optional). Note that the home
; area can also be used to store other transient data values.
        mov qword ptr [rbp+Cc4_OffsetHomeRCX],rcx
        mov qword ptr [rbp+Cc4_OffsetHomeRDX],rdx
        mov qword ptr [rbp+Cc4_OffsetHomeR8],r8
        mov qword ptr [rbp+Cc4_OffsetHomeR9],r9

; Initialize processing loop pointers.  Note that the pointers are
; maintained in non-volatile registers, which eliminates reloads
; after the calls to pow().
        test r8d,r8d                        ;is n > 0?
        jg @F                               ;jump if n > 0

        xor eax,eax                         ;set error return code
        jmp Done
```

```
@@:     mov [rbp],r8d                           ;save n to local var
        mov r12,rcx                             ;r12 = ptr to ht
        mov r13,rdx                             ;r13 = ptr to wt
        mov r14,r9                              ;r14 = ptr to bsa1
        mov r15,[rbp+Cc4_OffsetStackArgs]       ;r15 = ptr to bsa2
        mov rbx,[rbp+Cc4_OffsetStackArgs+8]     ;rbx = ptr to bsa3
        xor rsi,rsi                             ;array element offset

; Allocate home space on stack for use by pow()
        sub rsp,32

; Calculate bsa1 = 0.007184 * pow(ht, 0.725) * pow(wt, 0.425);
@@:     vmovsd xmm0,real8 ptr [r12+rsi]             ;xmm0 = height
        vmovsd xmm8,xmm8,xmm0
        vmovsd xmm1,real8 ptr [r8_0p725]
        call pow                                    ;xmm0 = pow(ht, 0.725)
        vmovsd xmm6,xmm6,xmm0

        vmovsd xmm0,real8 ptr [r13+rsi]             ;xmm0 = weight
        vmovsd xmm9,xmm9,xmm0
        vmovsd xmm1,real8 ptr [r8_0p425]
        call pow                                    ;xmm0 = pow(wt, 0.425)
        vmulsd xmm6,xmm6,real8 ptr [r8_0p007184]
        vmulsd xmm6,xmm6,xmm0                       ;xmm6 = bsa1

; Calculate bsa2 = 0.0235 * pow(ht, 0.42246) * pow(wt, 0.51456);
        vmovsd xmm0,xmm0,xmm8                       ;xmm0 = height
        vmovsd xmm1,real8 ptr [r8_0p42246]
        call pow                                    ;xmm0 = pow(ht, 0.42246)
        vmovsd xmm7,xmm7,xmm0

        vmovsd xmm0,xmm0,xmm9                       ;xmm0 = weight
        vmovsd xmm1,real8 ptr [r8_0p51456]
        call pow                                    ;xmm0 = pow(wt, 0.51456)
        vmulsd xmm7,xmm7,real8 ptr [r8_0p0235]
        vmulsd xmm7,xmm7,xmm0                       ;xmm7 = bsa2

; Calculate bsa3 = sqrt(ht * wt / 60.0);
        vmulsd xmm8,xmm8,xmm9
        vdivsd xmm8,xmm8,real8 ptr [r8_3600p0]
        vsqrtsd xmm8,xmm8,xmm8                  ;xmm8 = bsa3

; Save BSA results
        vmovsd real8 ptr [r14+rsi],xmm6         ;save bsa1 result
        vmovsd real8 ptr [r15+rsi],xmm7         ;save bsa2 result
        vmovsd real8 ptr [rbx+rsi],xmm8         ;save bsa3 result

        add rsi,8                               ;update array offset
        dec dword ptr [rbp]                     ;n = n - 1
        jnz @B
        mov eax,1                               ;set success return code
```

```
Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame rbx,rsi,r12,r13,r14,r15
        ret

Cc4_ endp
        end
```

The purpose of the code in `main` is to initialize several test cases and exercise the assembly language function `Cc4_`. This function computes estimates of human body surface area (BSA) using several well-known equations. These equations are defined in Table 5-4. In this table, each equation uses the symbol *H* for height in centimeters, *W* for weight in kilograms, and *BSA* for body surface area in square meters.

***Table 5-4.*** *Body Surface Area Equations*

| Formula | Equation |
|---|---|
| DuBois and DuBois | $BSA = 0.007184 \times H^{0.725} \times W^{0.425}$ |
| Gehan and George | $BSA = 0.0235 \times H^{0.42246} \times W^{0.51456}$ |
| Mosteller | $BSA = \sqrt{H \times W / 3600}$ |

The assembly language code for example `Ch05_12` begins with an `include` statement that incorporates the contents of the file `MacrosX86-64-AVX.asmh`. This file (source code not shown but included with the Chapter 5 download package) contains a number of macros that help automate much of the coding grunt work that's associated with the Visual C++ calling convention. A macro is an assembler text substitution mechanism that enables a programmer to represent a sequence of assembly language instructions, data definitions, or other statements using a single text string. Assembly language macros are typically employed to generate sequences of instructions that will be used more than once. Macros are also frequently used to avoid the performance overhead of a function call. Source code example `Ch05_12` demonstrates the use of the calling convention macros. You learn how to define your own macros later in this book.

Figure 5-6 shows a generic stack layout diagram for a non-leaf function. Note the similarities between this figure and the more detailed stack layouts of Figures 5-4 and 5-5. The macros defined in `MacrosX86-64-AVX.asmh` assume that a function's basic stack layout will conform to what's shown in Figure 5-6. They enable a function to tailor its own detailed stack frame by specifying the amount of local stack space that's needed and which non-volatile registers must be preserved. The macros also perform most of the required stack offset calculations, which reduces the risk of a programming error in the prolog or epilog.

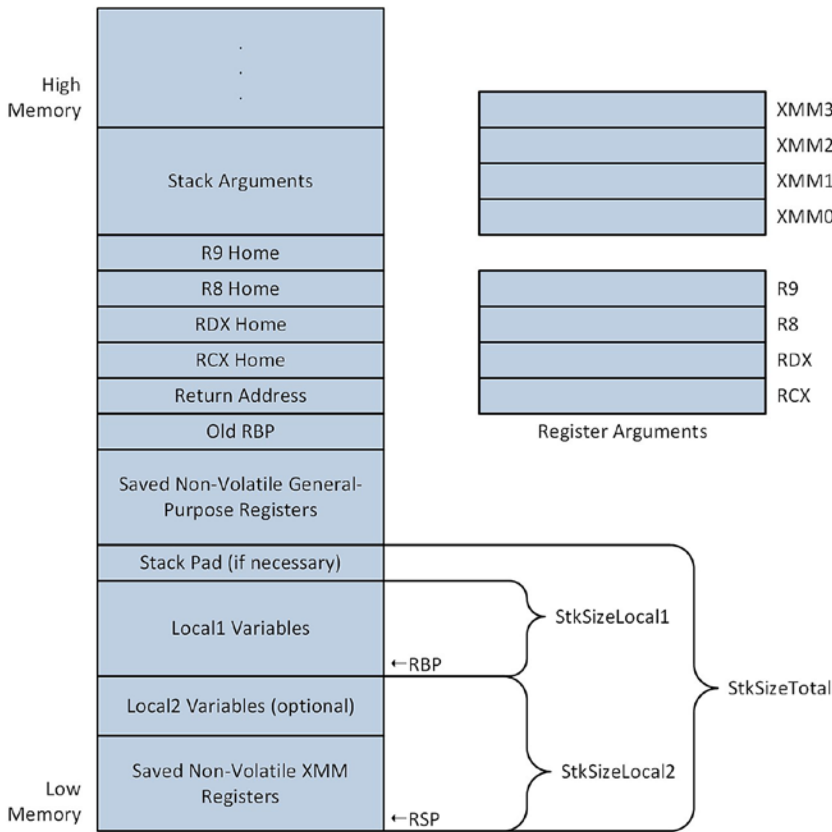



***Figure 5-6.*** *Generic stack layout for a non-leaf function*

Returning to the assembly code, immediately after the `include` statement is a `.const` section that contains definitions for the various floating-point constant values used in the BSA equations. The line `extern pow:proc` enables use of the external C++ library function `pow`. Following the `Cc4_ proc frame` statement, the macro `_CreateFrame` is used to generate the code that initializes the function's stack frame. It also saves the specified non-volatile general-purpose registers on the stack. The macro requires several additional parameters, including a prefix string and the size in bytes of `StkSizeLocal1` and `StkSizeLocal2` (see Figure 5-6). The macro `_CreateFrame` uses the specified prefix string to create symbolic names that can be employed to reference items on the stack. It's somewhat convenient to use a shortened version of the function name as the prefix string but any unique text string can be used. Both `StkSizeLocal1` and `StkSizeLocal2` must be evenly divisible by 16. `StkSizeLocal2` must also be less than or equal to 240, and greater than or equal to the number of saved XMM registers multiplied by 16.

The next statement uses the `_SaveXmmRegs` macro to save the specified non-volatile XMM registers to the XMM save area on the stack. This is followed by the `_EndProlog` macro, which signifies the end of the function's prolog. Subsequent to the completion of the prolog, register RBP is configured as the function's stack frame pointer. It is also safe to use any of the saved non-volatile general-purpose or XMM registers subsequent to the `_EndProlog` macro.

The block of instructions that follows `_EndProlog` saves the argument registers to their home locations on the stack. Note that each `mov` instruction includes a symbolic name that equates to the offset of the register's home area on the stack relative to the RBP register. The symbolic names and the corresponding

offset values were automatically generated by the `_CreateFrame` macro. The home area can also be used to store temporary data instead of the argument registers, as mentioned earlier in this chapter.

Initialization of the processing loop variables occurs next. The value `n` in register `R8D` is checked for validity and saved on the stack as a local variable. Several non-volatile registers are then initialized as pointer registers. Non-volatile registers are used in order to avoid register reloads following each call to the C++ library function `pow`. Note that the pointer to array `bsa2` is loaded from the stack using a `mov r15,[rbp+Cc4_OffsetStackArgs]` instruction. The symbolic constant `Cc4_OffsetStackArgs` also was automatically generated by the macro `_CreateFrame` and equates to the offset of the first stack argument relative to the RBP register. A `mov rbx,[rbp+Cc4_OffsetStackArgs+8]` instruction loads argument `bsa3` into register RBX; the constant 8 is included as part of the source operand displacement since `bsa3` is the second argument passed via the stack.

The Visual C++ calling convention requires the caller of a function to allocate that function's home area on the stack. The `sub rsp,32` instruction performs this operation. The ensuing block of code calculates the BSA values using the equations shown in Table 5-4. Note that registers XMM0 and XMM1 are loaded with the necessary argument values prior to each call to `pow`. Also note that some of the return values from `pow` are preserved in non-volatile XMM registers prior to their actual use.

Following completion of the BSA processing loop is the epilog for `Cc4_`. Before execution of the `ret` instruction, the function must restore all non-volatile XMM and general-purpose registers that it saved in the prolog. The stack frame must also be properly deleted. The `_RestoreXmmRegs` macro restores the non-volatile XMM registers. Note that this macro requires the order of the registers in its argument list to match the register list that was used with the `_SaveXmmRegs` macro. Stack frame cleanup and general-purpose register restores are handled by the `_DeleteFrame` macro. The order of the registers specified in this macro's argument list must be identical to the prolog's `_CreateFrame` macro. The `_DeleteFrame` macro also restores register RSP from RBP, which means that it's not necessary to include an explicit add `rsp,32` instruction to release the home area allocated on the stack for `pow`. Here's the output for example `Ch05_12`.

```
Results for Cc4

height:  150.0 cm
weight:   50.0 kg
BSA (C++):       1.432500   1.460836   1.443376 (sq. m)
BSA (X86-64):    1.432500   1.460836   1.443376 (sq. m)

height:  160.0 cm
weight:   60.0 kg
BSA (C++):       1.622063   1.648868   1.632993 (sq. m)
BSA (X86-64):    1.622063   1.648868   1.632993 (sq. m)

height:  170.0 cm
weight:   70.0 kg
BSA (C++):       1.809708   1.831289   1.818119 (sq. m)
BSA (X86-64):    1.809708   1.831289   1.818119 (sq. m)

height:  180.0 cm
weight:   80.0 kg
BSA (C++):       1.996421   2.009483   2.000000 (sq. m)
BSA (X86-64):    1.996421   2.009483   2.000000 (sq. m)
```

```
height:  190.0 cm
weight:   90.0 kg
BSA (C++):      2.182809   2.184365   2.179449 (sq. m)
BSA (X86-64):   2.182809   2.184365   2.179449 (sq. m)

height:  200.0 cm
weight:  100.0 kg
BSA (C++):      2.369262   2.356574   2.357023 (sq. m)
BSA (X86-64):   2.369262   2.356574   2.357023 (sq. m)
```

# Summary

Here are the key learning points for Chapter 5:

- The vadds[d|s], vsubs[d|s], vmuls[d|s], vdivs[d|s], and vsqrts[d|s] instructions perform basic double-precision and single-precision floating-point arithmetic.
- The vmovs[d|s] instructions copy a scalar floating-point value from one XMM register to another; they are also used to load/store scalar floating-point values from/to memory.
- The vcoms[d|s] instructions compare two scalar floating-point values and set the status flags in RFLAGS to signify the result.
- The vcmps[d|s] instructions compare two scalar floating-point values using a compare predicate. If the compare predicate is true, the destination operand is set to all ones; otherwise, it is set to all zeros.
- The vcvts[d|s]2si instructions convert a scalar floating-point value to a signed integer value; the vcvtsi2s[d|s] instructions perform the opposite conversion.
- The vcvtsd2ss instruction converts a scalar double-precision floating-point value to single-precision; the vcvtss2sd instruction performs the opposite conversion.
- The vldmxcsr instruction loads a value into the MXCSR register; the vstmxcsr instruction saves the current contents of the MXCSR register.
- Leaf functions can be used for simple processing tasks and do not require a prolog or epilog. A non-leaf function must use a prolog and epilog to save and restore non-volatile registers, initialize a stack frame pointer, allocate local storage space on the stack, or call other functions.

CHAPTER 6

# AVX Programming – Packed Floating-Point

The source code examples of the previous chapter elucidated the fundamentals of AVX programming using scalar floating-point arithmetic. In this chapter, you'll learn how to use the AVX instruction set to perform operations using packed floating-point operands. The chapter begins with three source code examples that demonstrate common packed floating-point operations, including basic arithmetic, data comparisons, and data conversions. The next set of source code examples illustrate how to carry out SIMD computations using floating-point arrays. The final two source code examples explain how to use the AVX instruction set to accelerate matrix transposition and multiplication.

In Chapter 4 you learned that AVX supports packed floating-point operations using either 128-bit or 256-bit wide operands. All of the source code examples in this chapter use 128-bit wide packed floating-point operands, both single-precision and double-precision, and the XMM register set. You'll learn how to use 256-bit wide packed floating-point operands and the YMM register set in Chapter 9.

## Packed Floating-Point Arithmetic

Listing 6-1 shows the source code for example `Ch06_01`, which demonstrates how to perform common arithmetic operations using packed single-precision and double-precision floating-point operands. It also highlights proper alignment techniques for packed floating-point operands in memory.

***Listing 6-1.*** Example Ch06_01

```
//------------------------------------------------
//               XmmVal.h
//------------------------------------------------

#pragma once
#include <string>
#include <cstdint>
#include <sstream>
#include <iomanip>

struct XmmVal
{
public:
    union
```

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_6

```
    {
        int8_t m_I8[16];
        int16_t m_I16[8];
        int32_t m_I32[4];
        int64_t m_I64[2];
        uint8_t m_U8[16];
        uint16_t m_U16[8];
        uint32_t m_U32[4];
        uint64_t m_U64[2];
        float m_F32[4];
        double m_F64[2];
    };

//-------------------------------------------------
//               Ch06_01.cpp
//-------------------------------------------------

#include "stdafx.h"
#include <iostream>
#define _USE_MATH_DEFINES
#include <math.h>
#include "XmmVal.h"

using namespace std;

extern "C" void AvxPackedMathF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);
extern "C" void AvxPackedMathF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

void AvxPackedMathF32(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[8];

    a.m_F32[0] = 36.0f;                 b.m_F32[0] = -(float)(1.0 / 9.0);
    a.m_F32[1] = (float)(1.0 / 32.0);   b.m_F32[1] = 64.0f;
    a.m_F32[2] = 2.0f;                  b.m_F32[2] = -0.0625f;
    a.m_F32[3] = 42.0f;                 b.m_F32[3] = 8.666667f;

    AvxPackedMathF32_(a, b, c);

    cout << ("\nResults for AvxPackedMathF32\n");
    cout << "a:       " << a.ToStringF32() << '\n';
    cout << "b:       " << b.ToStringF32() << '\n';
    cout << '\n';
    cout << "addps:   " << c[0].ToStringF32() << '\n';
    cout << "subps:   " << c[1].ToStringF32() << '\n';
    cout << "mulps:   " << c[2].ToStringF32() << '\n';
    cout << "divps:   " << c[3].ToStringF32() << '\n';
    cout << "absps b: " << c[4].ToStringF32() << '\n';
    cout << "sqrtps a:" << c[5].ToStringF32() << '\n';
```

```
    cout << "minps:   " << c[6].ToStringF32() << '\n';
    cout << "maxps:   " << c[7].ToStringF32() << '\n';
}

void AvxPackedMathF64(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[8];

    a.m_F64[0] = 2.0;       b.m_F64[0] = M_E;
    a.m_F64[1] = M_PI;      b.m_F64[1] = -M_1_PI;

    AvxPackedMathF64_(a, b, c);

    cout << ("\nResults for AvxPackedMathF64\n");
    cout << "a:       " << a.ToStringF64() << '\n';
    cout << "b:       " << b.ToStringF64() << '\n';
    cout << '\n';
    cout << "addpd:   " << c[0].ToStringF64() << '\n';
    cout << "subpd:   " << c[1].ToStringF64() << '\n';
    cout << "mulpd:   " << c[2].ToStringF64() << '\n';
    cout << "divpd:   " << c[3].ToStringF64() << '\n';
    cout << "abspd b: " << c[4].ToStringF64() << '\n';
    cout << "sqrtpd a:" << c[5].ToStringF64() << '\n';
    cout << "minpd:   " << c[6].ToStringF64() << '\n';
    cout << "maxpd:   " << c[7].ToStringF64() << '\n';
}

int main()
{
    AvxPackedMathF32();
    AvxPackedMathF64();
    return 0;
}

;-------------------------------------------------
;               Ch06_01.asm
;-------------------------------------------------

            .const
            align 16
AbsMaskF32  dword 7fffffffh, 7fffffffh, 7fffffffh, 7fffffffh  ;Absolute value mask for SPFP
AbsMaskF64  qword 7fffffffffffffffh, 7fffffffffffffffh        ;Absolute value mask for DPFP

; extern "C" void AvxPackedMathF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

            .code
AvxPackedMathF32_ proc
; Load packed SPFP values
        vmovaps xmm0,xmmword ptr [rcx]       ;xmm0 = a
        vmovaps xmm1,xmmword ptr [rdx]       ;xmm1 = b
```

```
; Packed SPFP addition
        vaddps xmm2,xmm0,xmm1
        vmovaps [r8+0],xmm2

; Packed SPFP subtraction
        vsubps xmm2,xmm0,xmm1
        vmovaps [r8+16],xmm2

; Packed SPFP multiplication
        vmulps xmm2,xmm0,xmm1
        vmovaps [r8+32],xmm2

; Packed SPFP division
        vdivps xmm2,xmm0,xmm1
        vmovaps [r8+48],xmm2

; Packed SPFP absolute value (b)
        vandps xmm2,xmm1,xmmword ptr [AbsMaskF32]
        vmovaps [r8+64],xmm2

; Packed SPFP square root (a)
        vsqrtps xmm2,xmm0
        vmovaps [r8+80],xmm2

; Packed SPFP minimum
        vminps xmm2,xmm0,xmm1
        vmovaps [r8+96],xmm2

; Packed SPFP maximum
        vmaxps xmm2,xmm0,xmm1
        vmovaps [r8+112],xmm2
        ret
AvxPackedMathF32_ endp

; extern "C" void AvxPackedMathF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

AvxPackedMathF64_ proc
; Load packed DPFP values
        vmovapd xmm0,xmmword ptr [rcx]       ;xmm0 = a
        vmovapd xmm1,xmmword ptr [rdx]       ;xmm1 = b

; Packed DPFP addition
        vaddpd xmm2,xmm0,xmm1
        vmovapd [r8+0],xmm2

; Packed DPFP subtraction
        vsubpd xmm2,xmm0,xmm1
        vmovapd [r8+16],xmm2
```

```
; Packed DPFP multiplication
        vmulpd xmm2,xmm0,xmm1
        vmovapd [r8+32],xmm2

; Packed DPFP division
        vdivpd xmm2,xmm0,xmm1
        vmovapd [r8+48],xmm2

; Packed DPFP absolute value (b)
        vandpd xmm2,xmm1,xmmword ptr [AbsMaskF64]
        vmovapd [r8+64],xmm2

; Packed DPFP square root (a)
        vsqrtpd xmm2,xmm0
        vmovapd [r8+80],xmm2

; Packed DPFP minimum
        vminpd xmm2,xmm0,xmm1
        vmovapd [r8+96],xmm2

; Packed DPFP maximum
        vmaxpd xmm2,xmm0,xmm1
        vmovapd [r8+112],xmm2
        ret
AvxPackedMathF64_ endp
        end
```

Listing 6-1 begins with the declaration of a C++ structure named XmmVal that's declared in the header file XmmVal.h. This structure contains a publicly-accessible anonymous union that facilitates packed operand data exchange between functions written in C++ and x86 assembly language. The members of this union correspond to the packed data types that can be used with an XMM register. The structure XmmVal also includes several member functions that format and display the contents of an XmmVal variable (the source code for these member functions is not shown but included with the chapter download package).

Near the top of the C++ code are the declarations for the x86-64 assembly language functions AvxPackedMath32_ and AvxPackedMath64_. These functions carry out ordinary packed arithmetic operations using the supplied XmmVal argument values. Note that for both AvxPackedMath32_ and AvxPackedMath64_, arguments a and b are passed by reference instead of value in order to avoid the overhead of an XmmVal copy operation. Using pointers to pass a and b would also work in this example since pointers and references are the same from the perspective of the x86-64 assembly language functions.

Immediately following the assembly language function declarations is the definition for function AvxPackedMathF32. This function contains code that demonstrates packed single-precision floating-point arithmetic. Note that the XmmVal variables a, b, and c are all defined using the specifier alignas(16), which instructs the C++ compiler to align each variable on a 16-byte boundary. The next set of statements initializes the arrays a.m_F32 and b.m_F32 with test values. The C++ code then calls the assembly language function AvxPackedMathF32_ to perform various arithmetic operations using the packed single-precision floating-point operands. The results are then displayed using a series of stream writes to cout. The C++ code also contains a function named AvxPackedMath64 that illustrates arithmetic operations using packed double-precision floating-point operands. The organization of this function is similar to AvxPackedMath32.

The x86-64 assembly language code for example Ch06_01 begins with a .const section that defines packed mask values for calculating floating-point absolute values. The align 16 statement is a MASM directive that instructs the assembler to align the next variable (or instruction) to a 16-byte boundary. Using

this statement guarantees that the mask AbsMaskF32 is properly aligned. Note that unlike x86-SSE, x86-AVX instruction operands in memory need not be properly aligned except for instructions that explicitly specify aligned operands (e.g., vmovaps). However, proper alignment of packed operands in memory is strongly recommended whenever possible in order to avoid the performance penalties that can occur when the processor accesses an unaligned operand. A second align 16 directive is not necessary to ensure alignment of AbsMaskF64 since the size of AbsMaskF32 is 16 bytes, but it would be okay to include such a statement.

The first instruction of AvxPackedMathF32_, vmovaps xmm0,xmmword ptr [rcx] loads argument a (i.e., the four floating-point values stored in XmmVal a) into register XMM0. As mentioned in the previous paragraph, the vmovaps (Move Aligned Packed Single-Precision Floating-Point Values) instruction *requires* source operands in memory to be properly aligned. This is why the alignas(16) specifiers were used in the C++ code. The operator xmmword ptr directs the assembler to treat the memory location pointer to by RCX as a 128-bit operand. In this instance, use of the xmmword ptr operator is optional and employed to improve code readability. The ensuing vmovaps xmm1,xmmword ptr [rdx] instruction loads b into register XMM1. The vaddps xmm2,xmm0,xmm1 instruction (Add Packed Single-Precision Floating-Point Values) performs packed single-precision floating-point addition using the contents of registers XMM0 and XMM1. It then saves the calculated sum to register XMM2, as shown in Figure 6-1. Note that the vaddps instruction does not modify the contents of its two source operands. The vmovaps xmmword ptr [r8],xmm2 that follows saves the result of the packed arithmetic addition to c[0].

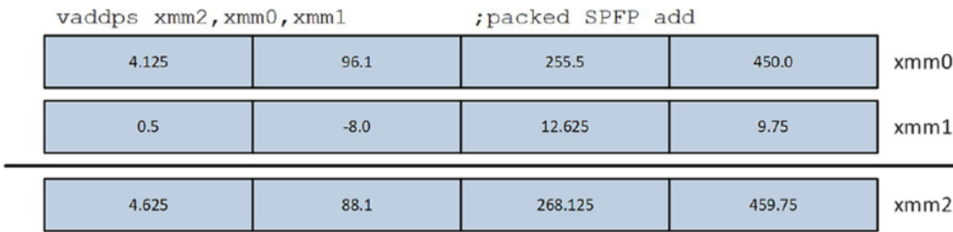



***Figure 6-1.*** *Execution of vaddps instruction*

The ensuing vsubps, vmulps, and vdivps instructions carry out packed single-precision floating-point subtraction, multiplication, and division. This is followed by a vandps xmm2,xmm1,xmmword ptr [AbsMaskF32] instruction that calculates packed absolute values using argument b. The vandps (Bitwise AND of Packed Single-Precision Floating-Point Values) instruction performs a bitwise AND using its two source operands. Note that all of the bits in each AbsMaskF32 doubleword are set to one except the most significant bit, which corresponds to the sign bit of a single-precision floating-point value. A sign bit value of zero corresponds to a positive floating-point number as discussed in Chapter 4. Performing a bitwise AND using this 128-bit wide mask and the packed single-precision floating-point operand b sets the sign bit of each element to zero and generates packed absolute values.

The remaining instructions in AvxPackedMathF32_ calculate packed single-precision floating-point square roots (vsqrtps), minimums (vminps), and maximums (vmaxps). The organization of function AvxPackedMathF64_ is similar to AvxPackedMathF32_. AvxPackedMathF64_ carries out its calculations using the packed double-precision floating-point versions of the same instructions that are used in AvxPackedMathF32_. Here is the output for source code example Ch06_01:

```
Results for AvxPackedMathF32
a:              36.000000        0.031250   |        2.000000       42.000000
b:              -0.111111       64.000000   |       -0.062500        8.666667

addps:          35.888889       64.031250   |        1.937500       50.666668
subps:          36.111111      -63.968750   |        2.062500       33.333332
mulps:          -4.000000        2.000000   |       -0.125000      364.000000
divps:        -324.000000        0.000488   |      -32.000000        4.846154
absps b:         0.111111       64.000000   |        0.062500        8.666667
sqrtps a:        6.000000        0.176777   |        1.414214        6.480741
minps:          -0.111111        0.031250   |       -0.062500        8.666667
maxps:          36.000000       64.000000   |        2.000000       42.000000

Results for AvxPackedMathF64
a:                         2.000000000000   |                  3.141592653590
b:                         2.718281828459   |                 -0.318309886184

addpd:                     4.718281828459   |                  2.823282767406
subpd:                    -0.718281828459   |                  3.459902539774
mulpd:                     5.436563656918   |                 -1.000000000000
divpd:                     0.735758882343   |                 -9.869604401089
abspd b:                   2.718281828459   |                  0.318309886184
sqrtpd a:                  1.414213562373   |                  1.772453850906
minpd:                     2.000000000000   |                 -0.318309886184
maxpd:                     2.718281828459   |                  3.141592653590
```

# Packed Floating-Point Compares

In Chapter 5, you learned how to compare scalar single-precision and double-precision floating-point values using the `vcmps[d|s]` instructions. In this section, you'll learn how to compare packed single-precision and double-precision floating-point values using the `vcmpp[d|s]` instructions. Similar to their scalar counterparts, the packed compare instructions require four operands: a destination operand, two source operands, and an immediate compare predicate. The packed compare instructions signify their results using quadword (`vcmppd`) or doubleword (`vcmpps`) masks of all zeros (false compare result) or all ones (true compare result). Listing 6-2 shows the source code for example `Ch06_02`.

***Listing 6-2.*** Example Ch06_02

```
//------------------------------------------------
//               Ch06_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>
#include <limits>
#include "XmmVal.h"
```

```
using namespace std;

extern "C" void AvxPackedCompareF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);
extern "C" void AvxPackedCompareF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

const char* c_CmpStr[8] =
{
    "EQ", "NE", "LT", "LE", "GT", "GE", "ORDERED", "UNORDERED"
};

void AvxPackedCompareF32(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[8];

    a.m_F32[0] = 2.0;         b.m_F32[0] = 1.0;
    a.m_F32[1] = 7.0;         b.m_F32[1] = 12.0;
    a.m_F32[2] = -6.0;        b.m_F32[2] = -6.0;
    a.m_F32[3] = 3.0;         b.m_F32[3] = 8.0;

    for (int i = 0; i < 2; i++)
    {
        if (i == 1)
            a.m_F32[0] = numeric_limits<float>::quiet_NaN();

        AvxPackedCompareF32_(a, b, c);

        cout << "\nResults for AvxPackedCompareF32 (iteration = " << i << ")\n";
        cout << setw(11) << 'a' << ':' << a.ToStringF32() << '\n';
        cout << setw(11) << 'b' << ':' << b.ToStringF32() << '\n';
        cout << '\n';

        for (int j = 0; j < 8; j++)
            cout << setw(11) << c_CmpStr[j] << ':' << c[j].ToStringX32() << '\n';
    }
}

void AvxPackedCompareF64(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[8];

    a.m_F64[0] = 2.0;       b.m_F64[0] = M_E;
    a.m_F64[1] = M_PI;      b.m_F64[1] = -M_1_PI;

    for (int i = 0; i < 2; i++)
    {
        if (i == 1)
        {
```

```
            a.m_F64[0] = numeric_limits<double>::quiet_NaN();
            b.m_F64[1] = a.m_F64[1];
        }

        AvxPackedCompareF64_(a, b, c);

        cout << "\nResults for AvxPackedCompareF64 (iteration = " << i << ")\n";
        cout << setw(11) << 'a' << ':' << a.ToStringF64() << '\n';
        cout << setw(11) << 'b' << ':' << b.ToStringF64() << '\n';
        cout << '\n';

        for (int j = 0; j < 8; j++)
            cout << setw(11) << c_CmpStr[j] << ':' << c[j].ToStringX64() << '\n';
    }
}

int main()
{
    AvxPackedCompareF32();
    AvxPackedCompareF64();
    return 0;
}

;-------------------------------------------------
;               Ch06_02.asm
;-------------------------------------------------

        include <cmpequ.asmh>

; extern "C" void AvxPackedCompareF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

        .code
AvxPackedCompareF32_ proc
        vmovaps xmm0,[rcx]                  ;xmm0 = a
        vmovaps xmm1,[rdx]                  ;xmm1 = b

; Perform packed EQUAL compare
        vcmpps xmm2,xmm0,xmm1,CMP_EQ
        vmovdqa xmmword ptr [r8],xmm2

; Perform packed NOT EQUAL compare
        vcmpps xmm2,xmm0,xmm1,CMP_NEQ
        vmovdqa xmmword ptr [r8+16],xmm2

; Perform packed LESS THAN compare
        vcmpps xmm2,xmm0,xmm1,CMP_LT
        vmovdqa xmmword ptr [r8+32],xmm2

; Perform packed LESS THAN OR EQUAL compare
        vcmpps xmm2,xmm0,xmm1,CMP_LE
        vmovdqa xmmword ptr [r8+48],xmm2
```

```
 ; Perform packed GREATER THAN compare
        vcmpps xmm2,xmm0,xmm1,CMP_GT
        vmovdqa xmmword ptr [r8+64],xmm2

; Perform packed GREATER THAN OR EQUAL compare
        vcmpps xmm2,xmm0,xmm1,CMP_GE
        vmovdqa xmmword ptr [r8+80],xmm2

; Perform packed ORDERED compare
        vcmpps xmm2,xmm0,xmm1,CMP_ORD
        vmovdqa xmmword ptr [r8+96],xmm2

; Perform packed UNORDERED compare
        vcmpps xmm2,xmm0,xmm1,CMP_UNORD
        vmovdqa xmmword ptr [r8+112],xmm2
        ret
AvxPackedCompareF32_ endp

; extern "C" void AvxPackedCompareF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

AvxPackedCompareF64_ proc
        vmovapd xmm0,[rcx]                   ;xmm0 = a
        vmovapd xmm1,[rdx]                   ;xmm1 = b

; Perform packed EQUAL compare
        vcmppd xmm2,xmm0,xmm1,CMP_EQ
        vmovdqa xmmword ptr [r8],xmm2

; Perform packed NOT EQUAL compare
        vcmppd xmm2,xmm0,xmm1,CMP_NEQ
        vmovdqa xmmword ptr [r8+16],xmm2

; Perform packed LESS THAN compare
        vcmppd xmm2,xmm0,xmm1,CMP_LT
        vmovdqa xmmword ptr [r8+32],xmm2

; Perform packed LESS THAN OR EQUAL compare
        vcmppd xmm2,xmm0,xmm1,CMP_LE
        vmovdqa xmmword ptr [r8+48],xmm2

 ; Perform packed GREATER THAN compare
        vcmppd xmm2,xmm0,xmm1,CMP_GT
        vmovdqa xmmword ptr [r8+64],xmm2

; Perform packed GREATER THAN OR EQUAL compare
        vcmppd xmm2,xmm0,xmm1,CMP_GE
        vmovdqa xmmword ptr [r8+80],xmm2

; Perform packed ORDERED compare
        vcmppd xmm2,xmm0,xmm1,CMP_ORD
        vmovdqa xmmword ptr [r8+96],xmm2
```

```
; Perform packed UNORDERED compare
        vcmppd xmm2,xmm0,xmm1,CMP_UNORD
        vmovdqa xmmword ptr [r8+112],xmm2
        ret
AvxPackedCompareF64_ endp
        end
```

Figure 6-2 illustrates execution of the `vcmpps xmm2,xmm0,xmm1,0` and `vcmppd xmm2,xmm0,xmm1,1` instructions. In these examples, the compare predicate operands `0` and `1` test for equality and less than, respectively.

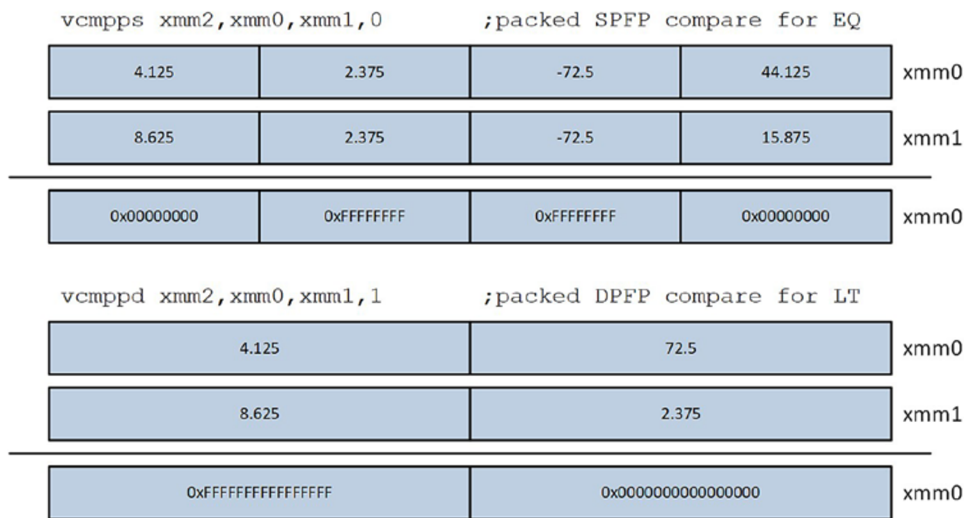



***Figure 6-2.*** *Execution of the `vcmpps` and `vcmppd` instructions*

The C++ function `AvxPackedCompareF32` begins by initializing a couple of `XmmVal` test variables. Similar to the example that you saw in the previous section, the `alignas(16)` specifier is used with each `XmmVal` variable to force proper alignment to a 16-byte boundary. The remaining code in this function invokes the assembly language `AvxPackedCompareF32_` and displays the results. Note that on the second iteration of the `for` loop, the constant `numeric_limits<float>::quiet_NaN()` is substituted for one of the values in `XmmVal a` to exemplify operation of the ordered and unordered compare predicates. An ordered compare is true when both operands are valid values. An unordered compare is true when one or both of the operands is a NaN or erroneously encoded. Substituting `numeric_limits<float>::quiet_NaN()` for one of the values in `XmmVal a` generates a true result for an unordered compare. The C++ code also includes the function `AvxPackedCompareF64`, which is the double-precision counterpart of `AvxPackedCompareF32`.

The x86-64 assembly language code begins with an `include <cmpequ.asmh>` statement. This file, which also was used in example `Ch05_05`, contains compare predicate equates that will be used in this source code example with the `vcmpp[d|s]` instructions. The assembly language function `AvxPackeCompareF32_` starts with two `vmovaps` instructions that load arguments `a` and `b` into registers XMM0 and XMM1, respectively. The ensuing `vcmpps xmm2,xmm0,xmm1,CMP_EQ` instruction compares packed operands `a` and `b` for equality and saves the packed result (four doubleword mask values) to register XMM2. The contents of register XMM2 are then saved to the result array `c` using a `vmovdqa xmmword ptr [r8],xmm2` instruction. The remaining code in `AvxPackedCompareF32_` performs additional compare operations using recognizable

compare predicates. The assembly language function AvxPackedCompareF64_ demonstrates how to use the vcmppd instruction to carry out packed double-precision floating-point compares. Here is the output for example Ch06_02:

```
Results for AvxPackedCompareF32 (iteration = 0)
         a:        2.000000       7.000000    |      -6.000000       3.000000
         b:        1.000000      12.000000    |      -6.000000       8.000000

        EQ:        00000000       00000000    |       FFFFFFFF       00000000
        NE:        FFFFFFFF       FFFFFFFF    |       00000000       FFFFFFFF
        LT:        00000000       FFFFFFFF    |       00000000       FFFFFFFF
        LE:        00000000       FFFFFFFF    |       FFFFFFFF       FFFFFFFF
        GT:        FFFFFFFF       00000000    |       00000000       00000000
        GE:        FFFFFFFF       00000000    |       FFFFFFFF       00000000
   ORDERED:        FFFFFFFF       FFFFFFFF    |       FFFFFFFF       FFFFFFFF
 UNORDERED:        00000000       00000000    |       00000000       00000000

Results for AvxPackedCompareF32 (iteration = 1)
         a:             nan       7.000000    |      -6.000000       3.000000
         b:        1.000000      12.000000    |      -6.000000       8.000000

        EQ:        00000000       00000000    |       FFFFFFFF       00000000
        NE:        FFFFFFFF       FFFFFFFF    |       00000000       FFFFFFFF
        LT:        00000000       FFFFFFFF    |       00000000       FFFFFFFF
        LE:        00000000       FFFFFFFF    |       FFFFFFFF       FFFFFFFF
        GT:        00000000       00000000    |       00000000       00000000
        GE:        00000000       00000000    |       FFFFFFFF       00000000
   ORDERED:        00000000       FFFFFFFF    |       FFFFFFFF       FFFFFFFF
 UNORDERED:        FFFFFFFF       00000000    |       00000000       00000000

Results for AvxPackedCompareF64 (iteration = 0)
         a:                  2.000000000000   |                 3.141592653590
         b:                  2.718281828459   |                -0.318309886184

        EQ:                0000000000000000   |               0000000000000000
        NE:                FFFFFFFFFFFFFFFF   |               FFFFFFFFFFFFFFFF
        LT:                FFFFFFFFFFFFFFFF   |               0000000000000000
        LE:                FFFFFFFFFFFFFFFF   |               0000000000000000
        GT:                0000000000000000   |               FFFFFFFFFFFFFFFF
        GE:                0000000000000000   |               FFFFFFFFFFFFFFFF
   ORDERED:                FFFFFFFFFFFFFFFF   |               FFFFFFFFFFFFFFFF
 UNORDERED:                0000000000000000   |               0000000000000000

Results for AvxPackedCompareF64 (iteration = 1)
         a:                             nan   |                 3.141592653590
         b:                  2.718281828459   |                 3.141592653590

        EQ:                0000000000000000   |               FFFFFFFFFFFFFFFF
        NE:                FFFFFFFFFFFFFFFF   |               0000000000000000
        LT:                0000000000000000   |               0000000000000000
        LE:                0000000000000000   |               FFFFFFFFFFFFFFFF
```

```
      GT:         0000000000000000   |    0000000000000000
      GE:         0000000000000000   |    FFFFFFFFFFFFFFFF
 ORDERED:         0000000000000000   |    FFFFFFFFFFFFFFFF
UNORDERED:        FFFFFFFFFFFFFFFF   |    0000000000000000
```

## Packed Floating-Point Conversions

The next source code example is named `Ch06_03`. This example shows packed signed doubleword integers to floating-point conversions and vice versa. It also illustrates conversions between packed single-precision and packed double-precision floating-point values. Listing 6-3 shows the source code for example `Ch06_03`.

***Listing 6-3.*** Example Ch06_03

```
//------------------------------------------------
//               Ch06_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>
#include "XmmVal.h"

using namespace std;

// The order of values in the following enum must match the jump table
// that's defined in Ch06_03_.asm.
enum CvtOp : unsigned int
{
    I32_F32, F32_I32, I32_F64, F64_I32, F32_F64, F64_F32,
};

extern "C" bool AvxPackedConvertFP_(const XmmVal& a, XmmVal& b, CvtOp cvt_op);

void AvxPackedConvertF32(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;

    a.m_I32[0] = 10;
    a.m_I32[1] = -500;
    a.m_I32[2] = 600;
    a.m_I32[3] = -1024;
    AvxPackedConvertFP_(a, b, CvtOp::I32_F32);
    cout << "\nResults for CvtOp::I32_F32\n";
    cout << "a: " << a.ToStringI32() << '\n';
    cout << "b: " << b.ToStringF32() << '\n';
```

```
    a.m_F32[0] = 1.0f / 3.0f;
    a.m_F32[1] = 2.0f / 3.0f;
    a.m_F32[2] = -a.m_F32[0] * 2.0f;
    a.m_F32[3] = -a.m_F32[1] * 2.0f;
    AvxPackedConvertFP_(a, b, CvtOp::F32_I32);
    cout << "\nResults for CvtOp::F32_I32\n";
    cout << "a: " << a.ToStringF32() << '\n';
    cout << "b: " << b.ToStringI32() << '\n';

    // F32_F64 converts the two low-order SPFP values of 'a'
    a.m_F32[0] = 1.0f / 7.0f;
    a.m_F32[1] = 2.0f / 9.0f;
    a.m_F32[2] = 0;
    a.m_F32[3] = 0;
    AvxPackedConvertFP_(a, b, CvtOp::F32_F64);
    cout << "\nResults for CvtOp::F32_F64\n";
    cout << "a: " << a.ToStringF32() << '\n';
    cout << "b: " << b.ToStringF64() << '\n';
}

void AvxPackedConvertF64(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;

    // I32_F64 converts the two low-order doubleword integers of 'a'
    a.m_I32[0] = 10;
    a.m_I32[1] = -20;
    a.m_I32[2] = 0;
    a.m_I32[3] = 0;
    AvxPackedConvertFP_(a, b, CvtOp::I32_F64);
    cout << "\nResults for CvtOp::I32_F64\n";
    cout << "a: " << a.ToStringI32() << '\n';
    cout << "b: " << b.ToStringF64() << '\n';

    // F64_I32 sets the two high-order doublewords of 'b' to zero
    a.m_F64[0] = M_PI;
    a.m_F64[1] = M_E;
    AvxPackedConvertFP_(a, b, CvtOp::F64_I32);
    cout << "\nResults for CvtOp::F64_I32\n";
    cout << "a: " << a.ToStringF64() << '\n';
    cout << "b: " << b.ToStringI32() << '\n';

    // F64_F32 sets the two high-order SPFP values of 'b' to zero
    a.m_F64[0] = M_SQRT2;
    a.m_F64[1] = M_SQRT1_2;
    AvxPackedConvertFP_(a, b, CvtOp::F64_F32);
    cout << "\nResults for CvtOp::F64_F32\n";
    cout << "a: " << a.ToStringF64() << '\n';
    cout << "b: " << b.ToStringF32() << '\n';
}
```

```
int main()
{
    AvxPackedConvertF32();
    AvxPackedConvertF64();
    return 0;
}

;-------------------------------------------------
;               Ch06_03.asm
;-------------------------------------------------

; extern "C" bool AvxPackedConvertFP_(const XmmVal& a, XmmVal& b, CvtOp cvt_op);
;
; Note:         This function requires linker option /LARGEADDRESSAWARE:NO
;               to be explicitly set.

        .code
AvxPackedConvertFP_ proc

; Make sure cvt_op is valid
        mov r9d,r8d                         ;r9 = cvt_op (zero extended)
        cmp r9,CvtOpTableCount              ;is cvt_op valid?
        jae InvalidCvtOp                    ;jmp if cvt_op is invalid

        mov eax,1                           ;set valid cvt_op return code
        jmp [CvtOpTable+r9*8]               ;jump to specified conversion

; Convert packed signed doubleword integers to packed SPFP values
I32_F32:
        vmovdqa xmm0,xmmword ptr [rcx]
        vcvtdq2ps xmm1,xmm0
        vmovaps xmmword ptr [rdx],xmm1
        ret

; Convert packed SPFP values to packed signed doubleword integers
F32_I32:
        vmovaps xmm0,xmmword ptr [rcx]
        vcvtps2dq xmm1,xmm0
        vmovdqa xmmword ptr [rdx],xmm1
        ret

; Convert packed signed doubleword integers to packed DPFP values
I32_F64:
        vmovdqa xmm0,xmmword ptr [rcx]
        vcvtdq2pd xmm1,xmm0
        vmovapd xmmword ptr [rdx],xmm1
        ret

; Convert packed DPFP values to packed signed doubleword integers
F64_I32:
        vmovapd xmm0,xmmword ptr [rcx]
```

```
        vcvtpd2dq xmm1,xmm0
        vmovdqa xmmword ptr [rdx],xmm1
        ret

; Convert packed SPFP to packed DPFP
F32_F64:
        vmovaps xmm0,xmmword ptr [rcx]
        vcvtps2pd xmm1,xmm0
        vmovapd xmmword ptr [rdx],xmm1
        ret

; Convert packed DPFP to packed SPFP
F64_F32:
        vmovapd xmm0,xmmword ptr [rcx]
        vcvtpd2ps xmm1,xmm0
        vmovaps xmmword ptr [rdx],xmm1
        ret

InvalidCvtOp:
        xor eax,eax                         ;set invalid cvt_op return code
        ret

; The order of values in the following table must match the enum CvtOp
; that's defined in Ch06_03.cpp.

            align 8
CvtOpTable  qword I32_F32, F32_I32
            qword I32_F64, F64_I32
            qword F32_F64, F64_F32
CvtOpTableCount equ ($ - CvtOpTable) / size qword

AvxPackedConvertFP_ endp
        end
```

The C++ code begins with an enum named CvtOP that defines the conversion operations supported by the assembly language function AvxPackedConvertFP_. The actual enumerator values in CvtOp are critical since the assembly language code uses them as indices into a jump table. The function that follows CvtOp, AvxPackedConvertF32, exercises some test cases using packed single-precision floating-point operands. Similarly, the function AvxPackedConvertF64 contains test cases for packed double-precision floating-point operands. As in the previous examples of this chapter, all XmmVal variable declarations in these functions use the alignas(16) specifier to ensure proper alignment.

Toward the bottom of the assembly language code in Listing 6-3 is the previously mentioned jump table. CvtOpTable contains a list of labels that are defined in the function AvxPackedConvertFP_. The target of each label is a short code block that performs a specific conversion. The equate CvtOpTableCount defines the number of items in the jump table and is used to validate the argument value cvt_op. The align 8 directive instructs the assembler to align CvtOpTable on a quadword boundary in order to avoid unaligned memory accesses when referencing elements in the table. Note that CvtOpTable is defined *inside* the assembly language function AvxPackedConvertFP_ (i.e., between the proc and endp directives), which means that storage for the table is allocated in a .code section. Clearly, the jump table does not contain any intentional executable instructions, and this is why the table is positioned after the ret instruction. This also means that the jump table is read-only; the processor will generate an exception on any write attempt to the table.

The assembly language function AvxPackedConvertFP_ begins its execution by validating the argument value cvt_op. The ensuing jmp [CvtOpTable+r9*8] instruction transfers control to a code block that performs the actual packed data conversion. During execution of this instruction, the processor loads register RIP with the contents of memory that's specified by [CvxOpTable+r9*8]. In the current example, register R9 contains cvt_op and this value is used as an index into CvtOpTable.

The conversion code blocks in AvxPackedConvertFP_ use the aligned move instructions vmovaps, vmovapd, and vmovdqa to transfer packed operands to and from memory. Specific AVX conversion instructions carry out the requested operations. For example, the vcvtps2dq and vcvtdq2ps instructions perform conversions between packed single-precision floating-point and signed doubleword integer values and vice versa. When used with 128-bit wide operands, these instructions convert four values simultaneously. The counterpart double-precision instructions, vcvtpd2dq and vcvtdq2pd, are slightly different in that only two values are converted due to the element size differences (32 and 64 bits). The vcvtps2pd and vcvtpd2ps instructions perform their conversions in a similar manner. Note that the vcvtpd2dq and vcvtpd2ps instructions set the high-order 64 bits of the destination operand to zero. All of the AVX packed conversion instructions use the rounding mode that's specified by the rounding control field MXCSR.RC, as described in Chapter 4. The default rounding mode for Visual C++ is round to nearest. Here is the output for example Ch06_03:

```
Results for CvtOp::I32_F32
a:            10            -500   |           600           -1024
b:     10.000000     -500.000000   |    600.000000    -1024.000000

Results for CvtOp::F32_I32
a:      0.333333        0.666667   |     -0.666667       -1.333333
b:             0               1   |            -1              -1

Results for CvtOp::F32_F64
a:      0.142857        0.222222   |      0.000000        0.000000
b:                0.142857149243   |                0.222222223878

Results for CvtOp::I32_F64
a:            10             -20   |             0               0
b:               10.000000000000   |              -20.000000000000

Results for CvtOp::F64_I32
a:                3.141592653590   |                2.718281828459
b:             3               3   |             0               0

Results for CvtOp::F64_F32
a:                1.414213562373   |                0.707106781187
b:      1.414214        0.707107   |      0.000000        0.000000
```

# Packed Floating-Point Arrays

The computational resources of AVX are often employed to accelerate calculations using arrays of single-precision or double-precision floating-point values. In this section, you learn how to use packed arithmetic to process multiple elements of a floating-point array simultaneously. You also see examples of additional AVX instructions and learn how to perform runtime alignment checks of operands in memory.

## Packed Floating-Point Square Roots

Listing 6-4 shows the code for example Ch06_04, which illustrates how to perform a simple packed arithmetic calculation using a single-precision floating-point array. It also explains how to perform a runtime check of an array's address to ensure that its properly aligned in memory.

***Listing 6-4.*** Example Ch06_04

```
//------------------------------------------------
//               Ch06_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>

using namespace std;

extern "C" bool AvxCalcSqrts_(float* y, const float* x, size_t n);

void Init(float* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 2000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (float)ui_dist(rng);
}

bool AvxCalcSqrtsCpp(float* y, const float* x, size_t n)
{
    const size_t alignment = 16;

    if (n == 0)
        return false;

    if (((uintptr_t)x % alignment) != 0)
        return false;

    if (((uintptr_t)y % alignment) != 0)
        return false;

    for (size_t i = 0; i < n; i++)
        y[i] = sqrt(x[i]);

    return true;
}
```

```
int main()
{
    const size_t n = 19;
    alignas(16) float x[n];
    alignas(16) float y1[n];
    alignas(16) float y2[n];

    Init(x, n, 53);

    bool rc1 = AvxCalcSqrtsCpp(y1, x, n);
    bool rc2 = AvxCalcSqrts_(y2, x, n);

    cout << fixed << setprecision(4);
    cout << "\nResults for AvxCalcSqrts\n";

    if (!rc1 || !rc2)
        cout << "Invalid return code\n";
    else
    {
        const char* sp = "   ";

        for (size_t i = 0; i < n; i++)
        {
            cout << "i:  " << setw(2) << i << sp;
            cout << "x:  " << setw(9) << x[i] << sp;
            cout << "y1: " << setw(9) << y1[i] << sp;
            cout << "y2: " << setw(9) << y2[i] << '\n';
        }
    }
}

;-------------------------------------------------
;               Ch06_04.asm
;-------------------------------------------------

; extern "C" bool AvxCalcSqrts_(float* y, const float* x, size_t n);

        .code
AvxCalcSqrts_ proc
        xor eax,eax                         ;set error return code (also array offset)

        test r8,r8
        jz Done                             ;jump if n is zero

        test rcx,0fh
        jnz Done                            ;jump if 'y' is not aligned

        test rdx,0fh
        jnz Done                            ;jump if 'x' is not aligned
```

```
; Calculate packed square roots
        cmp r8,4
        jb FinalVals                    ;jump if n < 4
@@:     vsqrtps xmm0,xmmword ptr [rdx+rax]  ;calculate 4 square roots x[i+3:i]
        vmovaps xmmword ptr [rcx+rax],xmm0  ;save results to y[i+3:i]

        add rax,16                      ;update offset to next set of values
        sub r8,4
        cmp r8,4                        ;are there 4 or more elements remaining?
        jae @B                          ;jump if yes

; Calculate square roots of final 1 - 3 values, note switch to scalar instructions
FinalVals:
        test r8,r8                      ;more elements to process?
        jz SetRC                        ;jump if no more elements

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]   ;calculate sqrt(x[i])
        vmovss real4 ptr [rcx+rax],xmm0         ;save result to y[i]
        add rax,4
        dec r8
        jz SetRC

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]
        vmovss real4 ptr [rcx+rax],xmm0
        add rax,4
        dec r8
        jz SetRC

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]
        vmovss real4 ptr [rcx+rax],xmm0

SetRC:  mov eax,1                       ;set success return code

Done:   ret
AvxCalcSqrts_ endp
        end
```

The C++ code in Listing 6-4 includes a function named `AvxCalcSqrtsCpp`, which calculates `y[i] = sqrt(x[i])`. Before performing any of the required calculations, array size argument `n` is tested to make sure that's not equal to zero. The pointers `y` and `x` are also tested to ensure that the respective arrays are properly aligned to a 16-byte boundary. An array is aligned to a 16-byte boundary if its address is evenly divisible by 16. The function returns an error code if any of these checks fail.

Assembly language function `AvxCalcSqrts_` mimics the functionality of its C++ counterpart. The `test r8,r8` and `jz Done` instructions ensure that the number of array elements `n` is greater than zero. The ensuing `test rcx,0fh` instruction checks array `y` for alignment to a 16-byte boundary. Recall that the `test` instruction performs a bitwise AND of its two operands and sets the status flags in RFLAGS according to the result (the actual result of the bitwise AND is discarded). If the `test rcx,0fh` instruction yields a non-zero value, array `y` is not aligned on a 16-byte boundary, and the function exits without performing any calculations. A similar test is used to ensure that array `x` is properly aligned.

The processing loop uses a `vsqrtps` instruction to calculate the required square roots. When used with 128-bit wide operands, this instruction calculates four single-precision floating-point square roots

simultaneously. Using 128-bit wide operands means that the processing loop cannot execute a `vsqrtps` instruction if there are fewer than four element values remaining to be processed. Before performing any calculations using `vsqrtps`, R8 is checked to make sure that it's greater than or equal to four. If R8 is less than four, the processing loop is skipped. The processing loop employs a `vsqrtps xmm0,xmmword ptr [rdx+rax]` instruction to calculate square roots of the four single-precision floating-point values located at the memory address specified by the source operand. It then stores the calculated square roots in register XMM0. A `vmovaps xmmword ptr [rcx+rax],xmm0` instruction saves the four calculated square roots to `y`. Execution of the `vsqrtps` and `vmovaps` instructions continues until the number of elements remaining to be processed is less than four.

Following execution of the processing loop, the block of code starting at label `FinalVals` calculates the square roots for the final few values of array `x`. Note that the scalar AVX instructions `vsqrtss` and `vmovss` instructions perform these final (one, two, or three) calculations. Here is the output for source code example `Ch06_04`.

```
Results for AvxCalcSqrts

i:   0   x:  1354.0000   y1:   36.7967   y2:   36.7967
i:   1   x:   494.0000   y1:   22.2261   y2:   22.2261
i:   2   x:  1638.0000   y1:   40.4722   y2:   40.4722
i:   3   x:   278.0000   y1:   16.6733   y2:   16.6733
i:   4   x:  1004.0000   y1:   31.6860   y2:   31.6860
i:   5   x:   318.0000   y1:   17.8326   y2:   17.8326
i:   6   x:  1735.0000   y1:   41.6533   y2:   41.6533
i:   7   x:  1221.0000   y1:   34.9428   y2:   34.9428
i:   8   x:   544.0000   y1:   23.3238   y2:   23.3238
i:   9   x:  1568.0000   y1:   39.5980   y2:   39.5980
i:  10   x:  1633.0000   y1:   40.4104   y2:   40.4104
i:  11   x:  1577.0000   y1:   39.7115   y2:   39.7115
i:  12   x:  1659.0000   y1:   40.7308   y2:   40.7308
i:  13   x:  1565.0000   y1:   39.5601   y2:   39.5601
i:  14   x:    74.0000   y1:    8.6023   y2:    8.6023
i:  15   x:  1195.0000   y1:   34.5688   y2:   34.5688
i:  16   x:   406.0000   y1:   20.1494   y2:   20.1494
i:  17   x:   483.0000   y1:   21.9773   y2:   21.9773
i:  18   x:  1307.0000   y1:   36.1525   y2:   36.1525
```

The source code in Listing 6-4 can be easily adapted to process double-precision instead of single-precision floating-point values. In the C++ code, changing all `float` variables `double` is the only required modification. In the assembly language code, the `vsqrtpd` and `vmovapd` instructions must be used instead of `vsqrtps` and `vmovaps`. The counting variables in `AvxCalcSqrts_` must also be changed to process two double-precision instead of four single-precision floating-point values per iteration.

## Packed Floating-Point Array Min-Max

Listing 6-5 shows the source code for example Ch06_05. This example demonstrates how to compute the minimum and maximum value of a single-precision floating-point array using packed AVX instructions.

***Listing 6-5.*** Example Ch06_05

```
//------------------------------------------------
//               Ch06_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <limits>
#include "AlignedMem.h"

using namespace std;

extern "C" float g_MinValInit = numeric_limits<float>::max();
extern "C" float g_MaxValInit = -numeric_limits<float>::max();

extern "C" bool CalcArrayMinMaxF32_(float* min_val, float* max_val, const float* x, size_t n);

void Init(float* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 10000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (float)ui_dist(rng);
}

bool CalcArrayMinMaxF32Cpp(float* min_val, float* max_val, const float* x, size_t n)
{
    // Make sure x is properly aligned
    if (!AlignedMem::IsAligned(x, 16))
        return false;

    // Find array minimum and maximum values
    float min_val_temp = g_MinValInit;
    float max_val_temp = g_MaxValInit;

    if (n > 0)
    {
        for (size_t i = 0; i < n; i++)
        {
            if (x[i] < min_val_temp)
                min_val_temp = x[i];
```

```
            if (x[i] > max_val_temp)
                max_val_temp = x[i];
        }
    }

    *min_val = min_val_temp;
    *max_val = max_val_temp;
    return true;
}

int main()
{
    const size_t n = 31;
    alignas(16) float x[n];

    Init(x, n, 73);

    float min_val1, max_val1;
    float min_val2, max_val2;

    CalcArrayMinMaxF32Cpp(&min_val1, &max_val1, x, n);
    CalcArrayMinMaxF32_(&min_val2, &max_val2, x, n);

    cout << fixed << setprecision(1);
    cout << "-------------- Array x --------------\n";

    for (size_t i = 0; i < n; i++)
    {
        cout << "x[" << setw(2) << i << "]: " << setw(9) << x[i];

        if (i & 1)
            cout << '\n';
        else
            cout << "    ";
    }

    cout << '\n';

    cout << "\nResults for CalcArrayMinMaxF32Cpp\n";
    cout << "  min_val = " << setw(9) << min_val1 << ", ";
    cout << "  max_val = " << setw(9) << max_val1 << '\n';

    cout << "\nResults for CalcArrayMinMaxF32_\n";
    cout << "  min_val = " << setw(9) << min_val2 << ", ";
    cout << "  max_val = " << setw(9) << max_val2 << '\n';

    return 0;
}
```

```
;-------------------------------------------------
;               Ch06_05.asm
;-------------------------------------------------

        extern g_MinValInit:real4
        extern g_MaxValInit:real4

; extern "C" bool CalcArrayMinMaxF32_(float* min_val, float* max_val, const float* x, size_t n)

        .code
CalcArrayMinMaxF32_ proc
; Validate arguments
        xor eax,eax                         ;set error return code

        test r8,0fh                         ;is x aligned to 16-byte boundary?
        jnz Done                            ;jump if no

        vbroadcastss xmm4,real4 ptr [g_MinValInit]     ;xmm4 = min values
        vbroadcastss xmm5,real4 ptr [g_MaxValInit]     ;xmm5 = max values

        cmp r9,4
        jb FinalVals                        ;jump if n < 4

; Main processing loop
@@:     vmovaps xmm0,xmmword ptr [r8]       ;load next set of array values
        vminps xmm4,xmm4,xmm0               ;update packed min values
        vmaxps xmm5,xmm5,xmm0               ;update packed max values

        add r8,16
        sub r9,4
        cmp r9,4
        jae @B

; Process the final 1 - 3 values of the input array
FinalVals:
        test r9,r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8]     ;update packed min values
        vmaxss xmm5,xmm5,real4 ptr [r8]     ;update packed max values
        dec r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8+4]
        vmaxss xmm5,xmm5,real4 ptr [r8+4]
        dec r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8+8]
        vmaxss xmm5,xmm5,real4 ptr [r8+8]
```

```
; Calculate and save final min & max values
SaveResults:
        vshufps xmm0,xmm4,xmm4,00001110b    ;xmm0[63:0] = xmm4[128:64]
        vminps xmm1,xmm0,xmm4               ;xmm1[63:0] contains final 2 values
        vshufps xmm2,xmm1,xmm1,00000001b    ;xmm2[31:0] = xmm1[63:32]
        vminps xmm3,xmm2,xmm1               ;xmm3[31:0] contains final value
        vmovss real4 ptr [rcx],xmm3         ;save array min value

        vshufps xmm0,xmm5,xmm5,00001110b
        vmaxps xmm1,xmm0,xmm5
        vshufps xmm2,xmm1,xmm1,00000001b
        vmaxps xmm3,xmm2,xmm1
        vmovss real4 ptr [rdx],xmm3         ;save array max value

        mov eax,1                           ;set success return code
Done:   ret
CalcArrayMinMaxF32_ endp
        end
```

The structure of the C++ source code that's shown in Listing 6-5 is similar to the previous array example. The function `CalcArrayMinMaxF32Cpp` uses a simple `for` loop to determine the array's minimum and maximum values. Prior to the `for` loop, the template function `AlignedMem::IsAligned` verifies that source array `x` is properly aligned. You'll learn more about class `AlignedMem` in Chapter 7. The initial minimum and maximum values are obtained from the global variables `g_MinValInit` and `g_MaxValInit`, which were initialized using the C++ template constant `numeric_limits<float>::max()`. Global variables are employed here to ensure that the functions `CalcArrayMinMaxF32Cpp` and `CalcArrayMinMaxF32_` use the same initial values.

Upon entry to the assembly language function `CalcArrayMinMaxF32_`, the array `x` is tested for proper alignment. If array `x` is properly aligned, a `vbroadcastss xmm4,real4 ptr [g_MinValInit]` instruction initializes all four single-precision floating-point elements in register XMM4 with the value `g_MinValInit`. The subsequent `vbroadcastss xmm5,real4 ptr [g_MaxValInit]` instruction broadcasts `g_MaxValInit` to all four element positions in register XMM5.

Like the previous example, the processing loop in `CalcArrayMinMaxF32_` examines four array elements during each iteration. The `vminps xmm4,xmm4,xmm0` and `vmaxps xmm5,xmm5,xmm0` instructions maintain intermediate packed minimum and maximum values in registers XMM4 and XMM5, respectively. The processing loop continues until there fewer than four elements remaining. The final elements in the array are tested using the scalar instructions `vminss` and `vmaxss`.

Subsequent to the execution of the `vmaxss` instruction that's immediately above the label `SaveResults`, register XMM4 contains four single-precision floating-point values, and one of these values is the minimum for array `x`. A series of `vshufps` (Packed Interleave Shuffle Single-Precision Floating-Point Values) and `vminps` instructions is then used to determine the final minimum value. The `vshufps xmm0,xmm4,xmm4,00001110b` instruction copies the two high-order floating-point elements in register XMM4 to the low-order element positions in XMM0 (i.e., XMM0[63:0] = XMM4[127:64]). This instruction uses the bit values of its immediate operand as indices for selecting elements to copy.

The immediate operand that's used by the `vshufps` instruction warrants further explanation. In the current example, bits 1:0 (`10b`) of the immediate operand instruct the processor to copy single-precision floating-point element #2 (XMM4[95:64]) from the first source operand to element position #0 (XMM0[31:0]) of the destination operand. Bits 3:2 (`11b`) of the immediate operand also instruct the processor to copy element #3 (XMM4[127:64]) of the first source operand to element position #1 (XMM0[63:32]) of the destination operand. Bits 7:6 and 5:4 of the immediate operand can be used to copy elements from the second source operand to element positions #2 (XMM0[95:64]) and #3 (XMM0[127:96]) of the destination

operand, but they're not needed in the current example. The vshufps instruction is followed by a vminps xmm1,xmm0,xmm4 instruction that yields the final two minimum values in XMM1[63:32] and XMM1[31:0]. Another sequence of vshufps and vminps instructions is then used to extract the final minimum value. Figure 6-3 illustrates this reduction process in greater detail.

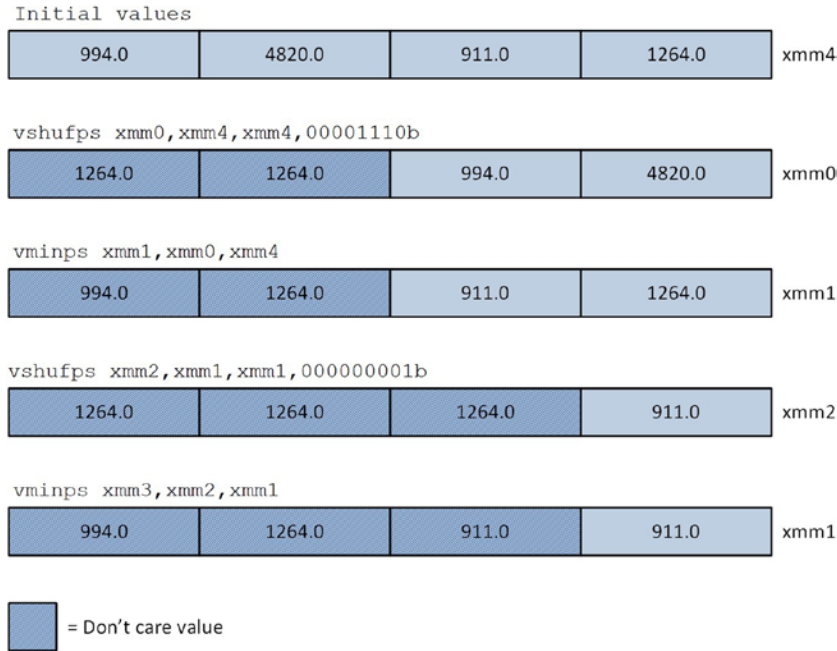



***Figure 6-3.*** *Packed minimum reduction using vshufps and vminps instructions*

Following calculation of the array's minimum value, an analogous series of vshufps and vmaxps instructions determines the maximum value using the same reduction technique. Here are the results for example Ch06_05:

```
-------------- Array x --------------

x[ 0]:     2183.0    x[ 1]:     4547.0
x[ 2]:     9279.0    x[ 3]:     7291.0
x[ 4]:     5105.0    x[ 5]:     6505.0
x[ 6]:     4820.0    x[ 7]:      994.0
x[ 8]:     1559.0    x[ 9]:     3867.0
x[10]:     7272.0    x[11]:     9698.0
x[12]:     6181.0    x[13]:     4742.0
x[14]:     7279.0    x[15]:     1224.0
x[16]:     4840.0    x[17]:     8453.0
x[18]:     6876.0    x[19]:     1786.0
x[20]:     4022.0    x[21]:      911.0
x[22]:     6676.0    x[23]:     2979.0
```

```
x[24]:    4431.0    x[25]:    6133.0
x[26]:    7093.0    x[27]:    9892.0
x[28]:    9622.0    x[29]:    5058.0
x[30]:    1264.0

Results for CalcArrayMinMaxF32Cpp
  min_val =      911.0,   max_val =    9892.0

Results for CalcArrayMinMaxF32_
  min_val =      911.0,   max_val =    9892.0
```

## Packed Floating-Point Least Squares

Source code example Ch06_06 details the calculation of a least squares regression line using packed double-precision floating-point arithmetic. Listing 6-6 shows the C++ and x86 assembly language source code for example Ch06_06.

***Listing 6-6.*** Example Ch06_06

```
//------------------------------------------------
//               Ch06_06.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstddef>
#include "AlignedMem.h"

using namespace std;

extern "C" double LsEpsilon = 1.0e-12;
extern "C" bool AvxCalcLeastSquares_(const double* x, const double* y, int n, double* m,
double* b);

bool AvxCalcLeastSquaresCpp(const double* x, const double* y, int n, double* m, double* b)
{
    if (n < 2)
        return false;
    if (!AlignedMem::IsAligned(x, 16) || !AlignedMem::IsAligned(y, 16))
        return false;

    double sum_x = 0, sum_y = 0.0, sum_xx = 0, sum_xy = 0.0;

    for (int i = 0; i < n; i++)
    {
        sum_x += x[i];
        sum_xx += x[i] * x[i];
        sum_xy += x[i] * y[i];
        sum_y += y[i];
    }
```

```
    double denom = n * sum_xx - sum_x * sum_x;

    if (fabs(denom) >= LsEpsilon)
    {
        *m = (n * sum_xy - sum_x * sum_y) / denom;
        *b = (sum_xx * sum_y - sum_x * sum_xy) / denom;
        return true;
    }
    else
    {
        *m = *b = 0.0;
        return false;
    }
}

int main()
{
    const int n = 11;
    alignas(16) double x[n] = {10, 13, 17, 19, 23, 7, 35, 51, 89, 92, 99};
    alignas(16) double y[n] = {1.2, 1.1, 1.8, 2.2, 1.9, 0.5, 3.1, 5.5, 8.4, 9.7, 10.4};
    double m1 = 0, m2 = 0;
    double b1 = 0, b2 = 0;

    bool rc1 = AvxCalcLeastSquaresCpp(x, y, n, &m1, &b1);
    bool rc2 = AvxCalcLeastSquares_(x, y, n, &m2, &b2);

    cout << fixed << setprecision(8);

    cout << "\nResults from AvxCalcLeastSquaresCpp\n";
    cout << "  rc:         " << setw(12) << boolalpha << rc1 << '\n';
    cout << "  slope:      " << setw(12) << m1 << '\n';
    cout << "  intercept:: " << setw(12) << b1 << '\n';

    cout << "\nResults from AvxCalcLeastSquares_\n";
    cout << "  rc:         " << setw(12) << boolalpha << rc2 << '\n';
    cout << "  slope:      " << setw(12) << m2 << '\n';
    cout << "  intercept:: " << setw(12) << b2 << '\n';

    return 0;
}

;-------------------------------------------------
;               Ch06_06.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

        extern LsEpsilon:real8              ;global value defined in C++ file
```

```
; extern "C" bool AvxCalcLeastSquares_(const double* x, const double* y, int n, double* m,
double* b);
;
; Returns       0 = error (invalid n or improperly aligned array), 1 = success

        .const
        align 16
AbsMaskF64  qword 7fffffffffffffffh, 7fffffffffffffffh  ;mask for DPFP absolute value

        .code
AvxCalcLeastSquares_ proc frame
        _CreateFrame LS_,0,48,rbx
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Validate arguments
        xor eax,eax                         ;set error return code
        cmp r8d,2
        jl Done                             ;jump if n < 2
        test rcx,0fh
        jnz Done                            ;jump if x not aligned to 16-byte boundary
        test rdx,0fh
        jnz Done                            ;jump if y not aligned to 16-byte boundary

; Perform required initializations
        vcvtsi2sd xmm3,xmm3,r8d             ;xmm3 = n
        mov eax,r8d
        and r8d,0fffffffeh                  ;rd8 = n / 2 * 2
        and eax,1                           ;eax = n % 2

        vxorpd xmm4,xmm4,xmm4               ;sum_x (both qwords)
        vxorpd xmm5,xmm5,xmm5               ;sum_y (both qwords)
        vxorpd xmm6,xmm6,xmm6               ;sum_xx (both qwords)
        vxorpd xmm7,xmm7,xmm7               ;sum_xy (both qwords)

        xor ebx,ebx                         ;rbx = array offset
        mov r10,[rbp+LS_OffsetStackArgs]    ;r10 = b

; Calculate sum variables. Note that two values are processed each iteration.
@@:     vmovapd xmm0,xmmword ptr [rcx+rbx]  ;load next two x values
        vmovapd xmm1,xmmword ptr [rdx+rbx]  ;load next two y values

        vaddpd xmm4,xmm4,xmm0               ;update sum_x
        vaddpd xmm5,xmm5,xmm1               ;update sum_y

        vmulpd xmm2,xmm0,xmm0               ;calc x * x
        vaddpd xmm6,xmm6,xmm2               ;update sum_xx

        vmulpd xmm2,xmm0,xmm1               ;calc x * y
        vaddpd xmm7,xmm7,xmm2               ;update sum_xy
```

```
        add rbx,16                          ;rbx = next offset
        sub r8d,2                           ;adjust counter
        jnz @B                              ;repeat until done

; Update sum variables with the final x, y values if 'n' is odd
        or eax,eax
        jz CalcFinalSums                    ;jump if n is even
        vmovsd xmm0,real8 ptr [rcx+rbx]     ;load final x
        vmovsd xmm1,real8 ptr [rdx+rbx]     ;load final y

        vaddsd xmm4,xmm4,xmm0               ;update sum_x
        vaddsd xmm5,xmm5,xmm1               ;update sum_y

        vmulsd xmm2,xmm0,xmm0               ;calc x * x
        vaddsd xmm6,xmm6,xmm2               ;update sum_xx

        vmulsd xmm2,xmm0,xmm1               ;calc x * y
        vaddsd xmm7,xmm7,xmm2               ;update sum_xy

; Calculate final sum_x, sum_y, sum_xx, sum_xy
CalcFinalSums:
        vhaddpd xmm4,xmm4,xmm4              ;xmm4[63:0] = final sum_x
        vhaddpd xmm5,xmm5,xmm5              ;xmm5[63:0] = final sum_y
        vhaddpd xmm6,xmm6,xmm6              ;xmm6[63:0] = final sum_xx
        vhaddpd xmm7,xmm7,xmm7              ;xmm7[63:0] = final sum_xy

; Compute denominator and make sure it's valid
; denom = n * sum_xx - sum_x * sum_x
        vmulsd xmm0,xmm3,xmm6               ;n * sum_xx
        vmulsd xmm1,xmm4,xmm4               ;sum_x * sum_x
        vsubsd xmm2,xmm0,xmm1               ;denom
        vandpd xmm8,xmm2,xmmword ptr [AbsMaskF64] ;fabs(denom)
        vcomisd xmm8,real8 ptr [LsEpsilon]
        jb BadDen                           ;jump if denom < fabs(denom)

; Compute and save slope
; slope = (n * sum_xy - sum_x * sum_y) / denom
        vmulsd xmm0,xmm3,xmm7               ;n * sum_xy
        vmulsd xmm1,xmm4,xmm5               ;sum_x * sum_y
        vsubsd xmm2,xmm0,xmm1               ;slope numerator
        vdivsd xmm3,xmm2,xmm8               ;final slope
        vmovsd real8 ptr [r9],xmm3          ;save slope

; Compute and save intercept
; intercept = (sum_xx * sum_y - sum_x * sum_xy) / denom
        vmulsd xmm0,xmm6,xmm5               ;sum_xx * sum_y
        vmulsd xmm1,xmm4,xmm7               ;sum_x * sum_xy
        vsubsd xmm2,xmm0,xmm1               ;intercept numerator
        vdivsd xmm3,xmm2,xmm8               ;final intercept
        vmovsd real8 ptr [r10],xmm3         ;save intercept
```

```
        mov eax,1                           ;success return code
        jmp Done

; Bad denominator detected, set m and b to 0.0
BadDen: vxorpd xmm0,xmm0,xmm0
        vmovsd real8 ptr [r9],xmm0          ;*m = 0.0
        vmovsd real8 ptr [r10],xmm0         ;*b = 0.0
        xor eax,eax                         ;set error code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame rbx
        ret
AvxCalcLeastSquares_ endp
        end
```

Simple linear regression is a statistical technique that models a linear relationship between two variables. One popular method of simple linear regression is called *least squares fitting,* which uses a set of sample data points to determine a best fit or optimal curve between two variables. When used with a simple linear regression model, the curve is a straight line whose equation is $y = mx + b$. In this equation, $x$ denotes the independent variable, $y$ represents the dependent (or measured) variable, $m$ is the line's slope, and $b$ is the line's y-axis intercept point. The slope and intercept point of a least squares line are determined using a series of computations that minimize the sum of the squared deviations between the line and sample data points. Following calculation of its slope and intercept point, a least squares line is frequently used to predict an unknown $y$ value using a known $x$ value. If you're interested in learning more about the theory of simple linear regression and least squares fitting, consult the references listed in Appendix A.

In sample program Ch06_06, the following equations are used to calculate the least squares slope and intercept point:

$$m = \frac{n\sum_i x_i y_i - \sum_i x_i \sum_i y_i}{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2}$$

$$b = \frac{\sum_i x_i^2 \sum_i y_i - \sum_i x_i \sum_i x_i y_i}{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2}$$

At first glance, the slope and intercept equations may appear a little daunting. However, upon closer examination, a couple of simplifications become apparent. First, the slope and intercept point denominators are the same, which means that this value only needs to be computed once. Second, it is only necessary to calculate four simple summation quantities (or sum variables), as shown in the following equations:

$$sum_x = \sum_i x_i$$

$$sum_y = \sum_i y_i$$

$$sum_xy = \sum_i x_i y_i$$

$$sum_xx = \sum_i x_i^2$$

Subsequent to the calculation of the sum variables, the least-squares slope and intercept point are easily derived using straightforward multiplication, subtraction, and division.

The C++ source code in Listing 6-6 includes a function named AvxCalcLeastSquaresCpp that calculates a least-squares slope and intercept point for comparison purposes. AvxCalcLeastSquaresCpp uses AlignedMem::IsAligned() to validate proper alignment of the two data arrays. The C++ class AlignedMem (source code not shown but included in the download package) contains a few simple member functions that perform aligned memory management and validation. These functions have been incorporated into a C++ class to facilitate code reuse in this example and subsequent chapters. The C++ function main defines a couple of test arrays named x and y using the C++ specifier alignas(16), which instructs the compiler to align each of these arrays on a 16-byte boundary. The remainder of main contains code that exercises both the C++ and x86 assembly language implementations of the least squares algorithm and streams the results to cout.

The x86-64 assembly language code for function AvxCalcLeastSquares_ begins with saves of non-volatile registers RBX, XMM6, XMM7, and XMM8 using the macros _CreateFrame and _SaveXmmRegs. Argument value n is then validated for size, and the array pointers x and y are tested for proper alignment. Following validation of the function arguments, a series of initializations is performed. The vcvtsi2sd xmm3,xmm3,r8d instruction converts the value n to double-precision floating-point for later use. The value n in R8D is then rounded down to the nearest even number using an and r8d,0fffffffeh instruction and EAX is set to zero or one depending on whether the original value of n is even or odd. These adjustments are carried out to ensure proper processing of arrays x and y using packed arithmetic.

Recall from the discussions earlier in this section that in order to compute the slope and intercept point of a least squares regression line, you need to calculate four intermediate sum values: sum_x, sum_y, sum_xx, and sum_xy. The summation loop that calculates these values in AvxCalcLeastSquares_ uses packed double-precision floating-point arithmetic. This means that AvxCalcLeastSquares_ can process two elements from arrays x and y during each loop iteration, which halves the number of required iterations. The sum values for array elements with even-numbered indices are computed using the low-order quadwords of XMM4-XMM7, while the high-order quadwords are used to calculate the sum values for array elements with odd-numbered indices.

Prior to entering the summation loop, each sum value register is initialized to zero using a vxorpd instruction. At the top of the summation loop, a vmovapd xmm0,xmmword ptr [rcx+rbx] instruction copies x[i] and x[i+1] into the low-order and high-order quadwords of XMM0, respectively. The next instruction, vmovapd xmm1,xmmword ptr [rdx+rbx], loads y[i] and y[i+1] into the low-order and high-order quadwords of XMM1. A series of vaddpd and vmulpd instructions update the packed sum values that are maintained in XMM4 - XMM7. Array offset register RBX is then incremented by 16 (or the size of two double-precision floating-point values) and the count value in R8D is adjusted before the next summation loop iteration. Following completion of the summation loop, a check is made to determine if the original value of n was odd. If true, the final element of array x and array y must be added to the packed sum values. The AVX scalar instructions vaddsd and vmulsd carry out this operation.

Following computation of the packed sum values, a series of vhaddpd (Packed Double-FP Horizontal Add) instructions compute the final values of sum_x, sum_y, sum_xx, and sum_xy. Each vhaddpd DesOp, SrcOp1,SrcOp2 instruction computes DesOp[63:0] = SrcOp1[127:64] + SrcOp1[63:0] and DesOp[127:64] = SrcOp2[127:64] + SrcOp2[63:0] (see Figure 4-14). Subsequent to the execution of the vhaddpd instructions, the low-order quadwords of registers XMM4 - XMM7 contain the final sum values. The high-order quadwords of these registers also contain the final sum values, but this is a consequence of using the same register for both source operands. The value of denom is computed next and tested to make sure its absolute value is greater than or equal to LsEpsilon; an absolute value less than LsEpsilon is considered too close to zero to be valid. Note that a vandpd instruction is used to calculate fabs(denom). After validation

of denom, the slope and intercept values are calculated using straightforward scalar arithmetic. Here is the output for source code example Ch06_06:

```
Results from AvxCalcLeastSquaresCpp
  rc:                 true
  slope:        0.10324631
  intercept::  -0.10700632

Results from AvxCalcLeastSquares_
  rc:                 true
  slope:        0.10324631
  intercept::  -0.10700632
```

# Packed Floating-Point Matrices

Software applications such as computer graphics and computer-aided design programs often make extensive use of matrices. For example, three-dimensional (3D) computer graphics software typically employs matrices to perform common transformations such as translation, scaling, and rotation. When using homogeneous coordinates, each of these operations can be efficiently represented using a single 4 × 4 matrix. Multiple transformations can also be applied by merging a series of distinct transformation matrices into a single transformation matrix using matrix multiplication. This combined matrix is typically applied to an array of object vertices that defines a 3D model. It is important for 3D computer graphics software to carry out operations such as matrix multiplication and matrix-vector multiplication as quickly as possible since a 3D model may contain thousands or even millions of object vertices.

In this section, you learn how to perform matrix transposition and multiplication using 4 × 4 matrices and the AVX instruction set. You also learn more about assembly language macros, how to write macro code, and some simple techniques for benchmarking algorithm performance.

## Matrix Transposition

The transpose of a matrix is calculated by interchanging its rows and columns. More formally, if **A** is an $m \times n$ matrix, the transpose of **A** (denoted here by **B**) is an $n \times m$ matrix, where $b(i,j) = a(j,i)$. Figure 6-4 illustrates the transposition of a 4 × 4 matrix.

Matrix A

$$\mathbf{A} = \begin{bmatrix} 2 & 7 & 8 & 3 \\ 11 & 14 & 16 & 10 \\ 24 & 21 & 27 & 29 \\ 31 & 34 & 38 & 33 \end{bmatrix}$$

Transpose of Matrix A

$$\mathbf{B} = \begin{bmatrix} 2 & 11 & 24 & 31 \\ 7 & 14 & 21 & 34 \\ 8 & 16 & 27 & 38 \\ 3 & 10 & 29 & 33 \end{bmatrix}$$

***Figure 6-4.*** *Transposition of a 4 × 4 matrix*

Listing 6-7 shows the source code for example Ch06_07, which demonstrates how to transpose a 4 × 4 matrix of single-precision floating-point values.

***Listing 6-7.*** Example Ch06_07

```
//------------------------------------------------
//               Ch06_07.h
//------------------------------------------------

#pragma once

// Ch06_07_.asm
extern "C" void AvxMat4x4TransposeF32_(float* m_des, const float* m_src);

// Ch06_07_BM.cpp
extern void AvxMat4x4TransposeF32_BM(void);

//------------------------------------------------
//               Ch06_07.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "Ch06_07.h"
#include "Matrix.h"

using namespace std;

void AvxMat4x4TransposeF32(Matrix<float>& m_src)
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<float> m_des1(nr ,nc);
    Matrix<float> m_des2(nr ,nc);

    Matrix<float>::Transpose(m_des1, m_src);
    AvxMat4x4TransposeF32_(m_des2.Data(), m_src.Data());

    cout << fixed << setprecision(1);
    m_src.SetOstream(12, "  ");
    m_des1.SetOstream(12, "  ");
    m_des2.SetOstream(12, "  ");

    cout << "Results for AvxMat4x4TransposeF32\n";
    cout << "Matrix m_src \n" << m_src << '\n';
    cout << "Matrix m_des1\n" << m_des1 << '\n';
    cout << "Matrix m_des2\n" << m_des2 << '\n';

    if (m_des1 != m_des2)
        cout << "\nMatrix compare failed - AvxMat4x4TransposeF32\n";
}
```

```
int main()
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<float> m_src(nr ,nc);

    const float src_row0[] = {  2,  7,  8,  3 };
    const float src_row1[] = { 11, 14, 16, 10 };
    const float src_row2[] = { 24, 21, 27, 29 };
    const float src_row3[] = { 31, 34, 38, 33 };

    m_src.SetRow(0, src_row0);
    m_src.SetRow(1, src_row1);
    m_src.SetRow(2, src_row2);
    m_src.SetRow(3, src_row3);

    AvxMat4x4TransposeF32(m_src);
    AvxMat4x4TransposeF32_BM();
    return 0;
}

;-------------------------------------------------
;               Ch06_07.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; _Mat4x4TransposeF32 macro
;
; Description:  This macro transposes a 4x4 matrix of single-precision
;               floating-point values.
;
;  Input Matrix                    Output Matrix
;  ---------------------------------------------------
;  xmm0    a3 a2 a1 a0             xmm4    d0 c0 b0 a0
;  xmm1    b3 b2 b1 b0             xmm5    d1 c1 b1 a1
;  xmm2    c3 c2 c1 c0             xmm6    d2 c2 b2 a2
;  xmm3    d3 d2 d1 d0             xmm7    d3 c3 b3 a3

_Mat4x4TransposeF32 macro
        vunpcklps xmm6,xmm0,xmm1            ;xmm6 = b1 a1 b0 a0
        vunpckhps xmm0,xmm0,xmm1            ;xmm0 = b3 a3 b2 a2
        vunpcklps xmm7,xmm2,xmm3            ;xmm7 = d1 c1 d0 c0
        vunpckhps xmm1,xmm2,xmm3            ;xmm1 = d3 c3 d2 c2

        vmovlhps xmm4,xmm6,xmm7             ;xmm4 = d0 c0 b0 a0
        vmovhlps xmm5,xmm7,xmm6             ;xmm5 = d1 c1 b1 a1
        vmovlhps xmm6,xmm0,xmm1             ;xmm6 = d2 c2 b2 a2
        vmovhlps xmm7,xmm1,xmm0             ;xmm7 = d3 c3 b2 a3
        endm
```

```
; extern "C" void AvxMat4x4TransposeF32_(float* m_des, const float* m_src)

        .code
AvxMat4x4TransposeF32_ proc frame
        _CreateFrame MT_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Transpose matrix m_src1
        vmovaps xmm0,[rdx]                  ;xmm0 = m_src.row_0
        vmovaps xmm1,[rdx+16]               ;xmm1 = m_src.row_1
        vmovaps xmm2,[rdx+32]               ;xmm2 = m_src.row_2
        vmovaps xmm3,[rdx+48]               ;xmm3 = m_src.row_3

        _Mat4x4TransposeF32

        vmovaps [rcx],xmm4                  ;save m_des.row_0
        vmovaps [rcx+16],xmm5               ;save m_des.row_1
        vmovaps [rcx+32],xmm6               ;save m_des.row_2
        vmovaps [rcx+48],xmm7               ;save m_des.row_3

Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4TransposeF32_ endp
        end

//------------------------------------------------
//              Ch06_07_BM.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <string>
#include "Ch06_07.h"
#include "Matrix.h"
#include "BmThreadTimer.h"
#include "OS.h"

using namespace std;

extern void AvxMat4x4TransposeF32_BM(void)
{
    OS::SetThreadAffinityMask();
    cout << "\nRunning benchmark function AvxMat4x4TransposeF32_BM - please wait\n";

    const size_t num_rows = 4;
    const size_t num_cols = 4;
    Matrix<float> m_src(num_rows, num_cols);
    Matrix<float> m_des1(num_rows, num_cols);
    Matrix<float> m_des2(num_rows, num_cols);
```

```
    const float m_src_r0[] = { 10, 11, 12, 13 };
    const float m_src_r1[] = { 14, 15, 16, 17 };
    const float m_src_r2[] = { 18, 19, 20, 21 };
    const float m_src_r3[] = { 22, 23, 24, 25 };

    m_src.SetRow(0, m_src_r0);
    m_src.SetRow(1, m_src_r1);
    m_src.SetRow(2, m_src_r2);
    m_src.SetRow(3, m_src_r3);

    const size_t num_it = 500;
    const size_t num_alg = 2;
    const size_t num_ops = 1000000;

    BmThreadTimer bmtt(num_it, num_alg);

    for (size_t i = 0; i < num_it; i++)
    {
        bmtt.Start(i, 0);
        for (size_t j = 0; j < num_ops; j++)
            Matrix<float>::Transpose(m_des1, m_src);
        bmtt.Stop(i, 0);

        bmtt.Start(i, 1);
        for (size_t j = 0; j < num_ops; j++)
            AvxMat4x4TransposeF32_(m_des2.Data(), m_src.Data());
        bmtt.Stop(i, 1);
    }

    string fn = bmtt.BuildCsvFilenameString("Ch06_07_AvxMat4x4TransposeF32_BM");
    bmtt.SaveElapsedTimes(fn, BmThreadTimer::EtUnit::MicroSec, 2);
    cout << "Benchmark times save to file " << fn << '\n';
}
```

The function main begins by instantiating a 4 × 4 single-precision floating-point test matrix named m_src using the C++ template Matrix. This template, which is defined in the header file Matrix.h (source code not shown), contains C++ code that implements a simple matrix class for test and benchmarking purposes. The internal buffer allocated by Matrix is aligned on a 64-byte boundary, which means that objects of type Matrix are properly aligned for use with AVX, AVX2, and AVX-512 instructions. The function main calls AvxMat4x4TransposeF32, which exercises the matrix transposition functions written in C++ and assembly language. The results of these transpositions are then streamed to cout. The function main also invokes a benchmarking function named AvxMat4x4TransposeF32_BM that measures the performance of each transposition function as explained later in this section.

Near the top of assembly language code is a macro named _Mat4x4TransposeF32. You learned in Chapter 5 that a macro is an assembler text substitution mechanism that allows a single text string to represent a sequence of assembly language instructions, data definitions, or other statements. During assembly of an x86 assembly language source code file, the assembler replaces any occurrence of the macro name with the statements that are declared between the macro and endm directives. Assembly language macros are typically employed to generate sequences of instructions that will be used more than once. Macros are also frequently used to avoid the performance overhead of a function call.

The macro `_Mat4x4TransposeF32` contains AVX instructions that transpose a 4 × 4 matrix of single-precision floating-point values. This macro requires the rows of the source matrix to be loaded into registers XMM0 – XMM3 prior to its use. It then employs a series of `vunpcklps`, `vunpckhps`, `vmovlhps`, and `vmovhlps` instructions to transpose the source matrix, as illustrated in Figure 6-5. Following execution of these instructions, the transposed matrix is stored in registers XMM4–XMM7.

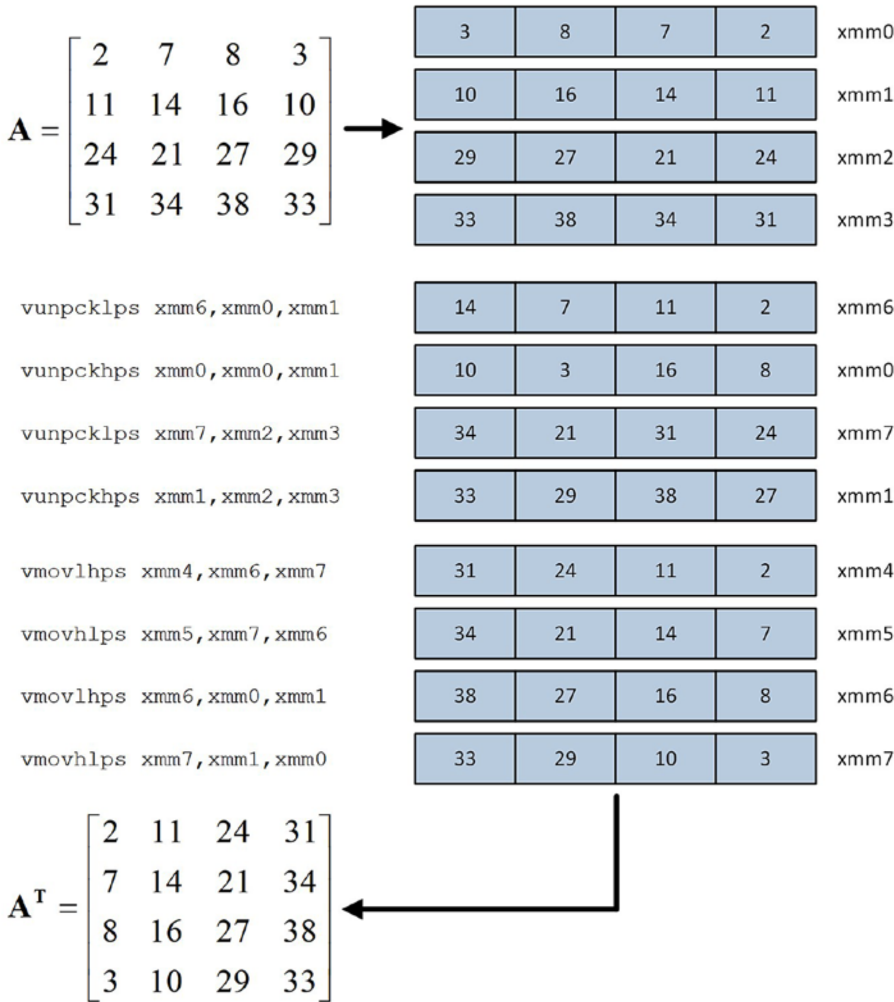



***Figure 6-5.** Instruction sequence used by `_Mat4x4TransposeF32` to transpose a 4 × 4 matrix of single-precision floating-point values*

The macro `_Mat4x4TransposeF32` is used by the assembly language function `AvxMat4x4Transpose4x4_`. Immediately following its function prolog, function `AvxMat4x4Transpose4x4_` executes a series of `vmovaps` instructions to load the source matrix into registers XMM0 – XMM3. Each XMM register contains one row of the source matrix. The macro `_Mat4x4TransposeF32` is then employed to transpose the matrix. Figure 6-6 contains an excerpt from the MASM listing file that shows the macro expansion of `_Mat4x4TransposeF32`. This figure also shows the expansions of the prolog and epilog macros. The listing file symbolizes macro

expanded instructions by placing a `1` in a column that's located to the left of the mnemonic. Following calculation of the transpose, the resultant matrix is saved to the destination buffer using another series of `vmovaps` instructions.

```
00000000                          AvxMat4x4TransposeF32_ proc frame
                                                  _CreateFrame MT_,0,32
 00000000  55                          1          push rbp
 00000001  48/ 83 EC 20                1          sub rsp,StackSizeTotal
 00000005  48/ 8D 6C 24                1          lea rbp,[rsp+32]
           20
                                                  _SaveXmmRegs xmm6,xmm7
 0000000A  C5 F9/ 7F 75                1          vmovdqa xmmword ptr [rbp-ValNameOffsetSaveXmmRegs],xmm6
           E0
 0000000F  C5 F9/ 7F 7D                1          vmovdqa xmmword ptr [rbp-ValNameOffsetSaveXmmRegs+16],xmm7
           F0
                                                  _EndProlog

                                        ; Transpose matrix m_src1
 00000014  C5 F8/ 28 02                           vmovaps xmm0,[rdx]                ;xmm0 = m_src.row_0
 00000018  C5 F8/ 28 4A                           vmovaps xmm1,[rdx+16]             ;xmm1 = m_src.row_1
           10
 0000001D  C5 F8/ 28 52                           vmovaps xmm2,[rdx+32]             ;xmm2 = m_src.row_2
           20
 00000022  C5 F8/ 28 5A                           vmovaps xmm3,[rdx+48]             ;xmm3 = m_src.row_3
           30

                                                  _Mat4x4TransposeF32
 00000027  C5 F8/ 14 F1                1          vunpcklps xmm6,xmm0,xmm1          ;xmm6 = b1 a1 b0 a0
 0000002B  C5 F8/ 15 C1                1          vunpckhps xmm0,xmm0,xmm1          ;xmm0 = b3 a3 b2 a2
 0000002F  C5 E8/ 14 FB                1          vunpcklps xmm7,xmm2,xmm3          ;xmm7 = d1 c1 d0 c0
 00000033  C5 E8/ 15 CB                1          vunpckhps xmm1,xmm2,xmm3          ;xmm1 = d3 c3 d2 c2
 00000037  C5 C8/ 16 E7                1          vmovlhps xmm4,xmm6,xmm7           ;xmm4 = d0 c0 b0 a0
 0000003B  C5 C0/ 12 EE                1          vmovhlps xmm5,xmm7,xmm6           ;xmm5 = d1 c1 b1 a1
 0000003F  C5 F8/ 16 F1                1          vmovlhps xmm6,xmm0,xmm1           ;xmm6 = d2 c2 b2 a2
 00000043  C5 F0/ 12 F8                1          vmovhlps xmm7,xmm1,xmm0           ;xmm7 = d3 c3 b2 a3

 00000047  C5 F8/ 29 21                           vmovaps [rcx],xmm4                ;save m_des.row_0
 0000004B  C5 F8/ 29 69                           vmovaps [rcx+16],xmm5             ;save m_des.row_1
           10
 00000050  C5 F8/ 29 71                           vmovaps [rcx+32],xmm6             ;save m_des.row_2
           20
 00000055  C5 F8/ 29 79                           vmovaps [rcx+48],xmm7             ;save m_des.row_3
           30

 0000005A                         Done:    _RestoreXmmRegs xmm6,xmm7
 0000005A  C5 F9/ 6F 75                1          vmovdqa xmm6,xmmword ptr [rbp-ValNameOffsetSaveXmmRegs]
           E0
 0000005F  C5 F9/ 6F 7D                1          vmovdqa xmm7,xmmword ptr [rbp-ValNameOffsetSaveXmmRegs+16]
           F0
                                                  _DeleteFrame
 00000064  48/ 8B E5                   1          mov rsp,rbp
 00000067  5D                          1          pop rbp
 00000068  C3                                     ret
 00000069                         AvxMat4x4TransposeF32_ endp
                                                  end
```

***Figure 6-6.*** *Expansion of macro `_Mat4x4TransposeF32`*

Here is the output for source code example Ch06_07:

```
Results for AvxMat4x4TransposeF32

Matrix m_src
       2.0           7.0           8.0           3.0
      11.0          14.0          16.0          10.0
      24.0          21.0          27.0          29.0
      31.0          34.0          38.0          33.0

Matrix m_des1
       2.0          11.0          24.0          31.0
       7.0          14.0          21.0          34.0
       8.0          16.0          27.0          38.0
       3.0          10.0          29.0          33.0

Matrix m_des2
       2.0          11.0          24.0          31.0
       7.0          14.0          21.0          34.0
       8.0          16.0          27.0          38.0
       3.0          10.0          29.0          33.0

Running benchmark function AvxMat4x4TransposeF32_BM - please wait
Benchmark times save to file Ch06_07_AvxMat4x4TransposeF32_BM_CHROMIUM.csv
```

Source code example Ch06_07 includes a function named AvxMat4x4TransposeF32_BM that contains code for measuring execution times of the C++ and assembly language matrix transposition functions. Most of the timing measurement code is encapsulated in a C++ class named BmThreadTimer. This class includes two member functions, BmThreadTimer::Start and BmThreadTimer::Stop, that implement a simple software stopwatch. Class BmThreadTimer also includes a member function named BmThreadTimer::SaveElapsedTimes, which saves the timing measurements to a comma-separated text file. AvxMat4x4Transpose_BM also uses a C++ class named OS. This class includes member functions that manage process and thread affinity. In the current example, OS::SetThreadAffinityMask selects a specific processor for benchmark thread execution. Doing this improves the accuracy of the timing measurements. The source code for classes BmThreadTimer and OS is not shown in Listing 6-7, but is included as part of the chapter download package.

Table 6-1 contains matrix transposition timing measurements using several different Intel processors. The measurements were made using an EXE file that was built with the Visual C++ Release configuration and the default settings for code optimization except for the following options: AVX code generation (/arch:AVX) was selected to facilitate "apples-to-apples" comparisons between the C++ and x86-64 assembly language code (the default code generation option for 64-bit Visual C++ is SSE2); whole program optimization was disabled. All timing measurements were made using ordinary desktop PCs running Windows 10. No attempt was made to account for any hardware, software, operating system, or configuration differences between the PCs prior to running the benchmark executable file. The test conditions described in this section are also used in subsequent chapters.

***Table 6-1.*** *Matrix Transposition Mean Execution Times (Microseconds), 1,000,000 Transpositions*

| CPU | C++ | Assembly Language |
|---|---|---|
| Intel Core i7-4790S | 15885 | 2575 |
| Intel Core i9-7900X | 13381 | 2203 |
| Intel Core i7-8700K | 12216 | 1825 |

The values shown in Table 6-1 were computed using the CSV file execution times and the Excel spreadsheet function `TRIMMEAN(array,0.10)`. The assembly language implementation of the matrix transposition algorithm clearly outperforms the C++ version by a wide margin. It is not uncommon to achieve significant speed improvements using x86 assembly language, especially by algorithms that can exploit the SIMD parallelism of an x86 processor. You'll see additional examples of accelerated algorithmic performance throughout the remainder of this book.

The benchmark timing measurements cited in this book provide reasonable approximations of function execution times. Like automobile fuel economy and battery runtime estimates, software performance benchmarking is not an exact science and subject to a variety of pitfalls. It is also important to keep mind that this book is an introductory primer about x86-64 assembly language programming and not benchmarking. The source code examples are structured to hasten the study of a new programming language and not maximum performance. In addition, the Visual C++ options described earlier were selected mostly for practical reasons and may not yield optimal performance in all cases. Like many high-level compilers, Visual C++ includes a plethora of code generation and speed options that can affect performance. Benchmark timing measurements should always be construed in a context that's correlated with the software's purpose. The methods described in this section are generally worthwhile, but results can vary.

## Matrix Multiplication

The product of two matrices is defined as follows. Let **A** be an $m \times n$ matrix where $m$ and $n$ denote the number of rows and columns, respectively. Let **B** be an $n \times p$ matrix. Let **C** be the product of **A** and **B**, which is an $m \times p$ matrix. The value of each element $c(i, j)$ in **C** can be calculated using the following equation:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \quad i = 0,\ldots,m-1;\ j = 0,\ldots,p-1$$

Before proceeding to the sample code, a few comments are warranted. According to the definition of matrix multiplication, the number of columns in **A** must equal the number of rows in **B**. For example, if **A** is a $3 \times 4$ matrix and **B** is a $4 \times 2$ matrix, the product **AB** (a $3 \times 2$ matrix) can be calculated but the product **BA** is undefined. Note that the value of each $c(i, j)$ in **C** is simply the dot product of row $i$ in matrix **A** and column $j$ in matrix **B**. The assembly language code will exploit this fact to perform matrix multiplications using packed AVX instructions. Also note that unlike most mathematical texts, the subscripts in the matrix multiplication equation use zero-based indexing. This simplifies translating the equation into C++ and assembly language code.

Listing 6-8 shows the source code for example `Ch06_08`. This example demonstrates how to perform matrix multiplication using two $4 \times 4$ matrices of single-precision floating-point values. Similar to the previous example, `main` calls a function named `AvxMat4x4MulF32` that exercises a matrix multiplication test case using functions written in C++ and assembly language. The template member function `Matrix<float>::Mul` (source code not shown) carries out C++ matrix multiplication using the previously described equation. The assembly language function `AvxMat4x4MulF32_` uses SIMD arithmetic to perform matrix multiplication as you'll soon see.

***Listing 6-8.*** Example Ch06_08

```
//------------------------------------------------
//               Ch06_08.h
//------------------------------------------------

#pragma once

// Ch06_08_.asm
extern "C" void AvxMat4x4MulF32_(float* m_des, const float* m_src1, const float* m_src2);

// Ch06_08_BM.cpp
extern void AvxMat4x4MulF32_BM(void);

//------------------------------------------------
//               Ch06_08.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "Ch06_08.h"
#include "Matrix.h"

using namespace std;

void AvxMat4x4MulF32(Matrix<float>& m_src1, Matrix<float>& m_src2)
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<float> m_des1(nr ,nc);
    Matrix<float> m_des2(nr ,nc);

    Matrix<float>::Mul(m_des1, m_src1, m_src2);
    AvxMat4x4MulF32_(m_des2.Data(), m_src1.Data(), m_src2.Data());

    cout << fixed << setprecision(1);

    m_src1.SetOstream(12, "  ");
    m_src2.SetOstream(12, "  ");
    m_des1.SetOstream(12, "  ");
    m_des2.SetOstream(12, "  ");

    cout << "\nResults for AvxMat4x4MulF32\n";
    cout << "Matrix m_src1\n" << m_src1 << '\n';
    cout << "Matrix m_src2\n" << m_src2 << '\n';
    cout << "Matrix m_des1\n" << m_des1 << '\n';
    cout << "Matrix m_des2\n" << m_des2 << '\n';

    if (m_des1 != m_des2)
        cout << "\nMatrix compare failed - AvxMat4x4MulF32\n";
}
```

```
int main()
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<float> m_src1(nr ,nc);
    Matrix<float> m_src2(nr ,nc);

    const float src1_row0[] = { 10, 11, 12, 13 };
    const float src1_row1[] = { 20, 21, 22, 23 };
    const float src1_row2[] = { 30, 31, 32, 33 };
    const float src1_row3[] = { 40, 41, 42, 43 };

    const float src2_row0[] = { 100, 101, 102, 103 };
    const float src2_row1[] = { 200, 201, 202, 203 };
    const float src2_row2[] = { 300, 301, 302, 303 };
    const float src2_row3[] = { 400, 401, 402, 403 };

    m_src1.SetRow(0, src1_row0);
    m_src1.SetRow(1, src1_row1);
    m_src1.SetRow(2, src1_row2);
    m_src1.SetRow(3, src1_row3);

    m_src2.SetRow(0, src2_row0);
    m_src2.SetRow(1, src2_row1);
    m_src2.SetRow(2, src2_row2);
    m_src2.SetRow(3, src2_row3);

    AvxMat4x4MulF32(m_src1, m_src2);
    AvxMat4x4MulF32_BM();
    return 0;
}

;-------------------------------------------------
;               Ch06_08.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; _Mat4x4MulCalcRowF32 macro
;
; Description:  This macro is used to compute one row of a 4x4 matrix
;               multiply.
;
; Registers:    xmm0 = m_src2.row0
;               xmm1 = m_src2.row1
;               xmm2 = m_src2.row2
;               xmm3 = m_src2.row3
;               rcx = m_des ptr
;               rdx = m_src1 ptr
;               xmm4 - xmm7 = scratch registers
```

```
_Mat4x4MulCalcRowF32 macro disp
        vbroadcastss xmm4,real4 ptr [rdx+disp]      ;broadcast m_src1[i][0]
        vbroadcastss xmm5,real4 ptr [rdx+disp+4]    ;broadcast m_src1[i][1]
        vbroadcastss xmm6,real4 ptr [rdx+disp+8]    ;broadcast m_src1[i][2]
        vbroadcastss xmm7,real4 ptr [rdx+disp+12]   ;broadcast m_src1[i][3]

        vmulps xmm4,xmm4,xmm0                      ;m_src1[i][0] * m_src2.row_0
        vmulps xmm5,xmm5,xmm1                      ;m_src1[i][1] * m_src2.row_1
        vmulps xmm6,xmm6,xmm2                      ;m_src1[i][2] * m_src2.row_2
        vmulps xmm7,xmm7,xmm3                      ;m_src1[i][3] * m_src2.row_3

        vaddps xmm4,xmm4,xmm5                      ;calc m_des.row_i
        vaddps xmm6,xmm6,xmm7
        vaddps xmm4,xmm4,xmm6

        vmovaps[rcx+disp],xmm4                     ;save m_des.row_i
        endm

; extern "C" void AvxMat4x4MulF32_(float* m_des, const float* m_src1, const float* m_src2)
;
; Description:  The following function computes the product of two
;               single-precision floating-point 4x4 matrices.

        .code
AvxMat4x4MulF32_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Compute matrix product m_des = m_src1 * m_src2
        vmovaps xmm0,[r8]                  ;xmm0 = m_src2.row_0
        vmovaps xmm1,[r8+16]               ;xmm1 = m_src2.row_1
        vmovaps xmm2,[r8+32]               ;xmm2 = m_src2.row_2
        vmovaps xmm3,[r8+48]               ;xmm3 = m_src2.row_3

        _Mat4x4MulCalcRowF32 0             ;calculate m_des.row_0
        _Mat4x4MulCalcRowF32 16            ;calculate m_des.row_1
        _Mat4x4MulCalcRowF32 32            ;calculate m_des.row_2
        _Mat4x4MulCalcRowF32 48            ;calculate m_des.row_3

Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4MulF32_ endp
        end

//------------------------------------------------
//               Ch06_08_BM.cpp
//------------------------------------------------
```

```
#include "stdafx.h"
#include <iostream>
#include "Ch06_08.h"
#include "Matrix.h"
#include "BmThreadTimer.h"
#include "OS.h"

using namespace std;

void AvxMat4x4MulF32_BM(void)
{
    OS::SetThreadAffinityMask();
    cout << "\nRunning benchmark function AvxMat4x4MulF32_BM - please wait\n";

    const size_t num_rows = 4;
    const size_t num_cols = 4;
    Matrix<float> m_src1(num_rows, num_cols);
    Matrix<float> m_src2(num_rows, num_cols);
    Matrix<float> m_des1(num_rows, num_cols);
    Matrix<float> m_des2(num_rows, num_cols);

    const float m_src1_r0[] = { 10, 11, 12, 13 };
    const float m_src1_r1[] = { 14, 15, 16, 17 };
    const float m_src1_r2[] = { 18, 19, 20, 21 };
    const float m_src1_r3[] = { 22, 23, 24, 25 };
    const float m_src2_r0[] = { 0, 1, 2, 3 };
    const float m_src2_r1[] = { 4, 5, 6, 7 };
    const float m_src2_r2[] = { 8, 9, 10, 11 };
    const float m_src2_r3[] = { 12, 13, 14, 15 };

    m_src1.SetRow(0, m_src1_r0);
    m_src1.SetRow(1, m_src1_r1);
    m_src1.SetRow(2, m_src1_r2);
    m_src1.SetRow(3, m_src1_r3);
    m_src2.SetRow(0, m_src2_r0);
    m_src2.SetRow(1, m_src2_r1);
    m_src2.SetRow(2, m_src2_r2);
    m_src2.SetRow(3, m_src2_r3);

    const size_t num_it = 500;
    const size_t num_alg = 2;
    const size_t num_ops = 1000000;

    BmThreadTimer bmtt(num_it, num_alg);

    for (size_t i = 0; i < num_it; i++)
    {
        bmtt.Start(i, 0);
        for (size_t j = 0; j < num_ops; j++)
            Matrix<float>::Mul(m_des1, m_src1, m_src2);
        bmtt.Stop(i, 0);
```

```
        bmtt.Start(i, 1);
        for (size_t j = 0; j < num_ops; j++)
            AvxMat4x4MulF32_(m_des2.Data(), m_src1.Data(), m_src2.Data());
        bmtt.Stop(i, 1);
    }

    string fn = bmtt.BuildCsvFilenameString("Ch06_08_AvxMat4x4MulF32_BM");
    bmtt.SaveElapsedTimes(fn, BmThreadTimer::EtUnit::MicroSec, 2);
    cout << "Benchmark times save to file " << fn << '\n';
}
```

The standard technique for performing matrix multiplication requires three nested `for` loops that employ scalar floating-point multiplication and addition (see the code for `Matrix<T>::Mul` in the header file `Matrix.h`). Figure 6-7 shows the explicit equations that can be used to calculate the elements of row 0 for the matrix product **C** = **AB**. Note that each row of matrix **B** is multiplied by the same element from matrix **A**. Similar sets of equations can be used to calculate rows 1, 2, and 3 of matrix **C**. The assembly language code in function `AvxMatMul4x4F32_` uses these equations to carry out matrix multiplication using SIMD arithmetic.

$$\mathbf{C} = \mathbf{AB}$$

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$\begin{aligned} c_{00} &= a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30} \\ c_{01} &= a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} + a_{03}b_{31} \\ c_{02} &= a_{00}b_{02} + a_{01}b_{12} + a_{02}b_{22} + a_{03}b_{32} \\ c_{03} &= a_{00}b_{03} + a_{01}b_{13} + a_{02}b_{23} + a_{03}b_{33} \end{aligned}$$

row0 row1 row2 row3

***Figure 6-7.*** *Equations for first row of matrix* ***C*** *=* ***AB***

Following its prolog, `AvxMatMul4x4F32_` loads matrix `m_src2` (or **B**) into registers XMM0–XMM3. The next four lines use the macro `_Mat4x4MulCalcRowF32` to calculate the products for rows 0–3 of `m_des` (or **C**). This macro implements the four equations that are shown in Figure 6-7. The macro parameter `disp` specifies which row to use. Macro `_Mat4x4MulCalcRowF32` uses four `vbroadcastss` instructions to load the required elements from matrix `m_src1` (or **A**) into registers XMM4–XMM7. It then uses four `vmulps` instructions to multiply these values by an entire row from matrix `m_src2`. A series of `vaddps` instructions computes the final element values for the row. The `vmovaps [rcx+disp],xmm4` instruction saves the entire row to the specified destination buffer. Here is the output for example `Ch06_08`:

```
Results for AvxMat4x4MulF32

Matrix m_src1
        10.0         11.0         12.0         13.0
        20.0         21.0         22.0         23.0
        30.0         31.0         32.0         33.0
        40.0         41.0         42.0         43.0

Matrix m_src2
       100.0        101.0        102.0        103.0
       200.0        201.0        202.0        203.0
       300.0        301.0        302.0        303.0
       400.0        401.0        402.0        403.0

Matrix m_des1
     12000.0      12046.0      12092.0      12138.0
     22000.0      22086.0      22172.0      22258.0
     32000.0      32126.0      32252.0      32378.0
     42000.0      42166.0      42332.0      42498.0

Matrix m_des2
     12000.0      12046.0      12092.0      12138.0
     22000.0      22086.0      22172.0      22258.0
     32000.0      32126.0      32252.0      32378.0
     42000.0      42166.0      42332.0      42498.0

Running benchmark function AvxMat4x4MulF32_BM - please wait
Benchmark times save to file Ch06_08_AvxMat4x4MulF32_BM_CHROMIUM.csv
```

Source code example Ch06_08 also includes a function named AvxMat4x4MulF32_BM that performs benchmark timing measurements of the matrix multiplication functions. Table 6-2 shows the timing measurements for several different Intel processors. These measurements were made using the procedure described in the previous section.

***Table 6-2.*** *Matrix Multiplication Mean Execution Times (Microseconds), 1,000,000 Multiplications*

| CPU | C++ | Assembly Language |
|---|---|---|
| Intel Core i7-4790S | 55195 | 5333 |
| Intel Core i9-7900X | 46008 | 4897 |
| Intel Core i7-8700K | 42260 | 4493 |

# Summary

Here are the key learning points for Chapter 6:

- The `vaddp[d|s]`, `vsubp[d|s]`, `vmulp[d|s]`, `vdivp[d|s]`, and `vsqrtp[d|s]` instructions carry out common arithmetic operation using packed double-precision and packed single-precision floating-point operands.
- The `vcvtp[d|s]2dq` and `vcvtdq2p[d|s]` instructions perform conversions between packed floating-point and packed signed-doubleword operands. The `vcvtps2pd` and `vcvtpd2ps` perform conversions between packed single-precision and double-precision operands.
- The `vminp[d|s]` and `vmaxp[d|s]` instructions perform packed minimum and maximum value calculations using double-precision and single-precision floating-point operands.
- The `vbroadcasts[d|s]` instructions broadcast (or copy) a single scalar double-precision or single-precision value to all element positions of an x86 SIMD register.
- Assembly language functions that use the `vmovap[d|s]` and `vmovdqa` instructions can only be used with operands in memory that are properly aligned. The MASM `align 16` directive aligns data items in a `.const` or `.data` section to a 16-byte boundary. C++ functions can use the `alignas` specifier to guarantee proper alignment.
- Assembly language functions can use the `vunpck[h|l]p[d|s]` instructions to accelerate common matrix operations, especially 4 × 4 matrices.
- Assembly language functions can use the `vhaddp[d|s]` and `vshufp[d|s]` instructions to perform data reductions of intermediate packed values.
- Many algorithms can achieve significant performance gains by using SIMD programming techniques and the x86-AVX instruction set.

CHAPTER 7

# AVX Programming – Packed Integers

In the previous chapter, you learned how to use the AVX instruction set to perform calculations using packed floating-point operands. In this chapter, you learn how to carry out computations using packed integer operands. Similar to the previous chapter, the first few source code examples in this chapter demonstrate basic arithmetic operations using packed integers. The remaining source code examples illustrate how to use the computational resources of AVX to perform common image processing operations, including histogram creation and thresholding.

AVX supports packed integer operations using 128-bit wide operands, and that is the focus of the source code examples in this chapter. Performing packed integer operations using 256-bit operands requires a processor that supports AVX2. You learn about AVX2 programming with packed integers in Chapter 10.

## Packed Integer Addition and Subtraction

Listing 7-1 shows the C++ and assembly language source code for example Ch07_01. This example demonstrates how to perform packed integer addition and subtraction using signed and unsigned 16-bit integers. It also illustrates both wraparound and saturated arithmetic.

***Listing 7-1.*** Example Ch07_01

```
//------------------------------------------------
//               Ch07_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <string>
#include "XmmVal.h"

using namespace std;

extern "C" void AvxPackedAddI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2]);
extern "C" void AvxPackedSubI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2]);
extern "C" void AvxPackedAddU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2]);
extern "C" void AvxPackedSubU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2]);

//
// Signed packed addition and subtraction
//
```

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_7

```
void AvxPackedAddI16(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[2];

    a.m_I16[0] = 10;          b.m_I16[0] = 100;
    a.m_I16[1] = 200;         b.m_I16[1] = -200;
    a.m_I16[2] = 30;          b.m_I16[2] = 32760;
    a.m_I16[3] = -32766;      b.m_I16[3] = -400;
    a.m_I16[4] = 50;          b.m_I16[4] = 500;
    a.m_I16[5] = 60;          b.m_I16[5] = -600;
    a.m_I16[6] = 32000;       b.m_I16[6] = 1200;
    a.m_I16[7] = -32000;      b.m_I16[7] = -950;

    AvxPackedAddI16_(a, b, c);

    cout << "\nResults for AxvPackedAddI16 - Wraparound Addition\n";
    cout << "a:     " << a.ToStringI16() << '\n';
    cout << "b:     " << b.ToStringI16() << '\n';
    cout << "c[0]:  " << c[0].ToStringI16() << '\n';
    cout << "\nResults for AxvPackedAddI16 - Saturated Addition\n";
    cout << "a:     " << a.ToStringI16() << '\n';
    cout << "b:     " << b.ToStringI16() << '\n';
    cout << "c[1]:  " << c[1].ToStringI16() << '\n';
}

void AvxPackedSubI16(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[2];

    a.m_I16[0] = 10;          b.m_I16[0] = 100;
    a.m_I16[1] = 200;         b.m_I16[1] = -200;
    a.m_I16[2] = -30;         b.m_I16[2] = 32760;
    a.m_I16[3] = -32766;      b.m_I16[3] = 400;
    a.m_I16[4] = 50;          b.m_I16[4] = 500;
    a.m_I16[5] = 60;          b.m_I16[5] = -600;
    a.m_I16[6] = 32000;       b.m_I16[6] = 1200;
    a.m_I16[7] = -32000;      b.m_I16[7] = 950;

    AvxPackedSubI16_(a, b, c);

    cout << "\nResults for AxvPackedSubI16 - Wraparound Subtraction\n";
    cout << "a:     " << a.ToStringI16() << '\n';
    cout << "b:     " << b.ToStringI16() << '\n';
    cout << "c[0]:  " << c[0].ToStringI16() << '\n';
    cout << "\nResults for AxvPackedSubI16 - Saturated Subtraction\n";
    cout << "a:     " << a.ToStringI16() << '\n';
```

```
    cout << "b:     " << b.ToStringI16() << '\n';
    cout << "c[1]:  " << c[1].ToStringI16() << '\n';
}

//
// Unsigned packed addition and subtraction
//

void AvxPackedAddU16(void)
{
    XmmVal a;
    XmmVal b;
    XmmVal c[2];

    a.m_U16[0] = 10;          b.m_U16[0] = 100;
    a.m_U16[1] = 200;         b.m_U16[1] = 200;
    a.m_U16[2] = 300;         b.m_U16[2] = 65530;
    a.m_U16[3] = 32766;       b.m_U16[3] = 40000;
    a.m_U16[4] = 50;          b.m_U16[4] = 500;
    a.m_U16[5] = 20000;       b.m_U16[5] = 25000;
    a.m_U16[6] = 32000;       b.m_U16[6] = 1200;
    a.m_U16[7] = 32000;       b.m_U16[7] = 50000;

    AvxPackedAddU16_(a, b, c);

    cout << "\nResults for AxvPackedAddU16 - Wraparound Addition\n";
    cout << "a:     " << a.ToStringU16() << '\n';
    cout << "b:     " << b.ToStringU16() << '\n';
    cout << "c[0]:  " << c[0].ToStringU16() << '\n';
    cout << "\nResults for AxvPackedAddU16 - Saturated Addition\n";
    cout << "a:     " << a.ToStringU16() << '\n';
    cout << "b:     " << b.ToStringU16() << '\n';
    cout << "c[1]:  " << c[1].ToStringU16() << '\n';
}

void AvxPackedSubU16(void)
{
    XmmVal a;
    XmmVal b;
    XmmVal c[2];

    a.m_U16[0] = 10;          b.m_U16[0] = 100;
    a.m_U16[1] = 200;         b.m_U16[1] = 200;
    a.m_U16[2] = 30;          b.m_U16[2] = 7;
    a.m_U16[3] = 65000;       b.m_U16[3] = 5000;
    a.m_U16[4] = 60;          b.m_U16[4] = 500;
    a.m_U16[5] = 25000;       b.m_U16[5] = 28000;
    a.m_U16[6] = 32000;       b.m_U16[6] = 1200;
    a.m_U16[7] = 1200;        b.m_U16[7] = 950;
```

```
    AvxPackedSubU16_(a, b, c);

    cout << "\nResults for AxvPackedSubU16 - Wraparound Subtraction\n";
    cout << "a:      " << a.ToStringU16() << '\n';
    cout << "b:      " << b.ToStringU16() << '\n';
    cout << "c[0]:   " << c[0].ToStringU16() << '\n';
    cout << "\nResults for AxvPackedSubI16 - Saturated Subtraction\n";
    cout << "a:      " << a.ToStringU16() << '\n';
    cout << "b:      " << b.ToStringU16() << '\n';
    cout << "c[1]:   " << c[1].ToStringU16() << '\n';
}

int main()
{
    string sep(75, '-');

    AvxPackedAddI16();
    AvxPackedSubI16();
    cout << '\n' << sep << '\n';
    AvxPackedAddU16();
    AvxPackedSubU16();
    return 0;
}

;-------------------------------------------------
;               Ch07_01.asm
;-------------------------------------------------

; extern "C" void AvxPackedAddI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])

        .code
AvxPackedAddI16_ proc

; Packed signed word addition
        vmovdqa xmm0,xmmword ptr [rcx]      ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [rdx]      ;xmm1 = b

        vpaddw xmm2,xmm0,xmm1               ;packed add - wraparound
        vpaddsw xmm3,xmm0,xmm1              ;packed add - saturated

        vmovdqa xmmword ptr [r8],xmm2       ;save c[0]
        vmovdqa xmmword ptr [r8+16],xmm3    ;save c[1]
        ret
AvxPackedAddI16_ endp

; extern "C" void AvxPackedSubI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])

AvxPackedSubI16_ proc

; Packed signed word subtraction
        vmovdqa xmm0,xmmword ptr [rcx]      ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [rdx]      ;xmm1 = b
```

```
        vpsubw xmm2,xmm0,xmm1                ;packed sub - wraparound
        vpsubsw xmm3,xmm0,xmm1               ;packed sub - saturated

        vmovdqa xmmword ptr [r8],xmm2        ;save c[0]
        vmovdqa xmmword ptr [r8+16],xmm3     ;save c[1]
        ret
AvxPackedSubI16_ endp

; extern "C" void AvxPackedAddU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])

AvxPackedAddU16_ proc

; Packed unsigned word addition
        vmovdqu xmm0,xmmword ptr [rcx]       ;xmm0 = a
        vmovdqu xmm1,xmmword ptr [rdx]       ;xmm1 = b

        vpaddw xmm2,xmm0,xmm1                ;packed add - wraparound
        vpaddusw xmm3,xmm0,xmm1              ;packed add - saturated

        vmovdqu xmmword ptr [r8],xmm2        ;save c[0]
        vmovdqu xmmword ptr [r8+16],xmm3     ;save c[1]
        ret
AvxPackedAddU16_ endp

; extern "C" void AvxPackedSubU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])

AvxPackedSubU16_ proc

; Packed unsigned word subtraction
        vmovdqu xmm0,xmmword ptr [rcx]       ;xmm0 = a
        vmovdqu xmm1,xmmword ptr [rdx]       ;xmm1 = b

        vpsubw xmm2,xmm0,xmm1                ;packed sub - wraparound
        vpsubusw xmm3,xmm0,xmm1              ;packed sub - saturated

        vmovdqu xmmword ptr [r8],xmm2        ;save c[0]
        vmovdqu xmmword ptr [r8+16],xmm3     ;save c[1]
        ret
AvxPackedSubU16_ endp
        end
```

Toward the top of the C++ code are the declarations for the assembly language functions that perform packed integer addition and subtraction. Each function takes two XmmVal arguments and saves its results to an XmmVal array. The structure XmmVal, which you learned about in Chapter 6 (see Listing 6-1), contains a publicly-accessible anonymous union with members that correspond to the packed data types that can be used with an XMM register. The XmmVal structure also defines several member functions that format the contents of an XmmVal for display.

The C++ function AvxPackedAddI16 contains test code that exercises the assembly language function AvxPackedAddI16_. This function performs packed signed 16-bit integer (word) addition using both wraparound and saturated arithmetic. Note that the XmmVal variables a, b, and c are all defined using the C++ specifier alignas(16), which aligns each XmmVal to a 16-byte boundary. Following the execution of the function AvxPackedaddI16_, the results are displayed using a series of stream writes to cout. The C++ function AvxPackedSubI16, which is similar to AvxPackedAddI16, uses the assembly language function AvxPackedSubI16_.

A parallel set of C++ functions, AvxPackedAddU16 and AvxPackedSubU16, contain code that exercise the assembly language functions AvxPackedAddU16_ and AvxPackedSubU16_. These functions perform packed unsigned 16-bit integer addition and subtraction, respectively. Note that the XmmVal variables in AvxPackedAddU16 and AvxPackedSubU16 do not use the alignas(16) specifier, which means that these values are not guaranteed to be aligned on a 16-byte boundary. The reason for doing this is to demonstrate the use of the AVX instruction vmovdqu (Move Unaligned Packed Integer Values), as you'll soon see.

The assembly language function AvxPackedAddI16_ starts with a vmovdqa xmm0,xmmword ptr [rcx] instruction that loads argument value a into register XMM0. The ensuing vmovdqa xmm1,xmmword ptr [rdx] instruction copies b into register XMM1. The next two instructions, vpaddw xmm2,xmm0,xmm1 and vpaddsw xmm3,xmm0,xmm1, carry out packed signed 16-bit integer addition using wrapround and saturated arithmetic, respectively. The final two vmovdqa instructions save the calculated results to XmmVal array c. Assembly language function AvxPackedSubI16_ is similar to AvxPackedAddI16_ and uses the instructions vpsubw and vpsubsw to carry out packed signed 16-bit integer subtraction.

The assembly language function AvxPackedAddU16_ begins with a vmovdqu xmm0,xmmword ptr [rcx] instruction that loads a into register XMM0. A vmovdqu instruction is used here since XmmVal a was defined in the C++ code without the alignas(16) specifier. Note that function AvxPackedAddU16_ uses vmovdqu for demonstration purposes only; a properly aligned XmmVal and a vmovdqa instruction should have been used instead. It's already been mentioned a number of times in this book but warrants repeating due to its importance: SIMD operands in memory should be properly aligned whenever possible in order to avoid potential performance penalties that can occur whenever the processor accesses an unaligned operand in memory.

Following the loading of argument values a and b into register XMM0 and XMM1, function AvxPackedAddU16_ performs packed unsigned 16-bit integer addition using the instructions vpaddw xmm2,xmm0,xmm1 (wraparound arithmetic) and vpaddusw xmm3,xmm0,xmm1 (saturated arithmetic). Two vmovdqu instructions save the results to array c. The function AvxPackedSubU16_ implements packed unsigned 16-bit integer subtraction using the vpsubw and vpsubusw instructions. This function also uses the vmovdqu instruction to load argument values and save results. Here are the results for source code example Ch07_01:

```
Results for AxvPackedAddI16 - Wraparound Addition
a:           10     200      30  -32766   |      50      60   32000  -32000
b:          100    -200   32760    -400   |     500    -600    1200    -950
c[0]:       110       0  -32746   32370   |     550    -540  -32336   32586

Results for AxvPackedAddI16 - Saturated Addition
a:           10     200      30  -32766   |      50      60   32000  -32000
b:          100    -200   32760    -400   |     500    -600    1200    -950
c[1]:       110       0   32767  -32768   |     550    -540   32767  -32768

Results for AxvPackedSubI16 - Wraparound Subtraction
a:           10     200     -30  -32766   |      50      60   32000  -32000
b:          100    -200   32760     400   |     500    -600    1200     950
c[0]:       -90     400   32746   32370   |    -450     660   30800   32586
```

```
Results for AxvPackedSubI16 - Saturated Subtraction
a:           10      200      -30   -32766   |      50      60   32000  -32000
b:          100     -200    32760      400   |     500    -600    1200     950
c[1]:       -90      400   -32768   -32768   |    -450     660   30800  -32768

---------------------------------------------------------------------------

Results for AxvPackedAddU16 - Wraparound Addition
a:           10      200      300    32766   |      50   20000   32000   32000
b:          100      200    65530    40000   |     500   25000    1200   50000
c[0]:       110      400      294     7230   |     550   45000   33200   16464

Results for AxvPackedAddU16 - Saturated Addition
a:           10      200      300    32766   |      50   20000   32000   32000
b:          100      200    65530    40000   |     500   25000    1200   50000
c[1]:       110      400    65535    65535   |     550   45000   33200   65535

Results for AxvPackedSubU16 - Wraparound Subtraction
a:           10      200       30    65000   |      60   25000   32000    1200
b:          100      200        7     5000   |     500   28000    1200     950
c[0]:     65446        0       23    60000   |   65096   62536   30800     250

Results for AxvPackedSubI16 - Saturated Subtraction
a:           10      200       30    65000   |      60   25000   32000    1200
b:          100      200        7     5000   |     500   28000    1200     950
c[1]:         0        0       23    60000   |       0       0   30800     250
```

AVX also supports packed integer addition and subtraction using 8-, 32-, and 64-bit integers. The vpaddb, vpaddsb, vpaddusb, vpsubb, vpsubsb, and vpsubusb instructions are the 8-bit (byte) versions of the packed 16-bit instructions that were demonstrated in this example. The vpadd[d|q] and vpsub[d|q] instructions can be employed to perform packed 32-bit (doubleword) or 64-bit (quadword) addition and subtraction using wraparound arithmetic. AVX does not support saturated addition and subtraction using packed doubleword or quadword integers.

# Packed Integer Shifts

The next source code example is named Ch07_02. This example illustrates how to perform logical and arithmetic shift operations using packed integer operands. Listing 7-2 shows the C++ and assembly language source code for example Ch07_02.

***Listing 7-2.*** Example Ch07_02

```
//------------------------------------------------
//               Ch07_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include "XmmVal.h"
```

```
using namespace std;

// The order of the name constants in the following enum must
// correspond to the table values defined in .asm file.

enum ShiftOp : unsigned int
{
    U16_LL,     // shift left logical - word
    U16_RL,     // shift right logical - word
    U16_RA,     // shift right arithmetic - word
    U32_LL,     // shift left logical - doubleword
    U32_RL,     // shift right logical - doubleword
    U32_RA,     // shift right arithmetic - doubleword
};

extern "C" bool AvxPackedIntegerShift_(XmmVal& b, const XmmVal& a, ShiftOp shift_op,
unsigned int count);

void AvxPackedIntegerShiftU16(void)
{
    unsigned int count = 2;
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;

    a.m_U16[0] = 0x1234;
    a.m_U16[1] = 0xFF00;
    a.m_U16[2] = 0x00CC;
    a.m_U16[3] = 0x8080;
    a.m_U16[4] = 0x00FF;
    a.m_U16[5] = 0xAAAA;
    a.m_U16[6] = 0x0F0F;
    a.m_U16[7] = 0x0101;

    AvxPackedIntegerShift_(b, a, U16_LL, count);
    cout << "\nResults for ShiftOp::U16_LL (count = " << count << ")\n";
    cout << "a: " << a.ToStringX16() << '\n';
    cout << "b: " << b.ToStringX16() << '\n';

    AvxPackedIntegerShift_(b, a, U16_RL, count);
    cout << "\nResults for ShiftOp::U16_RL (count = " << count << ")\n";
    cout << "a: " << a.ToStringX16() << '\n';
    cout << "b: " << b.ToStringX16() << '\n';

    AvxPackedIntegerShift_(b, a, U16_RA, count);
    cout << "\nResults for ShiftOp::U16_RA (count = " << count << ")\n";
    cout << "a: " << a.ToStringX16() << '\n';
    cout << "b: " << b.ToStringX16() << '\n';
}
```

```
void AvxPackedIntegerShiftU32(void)
{
    unsigned int count = 4;
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;

    a.m_U32[0] = 0x12345678;
    a.m_U32[1] = 0xFF00FF00;
    a.m_U32[2] = 0x03030303;
    a.m_U32[3] = 0x80800F0F;

    AvxPackedIntegerShift_(b, a, U32_LL, count);
    cout << "\nResults for ShiftOp::U32_LL (count = " << count << ")\n";
    cout << "a: " << a.ToStringX32() << '\n';
    cout << "b: " << b.ToStringX32() << '\n';

    AvxPackedIntegerShift_(b, a, U32_RL, count);
    cout << "\nResults for ShiftOp::U32_RL (count = " << count << ")\n";
    cout << "a: " << a.ToStringX32() << '\n';
    cout << "b: " << b.ToStringX32() << '\n';

    AvxPackedIntegerShift_(b, a, U32_RA, count);
    cout << "\nResults for ShiftOp::U32_RA (count = " << count << ")\n";
    cout << "a: " << a.ToStringX32() << '\n';
    cout << "b: " << b.ToStringX32() << '\n';
}

int main(void)
{
    string sep(75, '-');

    AvxPackedIntegerShiftU16();
    cout << '\n' << sep << '\n';
    AvxPackedIntegerShiftU32();
    return 0;
}

;-------------------------------------------------
;               Ch07_02.asm
;-------------------------------------------------

; extern "C" bool AvxPackedIntegerShift_(XmmVal& b, const XmmVal& a, ShiftOp shift_op,
unsigned int count)
;
; Returns:      0 = invalid shift_op argument, 1 = success
;
; Note:         This module requires linker option /LARGEADDRESSAWARE:NO
;               to be explicitly set.

        .code
```

```
AvxPackedIntegerShift_ proc
; Make sure 'shift_op' is valid
        mov r8d,r8d                     ;zero extend shift_op
        cmp r8,ShiftOpTableCount        ;compare against table count
        jae Error                       ;jump if shift_op is invalid

; Jump to the operation specified by shift_op
        vmovdqa xmm0,xmmword ptr [rdx]  ;xmm0 = a
        vmovd xmm1,r9d                  ;xmm1[31:0] = shift count
        mov eax,1                       ;set success return code
        jmp [ShiftOpTable+r8*8]

; Packed shift left logical - word
U16_LL: vpsllw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift right logical - word
U16_RL: vpsrlw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift right arithmetic - word
U16_RA: vpsraw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift left logical - doubleword
U32_LL: vpslld xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift right logical - doubleword
U32_RL: vpsrld xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Packed shift right arithmetic - doubleword
U32_RA: vpsrad xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

Error:  xor eax,eax                     ;set error code
        vpxor xmm0,xmm0,xmm0
        vmovdqa xmmword ptr [rcx],xmm0  ;set result to zero
        ret

; The order of the labels in the following table must correspond
; to the enums that are defined in .cpp file.
```

```
                align 8
ShiftOpTable    qword U16_LL, U16_RL, U16_RA
                qword U32_LL, U32_RL, U32_RA
ShiftOpTableCount equ ($ - ShiftOpTable) / size qword

AvxPackedIntegerShift_ endp
        end
```

The C++ code that's shown in Listing 7-2 begins with the definition of an enum named ShiftOp, which is used to select a shift operation. Supported shift operations include logical left, logical right, and arithmetic right using packed word and doubleword values. Following enum ShiftOp is the declaration for the function AvxPackedIntegerShift_. This function carries out the requested shift operation using the supplied XmmVal argument and the specified count value. The C++ functions AvxPackedIntegerShiftU16 and AvxPackedIntegerShiftU32 initialize test cases for performing various shift operations using packed words and doublewords, respectively.

Assembly language function AvxPackedIntegerShift_ uses a jump table to execute the specified shift operation. This is similar to what you saw in source code examples Ch05_06 (Chapter 5) and Ch06_03 (Chapter 6). Upon entry to AvxPackedIntegerShift_, the argument value shift_op is tested for validity. Following validation of shift_op, a vmovdqa xmm0,xmmword ptr [rdx] instruction loads a into register XMM0. The subsequent vmovd xmm1,r9d instruction copies argument value count into the low-order doubleword of register XMM1. This is followed by a jmp [ShiftOpTable+r8*8] instruction that transfers program control to the appropriate code block.

Each distinct code block in AvxPackedIntegerShift_ performs a particular shift operation. For example, the code block adjacent to the label U16_LL uses the AVX instruction vpsllw xmm2,xmm0,xmm1 to perform a logical left shift using packed words. It is important to note that every word element in XMM0 is independently shifted left by the number of bits specified in XMM1[31:0]. The code blocks adjacent to the labels U16_RL and U16_RA carry out logical and arithmetic right shifts of packed words using the instructions vpsrlw and vpsraw, respectively. The function AvxPackedIntegerShift_ employs a similar structure to perform packed shift operations on doublewords using the instructions vpslld, vpslrd, and vpsrad. All of the code blocks in AvxPackedIntegerShift_ conclude with a vmovdqa xmmword ptr [rcx],xmm2 instruction that saves the calculated result. Here is the output for source code example Ch07_02:

```
Results for ShiftOp::U16_LL (count = 2)
a:     1234     FF00     00CC     8080   |    00FF     AAAA     0F0F     0101
b:     48D0     FC00     0330     0200   |    03FC     AAA8     3C3C     0404

Results for ShiftOp::U16_RL (count = 2)
a:     1234     FF00     00CC     8080   |    00FF     AAAA     0F0F     0101
b:     048D     3FC0     0033     2020   |    003F     2AAA     03C3     0040

Results for ShiftOp::U16_RA (count = 2)
a:     1234     FF00     00CC     8080   |    00FF     AAAA     0F0F     0101
b:     048D     FFC0     0033     E020   |    003F     EAAA     03C3     0040

---------------------------------------------------------------------------

Results for ShiftOp::U32_LL (count = 4)
a:         12345678         FF00FF00     |        03030303         80800F0F
b:         23456780         F00FF000     |        30303030         0800F0F0
```

```
Results for ShiftOp::U32_RL (count = 4)
a:          12345678        FF00FF00    |        03030303        80800F0F
b:          01234567        0FF00FF0    |        00303030        080800F0

Results for ShiftOp::U32_RA (count = 4)
a:          12345678        FF00FF00    |        03030303        80800F0F
b:          01234567        FFF00FF0    |        00303030        F80800F0
```

The AVX instructions `vpsllq`, `vpslrq`, and `vpsraq` can be used to perform shift operations using packed quadwords. Somewhat surprisingly, AVX does not support shift operations using packed byte operands. AVX also includes the shift instructions `vps[l|r]dq`, which carry out logical left or logical right byte shifts of a 128-bit wide operand in an XMM register. You'll see how these instructions work in the next section.

# Packed Integer Multiplication

Besides packed integer addition and subtraction, AVX also includes instructions that perform packed integer multiplication. These instructions are slightly different than the corresponding packed addition and subtraction instructions. Part of this difference is due to the fact that in order to calculate a non-truncated product, integer multiplication requires a destination operand to be twice the size of the original source operands. For example, the non-truncated product of two signed 16-bit integers is a signed 32-bit integer. Listing 7-3 shows the source code for example `Ch07_03`. This example demonstrates how to perform packed integer multiplication using signed 16-bit and 32-bit integers.

***Listing 7-3.*** Example Ch07_03

```
//------------------------------------------------
//               Ch07_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include "XmmVal.h"

using namespace std;

extern "C" void AvxPackedMulI16_(XmmVal c[2], const XmmVal& a, const XmmVal& b);
extern "C" void AvxPackedMulI32A_(XmmVal c[2], const XmmVal& a, const XmmVal& b);
extern "C" void AvxPackedMulI32B_(XmmVal* c, const XmmVal& a, const XmmVal& b);

void AvxPackedMulI16(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[2];

    a.m_I16[0] = 10;        b.m_I16[0] = -5;
    a.m_I16[1] = 3000;      b.m_I16[1] = 100;
    a.m_I16[2] = -2000;     b.m_I16[2] = -9000;
    a.m_I16[3] = 42;        b.m_I16[3] = 1000;
    a.m_I16[4] = -5000;     b.m_I16[4] = 25000;
```

```
    a.m_I16[5] = 8;         b.m_I16[5] = 16384;
    a.m_I16[6] = 10000;     b.m_I16[6] = 3500;
    a.m_I16[7] = -60;       b.m_I16[7] = 6000;

    AvxPackedMulI16_(c, a, b);

    cout << "\nResults for AvxPackedMulI16\n";

    for (size_t i = 0; i < 8; i++)
    {
        cout << "a[" << i << "]: " << setw(8) << a.m_I16[i] << "  ";
        cout << "b[" << i << "]: " << setw(8) << b.m_I16[i] << "  ";

        if (i < 4)
        {
            cout << "c[0][" << i << "]: ";
            cout << setw(12) << c[0].m_I32[i] << '\n';
        }
        else
        {
            cout << "c[1][" << i - 4 << "]: ";
            cout << setw(12) << c[1].m_I32[i - 4] << '\n';
        }
    }
}

void AvxPackedMulI32A(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[2];

    a.m_I32[0] = 10;        b.m_I32[0] = -500;
    a.m_I32[1] = 3000;      b.m_I32[1] = 100;
    a.m_I32[2] = -40000;    b.m_I32[2] = -120000;
    a.m_I32[3] = 4200;      b.m_I32[3] = 1000;

    AvxPackedMulI32A_(c, a, b);

    cout << "\nResults for AvxPackedMulI32A\n";

    for (size_t i = 0; i < 4; i++)
    {
        cout << "a[" << i << "]: " << setw(10) << a.m_I32[i] << "  ";
        cout << "b[" << i << "]: " << setw(10) << b.m_I32[i] << "  ";

        if (i < 2)
        {
            cout << "c[0][" << i << "]: ";
            cout << setw(14) << c[0].m_I64[i] << '\n';
        }
```

```
        else
        {
            cout << "c[1][" << i - 2 << "]: ";
            cout << setw(14) << c[1].m_I64[i - 2] << '\n';
        }
    }
}

void AvxPackedMulI32B(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c;

    a.m_I32[0] = 10;        b.m_I32[0] = -500;
    a.m_I32[1] = 3000;      b.m_I32[1] = 100;
    a.m_I32[2] = -2000;     b.m_I32[2] = -12000;
    a.m_I32[3] = 4200;      b.m_I32[3] = 1000;

    AvxPackedMulI32B_(&c, a, b);

    cout << "\nResults for AvxPackedMulI32B\n";

    for (size_t i = 0; i < 4; i++)
    {
        cout << "a[" << i << "]: " << setw(10) << a.m_I32[i] << "  ";
        cout << "b[" << i << "]: " << setw(10) << b.m_I32[i] << "  ";
        cout << "c[" << i << "]: " << setw(10) << c.m_I32[i] << '\n';
    }
}

int main()
{
    string sep(75, '-');

    AvxPackedMulI16();
    cout << '\n' << sep << '\n';
    AvxPackedMulI32A();
    cout << '\n' << sep << '\n';
    AvxPackedMulI32B();
    return 0;
}
;-------------------------------------------------
;               Ch07_03.asm
;-------------------------------------------------

; extern "C" void AvxPackedMulI16_(XmmVal c[2], const XmmVal* a, const XmmVal* b)

        .code
AvxPackedMulI16_ proc
        vmovdqa xmm0,xmmword ptr [rdx]      ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [r8]       ;xmm1 = b
```

```
        vpmullw xmm2,xmm0,xmm1              ;xmm2 = packed a * b low result
        vpmulhw xmm3,xmm0,xmm1              ;xmm3 = packed a * b high result

        vpunpcklwd xmm4,xmm2,xmm3           ;merge low and high results
        vpunpckhwd xmm5,xmm2,xmm3           ;into final signed dwords

        vmovdqa xmmword ptr [rcx],xmm4      ;save final results
        vmovdqa xmmword ptr [rcx+16],xmm5
        ret
AvxPackedMulI16_ endp

; extern "C" void AvxPackedMulI32A_(XmmVal c[2], const XmmVal* a, const XmmVal* b)

AvxPackedMulI32A_ proc

; Perform packed signed dword multiplication.  Note that vpmuldq
; performs following operations:
;
; xmm2[63:0]   = xmm0[31:0]  * xmm1[31:0]
; xmm2[127:64] = xmm0[95:64] * xmm1[95:64]

        vmovdqa xmm0,xmmword ptr [rdx]      ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [r8]       ;xmm1 = b
        vpmuldq xmm2,xmm0,xmm1

; Shift source operands right by 4 bytes and repeat vpmuldq
        vpsrldq xmm0,xmm0,4
        vpsrldq xmm1,xmm1,4
        vpmuldq xmm3,xmm0,xmm1

; Save results
        vpextrq qword ptr [rcx],xmm2,0      ;save xmm2[63:0]
        vpextrq qword ptr [rcx+8],xmm3,0    ;save xmm3[63:0]
        vpextrq qword ptr [rcx+16],xmm2,1   ;save xmm2[127:63]
        vpextrq qword ptr [rcx+24],xmm3,1   ;save xmm3[127:63]
        ret
AvxPackedMulI32A_ endp

; extern "C" void AvxPackedMulI32B_(XmmVal*, const XmmVal* a, const XmmVal* b)

AvxPackedMulI32B_ proc

; Perform packed signed integer multiplication and save low packed dword result
        vmovdqa xmm0,xmmword ptr [rdx]              ;xmm0 = a
        vpmulld xmm1,xmm0,xmmword ptr [r8]          ;xmm1 = packed a * b
        vmovdqa xmmword ptr [rcx],xmm1              ;save packed dword result
        ret
AvxPackedMulI32B_ endp
        end
```

The C++ function AvxPackedMulI16 contains code that initializes XmmVal variables a and b using signed 16-bit integers. This function then invokes the assembly language function AxvPackedMulI16_, which performs packed multiplication using signed 16-bit integers. The results are then streamed to cout. Note that the results displayed by function AvxPackedMulI16 are signed 32-bit integer products. The other two C++ functions in Listing 7-3, AvxPackedMulI32A and AvxPackedMul32B, initialize test cases for performing packed signed 32-bit integer multiplication. The former of these functions computes a packed signed 64-bit integer product, while the latter calculates a packed signed 32-bit integer product.

The assembly language function AvxPackedMulI16_ begins with two vmovdqa instructions that load argument values a and b into registers XMM0 and XMM1, respectively. The ensuing vpmullw xmm2,xmm0,xmm1 instruction multiplies the packed signed 16-bit integers in XMM0 and XMM1 and saves the low-order 16 bits of each 32-bit product in XMM2. This is followed by a vpmulhw xmm3,xmm0,xmm1 instruction that calculates and saves the high-order 16 bits of each 32-bit product. The next two instructions, vpunpcklwd xmm4,xmm2,xmm3 and vpunpckhwd xmm5,xmm2,xmm3, create the final packed 32-bit signed integer products by interleaving the low-order (vpunpcklwd) or high-order (vpunpckhwd) words of their source operands. Figure 7-1 illustrates the instruction sequence that's employed by AvxPackedMulI16_.

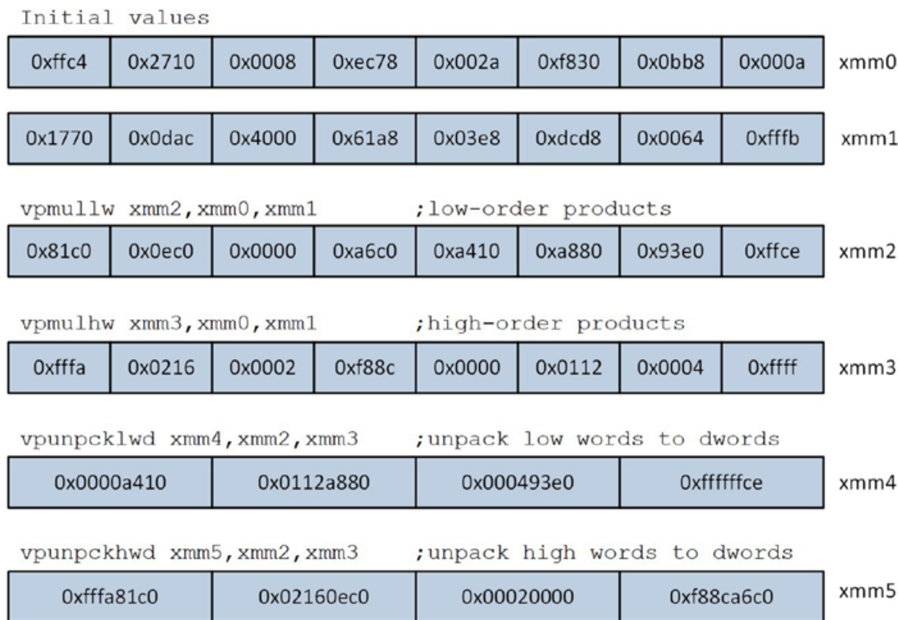



***Figure 7-1.*** *Instruction sequence used in AvxPackedMulI16_ to perform packed 16-bit signed integer multiplication*

The next assembly language function in Listing 7-3, AvxPackedMulI32A_, performs packed signed 32-bit integer multiplication. This function begins with two vmovdqa instructions that load XmmVal argument values a and b into registers XMM0 and XMM1, respectively. The vpmuldq xmm2,xmm0,xmm1 instruction that follows performs packed signed 32-bit multiplication using the even numbered elements of the two source operands. It then saves the signed 64-bit products in XMM2. Two vpsrldq instructions are then used to right shift by four bytes the contents of registers XMM0 and XMM1. This is followed by another vpmuldq instruction that calculates the remaining 64-bit products. Figure 7-2 show the execution details of this instruction sequence.

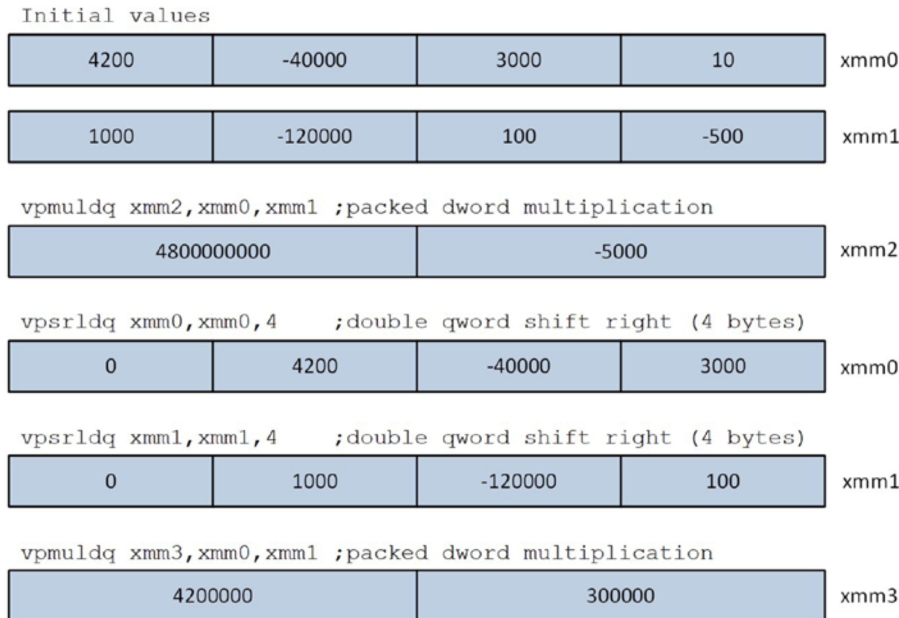



***Figure 7-2.*** *Execution of* `vpmuldq` *and* `vpsrldq` *instructions*

Following the execution of the second `vpmuldq` instruction, registers XMM2 and XMM3 contain the four signed 64-bit products. These values are then saved to the specified destination buffer using a series of `vpextrq` (Extract Quadword) instructions. This instruction copies the quadword element that's specified by the immediate (or second source) operand from the first source operand and saves it to the destination operand. For example, the instruction `vpextrq qword ptr [rcx],xmm2,0` saves the low-order quadword of XMM2 to the memory location specified by RCX. The first source operand of a `vpextrq` instruction must be an XMM register; the destination operand can be a general-purpose register or a memory location. AVX also includes instructions that you can use to extract byte (`vpextrb`), word (`vpextrw`), or doubleword (`vpextrd`) elements.

The final assembly language function in this source code example is named `AvxPackedMulI32B_`. This function also performs packed signed 32-bit integer multiplication but saves truncated 32-bit products. Function `AvxPackedMulI32B_` uses the `vpmulld` instruction that performs element-by-element doubleword multiplication similar to packed addition or subtraction. The low-order 32 bits of each product are then saved to the destination operand. The results for source code example `Ch07_03` are as follows:

```
Results for AvxPackedMulI16
a[0]:       10  b[0]:       -5  c[0][0]:          -50
a[1]:     3000  b[1]:      100  c[0][1]:       300000
a[2]:    -2000  b[2]:    -9000  c[0][2]:     18000000
a[3]:       42  b[3]:     1000  c[0][3]:        42000
a[4]:    -5000  b[4]:    25000  c[1][0]:   -125000000
a[5]:        8  b[5]:    16384  c[1][1]:       131072
a[6]:    10000  b[6]:     3500  c[1][2]:     35000000
a[7]:      -60  b[7]:     6000  c[1][3]:      -360000

----------------------------------------------------------------------------
```

```
Results for AvxPackedMulI32A
a[0]:          10  b[0]:        -500  c[0][0]:           -5000
a[1]:        3000  b[1]:         100  c[0][1]:          300000
a[2]:      -40000  b[2]:     -120000  c[1][0]:      4800000000
a[3]:        4200  b[3]:        1000  c[1][1]:         4200000

---------------------------------------------------------------------------

Results for AvxPackedMulI32B
a[0]:          10  b[0]:        -500  c[0]:       -5000
a[1]:        3000  b[1]:         100  c[1]:      300000
a[2]:       -2000  b[2]:      -12000  c[2]:    24000000
a[3]:        4200  b[3]:        1000  c[3]:     4200000
```

# Packed Integer Image Processing

The source code examples presented thus far were intended to familiarize you with AVX packed integer programming. Each example included a simple assembly language function that demonstrated the operation of several AVX instructions using instances of the structure XmmVal. For some real-world application programs, it may be appropriate to create a small set of functions similar to the ones you've seen thus far. However, in order to fully exploit the benefits of the AVX, you need to code functions that implement complete algorithms using common data structures.

The source code examples in this section present algorithms that process arrays of unsigned 8-bit integers using the AVX instruction set. In the first example, you learn how to determine the minimum and maximum value of an array. This sample program has a certain practicality to it since digital images often use arrays of unsigned 8-bit integers to represent images in memory, and many image-processing algorithms (e.g., contrast enhancement) often need to determine the minimum (darkest) and maximum (lightest) pixels in an image. The second sample program illustrates how to calculate the mean value of an array of unsigned 8-bit integers. This is another example of a realistic algorithm that is directly relevant to the province of image processing. The final three source code examples implement universal image processing algorithms, including pixel conversion, histogram creation, and thresholding.

## Pixel Minimum-Maximum Values

Source code example Ch07_04, shown in Listing 7-4, demonstrates how to find the minimum and maximum values in an array of unsigned 8-bit integers. This example also explains how to dynamically allocate aligned storage space for an array.

***Listing 7-4.*** Example Ch07_04

```
//------------------------------------------------
//               AlignedMem.h
//------------------------------------------------

#pragma once
#include <cstdint>
#include <malloc.h>
#include <stdexcept>
```

```
class AlignedMem
{
public:
    static void* Allocate(size_t mem_size, size_t mem_alignment)
    {
        void* p = _aligned_malloc(mem_size, mem_alignment);

        if (p == NULL)
            throw std::runtime_error("Memory allocation error: AllocateAlignedMem()");

        return p;
    }

    static void Release(void* p)
    {
        _aligned_free(p);
    }

    template <typename T> static bool IsAligned(const T* p, size_t alignment)
    {
        if (p == nullptr)
            return false;

        if (((uintptr_t)p % alignment) != 0)
            return false;

        return true;
    }
};

template <class T> class AlignedArray
{
    T* m_Data;
    size_t m_Size;

public:

    AlignedArray(void) = delete;
    AlignedArray(const AlignedArray& aa) = delete;
    AlignedArray(AlignedArray&& aa) = delete;
    AlignedArray& operator = (const AlignedArray& aa) = delete;
    AlignedArray& operator = (AlignedArray&& aa) = delete;

    AlignedArray(size_t size, size_t alignment)
    {
        m_Size = size;
        m_Data = (T*)AlignedMem::Allocate(size * sizeof(T), alignment);
    }
```

```
    ~AlignedArray()
    {
        AlignedMem::Release(m_Data);
    }

    T* Data(void)               { return m_Data; }
    size_t Size(void)           { return m_Size; }

    void Fill(T val)
    {
        for (size_t i = 0; i < m_Size; i++)
            m_Data[i] = val;
    }
};

//------------------------------------------------
//               Ch07_04.h
//------------------------------------------------

#pragma once
#include <cstdint>

// Ch07_04.cpp
extern void Init(uint8_t* x, size_t n, unsigned int seed);
extern bool AvxCalcMinMaxU8Cpp(const uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_max);

// Ch07_04_BM.cpp
extern void AvxCalcMinMaxU8_BM(void);

// Ch07_04_.asm
extern "C" bool AvxCalcMinMaxU8_(const uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_
max);

// c_NumElements must be > 0 and even multiple of 64
const size_t c_NumElements = 16 * 1024 * 1024;
const unsigned int c_RngSeedVal = 23;

//------------------------------------------------
//               Ch07_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cstdint>
#include <random>
#include "Ch07_04.h"
#include "AlignedMem.h"

using namespace std;

void Init(uint8_t* x, size_t n, unsigned int seed)
```

```
{
    uniform_int_distribution<> ui_dist {5, 250};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (uint8_t)ui_dist(rng);

    // Use known values for min & max (for test purposes)
    x[(n / 4) * 3 + 1] = 2;
    x[n / 4 + 11] = 3;
    x[n / 2] = 252;
    x[n / 2 + 13] = 253;
    x[n / 8 + 5] = 4;
    x[n / 8 + 7] = 254;
}

bool AvxCalcMinMaxU8Cpp(const uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_max)
{
    if (n == 0 || (n & 0x3f) != 0)
        return false;

    if (!AlignedMem::IsAligned(x, 16))
        return false;

    uint8_t x_min_temp = 0xff;
    uint8_t x_max_temp = 0;

    for (size_t i = 0; i < n; i++)
    {
        uint8_t val = *x++;

        if (val < x_min_temp)
            x_min_temp = val;
        else if (val > x_max_temp)
            x_max_temp = val;
    }

    *x_min = x_min_temp;
    *x_max = x_max_temp;
    return true;
}

void AvxCalcMinMaxU8()
{
    size_t n = c_NumElements;
    AlignedArray<uint8_t> x_aa(n, 16);
    uint8_t* x = x_aa.Data();

    Init(x, n, c_RngSeedVal);
```

```
    uint8_t x_min1 = 0, x_max1 = 0;
    uint8_t x_min2 = 0, x_max2 = 0;
    bool rc1 = AvxCalcMinMaxU8Cpp(x, n, &x_min1, &x_max1);
    bool rc2 = AvxCalcMinMaxU8_(x, n, &x_min2, &x_max2);

    cout << "\nResults for AvxCalcMinMaxU8\n";
    cout << "rc1: " << rc1 << "  x_min1: " << (int)x_min1;
    cout << "  x_max1: " << (int)x_max1 << '\n';
    cout << "rc2: " << rc1 << "  x_min2: " << (int)x_min2;
    cout << "  x_max2: " << (int)x_max2 << '\n';
}

int main()
{
    AvxCalcMinMaxU8();
    AvxCalcMinMaxU8_BM();
    return 0;
}

;-------------------------------------------------
;               Ch07_04_.asm
;-------------------------------------------------

; extern "C" bool AvxCalcMinMaxU8_(uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_max)
;
; Returns:      0 = invalid n or unaligned array, 1 = success

            .const
            align 16
StartMinVal qword 0ffffffffffffffffh    ;Initial packed min values
            qword 0ffffffffffffffffh

StartMaxVal qword 0000000000000000h     ;Initial packed max values
            qword 0000000000000000h

            .code
AvxCalcMinMaxU8_ proc

; Make sure 'n' is valid
        xor eax,eax                         ;set error return code
        or rdx,rdx                          ;is n == 0?
        jz Done                             ;jump if yes

        test rdx,3fh                        ;is n a multiple of 64?
        jnz Done                            ;jump if no

        test rcx,0fh                        ;is x properly aligned?
        jnz Done                            ;jump if no

; Initialize packed min-max values
        vmovdqa xmm2,xmmword ptr [StartMinVal]
        vmovdqa xmm3,xmm2                       ;xmm3:xmm2 = packed min values
```

```
        vmovdqa xmm4,xmmword ptr [StartMaxVal]
        vmovdqa xmm5,xmm4                       ;xmm5:xmm4 = packed max values

; Scan array for min & max values
@@:     vmovdqa xmm0,xmmword ptr [rcx]        ;xmm0 = x[i + 15] : x[i]
        vmovdqa xmm1,xmmword ptr [rcx+16]     ;xmm1 = x[i + 31] : x[i + 16]
        vpminub xmm2,xmm2,xmm0
        vpminub xmm3,xmm3,xmm1                ;xmm3:xmm2 = updated min values
        vpmaxub xmm4,xmm4,xmm0
        vpmaxub xmm5,xmm5,xmm1                ;xmm5:xmm4 = updated max values

        vmovdqa xmm0,xmmword ptr [rcx+32]     ;xmm0 = x[i + 47] : x[i + 32]
        vmovdqa xmm1,xmmword ptr [rcx+48]     ;xmm1 = x[i + 63] : x[i + 48]
        vpminub xmm2,xmm2,xmm0
        vpminub xmm3,xmm3,xmm1                ;xmm3:xmm2 = updated min values
        vpmaxub xmm4,xmm4,xmm0
        vpmaxub xmm5,xmm5,xmm1                ;xmm5:xmm4 = updated max values

        add rcx,64
        sub rdx,64
        jnz @B

; Determine final minimum value
        vpminub xmm0,xmm2,xmm3              ;xmm0[127:0] = final 16 min vals
        vpsrldq xmm1,xmm0,8                 ;xmm1[63:0] = xmm0[127:64]
        vpminub xmm2,xmm1,xmm0              ;xmm2[63:0] = final 8 min vals
        vpsrldq xmm3,xmm2,4                 ;xmm3[31:0] = xmm2[63:32]
        vpminub xmm0,xmm3,xmm2              ;xmm0[31:0] = final 4 min vals
        vpsrldq xmm1,xmm0,2                 ;xmm1[15:0] = xmm0[31:16]
        vpminub xmm2,xmm1,xmm0              ;xmm2[15:0] = final 2 min vals
        vpextrw eax,xmm2,0                  ;ax = final 2 min vals
        cmp al,ah
        jbe @F                              ;jump if al <= ah
        mov al,ah                           ;al = final min value
@@:     mov [r8],al                         ;save final min

; Determine final maximum value
        vpmaxub xmm0,xmm4,xmm5              ;xmm0[127:0] = final 16 max vals
        vpsrldq xmm1,xmm0,8                 ;xmm1[63:0] = xmm0[127:64]
        vpmaxub xmm2,xmm1,xmm0              ;xmm2[63:0] = final 8 max vals
        vpsrldq xmm3,xmm2,4                 ;xmm3[31:0] = xmm2[63:32]
        vpmaxub xmm0,xmm3,xmm2              ;xmm0[31:0] = final 4 max vals
        vpsrldq xmm1,xmm0,2                 ;xmm1[15:0] = xmm0[31:16]
        vpmaxub xmm2,xmm1,xmm0              ;xmm2[15:0] = final 2 max vals
        vpextrw eax,xmm2,0                  ;ax = final 2 min vals
        cmp al,ah
        jae @F                              ;jump if al >= ah
        mov al,ah                           ;al = final max value
@@:     mov [r9],al                         ;save final max
```

```
        mov eax,1                          ;set success return code
Done:   ret
AvxCalcMinMaxU8_ endp
        end
```

Listing 7-4 begins with the source code for the header file AlignedMem.h. This file defines a couple of simple C++ classes that facilitate dynamically allocated aligned arrays. The class AlignedMem is a basic wrapper class for the Visual C++ runtime functions _aligned_malloc and _aligned_free. This class also includes a template member function named AlignedMem::IsAligned that validates the alignment of an array in memory. The header file AlignedMem.h also defines a template class named AlignedArray. Class AlignedArray, which is used in this and subsequent source code examples, contains code that implements and manages dynamically allocated aligned arrays. Note that this class contains only minimal functionality to support the source code examples in this book, which is why many of the standard constructors and assignment operators are disabled.

The primary C++ code in example Ch07_04 begins with the definition of a function name Init. This function initializes an array of unsigned 8-bit integers with random values in order to simulate the pixel values of an image. Function Init uses the C++ standard template library (STL) classes uniform_int_distribution and default_random_engine to generate random values for the array. Appendix A contains a list of references that you can consult if you're interested in learning more about these classes. Note that function Init sets some of the pixel values in the target array to know values for test purposes.

The function AvxCalcMinMaxU8Cpp implements a C++ version of the pixel value min-max algorithm. Parameters for this function include a pointer to the array, the number of array elements, and pointers for the minimum and maximum values. The algorithm itself consists of an unsophisticated for loop that sweeps though the array to find the minimum and maximum pixel values. Note that function AvxCalcMinMaxU8Cpp (and its counterpart assembly language function AvxCalcMinMaxU8_) requires the size of the array to be an even multiple of 64. The reason for this is that the assembly language function AvxCalcMinMaxU8_ (arbitrarily) processes 64 pixels during each loop iteration, as you'll soon see. Also note that the source pixel array must be aligned to a 16-byte boundary. The C++ template function AlignedMem::IsAligned performs this check.

The C++ function AvxCalcMinMaxU8 contains code that initializes a test array and exercises the two pixel min-max functions. This function uses the aforementioned template class named AlignedArray to dynamically allocate an array of unsigned 8-bit integers that's aligned to a 16-byte boundary. The constructor arguments for this class include the number of array elements and the alignment boundary. Following the AlignedArray<uint8_t> x_aa(n, 16) statement, AvxCalcMinMaxU8 obtains a raw C++ pointer to the array buffer using the member function AlignedArray::Data(). This pointer is passed as an argument to the two min-max functions.

The assembly language function AvxCalcMinMaxU8_ implements the same algorithm as its C++ counterpart with one significant difference. It processes array elements using 16-byte packets, which is the maximum number of unsigned 8-bit integers that can be stored in an XMM register. The function AvxCalcMinMaxU8_ begins by validating the size of argument n. It then checks array x for proper alignment. Following argument validation, AvxCalcMinMaxU8_ loads register pairs XMM3:XMM2 and XMM5:XMM4 with the initial packed minimum and maximum values, respectively. This enables the processing loop to track 32 min-max values simultaneously.

During each processing loop iteration, the function AvxCalcMinMaxU8_ loads 32 pixel values into register pair XMM1:XMM0 using the instructions vmovdqa xmm0,xmmword ptr [rcx] and vmovdqa xmm1,xmmword ptr [rcx+16]. The next two instructions, vpminub xmm2,xmm2,xmm0 and vpminub xmm3,xmm3,xmm1, update the current pixel minimums in register pair XMM3:XMM2. The ensuing vpmaxub instructions update the current pixel maximums in register pair XMM5:XMM4. Another sequence of vmovdqa, vpminub, and vpmaxub instructions handles the next group of 32 pixels. The processing of multiple data items during each loop iteration reduces the number of executed jump instructions and often results

in faster code. This optimization technique is commonly called loop unrolling (or unwinding). You'll learn more about loop unrolling and jump instruction optimization techniques in Chapter 15.

Following the execution of the pixel value min-max processing loop, the values in register pairs XMM3:XMM2 and XMM5:XMM4 must be reduced in order to obtain the final minimum and maximum values. The `vpminub xmm0,xmm2,xmm3` instruction reduces the number of pixel minimum values from 32 to 16. The next instruction, `vpsrldq xmm1,xmm0,8`, right shifts the contents of XMM0 by eight bytes and saves the result in register XMM1 (i.e., XMM1[63:0] = XMM0[127:64]). This facilitates the use of the subsequent `vpminub xmm2,xmm1,xmm0` instruction that reduces the number of minimum values from 16 to 8. Two more `vpsrldq`-`vpminub` instruction sequences are then employed to reduce the number of pixel minimums to two as shown in Figure 7-3. The `vpextrw eax,xmm2,0` extracts the low-order word (XMM2[15:0]) from register XMM2 and saves it to the low-order word of register EAX (or register AX). The `cmp al,ah`, `jbe`, and `mov al,ah` instructions determine the final pixel minimum value. `AvxCalcMinMaxU8_` uses a similar reduction technique to determine the maximum pixel value.

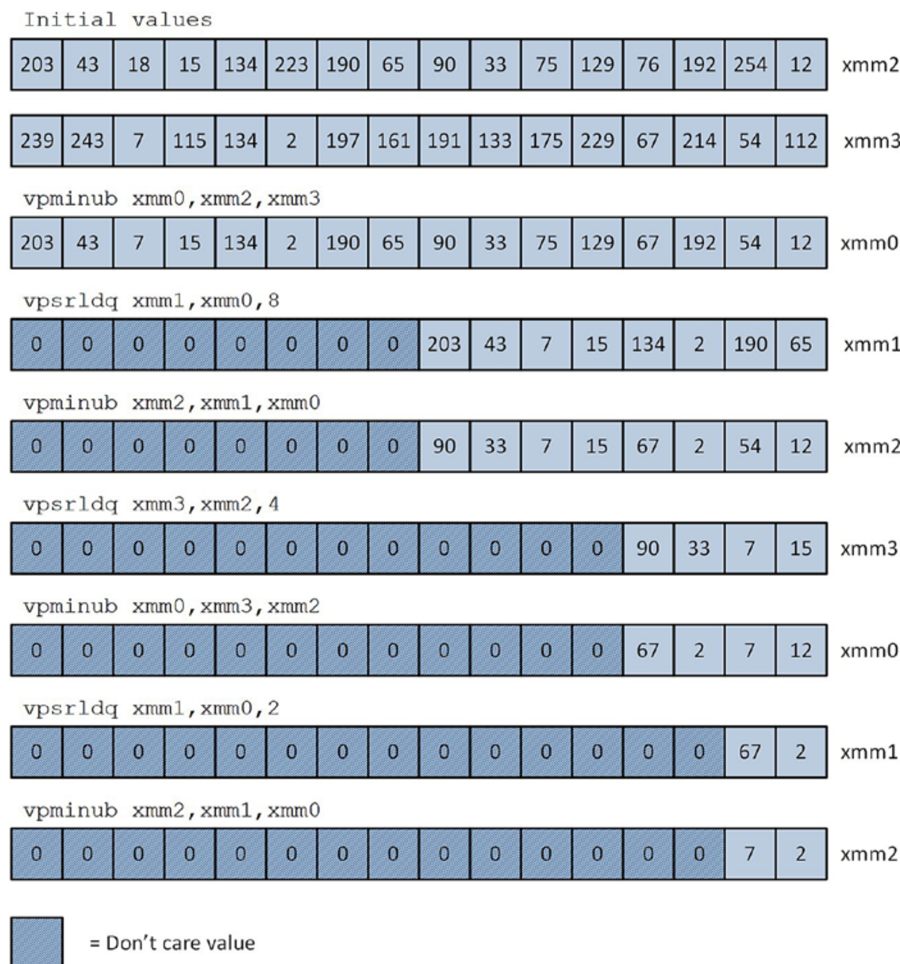



***Figure 7-3.*** *Reduction of pixel minimum values using the instructions* `vpminub` *and* `vpsrldq`

Here is the output for source code example Ch07_04:

```
Results for AvxCalcMinMaxU8
rc1: 1  x_min1: 2  x_max1: 254
rc2: 1  x_min2: 2  x_max2: 254

Running benchmark function AvxCalcMinMaxU8_BM - please wait
Benchmark times save to file Ch07_04_AvxCalcMinMaxU8_BM_CHROMIUM.csv
```

Table 7-1 shows some timing measurements for the functions AvxCalcMinMaxU8 and AvxCalcMinMaxU8_ using several different Intel processors. These measurements were made using the procedure that's described in Chapter 6. The benchmark source code for this and subsequent examples is not shown but included with the chapter download packages.

***Table 7-1.*** *Pixel Value Min-Max Mean Execution Times (Microseconds), Array Size = 16 MB*

| CPU | AvxCalcMinMaxU8Cpp | AvxCalcMinMaxU8_ |
|---|---|---|
| i7-4790S | 17642 | 1007 |
| i9-7900X | 13638 | 874 |
| i7-8700K | 12622 | 721 |

## Pixel Mean Intensity

The next source code example, Ch07_05, contains code that calculates the arithmetic mean of an array of 8-bit unsigned integers. This example also illustrates how to size-promote packed unsigned integers. Listing 7-5 shows the source code for example Ch07_05.

***Listing 7-5.*** Example Ch07_05

```
//------------------------------------------------
//               Ch07_05.h
//------------------------------------------------

#pragma once
#include <cstdint>

// Ch07_05.cpp
extern void Init(uint8_t* x, size_t n, unsigned int seed);
extern bool AvxCalcMeanU8Cpp(const uint8_t* x, size_t n, int64_t* sum_x, double* mean);

// Ch07_05_BM.cpp
extern void AvxCalcMeanU8_BM(void);

// Ch07_05_.asm
extern "C" bool AvxCalcMeanU8_(const uint8_t* x, size_t n, int64_t* sum_x, double* mean);

// Common constants
const size_t c_NumElements = 16 * 1024 * 1024;      // Must be multiple of 64
```

```
const size_t c_NumElementsMax = 64 * 1024 * 1024;   // Used to avoid overflows
const unsigned int c_RngSeedVal = 29;

//------------------------------------------------
//               Ch07_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include "Ch07_05.h"
#include "AlignedMem.h"

using namespace std;

extern "C" size_t g_NumElementsMax = c_NumElementsMax;  // Used in .asm code

void Init(uint8_t* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {0, 255};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (uint8_t)ui_dist(rng);
}

bool AvxCalcMeanU8Cpp(const uint8_t* x, size_t n, int64_t* sum_x, double* mean_x)
{
    if (n == 0 || n > c_NumElementsMax)
        return false;

    if ((n % 64) != 0)
        return false;

    if (!AlignedMem::IsAligned(x, 16))
        return false;

    int64_t sum_x_temp = 0;

    for (int i = 0; i < n; i++)
        sum_x_temp += x[i];

    *sum_x = sum_x_temp;
    *mean_x = (double)sum_x_temp / n;
    return true;
}
```

```
void AvxCalcMeanU8()
{
    const size_t n = c_NumElements;
    AlignedArray<uint8_t> x_aa(n, 16);
    uint8_t* x = x_aa.Data();

    Init(x, n, c_RngSeedVal);

    bool rc1, rc2;
    int64_t sum_x1 = -1, sum_x2 = -1;
    double mean_x1 = -1, mean_x2 = -1;

    rc1 = AvxCalcMeanU8Cpp(x, n, &sum_x1, &mean_x1);
    rc2 = AvxCalcMeanU8_(x, n, &sum_x2, &mean_x2);

    cout << "\nResults for MmxCalcMeanU8\n";
    cout << fixed << setprecision(6);
    cout << "rc1: " << rc1 << "  ";
    cout << "sum_x1: " << sum_x1 << "  ";
    cout << "mean_x1: " << mean_x1 << '\n';
    cout << "rc2: " << rc2 << "  ";
    cout << "sum_x2: " << sum_x2 << "  ";
    cout << "mean_x2: " << mean_x2 << '\n';
}

int main()
{
    AvxCalcMeanU8();
    AvxCalcMeanU8_BM();
    return 0;
}

;-------------------------------------------------
;               Ch07_05.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern g_NumElementsMax:qword

; extern "C" bool AvxCalcMeanU8_(const Uint8* x, size_t n, int64_t* sum_x, double* mean);
;
; Returns       0 = invalid n or unaligned array, 1 = success

        .code
AvxCalcMeanU8_ proc frame
        _CreateFrame CM_,0,64
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog
```

```
; Verify function arguments
        xor eax,eax                         ;set error return code
        or rdx,rdx
        jz Done                             ;jump if n == 0

        cmp rdx,[g_NumElementsMax]
        jae Done                            ;jump if n > NumElementsMax

        test rdx,3fh
        jnz Done                            ;jump if (n % 64) != 0

        test rcx,0fh
        jnz Done                            ;jump if x is not properly aligned

; Perform required initializations
        mov r10,rdx                         ;save n for later use
        add rdx,rcx                         ;rdx = end of array
        vpxor xmm8,xmm8,xmm8                ;xmm8 = packed intermediate sums (4 dwords)
        vpxor xmm9,xmm9,xmm9                ;xmm9 = packed zero for promotions

; Promote 32 pixel values from bytes to words, then sum the words
@@:     vmovdqa xmm0,xmmword ptr [rcx]
        vmovdqa xmm1,xmmword ptr [rcx+16]   ;xmm1:xmm0 = 32 pixels
        vpunpcklbw xmm2,xmm0,xmm9           ;xmm2 = 8 words
        vpunpckhbw xmm3,xmm0,xmm9           ;xmm3 = 8 words
        vpunpcklbw xmm4,xmm1,xmm9           ;xmm4 = 8 words
        vpunpckhbw xmm5,xmm1,xmm9           ;xmm5 = 8 words
        vpaddw xmm0,xmm2,xmm3
        vpaddw xmm1,xmm4,xmm5
        vpaddw xmm6,xmm0,xmm1               ;xmm6 = packed sums (8 words)

; Promote another 32 pixel values from bytes to words, then sum the words
        vmovdqa xmm0,xmmword ptr [rcx+32]
        vmovdqa xmm1,xmmword ptr [rcx+48]   ;xmm1:xmm0 = 32 pixels
        vpunpcklbw xmm2,xmm0,xmm9           ;xmm2 = 8 words
        vpunpckhbw xmm3,xmm0,xmm9           ;xmm3 = 8 words
        vpunpcklbw xmm4,xmm1,xmm9           ;xmm4 = 8 words
        vpunpckhbw xmm5,xmm1,xmm9           ;xmm5 = 8 words
        vpaddw xmm0,xmm2,xmm3
        vpaddw xmm1,xmm4,xmm5
        vpaddw xmm7,xmm0,xmm1               ;xmm7 = packed sums (8 words)

; Promote packed sums to dwords, then update dword sums
        vpaddw xmm0,xmm6,xmm7               ;xmm0 = packed sums (8 words)
        vpunpcklwd xmm1,xmm0,xmm9           ;xmm1 = packed sums (4 dwords)
        vpunpckhwd xmm2,xmm0,xmm9           ;xmm2 = packed sums (4 dwords)
        vpaddd xmm8,xmm8,xmm1
        vpaddd xmm8,xmm8,xmm2
```

```
        add rcx,64                          ;rcx = next 64 byte block
        cmp rcx,rdx
        jne @B                              ;repeat loop if not done

; Compute final sum_x (note vpextrd zero extends extracted dword to 64 bits)
        vpextrd eax,xmm8,0                  ;rax = partial sum 0
        vpextrd edx,xmm8,1                  ;rdx = partial sum 1
        add rax,rdx
        vpextrd ecx,xmm8,2                  ;rcx = partial sum 2
        vpextrd edx,xmm8,3                  ;rdx = partial sum 3
        add rax,rcx
        add rax,rdx
        mov [r8],rax                        ;save sum_x

; Compute mean value
        vcvtsi2sd xmm0,xmm0,rax             ;xmm0 = sum_x (DPFP)
        vcvtsi2sd xmm1,xmm1,r10             ;xmm1 = n (DPFP)
        vdivsd xmm2,xmm0,xmm1               ;calc mean = sum_x / n
        vmovsd real8 ptr [r9],xmm2          ;save mean
        mov eax,1                           ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame
        ret
AvxCalcMeanU8_ endp
        end
```

The organization of the C++ code in example Ch07_05 is somewhat similar to the previous example. The C++ function AvxCalcMeanU8Cpp uses a simple summing loop and scalar arithmetic to calculate the mean of an array of 8-bit unsigned integers. Like the previous example, the number of array elements must be an integral multiple of 64 and the source array must be aligned to a 16-byte boundary. Note that the function AvxCalcMeanU8Cpp also verifies that the number of array elements is not greater than c_NumElementsMax. This size restriction enables the assembly language function AvcCalcMeanU8_ to carry out its calculations using packed doublewords sans any safeguards for arithmetic overflows. The remaining C++ code that's shown in Listing 7-5 performs test array initialization and streams results to cout.

The assembly language function AvxCalcMeanU8_ begins by performing the same validations of the array size as its C++ counterpart. The address of the array is also check for proper alignment. Following argument validation, AvxCalcMeanU8_ carries out its required initializations. The add rdx,rcx instruction computes the address of the first byte beyond the end of the array. The function AvxCalcMeanU8_ uses this address instead of a counter to terminate the processing loop. Register XMM8 is then initialized to all zeros. The processing loop uses this register to maintain intermediate packed doubleword sums.

Each processing loop iteration begins with two vmovdqa instructions that load 32 unsigned byte values into registers XMM1:XMM0. The pixel values are then size-promoted to words using the vpunpcklbw (Unpack Low Data) and vpunpckhbw (Unpack High Data). These instructions interleave the byte values contained in the two source operands to form word values, as shown in Figure 7-4. Note that register XMM9 contains all zeros, which means that the unsigned byte values are zero extended during size promotion to words. A series of vpaddw instructions then sums the packed unsigned word values. The function AvxCalcMeanU8_ processes another block of 32 pixels using the same sequence of instructions. The unsigned word sums in registers XMM6 and XMM7 are then summed using a vpaddw instruction, size-promoted to

doublewords using vpunpcklwd and vpunpckhwd, and added to the intermediate packed doubleword sums in register XMM8. Figure 7-4 illustrates this instruction sequence in greater detail.

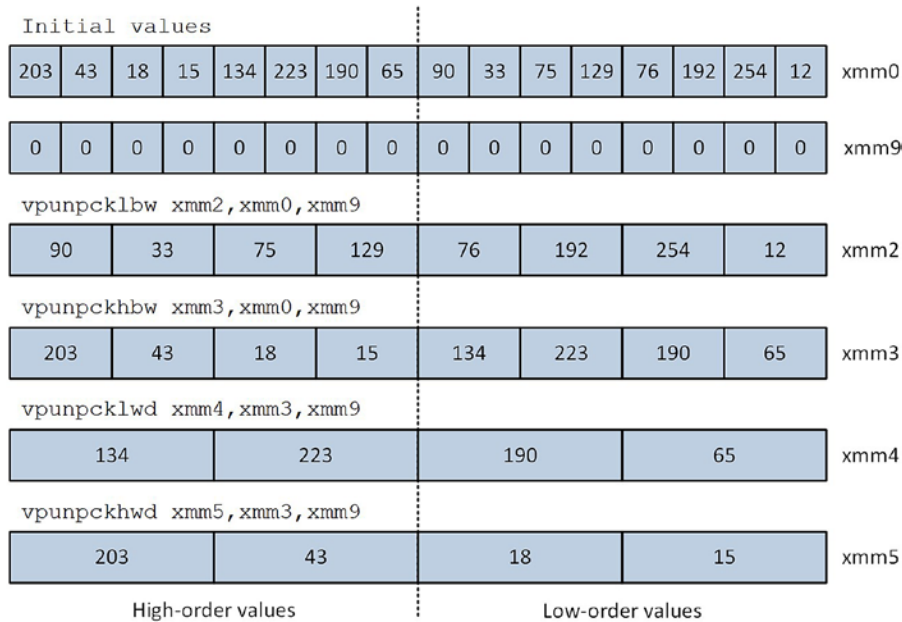



***Figure 7-4.*** *Execution of the vpunpck[h|l]bw, and vpunpck[h|l]wd instructions*

Following termination of the processing loop, the intermediate doubleword sums in register XMM8 are totaled to generate the final pixel sum. The function uses several vpextrd instructions to copy each doubleword value from XMM8 to a general-purpose register. Note that this instruction uses an immediate operand to specify which element value to copy. Following computation of the pixel sum, AvxCalcMeanU8_ calculates the final pixel mean using simple scalar arithmetic. Here are the results for source code example Ch07_05:

```
Results for AvxCalcMeanU8
rc1: 1  sum_x1: 2139023922  mean_x1: 127.495761
rc2: 1  sum_x2: 2139023922  mean_x2: 127.495761

Running benchmark function AvxCalcMeanU8_BM - please wait
Benchmark times save to file Ch07_05_AvxCalcMeanU8_BM_CHROMIUM.csv
```

Table 7-2 shows some benchmark timing measurements for source code example Ch07_05.

***Table 7-2.*** *Source Code Example Ch07_05 Mean Execution Times (Microseconds), Array Size = 16 MB*

| CPU | AvxCalcMeanU8Cpp | AvxCalcMeanU8_ |
|---|---|---|
| i7-4790S | 7103 | 1063 |
| i9-7900X | 6332 | 1048 |
| i7-8700K | 5870 | 861 |

## Pixel Conversions

In order to implement certain image-processing algorithms, it is often necessary to convert the pixels of an 8-bit grayscale image from unsigned integer to single-precision floating-point values and vice versa. The sample code example of this section, Ch07_06, illustrates how to do this using the AVX instruction set. Listing 7-6 shows the source code for example Ch07_06.

***Listing 7-6.*** Example Ch07_06

```
//------------------------------------------------
//               Ch07_06.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>
#include <random>
#include "AlignedMem.h"

using namespace std;

// Ch07_06_Misc.cpp
extern uint32_t ConvertImgVerify(const float* src1, const float* src2, uint32_t num_pixels);
extern uint32_t ConvertImgVerify(const uint8_t* src1, const uint8_t* src2, uint32_t num_
pixels);

// Ch07_06_.asm
extern "C" bool ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_pixels);
extern "C" bool ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_pixels);

extern "C" uint32_t c_NumPixelsMax = 16777216;

template <typename T> void Init(T* x, size_t n, unsigned int seed, T scale)
{
    uniform_int_distribution<> ui_dist {0, 255};
    default_random_engine rng {seed};
```

```
    for (size_t i = 0; i < n; i++)
    {
        T temp = (T)ui_dist(rng);
        x[i] = (scale == 1) ? temp : temp / scale;
    }
}

bool ConvertImgU8ToF32Cpp(float* des, const uint8_t* src, uint32_t num_pixels)
{
    // Make sure num_pixels is valid
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if ((num_pixels % 32) != 0)
        return false;

    // Make sure src and des are aligned to a 16-byte boundary
    if (!AlignedMem::IsAligned(src, 16))
        return false;
    if (!AlignedMem::IsAligned(des, 16))
        return false;

    // Convert the image
    for (uint32_t i = 0; i < num_pixels; i++)
        des[i] = src[i] / 255.0f;

    return true;
}

bool ConvertImgF32ToU8Cpp(uint8_t* des, const float* src, uint32_t num_pixels)
{
    // Make sure num_pixels is valid
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if ((num_pixels % 32) != 0)
        return false;

    // Make sure src and des are aligned to a 16-byte boundary
    if (!AlignedMem::IsAligned(src, 16))
        return false;
    if (!AlignedMem::IsAligned(des, 16))
        return false;

    for (uint32_t i = 0; i < num_pixels; i++)
    {
        if (src[i] > 1.0f)
            des[i] = 255;
        else if (src[i] < 0.0)
            des[i] = 0;
        else
            des[i] = (uint8_t)(src[i] * 255.0f);
    }
```

```
    return true;
}

void ConvertImgU8ToF32(void)
{
    const uint32_t num_pixels = 1024;
    AlignedArray<uint8_t> src_aa(num_pixels, 16);
    AlignedArray<float> des1_aa(num_pixels, 16);
    AlignedArray<float> des2_aa(num_pixels, 16);
    uint8_t* src = src_aa.Data();
    float* des1 = des1_aa.Data();
    float* des2 = des2_aa.Data();

    Init(src, num_pixels, 12, (uint8_t)1);

    bool rc1 = ConvertImgU8ToF32Cpp(des1, src, num_pixels);
    bool rc2 = ConvertImgU8ToF32_(des2, src, num_pixels);

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    uint32_t num_diff = ConvertImgVerify(des1, des2, num_pixels);
    cout << "\nResults for ConvertImgU8ToF32\n";
    cout << "  num_pixels = " << num_pixels << '\n';
    cout << "  num_diff = " << num_diff << '\n';
}

void ConvertImgF32ToU8(void)
{
    const uint32_t num_pixels = 1024;
    AlignedArray<float> src_aa(num_pixels, 16);
    AlignedArray<uint8_t> des1_aa(num_pixels, 16);
    AlignedArray<uint8_t> des2_aa(num_pixels, 16);
    float* src = src_aa.Data();
    uint8_t* des1 = des1_aa.Data();
    uint8_t* des2 = des2_aa.Data();

    // Initialize the src pixel buffer. The first few entries in src
    // are set to known values for test purposes.

    Init(src, num_pixels, 20, 1.0f);

    src[0] = 0.125f;        src[8] = 0.01f;
    src[1] = 0.75f;         src[9] = 0.99f;
    src[2] = -4.0f;         src[10] = 1.1f;
    src[3] = 3.0f;          src[11] = -1.1f;
```

```
    src[4] = 0.0f;          src[12] = 0.99999f;
    src[5] = 1.0f;          src[13] = 0.5f;
    src[6] = -0.01f;        src[14] = -0.0;
    src[7] = 1.01f;         src[15] = .333333f;

    bool rc1 = ConvertImgF32ToU8Cpp(des1, src, num_pixels);
    bool rc2 = ConvertImgF32ToU8_(des2, src, num_pixels);

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    uint32_t num_diff = ConvertImgVerify(des1, des2, num_pixels);
    cout << "\nResults for ConvertImgF32ToU8\n";
    cout << "  num_pixels = " << num_pixels << '\n';
    cout << "  num_diff = " << num_diff << '\n';
}

int main()
{
    ConvertImgU8ToF32();
    ConvertImgF32ToU8();
    return 0;
}

;-------------------------------------------------
;               Ch07_06.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        include <cmpequ.asmh>

                    .const
                    align 16
Uint8ToFloat        real4 255.0, 255.0, 255.0, 255.0
FloatToUint8Min     real4 0.0, 0.0, 0.0, 0.0
FloatToUint8Max     real4 1.0, 1.0, 1.0, 1.0
FloatToUint8Scale   real4 255.0, 255.0, 255.0, 255.0

        extern c_NumPixelsMax:dword

; extern "C" bool ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_pixels)

        .code
ConvertImgU8ToF32_ proc frame
        _CreateFrame U2F_,0,160
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog
```

```
; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                         ;set error return code
        or r8d,r8d
        jz Done                             ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                             ;jump if num_pixels too big
        test r8d,1fh
        jnz Done                            ;jump if num_pixels % 32 != 0
        test rcx,0fh
        jnz Done                            ;jump if des not aligned
        test rdx,0fh
        jnz Done                            ;jump if src not aligned

; Initialize processing loop registers
        shr r8d,5                               ;number of pixel blocks
        vmovaps xmm6,xmmword ptr [Uint8ToFloat] ;xmm6 = packed 255.0f
        vpxor xmm7,xmm7,xmm7                    ;xmm7 = packed 0

; Load the next block of 32 pixels
@@:     vmovdqa xmm0,xmmword ptr [rdx]      ;xmm0 = 16 pixels (x[i+15]:x[i])
        vmovdqa xmm1,xmmword ptr [rdx+16]   ;xmm8 = 16 pixels (x[i+31]:x[i+16])

; Promote the pixel values in xmm0 from unsigned bytes to unsigned dwords
        vpunpcklbw xmm2,xmm0,xmm7
        vpunpckhbw xmm3,xmm0,xmm7
        vpunpcklwd xmm8,xmm2,xmm7
        vpunpckhwd xmm9,xmm2,xmm7
        vpunpcklwd xmm10,xmm3,xmm7
        vpunpckhwd xmm11,xmm3,xmm7          ;xmm11:xmm8 = 16 dword pixels

; Promote the pixel values in xmm1 from unsigned bytes to unsigned dwords
        vpunpcklbw xmm2,xmm1,xmm7
        vpunpckhbw xmm3,xmm1,xmm7
        vpunpcklwd xmm12,xmm2,xmm7
        vpunpckhwd xmm13,xmm2,xmm7
        vpunpcklwd xmm14,xmm3,xmm7
        vpunpckhwd xmm15,xmm3,xmm7          ;xmm15:xmm12 = 16 dword pixels

; Convert pixel values from dwords to SPFP
        vcvtdq2ps xmm8,xmm8
        vcvtdq2ps xmm9,xmm9
        vcvtdq2ps xmm10,xmm10
        vcvtdq2ps xmm11,xmm11               ;xmm11:xmm8 = 16 SPFP pixels

        vcvtdq2ps xmm12,xmm12
        vcvtdq2ps xmm13,xmm13
        vcvtdq2ps xmm14,xmm14
        vcvtdq2ps xmm15,xmm15               ;xmm15:xmm12 = 16 SPFP pixels
```

```
; Normalize all pixel values to [0.0, 1.0] and save the results
        vdivps xmm0,xmm8,xmm6
        vmovaps xmmword ptr [rcx],xmm0      ;save pixels 0 - 3
        vdivps xmm1,xmm9,xmm6
        vmovaps xmmword ptr [rcx+16],xmm1   ;save pixels 4 - 7
        vdivps xmm2,xmm10,xmm6
        vmovaps xmmword ptr [rcx+32],xmm2   ;save pixels 8 - 11
        vdivps xmm3,xmm11,xmm6
        vmovaps xmmword ptr [rcx+48],xmm3   ;save pixels 12 - 15

        vdivps xmm0,xmm12,xmm6
        vmovaps xmmword ptr [rcx+64],xmm0   ;save pixels 16 - 19
        vdivps xmm1,xmm13,xmm6
        vmovaps xmmword ptr [rcx+80],xmm1   ;save pixels 20 - 23
        vdivps xmm2,xmm14,xmm6
        vmovaps xmmword ptr [rcx+96],xmm2   ;save pixels 24 - 27
        vdivps xmm3,xmm15,xmm6
        vmovaps xmmword ptr [rcx+112],xmm3  ;save pixels 28 - 31

        add rdx,32                          ;update src ptr
        add rcx,128                         ;update des ptr
        sub r8d,1
        jnz @B                              ;repeat until done
        mov eax,1                           ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret

ConvertImgU8ToF32_ endp

; extern "C" bool ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_pixels)

ConvertImgF32ToU8_ proc frame
        _CreateFrame F2U_,0,96
        _SaveXmmRegs xmm6,xmm7,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                         ;set error return code
        or r8d,r8d
        jz Done                             ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                             ;jump if num_pixels too big
        test r8d,1fh
        jnz Done                            ;jump if num_pixels % 32 != 0
        test rcx,0fh
        jnz Done                            ;jump if des not aligned
        test rdx,0fh
        jnz Done                            ;jump if src not aligned
```

```
; Load required packed constants into registers
        vmovaps xmm13,xmmword ptr [FloatToUint8Scale] ;xmm13 = packed 255.0
        vmovaps xmm14,xmmword ptr [FloatToUint8Min]   ;xmm14 = packed 0.0
        vmovaps xmm15,xmmword ptr [FloatToUint8Max]   ;xmm15 = packed 1.0

        shr r8d,4                           ;number of pixel blocks
LP1:    mov r9d,4                           ;num pixel quartets per block

; Convert 16 float pixels to uint8_t
LP2:    vmovaps xmm0,xmmword ptr [rdx]      ;xmm0 = next pixel quartet
        vcmpps xmm1,xmm0,xmm14,CMP_LT       ;compare pixels to 0.0
        vandnps xmm2,xmm1,xmm0              ;clip pixels < 0.0 to 0.0

        vcmpps xmm3,xmm2,xmm15,CMP_GT       ;compare pixels to 1.0
        vandps xmm4,xmm3,xmm15              ;clip pixels > 1.0 to 1.0
        vandnps xmm5,xmm3,xmm2              ;xmm5 = pixels <= 1.0
        vorps xmm6,xmm5,xmm4                ;xmm6 = final clipped pixels
        vmulps xmm7,xmm6,xmm13              ;xmm7 = FP pixels [0.0, 255.0]

        vcvtps2dq xmm0,xmm7                 ;xmm0 = dword pixels [0, 255]
        vpackusdw xmm1,xmm0,xmm0            ;xmm1[63:0] = word pixels
        vpackuswb xmm2,xmm1,xmm1            ;xmm2[31:0] = bytes pixels

; Save the current byte pixel quartet
        vpextrd eax,xmm2,0                  ;eax = new pixel quartet
        vpsrldq xmm12,xmm12,4               ;adjust xmm12 for new quartet
        vpinsrd xmm12,xmm12,eax,3           ;xmm12[127:96] = new quartet

        add rdx,16                          ;update src ptr
        sub r9d,1
        jnz LP2                             ;repeat until done

; Save the current byte pixel block (16 pixels)
        vmovdqa xmmword ptr [rcx],xmm12     ;save current pixel block
        add rcx,16                          ;update des ptr
        sub r8d,1
        jnz LP1                             ;repeat until done
        mov eax,1                           ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret
ConvertImgF32ToU8_ endp
        end
```

The C++ code in Listing 7-6 is straightforward. The function ConvertImgU8ToF32Cpp contains code that converts pixel values from uint8_t [0, 255] to single-precision floating-point [0.0, 1.0]. This function contains a simple for loop that calculates des[i] = src[i] / 255.0. The counterpart function ConvertImgF32ToU8Cpp performs the inverse operation. Note that this function clips any pixel values greater than 1.0 or less than 0.0 before performing the floating-point to uint8_t conversion. The functions ConvertImgU8ToF32 and ConvertImgF32ToU8 contain code that initialize test arrays and exercise the C++ and assembly language conversion routines. Note that the latter function initializes the first few entries of the source buffer to known values in order to demonstrate the aforementioned clipping operation.

The processing loop of the assembly language function ConvertImgU8ToF32_ converts 32 pixels from uint8_t (or byte) to single-precision floating-point during each iteration. The conversion technique begins with the size promotion of packed pixels from unsigned byte to unsigned doubleword integers using a series of vpunpck[h|l]bw and vpunpck[h|l]wd instructions. The doubleword values are then converted to single-precision floating-point values using the instruction vcvtdq2ps (Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values). The resultant packed floating-point values are normalized to [0.0, 1.0] and saved to the destination buffer.

The assembly language function ConvertImgF32ToU8_ performs packed single-precision floating-point to packed unsigned byte conversions. The inner loop (starting at the label LP2) of this conversion function uses the instructions vcmpps xmm1,xmm0,xmm14,CMP_LT, vcmpps xmm3,xmm2,xmm15,CMP_GT, and some Boolean logic to clip any pixels values less than 0.0 or greater than 1.0. Figure 7-5 illustrates this technique in greater detail. The vcvtps2dq xmm0,xmm7 instruction converts the four single-precision floating-point values in XMM7 to doubleword integers and saves the results in register XMM0. The next two instructions, vpackusdw xmm1,xmm0,xmm0 and vpackuswb xmm2,xmm1,xmm1, size-reduce the packed doubleword integers to packed unsigned bytes. Following the execution of the vpackuswb instruction, register XMM2[31:0] contains four packed unsigned byte values. This pixel quartet is then copied to XMM12[127:96] using the instruction sequence vpextrd eax,xmm2,0, vpsrldq xmm12,xmm12,4, and vpinsrd xmm12,xmm12,eax,3. The vpinsrd (Insert Dword) instructions that's used here copies the doubleword value in register EAX to doubleword element position 3 in register XMM12 (or XMM12[127:96]).

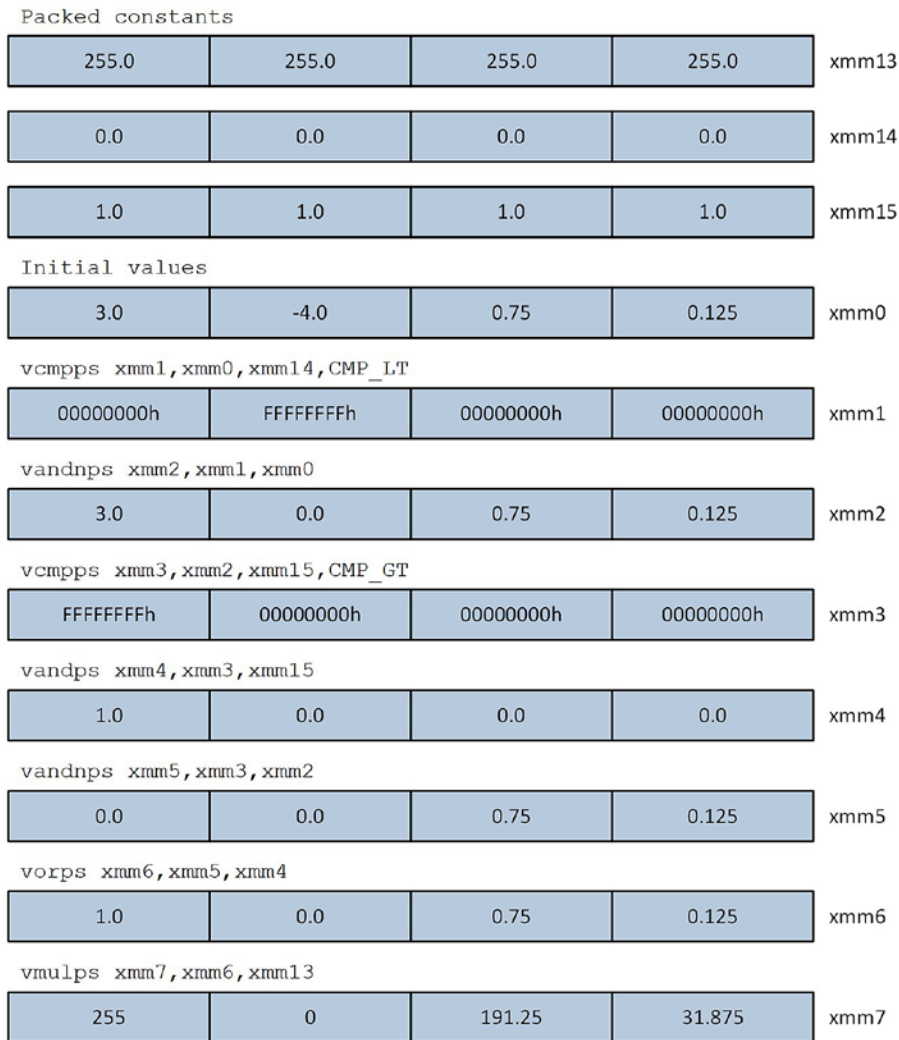



***Figure 7-5.*** *Illustration of floating-point clipping technique used in function* `ConvertImgF32ToU8_`

The inner loop conversion process that's described in the previous paragraph executes for four iterations. Following the completion of the inner loop, XMM12 contains 16 unsigned byte pixel values. This pixel block is then saved to the destination buffer using a `vmovdqa xmmword ptr [rcx],xmm12` instruction. The outer loop repeats until all pixels have been converted. Here is the output for source code example `Ch07_06`.

```
Results for ConvertImgU8ToF32
  num_pixels = 1024
  num_diff = 0

Results for ConvertImgF32ToU8
  num_pixels = 1024
  num_diff = 0
```

## Image Histograms

Many image-processing algorithms require a histogram of an image's pixel intensity values. Figure 7-6 shows a sample grayscale image and its histogram. The next source code example, Ch07_07, illustrates how to build a histogram of pixel intensity values for an image containing 8-bit grayscale pixel values. This example also explains how to use the stack in an assembly language function to store intermediate results. Listing 7-7 shows the source code for example Ch07_07.

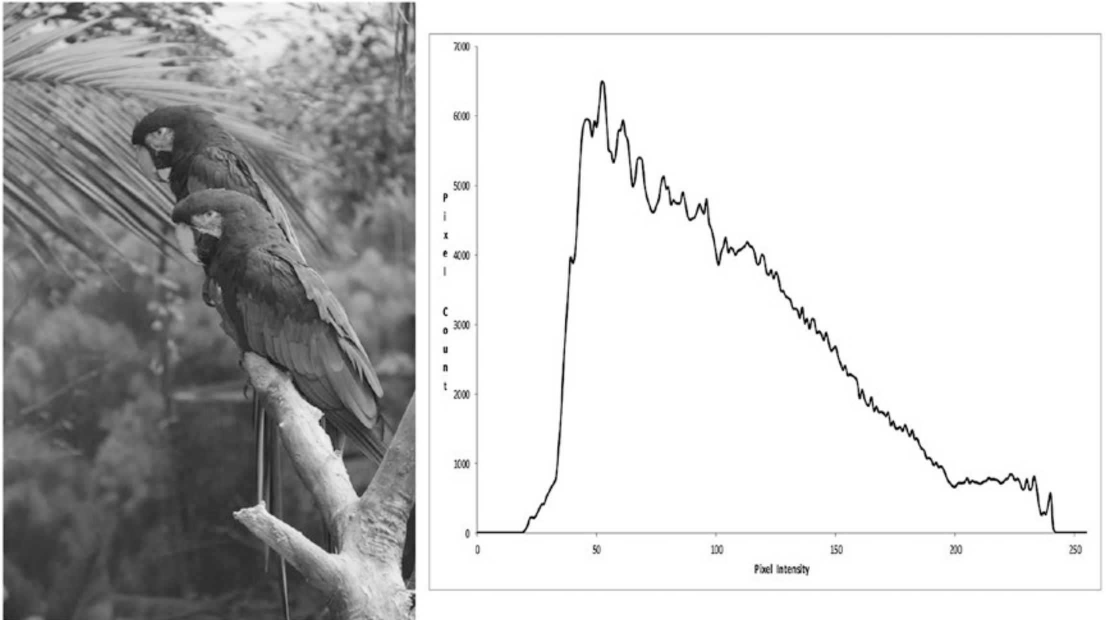



***Figure 7-6.** Sample grayscale image and its histogram*

***Listing 7-7.*** Example Ch07_07

```
//------------------------------------------------
//               Ch07_07.h
//------------------------------------------------

#pragma once
#include <cstdint>

// Ch07_07.cpp
extern bool AvxBuildImageHistogramCpp(uint32_t* histo, const uint8_t* pixel_buff, uint32_t
num_pixels);

// Ch07_07_.asm
// Functions defined in Sse64ImageHistogram_.asm
extern "C" bool AvxBuildImageHistogram_(uint32_t* histo, const uint8_t* pixel_buff, uint32_t
num_pixels);

// Ch07_07_BM.cpp
extern void AvxBuildImageHistogram_BM(void);

//------------------------------------------------
//               Ch07_07.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "Ch07_07.h"
#include "AlignedMem.h"
#include "ImageMatrix.h"

using namespace std;

extern "C" uint32_t c_NumPixelsMax = 16777216;

bool AvxBuildImageHistogramCpp(uint32_t* histo, const uint8_t* pixel_buff, uint32_t num_
pixels)
{
    // Make sure num_pixels is valid
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;

    if (num_pixels % 32 != 0)
        return false;

    // Make sure histo is aligned to a 16-byte boundary
    if (!AlignedMem::IsAligned(histo, 16))
        return false;
```

```
    // Make sure pixel_buff is aligned to a 16-byte boundary
    if (!AlignedMem::IsAligned(pixel_buff, 16))
        return false;

    // Build the histogram
    memset(histo, 0, 256 * sizeof(uint32_t));

    for (uint32_t i = 0; i < num_pixels; i++)
        histo[pixel_buff[i]]++;

    return true;
}

void AvxBuildImageHistogram(void)
{
    const wchar_t* image_fn = L"..\\Ch07_Data\\TestImage1.bmp";
    const wchar_t* csv_fn = L"Ch07_07_AvxBuildImageHistogram_Histograms.csv";

    ImageMatrix im(image_fn);
    uint32_t num_pixels = im.GetNumPixels();
    uint8_t* pixel_buff = im.GetPixelBuffer<uint8_t>();
    AlignedArray<uint32_t> histo1_aa(256, 16);
    AlignedArray<uint32_t> histo2_aa(256, 16);

    bool rc1 = AvxBuildImageHistogramCpp(histo1_aa.Data(), pixel_buff, num_pixels);
    bool rc2 = AvxBuildImageHistogram_(histo2_aa.Data(), pixel_buff, num_pixels);

    cout << "\nResults for AvxBuildImageHistogram\n";

    if (!rc1 || !rc2)
    {
        cout << "Bad return code: ";
        cout << "rc1 = " << rc1 << ", rc2 = " << rc2 << '\n';
        return;
    }

    ofstream ofs(csv_fn);

    if (ofs.bad())
        cout << "File create error - " << csv_fn << '\n';
    else
    {
        bool compare_error = false;
        uint32_t* histo1 = histo1_aa.Data();
        uint32_t* histo2 = histo2_aa.Data();
        const char* delim = ", ";

        for (uint32_t i = 0; i < 256; i++)
        {
            ofs << i << delim;
            ofs << histo1[i] << delim << histo2[i] << '\n';
```

```
            if (histo1[i] != histo2[i])
            {
                compare_error = true;
                cout << "  Histogram compare error at index " << i << '\n';
                cout << "  counts: " << histo1[i] << delim << histo2[i] << '\n';
            }
        }

        if (!compare_error)
            cout << "  Histograms are identical\n";

        ofs.close();
    }
}

int main()
{
    try
    {
        AvxBuildImageHistogram();
        AvxBuildImageHistogram_BM();
    }

    catch (...)
    {
        cout << "Unexpected exception has occurred\n";
        cout << "File = " << __FILE__ << '\n';
    }

    return 0;
}

;-------------------------------------------------
;               Ch07_07.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; extern bool AvxBuildImageHistogram_(uint32_t* histo, const uint8_t* pixel_buff, uint32_t
num_pixels)
;
; Returns:      0 = invalid argument value, 1 = success

        .code
        extern c_NumPixelsMax:dword

AvxBuildImageHistogram_ proc frame
        _CreateFrame BIH_,1024,0,rbx,rsi,rdi
        _EndProlog

; Make sure num_pixels is valid
```

```
        xor eax,eax                     ;set error code
        test r8d,r8d
        jz Done                         ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                         ;jump if num_pixels too big
        test r8d,1fh
        jnz Done                        ;jump if num_pixels % 32 != 0

; Make sure histo & pixel_buff are properly aligned
        mov rsi,rcx                     ;rsi = ptr to histo
        test rsi,0fh
        jnz Done                        ;jump if histo misaligned
        mov r9,rdx
        test r9,0fh
        jnz Done                        ;jump if pixel_buff misaligned

; Initialize local histogram buffers (set all entries to zero)
        xor eax,eax
        mov rdi,rsi                     ;rdi = ptr to histo
        mov rcx,128                     ;rcx = size in qwords
        rep stosq                       ;zero histo
        mov rdi,rbp                     ;rdi = ptr to histo2
        mov rcx,128                     ;rcx = size in qwords
        rep stosq                       ;zero histo2

; Perform processing loop initializations
        shr r8d,5                       ;number of pixel blocks (32 pixels/block)
        mov rdi,rbp                     ;ptr to histo2

; Build the histograms
        align 16                        ;align jump target
@@:     vmovdqa xmm0,xmmword ptr [r9]   ;load pixel block
        vmovdqa xmm1,xmmword ptr [r9+16] ;load pixel block

; Process pixels 0 - 3
        vpextrb rax,xmm0,0
        add dword ptr [rsi+rax*4],1     ;count pixel 0
        vpextrb rbx,xmm0,1
        add dword ptr [rdi+rbx*4],1     ;count pixel 1
        vpextrb rcx,xmm0,2
        add dword ptr [rsi+rcx*4],1     ;count pixel 2
        vpextrb rdx,xmm0,3
        add dword ptr [rdi+rdx*4],1     ;count pixel 3

; Process pixels 4 - 7
        vpextrb rax,xmm0,4
        add dword ptr [rsi+rax*4],1     ;count pixel 4
        vpextrb rbx,xmm0,5
        add dword ptr [rdi+rbx*4],1     ;count pixel 5
        vpextrb rcx,xmm0,6
        add dword ptr [rsi+rcx*4],1     ;count pixel 6
```

```
        vpextrb rdx,xmm0,7
        add dword ptr [rdi+rdx*4],1         ;count pixel 7

; Process pixels 8 - 11
        vpextrb rax,xmm0,8
        add dword ptr [rsi+rax*4],1         ;count pixel 8
        vpextrb rbx,xmm0,9
        add dword ptr [rdi+rbx*4],1         ;count pixel 9
        vpextrb rcx,xmm0,10
        add dword ptr [rsi+rcx*4],1         ;count pixel 10
        vpextrb rdx,xmm0,11
        add dword ptr [rdi+rdx*4],1         ;count pixel 11

; Process pixels 12 - 15
        vpextrb rax,xmm0,12
        add dword ptr [rsi+rax*4],1         ;count pixel 12
        vpextrb rbx,xmm0,13
        add dword ptr [rdi+rbx*4],1         ;count pixel 13
        vpextrb rcx,xmm0,14
        add dword ptr [rsi+rcx*4],1         ;count pixel 14
        vpextrb rdx,xmm0,15
        add dword ptr [rdi+rdx*4],1         ;count pixel 15

; Process pixels 16 - 19
        vpextrb rax,xmm1,0
        add dword ptr [rsi+rax*4],1         ;count pixel 16
        vpextrb rbx,xmm1,1
        add dword ptr [rdi+rbx*4],1         ;count pixel 17
        vpextrb rcx,xmm1,2
        add dword ptr [rsi+rcx*4],1         ;count pixel 18
        vpextrb rdx,xmm1,3
        add dword ptr [rdi+rdx*4],1         ;count pixel 19

; Process pixels 20 - 23
        vpextrb rax,xmm1,4
        add dword ptr [rsi+rax*4],1         ;count pixel 20
        vpextrb rbx,xmm1,5
        add dword ptr [rdi+rbx*4],1         ;count pixel 21
        vpextrb rcx,xmm1,6
        add dword ptr [rsi+rcx*4],1         ;count pixel 22
        vpextrb rdx,xmm1,7
        add dword ptr [rdi+rdx*4],1         ;count pixel 23

; Process pixels 24 - 27
        vpextrb rax,xmm1,8
        add dword ptr [rsi+rax*4],1         ;count pixel 24
        vpextrb rbx,xmm1,9
        add dword ptr [rdi+rbx*4],1         ;count pixel 25
        vpextrb rcx,xmm1,10
        add dword ptr [rsi+rcx*4],1         ;count pixel 26
        vpextrb rdx,xmm1,11
        add dword ptr [rdi+rdx*4],1         ;count pixel 27
```

```
; Process pixels 28 - 31
        vpextrb rax,xmm1,12
        add dword ptr [rsi+rax*4],1         ;count pixel 28
        vpextrb rbx,xmm1,13
        add dword ptr [rdi+rbx*4],1         ;count pixel 29
        vpextrb rcx,xmm1,14
        add dword ptr [rsi+rcx*4],1         ;count pixel 30
        vpextrb rdx,xmm1,15
        add dword ptr [rdi+rdx*4],1         ;count pixel 31

        add r9,32                           ;r9  = next pixel block
        sub r8d,1
        jnz @B                              ;repeat loop if not done

; Merge intermediate histograms into final histogram
        mov ecx,32                          ;ecx = num iterations
        xor eax,eax                         ;rax = common offset

@@:     vmovdqa xmm0,xmmword ptr [rsi+rax]          ;load histo counts
        vmovdqa xmm1,xmmword ptr [rsi+rax+16]
        vpaddd xmm0,xmm0,xmmword ptr [rdi+rax]      ;add counts from histo2
        vpaddd xmm1,xmm1,xmmword ptr [rdi+rax+16]
        vmovdqa xmmword ptr [rsi+rax],xmm0          ;save final result
        vmovdqa xmmword ptr [rsi+rax+16],xmm1

        add rax,32
        sub ecx,1
        jnz @B
        mov eax,1                           ;set success return code

Done:  _DeleteFrame rbx,rsi,rdi
        ret
AvxBuildImageHistogram_ endp
        end
```

Near the top of the C++ code is a function named AvxBuildImageHistogramCpp. This function constructs an image histogram using a rudimentary technique. Prior to the histogram's actual construction, the number of image pixels is validated for size (greater than 0 and not greater than c_NumPixelMax) and divisibility by 32. The divisibility test is performed to ensure compatibility with the assembly language function AvxBuildImageHistogram_. Next, the addresses of histo and pixel_buff are verified for proper alignment. The call to memset initializes each histogram pixel count bin to zero. A simple for loop is then used to construct the histogram.

The function AvxBuildImageHistogram uses a C++ class named ImageMatrix to load the pixels of an image into memory. (The source code for ImageMatrix is not shown but included as part of the chapter download package.) The variables num_pixels and pixel_buff are then initialized using the member functions ImageMatrix::GetNumPixels and ImageMatrix::GetPixelBuffer. Two histogram buffers then are allocated using the C++ template class AlignedArray<uint32_t>. Following the construction of the histograms using the functions AvxBuildImageHistogramCpp and AvxBuildImageHistogram_, the pixel counts in the two histogram buffers are compared for equivalence and written to a comma-separated-value text file.

The assembly language function AvxBuildImageHistogram_ constructs an image histogram using the AVX instruction set. In order to improve performance, this function builds two intermediate histograms and merges them into a final histogram. AvxBuildImageHistogram_ begins by creating a stack frame using the _CreateFrame macro. Note that the stack frame created by _CreateFrame includes 1024 bytes (256 doublewords, one for each grayscale intensity level) of local storage space, which is used for one of the intermediate histogram buffers. Following the execution of the code generated by _CreateFrame, register RBP points to the intermediate histogram on the stack (see Figure 5-6). The caller-provided buffer histo is used as the second intermediate histogram buffer. Following the _EndProlog macro, the function AvxBuildImageHistogram_ validates num_pixels for size and divisibility by 32; it the checks the addresses of histo and pixel_buff for proper alignment. The count values in both intermediate histograms are then initialized to zero using the stosq instruction.

The main processing loop begins with two vmovdqa instructions that load 32 image pixels into registers XMM1:XMM0. Note that prior to the first vmovdqa instruction, the MASM directive align 16 is used to align this instruction on a 16-byte boundary. Aligning the target of a jump instruction on a 16-byte boundary is an optimization technique that often improves performance. Chapter 15 discusses this and other optimization techniques in greater detail. Next, a vpextrb rax,xmm0,0 instruction extracts pixel element 0 (i.e., XMM0[7:0]) from register XMM0 and copies it to the low-order bits of register RAX; the high-order bits of RAX are set to zero. The ensuing add dword ptr [rsi+rax*4],1 instruction updates the appropriate pixel count bin in the first intermediate histogram. The next two instructions, vpextrb rbx,xmm0,1 and add dword ptr [rdi+rbx*4],1, process pixel element 1 in the same manner using the second intermediate histogram. This pixel-processing technique is then repeated for the remaining pixels in the current block.

Following the execution of the processing loop, the pixel count values in the two intermediate histograms are summed using packed integer arithmetic to create the final histogram. The _DeleteFrame macro is then used to release the local stack frame and restore the previously-saved non-volatile general-purpose registers. Here is the output for source code example Ch07_07:

```
Results for AvxBuildImageHistogram
  Histograms are identical

Running benchmark function AvxBuildImageHistogram_BM - please wait
Benchmark times save to file Ch07_07_AvxBuildImageHistogram_BM_CHROMIUM.csv
```

Table 7-3 shows benchmark timing measurements for the histogram build functions.

***Table 7-3.*** *Histogram Build Mean Execution Times (Microseconds) Using TestImage1.bmp*

| CPU | AvxBuildImageHistogramCpp | AvxBuildImageHistogram_ |
|---|---|---|
| i7-4790S | 277 | 230 |
| i9-7900X | 255 | 199 |
| i7-8700K | 241 | 191 |

## Image Thresholding

Image thresholding is an image-processing technique that creates a binary image (i.e., an image with only two colors) from a grayscale image. This binary (or mask) image signifies which pixels in the original image are greater than a predetermined or algorithmically derived intensity threshold value. Figure 7-7 illustrates a thresholding operation. Mask images are often employed to perform additional calculations using the grayscale pixels values of the original image. For example, one typical use of the mask image that's shown

in Figure 7-7 is to compute the mean intensity value of all above-threshold pixels in the original image. The application of a mask image simplifies calculating the mean since it facilitates the use of simple Boolean expressions to exclude unwanted pixels from the computations.

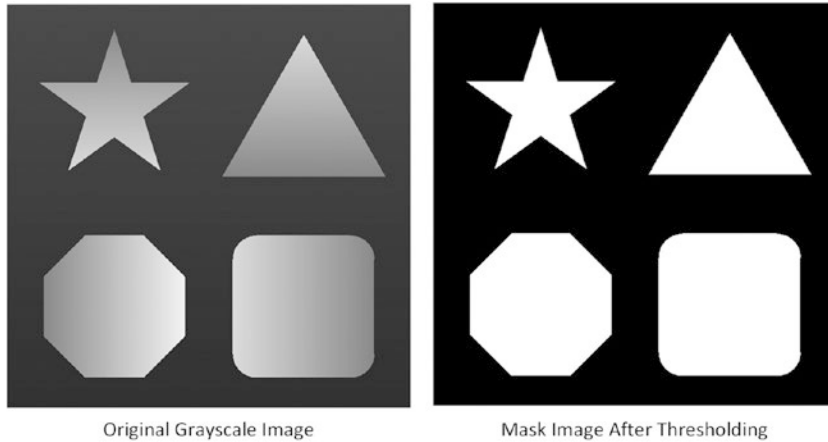



***Figure 7-7.*** *Sample grayscale and mask images*

Source code example Ch07_08 demonstrates how to calculate the mean intensity of image pixels above a specified threshold. It also shows how to call a C++ function from an assembly language function. Listing 7-8 shows the source code for example Ch07_08.

***Listing 7-8.*** Example Ch07_08

```
//------------------------------------------------
//               Ch07_08.h
//------------------------------------------------

#pragma once
#include <cstdint>

// Image threshold data structure. This structure must agree with the
// structure that's defined in Ch07_08_.asm
struct ITD
{
    uint8_t* m_PbSrc;                 // Source image pixel buffer
    uint8_t* m_PbMask;                // Mask mask pixel buffer
    uint32_t m_NumPixels;             // Number of source image pixels
    uint32_t m_NumMaskedPixels;       // Number of masked pixels
    uint32_t m_SumMaskedPixels;       // Sum of masked pixels
    uint8_t m_Threshold;              // Image threshold value
    uint8_t m_Pad[3];                 // Available for future use
    double m_MeanMaskedPixels;        // Mean of masked pixels
};

// Functions defined in Ch07_08.cpp
extern bool AvxThresholdImageCpp(ITD* itd);
```

```
extern bool AvxCalcImageMeanCpp(ITD* itd);
extern "C" bool IsValid(uint32_t num_pixels, const uint8_t* pb_src, const uint8_t* pb_mask);

// Functions defined in Ch07_08_.asm
extern "C" bool AvxThresholdImage_(ITD* itd);
extern "C" bool AvxCalcImageMean_(ITD* itd);

// Functions defined in Ch07_08_BM.cpp
extern void AvxThreshold_BM(void);

// Miscellaneous constants
const uint8_t c_TestThreshold = 96;

//-------------------------------------------------
//               Ch07_08.cpp
//-------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include "Ch07_08.h"
#include "AlignedMem.h"
#include "ImageMatrix.h"

using namespace std;

extern "C" uint32_t c_NumPixelsMax = 16777216;

bool IsValid(uint32_t num_pixels, const uint8_t* pb_src, const uint8_t* pb_mask)
{
    const size_t alignment = 16;

    // Make sure num_pixels is valid
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if ((num_pixels % 64) != 0)
        return false;

    // Make sure image buffers are properly aligned
    if (!AlignedMem::IsAligned(pb_src, alignment))
        return false;
    if (!AlignedMem::IsAligned(pb_mask, alignment))
        return false;

    return true;
}

bool AvxThresholdImageCpp(ITD* itd)
{
    uint8_t* pb_src = itd->m_PbSrc;
```

```
    uint8_t* pb_mask = itd->m_PbMask;
    uint8_t threshold = itd->m_Threshold;
    uint32_t num_pixels = itd->m_NumPixels;

    // Verify pixel count and buffer alignment
    if (!IsValid(num_pixels, pb_src, pb_mask))
        return false;

    // Threshold the image
    for (uint32_t i = 0; i < num_pixels; i++)
        *pb_mask++ = (*pb_src++ > threshold) ? 0xff : 0x00;

    return true;
}

bool AvxCalcImageMeanCpp(ITD* itd)
{
    uint8_t* pb_src = itd->m_PbSrc;
    uint8_t* pb_mask = itd->m_PbMask;
    uint32_t num_pixels = itd->m_NumPixels;

    // Verify pixel count and buffer alignment
    if (!IsValid(num_pixels, pb_src, pb_mask))
        return false;

    // Calculate mean of masked pixels
    uint32_t sum_masked_pixels = 0;
    uint32_t num_masked_pixels = 0;

    for (uint32_t i = 0; i < num_pixels; i++)
    {
        uint8_t mask_val = *pb_mask++;
        num_masked_pixels += mask_val & 1;
        sum_masked_pixels += (*pb_src++ & mask_val);
    }

    itd->m_NumMaskedPixels = num_masked_pixels;
    itd->m_SumMaskedPixels = sum_masked_pixels;

    if (num_masked_pixels > 0)
        itd->m_MeanMaskedPixels = (double)sum_masked_pixels / num_masked_pixels;
    else
        itd->m_MeanMaskedPixels = -1.0;

    return true;
}

void AvxThreshold(void)
{
    const wchar_t* fn_src = L"..\\Ch07_Data\\TestImage2.bmp";
    const wchar_t* fn_mask1 = L"Ch07_08_AvxThreshold_TestImage2_Mask1.bmp";
    const wchar_t* fn_mask2 = L"Ch07_08_AvxThreshold_TestImage2_Mask2.bmp";
```

```
    ImageMatrix im_src(fn_src);
    int im_h = im_src.GetHeight();
    int im_w = im_src.GetWidth();
    ImageMatrix im_mask1(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_mask2(im_h, im_w, PixelType::Gray8);
    ITD itd1, itd2;

    itd1.m_PbSrc = im_src.GetPixelBuffer<uint8_t>();
    itd1.m_PbMask = im_mask1.GetPixelBuffer<uint8_t>();
    itd1.m_NumPixels = im_src.GetNumPixels();
    itd1.m_Threshold = c_TestThreshold;

    itd2.m_PbSrc = im_src.GetPixelBuffer<uint8_t>();
    itd2.m_PbMask = im_mask2.GetPixelBuffer<uint8_t>();
    itd2.m_NumPixels = im_src.GetNumPixels();
    itd2.m_Threshold = c_TestThreshold;

    // Threshold image
    bool rc1 = AvxThresholdImageCpp(&itd1);
    bool rc2 = AvxThresholdImage_(&itd2);

    if (!rc1 || !rc2)
    {
        cout << "\nInvalid return code: ";
        cout << "rc1 = " << rc1 << ", rc2 = " << rc2 << '\n';
        return;
    }

    im_mask1.SaveToBitmapFile(fn_mask1);
    im_mask2.SaveToBitmapFile(fn_mask2);

    // Calculate mean of masked pixels
    rc1 = AvxCalcImageMeanCpp(&itd1);
    rc2 = AvxCalcImageMean_(&itd2);

    if (!rc1 || !rc2)
    {
        cout << "\nInvalid return code: ";
        cout << "rc1 = " << rc1 << ", rc2 = " << rc2 << '\n';
        return;
    }

    // Print results
    const int w = 12;
    cout << fixed << setprecision(4);

    cout << "\nResults for AvxThreshold\n\n";
    cout << "                            C++       X86-AVX\n";
    cout << "---------------------------------------------\n";
```

```
    cout << "SumPixelsMasked:   ";
    cout << setw(w) << itd1.m_SumMaskedPixels << "  ";
    cout << setw(w) << itd2.m_SumMaskedPixels << '\n';
    cout << "NumPixelsMasked:   ";
    cout << setw(w) << itd1.m_NumMaskedPixels << "  ";
    cout << setw(w) << itd2.m_NumMaskedPixels << '\n';
    cout << "MeanMaskedPixels:  ";
    cout << setw(w) << itd1.m_MeanMaskedPixels << "  ";
    cout << setw(w) << itd2.m_MeanMaskedPixels << '\n';
}

int main()
{
    try
    {
        AvxThreshold();
        AvxThreshold_BM();
    }

    catch (...)
    {
        cout << "Unexpected exception has occurred\n";
    }

    return 0;
}

;-------------------------------------------------
;               Ch07_08.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; Image threshold data structure (see Ch07_08.h)
ITD                 struct
PbSrc               qword ?
PbMask              qword ?
NumPixels           dword ?
NumMaskedPixels     dword ?
SumMaskedPixels     dword ?
Threshold           byte ?
Pad                 byte 3 dup(?)
MeanMaskedPixels    real8 ?
ITD                 ends

                .const
                align 16
PixelScale      byte 16 dup(80h)            ;uint8 to int8 scale value
CountPixelsMask byte 16 dup(01h)            ;mask to count pixels
R8_MinusOne     real8 -1.0                  ;invalid mean value
```

```
                .code
                extern IsValid:proc

; extern "C" bool AvxThresholdImage_(ITD* itd);
;
; Returns:      0 = invalid size or unaligned image buffer, 1 = success

AvxThresholdImage_ proc frame
        _CreateFrame TI_,0,0,rbx
        _EndProlog

; Verify the arguments in the ITD structure
        mov rbx,rcx                         ;copy itd ptr to non-volatile register
        mov ecx,[rbx+ITD.NumPixels]         ;ecx = num_pixels
        mov rdx,[rbx+ITD.PbSrc]             ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]             ;r8 = pb_mask
        sub rsp,32                          ;allocate home area for IsValid
        call IsValid                        ;validate args
        or al,al
        jz Done                             ;jump if invalid

; Initialize registers for processing loop
        mov ecx,[rbx+ITD.NumPixels]             ;ecx = num_pixels
        shr ecx,6                               ;ecx = number of 64b pixel blocks
        mov rdx,[rbx+ITD.PbSrc]                 ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]                 ;r8 = pb_mask

        movzx r9d,byte ptr [rbx+ITD.Threshold]  ;r9d = threshold
        vmovd xmm1,r9d                          ;xmm1[7:0] = threshold
        vpxor xmm0,xmm0,xmm0                    ;mask for vpshufb
        vpshufb xmm1,xmm1,xmm0                  ;xmm1 = packed threshold

        vmovdqa xmm4,xmmword ptr [PixelScale]   ;packed pixel scale factor
        vpsubb xmm5,xmm1,xmm4                   ;scaled threshold

; Create the mask image
@@:     vmovdqa xmm0,xmmword ptr [rdx]      ;original image pixels
        vpsubb xmm1,xmm0,xmm4               ;scaled image pixels
        vpcmpgtb xmm2,xmm1,xmm5             ;mask pixels
        vmovdqa xmmword ptr [r8],xmm2       ;save mask result

        vmovdqa xmm0,xmmword ptr [rdx+16]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+16],xmm2

        vmovdqa xmm0,xmmword ptr [rdx+32]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+32],xmm2
```

```
        vmovdqa xmm0,xmmword ptr [rdx+48]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+48],xmm2

        add rdx,64
        add r8,64                           ;update pointers
        sub ecx,1                           ;update counter
        jnz @B                              ;repeat until done

        mov eax,1                           ;set success return code

Done:   _DeleteFrame rbx
        ret
AvxThresholdImage_ endp

;
; Macro _UpdateBlockSums
;

_UpdateBlockSums macro disp
        vmovdqa xmm0,xmmword ptr [rdx+disp] ;xmm0 = 16 image pixels
        vmovdqa xmm1,xmmword ptr [r8+disp]  ;xmm1 = 16 mask pixels
        vpand xmm2,xmm1,xmm8                ;xmm2 = 16 mask pixels (0x00 or 0x01)
        vpaddb xmm6,xmm6,xmm2               ;update block num_masked_pixels
        vpand xmm2,xmm0,xmm1                ;zero out unmasked image pixel
        vpunpcklbw xmm3,xmm2,xmm9           ;promote image pixels from byte to word
        vpunpckhbw xmm4,xmm2,xmm9
        vpaddw xmm4,xmm4,xmm3
        vpaddw xmm7,xmm7,xmm4               ;update block sum_mask_pixels
        endm

; extern "C" bool AvxCalcImageMean_(ITD* itd);
;
; Returns:  0 = invalid image size or unaligned image buffer, 1 = success

AvxCalcImageMean_ proc frame
        _CreateFrame CIM_,0,64,rbx
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Verify the arguments in the ITD structure
        mov rbx,rcx                         ;rbx = itd ptr
        mov ecx,[rbx+ITD.NumPixels]         ;ecx = num_pixels
        mov rdx,[rbx+ITD.PbSrc]             ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]             ;r8 = pb_mask
        sub rsp,32                          ;allocate home area for IsValid
        call IsValid                        ;validate args
        or al,al
        jz Done                             ;jump if invalid
```

```
; Initialize registers for processing loop
        mov ecx,[rbx+ITD.NumPixels]         ;ecx = num_pixels
        shr ecx,6                           ;ecx = number of 64b pixel blocks
        mov rdx,[rbx+ITD.PbSrc]             ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]             ;r8 = pb_mask

        vmovdqa xmm8,xmmword ptr [CountPixelsMask]  ;mask for counting pixels
        vpxor xmm9,xmm9,xmm9                        ;xmm9 = packed zero

        xor r10d,r10d                       ;r10d = num_masked_pixels (1 dword)
        vpxor xmm5,xmm5,xmm5                ;sum_masked_pixels (4 dwords)

;Calculate num_mask_pixels and sum_mask_pixels
LP1:    vpxor xmm6,xmm6,xmm6                ;num_masked_pixels_tmp (16 byte values)
        vpxor xmm7,xmm7,xmm7                ;sum_masked_pixels_tmp (8 word values)

        _UpdateBlockSums 0
        _UpdateBlockSums 16
        _UpdateBlockSums 32
        _UpdateBlockSums 48

; Update num_masked_pixels
        vpsrldq xmm0,xmm6,8
        vpaddb xmm6,xmm6,xmm0               ;num_mask_pixels_tmp (8 byte vals)
        vpsrldq xmm0,xmm6,4
        vpaddb xmm6,xmm6,xmm0               ;num_mask_pixels_tmp (4 byte vals)
        vpsrldq xmm0,xmm6,2
        vpaddb xmm6,xmm6,xmm0               ;num_mask_pixels_tmp (2 byte vals)
        vpsrldq xmm0,xmm6,1
        vpaddb xmm6,xmm6,xmm0               ;num_mask_pixels_tmp (1 byte val)
        vpextrb eax,xmm6,0
        add r10d,eax                        ;num_mask_pixels += num_mask_pixels_tmp

; Update sum_masked_pixels
        vpunpcklwd xmm0,xmm7,xmm9           ;promote sum_mask_pixels_tmp to dwords
        vpunpckhwd xmm1,xmm7,xmm9
        vpaddd xmm5,xmm5,xmm0
        vpaddd xmm5,xmm5,xmm1               ;sum_mask_pixels += sum_masked_pixels_tmp

        add rdx,64                          ;update pb_src pointer
        add r8,64                           ;update pb_mask pointer

        sub rcx,1                           ;update loop counter
        jnz LP1                             ;repeat if not done

; Compute mean of masked pixels
        vphaddd xmm0,xmm5,xmm5
        vphaddd xmm1,xmm0,xmm0
        vmovd eax,xmm1                      ;eax = final sum_mask_pixels

        test r10d,r10d                      ;is num_mask_pixels zero?
```

```
        jz NoMean                           ;if yes, skip calc of mean
        vcvtsi2sd xmm0,xmm0,eax             ;xmm0 = sum_masked_pixels
        vcvtsi2sd xmm1,xmm1,r10d            ;xmm1 = num_masked_pixels
        vdivsd xmm2,xmm0,xmm1               ;xmm2 = mean_masked_pixels
        jmp @F

NoMean: vmovsd xmm2,[R8_MinusOne]               ;use -1.0 for no mean

@@:     mov [rbx+ITD.SumMaskedPixels],eax       ;save sum_masked_pixels
        mov [rbx+ITD.NumMaskedPixels],r10d      ;save num_masked_pixels
        vmovsd [rbx+ITD.MeanMaskedPixels],xmm2  ;save mean
        mov eax,1                               ;set success return code

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame rbx
        ret
AvxCalcImageMean_   endp
        end
```

The algorithm that's used in example Ch07_08 consists of two phases. Phase 1 constructs the mask image that's shown in Figure 7-7. Phase 2 computes the mean intensity of all pixels in the grayscale image whose corresponding mask image pixel is white (i.e., above the specified threshold). The file Ch07_08.h that's shown in Listing 7-8 defines a structure named ITD that maintains data required by the algorithm. Note this structure contains two count values: m_NumPixels and m_NumMaskedPixels. The former value is the total number of image pixels, while the latter value represents the number of image pixels greater than m_Threshold.

The C++ code in Listing 7-8 contains separate thresholding and mean calculating functions. The function AvxThresholdImageCpp constructs the mask image by comparing each pixel in the grayscale image to the threshold value that's specified by itd->m_Threshold. If a grayscale image pixel is greater than this value, its corresponding pixel in the mask image is set to 0xff; otherwise, the mask image pixel is set to 0x00. The function AvxCalcImageMeanCpp uses this mask image to calculate the mean intensity value of all grayscale image pixels greater than the threshold value. Note that the for loop in this function computes num_mask_pixels and sum_mask_pixels using simple Boolean expressions instead of logical compare operations. The former technique is often faster and easier to implement using SIMD arithmetic.

Listing 7-8 also shows assembly language implementations of the thresholding and mean calculating functions. Following its prolog, the function AvxThresholdImage_ validates the arguments in the supplied ITD structure by calling the C++ function IsValid. Prior to the call instruction, AvxThresholdImage_ loads the required argument values for IsValid into the appropriate registers and allocates a home area using a sub rsp,32 instruction. After argument validation, the movzx r9d,byte ptr [rbx+ITD.Threshold] instruction loads the threshold value into register R9D. The ensuing vpshufb xmm1,xmm1,xmm0 instruction "broadcasts" the threshold value to all byte positions in register XMM1. The vpshufb instruction uses the low-order four bits of each byte in the second source operand as an index to permute the bytes in the destination operand (a zero is copied if the high-order bit is set in the seconds source operand byte). Figure 7-8 illustrates this process. The packed threshold value is then scaled using a vpsubb instruction. The reason for doing this is explained in the next paragraph.

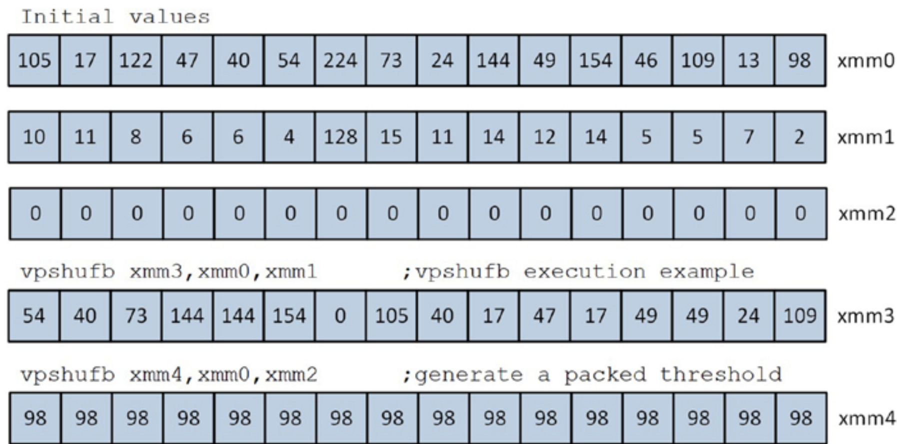



***Figure 7-8.*** *Execution examples of the instruction vpshufb*

The processing loop in function `AvxThreshholdImage_` uses the `vpcmpgtb` (Compare Packed Signed Integers for Greater Than) instruction to create the mask image. This instruction performs pairwise compares of the byte elements in the two source operands. If a byte in the first source operand is greater than the corresponding byte in the second operand, the destination operand byte is set to `0xff`; otherwise, the destination operand byte is set to `0x00`. Figure 7-9 illustrates execution of the `vpcmpgtb` instruction. It is important to note that `vpcmpgtb` executes its compares using *signed* integer arithmetic. This means that the pixels values in the grayscale image, which are unsigned byte values, must be re-scaled for compatibility with the `vpcmpgtb` instruction. The `vpsubb` instruction remaps the image's grayscale pixels values from [0, 255] to [-128, 127]. This is also the reason that a `vpsubb` instruction was used on the packed threshold value prior to the start of the loop. Following each compare operation, the `vmovdqa` instruction saves the mask pixels to the specified buffer. Similar to example `Ch07_04`, the function `AvxThresholdImage_` uses a partially unrolled processing loop to handle 64 pixels per iteration.

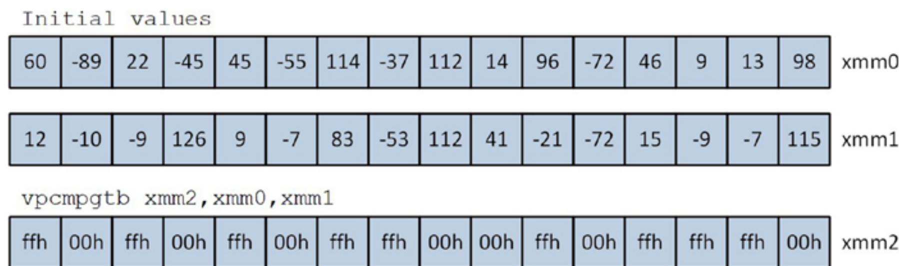



***Figure 7-9.*** *Execution of the instruction vpcmpgtb*

The assembly language function `AvxCalcImageMean_` also begins by validating its arguments using the C++ function `IsValid`. Following argument validation, the `xor r10d,r10d` and `vpxor xmm5,xmm5,xmm5` instructions initialize `num_masked_pixels` and `sum_masked_pixels` (four doublewords) to zero, respectively. The processing loop in function `AvxCalcImageMean_` uses a macro named `_UpdateBlockSums` to compute the intermediate values `num_masked_pixels_tmp` and `sum_masked_pixels_tmp` for a block of 64 pixels. This macro performs its calculations using packed byte and word arithmetic, which reduces the number of byte to doubleword size-promotions that must be carried out. Figure 7-10 illustrates the arithmetic and

Boolean operations that are performed by `_UpdateBlockSums`. The values `num_masked_pixels` (R10D) and `sum_masked_pixels` (XMM5) are then updated and the processing loop repeats until all pixels have been processed.

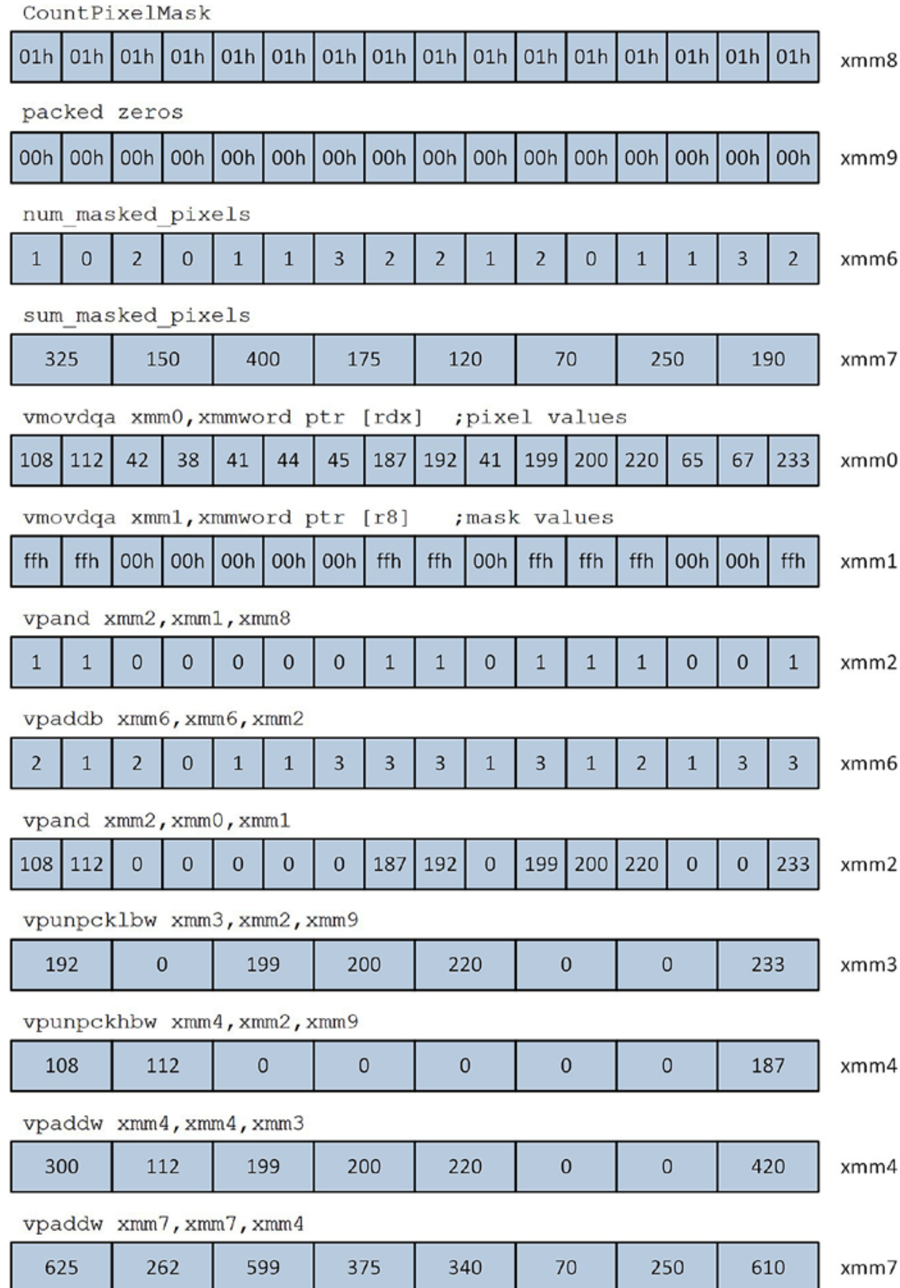



***Figure 7-10.*** *Masked pixel sum and pixel count calculations performed by macro `_UpdateBlockSums`*

Following completion of its processing loop, function AvxCalcImageMean_ calculates the final mean intensity value using scalar double-precision floating-point arithmetic. Note that num_mask_pixels is tested prior to calculating the mean in order to avoid a division-by-zero error. Here is the output for source code example Ch07_08:

```
Results for AvxThreshold

                          C++       X86-AVX
---------------------------------------------
SumPixelsMasked:       23813043      23813043
NumPixelsMasked:         138220        138220
MeanMaskedPixels:      172.2836      172.2836

Running benchmark function AvxThreshold_BM - please wait
Benchmark times save to file Ch07_08_AvxThreshold_BM_CHROMIUM.csv
```

Table 7-4 shows timing measurements for the source code example Ch07_08. Note that the measurements in this table are for an entire image thresholding and mean calculation sequence.

***Table 7-4.*** *Mean Execution Times (Microseconds) to Perform Image Thresholding and Mean Calculation Using TestImage2.bmp*

| CPU | C++ | Assembly Language |
|---|---|---|
| i7-4790S | 289 | 50 |
| i9-7900X | 250 | 40 |
| i7-8700K | 242 | 39 |

# Summary

Here are the key learning points for Chapter 7:

- The vpadd[b|w|d|q] instructions perform packed addition. The vpadds[b|w] and vpaddus[b|w] instructions perform packed signed and unsigned saturated addition.
- The vpsub[b|w|d|q] instructions perform packed subtraction. The vpsubs[b|w] and vpsubus[b|w] instructions perform packed signed and unsigned saturated subtraction.
- The vpmul[h|l]w instructions carry out multiplication using packed word operands. The vpmuldq and vpmulld instructions carry out multiplication using packed doubleword operands.
- The vpsll[w|d|q] and vpsrl[w|d|q] instructions execute logical left and right shifts using packed operands. The vpsra[w|d|q] instructions execute arithmetic right shifts using packed operands. The vps[l|r]dq instructions execute logical left and right shifts using 128-bit wide operands.
- Assembly language functions can use the vpand, vpor, and vpxor instructions to perform bitwise AND, inclusive OR, and exclusive OR operations using packed integer operands.

- The instructions `vpextr[b|w|d|q]` extract an element value from a packed operand. The `vpinsr[b|w|d|q]` instructions insert an element value into a packed operand.
- The `vpunpckl[bw|dw|dq]` and `vpunpckh[bw|dw|dq]` instructions unpack and interleave the contents of their two source operands. These instructions are frequently used to size-promote packed integer operands. The `vpackus[bw|dw]` instructions size-reduce packed integer operands using unsigned saturated arithmetic.
- The `vpminu[b|w|d]` and `vpmaxu[b|w|d]` instructions perform packed unsigned integer minimum-maximum compares.
- The `vpshufb` instruction rearranges the bytes of a packed operand according to a control mask.
- The `vpcmpgt[b|w|d|q]` instructions perform signed integer greater than compares using packed operands.
- Aligning the target of a jump instruction to a 16-byte boundary often results in faster executing `for` loops.

CHAPTER 8

# Advanced Vector Extensions 2

In the previous four chapters, you learned about the architecture and processing capabilities of AVX. These chapters explicated AVX's register sets, data types, and instructions. They also included numerous source code examples that illustrated how to perform scalar floating-point arithmetic, packed floating-point computations, and packed integer calculations. Many of the packed floating-point and packed integer source code examples exemplified important SIMD programming strategies and techniques whose exploitation often results in faster executing code.

This chapter explains the architecture and computational resources of Advanced Vector Extensions 2 (AVX2). You'll learn about AVX2's augmented capabilities for processing packed floating-point and packed integer operands. You'll also review important details regarding recent x86 platform instruction set extensions, including half-precision floating-point conversions, fused-multiply-add (FMA) operations, and new general-purpose register instructions.

The material presented in this chapter assumes that you have a solid understanding of AVX. If you feel that your understanding of AVX's register sets, data types, or SIMD processing capabilities is lacking in any way, you may want to review the relevant sections in the previous chapters before proceeding.

## AVX2 Execution Environment

AVX2 uses the same YMM and XMM register sets as AVX (see Figure 4-6). AVX2 also uses the MXCSR control-status register to signal floating-point arithmetic errors, configure rounding options, and control the generation of floating-point exceptions (see Figure 4-11). Like AVX, AVX2 supports floating-point SIMD operations using 128-bit or 256-bit wide operands containing either single-precision or double-precision values. AVX2 extends the packed integer processing capabilities of AVX to include both 128-bit and 256-bit wide operands (AVX only supports 128-bit wide integer operands). When used with a 256-bit wide packed integer operand, an AVX2 instruction can simultaneously process 32 byte, 16 word, 8 doubleword, or 4 quadword values. AVX2 also adds a number of useful instructions that administer packed floating-point and packed integer operands. You'll learn more about these instructions later in this chapter.

AVX2 instructions use the same instruction syntax as AVX. Most AVX2 instructions employ a three-operand format that consists of two source operands and one destination operand. Nearly all AVX2 instruction source operands are non-destructive. This means that source operands are not modified during instruction execution, except in cases where the destination operand register is the same as one of the source operand registers. A small set of AVX2 instructions employ a third immediate source operand that's typically used as a control mask.

The alignment requirements for AVX2 operands in memory are the same as AVX. Except for data transfer instructions that explicitly reference an aligned operand in memory (e.g., `vmovdqa`, `vmovap[d|s]`, etc.), proper alignment of an AVX2 operand in memory is not mandatory. However, 128-bit wide operands in memory should always be aligned to a 16-byte boundary whenever possible in order to maximize processing performance. Similarly, 256-bit wide operands should be aligned to a 32-byte boundary.

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_8

# AVX2 Packed Floating-Point

AVX2 expands the packed floating-point processing capabilities of AVX with the addition of data gather operations. The `vgatherdp[d|s]` and `vgatherqp[d|s]` instructions load multiple elements from non-contiguous memory locations (usually an array) into an XMM or YMM register. These instructions use a special memory addressing mode called vector scale-index-base (VSIB). VSIB memory addressing employs the following components to specify element locations in memory:

- *Scale:* The element size scale factor (1, 2, 4, or 8).
- *Index:* A vector index register (XMM or YMM) that contains signed doubleword or signed quadword indices.
- *Base:* A general-purpose register that points to the start of an array in memory.
- *Displacement:* An optional fixed offset from the start of the array.

Prior to the execution of a `vgatherdp[d|s]` or `vgatherqp[d|s]` instruction, the vector index register operand must be loaded with the correct indices. The processor uses these indices to select elements from the array. Figure 8-1 illustrates execution of the instruction `vgatherdps xmm0,[rax+xmm1*4],xmm2`. In this example, register RAX points to the start of an array containing single-precision floating-point values; register XMM1 holds four signed doubleword array indices; and register XMM2 contains a copy control mask. The copy control mask determines whether or not the `vgatherdps` instruction copies a particular array element to the destination operand. If the most significant bit of a control mask element is set, the corresponding array element that's specified in the vector index register is copied to the destination operand; otherwise, the destination operand element is not modified. Following successful execution of a `vgatherdp[d|s]` or `vgatherqp[d|s]` instruction, the copy control mask register (which is a source operand) contains all zeros.

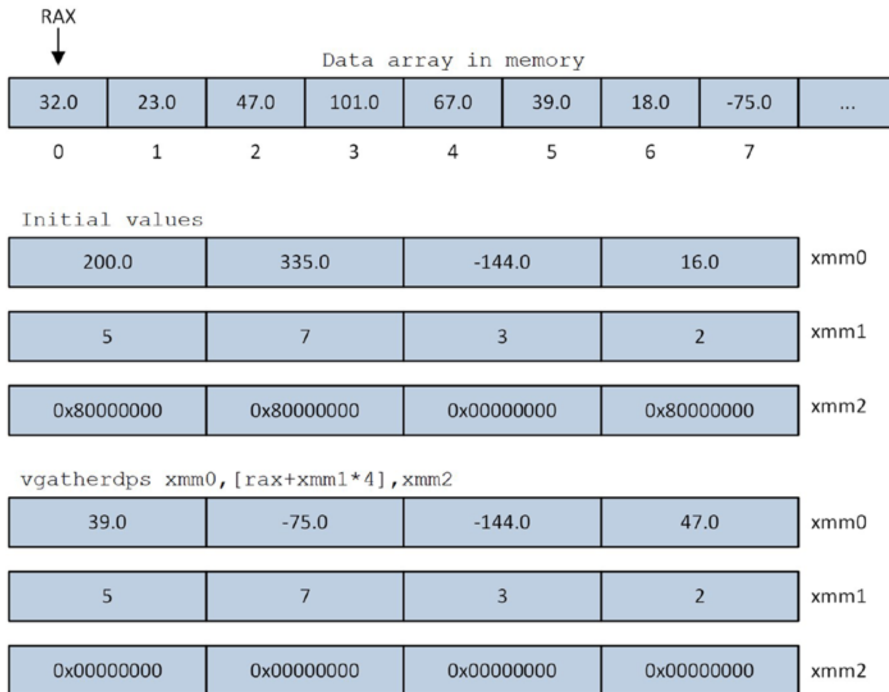



***Figure 8-1.*** *Illustration of the `vgatherdps` instruction execution*

The destination operand and second source operand (the copy control mask) of a `vgatherdp[d|s]` or `vgatherqp[d|s]` instruction must be an XMM or YMM register. The first source operand specifies the VSIB components (i.e., base register, vector index register, scale factor, and optional displacement). The `vgatherdp[d|s]` or `vgatherqp[d|s]` instructions do not perform any checks for invalid indices. An invalid index is any vector index register value that directs a gather instruction to load an element from a memory location that's outside the limits of the array. Using an invalid index will yield an incorrect result and possibly cause the processor to generate an exception.

The other notable AVX2 packed floating-point enhancement involves the `vbroadcasts[d|s]` instructions. On processors that support AVX2, the source operand for these instructions can be an XMM register (AVX only supports `vbroadcasts[d|s]` source operands in memory). When used in this manner, the `vbroadcasts[d|s]` instructions copy the low-order double-precision or single-precision floating-point element of an XMM register to each element position in the destination operand.

## AVX2 Packed Integer

As mentioned earlier in this chapter, AVX2 extends the packed integer capabilities of AVX to support both 128-bit and 256-bit wide operands. On systems that support AVX2, most packed integer instructions can use either the XMM or YMM registers as operands. The most notable exception to this rule are the `vpextr[b|w|d|q]` (Extract Integer Value) and `vpinsr[b|w|d|q]` (Insert Integer Value) instructions, which cannot be used with a YMM register operand. AVX2 also adds a number of new packed integer instructions that have no corresponding AVX (or x86-SSE) counterpart. Table 8-1 lists these instructions in alphabetical order.

***Table 8-1.*** *Summary of New AVX2 Packed Integer Instructions*

| Mnemonic | Description |
|---|---|
| `vbroadcasti128` | Broadcast 128 bits of integer data |
| `vextracti128` | Extract 128 bits of integer data |
| `vinserti128` | Insert 128 bits of integer data |
| `vpblendd` | Blend packed doublewords |
| `vpbroadcast[b|w|d|q]` | Broadcast integer value |
| `vperm2i128` | Permute 128-bit integer data |
| `vperm[d|q]` | Permute packed integers |
| `vpgatherd[d|q]` | Packed integer gather using signed doubleword indices |
| `vpgatherq[d|q]` | Packed integer gather using signed quadword indices |
| `vpmaskmov[d|q]` | Conditional packed integer data move |
| `vpsllv[d|q]` | Left logical shift using individual element bit counts |
| `vpsravd` | Right arithmetic shift using individual element bit counts |
| `vpsrlv[d|q]` | Right logical shift using individual element bit counts |

The `vpgatherd[d|q]` and `vpgatherq[d|q]` instructions that are shown in Table 8-1 use the same VSIB memory addressing scheme as their floating-point counterparts.

# X86 Instruction Set Extensions

In recent years, a number of instruction set extensions besides AVX and AVX2 have been added to the x86 platform. Many of these extensions include instructions that carry out specialized operations or accelerate the performance of specific algorithms. Table 8-2 lists the x86 instruction set extensions whose use is discussed and illustrated in subsequent chapters. It is important to keep in mind that all of the extensions shown in this table are distinct processor instruction sets. What this means from a programming perspective is that you should not assume that a particular instruction set or specific instruction is available based on whether or not the executing processor supports AVX or AVX2. The availability of a specific instruction set extension, including AVX and AVX2, should always be explicitly tested for using the `cupid` instruction. This is especially important for software compatibility with future processors from both AMD and Intel. You'll learn how to do this in Chapter 16.

***Table 8-2.*** *Recent x86 Instruction Set Extensions*

| Instruction Set Extension | CPUID Feature Flag |
|---|---|
| Enhanced unsigned integer addition | ADX |
| Advanced bit manipulation (group 1) | BMI1 |
| Advanced bit manipulation (group 2) | BMI2 |
| Half-precision floating-point conversions | F16C |
| Fused-multiply-add | FMA |
| Count leading zero bits | LZCNT |
| Count set bits | POPCNT |

The remainder of this section briefly describes the instruction set extensions that are shown in Table 8-2. Chapters 10 and 11 contain source code examples that illustrate how to use some of the instructions that are included in these extensions. Information regarding the instruction set extensions not shown in Table 8-2 can be found in the programming reference manuals published by AMD and Intel. Appendix A contains a list of these manuals.

## Half-Precision Floating-Point

Recent processors from both AMD and Intel incorporate instructions that carry out half-precision floating-point conversions. Compared to a standard single-precision floating-point value, a half-precision floating-point value is a reduced-precision floating-point number that contains three fields: an exponent (5 bits), a significand (11 bits), and a sign bit. Each half-precision floating-point value is 16 bits wide; the leading digit of the significand is implied. Compatible processors include instructions that can convert packed half-precision floating-point values to packed single-precision floating-point and vice versa. Table 8-3 shows these instructions. Half-precision floating-point values are primarily intended to reduce data storage space requirements, either in memory or on a physical device. The drawbacks of using half-precision floating-point values include reduced precision and limited range. Processors that support the conversion instructions shown in Table 8-3 *do not* include instructions for performing common arithmetic operations such as addition, subtraction, multiplication, and division using half-precision floating-point values.

***Table 8-3.*** *Half-Precision Floating-Point Conversion Instructions*

| Mnemonic | Description |
|---|---|
| `vcvtph2ps` | Convert half-precision floating-point to single-precision floating-point |
| `vcvtps2ph` | Convert single-precision floating-point to half-precision floating-point |

## Fused-Multiply-Add (FMA)

Modern processors from both AMD and Intel also include instructions that perform FMA operations. A FMA instruction combines multiplication and addition (or subtraction) into a single operation. More specifically, a fused-multiply-add (or fused-multiply-subtract) calculation performs a floating-point multiplication followed by a floating-point addition (or subtraction) using a single rounding operation. For example, consider the expression `d = b * c + a`. Using standard floating-point arithmetic, the processor initially calculates the product `b * c`, which includes a rounding operation. This is followed by a floating-point addition computation that also includes a rounding operation. If the expression is evaluated using FMA arithmetic, the processor does not round the intermediate product `b * c`. Rounding is carried out only once using the calculated product-sum `b * c + a`. FMA instructions are often used to improve the performance and accuracy of multiply-accumulate computations such as dot products and matrix-vector multiplications. Many signal-processing algorithms also make extensive use of FMA operations.

FMA instruction mnemonics employ a three-digit operand-ordering scheme that specifies which operands to use for multiplication and addition (or subtraction). In this scheme, all three instruction operands are used as source operands. The first mnemonic digit specifies the source operand to use as the multiplicand; the second digit specifies the source operand to use as the multiplier; and the third digit specifies the source operand that is added to (or subtracted from) the product. For example, consider the instruction `vfmadd132sd xmm4,xmm5,xmm6` (Fused Multiply-Add of Scalar Double-Precision Floating-Point Values). In this example, registers XMM4, XMM5, and XMM6 are source operands 1, 2, and 3, respectively. The `vfmadd132sd` instruction computes `xmm4[63:0] * xmm6[63:0] + xmm5[63:0]`, rounds the product-sum according to the rounding mode specified by MXCSR.RC, and saves the final result to `xmm4[63:0]`.

The x86 FMA instruction set extension supports operations using scalar or packed floating-point values, both single-precision and double-precision. Packed FMA operations can be performed using either the XMM or YMM registers. The XMM (YMM) registers support packed FMA calculations using two (four) double-precision or four (eight) single-precision floating-point values. Scalar FMA calculations are carried out using the XMM register set. For all FMA instructions, the first and second source operands must be a register. The third source operand can be a register or a memory location. If an FMA instruction uses an XMM register as a destination operand, the high-order 128 bits of the corresponding YMM register are set to zero. FMA instructions carry out their sole rounding operation using the mode that's specified by MXCSR.RC, as explained in the previous paragraph.

Table 8-4 shows the FMA instruction set. The instruction mnemonics in this table use the following two-letter suffixes: `pd` (packed double-precision floating-point), `ps` (packed single-precision floating-point), `sd` (scalar double-precision floating-point), and `ss` (scalar single-precision floating-point). The symbols `src1`, `src2`, and `src3` denote the three source operands; the destination operand `des` is always the same as `src1`.

***Table 8-4.*** *Overview of FMA Instructions*

| Subgroup | Mnemonic | Operation |
|---|---|---|
| VFMADD | vfmadd132[pd\|ps\|sd\|ss] | des = src1 * src3 + src2 |
| | vfmadd213[pd\|ps\|sd\|ss] | des = src2 * src1 + src3 |
| | vfmadd231[pd\|ps\|sd\|ss] | des = src2 * src3 + src1 |
| VFMSUB | vfmsub132[pd\|ps\|sd\|ss] | des = src1 * src3 - src2 |
| | vfmsub213[pd\|ps\|sd\|ss] | des = src2 * src1 - src3 |
| | vfmsub231[pd\|ps\|sd\|ss] | des = src2 * src3 - src1 |
| VFMADDSUB | vfmaddsub132[pd\|ps] | des = src1 * src3 + src2 (odd elements)<br>des = src1 * src3 - src2 (even elements) |
| | vfmaddsub213[pd\|ps] | des = src2 * src1 + src3 (odd elements)<br>des = src2 * src1 - src3 (even elements) |
| | vfmaddsub231[pd\|ps] | des = src2 * src3 + src1 (odd elements)<br>des = src2 * src3 - src1 (even elements) |
| VFMSUBADD | vfmsubadd132[pd\|ps] | des = src1 * src3 - src2 (odd elements)<br>des = src1 * src3 + src2 (even elements) |
| | vfmsubadd213[pd\|ps] | des = src2 * src1 - src3 (odd elements)<br>des = src2 * src1 + src3 (even elements) |
| | vfmsubadd231[pd\|ps] | des = src2 * src3 - src1 (odd elements)<br>des = src2 * src3 + src1 (even elements) |
| VFNMADD | vfnmadd132[pd\|ps\|sd\|ss] | des = -(src1 * src3) + src2 |
| | vfnmadd213[pd\|ps\|sd\|ss] | des = -(src2 * src1) + src3 |
| | vfnmadd231[pd\|ps\|sd\|ss] | des = -(src2 * src3) + src1 |
| VFNMSUB | vfnmsub132[pd\|ps\|sd\|ss] | des = -(src1 * src3) - src2 |
| | vfnmsub213[pd\|ps\|sd\|ss] | des = -(src2 * src1) - src3 |
| | vfnmsub231[pd\|ps\|sd\|ss] | des = -(src2 * src3) - src1 |

The FMA instructions that are shown in Table 8-4 are often identified as FMA3 instructions by many CPU feature detection utilities and online documentation sources. Some AMD processors also include supplemental FMA4 instructions, which carry out their FMA operations using three source operands and one destination operand (the three-digit operand ordering scheme is not used). These instructions are not shown in Table 8-4.

## General-Purpose Register Instruction Set Extensions

Recent enhancements to the x86 platform have also included a number of general-purpose register instruction set extensions. The ADX, BMI1, BMI2, LZCNT, and POPCNT instruction set extensions support enhanced unsigned integer arithmetic, advanced bit manipulations, and flagless register rotate and shift operations (a flagless operation does not update any of the status flags in RFLAGS). Many of these instructions are designed to accelerate the performance of specific algorithms such as large-integer

arithmetic and data encryption. Some of these general-purpose register instructions use a three-operand assembly-language syntax that's similar to AVX. Table 8-5 lists the instructions that comprise the ADX, BMI1, BMI2, LZCNT, and POPCNT extensions.

***Table 8-5.*** *Overview of ADX, BMI1, BMI2, LZCNT, and POPCNT Instructions*

| Mnemonic | CPUID Feature Flag | Description |
|---|---|---|
| `adcx` | ADX | Unsigned integer addition with carry flag |
| `adox` | ADX | Unsigned integer addition with overflow flag |
| `andn` | BMI1 | Bitwise AND of inverted operand1 with operand2 |
| `bextr` | BMI1 | Bitfield extract |
| `blsi` | BMI1 | Extract lowest set bit |
| `blsmsk` | BMI1 | Get mask up to lowest set bit |
| `blsr` | BMI1 | Reset lowest set bit |
| `bzhi` | BMI2 | Zero high bits |
| `lzcnt` | LZCNT | Count number of leading zero bits |
| `mulx` | BMI2 | Flagless unsigned integer multiplication |
| `pdep` | BMI2 | Parallel bits deposit |
| `pext` | BMI2 | Parallel bits extract |
| `popcnt` | POPCNT | Count number of set bits |
| `rorx` | BMI2 | Flagless rotate right |
| `sarx` | BMI2 | Flagless arithmetic shift right |
| `shlx` | BMI2 | Flagless logical shift left |
| `shrx` | BMI2 | Flagless logical shift right |
| `tzcnt` | BMI1 | Count number of trailing zero bits |

# Summary

Here are the key learning points of Chapter 8:

- AVX2 uses the same register sets, data types, and instruction syntax as AVX.
- AVX2 extends the packed integer processing capabilities of AVX to support operations using 256-bit wide operands.
- AVX2 includes new packed integer processing instructions that perform broadcast, permute, and variable bit-shift operations.
- The `vgather[d|q]p[d|s]` and `vpgather[d|q][d|q]` instructions load floating-point or integer values into an XMM or YMM register from non-contiguous locations in memory. These instructions use the VSIB addressing mode to carry out their operations.

- The `vcvtph2ps` and `vcvtps2ph` instructions perform conversions between packed half-precision to single-precision floating-point values.
- All FMA instructions execute a floating-point multiplication followed by a floating-point addition (or subtraction) using a single rounding operation. The x86 FMA instruction set extension supports a variety of FMA operations using both scalar and packed single-precision or double-precision floating-point values.
- The ADX, BMI1, BMI2, LZCNT, and POPCNT instruction set extensions include instructions that support enhanced unsigned integer addition, advanced bit manipulation, and flagless shift and rotate operations.

CHAPTER 9

# AVX2 Programming – Packed Floating-Point

In Chapter 6, you learned how to use the AVX instruction set to perform packed floating-point operations using the XMM register set and 128-bit wide operands. In this chapter, you learn how carry out packed floating-point operations using the YMM register set and 256-bit wide operands. The chapter begins with a simple example that demonstrates the basics of packed floating-point arithmetic and YMM register use. This is followed by three source code examples that illustrate how to perform packed calculations with floating-point arrays.

Chapter 6 also presented source code examples that exploited the AVX instruction set to accelerate matrix transposition and multiplication using single-precision floating-point values. In this chapter, you learn how to perform these same calculations using double-precision floating-point values. You also study a source code example that computes the inverse of a matrix. The final two source code examples in this chapter explain how to perform data blends, permutes, and gathers using packed floating-point operands.

You may recall that the source code examples in Chapter 6 used only XMM register operands with AVX instructions. This was done to avoid information overload and maintain a reasonable chapter length. Nearly all AVX floating-point instructions can use either the XMM or YMM registers as operands. Many of the source code examples in this chapter will run on a processor that supports AVX. The function names in these examples use the prefix `Avx`. Similarly, source code examples that required an AVX2-compatible processor use the function name prefix `Avx2`. You can use one of the freely-available tools listed in Appendix A to determine whether your computer supports only AVX or both AVX and AVX2.

## Packed Floating-Point Arithmetic

Listing 9-1 shows the source code for example `Ch09_01`. This example illustrates how to perform common arithmetic operations using 256-bit wide single-precision and double-precision floating-point operands. It also illustrates how to use the `vzeroupper` instruction and several MASM directives for 256-bit wide operands.

***Listing 9-1.*** Example Ch09_01

```
//------------------------------------------------
//               YmmVal.h
//------------------------------------------------

#pragma once
#include <string>
```

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_9

```
#include <cstdint>
#include <sstream>
#include <iomanip>

struct YmmVal
{
public:
    union
    {
        int8_t m_I8[32];
        int16_t m_I16[16];
        int32_t m_I32[8];
        int64_t m_I64[4];
        uint8_t m_U8[32];
        uint16_t m_U16[16];
        uint32_t m_U32[8];
        uint64_t m_U64[4];
        float m_F32[8];
        double m_F64[4];
    };

//------------------------------------------------
//               Ch09_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>
#include "YmmVal.h"

using namespace std;

extern "C" void AvxPackedMathF32_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);
extern "C" void AvxPackedMathF64_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);

void AvxPackedMathF32(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c[8];

    a.m_F32[0] = 36.0f;                 b.m_F32[0] = -0.1111111f;
    a.m_F32[1] = 0.03125f;              b.m_F32[1] = 64.0f;
    a.m_F32[2] = 2.0f;                  b.m_F32[2] = -0.0625f;
    a.m_F32[3] = 42.0f;                 b.m_F32[3] = 8.666667f;
    a.m_F32[4] = 7.0f;                  b.m_F32[4] = -18.125f;
    a.m_F32[5] = 20.5f;                 b.m_F32[5] = 56.0f;
    a.m_F32[6] = 36.125f;               b.m_F32[6] = 24.0f;
    a.m_F32[7] = 0.5f;                  b.m_F32[7] = -98.6f;
```

```
    AvxPackedMathF32_(a, b, c);

    cout << ("\nResults for AvxPackedMathF32\n");

    cout << "a[0]:       " << a.ToStringF32(0) << '\n';
    cout << "b[0]:       " << b.ToStringF32(0) << '\n';
    cout << "addps[0]:   " << c[0].ToStringF32(0) << '\n';
    cout << "subps[0]:   " << c[1].ToStringF32(0) << '\n';
    cout << "mulps[0]:   " << c[2].ToStringF32(0) << '\n';
    cout << "divps[0]:   " << c[3].ToStringF32(0) << '\n';
    cout << "absps b[0]: " << c[4].ToStringF32(0) << '\n';
    cout << "sqrtps a[0]:" << c[5].ToStringF32(0) << '\n';
    cout << "minps[0]:   " << c[6].ToStringF32(0) << '\n';
    cout << "maxps[0]:   " << c[7].ToStringF32(0) << '\n';

    cout << '\n';

    cout << "a[1]:       " << a.ToStringF32(1) << '\n';
    cout << "b[1]:       " << b.ToStringF32(1) << '\n';
    cout << "addps[1]:   " << c[0].ToStringF32(1) << '\n';
    cout << "subps[1]:   " << c[1].ToStringF32(1) << '\n';
    cout << "mulps[1]:   " << c[2].ToStringF32(1) << '\n';
    cout << "divps[1]:   " << c[3].ToStringF32(1) << '\n';
    cout << "absps b[1]: " << c[4].ToStringF32(1) << '\n';
    cout << "sqrtps a[1]:" << c[5].ToStringF32(1) << '\n';
    cout << "minps[1]:   " << c[6].ToStringF32(1) << '\n';
    cout << "maxps[1]:   " << c[7].ToStringF32(1) << '\n';
}

void AvxPackedMathF64(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c[8];

    a.m_F64[0] = 2.0;           b.m_F64[0] = M_PI;
    a.m_F64[1] = 4.0 ;          b.m_F64[1] = M_E;
    a.m_F64[2] = 7.5;           b.m_F64[2] = -9.125;
    a.m_F64[3] = 3.0;           b.m_F64[3] = -M_PI;

    AvxPackedMathF64_(a, b, c);
    cout << ("\nResults for AvxPackedMathF64\n");

    cout << "a[0]:       " << a.ToStringF64(0) << '\n';
    cout << "b[0]:       " << b.ToStringF64(0) << '\n';
    cout << "addpd[0]:   " << c[0].ToStringF64(0) << '\n';
    cout << "subpd[0]:   " << c[1].ToStringF64(0) << '\n';
    cout << "mulpd[0]:   " << c[2].ToStringF64(0) << '\n';
    cout << "divpd[0]:   " << c[3].ToStringF64(0) << '\n';
    cout << "abspd b[0]: " << c[4].ToStringF64(0) << '\n';
    cout << "sqrtpd a[0]:" << c[5].ToStringF64(0) << '\n';
```

```
    cout << "minpd[0]:   " << c[6].ToStringF64(0) << '\n';
    cout << "maxpd[0]:   " << c[7].ToStringF64(0) << '\n';

    cout << '\n';

    cout << "a[1]:       " << a.ToStringF64(1) << '\n';
    cout << "b[1]:       " << b.ToStringF64(1) << '\n';
    cout << "addpd[1]:   " << c[0].ToStringF64(1) << '\n';
    cout << "subpd[1]:   " << c[1].ToStringF64(1) << '\n';
    cout << "mulpd[1]:   " << c[2].ToStringF64(1) << '\n';
    cout << "divpd[1]:   " << c[3].ToStringF64(1) << '\n';
    cout << "abspd b[1]: " << c[4].ToStringF64(1) << '\n';
    cout << "sqrtpd a[1]:" << c[5].ToStringF64(1) << '\n';
    cout << "minpd[1]:   " << c[6].ToStringF64(1) << '\n';
    cout << "maxpd[1]:   " << c[7].ToStringF64(1) << '\n';
}

int main()
{
    AvxPackedMathF32();
    AvxPackedMathF64();
    return 0;
}
```

```
;-------------------------------------------------
;               Ch09_01.asm
;-------------------------------------------------

; Mask values used to calculate floating-point absolute values
            .const
AbsMaskF32  dword 8 dup(7fffffffh)
AbsMaskF64  qword 4 dup(7fffffffffffffffh)

; extern "C" void AvxPackedMathF32_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);

            .code
AvxPackedMathF32_ proc

; Load packed SP floating-point values
        vmovaps ymm0,ymmword ptr [rcx]      ;ymm0 = *a
        vmovaps ymm1,ymmword ptr [rdx]      ;ymm1 = *b

; Packed SP floating-point addition
        vaddps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8],ymm2

; Packed SP floating-point subtraction
        vsubps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+32],ymm2
```

```
; Packed SP floating-point multiplication
        vmulps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+64],ymm2

; Packed SP floating-point division
        vdivps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+96],ymm2

; Packed SP floating-point absolute value (b)
        vandps ymm2,ymm1,ymmword ptr [AbsMaskF32]
        vmovaps ymmword ptr [r8+128],ymm2

; Packed SP floating-point square root (a)
        vsqrtps ymm2,ymm0
        vmovaps ymmword ptr [r8+160],ymm2

; Packed SP floating-point minimum
        vminps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+192],ymm2

; Packed SP floating-point maximum
        vmaxps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+224],ymm2

        vzeroupper
        ret
AvxPackedMathF32_ endp

; extern "C" void AvxPackedMathF64_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);

AvxPackedMathF64_ proc

; Load packed DP floating-point values
        vmovapd ymm0,ymmword ptr [rcx]       ;ymm0 = *a
        vmovapd ymm1,ymmword ptr [rdx]       ;ymm1 = *b

; Packed DP floating-point addition
        vaddpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8],ymm2

; Packed DP floating-point subtraction
        vsubpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+32],ymm2

; Packed DP floating-point multiplication
        vmulpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+64],ymm2

; Packed DP floating-point division
        vdivpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+96],ymm2
```

```
; Packed DP floating-point absolute value (b)
        vandpd ymm2,ymm1,ymmword ptr [AbsMaskF64]
        vmovapd ymmword ptr [r8+128],ymm2

; Packed DP floating-point square root (a)
        vsqrtpd ymm2,ymm0
        vmovapd ymmword ptr [r8+160],ymm2

; Packed DP floating-point minimum
        vminpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+192],ymm2

; Packed DP floating-point maximum
        vmaxpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+224],ymm2

        vzeroupper
        ret
AvxPackedMathF64_ endp
        end
```

Listing 9-1 begins with the declaration of a C++ structure named YmmVal that's declared in the header file YmmVal.h. This structure is similar to the XmmVal structure that you saw in Chapter 6. YmmVal contains a publicly-accessible anonymous union that facilitates packed operand data exchange between functions written in C++ and x86 assembly language. The members of this union correspond to the packed data types that can be used with a YMM register. The structure YmmVal also includes several formatting and display functions (the source code for these member functions is not shown).

The C++ code for example Ch09_01 starts with declarations for the assembly language functions AvxPackedMathF32_ and AvxPackedMathF64_. These functions carry out various packed single-precision and double-precision floating-point arithmetic operations using the supplied YmmVal arguments. Following the assembly language function declarations is the function AvxPackedMathF32. This function starts by initializing YmmVal variables a and b. Note that the C++ specifier alignas(32) is used with each YmmVal declaration. This specifier instructs the C++ compiler to align each YmmVal variable on a 32-byte boundary. Following YmmVal variable initialization, AvxPackedMathF32 calls the assembly language function AvxPackedMathF32_ to perform the required arithmetic. The results are then streamed to cout. The function AvxPackedMathF64 is the double-precision floating-point counterpart of AvxPackedMathF32.

Near the top of the assembly language code in Listing 9-1 is a .const section that defines packed constant values for calculating floating-point absolute values. The text dup is a MASM operator that allocates and optionally initializes multiple data values. In the current example, the statement AbsMaskF32 dword 8 dup(7fffffffh) allocates storage space for eight doubleword values and each value is initialized to 0x7fffffff. The following statement, AbsMaskF64 qword 4 dup(7fffffffffffffffh), allocates four quadwords of 0x7fffffffffffffff. Note that neither of these 256-bit wide operands is preceded by an align statement, which means that they may not be properly aligned in memory. The reason for this is that the MASM align directive does not support 32-byte alignment within a .const, .data, or .code section. Later in this chapter, you learn how to define a custom segment of constant values that supports 32-byte alignment.

Following the .const section, the first instruction of AvxPackedMathF32_, vmovaps ymm0,ymmword ptr [rcx], loads argument a (i.e., the eight floating-point values of YmmVal a) into register YMM0. The vmovaps can be used here since YmmVal a was defined using the alignas(32) specifier in the C++ code. The operator ymmword ptr directs the assembler to treat the memory location pointed to by RCX as a 256-bit wide operand. Use of the ymmword ptr operator is optional in this instance and employed to improve code

readability. The ensuing `vmovaps ymm1,ymmword ptr [rdx]` instruction loads b into register YMM1. The `vaddps ymm2,ymm0,ymm1` instruction that follows sums the packed single-precision floating-point values in YMM0 and YMM1; it then saves the result to YMM2. The `vmovaps ymmword ptr [r8],ymm2` instruction saves the packed sums to `c[0]`.

The ensuing `vsubps`, `vmulps`, and `vdivps` instructions carry out packed single-precision floating-point subtraction, multiplication, and division. This is followed by a `vandps ymm2,ymm1,ymmword ptr [AbsMaskF32]` instruction that calculates packed absolute values using argument `b`. The remaining instructions in `AvxPackedMathF32_` calculate packed single-precision floating-point square roots, minimums, and maximums.

Prior to its `ret` instruction, the function `AvxPackedMath32_` uses a `vzeroupper` instruction, which zeros the high-order 128 bits of each YMM register. As explained in Chapter 4, the `vzeroupper` instruction is needed here to avoid potential performance delays that can occur whenever the processor transitions from executing x86-AVX instructions that use 256-bit wide operands to executing x86-SSE instructions. Any assembly language function that uses one or more YMM registers and is callable from code that potentially uses x86-SSE instructions should always ensure that a `vzeroupper` instruction is executed before program control is transferred back to the calling function. You'll see additional examples of `vzeroupper` instruction use in this and subsequent chapters.

The organization of function `AvxPackedMathF64_` is similar to `AvxPackedMathF32_`. `AvxPackedMathF64_` carries out its calculations using the double-precision versions of the same instructions that are used in `AvxPackedMathF32_`. Here is the output for source code example `Ch09_01`:

```
Results for AvxPackedMathF32
a[0]:            36.000000      0.031250    |       2.000000     42.000000
b[0]:            -0.111111     64.000000    |      -0.062500      8.666667
addps[0]:        35.888889     64.031250    |       1.937500     50.666668
subps[0]:        36.111111    -63.968750    |       2.062500     33.333332
mulps[0]:        -4.000000      2.000000    |      -0.125000    364.000000
divps[0]:      -324.000031      0.000488    |     -32.000000      4.846154
absps b[0]:       0.111111     64.000000    |       0.062500      8.666667
sqrtps a[0]:      6.000000      0.176777    |       1.414214      6.480741
minps[0]:        -0.111111      0.031250    |      -0.062500      8.666667
maxps[0]:        36.000000     64.000000    |       2.000000     42.000000

a[1]:             7.000000     20.500000    |      36.125000      0.500000
b[1]:           -18.125000     56.000000    |      24.000000    -98.599998
addps[1]:       -11.125000     76.500000    |      60.125000    -98.099998
subps[1]:        25.125000    -35.500000    |      12.125000     99.099998
mulps[1]:      -126.875000   1148.000000    |     867.000000    -49.299999
divps[1]:        -0.386207      0.366071    |       1.505208     -0.005071
absps b[1]:      18.125000     56.000000    |      24.000000     98.599998
sqrtps a[1]:      2.645751      4.527693    |       6.010407      0.707107
minps[1]:       -18.125000     20.500000    |      24.000000    -98.599998
maxps[1]:         7.000000     56.000000    |      36.125000      0.500000

Results for AvxPackedMathF64
a[0]:                       2.000000000000  |                 4.000000000000
b[0]:                       3.141592653590  |                 2.718281828459
addpd[0]:                   5.141592653590  |                 6.718281828459
subpd[0]:                  -1.141592653590  |                 1.281718171541
mulpd[0]:                   6.283185307180  |                10.873127313836
divpd[0]:                   0.636619772368  |                 1.471517764686
```

```
abspd b[0]:            3.141592653590   |          2.718281828459
sqrtpd a[0]:           1.414213562373   |          2.000000000000
minpd[0]:              2.000000000000   |          2.718281828459
maxpd[0]:              3.141592653590   |          4.000000000000

a[1]:                  7.500000000000   |          3.000000000000
b[1]:                 -9.125000000000   |         -3.141592653590
addpd[1]:             -1.625000000000   |         -0.141592653590
subpd[1]:             16.625000000000   |          6.141592653590
mulpd[1]:            -68.437500000000   |         -9.424777960769
divpd[1]:             -0.821917808219   |         -0.954929658551
abspd b[1]:            9.125000000000   |          3.141592653590
sqrtpd a[1]:           2.738612787526   |          1.732050807569
minpd[1]:             -9.125000000000   |         -3.141592653590
maxpd[1]:              7.500000000000   |          3.000000000000
```

# Packed Floating-Point Arrays

In previous chapters, you learned how to carry out integer and floating-point array calculations using the general-purpose and XMM register sets. In this section, you learn how to perform floating-point array operations using the YMM register set.

## Simple Calculations

Listing 9-2 shows the source code for example Ch09_02. This example illustrates how to perform simple array calculations using 256-bit wide packed floating-point operands. It also demonstrates how to detect and exclude invalid array elements from packed calculations. Source code example Ch09_02 is an array implementation of example Ch05_02 from Chapter 5, which calculated sphere surface areas and volumes. In that example, the assembly language function CalcSphereAreaVolume_ computed the surface area and volume of a single sphere. In this example, the sphere radii are passed via an array to calculating functions coded using C++ and assembly language. To make the example a little more interesting, both the C++ and assembly language calculating functions test for radii less than zero. If an invalid radius is detected, the calculating functions set the corresponding elements in the surface area and volume arrays to QNaN.

***Listing 9-2.*** Example Ch09_02

```
//------------------------------------------------
//               Ch09_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <limits>
#define _USE_MATH_DEFINES
#include <math.h>
```

```
using namespace std;

extern "C" void AvxCalcSphereAreaVolume_(float* sa, float* vol, const float* r, size_t n);

extern "C" float c_PI_F32 = (float)M_PI;
extern "C" float c_QNaN_F32 = numeric_limits<float>::quiet_NaN();

void Init(float* r, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 100};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        r[i] = (float)ui_dist(rng) / 10.0f;

    // Set invalid radii for test purposes
    if (n > 2)
    {
        r[2] = -r[2];
        r[n / 4] = -r[n / 4];
        r[n / 2] = -r[n / 2];
        r[n / 4 * 3] = -r[n / 4 * 3];
        r[n - 2] = -r[n - 2];
    }
}

void AvxCalcSphereAreaVolumeCpp(float* sa, float* vol, const float* r, size_t n)
{
    for (size_t i = 0; i < n; i++)
    {
        if (r[i] < 0.0f)
            sa[i] = vol[i] = c_QNaN_F32;
        else
        {
            sa[i] = r[i] * r[i] * 4.0f * c_PI_F32;
            vol[i] = sa[i] * r[i] / 3.0f;
        }
    }
}

void AvxCalcSphereAreaVolume(void)
{
    const size_t n = 21;
    alignas(32) float r[n];
    alignas(32) float sa1[n];
    alignas(32) float vol1[n];
    alignas(32) float sa2[n];
    alignas(32) float vol2[n];
```

```
    Init(r, n, 93);

    AvxCalcSphereAreaVolumeCpp(sa1, vol1, r, n);
    AvxCalcSphereAreaVolume_(sa2, vol2, r, n);

    cout << "\nResults for AvxCalcSphereAreaVolume\n";
    cout << fixed;

    const float eps = 1.0e-6f;

    for (size_t i = 0; i < n; i++)
    {
        cout << setw(2) << i << ":  ";
        cout << setprecision(2);
        cout << setw(5) << r[i] << " | ";
        cout << setprecision(6);
        cout << setw(12) << sa1[i] << "   ";
        cout << setw(12) << sa2[i] << " | ";
        cout << setw(12) << vol1[i] << "   ";
        cout << setw(12) << vol2[i];

        bool b0 = (fabs(sa1[i] - sa2[i]) > eps);
        bool b1 = (fabs(vol1[i] - vol2[i]) > eps);

        if (b0 || b1)
            cout << " Compare discrepancy";
        cout << '\n';
    }
}

int main()
{
    AvxCalcSphereAreaVolume();
    return 0;
}

;-------------------------------------------------
;               Ch09_02.asm
;-------------------------------------------------

        include <cmpequ.asmh>
        include <MacrosX86-64-AVX.asmh>

        .const
r4_3p0  real4 3.0
r4_4p0  real4 4.0

        extern c_PI_F32:real4
        extern c_QNaN_F32:real4
```

```
; extern "C" void AvxCalcSphereAreaVolume_(float* sa, float* vol, const float* r, size_t n);

        .code
AvxCalcSphereAreaVolume_ proc frame
        _CreateFrame CC_,0,64
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Initialize
        vbroadcastss ymm0,real4 ptr [r4_4p0]        ;packed 4.0
        vbroadcastss ymm1,real4 ptr [c_PI_F32]      ;packed PI
        vmulps ymm6,ymm0,ymm1                       ;packed 4.0 * PI
        vbroadcastss ymm7,real4 ptr [r4_3p0]        ;packed 3.0
        vbroadcastss ymm8,real4 ptr [c_QNaN_F32]    ;packed QNaN
        vxorps ymm9,ymm9,ymm9                       ;packed 0.0

        xor eax,eax                             ;common offset for arrays

        cmp r9,8
        jb FinalR                               ;skip main loop if n < 8

; Calculate surface area and volume values using packed arithmetic
@@:     vmovdqa ymm0,ymmword ptr [r8+rax]       ;load next 8 radii
        vmulps ymm2,ymm6,ymm0                   ;4.0 * PI * r
        vmulps ymm3,ymm2,ymm0                   ;4.0 * PI * r * r

        vcmpps ymm1,ymm0,ymm9,CMP_LT            ;ymm1 = mask of radii < 0.0

        vandps ymm4,ymm1,ymm8                   ;set surface area to QNaN for radii < 0.0
        vandnps ymm5,ymm1,ymm3                  ;keep surface area for radii >= 0.0
        vorps ymm5,ymm4,ymm5                    ;final packed surface area
        vmovaps ymmword ptr[rcx+rax],ymm5       ;save packed surface area

        vmulps ymm2,ymm3,ymm0                   ;4.0 * PI * r * r * r
        vdivps ymm3,ymm2,ymm7                   ;4.0 * PI * r * r * r / 3.0
        vandps ymm4,ymm1,ymm8                   ;set volume to QNaN for radii < 0.0
        vandnps ymm5,ymm1,ymm3                  ;keep volume for radii >= 0.0
        vorps ymm5,ymm4,ymm5                    ;final packed volume
        vmovaps ymmword ptr[rdx+rax],ymm5       ;save packed volume

        add rax,32                              ;rax = offset to next set of radii
        sub r9,8
        cmp r9,8
        jae @B                                  ;repeat until n < 8

; Perform final calculations using scalar arithmetic
FinalR: test r9,r9
        jz Done                                 ;skip loop of no more elements
```

```
@@:     vmovss xmm0,real4 ptr [r8+rax]
        vmulss xmm2,xmm6,xmm0                   ;4.0 * PI * r
        vmulss xmm3,xmm2,xmm0                   ;4.0 * PI * r * r

        vcmpss xmm1,xmm0,xmm9,CMP_LT

        vandps xmm4,xmm1,xmm8
        vandnps xmm5,xmm1,xmm3
        vorps xmm5,xmm4,xmm5
        vmovss real4 ptr[rcx+rax],xmm5          ;save surface area

        vmulss xmm2,xmm3,xmm0                   ;4.0 * PI * r * r * r
        vdivss xmm3,xmm2,xmm7                   ;4.0 * PI * r * r * r / 3.0
        vandps xmm4,xmm1,xmm8
        vandnps xmm5,xmm1,xmm3
        vorps xmm5,xmm4,xmm5
        vmovss real4 ptr[rdx+rax],xmm5          ;save volume

        add rax,4
        dec r9
        jnz @B                                  ;repeat until done

Done:   vzeroupper

        _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame
        ret
AvxCalcSphereAreaVolume_ endp
        end
```

The C++ code in Listing 9-2 includes a function named `AvxCalcSphereAreaVolumeCpp`. This function calculates sphere surface areas and volumes. The sphere radii are passed to `AvxCalcSphereAreaVolumeCpp` via an array. Prior to calculating a surface area or volume, the sphere's radius (`r[i]`) is tested to verify that it's not negative. If the radius is negative, the corresponding elements in the surface area and volume arrays (`sa[i]` and `vol[i]`) are set to `c_QNaN_F32`. The remaining C++ code performs the necessary initializations, exercises the C++ and assembly language calculating functions, and displays the results. Note that the function `AvxCalcSphereAreaVolume` employs the `alignas(32)` specifier with each array declaration.

The assembly language function `AvxCalcSphereAreaVolume_` performs the same calculations as its C++ counterpart. Following its prolog, `AvxCalcSphereAreaVolume_` uses a series of `vbroadcastss` instructions to initialize packed versions of the required constants. Prior to the start of the processing loop, a `cmp r9,8` instruction checks the value of `n`. The reason for this check is that the processing loop carries out eight surface area and volume calculations simultaneously using 256-bit wide operands. The `jb FinalR` conditional jump instruction skips the processing loop if there are fewer than eight radii to process.

Each processing loop iteration begins with a `vmovdqa ymm0,ymmword ptr [r8+rax]` instruction that loads eight single-precision floating-point radii into register YMM0. The ensuing `vmulps` instructions calculate the sphere surface areas. The next instruction, `vcmpps ymm1,ymm0,ymm9,CMP_LT`, tests each sphere radii for a value less than 0.0 (register YMM9 contains packed 0.0). Recall that the `vcmpps` instruction signifies its results by setting elements in the destination operand to either `0x00000000` (false compare predicate) or `0xffffffff` (true compare predicate). The `vandps`, `vandnps`, and `vorps` instructions that follow set the surface area of each sphere whose radius is less than 0.0 to `c_QNaN_F32`. Figure 9-1 illustrates this operation in greater detail. A `vmovaps ymmword ptr[rcx+rax],ymm5` instruction saves the eight sphere surface area values to the array `sa`.

Packed constants

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| QNaN | QNaN | QNaN | QNaN | QNaN | QNaN | QNaN | QNaN | ymm8 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ymm9 |

Radii

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 9.3 | 2.6 | -6.6 | 9.6 | 3.7 | -6.1 | 10.0 | 3.8 | ymm0 |

Calculated surface areas

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1086.86 | 84.94 | 547.39 | 1158.11 | 172.03 | 467.59 | 1256.63 | 181.45 | ymm3 |

vcmpps ymm1,ymm0,ymm9,CMP_LT

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00000000h | 00000000h | FFFFFFFFh | 00000000h | 00000000h | FFFFFFFFh | 00000000h | 00000000h | ymm1 |

vandps ymm4,ymm1,ymm8

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0.0 | 0.0 | QNaN | 0.0 | 0.0 | QNaN | 0.0 | 0.0 | ymm4 |

vandnps ymm5,ymm1,ymm3

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1086.86 | 84.94 | 0.0 | 1158.11 | 172.03 | 0.0 | 1256.63 | 181.45 | ymm5 |

vorps ymm5,ymm4,ymm5

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1086.86 | 84.94 | QNaN | 1158.11 | 172.03 | QNaN | 1256.63 | 181.45 | ymm5 |

***Figure 9-1.*** *Surface area QNaN assignment for spheres with radius less than 0.0*

Following the calculation of the surface areas, the `vmulps ymm2,ymm3,ymm0` and `vdivps ymm3,ymm2,ymm7` instructions compute the sphere volumes. The processing loop uses another `vandps`, `vandnps`, and `vorps` instruction sequence to set the volume of any negative-radius sphere to `c_QNaN_F32`. These values are then saved to the array `vol`. The processing loop repeats until there are fewer than eight remaining radii.

The next block of code computes sphere surface areas and volumes for the remaining (1–7) radii. Note that `AvxCalcSphereAreaVolume_` carries out these calculations using scalar single-precision floating-point arithmetic. The scalar processing loop performs the same arithmetic and Boolean operations as the packed processing loop. Similar to the previous example, `AvxCalcSphereAreaVolume_` uses a `vzeroupper` instruction immediately after the scalar processing loop. This instruction is needed since `AvxCalcSphereAreaVolume_` carried out its calculations using the YMM register set. When a `vzeroupper` instruction is required, it should always be positioned before any function epilog macros (e.g., `_RestoreXmmRegs` and `_DeleteFrame`) and the `ret` instruction. Here are the results for source code example `Ch09_02`:

```
Results for AvxCalcSphereAreaVolume
 0:   3.80 |   181.458389     181.458389 |   229.847290     229.847290
 1:  10.00 |  1256.637085    1256.637085 |  4188.790527    4188.790527
 2:  -6.10 |          nan            nan |          nan            nan
 3:   3.70 |   172.033630     172.033630 |   212.174805     212.174805
 4:   9.60 |  1158.116821    1158.116821 |  3705.973877    3705.973877
 5:  -6.60 |          nan            nan |          nan            nan
 6:   2.60 |    84.948662      84.948654 |    73.622169      73.622162 Compare discrepancy
 7:   9.30 |  1086.865479    1086.865479 |  3369.283203    3369.283203
 8:   9.00 |  1017.876038    1017.876038 |  3053.628174    3053.628174
 9:   5.80 |   422.732758     422.732758 |   817.283386     817.283386
10:  -2.90 |          nan            nan |          nan            nan
11:   8.10 |   824.479675     824.479675 |  2226.095215    2226.095215
12:   3.00 |   113.097336     113.097336 |   113.097328     113.097328
13:   8.00 |   804.247742     804.247742 |  2144.660645    2144.660645
14:   1.40 |    24.630087      24.630085 |    11.494040      11.494039 Compare discrepancy
15:  -1.80 |          nan            nan |          nan            nan
16:   4.30 |   232.352219     232.352219 |   333.038177     333.038177
17:   6.60 |   547.391113     547.391113 |  1204.260376    1204.260376
18:   4.50 |   254.469009     254.469009 |   381.703522     381.703522
19:  -1.20 |          nan            nan |          nan            nan
20:   4.50 |   254.469009     254.469009 |   381.703522     381.703522
```

The output for source code example Ch09_02 includes a couple of lines with the text "compare discrepancy." This text was generated by the compare code in AvxCalcSphereAreaVolume to exemplify the non-associativity of floating-point arithmetic. In this example, the functions AvxCalcSphereAreaVolumeCpp and AvxCalcSphereAreaVolume_ carried out their respective floating-point calculations using different operands orderings. For each sphere surface area, the C++ code calculates `sa[i] = r[i] * r[i] * 4.0 * c_PI_F32`, while the assembly language code calculates `sa[i] = 4.0 * c_PI_F32 * r[i] * r[i]`. Tiny numerical discrepancies like this are not unusual when comparing floating-point values that are calculated using different operand orderings irrespective of the programming language. This is something that you should keep in mind if you're developing production code that includes multiple versions of the same calculating function (e.g., one coded using C++ and an AVX/AVX2 accelerated version that's implemented using x86 assembly language).

Finally, you may have noticed that the function AvxCalcSphereAreaVolume_ handled invalid radii sans any x86 conditional jump instructions. Minimizing the number of conditional jump instructions in a function, especially data-dependent ones, often results in faster executing code. You'll learn more about jump instruction optimization techniques in Chapter 15.

## Column Means

Listing 9-3 shows the source code for example Ch09_03. This example illustrates how to calculate the arithmetic mean of each column in a two-dimensional array of double-precision floating-point values.

***Listing 9-3.*** Example Ch09_03

```
//------------------------------------------------
//               Ch09_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
```

```
#include <iomanip>
#include <random>
#include <memory>

using namespace std;

extern "C" size_t c_NumRowsMax = 1024 * 1024;
extern "C" size_t c_NumColsMax = 1024 * 1024;

extern "C" bool AvxCalcColumnMeans_(const double* x, size_t nrows, size_t ncols, double*
col_means);

void Init(double* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 2000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (double)ui_dist(rng) / 10.0;
}

bool AvxCalcColumnMeansCpp(const double* x, size_t nrows, size_t ncols, double* col_means)
{
    // Make sure nrows and ncols are valid
    if (nrows == 0 || nrows > c_NumRowsMax)
        return false;
    if (ncols == 0 || ncols > c_NumColsMax)
        return false;

    // Set initial column means to zero
    for (size_t i = 0; i < ncols; i++)
        col_means[i] = 0.0;

    // Calculate column means
    for (size_t i = 0; i < nrows; i++)
    {
        for (size_t j = 0; j < ncols; j++)
            col_means[j] += x[i * ncols + j];
    }

    for (size_t j = 0; j < ncols; j++)
        col_means[j] /= nrows;

    return true;
}

void AvxCalcColumnMeans(void)
{
    const size_t nrows = 20;
    const size_t ncols = 11;
    unique_ptr<double[]> x {new double[nrows * ncols]};
```

```
    unique_ptr<double[]> col_means1 {new double[ncols]};
    unique_ptr<double[]> col_means2 {new double[ncols]};

    Init(x.get(), nrows * ncols, 47);

    bool rc1 = AvxCalcColumnMeansCpp(x.get(), nrows, ncols, col_means1.get());
    bool rc2 = AvxCalcColumnMeans_(x.get(), nrows, ncols, col_means2.get());

    cout << "Results for AvxCalcColumnMeans\n";

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code: ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    cout << "\nTest Matrix\n";
    cout << fixed << setprecision(1);

    for (size_t i = 0; i < nrows; i++)
    {
        cout << "row " << setw(2) << i;

        for (size_t j = 0; j < ncols; j++)
            cout << setw(7) << x[i * ncols + j];
        cout << '\n';
    }

    cout << "\nColumn Means\n";
    cout << setprecision(2);

    for (size_t j = 0; j < ncols; j++)
    {
        cout << "col_means1[" << setw(2) << j << "] =";
        cout << setw(10) << col_means1[j] << "   ";
        cout << "col_means2[" << setw(2) << j << "] =";
        cout << setw(10) << col_means2[j] << '\n';
    }
}

int main()
{
    AvxCalcColumnMeans();
    return 0;
}

;-------------------------------------------------
;               Ch09_03.asm
;-------------------------------------------------

; extern "C" bool AvxCalcColMeans_(const double* x, size_t nrows, size_t ncols, double*
col_means)
```

```
        extern c_NumRowsMax:qword
        extern c_NumColsMax:qword

        .code
AvxCalcColumnMeans_ proc

; Validate nrows and ncols
        xor eax,eax                         ;error return code (also col_mean index)
        test rdx,rdx
        jz Done                             ;jump if nrows is zero
        cmp rdx,[c_NumRowsMax]
        ja Done                             ;jump if nrows is too large
        test r8,r8
        jz Done                             ;jump if ncols is zero
        cmp r8,[c_NumColsMax]
        ja Done                             ;jump if ncols is too large

; Initialize elements of col_means to zero
        vxorpd xmm0,xmm0,xmm0               ;xmm0[63:0] = 0.0
@@:     vmovsd real8 ptr[r9+rax*8],xmm0     ;col_means[i] = 0.0
        inc rax
        cmp rax,r8
        jb @B                               ;repeat until done

        vcvtsi2sd xmm2,xmm2,rdx             ;convert nrows for later use

; Compute the sum of each column in x
LP1:    mov r11,r9                          ;r11 = ptr to col_means
        xor r10,r10                         ;r10 = col_index

LP2:    mov rax,r10                         ;rax = col_index
        add rax,4
        cmp rax,r8                          ;4 or more columns remaining?
        ja @F                               ;jump if no (col_index + 4 > ncols)

; Update col_means using next four columns
        vmovupd ymm0,ymmword ptr [rcx]      ;load next 4 columns of current row
        vaddpd ymm1,ymm0,ymmword ptr [r11]  ;add to col_means
        vmovupd ymmword ptr [r11],ymm1      ;save updated col_means
        add r10,4                           ;col_index += 4
        add rcx,32                          ;update x ptr
        add r11,32                          ;update col_means ptr
        jmp NextColSet

@@:     sub rax,2
        cmp rax,r8                          ;2 or more columns remaining?
        ja @F                               ;jump if no (col_index + 2 > ncols)

; Update col_means using next two columns
        vmovupd xmm0,xmmword ptr [rcx]      ;load next 2 columns of current row
        vaddpd xmm1,xmm0,xmmword ptr [r11]  ;add to col_means
```

```
        vmovupd xmmword ptr [r11],xmm1      ;save updated col_means
        add r10,2                           ;col_index += 2
        add rcx,16                          ;update x ptr
        add r11,16                          ;update col_means ptr
        jmp NextColSet

; Update col_means using next column (or last column in the current row)
@@:     vmovsd xmm0,real8 ptr [rcx]         ;load x from last column
        vaddsd xmm1,xmm0,real8 ptr [r11]    ;add to col_means
        vmovsd real8 ptr [r11],xmm1         ;save updated col_means
        inc r10                             ;col_index += 1
        add rcx,8                           ;update x ptr

NextColSet:
        cmp r10,r8                          ;more columns in current row?
        jb LP2                              ;jump if yes
        dec rdx                             ;nrows -= 1
        jnz LP1                             ;jump if more rows

; Compute the final col_means
@@:     vmovsd xmm0,real8 ptr [r9]          ;xmm0 = col_means[i]
        vdivsd xmm1,xmm0,xmm2               ;compute final mean
        vmovsd real8 ptr [r9],xmm1          ;save col_mean[i]
        add r9,8                            ;update col_means ptr
        dec r8                              ;ncols -= 1
        jnz @B                              ;repeat until done

        mov eax,1                           ;set success return code

Done:   vzeroupper
        ret

AvxCalcColumnMeans_ endp
        end
```

Toward the top of the C++ code is a function named AvxCalcColumnMeansCpp. This function calculates the column means of a two-dimensional array using a straightforward set of nested for loops and some simple arithmetic. The function AvxCalcColumnMeans contains code that uses the C++ smart pointer class unique_ptr<> to help manage its dynamically-allocated arrays. Note that storage space for the test array x is allocated using the C++ new operator, which means that the array may not be aligned on a 16- or 32-byte boundary. In this particular example, aligning the start of array x to a specific boundary would be of little benefit since it's not possible to align the individual rows or columns of a standard C++ two-dimensional array (recall that the elements of a two-dimensional C++ array are stored in a contiguous block of memory using row-major ordering as described in Chapter 2).

The function AvxCalcColumnMeans also uses class unique_ptr<> and the new operator for the one-dimensional arrays col_means1 and col_means2. Using unique_ptr<> in this example simplifies the C++ code somewhat since its destructor automatically invokes the delete[] operator to release the storage space that was allocated by the new operator. If you're interested in learning more about the smart pointer class unique_ptr<>, Appendix A contains a list of C++ references that you can consult. The remaining code in AvxCalcColumnMeans invokes the C++ and assembly language column-mean calculating functions and streams the results to cout.

Following argument validation, the assembly language function AvxCalcColMeans_ initializes each element in col_means to 0.0. These elements will maintain the intermediate column sums. In order to maximize throughput, the column summation code uses slightly different instruction sequences depending on the current column and the total number of columns in the array. For example, assume that array x contains seven columns. For each row, the elements of the first four columns in x can be added to col_means using 256-bit wide packed addition; the elements of the next two columns can be added to col_means using 128-bit wide packed addition; and the final column element must be added to col_means using scalar addition. Figure 9-2 illustrates this technique in greater detail.

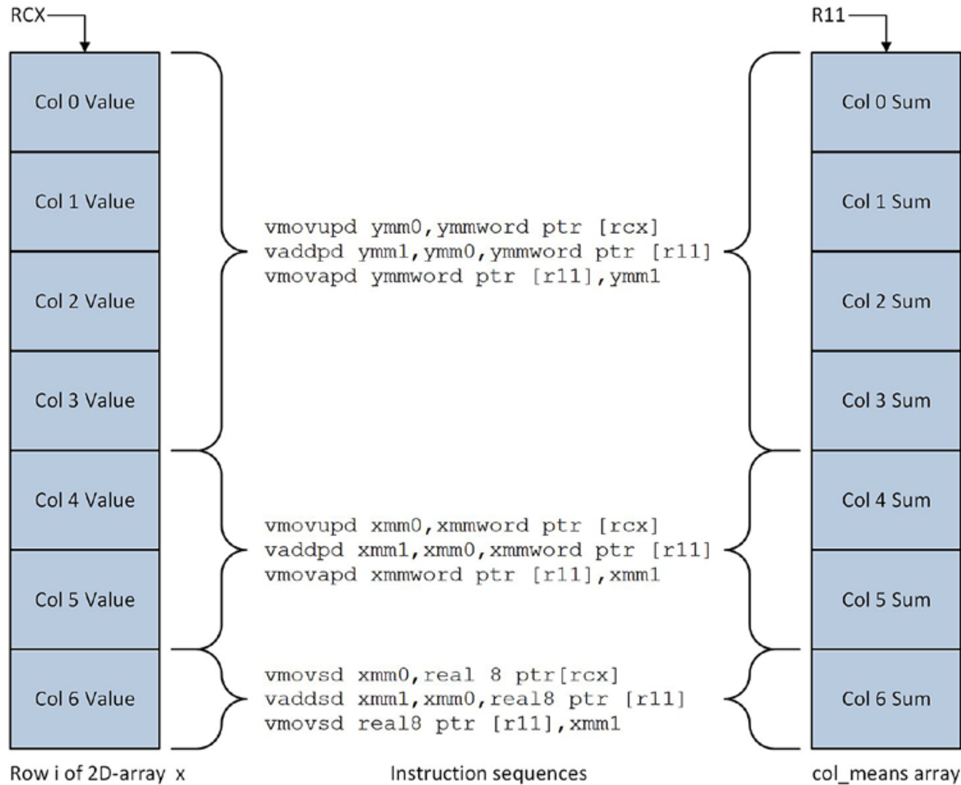



***Figure 9-2.*** *Updating the col_means array using different operand sizes*

The mov r11,r9 instruction next to the label LP1 is the starting point for adding elements in the current row of x to col_means. This instruction initializes R11 to first entry in col_means. The col_index counter in register R10 is then set to zero. The instruction group near the label LP2 determines the number of columns remaining to be processed in the current row. If four or more columns remain, the next four elements from the current row are added to the column sums in col_means. A vmovupd ymm0,ymmword ptr [rcx] instruction loads four double-precision floating-point values from x into YMM0 (a vmovapd instruction is not used here since alignment of the elements is unknown). The ensuing vaddpd ymm1,ymm0,ymmword ptr [r11] instruction sums the current array elements with the corresponding elements in col_means, and the vmovupd ymmword ptr [r11],ymm1 instruction saves the updated results back to col_means. The function's various pointers and counters are then updated in preparation for the next set of elements from the current row of x.

The summation code repeats the steps described in the previous paragraph until the number of array elements that remain in the current row is less than four. As soon as this condition is met, the elements in the remaining columns (if any) must be processed using 128-bit wide or 64-bit wide operands. This is the reason for the distinct blocks of code in `AvxCalcColumnMeans_` that process four elements, two elements, or a single element per row. Following computation of the column sums, each element in `col_means` is divided by `n`, which yields the final column mean. Here are the results for source code example `Ch09_03`:

```
Results for AvxCalcColumnMeans

Test Matrix
row  0  125.6   59.9  100.0  170.5  140.1  197.2   73.7   15.2   92.4  155.3  159.2
row  1   77.6  105.4   45.0  176.8   65.9   12.3  189.1  102.0   56.2  112.8   17.2
row  2  198.9  199.3   74.6  137.9   65.0  125.0   19.8   32.1   58.6   94.1  123.5
row  3    1.7   29.1   99.1  200.0  109.0  123.7  130.0  125.3  146.2   90.6   52.2
row  4    8.7   88.7   84.8  174.6  164.4  106.2  114.0  151.8  130.8  101.9  116.2
row  5   42.7  130.5  180.4  199.4  196.6   99.7  163.6   34.2    5.5  146.1  108.5
row  6  120.0  159.5   26.0   83.4   58.7   10.1  170.1   20.5   10.8   48.3  121.9
row  7  148.9  148.4  142.0  106.6  198.4   60.3   72.1  137.8   74.5   75.7   44.8
row  8   25.7  192.0   12.1   23.4   98.7  145.3  196.8   43.9  143.1   25.1  122.6
row  9    5.4  134.7  165.1   61.8   46.7  183.3  173.7  146.9   76.5  186.2   24.9
row 10  174.5  158.9  127.8   58.9   42.9  182.9    7.8   50.3   68.0   62.0   66.1
row 11   47.3  166.2    8.2   71.2   98.5   12.4  179.0  100.2   29.7  167.4  155.2
row 12   23.9  196.6  148.7    7.1  128.2  128.8   66.3  153.7   60.7  115.4   71.6
row 13  103.4  184.3  161.5   57.9  199.2   79.3   28.1   73.1   12.5   71.3  100.4
row 14  130.3  154.2  127.5   29.7  198.2  170.3  121.9   80.4  159.8   70.0   82.6
row 15   26.7   45.6   67.7  109.7    5.1   96.2  188.7  100.7   48.3  164.2   75.4
row 16  115.4   25.5   58.8  148.5   80.7  149.1  156.7  153.8   42.0  103.7    4.2
row 17   67.9  161.5   16.9  102.1   77.3    3.9  104.7   97.2  181.8  182.0  155.1
row 18  169.5  122.4  102.2    5.5   14.5  105.1  181.5   83.3  117.6   52.1  111.2
row 19   47.1  146.9   21.0    8.6  130.3   24.7   95.7    6.7  159.9   38.8   82.6

Column Means
col_means1[ 0] =     83.06   col_means2[ 0] =     83.06
col_means1[ 1] =    130.48   col_means2[ 1] =    130.48
col_means1[ 2] =     88.47   col_means2[ 2] =     88.47
col_means1[ 3] =     96.68   col_means2[ 3] =     96.68
col_means1[ 4] =    105.92   col_means2[ 4] =    105.92
col_means1[ 5] =    100.79   col_means2[ 5] =    100.79
col_means1[ 6] =    121.66   col_means2[ 6] =    121.66
col_means1[ 7] =     85.46   col_means2[ 7] =     85.46
col_means1[ 8] =     83.75   col_means2[ 8] =     83.75
col_means1[ 9] =    103.15   col_means2[ 9] =    103.15
col_means1[10] =     89.77   col_means2[10] =     89.77
```

## Correlation Coefficient

The next source code example illustrates how to calculate a correlation coefficient using packed double-precision floating-point arithmetic. This example also demonstrates how to perform a few common auxiliary operations with packed floating-point operands, including 128-bit wide extractions and horizontal addition. Listing 9-4 shows the source code for example Ch09_04.

***Listing 9-4.*** Example Ch09_04

```
//------------------------------------------------
//               Ch09_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <string>
#include <random>
#include "AlignedMem.h"

using namespace std;

extern "C" bool AvxCalcCorrCoef_(const double* x, const double* y, size_t n, double sums[5],
double epsilon, double* rho);

void Init(double* x, double* y, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 999};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
    {
        x[i] = (double)ui_dist(rng);
        y[i] = x[i] + (ui_dist(rng) % 6000) - 3000;
    }
}

bool AvxCalcCorrCoefCpp(const double* x, const double* y, size_t n, double sums[5], double
epsilon, double* rho)
{
    const size_t alignment = 32;

    // Make sure n is valid
    if (n == 0)
        return false;

    // Make sure x and y are properly aligned
    if (!AlignedMem::IsAligned(x, alignment))
        return false;
    if (!AlignedMem::IsAligned(y, alignment))
        return false;
```

```
    // Calculate and save sum variables
    double sum_x = 0, sum_y = 0, sum_xx = 0, sum_yy = 0, sum_xy = 0;

    for (size_t i = 0; i < n; i++)
    {
        sum_x += x[i];
        sum_y += y[i];
        sum_xx += x[i] * x[i];
        sum_yy += y[i] * y[i];
        sum_xy += x[i] * y[i];
    }

    sums[0] = sum_x;
    sums[1] = sum_y;
    sums[2] = sum_xx;
    sums[3] = sum_yy;
    sums[4] = sum_xy;

    // Calculate rho
    double rho_num = n * sum_xy - sum_x * sum_y;
    double rho_den = sqrt(n * sum_xx - sum_x * sum_x) * sqrt(n * sum_yy - sum_y * sum_y);

    if (rho_den >= epsilon)
    {
        *rho = rho_num / rho_den;
        return true;
    }
    else
    {
        *rho = 0;
        return false;
    }
}

int main()
{
    const size_t n = 103;
    const size_t alignment = 32;
    AlignedArray<double> x_aa(n, alignment);
    AlignedArray<double> y_aa(n, alignment);
    double sums1[5], sums2[5];
    double rho1, rho2;
    double epsilon = 1.0e-12;
    double* x = x_aa.Data();
    double* y = y_aa.Data();

    Init(x, y, n, 71);

    bool rc1 = AvxCalcCorrCoefCpp(x, y, n, sums1, epsilon, &rho1);
    bool rc2 = AvxCalcCorrCoef_(x, y, n, sums2, epsilon, &rho2);
```

```
    cout << "Results for AvxCalcCorrCoef\n\n";

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return 1;
    }

    int w = 14;
    string sep(w * 3, '-');

    cout << fixed << setprecision(8);
    cout << "Value    " << setw(w) << "C++" << " " << setw(w) << "x86-AVX" << '\n';
    cout << sep << '\n';
    cout << "rho:     " << setw(w) << rho1 << " " << setw(w) << rho2 << "\n\n";

    cout << setprecision(1);
    cout << "sum_x:   " << setw(w) << sums1[0] << " " << setw(w) << sums2[0] << '\n';
    cout << "sum_y:   " << setw(w) << sums1[1] << " " << setw(w) << sums2[1] << '\n';
    cout << "sum_xx:  " << setw(w) << sums1[2] << " " << setw(w) << sums2[2] << '\n';
    cout << "sum_yy:  " << setw(w) << sums1[3] << " " << setw(w) << sums2[3] << '\n';
    cout << "sum_xy:  " << setw(w) << sums1[4] << " " << setw(w) << sums2[4] << '\n';
    return 0;
}

;-------------------------------------------------
;               Ch09_04.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; extern "C" bool AvxCalcCorrCoef_(const double* x, const double* y, size_t n, double
sums[5], double epsilon, double* rho)
;
; Returns       0 = error, 1 = success

        .code
AvxCalcCorrCoef_ proc frame
        _CreateFrame CC_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Validate arguments
        or r8,r8
        jz BadArg                           ;jump if n == 0
        test rcx,1fh
        jnz BadArg                          ;jump if x is not aligned
        test rdx,1fh
        jnz BadArg                          ;jump if y is not aligned
```

```
; Initialize sum variables to zero
        vxorpd ymm3,ymm3,ymm3               ;ymm3 = packed sum_x
        vxorpd ymm4,ymm4,ymm4               ;ymm4 = packed sum_y
        vxorpd ymm5,ymm5,ymm5               ;ymm5 = packed sum_xx
        vxorpd ymm6,ymm6,ymm6               ;ymm6 = packed sum_yy
        vxorpd ymm7,ymm7,ymm7               ;ymm7 = packed sum_xy
        mov r10,r8                          ;r10 = n

        cmp r8,4
        jb LP2                              ;jump if n >= 1 && n <= 3

; Calculate intermediate packed sum variables
LP1:    vmovapd ymm0,ymmword ptr [rcx]      ;ymm0 = packed x values
        vmovapd ymm1,ymmword ptr [rdx]      ;ymm1 = packed y values

        vaddpd ymm3,ymm3,ymm0               ;update packed sum_x
        vaddpd ymm4,ymm4,ymm1               ;update packed sum_y

        vmulpd ymm2,ymm0,ymm1               ;ymm2 = packed xy values
        vaddpd ymm7,ymm7,ymm2               ;update packed sum_xy

        vmulpd ymm0,ymm0,ymm0               ;ymm0 = packed xx values
        vmulpd ymm1,ymm1,ymm1               ;ymm1 = packed yy values
        vaddpd ymm5,ymm5,ymm0               ;update packed sum_xx
        vaddpd ymm6,ymm6,ymm1               ;update packed sum_yy

        add rcx,32                          ;update x ptr
        add rdx,32                          ;update y ptr
        sub r8,4                            ;n -= 4
        cmp r8,4                            ;is n >= 4?
        jae LP1                             ;jump if yes

        or r8,r8                            ;is n == 0?
        jz FSV                              ;jump if yes

; Update sum variables with final x & y values
LP2:    vmovsd xmm0,real8 ptr [rcx]         ;xmm0[63:0] = x[i], ymm0[255:64] = 0
        vmovsd xmm1,real8 ptr [rdx]         ;xmm1[63:0] = y[i], ymm1[255:64] = 0

        vaddpd ymm3,ymm3,ymm0               ;update packed sum_x
        vaddpd ymm4,ymm4,ymm1               ;update packed sum_y

        vmulpd ymm2,ymm0,ymm1               ;ymm2 = packed xy values
        vaddpd ymm7,ymm7,ymm2               ;update packed sum_xy

        vmulpd ymm0,ymm0,ymm0               ;ymm0 = packed xx values
        vmulpd ymm1,ymm1,ymm1               ;ymm1 = packed yy values
        vaddpd ymm5,ymm5,ymm0               ;update packed sum_xx
        vaddpd ymm6,ymm6,ymm1               ;update packed sum_yy
```

```
        add rcx,8                           ;update x ptr
        add rdx,8                           ;update y ptr
        sub r8,1                            ;n -= 1
        jnz LP2                             ;repeat until done

; Calculate final sum variables
FSV:    vextractf128 xmm0,ymm3,1
        vaddpd xmm1,xmm0,xmm3
        vhaddpd xmm3,xmm1,xmm1              ;xmm3[63:0] = sum_x

        vextractf128 xmm0,ymm4,1
        vaddpd xmm1,xmm0,xmm4
        vhaddpd xmm4,xmm1,xmm1              ;xmm4[63:0] = sum_y

        vextractf128 xmm0,ymm5,1
        vaddpd xmm1,xmm0,xmm5
        vhaddpd xmm5,xmm1,xmm1              ;xmm5[63:0] = sum_xx

        vextractf128 xmm0,ymm6,1
        vaddpd xmm1,xmm0,xmm6
        vhaddpd xmm6,xmm1,xmm1              ;xmm6[63:0] = sum_yy

        vextractf128 xmm0,ymm7,1
        vaddpd xmm1,xmm0,xmm7
        vhaddpd xmm7,xmm1,xmm1              ;xmm7[63:0] = sum_xy

; Save final sum variables
        vmovsd real8 ptr [r9],xmm3          ;save sum_x
        vmovsd real8 ptr [r9+8],xmm4        ;save sum_y
        vmovsd real8 ptr [r9+16],xmm5       ;save sum_xx
        vmovsd real8 ptr [r9+24],xmm6       ;save sum_yy
        vmovsd real8 ptr [r9+32],xmm7       ;save sum_xy

; Calculate rho numerator
; rho_num = n * sum_xy - sum_x * sum_y;
        vcvtsi2sd xmm2,xmm2,r10             ;xmm2 = n
        vmulsd xmm0,xmm2,xmm7               ;xmm0 = = n * sum_xy
        vmulsd xmm1,xmm3,xmm4               ;xmm1 = sum_x * sum_y
        vsubsd xmm7,xmm0,xmm1               ;xmm7 = rho_num

; Calculate rho denominator
; t1 = sqrt(n * sum_xx - sum_x * sum_x)
; t2 = sqrt(n * sum_yy - sum_y * sum_y)
; rho_den = t1 * t2
        vmulsd xmm0,xmm2,xmm5       ;xmm0 = n * sum_xx
        vmulsd xmm3,xmm3,xmm3       ;xmm3 = sum_x * sum_x
        vsubsd xmm3,xmm0,xmm3       ;xmm3 = n * sum_xx - sum_x * sum_x
        vsqrtsd xmm3,xmm3,xmm3      ;xmm3 = t1

        vmulsd xmm0,xmm2,xmm6       ;xmm0 = n * sum_yy
        vmulsd xmm4,xmm4,xmm4       ;xmm4 = sum_y * sum_y
```

```
        vsubsd xmm4,xmm0,xmm4       ;xmm4 = n * sum_yy - sum_y * sum_y
        vsqrtsd xmm4,xmm4,xmm4      ;xmm4 = t2

        vmulsd xmm0,xmm3,xmm4       ;xmm0 = rho_den

; Calculate and save final rho
        xor eax,eax
        vcomisd xmm0,real8 ptr [rbp+CC_OffsetStackArgs] ;rho_den < epsilon?
        setae al                                ;set return code
        jb BadRho                               ;jump if rho_den < epsilon
        vdivsd xmm1,xmm7,xmm0                   ;xmm1 = rho

SavRho: mov rdx,[rbp+CC_OffsetStackArgs+8]      ;rdx = ptr to rho
        vmovsd real8 ptr [rdx],xmm1             ;save rho

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret

; Error handling code
BadRho: vxorpd xmm1,xmm1,xmm1               ;rho = 0
        jmp SavRho

BadArg: xor eax,eax                         ;eax = invalid arg ret code
        jmp Done

AvxCalcCorrCoef_ endp
        end
```

A correlation coefficient measures the strength of association between two variables. Correlation coefficients can range in value from -1.0 to +1.0, signifying a perfect negative or positive relationship between the two variables. Real-world correlation coefficients are rarely equal to these theoretical limits. A correlation coefficient of 0.0 indicates that the data variables are not associated. The C++ and assembly language code in this example calculate the well-known Pearson correlation coefficient using the following equation:

$$\rho = \frac{n\sum_i x_i y_i - \sum_i x_i \sum_i y_i}{\sqrt{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2}\sqrt{n\sum_i y_i^2 - \left(\sum_i y_i\right)^2}}$$

In order to calculate a correlation coefficient using this formula, a function must compute the following five sum variables:

$$sum_x = \sum_i x_i$$

$$sum_y = \sum_i y_i$$

$$sum_xx = \sum_i x_i^2$$

$$sum_yy = \sum_i y_i^2$$

$$sum_xy = \sum_i x_i y_i$$

The C++ function AvxCalcCorrCoefCpp shows how to calculate a correlation coefficient. This function begins by checking the value of n to make sure it's greater than zero. It also validates the two data arrays x and y for proper alignment. The aforementioned sum variables are then calculated using a simple for loop. Following completion of the for loop, the function AvxCalcCorrCoefCpp saves the sum variables to the array sums for comparison and display purposes. It then computes the intermediate values rho_num and rho_den. Before computing the final correlation coefficient rho, rho_den is tested to confirm that it's greater than or equal to epsilon.

Following its prolog, the assembly language function AvxCalcCorrCoef_ performs the same size and alignment checks as its C++ counterpart. It then initializes packed versions of sum_x, sum_y, sum_xx, sum_yy, and sum_xy to zero in registers YMM3–YMM7. During each iteration, the loop labeled LP1 processes four elements from arrays x and y using packed double-precision floating-point arithmetic. This means that registers YMM3–YMM7 maintain four distinct intermediate values for each sum variable. Execution of loop LP1 continues until there are fewer than four elements remaining to process.

Following completion of loop LP1, the loop labeled LP2 processes the final (1–3) entries in arrays x and y. The vmovsd xmm0,real8 ptr [rcx] and vmovsd xmm1,real8 ptr [rdx] instructions load x[i] and y[i] into registers XMM0 and XMM1, respectively. Note that these vmovsd instructions also zero out bits YMM0[255:64] and YMM1[255:64], which means that the same chain of vaddpd and vmulpd instructions used in loop LP1 to update the intermediate sum variables can also be used in loop LP2 (the scalar instructions vaddsd and vmulsd cannot be used here to update the sum variables without extra code since these instructions set bits 255:128 of their destination operand register to zero). Following completion of loop LP2, each packed sum variable is reduced to a single value using a vextractf128, vaddpd, and vhaddpd instruction, as illustrated in Figure 9-3. The final sum values are then saved to the sums array.

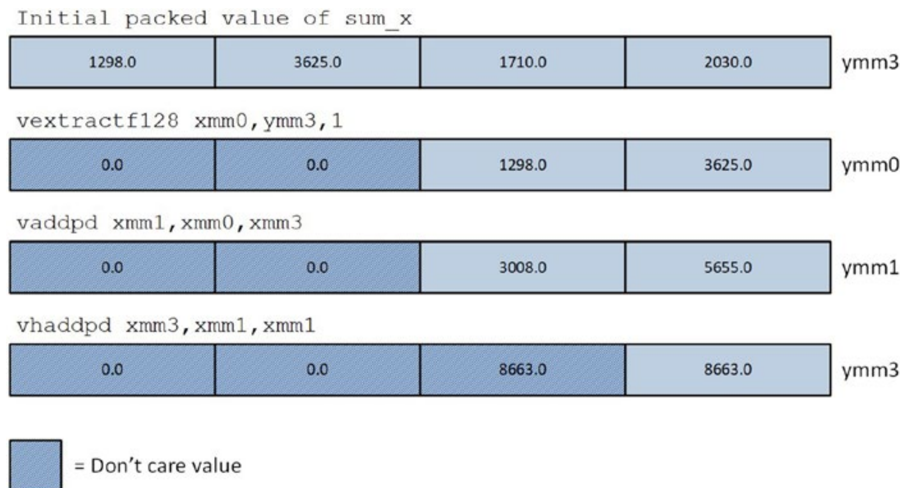



***Figure 9-3.*** *Calculation of sum_x using vextractf128, vaddpd, and vhaddpd*

Function AvxCalcCorrCoef_ uses simple scalar arithmetic to compute the intermediate values rho_num and rho_den. Like the corresponding C++ function, AvxCalcCorrCoef_ compares rho_den to see if it's less than epsilon (a value below epsilon is likely a rounding error and considered too close to zero to be valid). If rho_den is valid, the correlation coefficient rho is calculated and saved. Here are the results for source code example Ch09_04:

```
Results for AvxCalcCorrCoef

Value               C++         x86-AVX
-----------------------------------------
rho:          0.70128193     0.70128193

sum_x:           53081.0        53081.0
sum_y:         -199158.0      -199158.0
sum_xx:       35732585.0     35732585.0
sum_yy:      401708868.0    401708868.0
sum_xy:      -94360528.0    -94360528.0
```

# Matrix Multiplication and Transposition

In Chapter 6, you learned how to perform 4 × 4 matrix transposition and multiplication using single-precision floating-point values (see source code examples Ch06_07 and Ch06_08). The source code example in this section illustrates how to carry out these same matrix operations using double-precision floating-point values. Listing 9-5 shows the source code for example Ch09_05. The fundamentals of matrix transposition and multiplication are explained in Chapter 6. If your understanding of these mathematical operations is lacking, you may want to review the relevant sections in Chapter 6 before proceeding.

***Listing 9-5.*** Example Ch09_05

```
//-------------------------------------------------
//               Ch09_05.cpp
//-------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "Ch09_05.h"
#include "Matrix.h"

using namespace std;

void AvxMat4x4TransposeF64(Matrix<double>& m_src1)
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<double> m_des1(nr ,nc);
    Matrix<double> m_des2(nr ,nc);
```

```
    Matrix<double>::Transpose(m_des1, m_src1);
    AvxMat4x4TransposeF64_(m_des2.Data(), m_src1.Data());

    cout << fixed << setprecision(1);
    m_src1.SetOstream(12, "  ");
    m_des1.SetOstream(12, "  ");
    m_des2.SetOstream(12, "  ");

    cout << "Results for AvxMat4x4TransposeF64\n";
    cout << "Matrix m_src1\n" << m_src1 << '\n';
    cout << "Matrix m_des1\n" << m_des1 << '\n';
    cout << "Matrix m_des2\n" << m_des2 << '\n';

    if (m_des1 != m_des2)
        cout << "\nMatrix compare failed - AvxMat4x4TransposeF64\n";
}

void AvxMat4x4MulF64(Matrix<double>& m_src1, Matrix<double>& m_src2)
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<double> m_des1(nr ,nc);
    Matrix<double> m_des2(nr ,nc);

    Matrix<double>::Mul(m_des1, m_src1, m_src2);
    AvxMat4x4MulF64_(m_des2.Data(), m_src1.Data(), m_src2.Data());

    cout << fixed << setprecision(1);

    m_src1.SetOstream(12, "  ");
    m_src2.SetOstream(12, "  ");
    m_des1.SetOstream(12, "  ");
    m_des2.SetOstream(12, "  ");

    cout << "\nResults for AvxMat4x4MulF64\n";
    cout << "Matrix m_src1\n" << m_src1 << '\n';
    cout << "Matrix m_src2\n" << m_src2 << '\n';
    cout << "Matrix m_des1\n" << m_des1 << '\n';
    cout << "Matrix m_des2\n" << m_des2 << '\n';

    if (m_des1 != m_des2)
        cout << "\nMatrix compare failed - AvxMat4x4MulF64\n";
}

int main()
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<double> m_src1(nr ,nc);
    Matrix<double> m_src2(nr ,nc);
```

```
    const double src1_row0[] = { 10, 11, 12, 13 };
    const double src1_row1[] = { 20, 21, 22, 23 };
    const double src1_row2[] = { 30, 31, 32, 33 };
    const double src1_row3[] = { 40, 41, 42, 43 };

    const double src2_row0[] = { 100, 101, 102, 103 };
    const double src2_row1[] = { 200, 201, 202, 203 };
    const double src2_row2[] = { 300, 301, 302, 303 };
    const double src2_row3[] = { 400, 401, 402, 403 };

    m_src1.SetRow(0, src1_row0);
    m_src1.SetRow(1, src1_row1);
    m_src1.SetRow(2, src1_row2);
    m_src1.SetRow(3, src1_row3);

    m_src2.SetRow(0, src2_row0);
    m_src2.SetRow(1, src2_row1);
    m_src2.SetRow(2, src2_row2);
    m_src2.SetRow(3, src2_row3);

    // Test functions
    AvxMat4x4TransposeF64(m_src1);
    AvxMat4x4MulF64(m_src1, m_src2);

    // Benchmark functions
    AvxMat4x4TransposeF64_BM();
    AvxMat4x4MulF64_BM();
    return 0;
}
```

```
;-------------------------------------------------
;               Ch09_05.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; _Mat4x4TransposeF64 macro
;
; Description:  This macro computes the transpose of a 4x4
;               double-precision floating-point matrix.
;
;  Input Matrix                    Output Matrix
;  ---------------------------------------------------
;  ymm0    a3 a2 a1 a0             ymm0    d0 c0 b0 a0
;  ymm1    b3 b2 b1 b0             ymm1    d1 c1 b1 a1
;  ymm2    c3 c2 c1 c0             ymm2    d2 c2 b2 a2
;  ymm3    d3 d2 d1 d0             ymm3    d3 c3 b3 a3
;
```

```
_Mat4x4TransposeF64 macro
        vunpcklpd ymm4,ymm0,ymm1            ;ymm4 = b2 a2 b0 a0
        vunpckhpd ymm5,ymm0,ymm1            ;ymm5 = b3 a3 b1 a1
        vunpcklpd ymm6,ymm2,ymm3            ;ymm6 = d2 c2 d0 c0
        vunpckhpd ymm7,ymm2,ymm3            ;ymm7 = d3 c3 d1 c1

        vperm2f128 ymm0,ymm4,ymm6,20h       ;ymm0 = d0 c0 b0 a0
        vperm2f128 ymm1,ymm5,ymm7,20h       ;ymm1 = d1 c1 b1 a1
        vperm2f128 ymm2,ymm4,ymm6,31h       ;ymm2 = d2 c2 b2 a2
        vperm2f128 ymm3,ymm5,ymm7,31h       ;ymm3 = d3 c3 b3 a3
        endm

; extern "C" void AvxMat4x4TransposeF64_(double* m_des, const double* m_src1)

        .code
AvxMat4x4TransposeF64_ proc frame
        _CreateFrame MT_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Transpose matrix m_src1
        vmovaps ymm0,[rdx]                  ;ymm0 = m_src1.row_0
        vmovaps ymm1,[rdx+32]               ;ymm1 = m_src2.row_1
        vmovaps ymm2,[rdx+64]               ;ymm2 = m_src3.row_2
        vmovaps ymm3,[rdx+96]               ;ymm3 = m_src4.row_3

        _Mat4x4TransposeF64

        vmovaps [rcx],ymm0                  ;save m_des.row_0
        vmovaps [rcx+32],ymm1               ;save m_des.row_1
        vmovaps [rcx+64],ymm2               ;save m_des.row_2
        vmovaps [rcx+96],ymm3               ;save m_des.row_3

        vzeroupper
Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4TransposeF64_ endp

; _Mat4x4MulCalcRowF64 macro
;
; Description:  This macro computes one row of a 4x4 matrix multiplication.
;
; Registers:    ymm0 = m_src2.row0
;               ymm1 = m_src2.row1
;               ymm2 = m_src2.row2
;               ymm3 = m_src2.row3
;               rcx = m_des ptr
;               rdx = m_src1 ptr
;               ymm4 - ymm4 = scratch registers
```

```
_Mat4x4MulCalcRowF64 macro disp
        vbroadcastsd ymm4,real8 ptr [rdx+disp]      ;broadcast m_src1[i][0]
        vbroadcastsd ymm5,real8 ptr [rdx+disp+8]    ;broadcast m_src1[i][1]
        vbroadcastsd ymm6,real8 ptr [rdx+disp+16]   ;broadcast m_src1[i][2]
        vbroadcastsd ymm7,real8 ptr [rdx+disp+24]   ;broadcast m_src1[i][3]

        vmulpd ymm4,ymm4,ymm0                       ;m_src1[i][0] * m_src2.row_0
        vmulpd ymm5,ymm5,ymm1                       ;m_src1[i][1] * m_src2.row_1
        vmulpd ymm6,ymm6,ymm2                       ;m_src1[i][2] * m_src2.row_2
        vmulpd ymm7,ymm7,ymm3                       ;m_src1[i][3] * m_src2.row_3

        vaddpd ymm4,ymm4,ymm5                       ;calc m_des.row_i
        vaddpd ymm6,ymm6,ymm7
        vaddpd ymm4,ymm4,ymm6

        vmovapd [rcx+disp],ymm4                     ;save m_des.row_i
        endm

; extern "C" void AvxMat4x4MulF64_(double* m_des, const double* m_src1, const double* m_
src2)

AvxMat4x4MulF64_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Load m_src2 into YMM3:YMM0
        vmovapd ymm0,[r8]                   ;ymm0 = m_src2.row_0
        vmovapd ymm1,[r8+32]                ;ymm1 = m_src2.row_1
        vmovapd ymm2,[r8+64]                ;ymm2 = m_src2.row_2
        vmovapd ymm3,[r8+96]                ;ymm3 = m_src2.row_3

; Compute matrix product
        _Mat4x4MulCalcRowF64 0              ;calculate m_des.row_0
        _Mat4x4MulCalcRowF64 32             ;calculate m_des.row_1
        _Mat4x4MulCalcRowF64 64             ;calculate m_des.row_2
        _Mat4x4MulCalcRowF64 96             ;calculate m_des.row_3

        vzeroupper
Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4MulF64_ endp
        end
```

The C++ source code that's shown in Listing 9-5 is very similar to what you saw in Chapter 6. It begins with a function named `AvxMat4x4TransposeF64` that exercises both the C++ and assembly language matrix transposition calculating routines and displays the results. The function that follows, `AvxMat4x4MulF64`, implements the same tasks for matrix multiplication. Similar to the source code examples in Chapter 6, the C++ versions of matrix transposition and multiplication are implemented by the template functions

`Matrix<>::Transpose` and `Matrix<>::Mul`, respectively. Chapter 6 contains additional details regarding these template functions.

Near the top of the assembly language code is a macro named `_Mat4x4TransposeF64`. This macro contains instructions that transpose a 4 × 4 matrix of double-precision floating-point values. The four rows of the source double-precision floating-point matrix must be loaded in registers YMM0–YMM3 prior to its use. Macro `_Mat4x4TransposeF64` uses the `vperm2f128` instruction to permute the 128-bit wide floating-point fields of its two source operands. This instruction uses an immediate 8-bit control mask to select which fields are copied from the source operands to the destination operand, as outlined in Table 9-1. Figure 9-4 shows the entire 4 × 4 matrix transposition operation in greater detail. The assembly language function `AvxMat4x4TransposeF64_` uses the macro `_Mat4x4TransposeF64` to transpose a 4 × 4 matrix of double-precision floating-point values.

***Table 9-1.*** *Field Selection for `vperm2f128 ymm0,ymm1,ymm2,imm8` Instruction*

| Destination Field | Source Field | imm8[1:0] | imm8[4:3] |
|---|---|---|---|
| ymm0[127:0] | ymm1[127:0] | 0 | |
| | ymm1[255:128] | 1 | |
| | ymm2[127:0] | 2 | |
| | ymm2[255:128] | 3 | |
| ymm0[255:128] | ymm1[127:0] | | 0 |
| | ymm1[255:128] | | 1 |
| | ymm2[127:0] | | 2 |
| | ymm2[255:128] | | 3 |

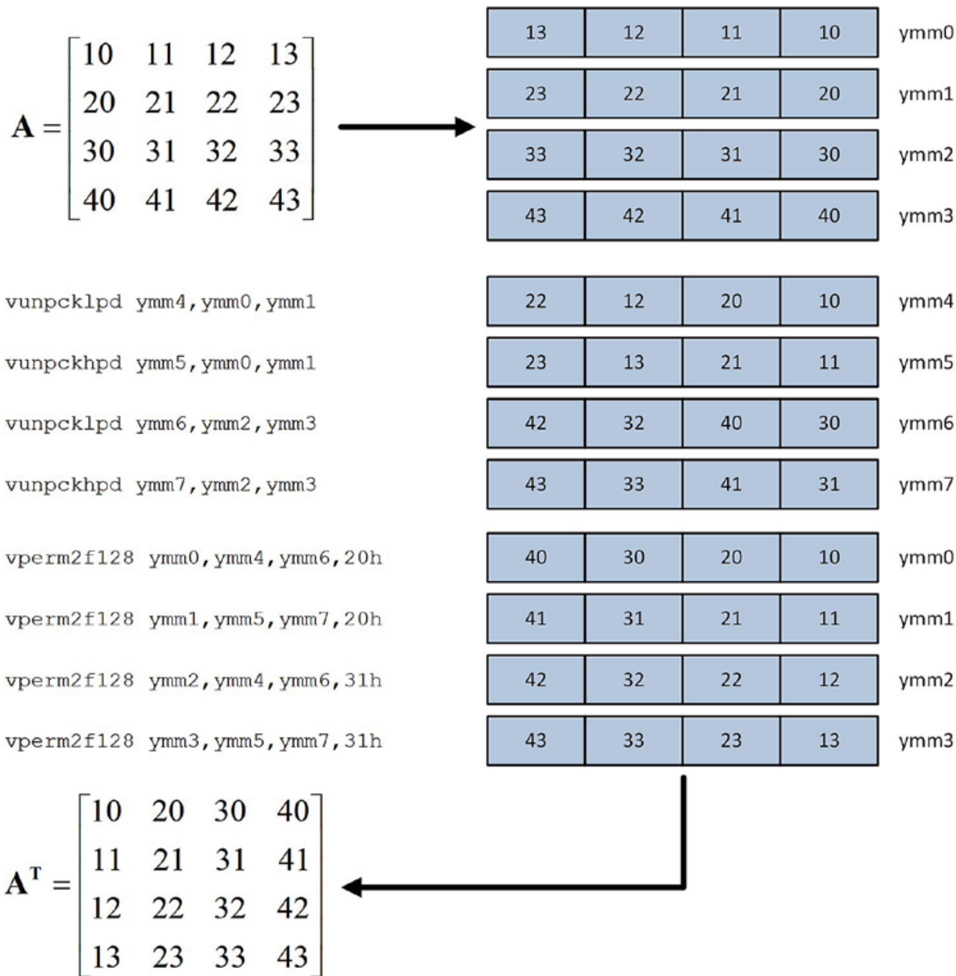



***Figure 9-4.*** *Instruction sequence used by `_Max4x4TransposeF64` to transpose a 4 × 4 matrix of double-precision floating-point values*

In Listing 9-5, the macro definition `_Mat4x4MulCalcRowF64` follows the function `AvxMat4x4TransposeF64_`. This macro contains instructions that calculate a single row of a 4 × 4 matrix multiplication. The row-multiplication technique that's used here is identical to the one that was used in source code example `Ch06_08` in Chapter 6 (see Figure 6-7). The function `AvxMat4x4MulF64_` uses the macro `_Mat4x4MulCalcRowF64` to multiply two 4 × 4 double-precision floating-point matrices. Here are the results for source code example `Ch09_05`:

```
Results for AvxMat4x4TransposeF64

Matrix m_src1
        10.0          11.0          12.0          13.0
        20.0          21.0          22.0          23.0
        30.0          31.0          32.0          33.0
        40.0          41.0          42.0          43.0

Matrix m_des1
        10.0          20.0          30.0          40.0
        11.0          21.0          31.0          41.0
        12.0          22.0          32.0          42.0
        13.0          23.0          33.0          43.0

Matrix m_des2
        10.0          20.0          30.0          40.0
        11.0          21.0          31.0          41.0
        12.0          22.0          32.0          42.0
        13.0          23.0          33.0          43.0

Results for AvxMat4x4MulF64

Matrix m_src1
        10.0          11.0          12.0          13.0
        20.0          21.0          22.0          23.0
        30.0          31.0          32.0          33.0
        40.0          41.0          42.0          43.0

Matrix m_src2
       100.0         101.0         102.0         103.0
       200.0         201.0         202.0         203.0
       300.0         301.0         302.0         303.0
       400.0         401.0         402.0         403.0

Matrix m_des1
     12000.0       12046.0       12092.0       12138.0
     22000.0       22086.0       22172.0       22258.0
     32000.0       32126.0       32252.0       32378.0
     42000.0       42166.0       42332.0       42498.0

Matrix m_des2
     12000.0       12046.0       12092.0       12138.0
     22000.0       22086.0       22172.0       22258.0
     32000.0       32126.0       32252.0       32378.0
     42000.0       42166.0       42332.0       42498.0

Running benchmark function AvxMat4x4TransposeF64_BM - please wait
Benchmark times save to file Ch09_05_AvxMat4x4TransposeF64_BM_CHROMIUM.csv

Running benchmark function AvxMat4x4MulF64_BM - please wait
Benchmark times save to file Ch09_05_AvxMat4x4MulF64_BM_CHROMIUM.csv
```

Tables 9-2 and 9-3 contain benchmark timing measurements for the matrix transposition and multiplication functions presented in this section. These measurements were made using the procedure that's described in Chapter 6.

***Table 9-2.*** *Matrix Transposition Mean Execution Times (Microseconds), 1,000,000 Transpositions*

| CPU | C++ | Assembly Language |
|---|---|---|
| i7-4790S | 15562 | 2670 |
| i9-7900X | 13167 | 2112 |
| i7-8700K | 12194 | 1963 |

***Table 9-3.*** *Matrix Multiplication Mean Execution Times (Microseconds), 1,000,000 Multiplications*

| CPU | C++ | Assembly Language |
|---|---|---|
| i7-4790S | 55652 | 5874 |
| i9-7900X | 46910 | 5286 |
| i7-8700K | 43118 | 4505 |

# Matrix Inversion

Besides transposition and multiplication, matrix inversion is another common operation that's often applied to 4 × 4 matrices. In this section, you examine a program that calculates the inverse of a 4 × 4 matrix of double-precision floating-point values. Listing 9-6 shows the source code for example Ch09_06.

***Listing 9-6.*** Example Ch09_06

```
//------------------------------------------------
//               Ch09_06.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cmath>
#include "Ch09_06.h"
#include "Matrix.h"

using namespace std;

bool Avx2Mat4x4InvF64Cpp(Matrix<double>& m_inv, const Matrix<double>& m, double epsilon,
bool* is_singular)
{
    // The intermediate matrices below are declared static for benchmarking purposes.
    static const size_t nrows = 4;
    static const size_t ncols = 4;
    static Matrix<double> m2(nrows, ncols);
    static Matrix<double> m3(nrows, ncols);
    static Matrix<double> m4(nrows, ncols);
```

```
    static Matrix<double> I(nrows, ncols, true);
    static Matrix<double> tempA(nrows, ncols);
    static Matrix<double> tempB(nrows, ncols);
    static Matrix<double> tempC(nrows, ncols);
    static Matrix<double> tempD(nrows, ncols);

    Matrix<double>::Mul(m2, m, m);
    Matrix<double>::Mul(m3, m2, m);
    Matrix<double>::Mul(m4, m3, m);

    double t1 = m.Trace();
    double t2 = m2.Trace();
    double t3 = m3.Trace();
    double t4 = m4.Trace();

    double c1 = -t1;
    double c2 = -1.0 / 2.0 * (c1 * t1 + t2);
    double c3 = -1.0 / 3.0 * (c2 * t1 + c1 * t2 + t3);
    double c4 = -1.0 / 4.0 * (c3 * t1 + c2 * t2 + c1 * t3 + t4);

    // Make sure matrix is not singular
    *is_singular = (fabs(c4) < epsilon);

    if (*is_singular)
        return false;

    // Calculate = -1.0 / c4 * (m3 + c1 * m2 + c2 * m + c3 * I)
    Matrix<double>::MulScalar(tempA, I, c3);
    Matrix<double>::MulScalar(tempB, m, c2);
    Matrix<double>::MulScalar(tempC, m2, c1);
    Matrix<double>::Add(tempD, tempA, tempB);
    Matrix<double>::Add(tempD, tempD, tempC);
    Matrix<double>::Add(tempD, tempD, m3);
    Matrix<double>::MulScalar(m_inv, tempD, -1.0 / c4);

    return true;
}

void Avx2Mat4x4InvF64(const Matrix<double>& m, const char* msg)
{
    cout << '\n' << msg << " - Test Matrix\n";
    cout << m << '\n';

    const double epsilon = 1.0e-9;
    const size_t nrows = m.GetNumRows();
    const size_t ncols = m.GetNumCols();
    Matrix<double> m_inv_a(nrows, ncols);
    Matrix<double> m_ver_a(nrows, ncols);
    Matrix<double> m_inv_b(nrows, ncols);
    Matrix<double> m_ver_b(nrows, ncols);
```

```
    for (int i = 0; i <= 1; i++)
    {
        string fn;
        const size_t nrows = m.GetNumRows();
        const size_t ncols = m.GetNumCols();
        Matrix<double> m_inv(nrows, ncols);
        Matrix<double> m_ver(nrows, ncols);
        bool rc, is_singular;

        if (i == 0)
        {
            fn = "Avx2Mat4x4InvF64Cpp";
            rc = Avx2Mat4x4InvF64Cpp(m_inv, m, epsilon, &is_singular);

            if (rc)
                Matrix<double>::Mul(m_ver, m_inv, m);
        }
        else
        {
            fn = "Avx2Mat4x4InvF64_";
            rc = Avx2Mat4x4InvF64_(m_inv.Data(), m.Data(), epsilon, &is_singular);

            if (rc)
                Avx2Mat4x4MulF64_(m_ver.Data(), m_inv.Data(), m.Data());
        }

        if (rc)
        {
            cout << msg << " - " << fn << " - Inverse Matrix\n";
            cout << m_inv << '\n';

            // Round to zero used for display purposes, can be removed.
            cout << msg << " - " << fn << " - Verify Matrix\n";
            m_ver.RoundToZero(epsilon);
            cout << m_ver << '\n';
        }
        else
        {
            if (is_singular)
                cout << msg << " - " << fn << " - Singular Matrix\n";
            else
                cout << msg << " - " << fn << " - Unexpected error occurred\n";
        }
    }
}
```

```
int main()
{
    cout << "\nResults for Avx2Mat4x4InvF64\n";

    // Test Matrix #1 - Non-Singular
    Matrix<double> m1(4, 4);
    const double m1_row0[] = { 2, 7, 3, 4 };
    const double m1_row1[] = { 5, 9, 6, 4.75 };
    const double m1_row2[] = { 6.5, 3, 4, 10 };
    const double m1_row3[] = { 7, 5.25, 8.125, 6 };
    m1.SetRow(0, m1_row0);
    m1.SetRow(1, m1_row1);
    m1.SetRow(2, m1_row2);
    m1.SetRow(3, m1_row3);

    // Test Matrix #2 - Non-Singular
    Matrix<double> m2(4, 4);
    const double m2_row0[] = { 0.5, 12, 17.25, 4 };
    const double m2_row1[] = { 5, 2, 6.75, 8 };
    const double m2_row2[] = { 13.125, 1, 3, 9.75 };
    const double m2_row3[] = { 16, 1.625, 7, 0.25 };
    m2.SetRow(0, m2_row0);
    m2.SetRow(1, m2_row1);
    m2.SetRow(2, m2_row2);
    m2.SetRow(3, m2_row3);

    // Test Matrix #3 - Singular
    Matrix<double> m3(4, 4);
    const double m3_row0[] = { 2, 0, 0, 1 };
    const double m3_row1[] = { 0, 4, 5, 0 };
    const double m3_row2[] = { 0, 0, 0, 7 };
    const double m3_row3[] = { 0, 0, 0, 6 };
    m3.SetRow(0, m3_row0);
    m3.SetRow(1, m3_row1);
    m3.SetRow(2, m3_row2);
    m3.SetRow(3, m3_row3);

    Avx2Mat4x4InvF64(m1, "Test #1");
    Avx2Mat4x4InvF64(m2, "Test #2");
    Avx2Mat4x4InvF64(m3, "Test #3");

    Avx2Mat4x4InvF64_BM(m1);
    return 0;
}
```

```
;-------------------------------------------------
;               Ch09_06.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; Custom segment for constants
ConstVals segment readonly align(32) 'const'
Mat4x4I real8 1.0, 0.0, 0.0, 0.0
        real8 0.0, 1.0, 0.0, 0.0
        real8 0.0, 0.0, 1.0, 0.0
        real8 0.0, 0.0, 0.0, 1.0

r8_SignBitMask  qword 4 dup (8000000000000000h)
r8_AbsMask      qword 4 dup (7fffffffffffffffh)

r8_1p0          real8 1.0
r8_N1p0         real8 -1.0
r8_N0p5         real8 -0.5
r8_N0p3333      real8 -0.33333333333333
r8_N0p25        real8 -0.25
ConstVals ends
        .code

; _Mat4x4TraceF64 macro
;
; Description:  This macro contains instructions that compute the trace
;               of the 4x4 double-precision floating-point matrix in ymm3:ymm0.

_Max4x4TraceF64 macro
        vblendpd ymm0,ymm0,ymm1,00000010b       ;ymm0[127:0] = row 1,0 diag vals
        vblendpd ymm1,ymm2,ymm3,00001000b       ;ymm1[255:128] = row 3,2 diag vals
        vperm2f128 ymm2,ymm1,ymm1,00000001b     ;ymm2[127:0] = row 3,2 diag vals
        vaddpd ymm3,ymm0,ymm2
        vhaddpd ymm0,ymm3,ymm3                  ;xmm0[63:0] = trace
        endm

; extern "C" double Avx2Mat4x4TraceF64_(const double* m_src1)
;
; Description:  The following function computes the trace of a
;               4x4 double-precision floating-point array.

Avx2Mat4x4TraceF64_ proc
            vmovapd ymm0,[rcx]              ;ymm0 = m_src1.row_0
            vmovapd ymm1,[rcx+32]           ;ymm1 = m_src1.row_1
            vmovapd ymm2,[rcx+64]           ;ymm2 = m_src1.row_2
            vmovapd ymm3,[rcx+96]           ;ymm3 = m_src1.row_3

            _Max4x4TraceF64                 ;xmm0[63:0] = m_src1.trace()
            vzeroupper
            ret
Avx2Mat4x4TraceF64_ endp
```

```
; _Mat4x4MulCalcRowF64 macro
;
; Description:  This macro is used to compute one row of a 4x4 matrix
;               multiply.
;
; Registers:    ymm0 = m_src2.row0
;               ymm1 = m_src2.row1
;               ymm2 = m_src2.row2
;               ymm3 = m_src2.row3
;               ymm4 - ymm7 = scratch registers

_Mat4x4MulCalcRowF64 macro dreg,sreg,disp
        vbroadcastsd ymm4,real8 ptr [sreg+disp]     ;broadcast m_src1[i][0]
        vbroadcastsd ymm5,real8 ptr [sreg+disp+8]   ;broadcast m_src1[i][1]
        vbroadcastsd ymm6,real8 ptr [sreg+disp+16]  ;broadcast m_src1[i][2]
        vbroadcastsd ymm7,real8 ptr [sreg+disp+24]  ;broadcast m_src1[i][3]

        vmulpd ymm4,ymm4,ymm0                       ;m_src1[i][0] * m_src2.row_0
        vmulpd ymm5,ymm5,ymm1                       ;m_src1[i][1] * m_src2.row_1
        vmulpd ymm6,ymm6,ymm2                       ;m_src1[i][2] * m_src2.row_2
        vmulpd ymm7,ymm7,ymm3                       ;m_src1[i][3] * m_src2.row_3

        vaddpd ymm4,ymm4,ymm5                       ;calc m_des.row_i
        vaddpd ymm6,ymm6,ymm7
        vaddpd ymm4,ymm4,ymm6
        vmovapd[dreg+disp],ymm4                     ;save m_des.row_i
        endm

; extern "C" void Avx2Mat4x4MulF64_(double* m_des, const double* m_src1, const double* m_
src2)

Avx2Mat4x4MulF64_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

        vmovapd ymm0,[r8]                   ;ymm0 = m_src2.row_0
        vmovapd ymm1,[r8+32]                ;ymm1 = m_src2.row_1
        vmovapd ymm2,[r8+64]                ;ymm2 = m_src2.row_2
        vmovapd ymm3,[r8+96]                ;ymm3 = m_src2.row_3

        _Mat4x4MulCalcRowF64 rcx,rdx,0      ;calculate m_des.row_0
        _Mat4x4MulCalcRowF64 rcx,rdx,32     ;calculate m_des.row_1
        _Mat4x4MulCalcRowF64 rcx,rdx,64     ;calculate m_des.row_2
        _Mat4x4MulCalcRowF64 rcx,rdx,96     ;calculate m_des.row_3

        vzeroupper
        _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
Avx2Mat4x4MulF64_ endp
```

```
; extern "C" bool Avx2Mat4x4InvF64_(double* m_inv, const double* m, double epsilon, bool*
is_singular);

; Offsets of intermediate matrices on stack relative to rsp
OffsetM2 equ 32
OffsetM3 equ 160
OffsetM4 equ 288

Avx2Mat4x4InvF64_ proc frame
        _CreateFrame MI_,0,160
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Save args to home area for later use
        mov qword ptr [rbp+MI_OffsetHomeRCX],rcx        ;save m_inv ptr
        mov qword ptr [rbp+MI_OffsetHomeRDX],rdx        ;save m ptr
        vmovsd real8 ptr [rbp+MI_OffsetHomeR8],xmm2     ;save epsilon
        mov qword ptr [rbp+MI_OffsetHomeR9],r9          ;save is_singular ptr

; Allocate 384 bytes of stack space for temp matrices + 32 bytes for function calls
        and rsp,0ffffffe0h                  ;align rsp to 32-byte boundary
        sub rsp,416                         ;alloc stack space

; Calculate m2
        lea rcx,[rsp+OffsetM2]              ;rcx = m2 ptr
        mov r8,rdx                          ;rdx, r8 = m ptr
        call Avx2Mat4x4MulF64_              ;calculate and save m2

; Calculate m3
        lea rcx,[rsp+OffsetM3]              ;rcx = m3 ptr
        lea rdx,[rsp+OffsetM2]              ;rdx = m2 ptr
        mov r8,[rbp+MI_OffsetHomeRDX]       ;r8 = m
        call Avx2Mat4x4MulF64_              ;calculate and save m3

; Calculate m4
        lea rcx,[rsp+OffsetM4]              ;rcx = m4 ptr
        lea rdx,[rsp+OffsetM3]              ;rdx = m3 ptr
        mov r8,[rbp+MI_OffsetHomeRDX]       ;r8 = m
        call Avx2Mat4x4MulF64_              ;calculate and save m4

; Calculate trace of m, m2, m3, and m4
        mov rcx,[rbp+MI_OffsetHomeRDX]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm8,xmm8,xmm0               ;xmm8 = t1

        lea rcx,[rsp+OffsetM2]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm9,xmm9,xmm0               ;xmm9 = t2
```

```
        lea rcx,[rsp+OffsetM3]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm10,xmm10,xmm0             ;xmm10 = t3

        lea rcx,[rsp+OffsetM4]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm11,xmm11,xmm0             ;xmm10 = t4

; Calculate the required coefficients
; c1 = -t1;
; c2 = -1.0f / 2.0f * (c1 * t1 + t2);
; c3 = -1.0f / 3.0f * (c2 * t1 + c1 * t2 + t3);
; c4 = -1.0f / 4.0f * (c3 * t1 + c2 * t2 + c1 * t3 + t4);
;
; Registers used:
;   t1-t4 = xmm8-xmm11
;   c1-c4 = xmm12-xmm15

        vxorpd xmm12,xmm8,real8 ptr [r8_SignBitMask]    ;xmm12 = c1

        vmulsd xmm13,xmm12,xmm8         ;c1 * t1
        vaddsd xmm13,xmm13,xmm9         ;c1 * t1 + t2
        vmulsd xmm13,xmm13,[r8_NOp5]    ;c2

        vmulsd xmm14,xmm13,xmm8         ;c2 * t1
        vmulsd xmm0,xmm12,xmm9          ;c1 * t2
        vaddsd xmm14,xmm14,xmm0         ;c2 * t1 + c1 * t2
        vaddsd xmm14,xmm14,xmm10        ;c2 * t1 + c1 * t2 + t3
        vmulsd xmm14,xmm14,[r8_NOp3333] ;c3

        vmulsd xmm15,xmm14,xmm8         ;c3 * t1
        vmulsd xmm0,xmm13,xmm9          ;c2 * t2
        vmulsd xmm1,xmm12,xmm10         ;c1 * t3
        vaddsd xmm2,xmm0,xmm1           ;c2 * t2 + c1 * t3
        vaddsd xmm15,xmm15,xmm2         ;c3 * t1 + c2 * t2 + c1 * t3
        vaddsd xmm15,xmm15,xmm11        ;c3 * t1 + c2 * t2 + c1 * t3 + t4
        vmulsd xmm15,xmm15,[r8_NOp25]   ;c4

; Make sure matrix is not singular
        vandpd xmm0,xmm15,[r8_AbsMask]                  ;compute fabs(c4)
        vmovsd xmm1,real8 ptr [rbp+MI_OffsetHomeR8]
        vcomisd xmm0,real8 ptr [rbp+MI_OffsetHomeR8]    ;compare against epsilon
        setp al                                         ;set al = if unordered
        setb ah                                         ;set ah = if fabs(c4) < epsilon
        or al,ah                                        ;al = is_singular
        mov rcx,[rbp+MI_OffsetHomeR9]                   ;rax = is_singular ptr
        mov [rcx],al                                    ;save is_singular state
        jnz Error                                       ;jump if singular
```

```
; Calculate m_inv = -1.0 / c4 * (m3 + c1 * m2 + c2 * m1 + c3 * I)
        vbroadcastsd ymm14,xmm14                        ;ymm14 = packed c3
        lea rcx,[Mat4x4I]                               ;rcx = I ptr
        vmulpd ymm0,ymm14,ymmword ptr [rcx]
        vmulpd ymm1,ymm14,ymmword ptr [rcx+32]
        vmulpd ymm2,ymm14,ymmword ptr [rcx+64]
        vmulpd ymm3,ymm14,ymmword ptr [rcx+96]          ;c3 * I

        vbroadcastsd ymm13,xmm13                        ;ymm13 = packed c2
        mov rcx,[rbp+MI_OffsetHomeRDX]                  ;rcx = m ptr
        vmulpd ymm4,ymm13,ymmword ptr [rcx]
        vmulpd ymm5,ymm13,ymmword ptr [rcx+32]
        vmulpd ymm6,ymm13,ymmword ptr [rcx+64]
        vmulpd ymm7,ymm13,ymmword ptr [rcx+96]          ;c2 * m1
        vaddpd ymm0,ymm0,ymm4
        vaddpd ymm1,ymm1,ymm5
        vaddpd ymm2,ymm2,ymm6
        vaddpd ymm3,ymm3,ymm7                           ;c2 * m1 + c3 * I

        vbroadcastsd ymm12,xmm12                        ;ymm12 = packed c1
        lea rcx,[rsp+OffsetM2]                          ;rcx = m2 ptr
        vmulpd ymm4,ymm12,ymmword ptr [rcx]
        vmulpd ymm5,ymm12,ymmword ptr [rcx+32]
        vmulpd ymm6,ymm12,ymmword ptr [rcx+64]
        vmulpd ymm7,ymm12,ymmword ptr [rcx+96]          ;c1 * m2
        vaddpd ymm0,ymm0,ymm4
        vaddpd ymm1,ymm1,ymm5
        vaddpd ymm2,ymm2,ymm6
        vaddpd ymm3,ymm3,ymm7                           ;c1 * m2 + c2 * m1 + c3 * I

        lea rcx,[rsp+OffsetM3]                          ;rcx = m3 ptr
        vaddpd ymm0,ymm0,ymmword ptr [rcx]
        vaddpd ymm1,ymm1,ymmword ptr [rcx+32]
        vaddpd ymm2,ymm2,ymmword ptr [rcx+64]
        vaddpd ymm3,ymm3,ymmword ptr [rcx+96]           ;m3 + c1 * m2 + c2 * m1 + c3 * I

        vmovsd xmm4,[r8_N1p0]
        vdivsd xmm4,xmm4,xmm15              ;xmm4 = -1.0 / c4
        vbroadcastsd ymm4,xmm4
        vmulpd ymm0,ymm0,ymm4
        vmulpd ymm1,ymm1,ymm4
        vmulpd ymm2,ymm2,ymm4
        vmulpd ymm3,ymm3,ymm4               ;ymm3:ymm0 = m_inv

; Save m_inv
        mov rcx,[rbp+MI_OffsetHomeRCX]
        vmovapd ymmword ptr [rcx],ymm0
        vmovapd ymmword ptr [rcx+32],ymm1
        vmovapd ymmword ptr [rcx+64],ymm2
        vmovapd ymmword ptr [rcx+96],ymm3
        mov eax,1                           ;set success return code
```

```
Done:    vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret

Error:  xor eax,eax
        jmp Done

Avx2Mat4x4InvF64_ endp
        end
```

The multiplicative inverse of a matrix is defined as follows: Let **A** and **X** represent $n \times n$ matrices. Matrix **X** is an inverse of **A** if **AX** = **XA** = **I**, where **I** denotes an $n \times n$ identity matrix (i.e., a matrix of all zeros except for the diagonal elements, which are equal to one). Figure 9-5 shows an example of an inverse matrix. It is important to note that inverses do not exist for all $n \times n$ matrices. A matrix without an inverse is called a *singular matrix*.

$$\mathbf{A} = \begin{bmatrix} 6 & 2 & 2 \\ 2 & -2 & 2 \\ 0 & 4 & 2 \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} 0.1875 & -0.0625 & -0.125 \\ 0.0625 & -0.1875 & 0.125 \\ -0.125 & 0.375 & 0.25 \end{bmatrix} \quad \mathbf{AX} = \mathbf{XA} = \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

***Figure 9-5.*** *Matrix A and its multiplicative inverse Matrix X*

The inverse of a 4 × 4 matrix can be calculated using a variety of mathematical techniques. Source code example Ch09_06 uses a computational method based on the Cayley-Hamilton theorem, which employs common matrix operations that are relatively easy to carry out using SIMD arithmetic. Here are the required equations:

$$\mathbf{A}^1 = \mathbf{A}; \mathbf{A}^2 = \mathbf{AA}; \mathbf{A}^3 = \mathbf{AAA}; \mathbf{A}^4 = \mathbf{AAAA}$$

$$trace(\mathbf{A}) = \sum_i a_{ii}$$

$$t_n = trace(\mathbf{A}^n)$$

$$c_1 = -t_1$$

$$c_2 = -\frac{1}{2}(c_1 t_1 + t_2)$$

$$c_3 = -\frac{1}{3}(c_2 t_1 + c_1 t_2 + t_3)$$

$$c_4 = -\frac{1}{4}(c_3 t_1 + c_2 t_2 + c_1 t_3 + t_4)$$

$$\mathbf{A}^{-1} = -\frac{1}{c_4}(\mathbf{A}^3 + c_1 \mathbf{A}^2 + c_2 \mathbf{A}^3 + c_3 \mathbf{I})$$

Toward the top of the C++ code is a function named `Avx2Mat4x4InvF64Cpp`. This function calculates the inverse of a 4 × 4 matrix of double-precision floating-point values using the aforementioned equations. Function `Avx2Mat4x4InvF64Cpp` uses the C++ class `Matrix<>` to perform many of the required intermediate computations, including matrix addition, multiplication, and trace. The source code for class `Matrix<>` is not

shown but included with the chapter download package. Note that the intermediate matrices are declared using the `static` qualifier in order to avoid constructor overhead when performing benchmark timing measurements. The drawback of using the `static` qualifier here means that the function is not thread-safe (a thread-safe function can be simultaneously used by multiple threads). Following calculation of the trace values `t1 - t4`, `Avx2Mat4x4InvF64Cpp` computes `c1`–`c4` using simple scalar arithmetic. It then checks to make sure the source matrix `m` is not singular by comparing `c4` against `epsilon`. If matrix `m` is not singular, the final inverse is calculated. The remaining C++ code performs test case initialization and exercises both the C++ and assembly language matrix inversion functions.

The assembly language code in Listing 9-6 begins with a custom segment that contains definitions of the constant values needed by the assembly language matrix inversion functions. The statement `ConstVals segment readonly align(32) 'const'` marks the start of a segment that begins on a 32-byte boundary and contains read-only data. The reason for using a custom segment here is that the MASM `align` directive does not support aligning data items on a 32-byte boundary. In this example, proper alignment of the packed constants is essential in order to maximize performance. Note that the scalar double-precision floating-point constants are defined after the 256-bit wide packed constants and are aligned on an 8-byte boundary. The MASM statement `ConstVals ends` terminates the custom segment.

Following the custom constant segment is the macro `_Max4x4TraceF64`. This macro contains instructions that calculate the trace of a 4 × 4 matrix of double-precision floating-point values. Macro `_Max4x4TraceF64` requires the four rows of the source matrix to be loaded in registers YMM0–YMM3 and uses the `vblendpd`, `vperm2f128`, and `vhaddpd` instructions to calculate the matrix trace, as shown in Figure 9-6. The `vblendpd` (Blend Packed Double-Precision Floating-Point Values) instruction merges values from its two source operands according to an immediate control mask. If bit 0 of the control mask equals 0, element 0 (i.e., bits 63:0) from the first source operand is copied to the corresponding element position in the destination operand; otherwise, element 0 from the second source operand is copied to the destination operand. Bits 1–3 of the control mask are used in a similar manner for the other three elements. Register XMM0[63:0] contains the trace value following execution of the `vhaddpd` instruction.

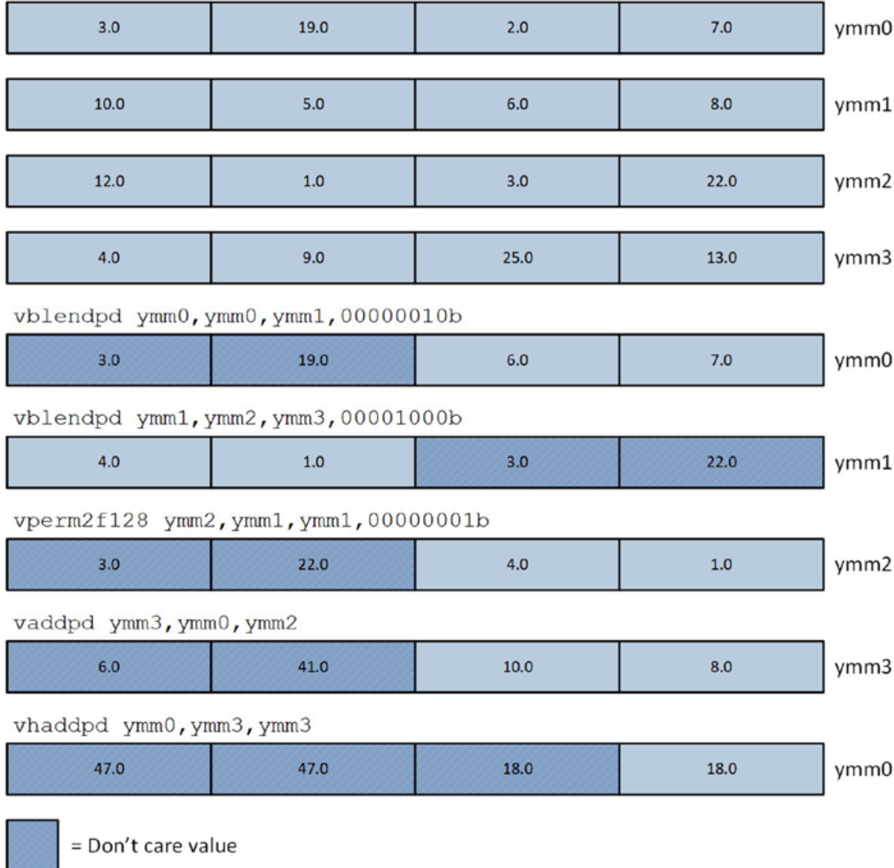



***Figure 9-6.*** *Trace calculation for a 4 × 4 matrix*

The assembly language function `Avx2Mat4x4InvF64_` calculates an inverse matrix using the same technique as the corresponding C++ function. Following its prolog, the function `Avx2Mat4x4InvF64_` saves its argument values to the home area for later use. It then allocates storage space on the stack to hold intermediate results. More specifically, the `and rsp,0ffffffe0h` instruction aligns RSP to a 32-byte boundary, and the `sub rsp,416` instruction allocates local stack space that's required for the intermediate matrices `m2`, `m3`, and `m4` plus 32 bytes for function calls. Next, a series of calls are made to the functions `Avx2Mat4x4MulF64_` and `Avx2Mat4x4TraceF64_` to calculate the trace values `t1`–`t4`. The matrix multiplication code that's used in this example is basically the same code that you saw in example `Ch09_05`. The algorithm coefficients `c1`–`c4` are calculated next using simple scalar floating-point arithmetic. Coefficient `c4` is then

tested to verify that the source matrix is not singular. If the source matrix is not singular, the function calculates the inverse matrix `m_inv`. Note that all of the arithmetic required to calculate `m_inv` is carried out using straightforward packed double-precision floating-point multiplication and addition. Here is the output for source code example `Ch09_06`:

```
Results for Avx2Mat4x4InvF64

Test #1 - Test Matrix
         2           7           3           4
         5           9           6        4.75
       6.5           3           4          10
         7        5.25       8.125           6

Test #1 - Avx2Mat4x4InvF64Cpp - Inverse Matrix
 -0.943926     0.91657    0.197547   -0.425579
-0.0568818    0.251148  0.00302831   -0.165952
  0.545399   -0.647656   -0.213597    0.505123
  0.412456   -0.412053   0.0561248    0.124363

Test #1 - Avx2Mat4x4InvF64Cpp - Verify Matrix
         1           0           0           0
         0           1           0           0
         0           0           1           0
         0           0           0           1

Test #1 - Avx2Mat4x4InvF64_ - Inverse Matrix
 -0.943926     0.91657    0.197547   -0.425579
-0.0568818    0.251148  0.00302831   -0.165952
  0.545399   -0.647656   -0.213597    0.505123
  0.412456   -0.412053   0.0561248    0.124363

Test #1 - Avx2Mat4x4InvF64_ - Verify Matrix
         1           0           0           0
         0           1           0           0
         0           0           1           0
         0           0           0           1

Test #2 - Test Matrix
       0.5          12       17.25           4
         5           2        6.75           8
    13.125           1           3        9.75
        16       1.625           7        0.25

Test #2 - Avx2Mat4x4InvF64Cpp - Inverse Matrix
0.00165165  -0.0690239   0.0549591   0.0389347
  0.135369   -0.359846    0.242038  -0.0903252
-0.0350097    0.239298   -0.183964   0.0772214
-0.0053352    0.056194   0.0603606  -0.0669085
```

```
Test #2 - Avx2Mat4x4InvF64Cpp - Verify Matrix
         1           0           0           0
         0           1           0           0
         0           0           1           0
         0           0           0           1

Test #2 - Avx2Mat4x4InvF64_ - Inverse Matrix
0.00165165  -0.0690239   0.0549591   0.0389347
  0.135369   -0.359846    0.242038  -0.0903252
-0.0350097    0.239298   -0.183964   0.0772214
-0.0053352    0.056194   0.0603606  -0.0669085

Test #2 - Avx2Mat4x4InvF64_ - Verify Matrix
         1           0           0           0
         0           1           0           0
         0           0           1           0
         0           0           0           1

Test #3 - Test Matrix
         2           0           0           1
         0           4           5           0
         0           0           0           7
         0           0           0           6

Test #3 - Avx2Mat4x4InvF64Cpp - Singular Matrix
Test #3 - Avx2Mat4x4InvF64_ - Singular Matrix

Running benchmark function Avx2Mat4x4InvF64_BM - please wait
Benchmark times save to file Ch09_06_Avx2Mat4x4InvF64_BM_CHROMIUM.csv
```

Table 9-4 contains benchmark timing measurements for the matrix inversion functions.

***Table 9-4.*** *Matrix Inverse Mean Execution Times (Microseconds), 100,000 Inversions*

| CPU | C++ | Assembly Language |
|---|---|---|
| i7-4790S | 30417 | 4168 |
| i9-7900X | 26646 | 3773 |
| i7-8700K | 24485 | 2941 |

# Blend and Permute Instructions

A data blend operation conditionally copies elements from two packed source operands to a packed destination operand using a control mask that specifies which elements to copy. A data permute operation rearranges the elements of a packed source operand according to a control mask. You've already seen several source code examples in this chapter that exploited data blend and permute operations. The next source code example is named Ch09_07 and includes code that demonstrates how to use additional blend and permute instructions. Listing 9-7 shows the source code for example Ch09_07.

***Listing 9-7.*** Example Ch09_07

```
//------------------------------------------------
//               Ch09_07.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include "YmmVal.h"

using namespace std;

extern "C" void AvxBlendF32_(YmmVal* des1, YmmVal* src1, YmmVal* src2, YmmVal* idx1);
extern "C" void Avx2PermuteF32_(YmmVal* des1, YmmVal* src1, YmmVal* idx1, YmmVal* des2,
YmmVal* src2, YmmVal* idx2);

void AvxBlendF32(void)
{
    const uint32_t sel0 = 0x00000000;
    const uint32_t sel1 = 0x80000000;
    alignas(32) YmmVal des1, src1, src2, idx1;

    src1.m_F32[0] = 10.0f;  src2.m_F32[0] = 100.0f;  idx1.m_I32[0] = sel1;
    src1.m_F32[1] = 20.0f;  src2.m_F32[1] = 200.0f;  idx1.m_I32[1] = sel0;
    src1.m_F32[2] = 30.0f;  src2.m_F32[2] = 300.0f;  idx1.m_I32[2] = sel0;
    src1.m_F32[3] = 40.0f;  src2.m_F32[3] = 400.0f;  idx1.m_I32[3] = sel1;
    src1.m_F32[4] = 50.0f;  src2.m_F32[4] = 500.0f;  idx1.m_I32[4] = sel1;
    src1.m_F32[5] = 60.0f;  src2.m_F32[5] = 600.0f;  idx1.m_I32[5] = sel0;
    src1.m_F32[6] = 70.0f;  src2.m_F32[6] = 700.0f;  idx1.m_I32[6] = sel1;
    src1.m_F32[7] = 80.0f;  src2.m_F32[7] = 800.0f;  idx1.m_I32[7] = sel0;

    AvxBlendF32_(&des1, &src1, &src2, &idx1);

    cout << "\nResults for AvxBlendF32 (vblendvps)\n";
    cout << fixed << setprecision(1);

    for (size_t i = 0; i < 8; i++)
    {
        cout << "i: " << setw(2) << i << "  ";
        cout << "src1: " << setw(8) << src1.m_F32[i] << "  ";
        cout << "src2: " << setw(8) << src2.m_F32[i] << "  ";
        cout << setfill('0');
        cout << "idx1: 0x" << setw(8) << hex << idx1.m_U32[i] << "  ";
        cout << setfill(' ');
        cout << "des1: " << setw(8) << des1.m_F32[i] << '\n';
    }
}
```

```
void Avx2PermuteF32(void)
{
    alignas(32) YmmVal des1, src1, idx1;
    alignas(32) YmmVal des2, src2, idx2;

    // idx1 values must be between 0 and 7.
    src1.m_F32[0] = 100.0f;      idx1.m_I32[0] = 3;
    src1.m_F32[1] = 200.0f;      idx1.m_I32[1] = 7;
    src1.m_F32[2] = 300.0f;      idx1.m_I32[2] = 0;
    src1.m_F32[3] = 400.0f;      idx1.m_I32[3] = 4;
    src1.m_F32[4] = 500.0f;      idx1.m_I32[4] = 6;
    src1.m_F32[5] = 600.0f;      idx1.m_I32[5] = 6;
    src1.m_F32[6] = 700.0f;      idx1.m_I32[6] = 1;
    src1.m_F32[7] = 800.0f;      idx1.m_I32[7] = 2;

    // idx2 values must be between 0 and 3.
    src2.m_F32[0] = 100.0f;      idx2.m_I32[0] = 3;
    src2.m_F32[1] = 200.0f;      idx2.m_I32[1] = 1;
    src2.m_F32[2] = 300.0f;      idx2.m_I32[2] = 1;
    src2.m_F32[3] = 400.0f;      idx2.m_I32[3] = 2;
    src2.m_F32[4] = 500.0f;      idx2.m_I32[4] = 3;
    src2.m_F32[5] = 600.0f;      idx2.m_I32[5] = 2;
    src2.m_F32[6] = 700.0f;      idx2.m_I32[6] = 0;
    src2.m_F32[7] = 800.0f;      idx2.m_I32[7] = 0;

    Avx2PermuteF32_(&des1, &src1, &idx1, &des2, &src2, &idx2);

    cout << "\nResults for Avx2PermuteF32 (vpermps)\n";
    cout << fixed << setprecision(1);

    for (size_t i = 0; i < 8; i++)
    {
        cout << "i: " << setw(2) << i << "  ";
        cout << "src1: " << setw(8) << src1.m_F32[i] << "  ";
        cout << "idx1: " << setw(8) << idx1.m_I32[i] << "  ";
        cout << "des1: " << setw(8) << des1.m_F32[i] << '\n';
    }

    cout << "\nResults for Avx2PermuteF32 (vpermilps)\n";

    for (size_t i = 0; i < 8; i++)
    {
        cout << "i: " << setw(2) << i << "  ";
        cout << "src2: " << setw(8) << src2.m_F32[i] << "  ";
        cout << "idx2: " << setw(8) << idx2.m_I32[i] << "  ";
        cout << "des2: " << setw(8) << des2.m_F32[i] << '\n';
    }
}
```

```
int main()
{
    AvxBlendF32();
    Avx2PermuteF32();
    return 0;
}

;-------------------------------------------------
;               Ch09_07.asm
;-------------------------------------------------

; extern "C" void AvxBlendF32_(YmmVal* des1, YmmVal* src1, YmmVal* src2, YmmVal* idx1)

        .code
AvxBlendF32_ proc
        vmovaps ymm0,ymmword ptr [rdx]  ;ymm0 = src1
        vmovaps ymm1,ymmword ptr [r8]   ;ymm1 = src2
        vmovdqa ymm2,ymmword ptr [r9]   ;ymm2 = idx1
        vblendvps ymm3,ymm0,ymm1,ymm2   ;blend ymm0 & ymm1, ymm2 "indices"
        vmovaps ymmword ptr [rcx],ymm3  ;Save result to des1

        vzeroupper
        ret
AvxBlendF32_ endp

; extern "C" void Avx2PermuteF32_(YmmVal* des1, YmmVal* src1, YmmVal* idx1, YmmVal* des2,
YmmVal* src2, YmmVal* idx2)

Avx2PermuteF32_ proc

; Perform vpermps permutation
        vmovaps ymm0,ymmword ptr [rdx]      ;ymm0 = src1
        vmovdqa ymm1,ymmword ptr [r8]       ;ymm1 = idx1
        vpermps ymm2,ymm1,ymm0              ;permute ymm0 using ymm1 indices
        vmovaps ymmword ptr [rcx],ymm2      ;save result to des1

; Perform vpermilps permutation
        mov rdx,[rsp+40]                    ;rdx = src2 ptr
        mov r8,[rsp+48]                     ;r8 = idx2 ptr
        vmovaps ymm3,ymmword ptr [rdx]      ;ymm3 = src2
        vmovdqa ymm4,ymmword ptr [r8]       ;ymm4 = idx1
        vpermilps ymm5,ymm3,ymm4            ;permute ymm3 using ymm4 indices
        vmovaps ymmword ptr [r9],ymm5       ;save result to des2

        vzeroupper
        ret
Avx2PermuteF32_ endp
        end
```

The C++ code in Listing 9-7 begins with a function named AvxBlendF32 that initializes YmmVal variables src1 and src2 using single-precision floating-point values. It also initializes a third YmmVal variable named src3 for use as a blend control mask. The high-order bit of each doubleword element in src3 specifies whether the corresponding element from src1 (high-order bit = 0) or src2 (high-order bit = 1) is copied to the destination operand. These three source operands are used by the vblendvps (Variable Blend Packed Single- Precision Floating-Point Values) instruction, which is located in the assembly language function AvxBlendF32_. Following execution of this function, the results are streamed to cout.

The C++ code in Listing 9-7 also includes a function named Avx2PermuteF32. This function initializes several YmmVal variables that demonstrate use of the vpermps and vpermips instructions. Both of these instructions require a set of indices that specify which source operand elements are copied to the destination operand. For example, the statement idx1.m_I32[0] = 3 is used to direct the vpermps instruction in Avx2PermuteF32_ to perform des1.m_F32[0] = src1.m_F32[3]. The vpermps instruction requires each index in idx1 to be between zero and seven. An index can be used more than once in idx1 in order to copy an element from src1 to multiple locations in des1. The vpermilps instruction requires its indices to be between zero and three.

The assembly language function AvxBlendF32_ begins by loading the source data operands into registers YMM0 and YMM1 using two vmovaps instructions. The vmovdqa instruction that follows loads the blend control mask into register YMM2. The ensuing vblendvps ymm3,ymm0,ymm1,ymm2 instruction blends elements from registers YMM0 and YMM1 into YMM3 according to the control values in YMM2. The high-order bit of each doubleword element in YMM2 specifies whether the corresponding element from YMM0 (high-order bit = 0) or YMM1 (high-order bit = 1) is copied to YMM3. Figure 9-7 illustrates the execution of this instruction in greater detail. The vblendvps instruction and its double-precision counterpart vblendvpd are examples of AVX instructions that require three source operands. Floating-point blend operations using an immediate control mask are also possible with the vblendp[d|s] instructions.

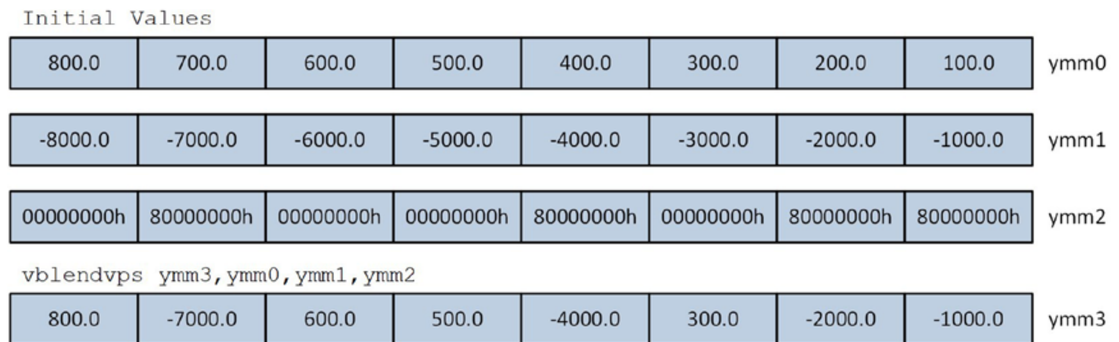



***Figure 9-7.*** *Execution of the vblendvps instruction*

Following AvxBlendF32_ in Listing 9-7 is the function Avx2PermuteF32_, which demonstrates use of the vpermps and vpermilps instructions. The vpermps instruction permutes (or rearranges) the elements of its first source operand (which is 256 bits wide and contains eight single-precision floating-point values) according to the indices in the second source operand. The vpermilps (In-Lane Permute of Single-Precision Floating-Point Values) instruction performs its permutations using two independent 128-bit wide lanes (i.e., bits [255:128] and bits [127:0]). The control indices for an in-lane permutation must range between zero and three, and each lane uses its own distinct set of indices. Figure 9-8 illustrates the execution of these instructions in greater detail. AVX and AVX2 also include the double-precision floating-point permute instructions vpermilpd and vpermpd.

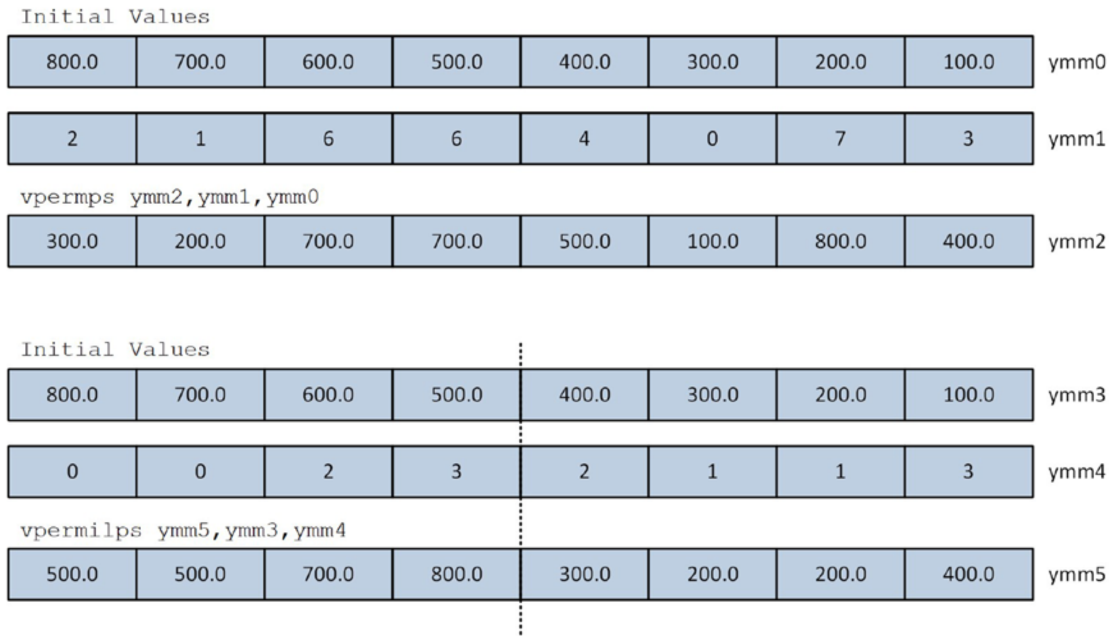



***Figure 9-8.*** *Execution of the* `vpermps` *and* `vpermilps` *instructions*

Here is the output for source code example `Ch09_07`:

```
Results for AvxBlendF32 (vblendvps)
i:  0  src1:     10.0  src2:    100.0  idx1: 0x80000000  des1:    100.0
i:  1  src1:     20.0  src2:    200.0  idx1: 0x00000000  des1:     20.0
i:  2  src1:     30.0  src2:    300.0  idx1: 0x00000000  des1:     30.0
i:  3  src1:     40.0  src2:    400.0  idx1: 0x80000000  des1:    400.0
i:  4  src1:     50.0  src2:    500.0  idx1: 0x80000000  des1:    500.0
i:  5  src1:     60.0  src2:    600.0  idx1: 0x00000000  des1:     60.0
i:  6  src1:     70.0  src2:    700.0  idx1: 0x80000000  des1:    700.0
i:  7  src1:     80.0  src2:    800.0  idx1: 0x00000000  des1:     80.0

Results for Avx2PermuteF32 (vpermps)
i:  0  src1:    100.0  idx1:        3  des1:    400.0
i:  1  src1:    200.0  idx1:        7  des1:    800.0
i:  2  src1:    300.0  idx1:        0  des1:    100.0
i:  3  src1:    400.0  idx1:        4  des1:    500.0
i:  4  src1:    500.0  idx1:        6  des1:    700.0
i:  5  src1:    600.0  idx1:        6  des1:    700.0
i:  6  src1:    700.0  idx1:        1  des1:    200.0
i:  7  src1:    800.0  idx1:        2  des1:    300.0

Results for Avx2PermuteF32 (vpermilps)
i:  0  src2:    100.0  idx2:        3  des2:    400.0
i:  1  src2:    200.0  idx2:        1  des2:    200.0
i:  2  src2:    300.0  idx2:        1  des2:    200.0
```

```
i:  3  src2:    400.0  idx2:        2  des2:    300.0
i:  4  src2:    500.0  idx2:        3  des2:    800.0
i:  5  src2:    600.0  idx2:        2  des2:    700.0
i:  6  src2:    700.0  idx2:        0  des2:    500.0
i:  7  src2:    800.0  idx2:        0  des2:    500.0
```

## Data Gather Instructions

The final source code example of this chapter, Ch09_08, explains how to use the AVX2 gather instructions. A gather instruction conditionally loads elements from non-contiguous memory locations (typically an array) into an XMM or YMM register. A gather instruction requires a set of indices and a merge control mask that specifies which elements to copy. Listing 9-8 shows the source code for example Ch09_08. Chapter 8 presented an overview of the AVX2 gather instructions, including a graphic (see Figure 8-1) that elucidated execution of the vgatherdps instruction. You may find it helpful to review that material prior to perusing the source code and discussions in this section.

***Listing 9-8.*** Example Ch09_08

```
//------------------------------------------------
//               Ch09_08.cpp
//------------------------------------------------

#include "stdafx.h"
#include <string>
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <array>
#include <stdexcept>

using namespace std;

extern "C" void Avx2Gather8xF32_I32_(float* y, const float* x,
    const int32_t* indices, const int32_t* masks);
extern "C" void Avx2Gather8xF32_I64_(float* y, const float* x,
    const int64_t* indices, const int32_t* masks);
extern "C" void Avx2Gather8xF64_I32_(double* y, const double* x,
    const int32_t* indices, const int64_t* masks);
extern "C" void Avx2Gather8xF64_I64_(double* y, const double* x,
    const int64_t* indices, const int64_t* masks);

template <typename T, typename I, typename M, size_t N>
    void Print(const string& msg, const array<T, N>& y, const array<I, N>& indices,
    const array<M, N>& merge)
{
    if (y.size() != indices.size() || y.size() != merge.size())
        throw runtime_error("Non-conforming arrays - Print");

    cout << '\n' << msg << '\n';
```

```
    for (size_t i = 0; i < y.size(); i++)
    {
        string merge_s = (merge[i] == 1) ? "Yes" : "No";

        cout << "i: " << setw(2) << i << "   ";
        cout << "y: " << setw(10) << y[i] << "   ";
        cout << "index: " << setw(4) << indices[i] << "   ";
        cout << "merge: " << setw(4) << merge_s << '\n';
    }
}

void Avx2Gather8xF32_I32()
{
    array<float, 20> x;

    for (size_t i = 0; i < x.size(); i++)
        x[i] = (float)(i * 10);

    array<float, 8> y { -1, -1, -1, -1, -1, -1, -1, -1 };
    array<int32_t, 8> indices { 2, 1, 6, 5, 4, 13, 11, 9 };
    array<int32_t, 8> merge { 1, 1, 0, 1, 1, 0, 1, 1 };

    cout << fixed << setprecision(1);
    cout << "\nResults for Avx2Gather8xF32_I32\n";

    Print("Values before", y, indices, merge);
    Avx2Gather8xF32_I32_(y.data(), x.data(), indices.data(), merge.data());
    Print("Values after", y, indices, merge);
}

void Avx2Gather8xF32_I64()
{
    array<float, 20> x;

    for (size_t i = 0; i < x.size(); i++)
        x[i] = (float)(i * 10);

    array<float, 8> y { -1, -1, -1, -1, -1, -1, -1, -1 };
    array<int64_t, 8> indices { 19, 1, 0, 5, 4, 3, 11, 11 };
    array<int32_t, 8> merge { 1, 1, 1, 1, 0, 0, 1, 1 };

    cout << fixed << setprecision(1);
    cout << "\nResults for Avx2Gather8xF32_I64\n";

    Print("Values before", y, indices, merge);
    Avx2Gather8xF32_I64_(y.data(), x.data(), indices.data(), merge.data());
    Print("Values after", y, indices, merge);
}
```

```
void Avx2Gather8xF64_I32()
{
    array<double, 20> x;

    for (size_t i = 0; i < x.size(); i++)
        x[i] = (double)(i * 10);

    array<double, 8> y { -1, -1, -1, -1, -1, -1, -1, -1 };
    array<int32_t, 8> indices { 12, 11, 6, 15, 4, 13, 18, 3 };
    array<int64_t, 8> merge { 1, 1, 0, 1, 1, 0, 1, 0 };

    cout << fixed << setprecision(1);
    cout << "\nResults for Avx2Gather8xF64_I32\n";

    Print("Values before", y, indices, merge);
    Avx2Gather8xF64_I32_(y.data(), x.data(), indices.data(), merge.data());
    Print("Values after", y, indices, merge);
}

void Avx2Gather8xF64_I64()
{
    array<double, 20> x;

    for (size_t i = 0; i < x.size(); i++)
        x[i] = (double)(i * 10);

    array<double, 8> y { -1, -1, -1, -1, -1, -1, -1, -1 };
    array<int64_t, 8> indices { 11, 17, 1, 6, 14, 13, 8, 8 };
    array<int64_t, 8> merge { 1, 0, 1, 1, 1, 0, 1, 1 };

    cout << fixed << setprecision(1);
    cout << "\nResults for Avx2Gather8xF64_I64\n";

    Print("Values before", y, indices, merge);
    Avx2Gather8xF64_I64_(y.data(), x.data(), indices.data(), merge.data());
    Print("Values after", y, indices, merge);
}

int main()
{
    Avx2Gather8xF32_I32();
    Avx2Gather8xF32_I64();
    Avx2Gather8xF64_I32();
    Avx2Gather8xF64_I64();
    return 0;
}
```

```
;-------------------------------------------------
;               Ch09_08.asm
;-------------------------------------------------

; For each of the following functions, the contents of y are loaded
; into ymm0 prior to execution of the vgatherXXX instruction in order to
; demonstrate the effects of conditional merging.

        .code
; extern "C" void Avx2Gather8xF32_I32_(float* y, const float* x, const int32_t* indices,
const int32_t* merge)

Avx2Gather8xF32_I32_ proc
        vmovups ymm0,ymmword ptr [rcx]      ;ymm0 = y[7]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]       ;ymm1 = indices[7]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]       ;ymm2 = merge[7]:merge[0]
        vpslld ymm2,ymm2,31                 ;shift merge vals to high-order bits
        vgatherdps ymm0,[rdx+ymm1*4],ymm2   ;ymm0 = gathered elements
        vmovups ymmword ptr [rcx],ymm0      ;save gathered elements

        vzeroupper
        ret
Avx2Gather8xF32_I32_ endp

; extern "C" void Avx2Gather8xF32_I64_(float* y, const float* x, const int64_t* indices,
const int32_t* merge)

Avx2Gather8xF32_I64_ proc
        vmovups xmm0,xmmword ptr [rcx]      ;xmm0 = y[3]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]       ;ymm1 = indices[3]:indices[0]
        vmovdqu xmm2,xmmword ptr [r9]       ;xmm2 = merge[3]:merge[0]
        vpslld xmm2,xmm2,31                 ;shift merge vals to high-order bits
        vgatherqps xmm0,[rdx+ymm1*4],xmm2   ;xmm0 = gathered elements
        vmovups xmmword ptr [rcx],xmm0      ;save gathered elements

        vmovups xmm3,xmmword ptr [rcx+16]   ;xmm0 = des[7]:des[4]
        vmovdqu ymm1,ymmword ptr [r8+32]    ;ymm1 = indices[7]:indices[4]
        vmovdqu xmm2,xmmword ptr [r9+16]    ;xmm2 = merge[7]:merge[4]
        vpslld xmm2,xmm2,31                 ;shift merge vals to high-order bits
        vgatherqps xmm3,[rdx+ymm1*4],xmm2   ;xmm0 = gathered elements
        vmovups xmmword ptr [rcx+16],xmm3   ;save gathered elements

        vzeroupper
        ret
Avx2Gather8xF32_I64_ endp

; extern "C" void Avx2Gather8xF64_I32_(double* y, const double* x, const int32_t* indices,
const int64_t* merge)
```

```
Avx2Gather8xF64_I32_ proc
        vmovupd ymm0,ymmword ptr [rcx]      ;ymm0 = y[3]:y[0]
        vmovdqu xmm1,xmmword ptr [r8]       ;xmm1 = indices[3]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]       ;ymm2 = merge[3]:merge[0]
        vpsllq ymm2,ymm2,63                 ;shift merge vals to high-order bits
        vgatherdpd ymm0,[rdx+xmm1*8],ymm2   ;ymm0 = gathered elements
        vmovupd ymmword ptr [rcx],ymm0      ;save gathered elements

        vmovupd ymm0,ymmword ptr [rcx+32]   ;ymm0 = y[7]:y[4]
        vmovdqu xmm1,xmmword ptr [r8+16]    ;xmm1 = indices[7]:indices[4]
        vmovdqu ymm2,ymmword ptr [r9+32]    ;ymm2 = merge[7]:merge[4]
        vpsllq ymm2,ymm2,63                 ;shift merge vals to high-order bits
        vgatherdpd ymm0,[rdx+xmm1*8],ymm2   ;ymm0 = gathered elements
        vmovupd ymmword ptr [rcx+32],ymm0   ;save gathered elements

        vzeroupper
        ret
Avx2Gather8xF64_I32_ endp

; extern "C" void Avx2Gather8xF64_I64_(double* y, const double* x, const int64_t* indices,
const int64_t* merge)

Avx2Gather8xF64_I64_ proc
        vmovupd ymm0,ymmword ptr [rcx]      ;ymm0 = y[3]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]       ;ymm1 = indices[3]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]       ;ymm2 = merge[3]:merge[0]
        vpsllq ymm2,ymm2,63                 ;shift merge vals to high-order bits
        vgatherqpd ymm0,[rdx+ymm1*8],ymm2   ;ymm0 = gathered elements
        vmovupd ymmword ptr [rcx],ymm0      ;save gathered elements

        vmovupd ymm0,ymmword ptr [rcx+32]   ;ymm0 = y[7]:y[4]
        vmovdqu ymm1,ymmword ptr [r8+32]    ;ymm1 = indices[7]:indices[4]
        vmovdqu ymm2,ymmword ptr [r9+32]    ;ymm2 = merge[7]:merge[4]
        vpsllq ymm2,ymm2,63                 ;shift merge vals to high-order bits
        vgatherqpd ymm0,[rdx+ymm1*8],ymm2   ;ymm0 = gathered elements
        vmovupd ymmword ptr [rcx+32],ymm0   ;save gathered elements

        vzeroupper
        ret
Avx2Gather8xF64_I64_ endp
        end
```

The C++ source code in example Ch09_08 includes four functions that initialize test cases to perform single-precision and double-precision floating-point gather operations using signed doubleword or quadword indices. The function Avx2Gather8xF32_I32 begins by initializing the elements of array x (the source array) with test values. Note that this function uses the STL class array<> instead of a raw C++ array to demonstrate use of the former with an assembly language function. Appendix A contains a list of C++ references that you can consult if you're interested in learning more about this class. Next, each element in array y (the destination array) is set to -1.0 in order to illustrate the effects of conditional merging. The arrays indices and merge are also primed with the required gather instruction indices and merge control mask values, respectively. The assembly language function Avx2Gather8xF32_I32_ is then called to carry out the

gather operation. Note that raw pointers for the various STL arrays are obtained using template function `array<>.data`. The other C++ functions in this source example—`Avx2Gather8xF32_I64`, `Avx2Gather8xF64_I32`, and `Avx2Gather8xF64_I64`—are similarly structured.

The assembly language function `Avx2Gather8xF32_I32_` begins by loading registers YMM0, YMM1, and YMM2 with the test arrays `y`, `indices`, and `merge`, respectively. Register RDX contains a pointer to the source array `x`. The `vpslld ymm2,ymm2,31` instruction shifts the merge control mask values (each value in this mask is zero or one) to the high-order bit of each doubleword element. The ensuing `vgatherdps ymm0,[rdx+ymm1*4],ymm2` instruction loads eight single-precision floating-point values from array `x` into register YMM0. The merge control mask in YMM2 dictates which array elements are actually copied into the destination operand YMM0. If the high-order bit of a merge control mask doubleword element is set to `1`, the corresponding element in YMM0 is updated; otherwise, it is not changed. Subsequent to the successful load of an array element, the `vgatherdps` instruction sets the corresponding doubleword element in the merge control mask to zero. The `vmovups ymmword ptr [rcx],ymm0` then saves the gather result to `y`.

The assembly language functions `Avx2Gather8xF32_I64_`, `Avx2Gather8xF64_I32_`, and `Avx2Gather8xF64_I64_` are analogous to `Avx2Gather8xF32_I32_`. Note that the gather instructions used in these functions—`vgatherqps`, `vgatherdpd`, and `vgatherqpd`—gather only four elements, which explains why they're used twice. Here are the results for source code example `Ch09_08`:

```
Results for Avx2Gather8xF32_I32

Values before
i:  0   y:       -1.0   index:    2   merge:  Yes
i:  1   y:       -1.0   index:    1   merge:  Yes
i:  2   y:       -1.0   index:    6   merge:   No
i:  3   y:       -1.0   index:    5   merge:  Yes
i:  4   y:       -1.0   index:    4   merge:  Yes
i:  5   y:       -1.0   index:   13   merge:   No
i:  6   y:       -1.0   index:   11   merge:  Yes
i:  7   y:       -1.0   index:    9   merge:  Yes

Values after
i:  0   y:       20.0   index:    2   merge:  Yes
i:  1   y:       10.0   index:    1   merge:  Yes
i:  2   y:       -1.0   index:    6   merge:   No
i:  3   y:       50.0   index:    5   merge:  Yes
i:  4   y:       40.0   index:    4   merge:  Yes
i:  5   y:       -1.0   index:   13   merge:   No
i:  6   y:      110.0   index:   11   merge:  Yes
i:  7   y:       90.0   index:    9   merge:  Yes

Results for Avx2Gather8xF32_I64

Values before
i:  0   y:       -1.0   index:   19   merge:  Yes
i:  1   y:       -1.0   index:    1   merge:  Yes
i:  2   y:       -1.0   index:    0   merge:  Yes
i:  3   y:       -1.0   index:    5   merge:  Yes
i:  4   y:       -1.0   index:    4   merge:   No
i:  5   y:       -1.0   index:    3   merge:   No
i:  6   y:       -1.0   index:   11   merge:  Yes
i:  7   y:       -1.0   index:   11   merge:  Yes
```

```
Values after
i:  0   y:      190.0   index:   19   merge:  Yes
i:  1   y:       10.0   index:    1   merge:  Yes
i:  2   y:        0.0   index:    0   merge:  Yes
i:  3   y:       50.0   index:    5   merge:  Yes
i:  4   y:       -1.0   index:    4   merge:   No
i:  5   y:       -1.0   index:    3   merge:   No
i:  6   y:      110.0   index:   11   merge:  Yes
i:  7   y:      110.0   index:   11   merge:  Yes

Results for Avx2Gather8xF64_I32

Values before
i:  0   y:       -1.0   index:   12   merge:  Yes
i:  1   y:       -1.0   index:   11   merge:  Yes
i:  2   y:       -1.0   index:    6   merge:   No
i:  3   y:       -1.0   index:   15   merge:  Yes
i:  4   y:       -1.0   index:    4   merge:  Yes
i:  5   y:       -1.0   index:   13   merge:   No
i:  6   y:       -1.0   index:   18   merge:  Yes
i:  7   y:       -1.0   index:    3   merge:   No

Values after
i:  0   y:      120.0   index:   12   merge:  Yes
i:  1   y:      110.0   index:   11   merge:  Yes
i:  2   y:       -1.0   index:    6   merge:   No
i:  3   y:      150.0   index:   15   merge:  Yes
i:  4   y:       40.0   index:    4   merge:  Yes
i:  5   y:       -1.0   index:   13   merge:   No
i:  6   y:      180.0   index:   18   merge:  Yes
i:  7   y:       -1.0   index:    3   merge:   No

Results for Avx2Gather8xF64_I64

Values before
i:  0   y:       -1.0   index:   11   merge:  Yes
i:  1   y:       -1.0   index:   17   merge:   No
i:  2   y:       -1.0   index:    1   merge:  Yes
i:  3   y:       -1.0   index:    6   merge:  Yes
i:  4   y:       -1.0   index:   14   merge:  Yes
i:  5   y:       -1.0   index:   13   merge:   No
i:  6   y:       -1.0   index:    8   merge:  Yes
i:  7   y:       -1.0   index:    8   merge:  Yes

Values after
i:  0   y:      110.0   index:   11   merge:  Yes
i:  1   y:       -1.0   index:   17   merge:   No
i:  2   y:       10.0   index:    1   merge:  Yes
i:  3   y:       60.0   index:    6   merge:  Yes
```

```
i:  4   y:      140.0   index:   14   merge:  Yes
i:  5   y:       -1.0   index:   13   merge:   No
i:  6   y:       80.0   index:    8   merge:  Yes
i:  7   y:       80.0   index:    8   merge:  Yes
```

# Summary

Here are the key learning points of Chapter 9:

- Nearly all AVX packed single-precision and double-precision floating-point instructions can be used with either 128-bit or 256-bit wide operands. Packed floating-point operands should always be properly aligned whenever possible, as described in this chapter.
- The MASM `align` directive cannot be used to align a 256-bit wide operand on a 32-byte boundary. Assembly language code can align 256-bit wide constant or mutable operands on a 32-byte boundary using the MASM `segment` directive.
- When performing packed arithmetic operations, the `vcmpp[d|s]` instructions can be used with the `vandp[d|s]`, `vandnp[d|s]`, and `vorp[d|s]` instructions to make logical decisions without any conditional jump instructions.
- The non-associativity of floating-point arithmetic means that minute numerical discrepancies may occur when comparing values calculated using C++ and assembly language functions.
- Assembly language functions can use the `vperm2f128`, `vpermp[d|s]`, and `vpermilp[d|s]` instructions to rearrange the elements of a packed floating-point operand.
- Assembly language functions can use the `vblendp[d|s]` and `vblendvp[d|s]` instructions to interleave the elements of two packed floating-point operands.
- Assembly language functions can use the `vgatherdp[d|s]` and `vgatherqp[d|s]` instructions to conditionally load floating-point values from non-contiguous memory locations into an XMM or YMM register.
- Assembly language functions that perform calculations using a YMM register should also use a `vzeroupper` instruction prior any epilog code or the `ret` instruction in order to avoid potential x86-AVX to x86-SSE state transition performance delays.

CHAPTER 10

# AVX2 Programming – Packed Integers

In Chapter 7, you learned how to use the AVX instruction set to perform packed integer operations using 128-bit wide operands and the XMM register set. In this chapter, you learn how to carry out similar operations using AVX2 instructions with 256-bit wide operands and the YMM register set. Chapter 10's source code examples are divided into two major sections. The first section contains elementary examples that illustrate basic operations using AVX2 instructions and 256-bit wide packed integer operands. The second section includes examples that are a continuation of the image processing techniques first presented in Chapter 7.

All of the source code examples in this chapter require a processor and operating system that supports AVX2. You can use one of the free utilities listed in Appendix A to verify the processing capabilities of your system.

## Packed Integer Fundamentals

In this section, you learn how to perform fundamental packed integer operations using AVX2 instructions. The first source code example expounds basic arithmetic using 256-bit wide operands and the YMM register set. The second source code example demonstrates AVX2 instructions that carry out integer pack and unpack operations. This example also explains how to return a structure by value from an assembly language function. The final source code example illuminates AVX2 instructions that execute packed integer size promotions using zero or sign extended values.

### Basic Arithmetic

Listing 10-1 shows the source code for example Ch10_01. This example illustrates how to perform basic arithmetic operations using packed word and doubleword operands.

***Listing 10-1.*** Example Ch10_01

```
//------------------------------------------------
//               Ch10_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "Ymmval.h"
```

© Daniel Kusswurm 2018

D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_10

```
using namespace std;

extern "C" void Avx2PackedMathI16_(const YmmVal& a, const YmmVal& b, YmmVal c[6]);
extern "C" void Avx2PackedMathI32_(const YmmVal& a, const YmmVal& b, YmmVal c[5]);

void Avx2PackedMathI16(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c[6];

    a.m_I16[0] = 10;       b.m_I16[0] = 1000;
    a.m_I16[1] = 20;       b.m_I16[1] = 2000;
    a.m_I16[2] = 3000;     b.m_I16[2] = 30;
    a.m_I16[3] = 4000;     b.m_I16[3] = 40;

    a.m_I16[4] = 30000;    b.m_I16[4] = 3000;       // add overflow
    a.m_I16[5] = 6000;     b.m_I16[5] = 32000;      // add overflow
    a.m_I16[6] = 2000;     b.m_I16[6] = -31000;     // sub overflow
    a.m_I16[7] = 4000;     b.m_I16[7] = -30000;     // sub overflow

    a.m_I16[8] = 4000;     b.m_I16[8] = -2500;
    a.m_I16[9] = 3600;     b.m_I16[9] = -1200;
    a.m_I16[10] = 6000;    b.m_I16[10] = 9000;
    a.m_I16[11] = -20000;  b.m_I16[11] = -20000;

    a.m_I16[12] = -25000;  b.m_I16[12] = -27000;    // add overflow
    a.m_I16[13] = 8000;    b.m_I16[13] = 28700;     // add overflow
    a.m_I16[14] = 3;       b.m_I16[14] = -32766;    // sub overflow
    a.m_I16[15] = -15000;  b.m_I16[15] = 24000;     // sub overflow

    Avx2PackedMathI16_(a, b, c);

    cout <<"\nResults for Avx2PackedMathI16_\n\n";
    cout << " i        a        b   vpaddw  vpaddsw   vpsubw  vpsubsw  vpminsw  vpmaxsw\n";
    cout << "--------------------------------------------------------------------------\n";

    for (int i = 0; i < 16; i++)
    {
        cout << setw(2)  << i << ' ';
        cout << setw(8) << a.m_I16[i] << ' ';
        cout << setw(8) << b.m_I16[i] << ' ';
        cout << setw(8) << c[0].m_I16[i] << ' ';
        cout << setw(8) << c[1].m_I16[i] << ' ';
        cout << setw(8) << c[2].m_I16[i] << ' ';
        cout << setw(8) << c[3].m_I16[i] << ' ';
        cout << setw(8) << c[4].m_I16[i] << ' ';
        cout << setw(8) << c[5].m_I16[i] << '\n';
    }
}
```

```
void Avx2PackedMathI32(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c[6];

    a.m_I32[0] = 64;        b.m_I32[0] = 4;
    a.m_I32[1] = 1024;      b.m_I32[1] = 5;
    a.m_I32[2] = -2048;     b.m_I32[2] = 2;
    a.m_I32[3] = 8192;      b.m_I32[3] = 5;
    a.m_I32[4] = -256;      b.m_I32[4] = 8;
    a.m_I32[5] = 4096;      b.m_I32[5] = 7;
    a.m_I32[6] = 16;        b.m_I32[6] = 3;
    a.m_I32[7] = 512;       b.m_I32[7] = 6;

    Avx2PackedMathI32_(a, b, c);

    cout << "\nResults for Avx2PackedMathI32\n\n";
    cout << " i      a      b   vpaddd   vpsubd  vpmulld  vpsllvd  vpsravd   vpabsd\n";
    cout << "----------------------------------------------------------------------\n";

    for (int i = 0; i < 8; i++)
    {
        cout << setw(2) << i << ' ';
        cout << setw(6) << a.m_I32[i] << ' ';
        cout << setw(6) << b.m_I32[i] << ' ';
        cout << setw(8) << c[0].m_I32[i] << ' ';
        cout << setw(8) << c[1].m_I32[i] << ' ';
        cout << setw(8) << c[2].m_I32[i] << ' ';
        cout << setw(8) << c[3].m_I32[i] << ' ';
        cout << setw(8) << c[4].m_I32[i] << ' ';
        cout << setw(8) << c[5].m_I32[i] << '\n';
    }
}

int main()
{
    Avx2PackedMathI16();
    Avx2PackedMathI32();
    return 0;
}

;-------------------------------------------------
;               Ch10_01.asm
;-------------------------------------------------

; extern "C" void Avx2PackedMathI16_(const YmmVal& a, const YmmVal& b, YmmVal c[6])

        .code
Avx2PackedMathI16_ proc
; Load values a and b, which must be properly aligned
```

```
        vmovdqa ymm0,ymmword ptr [rcx]      ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]      ;ymm1 = b

; Perform packed arithmetic operations
        vpaddw ymm2,ymm0,ymm1               ;add
        vmovdqa ymmword ptr [r8],ymm2       ;save vpaddw result

        vpaddsw ymm2,ymm0,ymm1              ;add with signed saturation
        vmovdqa ymmword ptr [r8+32],ymm2    ;save vpaddsw result

        vpsubw ymm2,ymm0,ymm1               ;sub
        vmovdqa ymmword ptr [r8+64],ymm2    ;save vpsubw result

        vpsubsw ymm2,ymm0,ymm1              ;sub with signed saturation
        vmovdqa ymmword ptr [r8+96],ymm2    ;save vpsubsw result

        vpminsw ymm2,ymm0,ymm1              ;signed minimums
        vmovdqa ymmword ptr [r8+128],ymm2   ;save vpminsw result

        vpmaxsw ymm2,ymm0,ymm1              ;signed maximums
        vmovdqa ymmword ptr [r8+160],ymm2   ;save vpmaxsw result

        vzeroupper
        ret
Avx2PackedMathI16_ endp

; extern "C" void Avx2PackedMathI32_(const YmmVal& a, const YmmVal& b, YmmVal c[6])

Avx2PackedMathI32_ proc
; Load values a and b, which must be properly aligned
        vmovdqa ymm0,ymmword ptr [rcx]      ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]      ;ymm1 = b

; Perform packed arithmetic operations
        vpaddd ymm2,ymm0,ymm1               ;add
        vmovdqa ymmword ptr [r8],ymm2       ;save vpaddd result

        vpsubd ymm2,ymm0,ymm1               ;sub
        vmovdqa ymmword ptr [r8+32],ymm2    ;save vpsubd result

        vpmulld ymm2,ymm0,ymm1              ;signed mul (low 32 bits)
        vmovdqa ymmword ptr [r8+64],ymm2    ;save vpmulld result

        vpsllvd ymm2,ymm0,ymm1              ;shift left logical
        vmovdqa ymmword ptr [r8+96],ymm2    ;save vpsllvd result

        vpsravd ymm2,ymm0,ymm1              ;shift right arithmetic
        vmovdqa ymmword ptr [r8+128],ymm2   ;save vpsravd result

        vpabsd ymm2,ymm0                    ;absolute value
        vmovdqa ymmword ptr [r8+160],ymm2   ;save vpabsd result
```

```
        vzeroupper
        ret
Avx2PackedMathI32_ endp
        end
```

The C++ function `Avx2PackedMathI16` contains code that demonstrates packed signed word arithmetic. This function begins with the definitions of `YmmVal` variables `a`, `b`, and `c`. Note that the C++ specifier `alignas(32)` is used with each `YmmVal` definition to ensure alignment on a 32-byte boundary. The signed word elements of both `a` and `b` are then initialized with test values. Following variable initialization, `Avx2PackedMathI16` calls the assembly language function `Avx2PackedMathI16_`, which performs several packed arithmetic operations. The results are then streamed to `cout`. The C++ function `Avx2PackedMathI32` is next. The structure of this function is similar to `Avx2PackedMathI16`, with the main difference being that it exercises packed doubleword operands.

The assembly language function `Avx2PackedMathI16_` begins with a `vmovdqa ymm0,ymmword ptr [rcx]` instruction that loads `YmmVal a` into register YMM0. The ensuing `vmovdqa ymm1,ymmword ptr [rdx]` instruction loads `YmmVal b` into register YMM1. This is followed by a `vpaddw ymm2,ymm0,ymm1` that performs packed word addition of `a` and `b`. The `vmovdqa ymmword ptr [r8],ymm2` instruction then saves packed word sums to `c[0]`. The remaining assembly language code in `Avx2PackedMathI16_` exercises the instructions `vpaddsw`, `vpsubw`, `vpsubsw`, `vpminsw`, and `vpmaxsw` to carry out additional arithmetic operations. Similar to the source code examples that you saw in Chapter 9, `Avx2PackedMathI16_` uses a `vzeroupper` instruction before its `ret` instruction. This avoids potential performance penalties that can occur when the processor transitions from executing x86-AVX instructions to x86-SSE instructions as explained in Chapter 8. The assembly language function `Avx2PackedMathI32_` employs a similar structure to exercise commonly-used packed doubleword instructions including `vpaddd`, `vpsubd`, `vpmulld`, `vpsllvd`, `vpsravd`, and `vpabsd`. Here are the results for source code example `Ch10_01`:

```
Results for Avx2PackedMathI16_

 i       a       b  vpaddw  vpaddsw   vpsubw  vpsubsw  vpminsw  vpmaxsw
----------------------------------------------------------------------
 0      10    1000    1010     1010     -990     -990       10     1000
 1      20    2000    2020     2020    -1980    -1980       20     2000
 2    3000      30    3030     3030     2970     2970       30     3000
 3    4000      40    4040     4040     3960     3960       40     4000
 4   30000    3000  -32536    32767    27000    27000     3000    30000
 5    6000   32000  -27536    32767   -26000   -26000     6000    32000
 6    2000  -31000  -29000   -29000   -32536    32767   -31000     2000
 7    4000  -30000  -26000   -26000   -31536    32767   -30000     4000
 8    4000   -2500    1500     1500     6500     6500    -2500     4000
 9    3600   -1200    2400     2400     4800     4800    -1200     3600
10    6000    9000   15000    15000    -3000    -3000     6000     9000
11  -20000  -20000   25536   -32768        0        0   -20000   -20000
12  -25000  -27000   13536   -32768     2000     2000   -27000   -25000
13    8000   28700  -28836    32767   -20700   -20700     8000    28700
14       3  -32766  -32763   -32763   -32767    32767   -32766        3
15  -15000   24000    9000     9000    26536   -32768   -15000    24000
```

```
Results for Avx2PackedMathI32

 i      a      b   vpaddd   vpsubd  vpmulld  vpsllvd  vpsravd   vpabsd
---------------------------------------------------------------------
 0     64      4       68       60      256     1024        4       64
 1   1024      5     1029     1019     5120    32768       32     1024
 2  -2048      2    -2046    -2050    -4096    -8192     -512     2048
 3   8192      5     8197     8187    40960   262144      256     8192
 4   -256      8     -248     -264    -2048   -65536       -1      256
 5   4096      7     4103     4089    28672   524288       32     4096
 6     16      3       19       13       48      128        2       16
 7    512      6      518      506     3072    32768        8      512
```

On systems that support AVX2, most of the instructions exercised in this example can be used with a variety of 256-bit wide packed integer operands. For example, the vpadd[b|q] and vpsub[b|q] instructions carry out addition and subtraction using 256-bit wide packed byte or quadword operands. The vpaddsb and vpsubsb instructions perform signed saturated addition and subtraction using packed byte operands. The instructions vpmins[b|d] and vpmaxs[b|d] calculate packed signed minimums and maximums, respectively. The variable bit shift instructions vpsllv[d|q], vpsravd, and vpsrlv[d|q] are new AVX2 instructions. These instructions are not available on systems that only support AVX.

## Pack and Unpack

Then next source code example illustrates how to perform integer pack and unpack operations. These operations are often employed to size-reduce or size-promote packed integer operands. This example also explains how to return a structure by value from an assembly language function. Listing 10-2 shows the source code for example Ch10_02

***Listing 10-2.*** Example Ch10_02

```
//------------------------------------------------
//              Ch10_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "YmmVal.h"

using namespace std;

struct alignas(32) YmmVal2
{
    YmmVal m_YmmVal0;
    YmmVal m_YmmVal1;
};

extern "C" YmmVal2 Avx2UnpackU32_U64_(const YmmVal& a, const YmmVal& b);
extern "C" void Avx2PackI32_I16_(const YmmVal& a, const YmmVal& b, YmmVal* c);
```

```
void Avx2UnpackU32_U64(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;

    a.m_U32[0] = 0x00000000;  b.m_U32[0] = 0x88888888;
    a.m_U32[1] = 0x11111111;  b.m_U32[1] = 0x99999999;
    a.m_U32[2] = 0x22222222;  b.m_U32[2] = 0xaaaaaaaa;
    a.m_U32[3] = 0x33333333;  b.m_U32[3] = 0xbbbbbbbb;

    a.m_U32[4] = 0x44444444;  b.m_U32[4] = 0xcccccccc;
    a.m_U32[5] = 0x55555555;  b.m_U32[5] = 0xdddddddd;
    a.m_U32[6] = 0x66666666;  b.m_U32[6] = 0xeeeeeeee;
    a.m_U32[7] = 0x77777777;  b.m_U32[7] = 0xffffffff;

    YmmVal2 c = Avx2UnpackU32_U64_(a, b);

    cout << "\nResults for Avx2UnpackU32_U64\n\n";

    cout << "a lo           " << a.ToStringX32(0) << '\n';
    cout << "b lo           " << b.ToStringX32(0) << '\n';
    cout << '\n';

    cout << "a hi           " << a.ToStringX32(1) << '\n';
    cout << "b hi           " << b.ToStringX32(1) << '\n';

    cout << "\nvpunpckldq result\n";
    cout << "c.m_YmmVal0 lo " << c.m_YmmVal0.ToStringX64(0) << '\n';
    cout << "c.m_YmmVal0 hi " << c.m_YmmVal0.ToStringX64(1) << '\n';

    cout << "\nvpunpckhdq result\n";
    cout << "c.m_YmmVal1 lo " << c.m_YmmVal1.ToStringX64(0) << '\n';
    cout << "c.m_YmmVal1 hi " << c.m_YmmVal1.ToStringX64(1) << '\n';
}

void Avx2PackI32_I16(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c;

    a.m_I32[0] = 10;          b.m_I32[0] = 32768;
    a.m_I32[1] = -200000;     b.m_I32[1] = 6500;
    a.m_I32[2] = 300000;      b.m_I32[2] = 42000;
    a.m_I32[3] = -4000;       b.m_I32[3] = -68000;

    a.m_I32[4] = 9000;        b.m_I32[4] = 25000;
    a.m_I32[5] = 80000;       b.m_I32[5] = 500000;
    a.m_I32[6] = 200;         b.m_I32[6] = -7000;
    a.m_I32[7] = -32769;      b.m_I32[7] = 12500;
```

```
    Avx2PackI32_I16_(a, b, &c);

    cout << "\nResults for Avx2PackI32_I16\n\n";

    cout << "a lo " << a.ToStringI32(0) << '\n';
    cout << "a hi " << a.ToStringI32(1) << '\n';
    cout << '\n';

    cout << "b lo " << b.ToStringI32(0) << '\n';
    cout << "b hi " << b.ToStringI32(1) << '\n';
    cout << '\n';

    cout << "c lo " << c.ToStringI16(0) << '\n';
    cout << "c hi " << c.ToStringI16(1) << '\n';
    cout << '\n';
}

int main()
{
    Avx2UnpackU32_U64();
    Avx2PackI32_I16();
    return 0;
}

;-------------------------------------------------
;               Ch10_02.asm
;-------------------------------------------------

; extern "C" YmmVal2 Avx2UnpackU32_U64_(const YmmVal& a, const YmmVal& b);

        .code
Avx2UnpackU32_U64_ proc

; Load argument values
        vmovdqa ymm0,ymmword ptr [rdx]      ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [r8]       ;ymm1 = b

; Perform dword to qword unpacks
        vpunpckldq ymm2,ymm0,ymm1           ;unpack low doublewords
        vpunpckhdq ymm3,ymm0,ymm1           ;unpack high doublewords

; Save result to YmmVal2 buffer
        vmovdqa ymmword ptr [rcx],ymm2      ;save low result
        vmovdqa ymmword ptr [rcx+32],ymm3   ;save high result

        mov rax,rcx                         ;rax = ptr to YmmVal2

        vzeroupper
        ret
Avx2UnpackU32_U64_ endp
```

```
; extern "C" void Avx2PackI32_I16_(const YmmVal& a, const YmmVal& b, YmmVal* c);

Avx2PackI32_I16_ proc

; Load argument values
        vmovdqa ymm0,ymmword ptr [rcx]      ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]      ;ymm1 = b

; Perform pack dword to word with signed saturation
        vpackssdw ymm2,ymm0,ymm1            ;ymm2 = packed words
        vmovdqa ymmword ptr [r8],ymm2       ;save result

        vzeroupper
        ret
Avx2PackI32_I16_ endp

Foo1_ proc
        ret
Foo1_ endp
        end
```

The C++ code in Listing 10-2 begins the declaration of a structure named YmmVal2. This structure contains two YmmVal members: m_YmmVal0 and m_YmmVal1. Note that the alignas(32) specifier is used immediately after the keyword struct. Using this specifier ensures that all instances of YmmVal2 are aligned on a 32-byte boundary including temporary instances created by the compiler. More on this in a moment. The assembly language function Avx2UnpackU32_U64_, whose declaration follows, returns an instance of YmmVal2 by value.

The C++ function AvxUnpackU32_U64 begins by initializing the unsigned doubleword elements of YmmVal variables a and b. Following variable initialization is the statement YmmVal2 c = Avx2UnpackU32_U64_(a, b), which calls the assembly language function Avx2UnpackU32_U64_ to unpack the elements of a and b from doublewords to quadwords. Unlike previous examples, Avx2UnpackU32_U64_ returns its YmmVal2 result by value. Before proceeding, it is important to note that in most cases, returning a user-defined structure like YmmVal2 by value is less efficient than passing a pointer argument to a variable of type YmmVal2. The function Avx2UnpackU32_U64_ uses return-by-value principally for demonstration purposes and to elucidate the Visual C++ calling convention protocols that an assembly language function must observe when returning a structure by value is warranted. The remaining statements in AvxUnpackU32_U64 stream the results from Avx2UnpackU32_U64_ to cout.

Following AvxUnpackU32_U64 is the C++ function Avx2PackI32_I16. This function initializes the signed doubleword elements of YmmVal variables a and b. These values will be size reduced to packed words. Subsequent to YmmVal variable initialization, Avx2PackI32_I16 calls the assembly language function Avx2PackI32_I16_ to carry out the aforementioned size reduction. The results are then streamed to cout.

The calling convention that Visual C++ uses for functions that return a structure by value varies somewhat from the normal calling convention. Upon entry to the assembly language function Avx2UnpackU32_U64_, register RCX points to a temporary buffer where Avx2UnpackU32_U64_ must store its YmmVal2 return result. It is important to note that this buffer is not necessarily the same memory location as the destination YmmVal2 variable in the C++ statement that called Avx2UnpackU32_U64_. In order to implement expression evaluation and operator overloading, a C++ compiler often generates code that allocates temporary variables (or *rvalues*) to hold intermediate results. An rvalue that needs to be saved is ultimately copied to a named variable (or *lvalue*) using either a default or overloaded assignment operator. This copy operation is the reason why returning a structure by value is usually slower than passing a pointer argument. The alignas(32) specifier that's used in the declaration of struct YmmVal2 directs the Visual C++ compiler to align all variables of type YmmVal2 *including* rvalues on a 32-byte boundary.

If the subject matter of the preceding paragraph seems a little abstract, don't worry. Temporary storage space allocation for return-by-value structures is handled automatically by the C++ compiler. It's more important to understand the following Visual C++ calling convention requirements that must be observed by any function that returns a large structure (any structure whose size is greater than eight bytes) by value:

- The caller of a function that returns a large structure by value must allocate storage space for the returned structure. A pointer to this storage space must be passed to the called function in register RCX.
- The normal calling convention argument registers are "right-shifted" by one. This means that the first three arguments are passed using registers RDX/XMM1, R8/XMM2, and R9/XMM3. Any remaining arguments are passed on the stack.
- Prior to returning, the called function must load register RAX with a pointer to the returned structure.

If the size of a return-by-value structure is less than or equal to eight bytes, it must be returned in register RAX. The normal calling convention argument registers are used in these situations.

Returning to the code, the first instruction of function `Avx2UnpackU32_U64_` uses a `vmovdqa ymm0,ymmword ptr [rdx]` instruction to load `YmmVal a` (the first function argument) into register YMM0. The ensuing `vmovdqa ymm1,ymmword ptr [r8]` instruction loads `YmmVal b` (the second function argument) into register YMM1. The next two instructions, `vpunpckldq ymm2,ymm0,ymm1` and `vpunpckhdq ymm3,ymm0,ymm1`, unpack the doublewords into quadwords, as shown in Figure 10-1. The results are then saved to the `YmmVal2` buffer pointed to by RCX using two `vmovdqa` instructions. Note that two `vmovdqu` instructions would be required here if the structure `YmmVal2` was declared without the `alignas(32)` specifier. As previously mentioned, the Visual C++ calling convention requires any function that returns a structure by value to load a copy of the structure buffer pointer into register RAX prior to returning. The `mov rax,rcx` instruction fulfills this requirement (recall that RCX contains a pointer to the structure buffer).

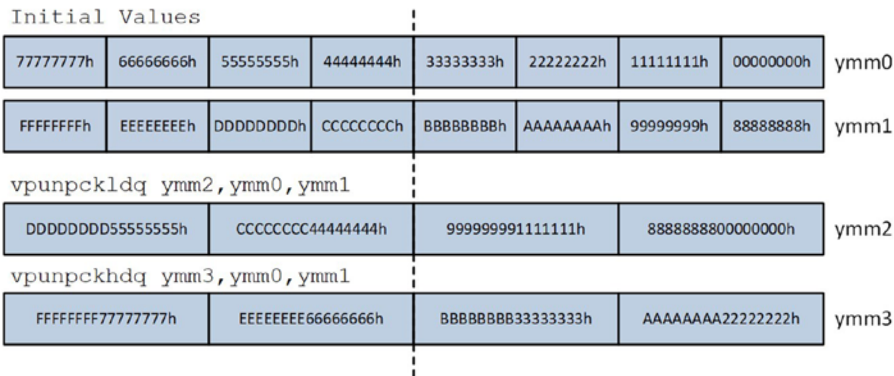



***Figure 10-1.*** *Execution of the `vpunpckldq` and `vpunpckhdq` instructions*

The assembly language function `Avx2PackI32_I16_` demonstrates use of the `vpackssdw` (Packed with Signed Saturation) instruction. In this function, the `vpackssdw ymm2,ymm0,ymm1` instruction converts the 16 doubleword integers in registers YMM0 and YMM1 to word integers using signed saturation. It then saves the 16 word integers in register YMM2. Figure 10-2 illustrates the execution of this instruction. X86-AVX also include a `vpacksswb` instruction that performs signed word to byte size reductions. The `vpackus[dw|wb]` instructions can be used for packed unsigned integer reductions.

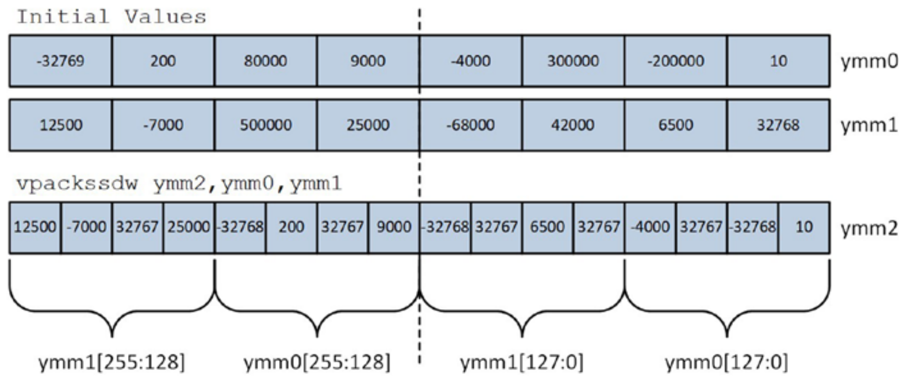



***Figure 10-2.** Execution of the vpackssdw instruction*

Note that in Figures 10-1 and 10-2, the vpunpckldq, vpunpckhdq, and vpackssdw instructions carry out their operations using two 128-bit wide independent lanes, as explained in Chapter 4. Here are the results for source code example Ch10_02:

```
Results for Avx2UnpackU32_U64

a lo                 00000000        11111111   |        22222222        33333333
b lo                 88888888        99999999   |        AAAAAAAA        BBBBBBBB

a hi                 44444444        55555555   |        66666666        77777777
b hi                 CCCCCCCC        DDDDDDDD   |        EEEEEEEE        FFFFFFFF

vpunpckldq result
c.m_YmmVal0 lo               8888888800000000   |                9999999911111111
c.m_YmmVal0 hi               CCCCCCCC44444444   |                DDDDDDDD55555555

vpunpckhdq result
c.m_YmmVal1 lo               AAAAAAAA22222222   |                BBBBBBBB33333333
c.m_YmmVal1 hi               EEEEEEEE66666666   |                FFFFFFFF77777777

Results for Avx2PackI32_I16

a lo              10          -200000   |          300000           -4000
a hi            9000            80000   |             200          -32769

b lo           32768             6500   |           42000          -68000
b hi           25000           500000   |           -7000           12500

c lo      10  -32768    32767   -4000   |   32767    6500   32767  -32768
c hi    9000   32767      200  -32768   |   25000   32767   -7000   12500
```

## Size Promotions

In Chapter 7, you learned how to use the used the vpunpckl[bw|dw] and vpunpckh[bw|wd] instructions to size-promote packed integers (see source code examples Ch07_05, Ch07_06, and Ch07_08). The next source code example, Ch10_03, demonstrates how to employ the vpmovzx[bw|bd] and vpmovsx[wd|wq] instructions to size-promote packed integers using either zero or sign extension. Listing 10-3 shows the source code for example Ch10_03.

***Listing 10-3.*** Example Ch10_03

```
//------------------------------------------------
//               Ch10_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <string>
#include "YmmVal.h"

using namespace std;

extern "C" void Avx2ZeroExtU8_U16_(YmmVal*a, YmmVal b[2]);
extern "C" void Avx2ZeroExtU8_U32_(YmmVal*a, YmmVal b[4]);
extern "C" void Avx2SignExtI16_I32_(YmmVal*a, YmmVal b[2]);
extern "C" void Avx2SignExtI16_I64_(YmmVal*a, YmmVal b[4]);

const string c_Line(80, '-');

void Avx2ZeroExtU8_U16(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b[2];

    for (int i = 0; i < 32; i++)
        a.m_U8[i] = (uint8_t)(i * 8);

    Avx2ZeroExtU8_U16_(&a, b);

    cout << "\nResults for Avx2ZeroExtU8_U16_\n";
    cout << c_Line << '\n';

    cout << "a (0:15):   " << a.ToStringU8(0) << '\n';
    cout << "a (16:31):  " << a.ToStringU8(1) << '\n';
    cout << '\n';
    cout << "b (0:7):    " << b[0].ToStringU16(0) << '\n';
    cout << "b (8:15):   " << b[0].ToStringU16(1) << '\n';
    cout << "b (16:23):  " << b[1].ToStringU16(0) << '\n';
    cout << "b (24:31):  " << b[1].ToStringU16(1) << '\n';
}
```

```
void Avx2ZeroExtU8_U32(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b[4];

    for (int i = 0; i < 32; i++)
        a.m_U8[i] = (uint8_t)(255 - i * 8);

    Avx2ZeroExtU8_U32_(&a, b);

    cout << "\nResults for Avx2ZeroExtU8_U32_\n";
    cout << c_Line << '\n';

    cout << "a (0:15):   " << a.ToStringU8(0) << '\n';
    cout << "a (16:31):  " << a.ToStringU8(1) << '\n';
    cout << '\n';
    cout << "b (0:3):    " << b[0].ToStringU32(0) << '\n';
    cout << "b (4:7):    " << b[0].ToStringU32(1) << '\n';
    cout << "b (8:11):   " << b[1].ToStringU32(0) << '\n';
    cout << "b (12:15):  " << b[1].ToStringU32(1) << '\n';
    cout << "b (16:19):  " << b[2].ToStringU32(0) << '\n';
    cout << "b (20:23):  " << b[2].ToStringU32(1) << '\n';
    cout << "b (24:27):  " << b[3].ToStringU32(0) << '\n';
    cout << "b (28:31):  " << b[3].ToStringU32(1) << '\n';
}

void Avx2SignExtI16_I32()
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b[2];

    for (int i = 0; i < 16; i++)
        a.m_I16[i] = (int16_t)(-32768 + i * 4000);

    Avx2SignExtI16_I32_(&a, b);

    cout << "\nResults for Avx2SignExtI16_I32_\n";
    cout << c_Line << '\n';

    cout << "a (0:7):    " << a.ToStringI16(0) << '\n';
    cout << "a (8:15):   " << a.ToStringI16(1) << '\n';
    cout << '\n';
    cout << "b (0:3):    " << b[0].ToStringI32(0) << '\n';
    cout << "b (4:7):    " << b[0].ToStringI32(1) << '\n';
    cout << "b (8:11):   " << b[1].ToStringI32(0) << '\n';
    cout << "b (12:15):  " << b[1].ToStringI32(1) << '\n';
}
```

```
void Avx2SignExtI16_I64()
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b[4];

    for (int i = 0; i < 16; i++)
        a.m_I16[i] = (int16_t)(32767 - i * 4000);

    Avx2SignExtI16_I64_(&a, b);

    cout << "\nResults for Avx2SignExtI16_I64_\n";
    cout << c_Line << '\n';

    cout << "a (0:7):    " << a.ToStringI16(0) << '\n';
    cout << "a (8:15):   " << a.ToStringI16(1) << '\n';
    cout << '\n';
    cout << "b (0:1):    " << b[0].ToStringI64(0) << '\n';
    cout << "b (2:3):    " << b[0].ToStringI64(1) << '\n';
    cout << "b (4:5):    " << b[1].ToStringI64(0) << '\n';
    cout << "b (6:7):    " << b[1].ToStringI64(1) << '\n';
    cout << "b (8:9):    " << b[2].ToStringI64(0) << '\n';
    cout << "b (10:11):  " << b[2].ToStringI64(1) << '\n';
    cout << "b (12:13):  " << b[3].ToStringI64(0) << '\n';
    cout << "b (14:15):  " << b[3].ToStringI64(1) << '\n';
}

int main()
{
    Avx2ZeroExtU8_U16();
    Avx2ZeroExtU8_U32();
    Avx2SignExtI16_I32();
    Avx2SignExtI16_I64();
    return 0;
}

;-------------------------------------------------
;               Ch10_03.asm
;-------------------------------------------------

; extern "C" void Avx2ZeroExtU8_U16_(YmmVal*a, YmmVal b[2]);

        .code
Avx2ZeroExtU8_U16_ proc
        vpmovzxbw ymm0,xmmword ptr [rcx]        ;zero extend a[0] - a[15]
        vpmovzxbw ymm1,xmmword ptr [rcx+16]     ;zero extend a[16] - a[31]

        vmovdqa ymmword ptr [rdx],ymm0          ;save results
        vmovdqa ymmword ptr [rdx+32],ymm1

        vzeroupper
        ret
```

```
Avx2ZeroExtU8_U16_ endp

; extern "C" void Avx2ZeroExtU8_U32_(YmmVal*a, YmmVal b[4]);

Avx2ZeroExtU8_U32_ proc
        vpmovzxbd ymm0,qword ptr [rcx]          ;zero extend a[0] - a[7]
        vpmovzxbd ymm1,qword ptr [rcx+8]        ;zero extend a[8] - a[15]
        vpmovzxbd ymm2,qword ptr [rcx+16]       ;zero extend a[16] - a[23]
        vpmovzxbd ymm3,qword ptr [rcx+24]       ;zero extend a[24] - a[31]

        vmovdqa ymmword ptr [rdx],ymm0          ;save results
        vmovdqa ymmword ptr [rdx+32],ymm1
        vmovdqa ymmword ptr [rdx+64],ymm2
        vmovdqa ymmword ptr [rdx+96],ymm3

        vzeroupper
        ret
Avx2ZeroExtU8_U32_ endp

; extern "C" void Avx2SignExtI16_I32_(YmmVal*a, YmmVal b[2])

Avx2SignExtI16_I32_ proc
        vpmovsxwd ymm0,xmmword ptr [rcx]        ;sign extend a[0] - a[7]
        vpmovsxwd ymm1,xmmword ptr [rcx+16]     ;sign extend a[8] - a[15]

        vmovdqa ymmword ptr [rdx],ymm0          ;save results
        vmovdqa ymmword ptr [rdx+32],ymm1

        vzeroupper
        ret
Avx2SignExtI16_I32_ endp

; extern "C" void Avx2SignExtI16_I64_(YmmVal*a, YmmVal b[4])

Avx2SignExtI16_I64_ proc
        vpmovsxwq ymm0,qword ptr [rcx]          ;sign extend a[0] - a[3]
        vpmovsxwq ymm1,qword ptr [rcx+8]        ;sign extend a[4] - a[7]
        vpmovsxwq ymm2,qword ptr [rcx+16]       ;sign extend a[8] - a[11]
        vpmovsxwq ymm3,qword ptr [rcx+24]       ;sign extend a[12] - a[15]

        vmovdqa ymmword ptr [rdx],ymm0          ;save results
        vmovdqa ymmword ptr [rdx+32],ymm1
        vmovdqa ymmword ptr [rdx+64],ymm2
        vmovdqa ymmword ptr [rdx+96],ymm3

        vzeroupper
        ret
Avx2SignExtI16_I64_ endp
        end
```

The C++ code in Listing 10-3 contains four functions that initialize test cases for various packed size-promotion operations. The first function, Avx2ZeroExtU8_U16, begins by initializing the unsigned byte elements of YmmVal a. It then calls the assembly language function Avx2ZeroExtU8_U16_ to size-promote the packed unsigned bytes into packed unsigned words. The function Avx2ZeroExtU8_U32 performs a similar set of initializations to demonstrate packed unsigned byte to packed unsigned doubleword promotions. The functions Avx2SignExtI16_I32 and Avx2SignExtI16_I64 initialize test cases for packed signed word to packed signed doubleword and packed signed quadword size promotions.

The first instruction in the assembly language function Avx2ZeroExtU8_U16_, vpmovzxbw ymm0,xmmword ptr [rcx], loads and zero-extends the 16 low-order bytes of YmmVal a (pointed to by register RCX) and saves these values in register YMM0. The ensuing vpmovzxbw ymm1,xmmword ptr [rcx+16] instruction performs the same operation using the 16 high-order bytes of YmmVal a. The function Avx2ZeroExtU8_U16_ then uses two vmovdqa instructions to save the size-promoted results.

The assembly language function Avx2ZeroExtU8_U32_ performs packed byte to doubleword size promotions. The first instruction, vpmovzxbd ymm0,qword ptr [rcx], loads and zero-extends the eight low-order bytes of YmmVal a into doublewords and saves these values in register YMM0. The three ensuing vpmovzxbd instructions size-promote the remaining byte values in YmmVal a. The results are then saved using a series of vmovdqa instructions. When working with unsigned 8-bit values, it is sometimes (depending on the algorithm) more expedient to use the vpmovzxbd instruction to perform a packed byte to packed doubleword size promotion instead of a semantically equivalent series of vpunpckl[bw|dw] and vpunpckh[bw|dw] instructions. You see an example of this in Chapter 14.

The assembly language functions Avx2SignExtI16_I32_ and Avx2SignExtI16_I64_ demonstrate how to use the vpmovsxwd and vpmovsxwq instructions, respectively. These instructions size-promote and sign-extend packed word integers to doublewords and quadwords. X86-AVX also includes the packed move with sign extension instructions vpmovsx[bw|bd|bq] and vpmovsxdq. Here is the output for source code example Ch10_03:

```
Results for Avx2ZeroExtU8_U16_
----------------------------------------------------------------------------
a (0:15):      0   8  16  24  32  40  48  56    |  64  72  80  88  96 104 112 120
a (16:31):   128 136 144 152 160 168 176 184    | 192 200 208 216 224 232 240 248

b (0:7):           0       8      16      24    |      32      40      48      56
b (8:15):         64      72      80      88    |      96     104     112     120
b (16:23):       128     136     144     152    |     160     168     176     184
b (24:31):       192     200     208     216    |     224     232     240     248

Results for Avx2ZeroExtU8_U32_
----------------------------------------------------------------------------
a (0:15):    255 247 239 231 223 215 207 199    | 191 183 175 167 159 151 143 135
a (16:31):   127 119 111 103  95  87  79  71    |  63  55  47  39  31  23  15   7

b (0:3):                 255             247    |             239             231
b (4:7):                 223             215    |             207             199
b (8:11):                191             183    |             175             167
b (12:15):               159             151    |             143             135
b (16:19):               127             119    |             111             103
b (20:23):                95              87    |              79              71
b (24:27):                63              55    |              47              39
b (28:31):                31              23    |              15               7
```

```
Results for Avx2SignExtI16_I32_
--------------------------------------------------------------------------------
a (0:7):      -32768  -28768  -24768  -20768  |  -16768  -12768   -8768   -4768
a (8:15):       -768    3232    7232   11232  |   15232   19232   23232   27232

b (0:3):              -32768          -28768  |          -24768          -20768
b (4:7):              -16768          -12768  |           -8768           -4768
b (8:11):               -768            3232  |            7232           11232
b (12:15):             15232           19232  |           23232           27232

Results for Avx2SignExtI16_I64_
--------------------------------------------------------------------------------
a (0:7):       32767   28767   24767   20767  |   16767   12767    8767    4767
a (8:15):        767   -3233   -7233  -11233  |  -15233  -19233  -23233  -27233

b (0:1):                               32767  |                           28767
b (2:3):                               24767  |                           20767
b (4:5):                               16767  |                           12767
b (6:7):                                8767  |                            4767
b (8:9):                                 767  |                           -3233
b (10:11):                             -7233  |                          -11233
b (12:13):                            -15233  |                          -19233
b (14:15):                            -23233  |                          -27233
```

# Packed Integer Image Processing

In Chapter 7, you learned how to use the AVX instruction set to perform some common image processing operations using 128-bit wide packed unsigned integer operands. The source code examples of this section demonstrate additional image processing methods using AXV2 instructions with 256-bit wide packed unsigned integer operands. The first source example illustrates how to clip the pixel intensity values of a grayscale image. This is followed by an example that determines the minimum and maximum pixel intensity values of an RGB image. The final source code example uses the AVX2 instruction set to perform RGB to grayscale image conversion.

## Pixel Clipping

Pixel clipping is an image processing technique that bounds the intensity values of each pixel in an image between two threshold limits. This technique is often used to reduce the dynamic range of an image by eliminating its extremely dark and light pixels. Source code example Ch10_04 illustrates how to use the AVX2 instruction set to clip the pixels of an 8-bit grayscale image. Listing 10-4 shows the C++ and assembly language source code for example Ch10_04.

***Listing 10-4.*** Example Ch10_04

```
//------------------------------------------------
//               Ch10_04.h
//------------------------------------------------

#pragma once
#include <cstdint>
```

```
// The following structure must match the structure that's declared in the file .asm file
struct ClipData
{
    uint8_t* m_Src;                 // source buffer pointer
    uint8_t* m_Des;                 // destination buffer pointer
    uint64_t m_NumPixels;           // number of pixels
    uint64_t m_NumClippedPixels;    // number of clipped pixels
    uint8_t m_ThreshLo;             // low threshold
    uint8_t m_ThreshHi;             // high threshold
};

// Functions defined in Ch10_04.cpp
extern void Init(uint8_t* x, uint64_t n, unsigned int seed);
extern bool Avx2ClipPixelsCpp(ClipData* cd);

// Functions defined in Ch10_04_.asm
extern "C" bool Avx2ClipPixels_(ClipData* cd);

// Functions defined in Ch10_04_BM.cpp
extern void Avx2ClipPixels_BM(void);

//------------------------------------------------
//               Ch10_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <random>
#include <memory.h>
#include <limits>
#include "Ch10_04.h"
#include "AlignedMem.h"

using namespace std;

void Init(uint8_t* x, uint64_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {0, 255};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (uint8_t)ui_dist(rng);
}

bool Avx2ClipPixelsCpp(ClipData* cd)
{
    uint8_t* src = cd->m_Src;
    uint8_t* des = cd->m_Des;
    uint64_t num_pixels = cd->m_NumPixels;
```

```
    if (num_pixels == 0 || (num_pixels % 32) != 0)
        return false;

    if (!AlignedMem::IsAligned(src, 32) || !AlignedMem::IsAligned(des, 32))
        return false;

    uint64_t num_clipped_pixels = 0;
    uint8_t thresh_lo = cd->m_ThreshLo;
    uint8_t thresh_hi = cd->m_ThreshHi;

    for (uint64_t i = 0; i < num_pixels; i++)
    {
        uint8_t pixel = src[i];

        if (pixel < thresh_lo)
        {
            des[i] = thresh_lo;
            num_clipped_pixels++;
        }
        else if (pixel > thresh_hi)
        {
            des[i] = thresh_hi;
            num_clipped_pixels++;
        }
        else
            des[i] = src[i];
    }

    cd->m_NumClippedPixels = num_clipped_pixels;
    return true;
}

void Avx2ClipPixels(void)
{
    const uint8_t thresh_lo = 10;
    const uint8_t thresh_hi = 245;
    const uint64_t num_pixels = 4 * 1024 * 1024;

    AlignedArray<uint8_t> src(num_pixels, 32);
    AlignedArray<uint8_t> des1(num_pixels, 32);
    AlignedArray<uint8_t> des2(num_pixels, 32);

    Init(src.Data(), num_pixels, 157);

    ClipData cd1;
    ClipData cd2;

    cd1.m_Src = src.Data();
    cd1.m_Des = des1.Data();
```

```
    cd1.m_NumPixels = num_pixels;
    cd1.m_NumClippedPixels = numeric_limits<uint64_t>::max();
    cd1.m_ThreshLo = thresh_lo;
    cd1.m_ThreshHi = thresh_hi;

    cd2.m_Src = src.Data();
    cd2.m_Des = des2.Data();
    cd2.m_NumPixels = num_pixels;
    cd2.m_NumClippedPixels = numeric_limits<uint64_t>::max();
    cd2.m_ThreshLo = thresh_lo;
    cd2.m_ThreshHi = thresh_hi;

    Avx2ClipPixelsCpp(&cd1);
    Avx2ClipPixels_(&cd2);

    cout << "\nResults for Avx2ClipPixels\n";
    cout << "  cd1.m_NumClippedPixels1: " << cd1.m_NumClippedPixels << '\n';
    cout << "  cd2.m_NumClippedPixels2: " << cd2.m_NumClippedPixels << '\n';

    if (cd1.m_NumClippedPixels != cd2.m_NumClippedPixels)
        cout << "  NumClippedPixels compare error\n";

    if (memcmp(des1.Data(), des2.Data(), num_pixels) == 0)
        cout << "  Pixel buffer memory compare passed\n";
    else
        cout << "  Pixel buffer memory compare passed\n";
}

int main(void)
{
    Avx2ClipPixels();
    Avx2ClipPixels_BM();
    return 0;
}

;-------------------------------------------------
;               Ch10_04.asm
;-------------------------------------------------

; The following structure must match the structure that's declared in the file .h file
ClipData            struct
Src                 qword ?             ;source buffer pointer
Des                 qword ?             ;destination buffer pointer
NumPixels           qword ?             ;number of pixels
NumClippedPixels    qword ?             ;number of clipped pixels
ThreshLo            byte ?              ;low threshold
ThreshHi            byte ?              ;high threshold
ClipData            ends

; extern "C" bool Avx2ClipPixels_(ClipData* cd)
```

```
            .code
Avx2ClipPixels_ proc

; Load and validate arguments
        xor eax,eax                         ;set error return code
        xor r8d,r8d                         ;r8 = number of clipped pixels

        mov rdx,[rcx+ClipData.NumPixels]    ;rdx = num_pixels
        or rdx,rdx
        jz Done                             ;jump of num_pixels is zero
        test rdx,1fh
        jnz Done                            ;jump if num_pixels % 32 != 0

        mov r10,[rcx+ClipData.Src]          ;r10 = Src
        test r10,1fh
        jnz Done                            ;jump if Src is misaligned

        mov r11,[rcx+ClipData.Des]          ;r11 = Des
        test r11,1fh
        jnz Done                            ;jump if Des is misaligned

; Create packed thresh_lo and thresh_hi data values
        vpbroadcastb ymm4,[rcx+ClipData.ThreshLo]   ;ymm4 = packed thresh_lo
        vpbroadcastb ymm5,[rcx+ClipData.ThreshHi]   ;ymm5 = packed thresh_hi

; Clip pixels to threshold values
@@:     vmovdqa ymm0,ymmword ptr [r10]      ;ymm0 = 32 pixels
        vpmaxub ymm1,ymm0,ymm4              ;clip to thresh_lo
        vpminub ymm2,ymm1,ymm5              ;clip to thresh_hi
        vmovdqa ymmword ptr [r11],ymm2      ;save clipped pixels

; Count number of clipped pixels
        vpcmpeqb ymm3,ymm2,ymm0             ;compare clipped pixels to original
        vpmovmskb eax,ymm3                  ;eax = mask of non-clipped pixels
        not eax                             ;eax = mask of clipped pixels
        popcnt eax,eax                      ;eax = number of clipped pixels
        add r8,rax                          ;update clipped pixel count

; Update pointers and loop counter
        add r10,32                          ;update Src ptr
        add r11,32                          ;update Des ptr
        sub rdx,32                          ;update loop counter
        jnz @B                              ;repeat if not done

        mov eax,1                           ;set success return code

; Save num_clipped_pixels
```

```
Done:   mov [rcx+ClipData.NumClippedPixels],r8  ;save num_clipped_pixels
        vzeroupper
        ret

Avx2ClipPixels_ endp
        end
```

The C++ code begins with declaration of a structure named ClipData. This structure and its assembly language equivalent are used to maintain the pixel-clipping algorithm's data. Following the function declarations in the header file Ch10_04.h is the definition of a C++ function named Init. This function initializes the elements of a uint8_t array using random values, which simulates the pixel values of a grayscale image. The function Avx2ClipPixelCpp is a C++ implementation of the pixel clipping algorithm. This function starts by validating num_pixels for correct size and divisibility by 32. Restricting the algorithm to images that contain an even multiple of 32 pixels is not as inflexible as it might appear. Most digital camera images are sized using multiples of 64 pixels due to the processing requirements of the JPEG compression algorithms. Following validation of num_pixels, the source and destination pixel buffers are checked for proper alignment.

The procedure used in Avx2ClipPixelCpp to perform pixel clipping is straightforward. A simple for loop examines each pixel element in the source image buffer. If a source image pixel buffer intensity value found to be below thresh_lo or above thresh_hi, the corresponding threshold limit is saved in the destination buffer. Source image pixels whose intensity values lie between the two threshold limits are copied to the destination pixel buffer unaltered. The processing loop in Avx2ClipPixelCpp also counts the number of clipped pixels for comparison purposes with the assembly language version of the algorithm.

Function Avx2ClipPixels exploits the C++ template class AlignedArray to allocate and manage the required image pixel buffers (see Chapter 7 for a description of this class). Following source image pixel buffer initialization, Avx2ClipPixels primes two instances of ClipData (cd1 and cd2) for use by the pixel clipping functions Avx2ClipPixelsCpp and Avx2ClipPixels_. It then invokes these functions and compares the results for any discrepancies.

Toward the top of the assembly language code is the declaration for data structure ClipPixel, which is semantically equivalent to its C++ counterpart. The function Avx2ClipPixels_ begins its execution by validating num_pixels for size and divisibility by 32. It then checks the source and destination pixels buffers for proper alignment. Following argument validation, Avx2ClipPixels_ employs two vpbroadcastb instructions to create packed versions of the threshold limit values thresh_lo and thresh_hi in registers YMM4 and YMM5, respectively. During each processing loop iteration, the vmovdqa ymm0,ymmword ptr [r10] instruction loads 32 pixel values from the source image pixel buffer into register YMM0. The ensuing vpmaxub ymm1,ymm0,ymm4 instruction clips the pixel values in YMM0 to thresh_lo. This is followed by a vpminub ymm2,ymm1,ymm5 instruction that clips the pixel values to thresh_hi. The vmovdqa ymmword ptr [r11],ymm2 instruction then saves the clipped pixel intensity values to the destination image pixel buffer.

Avx2ClipPixels_ counts the number of clipped pixels using a straightforward sequence of instructions. The vpcmpeqb ymm3,ymm2,ymm0 instruction compares the original pixel values in YMM0 to the clipped pixel values in YMM2 for equality. Each byte element in YMM3 is set to 0xff if the original and clipped pixel intensity values are equal; otherwise, the YMM3 byte element is set to 0x00. The vpmovmskb eax,ymm3 instruction that follows creates a mask of the most significant bit of each byte element in YMM3 and saves this mask to register EAX. More specifically, this instruction computes eax[i] = ymm3[i*8+7] for i = 0, 1, 2, … 31, which means that each 1 bit in register EAX signifies a non-clipped pixel. The ensuing not eax instruction converts the bit pattern in EAX to a mask of clipped pixels, and the popcnt eax,eax instruction counts the number of 1 bits in EAX. This count value, which corresponds to the number of clipped pixels in YMM2, is then added to the total number of clipped pixels in register R8. The processing loop repeats until all pixels have been processed. Here are the results for source code example Ch10_04:

```
Results for Avx2ClipPixels
  cd1.m_NumClippedPixels1: 328090
  cd2.m_NumClippedPixels2: 328090
  Pixel buffer memory compare passed

Running benchmark function Avx2ClipPixels_BM - please wait
Benchmark times save to file Ch10_04_Avx2ClipPixels_BM_CHROMIUM.csv
```

Table 10-1 shows the benchmark timing measurements for the pixel clipping functions `Avx2ClipPixelsCpp` and `Avx2ClipPixels_`.

***Table 10-1.*** *Mean Execution Times (Microseconds) for Pixel Clipping Functions (Image Buffer Size = 8 MB)*

| CPU | Avx2ClipPixelsCpp | Avx2ClipPixels_ |
|---|---|---|
| i7-4790S | 13005 | 1078 |
| i9-7900X | 11617 | 719 |
| i7-8700K | 11252 | 644 |

## RGB Pixel Min-Max Values

Listing 10-5 shows the C++ and assembly language source code for example `Ch10_05`, which illustrates how to calculate the minimum and maximum pixel intensity values in an RGB image. This example also explains how to exploit some of MASM's advanced macro processing capabilities.

***Listing 10-5.*** Example Ch10_05

```
//------------------------------------------------
//               Ch10_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <random>
#include "AlignedMem.h"

using namespace std;

extern "C" bool Avx2CalcRgbMinMax_(uint8_t* rgb[3], size_t num_pixels, uint8_t min_vals[3],
uint8_t max_vals[3]);

void Init(uint8_t* rgb[3], size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {5, 250};
    default_random_engine rng {seed};
```

```
    for (size_t i = 0; i < n; i++)
    {
        rgb[0][i] = (uint8_t)ui_dist(rng);
        rgb[1][i] = (uint8_t)ui_dist(rng);
        rgb[2][i] = (uint8_t)ui_dist(rng);
    }

    // Set known min & max values for validation purposes
    rgb[0][n / 4] = 4;   rgb[1][n / 2] = 1;       rgb[2][3 * n / 4] = 3;
    rgb[0][n / 3] = 254; rgb[1][2 * n / 5] = 251; rgb[2][n - 1] = 252;
}

bool Avx2CalcRgbMinMaxCpp(uint8_t* rgb[3], size_t num_pixels, uint8_t min_vals[3], uint8_t
max_vals[3])
{
    // Make sure num_pixels is valid
    if ((num_pixels == 0) || (num_pixels % 32 != 0))
        return false;

    if (!AlignedMem::IsAligned(rgb[0], 32))
        return false;
    if (!AlignedMem::IsAligned(rgb[1], 32))
        return false;
    if (!AlignedMem::IsAligned(rgb[2], 32))
        return false;

    // Find the min and max of each color plane
    min_vals[0] = min_vals[1] = min_vals[2] = 255;
    max_vals[0] = max_vals[1] = max_vals[2] = 0;

   for (size_t i = 0; i < 3; i++)
   {
        for (size_t j = 0; j < num_pixels; j++)
        {
            if (rgb[i][j] < min_vals[i])
                min_vals[i] = rgb[i][j];
            else if (rgb[i][j] > max_vals[i])
                max_vals[i] = rgb[i][j];
        }
    }

    return true;
}

int main(void)
{
    const size_t n = 1024;
    uint8_t* rgb[3];
    uint8_t min_vals1[3], max_vals1[3];
    uint8_t min_vals2[3], max_vals2[3];
```

```
    AlignedArray<uint8_t> r(n, 32);
    AlignedArray<uint8_t> g(n, 32);
    AlignedArray<uint8_t> b(n, 32);

    rgb[0] = r.Data();
    rgb[1] = g.Data();
    rgb[2] = b.Data();

    Init(rgb, n, 219);
    Avx2CalcRgbMinMaxCpp(rgb, n, min_vals1, max_vals1);
    Avx2CalcRgbMinMax_(rgb, n, min_vals2, max_vals2);

    cout << "Results for Avx2CalcRgbMinMax\n\n";
    cout << "                R   G   B\n";
    cout << "-------------------------\n";

    cout << "min_vals1: ";
    cout << setw(4) << (int)min_vals1[0]  << ' ';
    cout << setw(4) << (int)min_vals1[1]  << ' ';
    cout << setw(4) << (int)min_vals1[2]  << '\n';
    cout << "min_vals2: ";
    cout << setw(4) << (int)min_vals2[0]  << ' ';
    cout << setw(4) << (int)min_vals2[1]  << ' ';
    cout << setw(4) << (int)min_vals2[2]  << "\n\n";

    cout << "max_vals1: ";
    cout << setw(4) << (int)max_vals1[0]  << ' ';
    cout << setw(4) << (int)max_vals1[1]  << ' ';
    cout << setw(4) << (int)max_vals1[2]  << '\n';
    cout << "max_vals2: ";
    cout << setw(4) << (int)max_vals2[0]  << ' ';
    cout << setw(4) << (int)max_vals2[1]  << ' ';
    cout << setw(4) << (int)max_vals2[2]  << "\n\n";

    return 0;
}

;-------------------------------------------------
;               Ch10_05.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; 256-bit wide constants
ConstVals       segment readonly align(32) 'const'
InitialPminVal  db 32 dup(0ffh)
InitialPmaxVal  db 32 dup(00h)
ConstVals       ends

; Macro _YmmVpextrMinub
;
```

```
; This macro generates code that extracts the smallest unsigned byte from register YmmSrc.

_YmmVpextrMinub macro GprDes,YmmSrc,YmmTmp

; Make sure YmmSrc and YmmTmp are different
.erridni <YmmSrc>, <YmmTmp>, <Invalid registers>

; Construct text strings for the corresponding XMM registers
        YmmSrcSuffix SUBSTR <YmmSrc>,2
        XmmSrc CATSTR <X>,YmmSrcSuffix

        YmmTmpSuffix SUBSTR <YmmTmp>,2
        XmmTmp CATSTR <X>,YmmTmpSuffix

; Reduce the 32 byte values in YmmSrc to the smallest value
        vextracti128 XmmTmp,YmmSrc,1
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 16 min values

        vpsrldq XmmTmp,XmmSrc,8
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 8 min values

        vpsrldq XmmTmp,XmmSrc,4
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 4 min values

        vpsrldq XmmTmp,XmmSrc,2
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 2 min values

        vpsrldq XmmTmp,XmmSrc,1
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 1 min value

        vpextrb GprDes,XmmSrc,0             ;mov final min value to Gpr
        endm

; Macro _YmmVpextrMaxub
;
; This macro generates code that extracts the largest unsigned byte from register YmmSrc.

_YmmVpextrMaxub macro GprDes,YmmSrc,YmmTmp

; Make sure YmmSrc and YmmTmp are different
.erridni <YmmSrc>, <YmmTmp>, <Invalid registers>

; Construct text strings for the corresponding XMM registers
        YmmSrcSuffix SUBSTR <YmmSrc>,2
        XmmSrc CATSTR <X>,YmmSrcSuffix

        YmmTmpSuffix SUBSTR <YmmTmp>,2
        XmmTmp CATSTR <X>,YmmTmpSuffix

; Reduce the 32 byte values in YmmSrc to the largest value
        vextracti128 XmmTmp,YmmSrc,1
```

```
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 16 max values

        vpsrldq XmmTmp,XmmSrc,8
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 8 max values

        vpsrldq XmmTmp,XmmSrc,4
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 4 max values

        vpsrldq XmmTmp,XmmSrc,2
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 2 max values

        vpsrldq XmmTmp,XmmSrc,1
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = final 1 max value

        vpextrb GprDes,XmmSrc,0             ;mov final max value to Gpr
        endm

; extern "C" bool Avx2CalcRgbMinMax_(uint8_t* rgb[3], size_t num_pixels, uint8_t min_
vals[3], uint8_t max_vals[3])

        .code
Avx2CalcRgbMinMax_ proc frame
        _CreateFrame CalcMinMax_,0,48,r12
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Make sure num_pixels and the color plane arrays are valid
        xor eax,eax                         ;set error code

        test rdx,rdx
        jz Done                             ;jump if num_pixels == 0
        test rdx,01fh
        jnz Done                            ;jump if num_pixels % 32 != 0

        mov r10,[rcx]                       ;r10 = color plane R
        test r10,1fh
        jnz Done                            ;jump if color plane R is not aligned

        mov r11,[rcx+8]                     ;r11 = color plane G
        test r11,1fh
        jnz Done                            ;jump if color plane G is not aligned

        mov r12,[rcx+16]                    ;r12 = color plane B
        test r12,1fh
        jnz Done                            ;jump if color plane B is not aligned

; Initialize the processing loop registers
        vmovdqa ymm3,ymmword ptr [InitialPminVal]   ;ymm3 = R minimums
        vmovdqa ymm4,ymm3                           ;ymm4 = G minimums
        vmovdqa ymm5,ymm3                           ;ymm5 = B minimums
```

```
        vmovdqa ymm6,ymmword ptr [InitialPmaxVal]   ;ymm6 = R maximums
        vmovdqa ymm7,ymm6                           ;ymm7 = G maximums
        vmovdqa ymm8,ymm6                           ;ymm8 = B maximums

        xor rcx,rcx                         ;rcx = common array offset

; Scan RGB color plane arrays for packed minimums and maximums
        align 16
@@:     vmovdqa ymm0,ymmword ptr [r10+rcx]  ;ymm0 = R pixels
        vmovdqa ymm1,ymmword ptr [r11+rcx]  ;ymm1 = G pixels
        vmovdqa ymm2,ymmword ptr [r12+rcx]  ;ymm2 = B pixels

        vpminub ymm3,ymm3,ymm0              ;update R minimums
        vpminub ymm4,ymm4,ymm1              ;update G minimums
        vpminub ymm5,ymm5,ymm2              ;update B minimums

        vpmaxub ymm6,ymm6,ymm0              ;update R maximums
        vpmaxub ymm7,ymm7,ymm1              ;update G maximums
        vpmaxub ymm8,ymm8,ymm2              ;update B maximums

        add rcx,32
        sub rdx,32
        jnz @B

; Calculate the final RGB minimum values
        _YmmVpextrMinub rax,ymm3,ymm0
        mov byte ptr [r8],al                ;save min R
        _YmmVpextrMinub rax,ymm4,ymm0
        mov byte ptr [r8+1],al              ;save min G
        _YmmVpextrMinub rax,ymm5,ymm0
        mov byte ptr [r8+2],al              ;save min B

; Calculate the final RGB maximum values
        _YmmVpextrMaxub rax,ymm6,ymm1
        mov byte ptr [r9],al                ;save max R
        _YmmVpextrMaxub rax,ymm7,ymm1
        mov byte ptr [r9+1],al              ;save max G
        _YmmVpextrMaxub rax,ymm8,ymm1
        mov byte ptr [r9+2],al              ;save max B

        mov eax,1                           ;set success return code

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame r12
        ret
Avx2CalcRgbMinMax_ endp
        end
```

The function Avx2CalcRgbMinMaxCpp that's shown in Listing 10-5 is a C++ implementation of the RGB min-max algorithm. This function employs a set of nested for loops to determine the minimum and maximum pixel intensity values for each color plane. These values are maintained in the arrays min_vals and max_vals. The function main uses the C++ template class AlignedArray to allocate three arrays that simulate the color plane buffers of an RGB image. These buffers are loaded with random values by the function Init. Note that function Init assigns known values to several elements in each color plane buffer. These known values are used to verify correct execution of both the C++ and assembly language min-max functions.

Toward the top of the assembly language code is a custom constant segment named ConstVals that defines packed versions of the initial pixel minimum and maximum values. A custom segment is used here to ensure alignment of the 256-bit wide packed values on a 32-byte boundary, as explained in Chapter 9. The macro definitions _YmmVpextrMinub and _YmmVpextrMaxub are next. These macros contain instructions that extract the smallest and largest byte values from a YMM register. The inner workings of these macros will be explained shortly.

The function Avx2CalcRgbMinMax_ uses registers YMM3-YMM5 and YMM6-YMM8 to maintain the RGB minimum and maximum values, respectively. During each iteration of the main processing loop, a series of vpminub and vpmaxub instructions update the current RGB minimums and maximums. Upon completion of the main processing loop, the aforementioned YMM registers contain 32 minimum and maximum pixel intensity values for each color component. The _YmmVpextrMinub and _YmmVpextrMaxub macros are then used to extract the final RGB minimum and maximum pixel values. These values are then saved to the result arrays min_vals and max_vals, respectively.

The macros definitions _YmmVpextrMinub and _YmmVpextrMaxub are identical, except for the instructions vpminub and vpmaxub. In the text that follows, all explanatory comments made about _YmmVpextrMinub also apply to _YmmVpextrMaxub. The _YmmVpextrMinub macro requires three parameters: a destination general-purpose register (GprDes), a source YMM register (YmmSrc), and a temporary YMM register (YmmTmp). Note that macro parameters YmmSrc and YmmTmp must be different registers. If they're the same, the .erridni directive (Error if Text Items are Identical, Case Insensitive) generates an error message during assembly. MASM also supports several other conditional error directives besides .erridni, and these are described in the Visual Studio documentation.

In order to generate the correct assembly language code, the macro _YmmVpextrMinub requires an XMM register text string (XmmSrc) that corresponds to the low-order portion of the specified YmmSrc register. For example, if YmmSrc equals *YMM0*, then XmmSrc must equal *XMM0*. The MASM directives substr (Return Substring of Text Item) and catstr (Concatenate Text Items) are used to initialize XmmSrc. The statement YmmSrcSuffix SUBSTR <YmmSrc>,2 assigns a text string value to YmmSrcSuffix that excludes the leading character of macro parameter YmmSrc. For example, if YmmSrc equals *YMM0*, then YmmSrcSuffix equals *MM0*. The next statement, XmmSrc CATSTR <X>,YmmSrcSuffix, adds a leading *X* to the value of YmmSrcSuffix and assigns it to XmmSrc. Continuing with the earlier example, this means that the text string *XMM0* is assigned to XmmSrc. The SUBSTR and CATSTR directives are then used to assign a text string value to XmmTmp.

Following initialization of the required macro text strings are the instructions that extract the smallest byte value from the specified YMM register. The vextracti128 XmmTmp,YmmSrc,1 instruction copies the high-order 16 bytes of register YmmSrc to XmmTmp. (The vextracti128 instruction also supports using an immediate operand of 0 to copy the low-order 16 bytes.) A vpminub XmmSrc,XmmSrc,XmmTmp instruction loads the final 16 minimum values into XmmSrc. The vpsrldq XmmTmp,XmmSrc,8 instruction shifts a copy of the value that's in XmmSrc to the right by eight bytes and saves the result to XmmTmp. This facilitates the use of another vpminub instruction that reduces the number of minimum byte values from 16 to 8. Repeated sets of the vpsrldq and vpminub instructions are then employed until the final minimum value resides in the

low-order byte of XmmSrc. A vpextrb GprDes,XmmSrc,0 instruction copies the final minimum value to the specified general-purpose register. Here are the results for source code example Ch10_05:

```
Results for Avx2CalcRgbMinMax

                R   G   B
-------------------------
min_vals1:    4    1    3
min_vals2:    4    1    3

max_vals1:  254  251  252
max_vals2:  254  251  252
```

## RGB to Grayscale Conversion

The final source code example of this chapter, Ch10_06, explains how to perform an RGB to grayscale image conversion. This example intermixes the packed integer capabilities of AVX2 that you have learned in this chapter with the packed floating-point techniques presented in Chapter 9. Listing 10-6 shows the source code for example Ch10_06

***Listing 10-6.*** Example Ch10_06

```
//------------------------------------------------
//               ImageMatrix.h
//------------------------------------------------

struct RGB32
{
    uint8_t m_R;
    uint8_t m_G;
    uint8_t m_B;
    uint8_t m_A;
};

//------------------------------------------------
//               Ch10_06.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <stdexcept>
#include "Ch10_06.h"
#include "AlignedMem.h"
#include "ImageMatrix.h"

using namespace std;

// Image size limits
extern "C" const int c_NumPixelsMin = 32;
extern "C" const int c_NumPixelsMax = 256 * 1024 * 1024;
```

```
// RGB to grayscale conversion coefficients
const float c_Coef[4] {0.2126f, 0.7152f, 0.0722f, 0.0f};

bool CompareGsImages(const uint8_t* pb_gs1,const uint8_t* pb_gs2, int num_pixels)
{
    for (int i = 0; i < num_pixels; i++)
    {
        if (abs((int)pb_gs1[i] - (int)pb_gs2[i]) > 1)
            return false;
    }

    return true;
}

bool Avx2ConvertRgbToGsCpp(uint8_t* pb_gs, const RGB32* pb_rgb, int num_pixels, const float
coef[4])
{
    if (num_pixels < c_NumPixelsMin || num_pixels > c_NumPixelsMax)
        return false;
    if (num_pixels % 8 != 0)
        return false;

    if (!AlignedMem::IsAligned(pb_gs, 32))
        return false;
    if (!AlignedMem::IsAligned(pb_rgb, 32))
        return false;

    for (int i = 0; i < num_pixels; i++)
    {
        uint8_t r = pb_rgb[i].m_R;
        uint8_t g = pb_rgb[i].m_G;
        uint8_t b = pb_rgb[i].m_B;

        float gs_temp = r * coef[0] + g * coef[1] + b * coef[2] + 0.5f;

        if (gs_temp < 0.0f)
            gs_temp = 0.0f;
        else if (gs_temp > 255.0f)
            gs_temp = 255.0f;

        pb_gs[i] = (uint8_t)gs_temp;
    }

    return true;
}

void Avx2ConvertRgbToGs(void)
{
    const wchar_t* fn_rgb = L"..\\Ch10_Data\\TestImage3.bmp";
    const wchar_t* fn_gs1 = L"Ch10_06_Avx2ConvertRgbToGs_TestImage3_GS1.bmp";
    const wchar_t* fn_gs2 = L"Ch10_06_Avx2ConvertRgbToGs_TestImage3_GS2.bmp";
```

```
    ImageMatrix im_rgb(fn_rgb);
    int im_h = im_rgb.GetHeight();
    int im_w = im_rgb.GetWidth();
    int num_pixels = im_h * im_w;
    ImageMatrix im_gs1(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_gs2(im_h, im_w, PixelType::Gray8);
    RGB32* pb_rgb = im_rgb.GetPixelBuffer<RGB32>();
    uint8_t* pb_gs1 = im_gs1.GetPixelBuffer<uint8_t>();
    uint8_t* pb_gs2 = im_gs2.GetPixelBuffer<uint8_t>();

    cout << "Results for Avx2ConvertRgbToGs\n";
    wcout << "Converting RGB image " << fn_rgb << '\n';
    cout << "  im_h = " << im_h << " pixels\n";
    cout << "  im_w = " << im_w << " pixels\n";

    // Exercise conversion functions
    bool rc1 = Avx2ConvertRgbToGsCpp(pb_gs1, pb_rgb, num_pixels, c_Coef);
    bool rc2 = Avx2ConvertRgbToGs_(pb_gs2, pb_rgb, num_pixels, c_Coef);

    if (rc1 && rc2)
    {
        wcout << "Saving grayscale image #1 - " << fn_gs1 << '\n';
        im_gs1.SaveToBitmapFile(fn_gs1);

        wcout << "Saving grayscale image #2 - " << fn_gs2 << '\n';
        im_gs2.SaveToBitmapFile(fn_gs2);

        if (CompareGsImages(pb_gs1, pb_gs2, num_pixels))
            cout << "Grayscale image compare OK\n";
        else
            cout << "Grayscale image compare failed\n";
    }
    else
        cout << "Invalid return code\n";
}

int main()
{
    try
    {
        Avx2ConvertRgbToGs();
        Avx2ConvertRgbToGs_BM();
    }

    catch (runtime_error& rte)
    {
        cout << "'runtime_error' exception has occurred - " << rte.what() << '\n';
    }
```

```
    catch (...)
    {
        cout << "Unexpected exception has occurred\n";
    }

    return 0;
}

;-------------------------------------------------
;               Ch10_06.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

                .const
GsMask          dword 0ffffffffh, 0, 0, 0, 0ffffffffh, 0, 0, 0
r4_0p5          real4 0.5
r4_255p0        real4 255.0

                extern c_NumPixelsMin:dword
                extern c_NumPixelsMax:dword

; extern "C" bool Avx2ConvertRgbToGs_(uint8_t* pb_gs, const RGB32* pb_rgb, int num_pixels,
const float coef[4])
;
; Note: Memory pointed to by pb_rgb is ordered as follows:
;       R(0,0), G(0,0), B(0,0), A(0,0), R(0,1), G(0,1), B(0,1), A(0,1), ...

        .code
Avx2ConvertRgbToGs_ proc frame
        _CreateFrame RGBGS_,0,112
        _SaveXmmRegs xmm6,xmm7,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Validate argument values
        xor eax,eax                         ;set error return code
        cmp r8d,[c_NumPixelsMin]
        jl Done                             ;jump if num_pixels < min value
        cmp r8d,[c_NumPixelsMax]
        jg Done                             ;jump if num_pixels > max value
        test r8d,7
        jnz Done                            ;jump if (num_pixels % 8) != 0

        test rcx,1fh
        jnz Done                            ;jump if pb_gs is not aligned
        test rdx,1fh
        jnz Done                            ;jump if pb_rgb is not aligned

; Perform required initializations
        vbroadcastss ymm11,real4 ptr [r4_255p0]     ;ymm11 = packed 255.0
        vbroadcastss ymm12,real4 ptr [r4_0p5]       ;ymm12 = packed 0.5
```

```
        vpxor ymm13,ymm13,ymm13                     ;ymm13 = packed zero

        vmovups xmm0,xmmword ptr [r9]
        vperm2f128 ymm14,ymm0,ymm0,00000000b        ;ymm14 = packed coef

        vmovups ymm15,ymmword ptr [GsMask]          ;ymm15 = GsMask (SPFP)

; Load next 8 RGB32 pixel values (P0 - P7)
        align 16
@@:     vmovdqa ymm0,ymmword ptr [rdx]      ;ymm0 = 8 rgb32 pixels (P7 - P0)

; Size-promote RGB32 color components from bytes to dwords
        vpunpcklbw ymm1,ymm0,ymm13
        vpunpckhbw ymm2,ymm0,ymm13
        vpunpcklwd ymm3,ymm1,ymm13          ;ymm3 = P1, P0 (dword)
        vpunpckhwd ymm4,ymm1,ymm13          ;ymm4 = P3, P2 (dword)
        vpunpcklwd ymm5,ymm2,ymm13          ;ymm5 = P5, P4 (dword)
        vpunpckhwd ymm6,ymm2,ymm13          ;ymm6 = P7, P6 (dword)

; Convert color component values to single-precision floating-point
        vcvtdq2ps ymm0,ymm3                 ;ymm0 = P1, P0 (SPFP)
        vcvtdq2ps ymm1,ymm4                 ;ymm1 = P3, P2 (SPFP)
        vcvtdq2ps ymm2,ymm5                 ;ymm2 = P5, P4 (SPFP)
        vcvtdq2ps ymm3,ymm6                 ;ymm3 = P7, P6 (SPFP)

; Multiply color component values by color conversion coefficients
        vmulps ymm0,ymm0,ymm14
        vmulps ymm1,ymm1,ymm14
        vmulps ymm2,ymm2,ymm14
        vmulps ymm3,ymm3,ymm14

; Sum weighted color components for final grayscale values
        vhaddps ymm4,ymm0,ymm0
        vhaddps ymm4,ymm4,ymm4              ;ymm4[159:128] = P1, ymm4[31:0] = P0
        vhaddps ymm5,ymm1,ymm1
        vhaddps ymm5,ymm5,ymm5              ;ymm5[159:128] = P3, ymm4[31:0] = P2
        vhaddps ymm6,ymm2,ymm2
        vhaddps ymm6,ymm6,ymm6              ;ymm6[159:128] = P5, ymm4[31:0] = P4
        vhaddps ymm7,ymm3,ymm3
        vhaddps ymm7,ymm7,ymm7              ;ymm7[159:128] = P7, ymm4[31:0] = P6

; Merge SPFP grayscale values into a single YMM register
        vandps ymm4,ymm4,ymm15              ;mask out unneeded SPFP values
        vandps ymm5,ymm5,ymm15
        vandps ymm6,ymm6,ymm15
        vandps ymm7,ymm7,ymm15
        vpslldq ymm5,ymm5,4
        vpslldq ymm6,ymm6,8
        vpslldq ymm7,ymm7,12
        vorps ymm0,ymm4,ymm5                ;merge values
        vorps ymm1,ymm6,ymm7
        vorps ymm2,ymm0,ymm1                ;ymm2 = 8 GS pixel values (SPFP)
```

```
; Add 0.5 rounding factor and clip to 0.0 - 255.0
        vaddps ymm2,ymm2,ymm12              ;add 0.5f rounding factor
        vminps ymm3,ymm2,ymm11              ;clip pixels above 255.0
        vmaxps ymm4,ymm3,ymm13              ;clip pixels below 0.0

; Convert SPFP values to bytes and save
        vcvtps2dq ymm3,ymm2                 ;convert GS SPFP to dwords
        vpackusdw ymm4,ymm3,ymm13           ;convert GS dwords to words
        vpackuswb ymm5,ymm4,ymm13           ;convert GS words to bytes

        vperm2i128 ymm6,ymm13,ymm5,3        ;xmm5 = GS P3:P0, xmm6 = GS P7:P4

        vmovd dword ptr [rcx],xmm5          ;save P3 - P0
        vmovd dword ptr [rcx+4],xmm6        ;save P7 - P4

        add rdx,32                          ;update pb_rgb to next block
        add rcx,8                           ;update pb_gs to next block
        sub r8d,8                           ;num_pixels -= 8
        jnz @B                              ;repeat until done

        mov eax,1                           ;set success return code

Done:  vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret
Avx2ConvertRgbToGs_ endp
        end
```

A variety of algorithms exist to convert an RGB image into a grayscale image. One frequently-used technique calculates grayscale pixel values using a weighted of sum the RGB color components. In this source code example, RGB pixels are converted to grayscale pixels using the following equation:

$$GS(x,y) = R(x,y)W_r + G(x,y)W_g + B(x,y)W_b$$

Each RGB color component weight (or coefficient) is a floating-point number between 0.0 and 1.0, and the sum of the three component coefficients normally equals 1.0. The exact values used for the color component coefficients are usually based on published standards that reflect a multitude of visual factors including properties of the target color space, display device characteristics, and perceived image quality. If you're interested in learning more about RGB to grayscale image conversion, Appendix A contains some references that you can consult.

Source code Ch10_06 opens with the structure declaration RGB32. This structure is declared in the header file ImageMatrix.h and specifies the color component ordering scheme of each RGB pixel. The function Avx2ConvertRgbToGsCpp contains a C++ implementation of the RGB to grayscale conversion algorithm. This function uses an ordinary for loop that sweeps through the RGB32 image buffer pb_rgb and computes grayscale pixel values using the aforementioned conversion equation. Note that RGB32 element m_A is not used in any of the calculations in this example. Each calculated grayscale pixel value is adjusted by a rounding factor and clipped to [0.0, 255.0] before it is saved to the grayscale image buffer pointed to by pb_gs.

The assembly language code begins with a `.const` section that defines the necessary constants. Following its prolog, the function `Avx2ConvertRgbToGs_` performs the customary image size and buffer alignment checks. It then loads the algorithm's required packed constants into registers YMM11–YMM15. Note that register YMM14 contains a packed version of the color conversion coefficients, as illustrated in Figure 10-3. The assembly language processing loop begins with a `vmovdqa ymm0,ymmword ptr [rdx]` instruction that loads eight `RGB32` pixel values into register YMM0. The color components of these pixels are then size-promoted to doublewords using a series of `vpunpck[l|h]bw` and `vpunpck[l|h]wd` instructions. The ensuing `vcvtdq2ps` instructions convert the pixel color components from doublewords to single-precision floating-point values. Following execution of the four `vcvtdq2ps` instructions, registers YMM0–YMM3 each contain two `RGB32` pixels and each color component is a single-precision floating-point value. Figure 10-3 also shows the `RGB32` size promotions and conversions discussed in this paragraph.

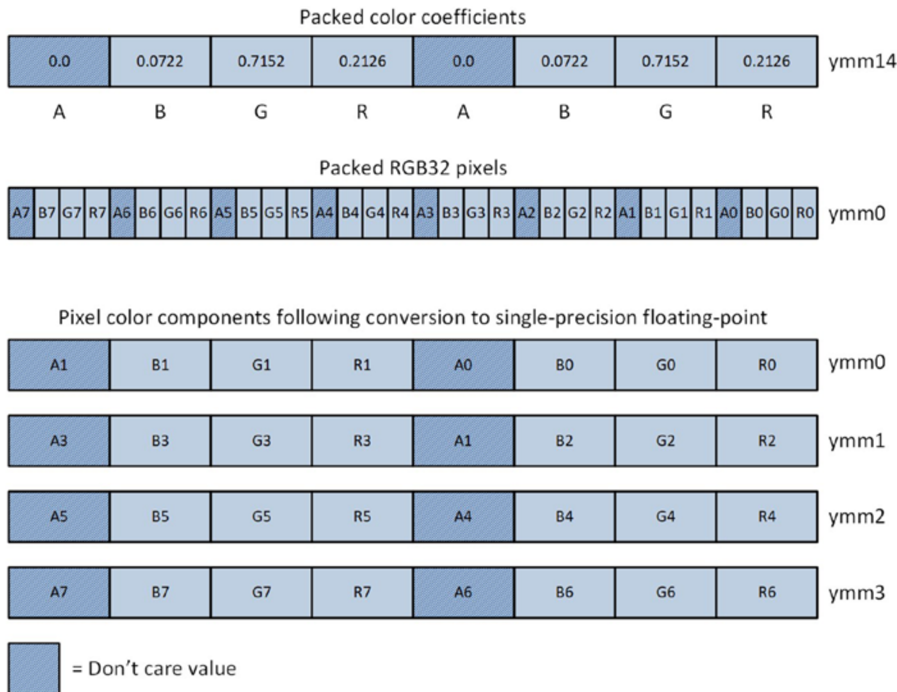



***Figure 10-3.*** *RGB32 pixel color component size promotions and conversions*

The four `vmulps` instructions multiply the eight `RGB32` pixels by the color conversion coefficients. The ensuing `vhaddps` instructions sum the weighted color components of each pixel to generate the required grayscale values. Following execution of these instructions, registers YMM4–YMM7 each contain two single-precision floating-point grayscale pixel values, one in element position [31:0] and the another in [159:128], as shown in Figure 10-4. The eight grayscale values in YMM4–YMM7 are then merged into YMM2 using a series of `vandps`, `vpslldq`, and `vorps` instructions. Figure 10-4 also shows the final merged result. The `vaddps`, `vminps`, and `vmaxps` instructions that follow add in the rounding factor (0.5) and clip the grayscale pixels to [0.0, 255.0]. These values are then converted to unsigned bytes using the instructions `vcvtps2dq`, `vpackusdw`, and `vpackuswb`. The two `vmovd` instructions save the four unsigned byte pixel values in both XMM5[31:0] and XMM6[31:0] to the grayscale image buffer.

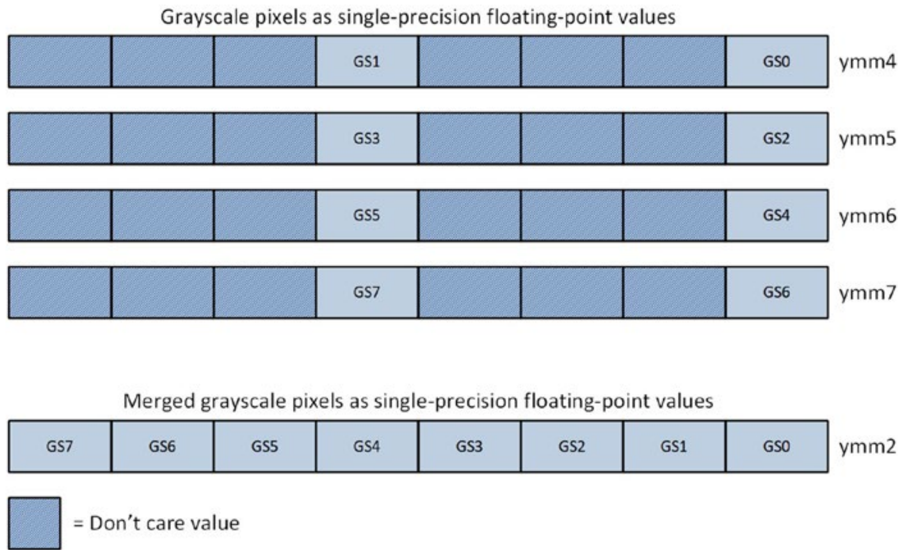



***Figure 10-4.*** *Grayscale single-precision floating-point pixel values before and after merging*

Here are the results of source code example Ch10_06:

```
Results for Avx2ConvertRgbToGs

Converting RGB image ..\Ch10_Data\TestImage3.bmp
  im_h = 960 pixels
  im_w = 640 pixels
Saving grayscale image #1 - Ch10_06_Avx2ConvertRgbToGs_TestImage3_GS1.bmp
Saving grayscale image #2 - Ch10_06_Avx2ConvertRgbToGs_TestImage3_GS2.bmp
Grayscale image compare OK

Running benchmark function Avx2ConvertRgbToGs_BM - please wait
Benchmark times save to file Ch10_06_Avx2ConvertRgbToGs_BM_CHROMIUM.csv
```

Table 10-2 shows the benchmark timing measurements for the RGB to grayscale image conversion functions Avx2ConvertRgbToGsCpp and Avx2ConvertRgbToGs_. The performance gains of this source code example are modest compared to some of the other examples in this book. The reason for this is that the RGB32 color components in the source image buffer are interleaved with each other, which necessitates the use of slower horizontal arithmetic. Rearranging the RGB32 data so that the pixels of each color component reside in separate image buffers often results in significantly faster performance. You see an example of this in Chapter 14.

***Table 10-2.** Mean Execution Times (Microseconds) for RGB to Grayscale Image Conversion Using* `TestImage3.bmp`

| CPU | Avx2ConvertRgbToGsCpp | Avx2ConvertRgbToGs_ |
|---|---|---|
| i7-4790S | 1504 | 843 |
| i9-7900X | 1075 | 593 |
| i7-8700K | 1031 | 565 |

# Summary

Here are the key learning points of Chapter 10:

- AVX2 extends the packed integer capabilities of AVX. Most x86-AVX packed integer instructions can be used with either 128-bit or 256-bit wide operands. These operands should always be properly aligned whenever possible.
- Similar to x86-AVX floating-point, assembly language functions that perform packed integer calculations using a YMM register should use a `vzeroupper` instruction prior any epilog code or the `ret` instruction. This avoids potential performance delays that can occur when the processor transitions from executing x86-AVX instructions to x86-SSE instructions.
- The Visual C++ calling convention differs for assembly language functions that return a structure by value. A function that returns a structure by value must copy a large structure (one greater than eight bytes) to the buffer pointed to by the RCX register. The normal calling convention registers are also “right-shifted” as explained in this chapter.
- Assembly language functions can use the `vpunpckl[bw|wd|dq]` and `vpunpckh[bw|wd|dq]` instructions to unpack 128-bit or 256-bit wide integer operands.
- Assembly language functions can use the `vpackss[dw|wb]` and `vpackus[dw|wb]` instructions to pack 128-bit or 256-bit wide integer operands using signed or unsigned saturation.
- Assembly language functions can use the `vmovzx[bw|bd|bq|wd|wq|dq]` and `vmovsx[bw|bd|bq|wd|wq|dq]` instructions to perform zero or sign extended packed integer size promotions.
- MASM supports directives that can perform rudimentary string processing operations, which can be employed to construct text strings for macro instruction mnemonics, operands, and labels. MASM also supports conditional error directives that can be used to signal error conditions during source code assembly.

CHAPTER 11

# AVX2 Programming – Extended Instructions

In this chapter, you learn how to use some of the instruction set extensions that were introduced in Chapter 8. The first section contains a couple of source code examples that exemplify use of the scalar and packed fused-multiply-add (FMA) instructions. The second section covers instructions that involve the general-purpose registers. This section includes source code examples that explain flagless multiplication and bit shifting. It also surveys some of the enhanced bit-manipulation instructions. The final section discusses the instructions that perform half-precision floating-point conversions.

The source code examples in the first two sections of this chapter execute correctly on most processors from AMD and Intel that that support AVX2. The half-precision floating-point source code example works on AMD and Intel processors that support AVX and the F16C instruction set extension. As a reminder, you should never assume that a specific instruction set extension is available based on whether the processor supports AVX or AVX2. Production code should always test for a specific instruction set extension using the `cupid` instruction. You learn how to do this in Chapter 16.

## FMA Programming

A FMA calculation performs a floating-point multiplication followed by a floating-point addition using a single rounding operation. Chapter 8 introduced FMA operations and discusses the particulars in greater detail. In this section, you learn how to use FMA instructions to implement discrete convolution functions. The section begins with a brief overview of convolution mathematics. The purpose of this overview is to explain just enough theory to understand the source code examples. This is followed by a source code example that implements a practical discrete convolution function using scalar FMA instructions. The section concludes with a source code example that exploits packed FMA instructions to accelerate the performance of a function that performs discrete convolutions.

### Convolutions

Convolution is a mathematical operation that blends an input signal with a response signal to produce an output signal. Formally, the convolution of input signal $f$ and response signal $g$ is defined as follows:

$$h(t) = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

where $h$ represents the output signal. The notation $f * g$ is commonly used to denote the convolution of signals (or functions) $f$ and $g$.

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_11

Convolutions are used extensively in a wide variety of scientific and engineering applications. Many signal processing and image processing techniques are based on convolution theory. In these domains, discrete arrays of sampled data points represent the input, response, and output signals. A discrete convolution can be calculated using the following equation:

$$h[i] = \sum_{k=-M}^{M} f[i-k]g[k]$$

where $i = 0,1,\ldots,N-1$ and $M = \lfloor N_g/2 \rfloor$. In the preceding equations, $N$ denotes the number of elements in both the input and output signal arrays and $N_g$ symbolizes the size of the response signal array. All of the explanations and source code examples in this section assume that $N_g$ is an odd integer greater than or equal to three. If you examine the discrete convolution equation carefully, you will notice that each element in output signal array $h$ is computed using a relatively uncomplicated sum-of-products calculation that encompasses elements of input signal array $f$ and response signal array $g$. These types of calculations are easy to implement using FMA instructions.

In digital signal processing, many applications use smoothing operators to reduce the amount of noise that's present in a raw signal. For example, the top plot in Figure 11-1 shows a raw data signal that contains a fair amount of noise. The bottom plot in Figure 11-1 shows the same signal following the application of a smoothing operator. In this instance, the smoothing operator convolved the original raw signal with a set of discrete coefficients that approximate a Gaussian (or low-pass) filter. These coefficients correspond to the response signal array $g$ that's incorporated in the discrete convolution equation. The response signal array is often called a *convolution kernel* or *convolution mask*.

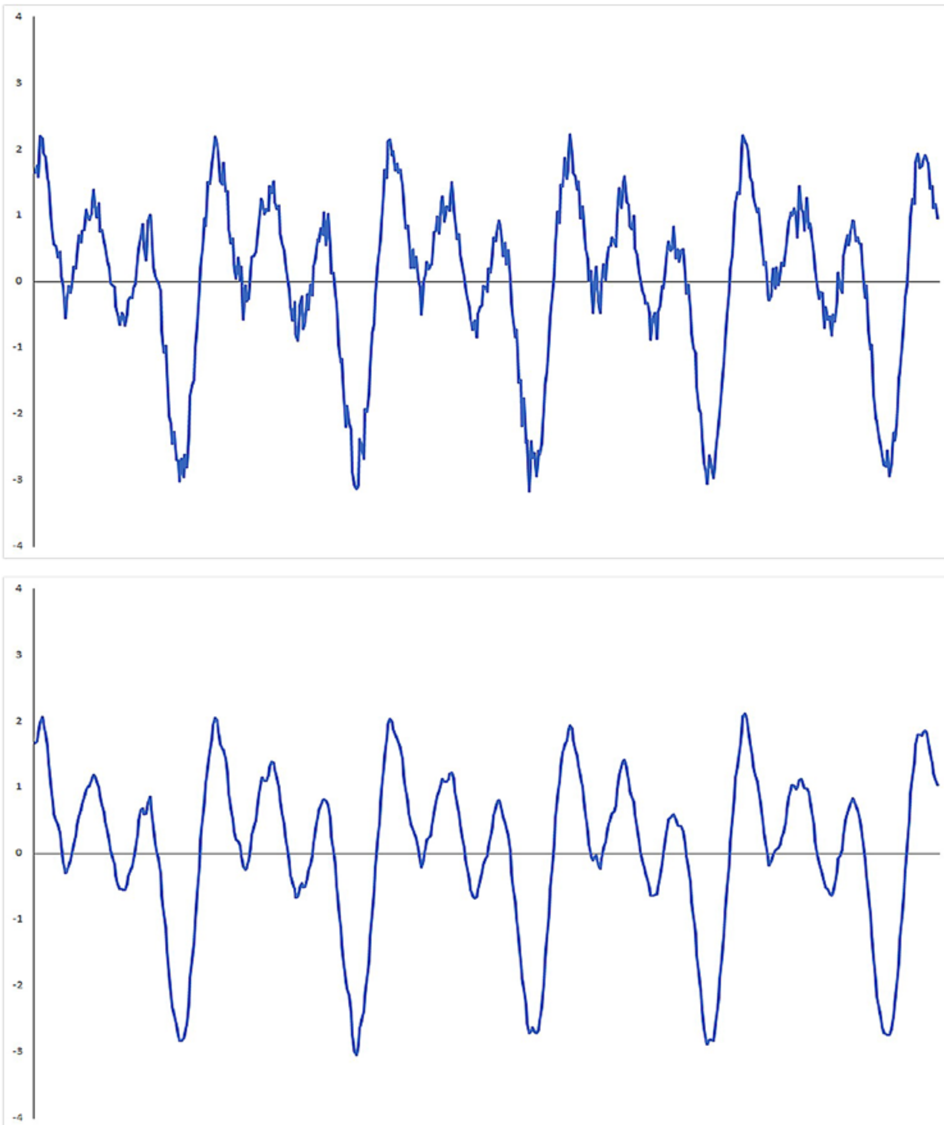



***Figure 11-1.*** *Raw data signal (top plot) and its smoothed counterpart (bottom plot)*

The discrete convolution equation can be implemented in source code using a couple of nested `for` loops. During each outer loop iteration, the convolution kernel center point `g[0]` is superimposed over the current input signal array element `f[i]`. The inner loop calculates the intermediate products, as shown in Figure 11-2. These intermediate products are then summed and saved to output signal array element `h[i]`, which is also shown in Figure 11-2. The FMA source code examples that are presented in this section implement convolution functions using the techniques described in this paragraph.

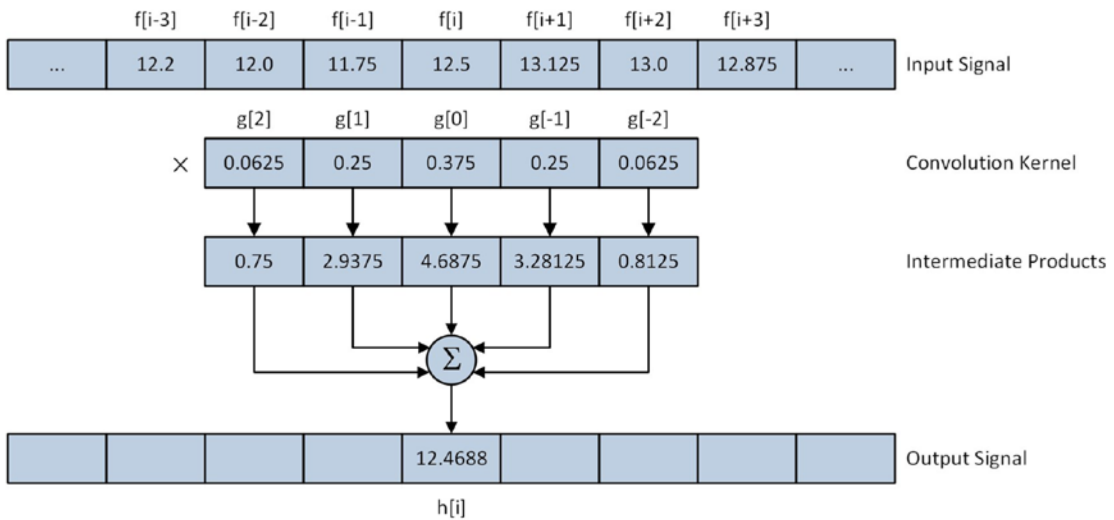



***Figure 11-2.*** *Application of a smoothing operator with an input signal element*

The purpose of the preceding overview was to provide just enough background math to understand the source code examples. Numerous tomes have been published that explain convolution and signal processing theory in significantly greater detail. Appendix A contains a list of introductory references that you can consult for additional information about convolution and signal processing theory.

## Scalar FMA

Source code example Ch11_01 explains how to implement a one-dimensional discrete convolution function using scalar FMA instructions. It also elucidates the performance benefits of convolution functions that use fixed-size versus variable-sized convolution kernels. Listing 11-1 shows the source code for example Ch11_01.

***Listing 11-1.*** Example Ch11_01

```
//------------------------------------------------
//               Ch11_01.h
//------------------------------------------------

#pragma once

// Ch11_01_Misc.cpp
extern void CreateSignal(float* x, int n, int kernel_size, unsigned int seed);
extern void PadSignal(float* x2, int n2, const float* x1, int n1, int ks2);
extern unsigned int g_RngSeedVal;

// Ch11_01.cpp
extern bool Convolve1Cpp(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size);
extern bool Convolve1Ks5Cpp(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size);
```

```
// Ch11_01_.asm
extern "C" bool Convolve1_(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size);
extern "C" bool Convolve1Ks5_(float* y, const float* x, int num_pts, const float* kernel,
int kernel_size);

// Ch11_01_BM.cpp
extern void Convolve1_BM(void);

//------------------------------------------------
//               Ch11_01_Misc.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <random>
#define _USE_MATH_DEFINES
#include <math.h>
#include "Ch11_01.h"

using namespace std;

void CreateSignal(float* x, int n, int kernel_size, unsigned int seed)
{
    const float degtorad = (float)(M_PI / 180.0);
    const float t_start = 0;
    const float t_step = 0.002f;
    const int m = 3;
    const float amp[m] {1.0f, 0.80f, 1.20f};
    const float freq[m] {5.0f, 10.0f, 15.0f};
    const float phase[m] {0.0f, 45.0f, 90.0f};
    const int ks2 = kernel_size / 2;

    uniform_int_distribution<> ui_dist {0, 500};
    default_random_engine rng {seed};
    float t = t_start;

    for (int i = 0; i < n; i++, t += t_step)
    {
        float x_val = 0;

        for (int j = 0; j < m; j++)
        {
            float omega = 2.0f * (float)M_PI * freq[j];
            float x_temp1 = amp[j] * sin(omega * t + phase[j] * degtorad);
            int rand_val = ui_dist(rng);
            float noise = (float)((rand_val) - 250) / 10.0f;
            float x_temp2 = x_temp1 + x_temp1 * noise / 100.0f;

            x_val += x_temp2;
        }
```

```
        x[i] = x_val;
    }
}

extern void PadSignal(float* x2, int n2, const float* x1, int n1, int ks2)
{
    if (n2 != n1 + ks2 * 2)
        throw runtime_error("InitPad - invalid size argument");

    for (int i = 0; i < n1; i++)
        x2[i + ks2] = x1[i];

    for (int i = 0; i < ks2; i++)
    {
        x2[i] = x1[ks2 - i - 1];
        x2[n1 + ks2 + i] = x1[n1 - i - 1];
    }
}

//------------------------------------------------
//               Ch11_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <memory>
#include <fstream>
#include <stdexcept>
#include "Ch11_01.h"

using namespace std;

extern "C" const int c_NumPtsMin = 32;
extern "C" const int c_NumPtsMax = 16 * 1024 * 1024;
extern "C" const int c_KernelSizeMin = 3;
extern "C" const int c_KernelSizeMax = 15;
unsigned int g_RngSeedVal = 97;

void Convolve1(void)
{
    const int n1 = 512;
    const float kernel[] { 0.0625f, 0.25f, 0.375f, 0.25f, 0.0625f };
    const int ks = sizeof(kernel) / sizeof(float);
    const int ks2 = ks / 2;
    const int n2 = n1 + ks2 * 2;

    // Create signal array
    unique_ptr<float[]> x1_up {new float[n1]};
    unique_ptr<float[]> x2_up {new float[n2]};
    float* x1 = x1_up.get();
```

```
    float* x2 = x2_up.get();

    CreateSignal(x1, n1, ks, g_RngSeedVal);
    PadSignal(x2, n2, x1, n1, ks2);

    // Perform convolutions
    const int num_pts = n1;
    unique_ptr<float[]> y1_up {new float[num_pts]};
    unique_ptr<float[]> y2_up {new float[num_pts]};
    unique_ptr<float[]> y3_up {new float[num_pts]};
    unique_ptr<float[]> y4_up {new float[num_pts]};
    float* y1 = y1_up.get();
    float* y2 = y2_up.get();
    float* y3 = y3_up.get();
    float* y4 = y4_up.get();

    bool rc1 = Convolve1Cpp(y1, x2, num_pts, kernel, ks);
    bool rc2 = Convolve1_(y2, x2, num_pts, kernel, ks);
    bool rc3 = Convolve1Ks5Cpp(y3, x2, num_pts, kernel, ks);
    bool rc4 = Convolve1Ks5_(y4, x2, num_pts, kernel, ks);

    cout << "Results for Convolve1\n";
    cout << "  rc1 = " << boolalpha << rc1 << '\n';
    cout << "  rc2 = " << boolalpha << rc2 << '\n';
    cout << "  rc3 = " << boolalpha << rc3 << '\n';
    cout << "  rc4 = " << boolalpha << rc4 << '\n';

    if (!rc1 || !rc2 || !rc3 || !rc4)
        return;

    // Save data
    const char* fn = "Ch11_01_Convolve1Results.csv";
    ofstream ofs(fn);

    if (ofs.bad())
        cout << "File create error - " << fn << '\n';
    else
    {
        const char* delim = ", ";

        ofs << fixed << setprecision(7);
        ofs << "i, x1, y1, y2, y3, y4\n";

        for (int i = 0; i < num_pts; i++)
        {
            ofs << setw(5) << i << delim;
            ofs << setw(10) << x1[i] << delim;
            ofs << setw(10) << y1[i] << delim;
            ofs << setw(10) << y2[i] << delim;
            ofs << setw(10) << y3[i] << delim;
            ofs << setw(10) << y4[i] << '\n';
```

```
        }

        ofs.close();
        cout << "\nConvolution results saved to file " << fn << '\n';
    }
}

bool Convolve1Cpp(float* y, const float* x, int num_pts, const float* kernel, int kernel_
size)
{
    int ks2 = kernel_size / 2;

    if ((kernel_size & 1) == 0)
        return false;

    if (kernel_size < c_KernelSizeMin || kernel_size > c_KernelSizeMax)
        return false;

    if (num_pts < c_NumPtsMin || num_pts > c_NumPtsMax)
        return false;

    x += ks2;   // x points to first signal point

    for (int i = 0; i < num_pts; i++)
    {
        float sum = 0;

        for (int k = -ks2; k <= ks2; k++)
        {
            float x_val = x[i - k];
            float kernel_val = kernel[k + ks2];

            sum += kernel_val * x_val;
        }

        y[i] = sum;
    }

    return true;
}

bool Convolve1Ks5Cpp(float* y, const float* x, int num_pts, const float* kernel, int kernel_
size)
{
    int ks2 = kernel_size / 2;

    if (kernel_size != 5)
        return false;
```

```
    if (num_pts < c_NumPtsMin || num_pts > c_NumPtsMax)
        return false;

    x += ks2;   // x points to first signal point

    for (int i = 0; i < num_pts; i++)
    {
        float sum = 0;
        int j = i + ks2;

        sum += x[j] * kernel[0];
        sum += x[j - 1] * kernel[1];
        sum += x[j - 2] * kernel[2];
        sum += x[j - 3] * kernel[3];
        sum += x[j - 4] * kernel[4];

        y[i] = sum;
    }

    return true;
}

int main()
{
    int ret_val = 1;

    try
    {
        Convolve1();
        Convolve1_BM();
        ret_val = 0;
    }

    catch (runtime_error& rte)
    {
        cout << "run_time exception has occurred\n";
        cout << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Unexpected exception has occurred\n";
    }

    return ret_val;
}
```

```
;-------------------------------------------------
;               Ch11_01_.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword

; extern "C" bool Convolve1_(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size)

        .code
Convolve1_ proc frame
        _CreateFrame CV_,0,0,rbx,rsi
        _EndProlog

; Verify argument values
        xor eax,eax                         ;set error code (rax is also loop index var)

        mov r10d,dword ptr [rbp+CV_OffsetStackArgs]
        test r10d,1
        jz Done                             ;jump if kernel_size is even
        cmp r10d,[c_KernelSizeMin]
        jl Done                             ;jump if kernel_size too small
        cmp r10d,[c_KernelSizeMax]
        jg Done                             ;jump if kernel_size too big

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;jump if num_pts too big

; Perform required initializations
        mov r8d,r8d                         ;r8 = num_pts
        shr r10d,1                          ;ks2 = ks / 2
        lea rdx,[rdx+r10*4]                 ;rdx = x + ks2 (first data point)

; Perform convolution
LP1:    vxorps xmm5,xmm5,xmm5               ;sum = 0.0;
        mov r11,r10
        neg r11                             ;k = -ks2

LP2:    mov rbx,rax
        sub rbx,r11                         ;rbx = i - k
        vmovss xmm0,real4 ptr [rdx+rbx*4]   ;xmm0 = x[i - k]
        mov rsi,r11
        add rsi,r10                         ;rsi = k + ks2
        vfmadd231ss xmm5,xmm0,[r9+rsi*4]    ;sum += x[i - k] * kernel[k + ks2]
```

```
        add r11,1                           ;k++
        cmp r11,r10
        jle LP2                             ;jump if k <= ks2

        vmovss real4 ptr [rcx+rax*4],xmm5   ;y[i] = sum

        add rax,1                           ;i += 1
        cmp rax,r8
        jl LP1                              ;jump if i < num_pts

        mov eax,1                           ;set success return code

Done:   vzeroupper
        _DeleteFrame rbx,rsi
        ret
Convolve1_ endp

; extern "C" bool Convolve1Ks5_(float* y, const float* x, int num_pts, const float* kernel,
int kernel_size)

Convolve1Ks5_ proc
; Verify argument values
        xor eax,eax                         ;set error code (rax is also loop index var)

        cmp dword ptr [rsp+40],5
        jne Done                            ;jump if kernel_size is not 5

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;jump if num_pts too big

; Perform required initializations
        mov r8d,r8d                         ;r8 = num_pts
        add rdx,8                           ;x += 2

; Perform convolution
@@:     vxorps xmm4,xmm4,xmm4                   ;initialize sum vars
        vxorps xmm5,xmm5,xmm5
        mov r11,rax
        add r11,2                               ;j = i + ks2

        vmovss xmm0,real4 ptr [rdx+r11*4]       ;xmm0 = x[j]
        vfmadd231ss xmm4,xmm0,[r9]              ;xmm4 += x[j] * kernel[0]

        vmovss xmm1,real4 ptr [rdx+r11*4-4]     ;xmm1 = x[j - 1]
        vfmadd231ss xmm5,xmm1,[r9+4]            ;xmm5 += x[j - 1] * kernel[1]

        vmovss xmm0,real4 ptr [rdx+r11*4-8]     ;xmm0 = x[j - 2]
        vfmadd231ss xmm4,xmm0,[r9+8]            ;xmm4 += x[j - 2] * kernel[2]
```

```
        vmovss xmm1,real4 ptr [rdx+r11*4-12]    ;xmm1 = x[j - 3]
        vfmadd231ss xmm5,xmm1,[r9+12]           ;xmm5 += x[j - 3] * kernel[3]

        vmovss xmm0,real4 ptr [rdx+r11*4-16]    ;xmm0 = x[j - 4]
        vfmadd231ss xmm4,xmm0,[r9+16]           ;xmm4 += x[j - 4] * kernel[4]

        vaddps xmm4,xmm4,xmm5
        vmovss real4 ptr [rcx+rax*4],xmm4       ;save y[i]

        inc rax                                 ;i += 1
        cmp rax,r8
        jl @B                                   ;jump if i < num_pts

        mov eax,1                               ;set success return code

Done:   vzeroupper
        ret
Convolve1Ks5_ endp
        end
```

The C++ code in Listing 11-1 begins with the header file `Ch11_01.h`, which contains the requisite function declarations for this example. The source code for function `CreateSignal` is next. This function constructs a synthetic input signal for test purposes. The synthetic input signal consists of three separate sinusoidal waveforms that are summed. Each waveform includes a small amount of random noise. The input signal generated by `CreateSignal` is the same signal that's shown in the top plot of Figure 11-1.

When performing convolutions, it is often necessary to pad the input signal array with extra elements to avoid invalid memory accesses when the center point of the convolution kernel is superimposed over input signal array elements located near the beginning and end of the array. The function `PadSignal` creates a padded copy of input signal array `x1` by reflecting the edge elements of `x1` and saving these elements along with the original input signal array elements in `x2`. Figure 11-3 shows an example of a padded input signal array that's compatible with a five-element convolution kernel. Note that `n2`, the size of the padded buffer, must equal `n1 + ks2 * 2`, where `n1` represents the number of input signal array elements in `x1` and `ks2` corresponds to `floor(kernel_size / 2)`.

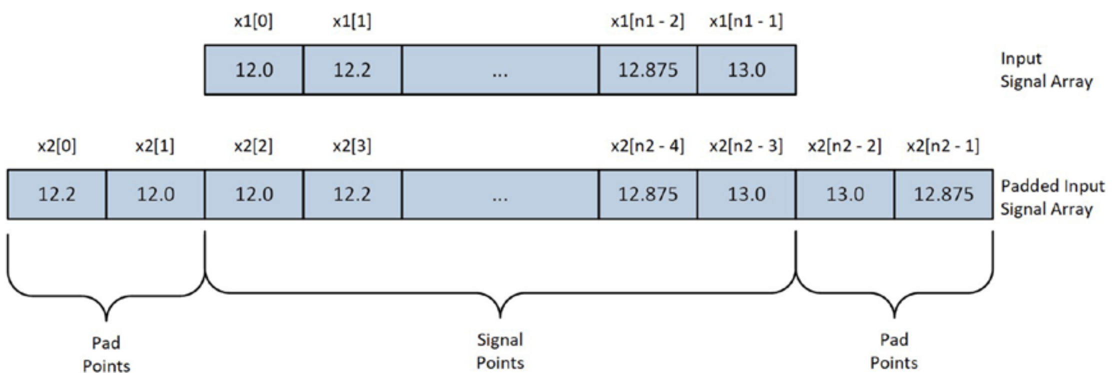



***Figure 11-3.*** *Padded input signal array following execution of* `PadSignal` *using a five-element convolution kernel*

The C++ function Convolve1 contains code that exercises several different implementations of the discrete convolution algorithm. Near the top of this function is a single-precision floating-point array named kernel, which contains the convolution kernel coefficients. The coefficients in kernel represent a discrete approximation of a Gaussian (or low pass) filter. When convolved with an input signal array, these coefficients reduce the amount of noise that's present in a signal, as shown in the bottom plot of Figure 11-1. The padded input signal array x2 is created next using the previously described functions CreateSignal and PadSignal. Note that the C++ template class unique_ptr<> is used to manage storage space for both x2 and the unpadded array x1.

Following the generation of the input signal array x2, Convolve1 allocates storage space for four output signal arrays. The functions that implement different variations of the convolution algorithms are then called. The first two functions, Convolve1Cpp and Convolve1_, contain C++ and assembly language code that carry out their convolutions using the nested for loop technique described earlier in this section. The functions Convolve1Ks5Cpp and Convolve1Ks5_ are optimized for convolution kernels containing five elements. Real-world signal processing software regularly employs convolution functions that are optimized for specific kernel sizes since they're often significantly faster, as you will soon see.

The function Convolve1Cpp begins its execution by validating argument values kernel_size and num_pts. The next statement, x += ks2, adjusts the input signal array pointer so that it points to the first true input signal array element. Recall that the input signal array x is padded with extra values to ensure correct processing when the convolution kernel is superimposed over the first two and last two input signal elements. Following the pointer x adjustment is the actual code that performs the convolution. The nested for loops implement the discrete convolution equation that was described earlier in this section. Note that the index value used for kernel is offset by ks2 to account for the negative indices of the inner loop. Following function Convolve1Cpp is the function Convolve1Ks5Cpp, which uses explicit C++ statements to calculate convolution sum-of-products instead of a for loop.

The functions Convolve1_ and Convolve1Ks5_ are the assembly language counterparts of Convolve1Cpp and Convolve1Ks5Cpp, respectively. After its prolog, Convolve1_ validates argument values kernel_size and num_pts. This is followed by an initialization code block that begins by loading num_pts into R8. The ensuing shr r10d,1 instruction loads ks2 into register R10D. The final initialization code block instruction, lea rdx,[rdx+r10*4], loads register RDX with the address of the first input signal array element in x.

Similar to the C++ code, Convole1_ uses two nested for loops to perform the convolution. The outer loop, which is labeled LP1, starts with a vxorps xmm5,xmm5,xmm5 instruction that sets sum equal to 0.0. The ensuing mov r11,r10 and neg r11 instructions set inner loop index counter k (R11) to -ks2. The label LP2 marks the start of the inner loop. The mov rbx,rax and sub rbx,r11 instructions calculate the index, or i - k, of the next element in x. This is followed by a vmovss xmm0,real4 ptr [rdx+rbx*4] instruction that loads x[i - k] into XMM0. Next, the mov rsi,r11 and add rsi,r10 instructions calculate k + ks2. The subsequent vfmadd231ss xmm5,xmm0,[r9+rsi*4] instruction calculates sum += x[i - k] * kernel[k + ks2]. As discussed in Chapter 8, the FMA instruction vfmadd231ss carries out its operations using a single rounding operation. Depending on the algorithm, the use of a vfmadd231ss instruction instead of an equivalent sequence of vmulss and vaddss instructions may result in slightly faster execution times.

Following the execution of the vfmadd231ss instruction, the add r11,1 instruction computes k++ and the inner loop repeats until k > ks2 is true. Subsequent to the completion of the inner loop, the vmovss real4 ptr [rcx+rax*4],xmm5 instruction saves the current sum-of-products result to y[i]. The add rax,1 instruction updates index counter i, and the outer loop LP1 repeats until all of the input signal data points have been processed.

The assembly language function Convolve1Ks5_ is size-optimized for convolution kernels that contain five elements. This function replaces the inner loop that was used in Convolve1_ with five explicit vfmadd231ss instructions. Note that this FMA instruction sequence employs two separate registers, XMM4 and XMM5, for the intermediate sums. Most Intel processors that support AVX2 and FMA can execute two scalar FMA instructions simultaneously, which accelerates the algorithm's overall performance. Using a single sum register here would create a performance-degrading data dependency since each vfmadd231ss

instruction would need to finish its operation before the next one could begin. You'll learn more about instruction-level data dependencies and FMA execution units in Chapter 15. Here is the output for source code example Ch11-01:

```
Results for Convolve1

  rc1 = true
  rc2 = true
  rc3 = true
  rc4 = true

Convolution results saved to file Ch11_01_Convolve1Results.csv

Running benchmark function Convolve1_BM - please wait
Benchmark times save to file Ch11_01_Convolve1_BM_CHROMIUM.csv
```

Table 11-1 contains mean execution times for the convolution functions presented in this section. As alluded to earlier, the size-optimized convolution functions Convolve1Ks5Cpp and Convolve1Ks5_ are considerably faster than their size-independent counterparts. Note that the performance of the C++ function Convolve1Cpp is somewhat better than its assembly language equivalent Convolve1_. The reason for this is that the Visual C++ complier generated code that partially unrolled the inner loop and replaced it with a series of sequential scalar single-precision floating-point multiply and add instructions. I could have easily implemented the same optimization technique in the function Convolve1_, but this improves performance only by a few percentage points. In order to achieve maximum FMA performance, an assembly language convolution function must use packed FMA instructions instead of scalar ones. You'll see an example of this in the next section.

***Table 11-1.*** *Mean Execution Times (Microseconds) for Convolution Functions Using Five-Element Convolution Kernel (2,000,000 Signal Points)*

| CPU | Convolve1Cpp | Convolve1_ | Convolve1Ks5Cpp | Convolve1Ks5_ |
|---|---|---|---|---|
| i7-4790S | 6148 | 6844 | 2926 | 2841 |
| i9-7900X | 5607 | 6072 | 2808 | 2587 |
| i7-8700K | 5149 | 5576 | 2539 | 2394 |

## Packed FMA

It's okay to use the convolution functions discussed in the previous section with small signal arrays. In many real-world applications, however, convolutions are often performed using signal arrays that contain thousands or millions of data points. For large signal arrays, the basic convolution algorithm can be adapted to use packed FMA instead of scalar FMA instructions to carry out the required calculations. Listing 11-2 shows the source code for example Ch11_02, which illustrates how to implement the discrete convolution equation using packed FMA instructions.

***Listing 11-2.*** Example Ch11_02

```
//------------------------------------------------
//               Ch11_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <fstream>
#include "Ch11_02.h"
#include "AlignedMem.h"

using namespace std;

extern "C" const int c_NumPtsMin = 32;
extern "C" const int c_NumPtsMax = 16 * 1024 * 1024;
extern "C" const int c_KernelSizeMin = 3;
extern "C" const int c_KernelSizeMax = 15;
unsigned int g_RngSeedVal = 97;

void Convolve2(void)
{
    const int n1 = 512;
    const float kernel[] { 0.0625f, 0.25f, 0.375f, 0.25f, 0.0625f };
    const int ks = sizeof(kernel) / sizeof(float);
    const int ks2 = ks / 2;
    const int n2 = n1 + ks2 * 2;
    const unsigned int alignment = 32;

    // Create signal array
    AlignedArray<float> x1_aa(n1, alignment);
    AlignedArray<float> x2_aa(n2, alignment);
    float* x1 = x1_aa.Data();
    float* x2 = x2_aa.Data();

    CreateSignal(x1, n1, ks, g_RngSeedVal);
    PadSignal(x2, n2, x1, n1, ks2);

    // Perform convolutions
    AlignedArray<float> y5_aa(n1, alignment);
    AlignedArray<float> y6_aa(n1, alignment);
    AlignedArray<float> y7_aa(n1, alignment);
    float* y5 = y5_aa.Data();
    float* y6 = y6_aa.Data();
    float* y7 = y7_aa.Data();

    bool rc5 = Convolve2_(y5, x2, n1, kernel, ks);
    bool rc6 = Convolve2Ks5_(y6, x2, n1, kernel, ks);
    bool rc7 = Convolve2Ks5Test_(y7, x2, n1, kernel, ks);
```

```
    cout << "Results for Convolve2\n";
    cout << "  rc5 = " << boolalpha << rc5 << '\n';
    cout << "  rc6 = " << boolalpha << rc6 << '\n';
    cout << "  rc7 = " << boolalpha << rc7 << '\n';

    if (!rc5 || !rc6 || !rc7)
        return;

    // Save data
    const char* fn = "Ch11_02_Convolve2Results.csv";
    ofstream ofs(fn);

    if (ofs.bad())
        cout << "File create error - " << fn << '\n';
    else
    {
        const char* delim = ", ";

        ofs << fixed << setprecision(7);
        ofs << "i, x1, y5, y6, y7\n";

        for (int i = 0; i < n1; i++)
        {
            ofs << setw(5) << i << delim;
            ofs << setw(10) << x1[i] << delim;
            ofs << setw(10) << y5[i] << delim;
            ofs << setw(10) << y6[i] << delim;
            ofs << setw(10) << y7[i];

            if (y6[i] != y7[i])
                ofs << delim << '*';

            ofs << '\n';
        }

        ofs.close();
        cout << "\nResults data saved to file " << fn << '\n';
    }
}

int main()
{
    int ret_val = 1;

    try
    {
        Convolve2();
        Convolve2_BM();
        ret_val = 0;
    }
```

```
    catch (runtime_error& rte)
    {
        cout << "run_time exception has occurred\n";
        cout << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Unexpected exception has occurred\n";
    }

    return ret_val;
}

;-------------------------------------------------
;               Ch11_02_.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword

; extern bool Convolve2_(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size)

        .code
Convolve2_ proc frame
        _CreateFrame CV2_,0,0,rbx
        _EndProlog

; Validate argument values
        xor eax,eax                         ;set error code

        mov r10d,dword ptr [rbp+CV2_OffsetStackArgs]
        test r10d,1
        jz Done                             ;kernel_size is even
        cmp r10d,[c_KernelSizeMin]
        jl Done                             ;kernel_size too small
        cmp r10d,[c_KernelSizeMax]
        jg Done                             ;kernel_size too big

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;num_pts too big
        test r8d,7
        jnz Done                            ;num_pts not even multiple of 8

        test rcx,1fh
        jnz Done                            ;y is not properly aligned
```

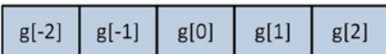
```
; Initialize convolution loop variables
        shr r10d,1                          ;r10 = kernel_size / 2 (ks2)
        lea rdx,[rdx+r10*4]                 ;rdx = x + ks2 (first data point)
        xor ebx,ebx                         ;i = 0

; Perform convolution
LP1:    vxorps ymm0,ymm0,ymm0               ;packed sum = 0.0;
        mov r11,r10                         ;r11 = ks2
        neg r11                             ;k = -ks2

LP2:    mov rax,rbx                             ;rax = i
        sub rax,r11                             ;rax = i - k
        vmovups ymm1,ymmword ptr [rdx+rax*4]    ;load x[i - k]:x[i - k + 7]

        mov rax,r11
        add rax,r10                             ;rax = k + ks2
        vbroadcastss ymm2,real4 ptr [r9+rax*4]  ;ymm2 = kernel[k + ks2]
        vfmadd231ps ymm0,ymm1,ymm2              ;ymm0 += x[i-k]:x[i-k+7] * kernel[k+ks2]

        add r11,1                               ;k += 1
        cmp r11,r10
        jle LP2                                 ;repeat until k > ks2

        vmovaps ymmword ptr [rcx+rbx*4],ymm0    ;save y[i]:y[i + 7]

        add rbx,8                           ;i += 8
        cmp rbx,r8
        jl LP1                              ;repeat until done
        mov eax,1                           ;set success return code

Done:   vzeroupper
        _DeleteFrame rbx
        ret
Convolve2_ endp

; extern bool Convolve2Ks5_(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size)

Convolve2Ks5_ proc frame
        _CreateFrame CKS5_,0,48
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Validate argument values
        xor eax,eax                         ;set error code (rax is also loop index var)

        cmp dword ptr [rbp+CKS5_OffsetStackArgs],5
        jne Done                            ;jump if kernel_size is not 5
```

```
        cmp r8d,[c_NumPtsMin]
        jl Done                             ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;jump if num_pts too big
        test r8d,7
        jnz Done                            ;num_pts not even multiple of 8

        test rcx,1fh
        jnz Done                            ;y is not properly aligned

; Perform required initializations
        vbroadcastss ymm4,real4 ptr [r9]        ;kernel[0]
        vbroadcastss ymm5,real4 ptr [r9+4]      ;kernel[1]
        vbroadcastss ymm6,real4 ptr [r9+8]      ;kernel[2]
        vbroadcastss ymm7,real4 ptr [r9+12]     ;kernel[3]
        vbroadcastss ymm8,real4 ptr [r9+16]     ;kernel[4]
        mov r8d,r8d                             ;r8 = num_pts
        add rdx,8                               ;x += 2

; Perform convolution
@@:     vxorps ymm2,ymm2,ymm2                   ;initialize sum vars
        vxorps ymm3,ymm3,ymm3
        mov r11,rax
        add r11,2                               ;j = i + ks2

        vmovups ymm0,ymmword ptr [rdx+r11*4]    ;ymm0 = x[j]:x[j + 7]
        vfmadd231ps ymm2,ymm0,ymm4              ;ymm2 += x[j]:x[j + 7] * kernel[0]

        vmovups ymm1,ymmword ptr [rdx+r11*4-4]  ;ymm1 = x[j - 1]:x[j + 6]
        vfmadd231ps ymm3,ymm1,ymm5              ;ymm3 += x[j - 1]:x[j + 6] * kernel[1]

        vmovups ymm0,ymmword ptr [rdx+r11*4-8]  ;ymm0 = x[j - 2]:x[j + 5]
        vfmadd231ps ymm2,ymm0,ymm6              ;ymm2 += x[j - 2]:x[j + 5] * kernel[2]

        vmovups ymm1,ymmword ptr [rdx+r11*4-12] ;ymm1 = x[j - 3]:x[j + 4]
        vfmadd231ps ymm3,ymm1,ymm7              ;ymm3 += x[j - 3]:x[j + 4] * kernel[3]

        vmovups ymm0,ymmword ptr [rdx+r11*4-16] ;ymm0 = x[j - 4]:x[j + 3]
        vfmadd231ps ymm2,ymm0,ymm8              ;ymm2 += x[j - 4]:x[j + 3] * kernel[4]

        vaddps ymm0,ymm2,ymm3                   ;final values
        vmovaps ymmword ptr [rcx+rax*4],ymm0    ;save y[i]:y[i + 7]

        add rax,8                               ;i += 8
        cmp rax,r8
        jl @B                                   ;jump if i < num_pts
        mov eax,1                               ;set success return code
```

```
Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame
        ret
Convolve2Ks5_ endp
        end
```

The convolution functions in source code example Ch11_01 used single-precision floating-point signal arrays and convolution kernels. Recall that a 256-bit wide YMM register can hold eight single-precision floating-point values, which means that a SIMD implementation of the convolution algorithm can carry out eight FMA calculations simultaneously. Figure 11-4 contains two graphics that illustrate a five-element convolution kernel along with an arbitrary segment of an input signal array. Below the graphics are the equations that can be used to convolve the eight input signal points f[i]:f[i+7] with a five-element convolution kernel. These equations are a simple expansion of the discrete convolution equation that was discussed earlier in this section. Note that each column of the SIMD convolution equation set includes a single kernel value and eight consecutive elements from the input signal array. This means that a SIMD convolution function can be easily implemented using data broadcast, packed move, and packed FMA instructions, as you'll soon see.

Convolution kernel




Input signal array

SIMD convolution equations (8 signal points)

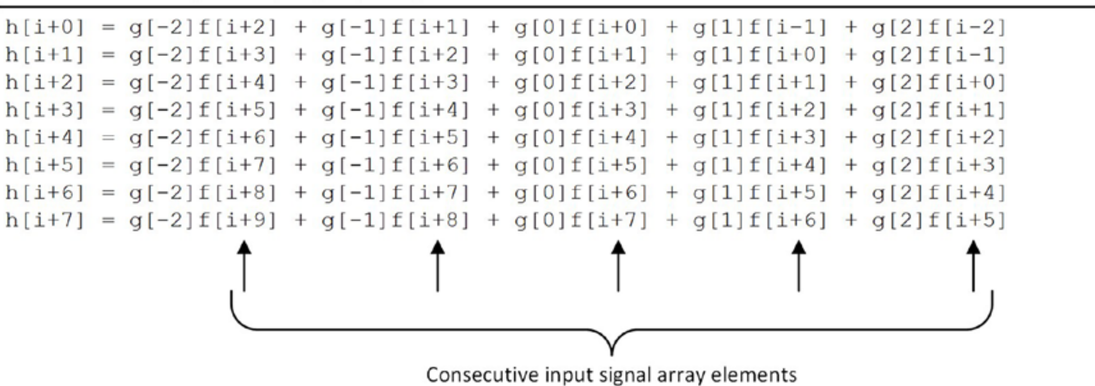



***Figure 11-4.*** *SIMD convolution equations for five-element convolution kernel*

The C++ function Convolve2 is located near the top of Listing 11-2. This function creates and initializes the padded input signal array x2 using the same CreateSignal and PadSignal functions that were employed in example Ch11_01. It also uses the C++ template class AlignedArray to allocate storage for the output signal arrays y5 and y6. In this example, the output signal arrays must be properly aligned since the assembly language functions use the vmovaps instruction to save calculated results. Proper alignment of the input signal arrays x1 and x2 is optional and used here for consistency. Following the allocation of the signal

arrays, Convolve2 invokes the assembly language functions Convolve2_ and ConvolveKs5_; it then saves the output signal array data to a CSV file.

The function Convolve2_ begins its execution by validating argument values kernel_size and num_pts. It also verifies that output signal array y is properly aligned. The convolution code block of Convolve2_ employs the same nested for loop construct that was used in example Ch11_01. Each outer loop LP1 iteration begins with a vxorps ymm0,ymm0,ymm0 instruction that initializes eight single-precision floating-point sum values to 0.0. The ensuing mov r11,r10 and neg r11 instructions initialize k with the value -ks2. The inner loop starts by calculating and saving i - k in register RAX. The subsequent vmovups ymm1,ymmword ptr [rdx+rax*4] instruction loads input signal array elements x[i - k]:x[i - k + 7] into register YMM1. This is followed by a vbroadcastss ymm2,real4 ptr [r9+rax*4] instruction that broadcasts kernel[k + ks2] to each single-precision floating-point element position in YMM2. The vfmadd231ps ymm0,ymm1,ymm2 instruction multiplies each input signal array element in YMM1 by kernel[k + ks2] and adds this intermediate result to the packed sums in register YMM0. The inner loop repeats until k > ks2 is true. Following the completion of the inner loop, the vmovaps ymmword ptr [rcx+rbx*4],ymm0 instruction saves the eight convolution results to output signal array elements y[i]:y[i + 7]. Outer loop index counter i is then updated and the loop repeats until all input signal elements have been processed.

The assembly language function Convolve2Ks5_ is optimized for five-element convolution kernels. Following the requisite argument validations, a series of vbroadcastss instructions load kernel coefficients kernel[0]–kernel[4] into registers YMM4–YMM8, respectively. The two vxorps instructions located at the top of the processing loop initialize the intermediate packed sums to 0.0. The array index j = i + ks2 is then calculated and saved in register R11. The ensuing vmovups ymm0,ymmword ptr [rdx+r11*4] instruction loads input signal array elements x[j]:x[j + 7] into register YMM0. This is followed by a vfmadd231ps ymm2,ymm0,ymm4 instruction that multiplies each input signal array element in YMM0 by kernel[0]; it then adds these values to the intermediate packed sums in YMM2. Convolve2Ks5_ then uses four additional sets of vmovups and vfmadd231ps instructions to compute results using coefficients kernel[1]–kernel[4]. Similar to the function Convolve1Ks5_ in source code example Ch11_01, Convolve2Ks5_ also uses two YMM registers for its intermediate FMA sums, which facilitates simultaneous execution of two vfmadd231ps instructions on processors with dual 256-bit wide FMA execution units. Following the FMA operations, a vmovaps ymmword ptr [rcx+rax*4],ymm0 instruction saves the eight output signal array elements to y[i]:y[i + 7].

The assembly language code for example Ch11_02 also includes a function named Convolve2Ks5Test_. This function replaces all occurrences of the instruction vfmadd231ps with an equivalent sequence of vmulps and vaddps instructions for benchmarking and value comparison purposes, which are discussed shortly. The source code for Convolve2Ks5Test_ is not shown in Listing 11-2 but is included with the chapter download package. Here are the results for source code example Ch11_02.

```
Results for Convolve2

  rc5 = true
  rc6 = true
  rc7 = true

Results data saved to file Ch11_02_Convolve2Results.csv

Running benchmark function Convolve2_BM - please wait
Benchmark times save to file Ch11_02_Convolve2_BM_CHROMIUM.csv
```

Table 11-2 shows the benchmark timing measurements for the SIMD implementations of the convolution functions. As expected, the SIMD versions are significantly faster than their non-SIMD counterparts (see Table 11-1). The mean execution times for the five-element convolution kernel functions Convolve2Ks5_ and Convolve2Ks5Test_ are essentially the same.

*Table 11-2. Mean Execution Times (Microseconds) for SIMD Convolution Functions Using Five-Element Convolution Kernel (2,000,000 Signal Points)*

| CPU | Convolve2_ | Convolve2Ks5_ | Convolve2Ks5Test_ |
|---|---|---|---|
| i7-4790S | 1244 | 1067 | 1074 |
| i9-7900X | 956 | 719 | 709 |
| i7-8700K | 859 | 595 | 597 |

It is not uncommon for small value discrepancies to occur between a function that uses FMA instructions and an equivalent function that uses distinct multiply and add instructions. This is confirmed by the comparing the output results of functions `Convolve2Ks5_` and `Convolve2Ks5Test_`. Table 11-3 shows a few examples of value discrepancies from the output file `Ch11_02_Convolve2Results.csv`. In a real-world application, the magnitudes of these value discrepancies would most likely be inconsequential. However, the potential for value discrepancies is something that you should always keep in mind if you're developing production code that includes both FMA and non-FMA versions of the same function, especially for functions that perform numerous FMA operations.

*Table 11-3. Examples of Value Discrepancies Using FMA and Non-FMA Instruction Sequences*

| Index | x[] | Convolve2Ks5_ | Convolve2Ks5Test_ |
|---|---|---|---|
| 33 | 1.3856432 | 1.1940877 | 1.1940879 |
| 108 | 1.3655651 | 1.4466031 | 1.4466029 |
| 180 | -2.8778596 | -2.7348523 | -2.7348526 |
| 277 | -1.7654022 | -2.0587211 | -2.0587208 |
| 403 | 2.0683382 | 2.0299273 | 2.0299270 |

# General-Purpose Register Instructions

As mentioned and discussed in Chapter 8, several general-purpose register instructions set extensions have been added to the x86 platform in recent years (see Tables 8-2 and 8-5). In this section, you learn how to use some of these instructions. The first source code example illustrates how to use the flagless multiplication and shift instructions. A flagless instruction executes its operation without modifying any of the status flags in RFLAGS, which can be faster than an equivalent flag-based instruction depending on the specific use case. The second source code example demonstrates several advanced bit-manipulation instructions. The source code examples of this section require a processor that supports the BMI1, BMI2, and LZCNT instruction set extensions.

## Flagless Multiplication and Shifts

Listing 11-3 shows the source code for example `Ch11_03`. This example demonstrates how to use the flagless unsigned integer multiplication instruction `mulx`. It also explains how to use the flagless shift instructions `sarx`, `shlx`, and `shrx`.

***Listing 11-3.*** Example Ch11_03

```
//------------------------------------------------
//               Ch11_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;

#include "stdafx.h"

extern "C" uint64_t GprMulx_(uint32_t a, uint32_t b, uint64_t flags[2]);
extern "C" void GprShiftx_(uint32_t x, uint32_t count, uint32_t results[3], uint64_t
flags[4]);

string ToString(uint64_t flags)
{
    ostringstream oss;

    oss << "OF=" << ((flags & (1ULL << 11)) ? '1' : '0') << ' ';
    oss << "SF=" << ((flags & (1ULL <<  7)) ? '1' : '0') << ' ';
    oss << "ZF=" << ((flags & (1ULL <<  6)) ? '1' : '0') << ' ';
    oss << "PF=" << ((flags & (1ULL <<  2)) ? '1' : '0') << ' ';
    oss << "CF=" << ((flags & (1ULL <<  0)) ? '1' : '0') << ' ';

    return oss.str();
}

void GprMulx(void)
{
    const int n = 3;
    uint32_t a[n] = {64, 3200, 100000000};
    uint32_t b[n] = {1001, 12, 250000000};

    cout << "\nResults for AvxGprMulx\n";

    for (int i = 0; i < n; i++)
    {
        uint64_t flags[2];
        uint64_t c = GprMulx_(a[i], b[i], flags);

        cout << "\nTest case " << i << '\n';
        cout << "  a: " << a[i] << "  b: " << b[i] << "  c: " << c << '\n';

        cout << setfill ('0') << hex;
        cout << "  status flags before mulx: " << ToString(flags[0]) << '\n';
        cout << "  status flags after mulx:  " << ToString(flags[1]) << '\n';
```

```
        cout << setfill (' ') << dec;
    }
}

void GprShiftx(void)
{
    const int n = 4;
    uint32_t x[n] = { 0x00000008, 0x80000080, 0x00000040, 0xfffffc10 };
    uint32_t count[n] = { 2, 5, 3, 4 };

    cout << "\nResults for AvxGprShiftx\n";

    for (int i = 0; i < n; i++)
    {
        uint32_t results[3];
        uint64_t flags[4];

        GprShiftx_(x[i], count[i], results, flags);

        cout << setfill(' ') << dec;
        cout << "\nTest case " << i << '\n';

        cout << setfill('0') << hex << "  x:    0x" << setw(8) << x[i] << " (";
        cout << setfill(' ') << dec << x[i] << ") count: " << count[i] << '\n';

        cout << setfill('0') << hex << "  sarx: 0x" << setw(8) << results[0] << " (";
        cout << setfill(' ') << dec << results[0] << ")\n";

        cout << setfill('0') << hex << "  shlx: 0x" << setw(8) << results[1] << " (";
        cout << setfill(' ') << dec << results[1] << ")\n";

        cout << setfill('0') << hex << "  shrx: 0x" << setw(8) << results[2] << " (";
        cout << setfill(' ') << dec << results[2] << ")\n";

        cout << "  status flags before shifts: " << ToString(flags[0]) << '\n';
        cout << "  status flags after sarx:    " << ToString(flags[1]) << '\n';
        cout << "  status flags after shlx:    " << ToString(flags[2]) << '\n';
        cout << "  status flags after shrx:    " << ToString(flags[3]) << '\n';
    }
}

int main()
{
    GprMulx();
    GprShiftx();
    return 0;
}
```

```
;-------------------------------------------------
;               Ch11_03_.asm
;-------------------------------------------------

; extern "C" uint64_t GprMulx_(uint32_t a, uint32_t b, uint64_t flags[2]);
;
; Requires      BMI2

        .code
GprMulx_ proc

; Save copy of status flags before mulx
        pushfq
        pop rax
        mov qword ptr [r8],rax              ;save original status flags

; Perform flagless multiplication. The mulx instruction below computes
; the product of explicit source operand ecx (a) and implicit source
; operand edx (b). The 64-bit result is saved to the register pair r11d:r10d.
        mulx r11d,r10d,ecx                  ;r11d:r10d = a * b

; Save copy of status flags after mulx
        pushfq
        pop rax
        mov qword ptr [r8+8],rax            ;save post mulx status flags

; Move 64-bit result to rax
        mov eax,r10d
        shl r11,32
        or rax,r11
        ret
GprMulx_ endp

; extern "C" void GprShiftx_(uint32_t x, uint32_t count, uint32_t results[3], uint64_t
flags[4])
;
; Requires      BMI2

GprShiftx_ proc

; Save copy of status flags before shifts
        pushfq
        pop rax
        mov qword ptr [r9],rax              ;save original status flags

; Load argument values and perform shifts.  Note that each shift
; instruction requires three operands: DesOp, SrcOp, and CountOp.

        sarx eax,ecx,edx                    ;shift arithmetic right
        mov dword ptr [r8],eax
        pushfq
```

```
        pop rax
        mov qword ptr [r9+8],rax

        shlx eax,ecx,edx                    ;shift logical left
        mov dword ptr [r8+4],eax
        pushfq
        pop rax
        mov qword ptr [r9+16],rax

        shrx eax,ecx,edx                    ;shift logical right
        mov dword ptr [r8+8],eax
        pushfq
        pop rax
        mov qword ptr [r9+24],rax

        ret
GprShiftx_ endp
        end
```

The C++ code for example Ch11_03 contains two functions named GprMulx and GprShiftx. These functions initialize test cases that demonstrate flagless multiplication and shift operations, respectively. Note that both GprMulx and GprShiftx define an array of type uint64_t named flags. This array is used to show the contents of the status flags in RFLAGS before and after the execution of each flagless instruction. The remaining code in both GprMulx and GprShiftx format and stream results to cout.

The assembly language function GprMulx begins its execution by saving a copy of RFLAGS. The ensuing mulx r11d,r10d,ecx instruction performs a 32-bit unsigned integer multiplication using implicit source operand EDX (argument value b) with explicit source operand ECX (argument value a). The 64-bit product is then saved in register pair R11D:R10D. Following the execution of the mulx instruction, the contents of RFLAGS are saved again for comparison purposes. The mulx instruction also supports flagless multiplications using 64-bit operands. When used with 64-bit operands, register RDX is employed as the implicit operand and two 64-bit general-purpose registers must be used for the destination operands.

The function GprShiftx_ includes examples of the sarx, shlx, and shrx instructions using 32-bit wide operands. These instructions use a three-operand syntax similar to AVX instructions. The first source operand is shifted by the count value that's specified in the second source operand. The result is then saved to the destination operand. The flagless shift instructions can also be used with 64-bit wide operands; 8- and 16-bit wide operands are not supported. Here is the output for source code example Ch11_03:

```
Results for AvxGprMulx

Test case 0
  a: 64  b: 1001  c: 64064
  status flags before mulx: OF=0 SF=0 ZF=1 PF=1 CF=0
  status flags after mulx:  OF=0 SF=0 ZF=1 PF=1 CF=0

Test case 1
  a: 3200  b: 12  c: 38400
  status flags before mulx: OF=0 SF=1 ZF=0 PF=0 CF=1
  status flags after mulx:  OF=0 SF=1 ZF=0 PF=0 CF=1
```

```
Test case 2
  a: 100000000  b: 250000000  c: 25000000000000000
  status flags before mulx: OF=0 SF=1 ZF=0 PF=1 CF=1
  status flags after mulx:  OF=0 SF=1 ZF=0 PF=1 CF=1

Results for AvxGprShiftx

Test case 0
  x:    0x00000008 (8) count: 2
  sarx: 0x00000002 (2)
  shlx: 0x00000020 (32)
  shrx: 0x00000002 (2)
  status flags before shifts: OF=0 SF=0 ZF=1 PF=1 CF=0
  status flags after sarx:    OF=0 SF=0 ZF=1 PF=1 CF=0
  status flags after shlx:    OF=0 SF=0 ZF=1 PF=1 CF=0
  status flags after shrx:    OF=0 SF=0 ZF=1 PF=1 CF=0

Test case 1
  x:    0x80000080 (2147483776) count: 5
  sarx: 0xfc000004 (4227858436)
  shlx: 0x00001000 (4096)
  shrx: 0x04000004 (67108868)
  status flags before shifts: OF=0 SF=1 ZF=0 PF=0 CF=1
  status flags after sarx:    OF=0 SF=1 ZF=0 PF=0 CF=1
  status flags after shlx:    OF=0 SF=1 ZF=0 PF=0 CF=1
  status flags after shrx:    OF=0 SF=1 ZF=0 PF=0 CF=1

Test case 2
  x:    0x00000040 (64) count: 3
  sarx: 0x00000008 (8)
  shlx: 0x00000200 (512)
  shrx: 0x00000008 (8)
  status flags before shifts: OF=0 SF=1 ZF=0 PF=0 CF=1
  status flags after sarx:    OF=0 SF=1 ZF=0 PF=0 CF=1
  status flags after shlx:    OF=0 SF=1 ZF=0 PF=0 CF=1
  status flags after shrx:    OF=0 SF=1 ZF=0 PF=0 CF=1

Test case 3
  x:    0xfffffc10 (4294966288) count: 4
  sarx: 0xffffffc1 (4294967233)
  shlx: 0xffffc100 (4294951168)
  shrx: 0x0fffffc1 (268435393)
  status flags before shifts: OF=0 SF=1 ZF=0 PF=1 CF=1
  status flags after sarx:    OF=0 SF=1 ZF=0 PF=1 CF=1
  status flags after shlx:    OF=0 SF=1 ZF=0 PF=1 CF=1
  status flags after shrx:    OF=0 SF=1 ZF=0 PF=1 CF=1
```

## Enhanced Bit Manipulation

Most of the instructions included in the BMI1 and BMI2 instruction set extensions are geared toward improving the performance of specific algorithms, such as data encryption and decryption. They also can be employed to simplify bit-manipulation operations in more mundane algorithms. Source code example Ch11_04 includes three simple assembly language functions that demonstrate use of the enhanced bit-manipulation instructions lzcnt, tzcnt, bextr, and andn. Listing 11-4 shows the C++ and assembly language source code for this example.

***Listing 11-4.*** Example Ch11_04

```
//------------------------------------------------
//               Ch11_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void GprCountZeroBits_(uint32_t x, uint32_t* lzcnt, uint32_t* tzcnt);
extern "C" uint32_t GprBextr_(uint32_t x, uint8_t start, uint8_t length);
extern "C" uint32_t GprAndNot_(uint32_t x, uint32_t y);

void GprCountZeroBits(void)
{
    const int n = 5;
    uint32_t x[n] = { 0x001000008, 0x00008000, 0x8000000, 0x00000001, 0 };

    cout << "\nResults for AvxGprCountZeroBits\n";

    for (int i = 0; i < n; i++)
    {
        uint32_t lzcnt, tzcnt;

        GprCountZeroBits_(x[i], &lzcnt, &tzcnt);

        cout << setfill('0') << hex;
        cout << "x: 0x" << setw(8) << x[i] << "  ";
        cout << setfill(' ') << dec;
        cout << "lzcnt: " << setw(3) << lzcnt << "  ";
        cout << "tzcnt: " << setw(3) << tzcnt << '\n';
    }
}

void GprExtractBitField(void)
{
    const int n = 3;
    uint32_t x[n] = { 0x12345678, 0x80808080, 0xfedcba98 };
    uint8_t start[n] = { 4, 7, 24 };
```

```
    uint8_t len[n] = { 16, 9, 8 };

    cout << "\nResults for GprExtractBitField\n";

    for (int i = 0; i < n; i++)
    {
        uint32_t bextr = GprBextr_(x[i], start[i], len[i]);

        cout << setfill('0') << hex;
        cout << "x: 0x" << setw(8) << x[i] << "  ";

        cout << setfill(' ') << dec;
        cout << "start: " << setw(3) << (uint32_t)start[i] << "  ";
        cout << "len:   " << setw(3) << (uint32_t)len[i] << "  ";
        cout << setfill('0') << hex;
        cout << "bextr: 0x" << setw(8) << bextr << '\n';
    }
}

void GprAndNot(void)
{
    const int n = 3;
    uint32_t x[n] = { 0xf000000f, 0xff00ff00, 0xaaaaaaaa };
    uint32_t y[n] = { 0x12345678, 0x12345678, 0xffaa5500 };

    cout << "\nResults for GprAndNot\n";

    for (int i = 0; i < n; i++)
    {
        uint32_t andn = GprAndNot_(x[i], y[i]);

        cout << setfill('0') << hex;
        cout << "x: 0x" << setw(8) << x[i] << "  ";
        cout << "y: 0x" << setw(8) << y[i] << "  ";
        cout << "andn: 0x" << setw(8) << andn << '\n';
    }
}

int main()
{
    GprCountZeroBits();
    GprExtractBitField();
    GprAndNot();
    return 0;
}
```

```
;-------------------------------------------------
;               Ch11_04_.asm
;-------------------------------------------------

; extern "C" void GprCountZeroBits_(uint32_t x, uint32_t* lzcnt, uint32_t* tzcnt);
;
; Requires:     BMI1, LZCNT

        .code
GprCountZeroBits_ proc
        lzcnt eax,ecx                       ;count leading zeros
        mov dword ptr [rdx],eax             ;save result

        tzcnt eax,ecx                       ;count trailing zeros
        mov dword ptr [r8],eax              ;save result
        ret
GprCountZeroBits_ endp

; extern "C" uint32_t GprBextr_(uint32_t x, uint8_t start, uint8_t length);
;
; Requires:     BMI1

GprBextr_ proc
        mov al,r8b
        mov ah,al                           ;ah = length
        mov al,dl                           ;al = start
        bextr eax,ecx,eax                   ;eax = extracted bit field (from x)
        ret
GprBextr_ endp

; extern "C" uint32_t GprAndNot_(uint32_t x, uint32_t y);
;
; Requires:     BMI1

GprAndNot_ proc
        andn eax,ecx,edx                    ;eax = ~x & y
        ret
GprAndNot_ endp
        end
```

The C++ code in Listing 11-4 contains three short functions that set up test cases for the assembly language functions. The first function, `GprCountZeroBits`, initializes a test array that's used to demonstrate the `lzcnt` (Count the Number of Leading Zero Bits) and `tzcnt` (Count the Number of Trailing Zero Bits) instructions. The second function, `GprExtractBitField`, prepares test data for the `bextr` (Bit Field Extract) instruction. The final C++ function in Listing 11-4 is named `GprAndNot`. This function loads test arrays with data that's used to illuminate execution of the `andn` (Bitwise AND NOT) instruction.

The first assembly language function is named `GprCountZeroBits_`. This function uses the `lzcnt` and `tzcnt` instructions to count the number of leading and trailing zero bits in their respective 32-bit wide source operands. The calculated bit counts are then saved in the specified destination operand. The next function, `GprBextr_`, exercises the `bextr` instruction. This instruction's first source operand contains the data from which the bit field will be extracted. Bits 7:0 and 15:8 of the second source operand specify the field's starting

bit position and length, respectively. Finally, the function `GprAndNot_` shows how to use the `andn` instruction. This instruction computes `DesOp = ~SrcOp1 & SrcOp2` and is often employed to simplify Boolean masking operations. Here are the results for source code example `Ch11_04`.

```
Results for AvxGprCountZeroBits
x: 0x01000008  lzcnt:   7  tzcnt:   3
x: 0x00008000  lzcnt:  16  tzcnt:  15
x: 0x08000000  lzcnt:   4  tzcnt:  27
x: 0x00000001  lzcnt:  31  tzcnt:   0
x: 0x00000000  lzcnt:  32  tzcnt:  32

Results for GprExtractBitField
x: 0x12345678  start:   4  len:    16  bextr: 0x00004567
x: 0x80808080  start:   7  len:     9  bextr: 0x00000101
x: 0xfedcba98  start:  24  len:     8  bextr: 0x000000fe

Results for GprAndNot
x: 0xf000000f  y: 0x12345678  andn: 0x02345670
x: 0xff00ff00  y: 0x12345678  andn: 0x00340078
x: 0xaaaaaaaa  y: 0xffaa5500  andn: 0x55005500
```

The BMI1 and BMI2 instruction set extensions also include other enhanced bit-manipulation instructions that can be used to implement specific algorithms or carry out specialized operations. These instructions are listed in Table 8-5.

# Half-Precision Floating-Point Conversions

The final source code example of this chapter, `Ch11_05`, exemplifies use of the half-precision conversion instructions `vcvtps2ph` and `vcvtph2ps`. Listing 11-5 shows the source code for this example. If your understanding of the half-precision floating-point data type is incomplete, you may want to review the content in Chapter 8 before perusing this section's source code and elucidations.

***Listing 11-5.*** Example Ch11_05

```
//------------------------------------------------
//               Ch11_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <string>
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void SingleToHalfPrecision_(uint16_t x_hp[8], float x_sp[8], int rc);
extern "C" void HalfToSinglePrecision_(float x_sp[8], uint16_t x_hp[8]);
```

```
int main()
{
    float x[8];

    x[0] = 4.125f;
    x[1] = 32.9f;
    x[2] = 56.3333f;
    x[3] = -68.6667f;
    x[4] = 42000.5f;
    x[5] = 75600.0f;
    x[6] = -6002.125f;
    x[7] = 170.0625f;

    uint16_t x_hp[8];
    float rn[8], rd[8], ru[8], rz[8];

    SingleToHalfPrecision_(x_hp, x, 0);
    HalfToSinglePrecision_(rn, x_hp);
    SingleToHalfPrecision_(x_hp, x, 1);
    HalfToSinglePrecision_(rd, x_hp);
    SingleToHalfPrecision_(x_hp, x, 2);
    HalfToSinglePrecision_(ru, x_hp);
    SingleToHalfPrecision_(x_hp, x, 3);
    HalfToSinglePrecision_(rz, x_hp);

    unsigned int w = 15;
    string line(76, '-');

    cout << fixed << setprecision(4);
    cout << setw(w) << "x";
    cout << setw(w) << "RoundNearest";
    cout << setw(w) << "RoundDown";
    cout << setw(w) << "RoundUp";
    cout << setw(w) << "RoundZero";
    cout << '\n' << line << '\n';

    for (int i = 0; i < 8; i++)
    {
        cout << setw(w) << x[i];
        cout << setw(w) << rn[i];
        cout << setw(w) << rd[i];
        cout << setw(w) << ru[i];
        cout << setw(w) << rz[i];
        cout << '\n';
    }

    return 0;
}
```

```
;-------------------------------------------------
;               Ch11_05_.asm
;-------------------------------------------------

; extern "C" void SingleToHalfPrecision_(uint16_t x_hp[8], float x_sp[8], int rc);

        .code
SingleToHalfPrecision_ proc

; Convert packed single-precision to packed half-precision
        vmovups ymm0,ymmword ptr [rdx]          ;ymm0 = 8 SPFP values

        cmp r8d,0
        jne @F
        vcvtps2ph xmm1,ymm0,0                   ;round to nearest
        jmp SaveResult

@@:     cmp r8d,1
        jne @F
        vcvtps2ph xmm1,ymm0,1                   ;round down
        jmp SaveResult

@@:     cmp r8d,2
        jne @F
        vcvtps2ph xmm1,ymm0,2                   ;round up
        jmp SaveResult

@@:     cmp r8d,3
        jne @F
        vcvtps2ph xmm1,ymm0,3                   ;truncate
        jmp SaveResult

@@:     vcvtps2ph xmm1,ymm0,4                   ;use MXCSR.RC

SaveResult:
        vmovdqu xmmword ptr [rcx],xmm1          ;save 8 HPFP values
        vzeroupper
        ret

SingleToHalfPrecision_ endp

; extern "C" void HalfToSinglePrecision_(float x_sp[8], uint16_t x_hp[8]);

HalfToSinglePrecision_ proc

; Convert packed half-precision to packed single-precision
        vcvtph2ps ymm0,xmmword ptr [rdx]
        vmovups ymmword ptr [rcx],ymm0          ;save 8 SPFP values
```

```
        vzeroupper
        ret

HalfToSinglePrecision_ endp
        end
```

The C++ function main starts its execution by loading single-precision floating-point test values into array x. It then exercises the half-precision floating-point conversion functions SingleToHalfPrecision_ and HalfToSinglePrecision_. Note that the function SingleToHalfPrecision_ requires a third argument, which specifies the rounding mode to use when converting floating-point values from single-precision to half-precision. Also note that an array type of uint16_t is used to store the half-precision floating-point results since C++ does not natively support a half-precision floating-point data type.

The assembly language function SingleToHalfPrecision_ uses the vcvtps2ph instruction to convert the eight single-precision floating-point values in array x_sp to half-precision floating-point. This instruction requires an immediate operand that specifies the rounding mode to use during the type conversion. Table 11-4 shows the rounding mode options for the vcvtps2ph instruction.

***Table 11-4.*** *Rounding Mode Options for the* vcvtps2ph *Instruction*

| Immediate Operand Bits | Value | Description |
|---|---|---|
| 1:0 | 00b | Round to nearest |
| | 01b | Round down (toward -∞) |
| | 10b | Round up (toward +∞) |
| | 11b | Round toward zero (truncate) |
| 2 | 0 | Use rounding mode specified in immediate operand bits 1:0 |
| | 1 | Use rounding mode specified in MXCSR.RC |
| 7:3 | Ignored | Not used |

The function HalfToSinglePrecision_ uses the vcvtph2ps instruction to convert eight half-precision floating-point values to single-precision floating-point. The output for source code example Ch11_05 follows this paragraph. Note the value differences between the various rounding modes. Also note that when the rounding mode RoundNearest or RoundUp is used, the value 76000.0f is converted to inf (or infinity) since this quantity exceeds the largest possible value for a half-precision floating-point value.

```
              x   RoundNearest      RoundDown        RoundUp      RoundZero
---------------------------------------------------------------------------
         4.1250         4.1250         4.1250         4.1250         4.1250
        32.9000        32.9063        32.8750        32.9063        32.8750
        56.3333        56.3438        56.3125        56.3438        56.3125
       -68.6667       -68.6875       -68.6875       -68.6250       -68.6250
     42000.5000     42016.0000     41984.0000     42016.0000     41984.0000
     75600.0000            inf     65504.0000            inf     65504.0000
     -6002.1250     -6004.0000     -6004.0000     -6000.0000     -6000.0000
       170.0625       170.0000       170.0000       170.1250       170.0000
```

# Summary

Here are the key learning points from Chapter 11:

- FMA instructions are often employed to implement numerically-oriented algorithms such as discrete convolutions, which are used extensively in a wide variety of problem domains, including signal processing and image processing.
- Assembly language functions that employ sequences of consecutive FMA instructions should use multiple XMM or YMM registers to store intermediate sum products. Using multiple registers helps avoid data dependencies that can preclude the processor from executing multiple FMA instructions simultaneously.
- Value discrepancies normally occur between functions that implement the exact same algorithm or operation using FMA and non-FMA instruction sequences. The significance of these discrepancies depends on the application.
- Assembly language functions can use the `mulx`, `sarx`, `shlx`, and `shrx` instructions to carry out flagless unsigned integer multiplication and shifts. These instructions may yield slightly faster performance than their flag-based counterparts in algorithms that perform consecutive multiplication and shift operations.
- Assembly language functions can use the `lzcnt`, `tzcnt`, `bextr`, and `andn` instructions to perform advanced bit-manipulation operations.
- Assembly language functions can use the `vcvtps2ph` and `vcvtph2ps` instructions to perform conversions between single-precision and half-precision floating-point values.

CHAPTER 12

# Advanced Vector Extensions 512

In the previous eight chapters, you learned about the scalar floating-point, packed floating-point, and packed integer capabilities of AVX and AVX2. In this chapter, you'll learn about Advance Vector Extensions 512 (AVX-512). AVX-512 is undoubtedly the largest and perhaps the most consequential extension of the x86 platform to date. It doubles the number of available SIMD registers and broadens the width of each register from 256 to 512 bits. AVX-512 also extends the instruction syntax of AVX and AVX2 to support additional capabilities not available in the earlier extensions, including conditional execution and merging, embedded broadcasts, and instruction-level rounding control for floating-point operations.

The content of this chapter is organized as follows. The first section presents a brief overview of AVX-512, which includes information about AVX-512's various instruction set extensions. This is followed by an examination of the AVX-512 execution environment, including its register sets, data types, instruction syntaxes, and enhanced computational features. The chapter concludes with a synopsis of the AVX-512 instruction set extensions that are included in recently marketed processors for server and workstation platforms.

## AVX-512 Overview

Unlike AVX and AVX2, AVX-512 is not a distinct instruction set extension. Rather, it's a congruous collection of interrelated instruction set extensions. An x86 processor is AVX-512 conforming if it supports the AVX512F (or foundation) instruction set extension. An AVX-512 conforming processor may optionally support additional AVX-512 instruction set extensions and these vary according to the processor's target market segment (e.g., high-performance computing, server, desktop, mobile, etc.). Table 12-1 lists the AVX-512 instruction set extensions that are currently available in some Intel processors. This table also includes the AVX-512 instruction set extensions that Intel has announced for inclusion in future processors. As of the writing of this text, AMD does not market any processors that support AVX-512.

The discussions in this chapter and the source code examples of Chapters 13 and 14 primarily focus on the AVX-512 instruction set extensions that are incorporated in Intel's Skylake Server microarchitecture, which was launched during 2017. This microarchitecture is used in Intel's Xeon Scalable (servers), Xeon W (workstations), and Core i7-7800X and i9-7900X series (high-end desktop) CPUs. Processors based on the Skylake Server microarchitecture contain the following AVX-512 instruction set extensions: AVX512F, AVX512CD, AVX512BW, AVX512DQ, and AVX512VL. Future mainstream processors from both AMD and Intel are expected to include these same AVX-512 extensions. Chapter 16 explains how to use the `cupid` instruction to detect the AVX-512 instructions set extensions that are shown in Table 12-1.

© Daniel Kusswurm 2018

D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_12

***Table 12-1.*** *Overview of AVX-512 Instruction Set Extensions*

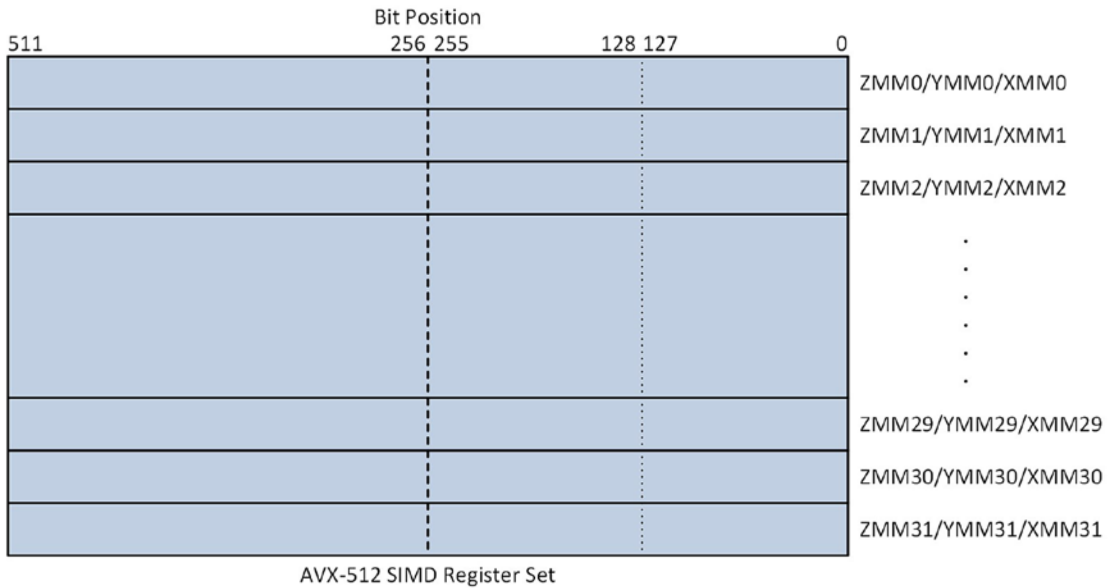
| CPUID Flag | Description |
|---|---|
| AVX512F | Foundation instructions |
| AVX512ER | Exponential and reciprocal instructions |
| AVX512PF | Prefetch instructions |
| AVX512CD | Conflict detect instructions |
| AVX512DQ | Doubleword and quadword instructions |
| AVX512BW | Byte and word instructions |
| AVX512VL | 128-bit and 256-bit vector instructions |
| AVX512_IFMA | Integer fused-multiply-add |
| AVX512_VBMI | Additional vector byte instructions |
| AVX512_4FMAPS | Packed single-precision FMA (4 iterations) |
| AVX512_4VNNI | Vector neural network instructions (4 iterations) |
| AVX512_VPOPCNTDQ | `vpopcnt[d\|q]` instructions |
| AVX512_VNNI | Vector neural net instructions |
| AVX512_VBMI2 | New vector byte, word, doubleword, and quadword instructions |
| AVX512_BITALG | `vpopcnt[b\|w]` and `vpshufbitqmb` instructions |

# AVX-512 Execution Environment

AVX-512 augments the execution environment of the x86 platform with the addition of new registers and data types. It also extends the assembly language instruction syntax of AVX and AVX2 to support enhanced operations such as conditional executions and merging, embedded broadcasts, and instruction level rounding control. This section discusses these enhancements in greater detail.

## Register Sets

Figure 12-1 illustrates the AVX-512 register sets. AVX-512 extends the width of each AVX SIMD register from 256 bits to 512 bits. The 512-bit wide registers are known as the ZMM register set. AVX-512 conforming processors include 32 ZMM registers named ZMM0–ZMM31. The YMM and XMM register sets are aliased to the low-order 256 bits and 128 bits of each ZMM register, respectively. AVX-512 processors also include eight new opmask registers named K0–K7. These registers are primarily used as predicate masks to perform conditional executions and merging operations. They can also be employed as destination operands for instructions that generate vector mask results. You'll learn more about these registers later in this chapter.



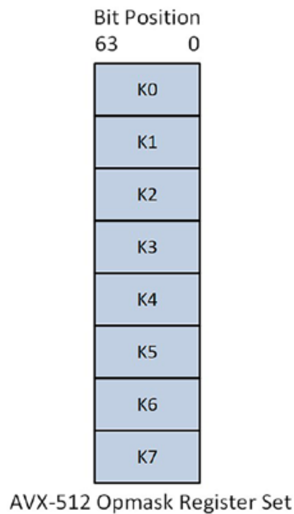




***Figure 12-1.*** *AVX-512 register sets*

## Data Types

Similar to the YMM and XMM registers, software functions can use the ZMM registers to carry out SIMD operations using packed integer or packed floating-point operands. Table 12-2 shows the maximum number of elements that a ZMM register can hold for each supported data type. This table also shows the maximum number of elements that a YMM and XMM register can hold for comparison purposes.

***Table 12-2.*** *Maximum Number of Elements for AVX-512 Register Operands*

| Data Type | ZMM | YMM | XMM |
|---|---|---|---|
| Integer byte | 64 | 32 | 16 |
| Integer word | 32 | 16 | 8 |
| Integer doubleword | 16 | 8 | 4 |
| Integer quadword | 8 | 4 | 2 |
| Single-precision floating-point | 16 | 8 | 4 |
| Double-precision floating-point | 8 | 4 | 2 |

The alignment requirements for 512-bit wide operands in memory are similar to other x86 SIMD operands. Except for instructions that explicitly specify an aligned operand (e.g., `vmovdqa[32|64]`, `vmovap[d|s]`, etc.), proper alignment of a 512-bit wide operand in memory is not mandatory. However, 512-bit wide operands should always be aligned on a 64-byte boundary whenever possible to avoid processing delays that can occur if the processor is forced to access an unaligned operand in memory. AVX-512 instructions that access 256-bit or 128-bit wide operands in memory should also ensure that these types of operands are properly aligned on their respective natural boundaries.

# Instruction Syntax

AVX-512 extends the instruction syntax of AVX and AVX2. Most AVX-512 instructions can use the same three-operand instruction syntax as AVX and AVX2 instructions, which consists of two non-destructive source operands and one destination operand. AVX-512 instructions can also exploit several new optional operands. These operands facilitate conditional executions and merging, embedded broadcast operations, and floating-point rounding control. The next few sections discuss AVX-512's optional instruction operands in greater detail.

## Conditional Execution and Merging

Most AVX-512 instructions support conditional execution and merging. A conditional execution and merge operation uses the bits of an opmask register as a predicate mask to control instruction execution and destination operand updates on a per-element basis. Figure 12-2 illustrates this concept in greater detail. In this figure, registers ZMM0, ZMM1, and ZMM2 each contain 16 single-precision floating-point values. The 16 low-order bits of opmask register K1 constitute the predicate mask. When an opmask register is used in this manner, each bit controls how the result of corresponding element position in the destination operand is calculated and updated.

Figure 12-2 also shows the outcome of three distinct executions of the `vaddps` instruction using the same initial values. The first example instruction, `vaddps zmm2,zmm0,zmm1`, performs a packed single-precision floating-point add of the elements in ZMM0 and ZMM1 and saves the resultant sums in register ZMM2. Execution of this instruction is no different than an AVX `vaddps` instruction that uses XMM or YMM register operands. The next example instruction, `vaddps zmm2{k1},zmm0,zmm1`, illustrates how the bits of opmask register K1 are used to conditionally add and update the destination operand on a per-element basis. More specifically, an element sum is calculated and saved in the destination operand only if the corresponding bit position of the opmask register is set to one; otherwise, the destination operand element position remains unchanged. This is called *merge masking*. The final example instruction in Figure 12-2, `vaddps zmm2{k1}{z},zmm0,zmm1`, is similar to the previous instruction. The extra `{z}` operand instructs the processor to perform zero masking instead of merge masking. Zero masking sets a destination operand element to zero if its corresponding bit position in the opmask register is set to zero; otherwise, the sum is calculated and saved.

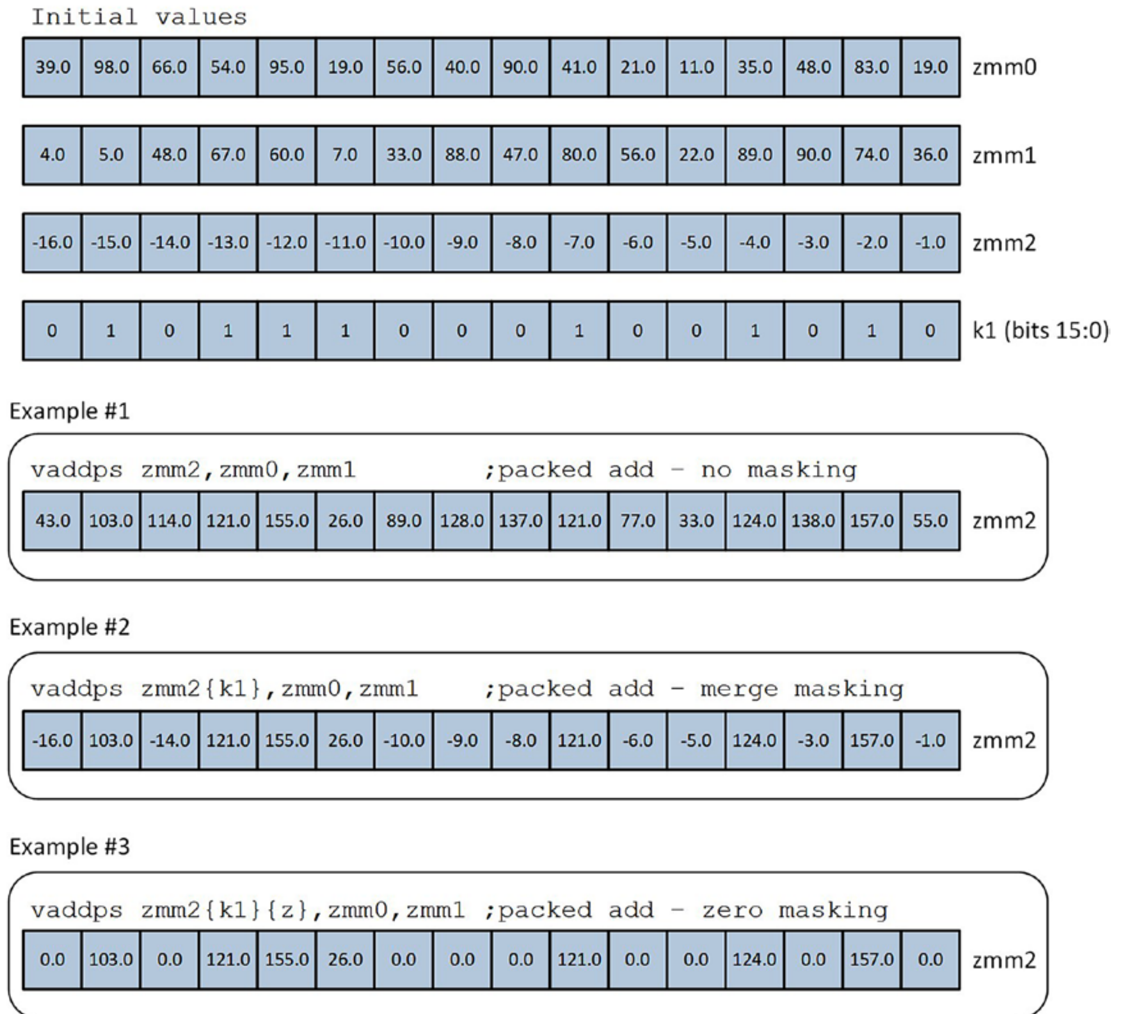



***Figure 12-2.*** *Execution examples of the* `vaddps` *instruction using no masking, merge masking, and zero masking*

At this point a few words about the opmask registers are warranted. The eight opmask registers are somewhat like the general-purpose registers. On processors that support AVX-512, each opmask register is 64-bits wide. However, when employed as a predicate mask, only the low-order bits are used during instruction execution. The exact number of used low-order bits varies depending on the number of vector elements. In Figure 12-2, bits 0–15 of opmask register K1 form the predicate mask since the `vaddps` instruction employs ZMM register operands that contain 16 single-precision floating-point values.

AVX-512 includes several new instructions that can be used to read values from and write values to an opmask register and perform Boolean operations. You'll learn about these instructions later in this chapter. An opmask register can also be used as destination operand with instructions that generate a vector mask result such as `vcmpp[d|s]` and `vpcmp[b|w|d|q]`. The source code examples in Chapters 13 and 14 illustrate how to use these instructions with an opmask register. AVX-512 instructions can use opmask registers K1–K7 as a predicate mask. Opmask register K0 cannot be employed as a predicate mask operand but it can be used

in any instruction that requires a source or destination operand opmask register. If an AVX-512 instruction attempts to use K0 as a predicate mask, the processor substitutes an implicit operand of all 1s, which disables all conditional execution and masking operations.

## Embedded Broadcast

Many AVX-512 instructions can carry out a SIMD computation using an embedded broadcast operand. An embedded broadcast operand is a memory-based scalar value that is replicated *N* times into a temporary packed value, where *N* represents the number of vector elements referenced by the instruction. This temporary packed value is then used as an operand in a SIMD calculation.

Figure 12-3 contains two example instruction sequences that illustrate broadcast operations. The first example uses the `vbroadcastss` instruction to load the single-precision floating-point constant 2.0 into each element position of ZMM1. The ensuing `vmulps zmm2,zmm0,zmm1` instruction multiplies each value in ZMM0 by 2.0 and saves the results to ZMM2. The second example instruction in Figure 12-3, `vmulps zmm2,zmm0,real4 bcst [rax]`, carries out this same operation using an embedded broadcast operand. The text `real4 bcst` is a MASM directive that instructs the assembler to treat the memory location pointed to by register RAX as an embedded broadcast operand.

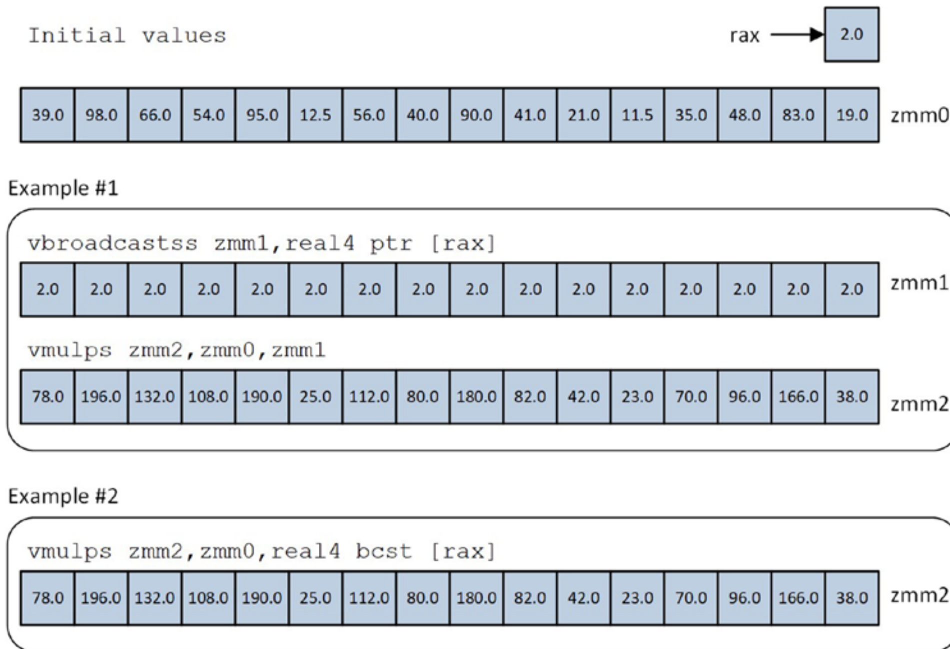



***Figure 12-3.*** *Packed single-precision floating-point multiplication using the `vbroadcastss` and `vmulps` instructions versus a `vmulps` instruction with an embedded broadcast operand*

AVX-512 supports embedded broadcast operations using 32-bit and 64-bit wide elements. Embedded broadcasts cannot be performed using 8-bit and 16-bit wide elements.

## Instruction Level Rounding

The final AVX-512 instruction syntax enhancement involves instruction-level rounding control for floating-point operations. In Chapter 5, you learned how to use the `vldmxcsr` and `vstmxcsr` instructions to change the processor's global rounding mode for floating-point operations (see example `Ch05_06`). AVX-512 allows some instructions to specify a floating-point rounding mode operand that overrides the current rounding mode in MXCSR.RC. Table 12-3 shows the supported rounding mode operands, which are also called static rounding modes. The `-sae` suffix that's appended to each static rounding mode operand string is an acronym for suppress all exceptions. This suffix serves as a reminder that floating-point exceptions are *always* masked whenever a static rounding mode operand is specified; MXCSR flag updates are also disabled.

***Table 12-3.*** *AVX-512 Instruction-Level Static Rounding Mode Operands*

| Rounding Mode Operand | Description |
|---|---|
| `{rn-sae}` | Round to nearest |
| `{rd-sae}` | Round down (toward –∞) |
| `{ru-sae}` | Round up (toward +∞) |
| `{rz-sae}` | Round toward zero (truncate) |

Static rounding mode operands can be used with many (but not all) AVX-512 instructions that perform floating-point operations using 512-bit wide packed operands; 256-bit and 128-bit wide packed operands are not supported. Static rounding mode operands can also be used with instructions that perform scalar floating-point operations. In both use cases, all instruction operands must be registers. For example, the instructions `vmulps zmm2,zmm0,zmm1 {rz-sae}` and `vmulss xmm2,xmm0,xmm1 {rz-sae}` are valid, whereas `vmulps zmm2,zmm0,zmmword ptr [rax] {rz-sae}` and `vmulss xmm2,xmm0,real4 ptr [rax] {rz-sae}` are invalid. Some AVX-512 floating-point instructions do not support the specification of a static rounding mode operand, but these instructions still can use the operand `{sae}` to suppress all exceptions.

# Instruction Set Overview

This section presents an overview of the following AVX-512 instruction set extensions: AVX512F, AVX512CD, AVX512BW, and AVX512DQ. It also includes a summary of the opmask register instructions. The tables in this section only include instructions that are new to AVX-512. They do not include instructions that are a simple promotion of an existing AVX or AVX2 instruction. Most of the instructions in these tables can be used with 512-bit wide operands; 256-bit and 128-bit wide operands can be used on processors that support AVX512VL.

### AVX512F

Table 12-4 lists the AVX512F instructions. As mentioned in the overview section of this chapter, all AVX-512 conforming processors must minimally support the instructions that are included in this table.

***Table 12-4.*** *AVX512F Instruction Set Overview*

| Mnemonic | Description |
|---|---|
| valign[d\|q] | Align doubleword \| quadword vectors |
| vblendmp[d\|s] | Blend floating-point vectors using opmask control |
| vbroadcastf[32x4\|64x4] | Broadcast floating-point tuples |
| vbroadcasti[32x4\|64x4] | Broadcast integer tuples |
| vcompressp[d\|s] | Store sparse packed floating-point values |
| vcvtp[d\|s]2udq | Convert packed floating-point to packed unsigned doubleword integers |
| vcvts[d\|s]2usi | Convert scalar floating-point to unsigned doubleword integer |
| vcvttp[d\|s]2udq | Convert packed floating-point to packed unsigned doubleword integers with truncation |
| vcvtts[d\|s]2usi | Convert scalar floating-point to unsigned doubleword integer with truncation |
| vcvtudq2p[d\|s] | Convert packed unsigned doubleword integers to packed floating-point |
| vcvtusi2s[d\|s] | Convert unsigned doubleword integer to floating-point |
| vexpandp[d\|s] | Load sparse packed floating-point values |
| vextractf[32x4\|64x4] | Extract packed floating-point values |
| vextracti[32x4\|64x4] | Extract packed integer values |
| vfixupimmp[d\|s] | Fix up special packed floating-point values |
| vfixupimms[d\|s] | Fix up special scalar floating-point values |
| vgetexpp[d\|s] | Convert exponents of packed floating-point values |
| vgetexps[d\|s] | Convert exponents of scalar floating-point values |
| vgetmantp[d\|s] | Get normalized mantissas from packed floating-point values |
| vgetmants[d\|s] | Get normalized mantissas from scalar floating-point value |
| vinsertf[32x4\|64x4] | Insert packed floating-point values |
| vinserti[32x4\|64x4] | Insert packed integer values |
| vmovdqa[32\|64] | Move aligned packed integers |
| vmovdqu[32\|64] | Move unaligned packed integers |
| vpblendm[d\|q] | Blend packed integers using opmask control |
| vpbroadcast[d\|q] | Broadcast integer from general-purpose register |
| vpcmp[d\|q] | Compare packed signed integers |
| vpcmpu[d\|q] | Compare packed unsigned integers |
| vpcompress[d\|q] | Store sparse packed integers |
| vpermi2[d\|q\|ps\|pd] | Permute from two tables overwriting the index |
| vpermt2[d\|q\|ps\|pd] | Permute from two tables overwriting one table |
| vpmov[db\|sdb\|usdb] | Down convert packed doublewords to packed bytes |
| vpexpand[d\|q] | Load sparse packed integers |

(*continued*)

***Table 12-4.*** (*continued*)

| Mnemonic | Description |
|---|---|
| vpmax[s\|u]q | Calculated packed quadword maximums |
| vpmin[s\|u]q | Calculate packed quadword minimums |
| vpmov[db\|sdb\|usdb] | Down convert packed doublewords to packed bytes |
| vpmov[dw\|sdw\|usdw] | Down convert packed doublewords to packed words |
| vpmov[qb\|sqb\|usqb] | Down convert packed quadwords to packed bytes |
| vpmov[qd\|sqd\|usqd] | Down convert packed quadwords to packed doublewords |
| vpmov[qw\|sqw\|usqw] | Down convert packed quadwords to packed words |
| vprol[d\|q] | Rotate left packed integers using constant count |
| vprolv[d\|q] | Rotate left pack integers using variable counts |
| vpror[d\|q] | Rotate right packed integers using constant count |
| vprorv[d\|q] | Rotate right packed integers using variable counts |
| vpscatterd[d\|q] | Scatter packed integers using doubleword indices |
| vpscatterq[d\|q] | Scatter packed integers using quadword indices |
| vpsraq | Shift right arithmetic packed quadword integers using constant count |
| vpsravq | Shift right arithmetic packed quadword integers using variable counts |
| vpternlog[d\|q] | Bitwise ternary logic |
| vptestm[d\|q] | Packed integer bitwise AND and set mask |
| vptestnm[d\|q] | Packed integer bitwise NAND and set mask |
| vrcp14p[d\|s] | Compute approximate reciprocals of packed floating-point values |
| vrcp14s[d\|s] | Compute approximate reciprocals of scalar floating-point value |
| vreducep[d\|s] | Perform reduction transformation on packed floating-point values |
| vreduces[d\|s] | Perform reduction transformation on scalar floating-point value |
| vrndscalep[d\|s] | Round packed floating-point values to number of fractional bits |
| vrndscales[d\|s] | Round floating-point value to number of fractional bits |
| vrsqrt14p[d\|s] | Compute approximate reciprocals of packed floating-point square roots |
| vrsqrt14s[d\|s] | Compute approximate reciprocals of scalar floating-point square root |
| vscalefp[d\|s] | Scale packed floating-point values |
| vscalefs[d\|s] | Scale scalar floating-point value |
| vscatterdp[d\|s] | Scatter packed floating-point values using doubleword indices |
| vscatterqp[d\|s] | Scatter packed floating-point values using quadword indices |
| vshuff[32x4\|64x2] | Shuffle packed floating-point values |
| vshufi[32x4\|64x2] | Shuffle packed integer values |

## AVX512CD

Table 12-5 lists the AVX512CD instructions. These instructions are frequently used to detect and mitigate data dependencies that can occur when performing sparse array calculations or scatter operations. They can also be used with other AVX-512 instructions to perform ordinary computations.

***Table 12-5.*** *AVX512CD Instruction Set Overview*

| Mnemonic | Description |
| --- | --- |
| `vpbroadcastm[b2q|w2d]` | Broadcast mask to vector register |
| `vpconflict[d|q]` | Detect conflicts within packed integers |
| `vplzcnt[d|q]` | Count number of leading zeros in packed integers |

## AVX512BW

Table 12-6 lists the AVX512BW instructions. These instructions carry out their operations using packed byte and word operands.

***Table 12-6.*** *AVX512BW Instruction Set Overview*

| Mnemonic | Description |
| --- | --- |
| `vdbpsadbw` | Double block packed sum-absolute-differences using unsigned bytes |
| `vmovdq[u8|u16]` | Move unaligned packed integers |
| `vpblendm[b|w]` | Blend packed integers using opmask control |
| `vpbroadcast[b|w]` | Broadcast integer from general-purpose register |
| `vpcmp[b|w]` | Compare packed signed integers |
| `vpcmpu[b|w]` | Compare packed unsigned integers |
| `vpermw` | Permute packed words |
| `vpermi2w` | Permute word integers from two tables overwriting the index |
| `vpermt2w` | Permute word integers from two tables overwriting one table |
| `vpmov[b|w]2m` | Convert vector register to mask register |
| `vpmovm2[b|w]` | Convert mask register to vector register |
| `vpmovw[b|sb|usb]` | Down convert packed words to packed bytes |
| `vpsllvw` | Packed word shift left logical using variable bit counts |
| `vpsravw` | Packed word shift right arithmetic using variable bit counts |
| `vpsrlvw` | Packed word shift right logical using variable bit counts |
| `vptestm[b|w]` | Packed integer bitwise AND and set mask |
| `vptestnm[b|w]` | Packed integer bitwise NAND and set mask |

## AVX512DQ

Table 12-7 lists the AVX512DQ instructions. These instructions carry out their operations using packed doubleword and quadword operands. AVX512DQ also includes instructions that perform conversions between packed floating-point and integer quadwords.

***Table 12-7.*** *AVX512DQ Instruction Set Overview*

| Mnemonic | Description |
|---|---|
| `vcvtp[d|s]2qq` | Convert packed floating-point to signed quadword integers |
| `vcvtp[d|s]2uqq` | Convert packed floating-point to unsigned quadword integers |
| `vcvttp[d|s]2qq` | Convert packed floating-point to signed quadword integers with truncation |
| `vcvttp[d|s]2uqq` | Convert packed floating-point to unsigned quadword integers with truncation |
| `vcvtuqq2p[d|s]` | Convert packed unsigned quadword integers to floating-point |
| `vextractf64x2` | Extract packed double-precision floating-point values |
| `vextracti64x2` | Extract packed quadword values |
| `vfpclass[pd|ps]` | Test packed floating-point class |
| `vfpclass[sd|ss]` | Test scalar floating-point class |
| `vinsertf64x2` | Insert packed double-precision floating-point values |
| `vinserti64x2` | Insert packed quadword values |
| `vpmov[d|q]2m` | Convert vector register to mask register |
| `vpmovm2[d|q]` | Convert mask register to vector register |
| `vpmullq` | Multiply packed quadword integers and store low result |
| `vrangep[d|s]` | Range restriction calculation for packed floating-point |
| `vranges[d|s]` | Range restriction calculation for scalar floating-point |
| `vreducep[d|s]` | Perform reduction on packed floating-point values |
| `vreduces[d|s]` | Perform reduction on scalar floating-point values |

## Opmask Registers

Table 12-8 lists the opmask register instructions. The word versions of these instructions require AVX512F except for `kaddw` and `ktestw`, which require AVX512DQ. The doubleword and quadword versions of the opmask register instructions require AVX512BW; the byte versions require AVX512DQ.

*Table 12-8.* *Opmask Register Instruction Set Overview*

| Mnemonic | Description |
|---|---|
| kadd[b\|w\|d\|q] | Add mask values |
| kand[b\|w\|d\|q] | Bitwise AND |
| kandn[b\|w\|d\|q] | Bitwise AND NOT |
| kmov[b\|w\|d\|q] | Move value to/from opmask register |
| knot[b\|w\|d\|q] | Bitwise NOT |
| kor[b\|w\|d\|q] | Bitwise inclusive OR |
| kortest[b\|w\|d\|q] | Bitwise inclusive OR; update RFLAGS.ZF and RFLAGS.CF |
| kshiftl[b\|w\|d\|q] | Shift left |
| kshiftr[b\|w\|d\|q] | Shift right |
| ktest[b\|w\|d\|q] | Bitwise AND and ANDN; update RFLAGS.ZF and RFLAGS.CF |
| kunpck[bw\|wd\|dq] | Unpack |
| kxnor[b\|w\|d\|q] | Bitwise exclusive NOR |
| kxor[b\|w\|d\|q] | Bitwise exclusive OR |

# Summary

Here are the key learning points for Chapter 12:

- All AVX-512 conforming processors support the AVX512F instruction set extension. Inclusion of additional AVX-512 instruction set extensions varies depending on the processor's target market.
- The AVX-512 register set includes 32 512-bit wide registers named ZMM0–ZMM31. The low-order 256 and 128 bits are aliased to registers YMM0–YMM31 and XMM0–XMM31, respectively.
- The AVX-512 register set also includes eight opmask registers named K0–K7. Opmask registers K1–K7 can be used to perform instruction-level conditional executions with merge masking or zero masking.
- Many AVX-512 instructions that require a packed operand of constant values can use an embedded broadcast operand instead of a separate broadcast instruction.
- A static rounding mode operand can be specified with many AVX-512 instructions that perform floating-point operations using 512-bit wide packed or scalar floating-point register operands.

CHAPTER 13

# AVX-512 Programming – Floating-Point

In previous chapters, you learned how to carry out scalar and packed floating-point operations using the AVX and AVX2 instruction sets. In this chapter, you learn how to perform these operations using the AVX-512 instruction set. The first part of this chapter contains source code examples that illustrate basic AVX-512 programming concepts using scalar floating-point operands. This includes examples that illustrate conditional executions, merge and zero masking, and instruction-level rounding. The second part of this chapter demonstrates how to use the AVX-512 instruction set to carry out packed floating-point calculations using 512-bit wide operands and the ZMM register set.

The source code examples of this chapter require a processor and operating system that support AVX-512 and the following instruction set extensions: AVX512F, AVX512CD, AVX512BW, AVX512DQ, and AVX512VL. As discussed in Chapter 12, these extensions are supported by processors that are based on the Intel Skylake Server microarchitecture. Future processors from both AMD and Intel are also likely to incorporate the previously-mentioned instruction set extensions. You can use one of the freely available utilities listed in Appendix A to determine which AVX-512 instruction sets your system supports. In Chapter 16, you learn how to use the `cupid` instruction to detect specific AVX-512 instruction set extensions at runtime.

## Scalar Floating-Point

AVX-512 extends the scalar floating-point capabilities of AVX to include merge masking, zero masking, and instruction-level rounding control. The source code examples of this section explain how to use these capabilities. They also exemplify some minor differences that you need to be aware of when writing scalar float-point code using AVX-512 instructions.

### Merge Masking

Listing 13-1 shows the source code for example `Ch13_01`. This example describes how to perform merge masking using AVX-512 scalar floating-point instructions. It also illustrates the use of several opmask register instructions.

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_13

***Listing 13-1.*** Example Ch13_01

```
//------------------------------------------------
//               Ch13_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <string>
#include <iostream>
#include <iomanip>
#include <limits>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

extern "C" double g_PI = M_PI;
extern "C" bool Avx512CalcSphereAreaVol_(double* sa, double* vol, double radius, double
error_val);

bool Avx512CalcSphereAreaVolCpp(double* sa, double* vol, double radius, double error_val)
{
    bool rc;

    if (radius < 0.0)
    {
        *sa = error_val;
        *vol = error_val;
        rc = false;
    }
    else
    {
        *sa = 4.0 * g_PI * radius * radius;
        *vol = *sa * radius / 3.0;
        rc = true;
    }

    return rc;
}

int main()
{
    const double error_val = numeric_limits<double>::quiet_NaN();
    const double radii[] = {-1.0, 0.0, 1.0, 2.0, 3.0, 4.0, -7.0, 10.0, -18.0, 20.0};
    int num_r = sizeof(radii) / sizeof(double);

    string sp {"  "};
    string sep(75, '-');

    cout << setw(10) << "radius" << sp;
    cout << setw(6) << "rc1" << sp;
    cout << setw(6) << "rc2" << sp;
```

```
    cout << setw(10) << "sa1" << sp;
    cout << setw(10) << "sa2" << sp;
    cout << setw(10) << "vol1" << sp;
    cout << setw(10) << "vol2" << '\n';
    cout << sep << '\n';

    cout << fixed << setprecision(4);

    for (int i = 0; i < num_r; i++)
    {
        double sa1, sa2;
        double vol1, vol2;
        double r = radii[i];

        bool rc1 = Avx512CalcSphereAreaVolCpp(&sa1, &vol1, r, error_val);
        bool rc2 = Avx512CalcSphereAreaVol_(&sa2, &vol2, r, error_val);

        cout << setw(10) << r << sp;
        cout << setw(6) << boolalpha << rc1 << sp;
        cout << setw(6) << boolalpha << rc2 << sp;
        cout << setw(10) << sa1 << sp;
        cout << setw(10) << sa2 << sp;
        cout << setw(10) << vol1 << sp;
        cout << setw(10) << vol2 << '\n';
    }

    return 0;
}

;-------------------------------------------------
;               Ch13_01.asm
;-------------------------------------------------

            include <cmpequ.asmh>
            .const
r8_three    real8 3.0
r8_four     real8 4.0

            extern g_PI:real8

; extern "C" bool Avx512CalcSphereAreaVol_(double* sa, double* v, double r, double error_val);
;
; Returns:  false = invalid radius, true = valid radius

        .code
Avx512CalcSphereAreaVol_ proc

; Test radius for value >= 0.0
        vmovsd xmm0,xmm0,xmm2               ;xmm0 = radius
        vxorpd xmm5,xmm5,xmm5               ;xmm5 = 0.0
        vmovsd xmm16,xmm16,xmm3             ;xmm16 = error_val
        vcmpsd k1,xmm0,xmm5,CMP_GE          ;k1[0] = 1 if radius >= 0.0
```

```
; Calculate surface area and volume using mask from compare
        vmulsd xmm1{k1},xmm0,xmm0           ;xmm1 = r * r
        vmulsd xmm2{k1},xmm1,[r8_four]      ;xmm2 = 4 * r * r
        vmulsd xmm3{k1},xmm2,[g_PI]         ;xmm3 = 4 * PI * r * r (sa)
        vmulsd xmm4{k1},xmm3,xmm0           ;xmm4 = 4 * PI * r * r * r
        vdivsd xmm5{k1},xmm4,[r8_three]     ;xmm5 = 4 * PI * r * r * r / 3 (vol)

; Set surface area and volume to error_val if radius < 0.0 is true
        knotw k2,k1                         ;k2[0] = 1 if radius < 0.0
        vmovsd xmm3{k2},xmm3,xmm16          ;xmm3 = error_val if radius < 0.0
        vmovsd xmm5{k2},xmm5,xmm16          ;xmm5 = error_val if radius < 0.0

; Save results
        vmovsd real8 ptr [rcx],xmm3         ;save surface area
        vmovsd real8 ptr [rdx],xmm5         ;save volume

        kmovw eax,k1                        ;eax = return code
        ret
Avx512CalcSphereAreaVol_ endp
        end
```

The C++ code in Listing 13-1 starts with the function `Avx512CalcSphereAreaVolCpp`. This function calculates the surface area and volume of any sphere whose radius is greater or equal to zero. If the sphere's radius is less than zero, `Avx512CalcSphereAreaVolCpp` sets the surface area and volume to `error_val`. The remaining C++ code in Listing 13-1 performs test case initialization, exercises the functions `Avx512CalcSphereAreaVolumeCpp` and `Avx512CalcSphereAreaVolume_`, and streams results to `cout`.

The assembly language function `Avx512CalcSphereAreaVol_` implements the same algorithm as its C++ counterpart. This function begins with a `vmovsd xmm0,xmm0,xmm2` instruction that copies argument value `r` to register XMM0. It then loads register XMM5 with 0.0. The `vmovsd xmm16,xmm16,xmm3` instruction copies `error_val` into register XMM16. According to the Visual C++ calling convention, the new AVX-512 registers ZMM16–ZMM31 along with the low-order YMM and XMM counterparts are volatile across function boundaries. This means that these registers can be used by any assembly language function without preserving their values. The next instruction, `vcmpsd k1,xmm0,xmm5,CMP_GE`, sets opmask register bit K1[0] to one if `r` is greater than or equal to zero; otherwise, this bit is set to zero.

The first instruction of the surface area and volume calculation code block, `vmulsd xmm1{k1},xmm0,xmm0`, computes `r * r` if bit K1[0] is set to one (`r >= 0.0` is true); it then saves the calculated product in XMM1[63:0]. If bit K1[0] is set to zero (`r < 0.0` is true), the processor skips the double-precision floating-point multiplication calculation and leaves register XMM1 unaltered. The next instruction, `vmulsd xmm2{k1},xmm1,[r8_four]`, computes `4.0 * r * r` using the same merge masking operation as the previous instruction. The ensuing `vmulsd` and `vdivsd` instructions complete the required surface area (XMM3) and volume (XMM5) calculations. The merge masking operations in this code block exemplify one of AVX-512's key computational capabilities: the processor carries out the double-precision floating-point arithmetic calculations *only* if bit K1[0] is set to one; otherwise no calculations are performed, and the respective destination operand registers remain unchanged.

Following the surface area and volume calculations, the `knotw k2,k1` negates the low-order 16 bits of K1 and saves this result to K2[15:0]. This instruction also sets bits K2[63:16] to zero. Bit K2[0] is now set to one if `r < 0.0` is true. The `knotw` instruction is used here since it's part of AVX512F instruction set extension; `knot[b|d|q]` would also work here. The next instruction, `vmovsd xmm3{k2},xmm3,xmm16`, sets the surface area to `error_val` if `r < 0.0` is true. The subsequent `vmovsd xmm5{k2},xmm5,xmm16` instruction performs the same operation for the volume value. The final `kmovw eax,k1` instruction loads EAX with the function return code. Here are the results for source code example `Ch13_01`:

```
   radius      rc1      rc2         sa1         sa2        vol1        vol2
---------------------------------------------------------------------------
  -1.0000    false    false         nan         nan         nan         nan
   0.0000     true     true      0.0000      0.0000      0.0000      0.0000
   1.0000     true     true     12.5664     12.5664      4.1888      4.1888
   2.0000     true     true     50.2655     50.2655     33.5103     33.5103
   3.0000     true     true    113.0973    113.0973    113.0973    113.0973
   4.0000     true     true    201.0619    201.0619    268.0826    268.0826
  -7.0000    false    false         nan         nan         nan         nan
  10.0000     true     true   1256.6371   1256.6371   4188.7902   4188.7902
 -18.0000    false    false         nan         nan         nan         nan
  20.0000     true     true   5026.5482   5026.5482  33510.3216  33510.3216
```

## Zero Masking

The next source code example is named Ch13_02. This example demonstrates how to use zero masking to eliminate data-dependent conditional jumps from a calculation. Listing 13-2 shows the source code for this example.

***Listing 13-2.*** Example Ch13_02

```
//------------------------------------------------
//               Ch13_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <array>
#include <random>

using namespace std;

extern "C" bool Avx512CalcValues_(double* c, const double* a, const double* b, size_t n);

template<typename T> void Init(T* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 200};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (T)(ui_dist(rng) - 25);
}

bool Avx512CalcValuesCpp(double* c, const double* a, const double* b, size_t n)
{
    if (n == 0)
        return false;
```

```
    for (size_t i = 0; i < n; i++)
    {
        double val = a[i] * b[i];
        c[i] = (val >= 0.0) ? sqrt(val) : val * val;
    }

    return true;
}

int main()
{
    const size_t n = 20;
    array<double, n> a;
    array<double, n> b;
    array<double, n> c1;
    array<double, n> c2;

    Init<double>(a.data(), n, 13);
    Init<double>(b.data(), n, 23);

    bool rc1 = Avx512CalcValuesCpp(c1.data(), a.data(), b.data(), n);
    bool rc2 = Avx512CalcValues_(c2.data(), a.data(), b.data(), n);

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code - ";
        cout << "rc1 = " << boolalpha << rc1 << "  ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
    }
    else
    {
        cout << fixed << setprecision(4);

        for (size_t i = 0; i < n; i++)
        {
            cout << "i:  " << setw(2) << i << "  ";
            cout << "a:  " << setw(9) << a[i] << "  ";
            cout << "b:  " << setw(9) << b[i] << "  ";
            cout << "c1: " << setw(13) << c1[i] << "  ";
            cout << "c2: " << setw(13) << c2[i] << "\n";
        }
    }
}

;-------------------------------------------------
;               Ch13_02.asm
;-------------------------------------------------

        include <cmpequ.asmh>

; extern "C" bool Avx512CalcValues_(double* c, const double* a, const double* b, size_t n);
```

```
        .code
Avx512CalcValues_ proc

; Validate n and initialize array index i
        xor eax,eax                         ;set error return code (also i = 0)
        test r9,r9                          ;is n == 0?
        jz Done                             ;jump if n is zero

        vxorpd xmm5,xmm5,xmm5               ;xmm5 = 0.0

; Load next a[i] and b[i], calculate val
@@:     vmovsd xmm0,real8 ptr [rdx+rax*8]   ;xmm0 = a[i];
        vmovsd xmm1,real8 ptr [r8+rax*8]    ;xmm1 = b[i];
        vmulsd xmm2,xmm0,xmm1               ;val = a[i] * b[i]

; Calculate c[i] = (val >= 0.0) ? sqrt(val) : val * val
        vcmpsd k1,xmm2,xmm5,CMP_GE          ;k1[0] = 1 if val >= 0.0
        vsqrtsd xmm3{k1}{z},xmm3,xmm2       ;xmm3 = (val > 0.0) ? sqrt(val) : 0.0
        knotw k2,k1                         ;k2[0] = 1 if val < 0.0
        vmulsd xmm4{k2}{z},xmm2,xmm2        ;xmm4 = (val < 0.0) ? val * val : 0.0
        vorpd xmm0,xmm4,xmm3                ;xmm0 = (val >= 0.0) ? sqrt(val) : val * val
        vmovsd real8 ptr [rcx+rax*8],xmm0   ;save result to c[i]

; Update index i and repeat until done
        inc rax                             ;i += 1
        cmp rax,r9
        jl @B
        mov eax,1                          ;set success return code

Done:   ret
Avx512CalcValues_ endp
        end
```

In the C++ code, the function `Avx512CalcValuesCpp` performs a simple arithmetic calculation using double-precision floating-point arrays. Each loop iteration begins by calculating the intermediate value `val = a[i] * b[i]`. The next statement, `c[i] = (val >= 0.0) ? sqrt(val) : val * val`, loads `c[i]` with a quantity that varies depending on the value of `val`. The assembly language function `Avx512CalcValues_` also performs the same computation. The C++ function `main` contains code that initializes the test arrays, exercises the functions `Avx512CalcValuesCpp` and `Avx512CalcValues_`, and displays the results.

The processing loop of `Avx512CalcValues_` begins with two `vmovsd` instructions that load `a[i]` and `b[i]` into registers XMM0 and XMM1, respectively. The ensuing `vmulsd xmm2,xmm0,xmm1` instruction computes the intermediate product `val = a[i] * b[i]`. Following the calculation of `val`, the `vcmpsd k1,xmm2,xmm5,CMP_GE` instruction compares `val` against 0.0 and sets bit K1[0] to one if `val` is greater than or equal to zero; otherwise bit K1[0] is set to zero. The next instruction, `vsqrtsd xmm3{k1}{z},xmm3,xmm2`, calculates the square root of `val` if K1[0] is set to one and saves the result in XMM3. If K1[0] is zero, the processor skips the square root calculation and sets register XMM3 to 0.0.

The `knotw k2,k1` instruction sets K2[0] to one if `val` is less than 0.0. The ensuing `vmulsd xmm4{k2}{z},xmm2,xmm2` instruction calculates and saves the product `val * val` in XMM4 if bit K2[0] is set to one; otherwise XMM4 is set equal to 0.0. Following execution of the `vmulsd` instruction, register XMM3 contains `sqrt(val)` and XMM4 contains 0.0, or XMM3 contains 0.0 and XMM4 holds `val * val`. These register values facilitate use of a `vorpd xmm0,xmm4,xmm3` instruction to load XMM0 with the final value for `c[i]`.

Like the previous source code example, the function Avx512CalcValues_ demonstrates an important capability of AVX-512. The use of zero masking and some simple Boolean logic allowed Avx512CalcValues_ to make logical decisions sans any conditional jump instructions. This is noteworthy since data-dependent conditional jump instructions are often slower than straight-line code. Here is the output for source code example Ch13_02:

```
i:   0  a:    -6.0000  b:    67.0000  c1:   161604.0000  c2:   161604.0000
i:   1  a:   128.0000  b:    22.0000  c1:       53.0660  c2:       53.0660
i:   2  a:   130.0000  b:    -8.0000  c1:  1081600.0000  c2:  1081600.0000
i:   3  a:   152.0000  b:    73.0000  c1:      105.3376  c2:      105.3376
i:   4  a:    94.0000  b:     6.0000  c1:       23.7487  c2:       23.7487
i:   5  a:     2.0000  b:    88.0000  c1:       13.2665  c2:       13.2665
i:   6  a:    12.0000  b:   103.0000  c1:       35.1568  c2:       35.1568
i:   7  a:   105.0000  b:   117.0000  c1:      110.8377  c2:      110.8377
i:   8  a:   140.0000  b:   -20.0000  c1:  7840000.0000  c2:  7840000.0000
i:   9  a:    74.0000  b:     3.0000  c1:       14.8997  c2:       14.8997
i:  10  a:    43.0000  b:    -9.0000  c1:   149769.0000  c2:   149769.0000
i:  11  a:     2.0000  b:   122.0000  c1:       15.6205  c2:       15.6205
i:  12  a:    36.0000  b:     9.0000  c1:       18.0000  c2:       18.0000
i:  13  a:   -18.0000  b:   123.0000  c1:  4901796.0000  c2:  4901796.0000
i:  14  a:   170.0000  b:   134.0000  c1:      150.9304  c2:      150.9304
i:  15  a:   102.0000  b:     3.0000  c1:       17.4929  c2:       17.4929
i:  16  a:   118.0000  b:   -19.0000  c1:  5026564.0000  c2:  5026564.0000
i:  17  a:    85.0000  b:   148.0000  c1:      112.1606  c2:      112.1606
i:  18  a:    61.0000  b:    65.0000  c1:       62.9682  c2:       62.9682
i:  19  a:    18.0000  b:    74.0000  c1:       36.4966  c2:       36.4966
```

## Instruction-Level Rounding

The final source code example of this section, Ch13_03, explains how to use instruction-level rounding operands. It also illustrates use of the AVX-512 instructions that perform conversions between floating-point and unsigned integer values. Listing 13-3 shows the source code for example Ch13_03.

***Listing 13-3.*** Example Ch13_03

```
//------------------------------------------------
//               Ch13_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

extern "C" void Avx512CvtF32ToU32_(uint32_t val_cvt[4], float val);
extern "C" void Avx512CvtF64ToU64_(uint64_t val_cvt[4], double val);
```

```
extern "C" void Avx512CvtF64ToF32_(float val_cvt[4], double val);

void ConvertF32ToU32(void)
{
    uint32_t val_cvt[4];
    const float val[] {(float)M_PI, (float)M_SQRT2};
    const int num_vals = sizeof(val) / sizeof(float);

    cout << "\nConvertF32ToU32\n";

    for (int i = 0; i < num_vals; i++)
    {
        Avx512CvtF32ToU32_(val_cvt, val[i]);

        cout << "  Test case #" << i << " val = " << val[i] << '\n';
        cout << "    val_cvt[0] {rn-sae} = " << val_cvt[0] << '\n';
        cout << "    val_cvt[1] {rd-sae} = " << val_cvt[1] << '\n';
        cout << "    val_cvt[2] {ru-sae} = " << val_cvt[2] << '\n';
        cout << "    val_cvt[3] {rz-sae} = " << val_cvt[3] << '\n';
    }
}

void ConvertF64ToU64(void)
{
    uint64_t val_cvt[4];
    const double val[] {(float)M_PI, (float)M_SQRT2};
    const int num_vals = sizeof(val) / sizeof(double);

    cout << "\nConvertF64ToU64\n";

    for (int i = 0; i < num_vals; i++)
    {
        Avx512CvtF64ToU64_(val_cvt, val[i]);

        cout << "  Test case #" << i << " val = " << val[i] << '\n';
        cout << "    val_cvt[0] {rn-sae} = " << val_cvt[0] << '\n';
        cout << "    val_cvt[1] {rd-sae} = " << val_cvt[1] << '\n';
        cout << "    val_cvt[2] {ru-sae} = " << val_cvt[2] << '\n';
        cout << "    val_cvt[3] {rz-sae} = " << val_cvt[3] << '\n';
    }
}

void ConvertF64ToF32(void)
{
    float val_cvt[4];
    const double val[] {M_PI, -M_SQRT2};
    const int num_vals = sizeof(val) / sizeof(double);

    cout << "\nConvertF64ToF32\n";

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

```
    {
        Avx512CvtF64ToF32_(val_cvt, val[i]);

        cout << fixed << setprecision(7);

        cout << "  Test case #" << i << " val = " << val[i] << '\n';
        cout << "    val_cvt[0] {rn-sae} = " << val_cvt[0] << '\n';
        cout << "    val_cvt[1] {rd-sae} = " << val_cvt[1] << '\n';
        cout << "    val_cvt[2] {ru-sae} = " << val_cvt[2] << '\n';
        cout << "    val_cvt[3] {rz-sae} = " << val_cvt[3] << '\n';
    }
}

int main()
{
    ConvertF32ToU32();
    ConvertF64ToU64();
    ConvertF64ToF32();
    return 0;
}

;-------------------------------------------------
;               Ch13_03.asm
;-------------------------------------------------

; extern "C" void Avx512CvtF32ToU32_(uint32_t val_cvt[4], float val);

        .code
Avx512CvtF32ToU32_ proc
        vcvtss2usi eax,xmm1{rn-sae}         ;Convert using round to nearest
        mov dword ptr [rcx],eax

        vcvtss2usi eax,xmm1{rd-sae}         ;Convert using round down
        mov dword ptr [rcx+4],eax

        vcvtss2usi eax,xmm1{ru-sae}         ;Convert using round up
        mov dword ptr [rcx+8],eax

        vcvtss2usi eax,xmm1{rz-sae}         ;Convert using round to zero (truncate)
        mov dword ptr [rcx+12],eax
        ret
Avx512CvtF32ToU32_ endp

; extern "C" void Avx512CvtF64ToU64_(uint64_t val_cvt[4], double val);

Avx512CvtF64ToU64_ proc
        vcvtsd2usi rax,xmm1{rn-sae}
        mov qword ptr [rcx],rax

        vcvtsd2usi rax,xmm1{rd-sae}
        mov qword ptr [rcx+8],rax
```

```
        vcvtsd2usi rax,xmm1{ru-sae}
        mov qword ptr [rcx+16],rax

        vcvtsd2usi rax,xmm1{rz-sae}
        mov qword ptr [rcx+24],rax
        ret
Avx512CvtF64ToU64_ endp

; extern "C" void Avx512CvtF64ToF32_(float val_cvt[4], double val);

Avx512CvtF64ToF32_ proc
        vcvtsd2ss xmm2,xmm2,xmm1{rn-sae}
        vmovss real4 ptr [rcx],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{rd-sae}
        vmovss real4 ptr [rcx+4],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{ru-sae}
        vmovss real4 ptr [rcx+8],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{rz-sae}
        vmovss real4 ptr [rcx+12],xmm2
        ret
Avx512CvtF64ToF32_ endp
        end
```

The C++ code in Listing 13-3 begins with the function ConvertF32ToU32. This function performs test case initialization and exercises the assembly language function Avx512CvtF32ToU32_, which converts a single-precision floating-point value to an unsigned doubleword (32-bit) integer using different rounding modes. The results are then streamed to cout. The C++ functions ConvertF64ToU64 and ConvertF64ToF32 carry out similar test case initializations for the assembly language functions Avx512CvtF64ToU64_ and Avx512CvtF64ToF32_, respectively.

The first instruction of assembly language function Avx512CvtF32ToU32_, vcvtss2usi eax,xmm1{rn-sae} converts the scalar single-precision floating-point value in XMM1 (or val) to an unsigned doubleword integer using the rounding mode round-to-nearest. As mentioned in Chapter 12, the -sae suffix that's appended to the embedded rounding mode string is a reminder that floating-point exceptions and MXCSR flag updates are always disabled when an instruction-level rounding control operand is specified. The ensuing mov dword ptr [rcx],eax instruction saves the converted result in val_cvt[0]. Avx512CvtF32ToU32_ and then employs additional vcvtss2usi instructions to carry out the same conversion operation using rounding modes round-down, round-up, and round-to-zero. The organization of function Avx512CvtF64ToU64_ is similar to Avx512CvtF32ToU32_ and uses the vcvtsd2usi instruction to convert a double-precision floating-point value to an unsigned quadword integer. Note that both vcvtss2usi and vcvtsd2usi are new AVX-512 instructions. AVX-512 also includes the instructions vcvtusi2s[d|s], which perform unsigned integer to floating-point conversions. Neither AVX nor AVX2 include instructions that perform these types of conversions.

The final assembly language function, `Avx512CvtF64ToF32_`, applies the `vcvtsd2ss` instruction to convert a double-precision floating-point value to single-precision floating-point. The `vcvtsd2ss` instruction is an existing AVX instruction that can be used with an instruction-level rounding control operand on systems that support AVX-512. Here is the output for source code example `Ch13_03`.

```
ConvertF32ToU32
  Test case #0 val = 3.14159
    val_cvt[0] {rn-sae} = 3
    val_cvt[1] {rd-sae} = 3
    val_cvt[2] {ru-sae} = 4
    val_cvt[3] {rz-sae} = 3
  Test case #1 val = 1.41421
    val_cvt[0] {rn-sae} = 1
    val_cvt[1] {rd-sae} = 1
    val_cvt[2] {ru-sae} = 2
    val_cvt[3] {rz-sae} = 1

ConvertF64ToU64
  Test case #0 val = 3.14159
    val_cvt[0] {rn-sae} = 3
    val_cvt[1] {rd-sae} = 3
    val_cvt[2] {ru-sae} = 4
    val_cvt[3] {rz-sae} = 3
  Test case #1 val = 1.41421
    val_cvt[0] {rn-sae} = 1
    val_cvt[1] {rd-sae} = 1
    val_cvt[2] {ru-sae} = 2
    val_cvt[3] {rz-sae} = 1

ConvertF64ToF32
  Test case #0 val = 3.1415927
    val_cvt[0] {rn-sae} = 3.1415927
    val_cvt[1] {rd-sae} = 3.1415925
    val_cvt[2] {ru-sae} = 3.1415927
    val_cvt[3] {rz-sae} = 3.1415925
  Test case #1 val = -1.4142136
    val_cvt[0] {rn-sae} = -1.4142135
    val_cvt[1] {rd-sae} = -1.4142137
    val_cvt[2] {ru-sae} = -1.4142135
    val_cvt[3] {rz-sae} = -1.4142135
```

# Packed Floating-Point

The source code examples of this section illustrate how to use AVX-512 instructions to carry out computations using packed floating-point operands. The first three source code examples demonstrate basic operations with 512-bit wide packed floating-point operands including simple arithmetic, compare operations, and merge masking. The remaining examples focus on specific algorithms including vector cross product calculations, matrix-vector multiplications, and convolutions.

## Packed Floating-Point Arithmetic

Listing 13-4 shows the source code for example Ch13_04. This example demonstrates how to perform common arithmetic operations using 512-bit wide single-precision and double-precision floating-point operands. It also highlights some of the similarities between AVX/AVX2 and AVX-512 programming.

***Listing 13-4.*** Example Ch13_04

```
//------------------------------------------------
//               ZmmVal.h
//------------------------------------------------

#pragma once
#include <string>
#include <cstdint>
#include <sstream>
#include <iomanip>

struct ZmmVal
{
public:
    union
    {
        int8_t m_I8[64];
        int16_t m_I16[32];
        int32_t m_I32[16];
        int64_t m_I64[8];
        uint8_t m_U8[64];
        uint16_t m_U16[32];
        uint32_t m_U32[16];
        uint64_t m_U64[8];
        float m_F32[16];
        double m_F64[8];
    };

//------------------------------------------------
//               Ch13_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>
#include "ZmmVal.h"

using namespace std;

extern "C" void Avx512PackedMathF32_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);
extern "C" void Avx512PackedMathF64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

void Avx512PackedMathF32(void)
```

```
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    alignas(64) ZmmVal c[8];

    a.m_F32[0] = 36.0f;                 b.m_F32[0] = -0.1111111f;
    a.m_F32[1] = 0.03125f;              b.m_F32[1] = 64.0f;
    a.m_F32[2] = 2.0f;                  b.m_F32[2] = -0.0625f;
    a.m_F32[3] = 42.0f;                 b.m_F32[3] = 8.666667f;

    a.m_F32[4] = 7.0f;                  b.m_F32[4] = -18.125f;
    a.m_F32[5] = 20.5f;                 b.m_F32[5] = 56.0f;
    a.m_F32[6] = 36.125f;               b.m_F32[6] = 24.0f;
    a.m_F32[7] = 0.5f;                  b.m_F32[7] = -158.6f;

    a.m_F32[8] = 136.0f;                b.m_F32[8] = -9.1111111f;
    a.m_F32[9] = 2.03125f;              b.m_F32[9] = 864.0f;
    a.m_F32[10] = 32.0f;                b.m_F32[10] = -70.0625f;
    a.m_F32[11] = 442.0f;               b.m_F32[11] = 98.666667f;

    a.m_F32[12] = 57.0f;                b.m_F32[12] = -518.125f;
    a.m_F32[13] = 620.5f;               b.m_F32[13] = 456.0f;
    a.m_F32[14] = 736.125f;             b.m_F32[14] = 324.0f;
    a.m_F32[15] = 80.5f;                b.m_F32[15] = -298.6f;

    Avx512PackedMathF32_(&a, &b, c);

    cout << ("\nResults for Avx512PackedMathF32\n");

    for (int i = 0; i < 4; i++)
    {
        cout << "Group #" << i << '\n';

        cout << "  a:       " << a.ToStringF32(i) << '\n';
        cout << "  b:       " << b.ToStringF32(i) << '\n';
        cout << "  addps:   " << c[0].ToStringF32(i) << '\n';
        cout << "  subps:   " << c[1].ToStringF32(i) << '\n';
        cout << "  mulps:   " << c[2].ToStringF32(i) << '\n';
        cout << "  divps:   " << c[3].ToStringF32(i) << '\n';
        cout << "  absps:   " << c[4].ToStringF32(i) << '\n';
        cout << "  sqrtps:  " << c[5].ToStringF32(i) << '\n';
        cout << "  minps:   " << c[6].ToStringF32(i) << '\n';
        cout << "  maxps:   " << c[7].ToStringF32(i) << '\n';

        cout << '\n';
    }
}
```

```
void Avx512PackedMathF64(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    alignas(64) ZmmVal c[8];

    a.m_F64[0] = 2.0;           b.m_F64[0] = M_PI;
    a.m_F64[1] = 4.0 ;          b.m_F64[1] = M_E;

    a.m_F64[2] = 7.5;           b.m_F64[2] = -9.125;
    a.m_F64[3] = 3.0;           b.m_F64[3] = -M_PI;

    a.m_F64[4] = 12.0;          b.m_F64[4] = M_PI / 2;
    a.m_F64[5] = 24.0;          b.m_F64[5] = M_E / 2;

    a.m_F64[6] = 37.5;          b.m_F64[6] = -9.125 / 2;
    a.m_F64[7] = 43.0;          b.m_F64[7] = -M_PI / 2;

    Avx512PackedMathF64_(&a, &b, c);
    cout << ("\nResults for Avx512PackedMathF64\n");

    for (int i = 0; i < 4; i++)
    {
        cout << "Group #" << i << '\n';

        cout << "  a:       " << a.ToStringF64(i) << '\n';
        cout << "  b:       " << b.ToStringF64(i) << '\n';
        cout << "  addpd:   " << c[0].ToStringF64(i) << '\n';
        cout << "  subpd:   " << c[1].ToStringF64(i) << '\n';
        cout << "  mulpd:   " << c[2].ToStringF64(i) << '\n';
        cout << "  divpd:   " << c[3].ToStringF64(i) << '\n';
        cout << "  abspd:   " << c[4].ToStringF64(i) << '\n';
        cout << "  sqrtpd:  " << c[5].ToStringF64(i) << '\n';
        cout << "  minpd:   " << c[6].ToStringF64(i) << '\n';
        cout << "  maxpd:   " << c[7].ToStringF64(i) << '\n';

        cout << '\n';
    }
}

int main()
{
    Avx512PackedMathF32();
    Avx512PackedMathF64();
    return 0;
}

;-------------------------------------------------
;               Ch13_04.asm
;-------------------------------------------------
```

```
; Mask values used to calculate floating-point absolute values
ConstVals   segment readonly align(64) 'const'
AbsMaskF32  dword 16 dup(7fffffffh)
AbsMaskF64  qword 8 dup(7fffffffffffffffh)
ConstVals   ends

; extern "C" void Avx512PackedMathF32_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

            .code
Avx512PackedMathF32_ proc

; Load packed SP floating-point values
        vmovaps zmm0,zmmword ptr [rcx]      ;zmm0 = *a
        vmovaps zmm1,zmmword ptr [rdx]      ;zmm1 = *b

; Packed SP floating-point addition
        vaddps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+0],zmm2

; Packed SP floating-point subtraction
        vsubps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+64],zmm2

; Packed SP floating-point multiplication
        vmulps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+128],zmm2

; Packed SP floating-point division
        vdivps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+192],zmm2

; Packed SP floating-point absolute value (b)
        vandps zmm2,zmm1,zmmword ptr [AbsMaskF32]
        vmovaps zmmword ptr [r8+256],zmm2

; Packed SP floating-point square root (a)
        vsqrtps zmm2,zmm0
        vmovaps zmmword ptr [r8+320],zmm2

; Packed SP floating-point minimum
        vminps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+384],zmm2

; Packed SP floating-point maximum
        vmaxps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+448],zmm2

        vzeroupper
        ret
Avx512PackedMathF32_ endp
```

```
; extern "C" void Avx512PackedMathF64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

Avx512PackedMathF64_ proc

; Load packed DP floating-point values
        vmovapd zmm0,zmmword ptr [rcx]       ;zmm0 = *a
        vmovapd zmm1,zmmword ptr [rdx]       ;zmm1 = *b

; Packed DP floating-point addition
        vaddpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+0],zmm2

; Packed DP floating-point subtraction
        vsubpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+64],zmm2

; Packed DP floating-point multiplication
        vmulpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+128],zmm2

; Packed DP floating-point division
        vdivpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+192],zmm2

; Packed DP floating-point absolute value (b)
        vandpd zmm2,zmm1,zmmword ptr [AbsMaskF64]
        vmovapd zmmword ptr [r8+256],zmm2

; Packed DP floating-point square root (a)
        vsqrtpd zmm2,zmm0
        vmovapd zmmword ptr [r8+320],zmm2

; Packed DP floating-point minimum
        vminpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+384],zmm2

; Packed DP floating-point maximum
        vmaxpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+448],zmm2

        vzeroupper
        ret
Avx512PackedMathF64_ endp
        end
```

Listing 13-4 starts with the declaration of the C++ structure `ZmmVal`, which is declared in the header file `ZmmVal.h`. This structure is analogous to the `XmmVal` and `YmmVal` structures that were used by the source code examples in Chapters 6 and 9. The structure `ZmmVal` contains a publicly-accessible anonymous union that simplifies packed operand data exchange between functions written in C++ and the x86 assembly language. The members of this union correspond to the packed data types that can be used with a ZMM register. The structure `ZmmVal` also includes several string formatting functions for display purposes (the source code for these member functions is not shown).

The remaining C++ code in Listing 13-4 is similar to the code that was used in example Ch09_01. The declarations for assembly language functions Avx512PackedMathF32_ and Avx512PackedMathF64_ follow the declaration of structure ZmmVal. These functions carry out various packed single-precision and double-precision floating-point arithmetic operations using the supplied ZmmVal arguments. The C++ functions Avx512PackedMathF32 and Avx512PackedMathF64 perform ZmmVal variable initializations, invoke the assembly language calculating functions, and display results. Note that the alignas(64) specifier is used with each ZmmVal variable definition.

The assembly language code in Listing 13-4 begins with a 64-byte aligned custom memory segment named ConstVals. This segment contains definitions for the packed constant values that are used in the calculating functions. A custom segment is used here since the MASM align directive does not support aligning data items on a 64-byte boundary. Chapter 9 contains additional information about custom memory segments. The segment ConstVals contains the constants AbsMaskF32 and AbsMaskF64, which are used to calculate absolute values for 512-bit wide packed single-precision and double-precision floating-point values.

The first instruction of Avx512PackedMathF32_, vmovaps zmm0,ymmword ptr [rcx], loads argument a (the 16 floating-point values in ZmmVal a) into register YMM0. The vmovaps can be used here since ZmmVal a was defined using the alignas(64) specifier. The operator zmmword ptr directs the assembler to treat the memory location pointed to by RCX as a 512-bit wide operand. Like the operators xmmword ptr and ymmword ptr, the zmmword ptr operator is often used to improve code readability even when it's not explicitly required. The ensuing vmovaps zmm1,zmmword ptr [rdx] instruction loads ZmmVal b into register ZMM1. The vaddps zmm2,zmm0,zmm1 instruction that follows sums the packed single-precision floating-point values in ZMM0 and ZMM1 and saves the result in ZMM2. The vmovaps zmmword ptr [r8],zmm2 instruction saves the packed sums to c[0].

The ensuing vsubps, vmulps, and vdivps instructions carry out packed single-precision floating-point subtraction, multiplication, and division. This is followed by a vandps zmm2,zmm1,zmmword ptr [AbsMaskF32] instruction that calculates packed absolute values using argument b. The remaining instructions in Avx512PackedMathF32_ calculate packed single-precision floating-point square roots, minimums, and maximums.

Prior to its ret instruction, the function AvxPackedMath32_ uses a vzeroupper instruction, which zeros the high-order 384 bits of registers ZMM0–ZMM15. As explained in Chapter 4, a vzeroupper instruction is used here to avoid potential performance delays that can occur whenever the processor transitions from executing x86-AVX code to executing x86-SSE code. Any assembly language function that uses one or more YMM or ZMM registers and is callable from code that potentially uses x86-SSE instructions should ensure that a vzeroupper instruction is executed before program control is transferred back to the calling function. It should be noted that according to the *Intel 64 and IA-32 Architectures Optimization Reference Manual*, the vzeroupper use recommendations apply to functions that employ x86-AVX instructions with registers ZMM0–ZMM15 or YMM0–YMM15. Functions that only exploit registers ZMM16–ZMM31 or YMM16–YMM31 do not need to observe the vzeroupper use recommendations.

The organization of function Avx512PackedMathF64_ is similar to Avx512PackedMathF32_. Avx512PackedMathF64_ carries out its calculations using the double-precision versions of the same AVX-512 instructions that are used in Avx512PackedMathF32_. Here is the output for source code example Ch13_04:

```
Results for Avx512PackedMathF32

Group #0
  a:             36.000000      0.031250   |     2.000000     42.000000
  b:             -0.111111     64.000000   |    -0.062500      8.666667
  addps:         35.888889     64.031250   |     1.937500     50.666668
  subps:         36.111111    -63.968750   |     2.062500     33.333332
  mulps:         -4.000000      2.000000   |    -0.125000    364.000000
  divps:       -324.000031      0.000488   |   -32.000000      4.846154
```

```
  absps:         0.111111      64.000000   |      0.062500       8.666667
  sqrtps:        6.000000       0.176777   |      1.414214       6.480741
  minps:        -0.111111       0.031250   |     -0.062500       8.666667
  maxps:        36.000000      64.000000   |      2.000000      42.000000

Group #1
  a:             7.000000      20.500000   |     36.125000       0.500000
  b:           -18.125000      56.000000   |     24.000000    -158.600006
  addps:       -11.125000      76.500000   |     60.125000    -158.100006
  subps:        25.125000     -35.500000   |     12.125000     159.100006
  mulps:      -126.875000    1148.000000   |    867.000000     -79.300003
  divps:        -0.386207       0.366071   |      1.505208      -0.003153
  absps:        18.125000      56.000000   |     24.000000     158.600006
  sqrtps:        2.645751       4.527693   |      6.010407       0.707107
  minps:       -18.125000      20.500000   |     24.000000    -158.600006
  maxps:         7.000000      56.000000   |     36.125000       0.500000

Group #2
  a:           136.000000       2.031250   |     32.000000     442.000000
  b:            -9.111111     864.000000   |    -70.062500      98.666664
  addps:       126.888885     866.031250   |    -38.062500     540.666687
  subps:       145.111115    -861.968750   |    102.062500     343.333344
  mulps:     -1239.111084    1755.000000   |  -2242.000000   43610.664063
  divps:       -14.926830       0.002351   |     -0.456735       4.479730
  absps:         9.111111     864.000000   |     70.062500      98.666664
  sqrtps:       11.661903       1.425219   |      5.656854      21.023796
  minps:        -9.111111       2.031250   |    -70.062500      98.666664
  maxps:       136.000000     864.000000   |     32.000000     442.000000

Group #3
  a:            57.000000     620.500000   |    736.125000      80.500000
  b:          -518.125000     456.000000   |    324.000000    -298.600006
  addps:      -461.125000    1076.500000   |   1060.125000    -218.100006
  subps:       575.125000     164.500000   |    412.125000     379.100006
  mulps:    -29533.125000  282948.000000   | 238504.500000  -24037.300781
  divps:        -0.110012       1.360746   |      2.271991      -0.269591
  absps:       518.125000     456.000000   |    324.000000     298.600006
  sqrtps:        7.549834      24.909838   |     27.131624       8.972179
  minps:      -518.125000     456.000000   |    324.000000    -298.600006
  maxps:        57.000000     620.500000   |    736.125000      80.500000

Results for Avx512PackedMathF64

Group #0
  a:                      2.000000000000   |                4.000000000000
  b:                      3.141592653590   |                2.718281828459
  addpd:                  5.141592653590   |                6.718281828459
  subpd:                 -1.141592653590   |                1.281718171541
  mulpd:                  6.283185307180   |               10.873127313836
  divpd:                  0.636619772368   |                1.471517764686
  abspd:                  3.141592653590   |                2.718281828459
```

```
  sqrtpd:               1.414213562373   |          2.000000000000
  minpd:                2.000000000000   |          2.718281828459
  maxpd:                3.141592653590   |          4.000000000000

Group #1
  a:                    7.500000000000   |          3.000000000000
  b:                   -9.125000000000   |         -3.141592653590
  addpd:               -1.625000000000   |         -0.141592653590
  subpd:               16.625000000000   |          6.141592653590
  mulpd:              -68.437500000000   |         -9.424777960769
  divpd:               -0.821917808219   |         -0.954929658551
  abspd:                9.125000000000   |          3.141592653590
  sqrtpd:               2.738612787526   |          1.732050807569
  minpd:               -9.125000000000   |         -3.141592653590
  maxpd:                7.500000000000   |          3.000000000000

Group #2
  a:                   12.000000000000   |         24.000000000000
  b:                    1.570796326795   |          1.359140914230
  addpd:               13.570796326795   |         25.359140914230
  subpd:               10.429203673205   |         22.640859085770
  mulpd:               18.849555921539   |         32.619381941509
  divpd:                7.639437268411   |         17.658213176229
  abspd:                1.570796326795   |          1.359140914230
  sqrtpd:               3.464101615138   |          4.898979485566
  minpd:                1.570796326795   |          1.359140914230
  maxpd:               12.000000000000   |         24.000000000000

Group #3
  a:                   37.500000000000   |         43.000000000000
  b:                   -4.562500000000   |         -1.570796326795
  addpd:               32.937500000000   |         41.429203673205
  subpd:               42.062500000000   |         44.570796326795
  mulpd:             -171.093750000000   |        -67.544242052181
  divpd:               -8.219178082192   |        -27.374650211806
  abspd:                4.562500000000   |          1.570796326795
  sqrtpd:               6.123724356958   |          6.557438524302
  minpd:               -4.562500000000   |         -1.570796326795
  maxpd:               37.500000000000   |         43.000000000000
```

## Packed Floating-Point Compares

In Chapter 6 you learned how to use the vcmpp[s|d] instructions to perform packed single-precision and double-precision floating-point compare operations (see source code example Ch06_02). Recall that the AVX version of these instructions set the elements of a SIMD operand to all zeros or all ones to indicate the result of a compare operation. In this section, you learn how to use the AVX-512 version of the vcmpps instruction, which saves its compare result in an opmask register. Listing 13-5 shows the C++ and assembly language code for example Ch13_05.

***Listing 13-5.*** Example Ch13_05

```
//------------------------------------------------
//              Ch13_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <limits>
#include "ZmmVal.h"

using namespace std;

extern "C" void Avx512PackedCompareF32_(const ZmmVal* a, const ZmmVal* b, uint16_t c[8]);

const char* c_CmpStr[8] =
{
    "EQ", "NE", "LT", "LE", "GT", "GE", "ORDERED", "UNORDERED"
};

void ToZmmVal(ZmmVal des[8], uint16_t src[8])
{
    for (size_t i = 0; i < 8; i++)
    {
        uint16_t val_src = src[i];

        for (size_t j = 0; j < 16; j++)
            des[i].m_U32[j] = val_src & (1 << j) ? 1 : 0;
    }
}

void Avx512PackedCompareF32(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    uint16_t c[8];

    a.m_F32[0] = 2.0;       b.m_F32[0] = 1.0;
    a.m_F32[1] = 7.0;       b.m_F32[1] = 12.0;
    a.m_F32[2] = -6.0;      b.m_F32[2] = -6.0;
    a.m_F32[3] = 3.0;       b.m_F32[3] = 8.0;

    a.m_F32[4] = -2.0;      b.m_F32[4] = 1.0;
    a.m_F32[5] = 17.0;      b.m_F32[5] = 17.0;
    a.m_F32[6] = 6.5;       b.m_F32[6] = -9.125;
    a.m_F32[7] = 4.875;     b.m_F32[7] = numeric_limits<float>::quiet_NaN();

    a.m_F32[8] = 2.0;       b.m_F32[8] = 101.0;
    a.m_F32[9] = 7.0;       b.m_F32[9] = -312.0;
```

```
    a.m_F32[10] = -5.0;     b.m_F32[10] = 15.0;
    a.m_F32[11] = -33.0;    b.m_F32[11] = -33.0;

    a.m_F32[12] = -12.0;    b.m_F32[12] = 198.0;
    a.m_F32[13] = 107.0;    b.m_F32[13] = 107.0;
    a.m_F32[14] = 16.125;   b.m_F32[14] = -2.75;
    a.m_F32[15] = 42.875;   b.m_F32[15] = numeric_limits<float>::quiet_NaN();

    Avx512PackedCompareF32_(&a, &b, c);

    cout << "\nResults for Avx512PackedCompareF32\n";

    ZmmVal c_display[8];

    ToZmmVal(c_display, c);

    for (int sel = 0; sel < 4; sel++)
    {
        cout << setw(12) << "a[" << sel << "]:" << a.ToStringF32(sel) << '\n';
        cout << setw(12) << "b[" << sel << "]:" << b.ToStringF32(sel) << '\n';
        cout << '\n';

        for (int j = 0; j < 8; j++)
            cout << setw(14) << c_CmpStr[j] << ':' << c_display[j].ToStringU32(sel)  <<
'\n';
        cout << '\n';
    }
}

int main()
{
    Avx512PackedCompareF32();
    return 0;
}

;-------------------------------------------------
;               Ch13_05.asm
;-------------------------------------------------

        include <cmpequ.asmh>

; extern "C" void Avx512PackedCompareF32_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

        .code
Avx512PackedCompareF32_ proc
        vmovaps zmm0,[rcx]                  ;zmm0 = a
        vmovaps zmm1,[rdx]                  ;zmm1 = b

; Perform packed EQUAL compare
        vcmpps k1,zmm0,zmm1,CMP_EQ
        kmovw word ptr [r8],k1
```

```
; Perform packed NOT EQUAL compare
        vcmpps k1,zmm0,zmm1,CMP_NEQ
        kmovw word ptr [r8+2],k1

; Perform packed LESS THAN compare
        vcmpps k1,zmm0,zmm1,CMP_LT
        kmovw word ptr [r8+4],k1

; Perform packed LESS THAN OR EQUAL compare
        vcmpps k1,zmm0,zmm1,CMP_LE
        kmovw word ptr [r8+6],k1

; Perform packed GREATER THAN compare
        vcmpps k1,zmm0,zmm1,CMP_GT
        kmovw word ptr [r8+8],k1

; Perform packed GREATER THAN OR EQUAL compare
        vcmpps k1,zmm0,zmm1,CMP_GE
        kmovw word ptr [r8+10],k1

; Perform packed ORDERED compare
        vcmpps k1,zmm0,zmm1,CMP_ORD
        kmovw word ptr [r8+12],k1

; Perform packed UNORDERED compare
        vcmpps k1,zmm0,zmm1,CMP_UNORD
        kmovw word ptr [r8+14],k1

        vzeroupper
        ret
Avx512PackedCompareF32_ endp
        end
```

The C++ function Avx512PackedCompareF32 that's shown in Listing 13-5 starts its execution by loading test values into the single-precision floating-point elements of ZmmVal variables a and b. Note that these variables are defined using the C++ alignas(64) specifier. Following variable initialization, the function Avx512PackedCompareF32 invokes the assembly language function Avx512PackedCompareF32_ to perform the packed compares. It then streams the results to cout.

The assembly language function Avx512PackedCompareF32_ begins its execution with two vmovaps instructions that load ZmmVal variables a and b into registers ZMM0 and ZMM1, respectively. The ensuing vcmpps k1,zmm0,zmm1,CMP_EQ instruction compares the single-precision floating-point elements in registers ZMM0 and ZMM1 for equality. For each element position, this instruction sets the corresponding bit position in opmask register K1 to one if the values in ZMM0 and ZMM1 are equal; otherwise, the opmask register bit is set to zero. Figure 13-1 illustrates this operation in greater detail. The kmovw word ptr [r8],k1 instruction that follows saves the resultant mask to c[0].

Initial values

| 39.0 | 98.0 | 66.0 | 54.0 | 95.0 | 19.0 | 56.0 | 40.0 | 90.0 | 41.0 | 21.0 | 11.0 | 35.0 | 48.0 | 83.0 | 19.0 | zmm0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4.0 | 98.0 | 48.0 | 54.0 | 95.0 | 19.0 | 33.0 | 88.0 | 47.0 | 41.0 | 56.0 | 22.0 | 35.0 | 90.0 | 83.0 | 36.0 | zmm1 |

vcmpps k1,zmm0,zmm1,CMP_EQ

| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | k1 (bits 15:0) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

***Figure 13-1.*** *Example execution of the `vcmpps k1,zmm0,zmm1,CMP_EQ` instruction*

The remaining code in `Avx512PackedCompareF32_` performs additional compare operations using the `vcmpps` instruction, `ZmmVal` variables `a` and `b`, and common compare predicates. Note that like the previous example, `Avx512PackedCompareF32_` uses a `vzeroupper` instruction prior to its `ret` instruction. Here are the results for source code example `Ch13_05`.

```
Results for Avx512PackedCompareF32

        a[0]:        2.000000        7.000000 |       -6.000000        3.000000
        b[0]:        1.000000       12.000000 |       -6.000000        8.000000

          EQ:               0               0 |               1               0
          NE:               1               1 |               0               1
          LT:               0               1 |               0               1
          LE:               0               1 |               1               1
          GT:               1               0 |               0               0
          GE:               1               0 |               1               0
     ORDERED:               1               1 |               1               1
   UNORDERED:               0               0 |               0               0

        a[1]:       -2.000000       17.000000 |        6.500000        4.875000
        b[1]:        1.000000       17.000000 |       -9.125000             nan

          EQ:               0               1 |               0               0
          NE:               1               0 |               1               1
          LT:               1               0 |               0               0
          LE:               1               1 |               0               0
          GT:               0               0 |               1               0
          GE:               0               1 |               1               0
     ORDERED:               1               1 |               1               0
   UNORDERED:               0               0 |               0               1

        a[2]:        2.000000        7.000000 |       -5.000000      -33.000000
        b[2]:      101.000000     -312.000000 |       15.000000      -33.000000

          EQ:               0               0 |               0               1
          NE:               1               1 |               1               0
          LT:               1               0 |               1               0
          LE:               1               0 |               1               1
```

```
       GT:           0           1    |          0           0
       GE:           0           1    |          0           1
  ORDERED:           1           1    |          1           1
UNORDERED:           0           0    |          0           0

     a[3]:  -12.000000  107.000000    |  16.125000   42.875000
     b[3]:  198.000000  107.000000    |  -2.750000         nan

       EQ:           0           1    |          0           0
       NE:           1           0    |          1           1
       LT:           1           0    |          0           0
       LE:           1           1    |          0           0
       GT:           0           0    |          1           0
       GE:           0           1    |          1           0
  ORDERED:           1           1    |          1           0
UNORDERED:           0           0    |          0           1
```

On systems that support AVX-512, assembly language functions can also use the vcmppd instruction with a destination operand opmask register to perform packed double-precision floating-point compares. In these instances, the resultant mask is saved in the low-order eight bits of the destination operand opmask register.

## Packed Floating-Point Column Means

Listing 13-6 shows the source code for example Ch13_06. This example, which is an AVX-512 implementation of source code example Ch09_03, calculates columns means for a two-dimensional array of double-precision floating-point values. To make the current source code example a little more interesting, the column means are calculated using only the array elements that are above a predetermined threshold value.

***Listing 13-6.*** Example Ch13_06

```
//------------------------------------------------
//               Ch13_06.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <memory>

using namespace std;

// Test size limits to illustrate argument checking
extern "C" size_t c_NumRowsMax = 1000000;
extern "C" size_t c_NumColsMax = 1000000;

extern "C" bool Avx512CalcColumnMeans_(const double* x, size_t nrows, size_t ncols, double*
col_means, size_t* col_counts, double x_min);
```

```
void Init(double* x, size_t n, int rng_min, int rng_max, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {rng_min, rng_max};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (double)ui_dist(rng);
}

bool Avx512CalcColumnMeansCpp(const double* x, size_t nrows, size_t ncols, double* col_
means, size_t* col_counts, double x_min)
{
    // Make sure nrows and ncols are valid
    if (nrows == 0 || nrows > c_NumRowsMax)
        return false;
    if (ncols == 0 || ncols > c_NumColsMax)
        return false;

    // Initialize column means and column counts to zero
    for (size_t i = 0; i < ncols; i++)
    {
        col_means[i] = 0.0;
        col_counts[i] = 0;
    }

    // Calculate column means
    for (size_t i = 0; i < nrows; i++)
    {
        for (size_t j = 0; j < ncols; j++)
        {
            double val = x[i * ncols + j];

            if (val >= x_min)
            {
                col_means[j] += val;
                col_counts[j]++;
            }
        }
    }

    for (size_t j = 0; j < ncols; j++)
        col_means[j] /= col_counts[j];

    return true;
}

void Avx512CalcColumnMeans(void)
{
    const size_t nrows = 20000;
    const size_t ncols = 23;
    const int rng_min = 1;
```

```
    const int rng_max = 999;
    const unsigned int rng_seed = 47;
    const double x_min = 75.0;

    unique_ptr<double[]> x {new double[nrows * ncols]};
    unique_ptr<double[]> col_means1 {new double[ncols]};
    unique_ptr<double[]> col_means2 {new double[ncols]};
    unique_ptr<size_t[]> col_counts1 {new size_t[ncols]};
    unique_ptr<size_t[]> col_counts2 {new size_t[ncols]};

    Init(x.get(), nrows * ncols, rng_min, rng_max, rng_seed);

    bool rc1 = Avx512CalcColumnMeansCpp(x.get(), nrows, ncols, col_means1.get(), col_
    counts1.get(), x_min);
    bool rc2 = Avx512CalcColumnMeans_(x.get(), nrows, ncols, col_means2.get(), col_counts2.
    get(), x_min);

    cout << "Results for Avx512CalcColumnMeans\n";

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code: ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    cout << "Test Matrix (nrows = " << nrows << ", ncols = " << ncols << ")\n";
    cout << "\nColumn Means\n";
    cout << fixed << setprecision(4);

    for (size_t j = 0; j < ncols; j++)
    {
        cout << setw(4) << j << ": ";
        cout << "col_means = ";
        cout << setw(10) << col_means1[j] << ", ";
        cout << setw(10) << col_means2[j] << "    ";

        cout << "col_counts = ";
        cout << setw(6) << col_counts1[j] << ", ";
        cout << setw(6) << col_counts2[j] << '\n';

        if (col_means1[j] != col_means2[j])
            cout << "col_means compare error\n";

        if (col_counts1[j] != col_counts2[j])
            cout << "col_counts compare error\n";
    }
}
```

```
int main()
{
    Avx512CalcColumnMeans();
    return 0;
}

;-------------------------------------------------
;               Ch13_06.asm
;-------------------------------------------------

        include <cmpequ.asmh>
        include <MacrosX86-64-AVX.asmh>

        extern c_NumRowsMax:qword
        extern c_NumColsMax:qword

; extern "C" bool Avx512CalcColumnMeans_(const double* x, size_t nrows, size_t ncols,
double* col_means, size_t* col_counts, double x_min);

        .code
Avx512CalcColumnMeans_ proc frame
        _CreateFrame CCM_,0,0,rbx,r12,r13
        _EndProlog

; Validate nrows and ncols
        xor eax,eax                         ;set error return code
        test rdx,rdx
        jz Done                             ;jump if nrows is zero
        cmp rdx,[c_NumRowsMax]
        ja Done                             ;jump if nrows is too large
        test r8,r8
        jz Done                             ;jump if ncols is zero
        cmp r8,[c_NumColsMax]
        ja Done                             ;jump if ncols is too large

; Load argument values col_counts and x_min
        mov ebx,1
        vpbroadcastq zmm4,rbx               ;zmm4 = 8 qwords of 1
        mov rbx,[rbp+CCM_OffsetStackArgs]   ;rbx = col_counts ptr
        lea r13,[rbp+CCM_OffsetStackArgs+8] ;r13 = ptr to x_min

; Set initial col_means and col_counts to zero
        xor r10,r10
        vxorpd xmm0,xmm0,xmm0
@@:     vmovsd real8 ptr[r9+rax*8],xmm0         ;col_means[i] = 0.0
        mov [rbx+rax*8],r10                     ;col_counts[i] = 0
        inc rax
        cmp rax,r8
        jne @B                                  ;repeat until done
```

```
; Compute the sum of each column in x
LP1:    xor r10,r10                         ;r10 = col_index
        mov r11,r9                          ;r11 = ptr to col_means
        mov r12,rbx                         ;r12 = ptr to col_counts

LP2:    mov rax,r10                         ;rax = col_index
        add rax,8
        cmp rax,r8                          ;8 or more columns remaining?
        ja @F                               ;jump if col_index + 8 > ncols

; Update col_means and col_counts using next eight columns
        vmovupd zmm0,zmmword ptr [rcx]         ;load next 8 cols of cur row
        vcmppd k1,zmm0,real8 bcst [r13],CMP_GE  ;k1 = mask of values >= x_min
        vmovupd zmm1{k1}{z},zmm0               ;values >= x_min or 0.0
        vaddpd zmm2,zmm1,zmmword ptr [r11]     ;add values to col_means
        vmovupd zmmword ptr [r11],zmm2         ;save updated col_means

        vpmovm2q zmm0,k1                       ;convert mask to vector
        vpandq zmm1,zmm0,zmm4                  ;qword values for add
        vpaddq zmm2,zmm1,zmmword ptr [r12]     ;update col_counts
        vmovdqu64 zmmword ptr [r12],zmm2       ;save updated col_counts

        add r10,8                           ;col_index += 8
        add rcx,64                          ;x += 8
        add r11,64                          ;col_means += 8
        add r12,64                          ;col_counts += 8
        jmp NextColSet

; Update col_means and col_counts using next four columns
@@:     sub rax,4
        cmp rax,r8                          ;4 or more columns remaining?
        ja @F                               ;jump if col_index + 4 > ncols

        vmovupd ymm0,ymmword ptr [rcx]         ;load next 4 cols of cur row
        vcmppd k1,ymm0,real8 bcst [r13],CMP_GE  ;k1 = mask of values >= x_min
        vmovupd ymm1{k1}{z},ymm0               ;values >= x_min or 0.0
        vaddpd ymm2,ymm1,ymmword ptr [r11]     ;add values to col_means
        vmovupd ymmword ptr [r11],ymm2         ;save updated col_means

        vpmovm2q ymm0,k1                       ;convert mask to vector
        vpandq ymm1,ymm0,ymm4                  ;qword values for add
        vpaddq ymm2,ymm1,ymmword ptr [r12]     ;update col_counts
        vmovdqu64 ymmword ptr [r12],ymm2       ;save updated col_counts

        add r10,4                           ;col_index += 4
        add rcx,32                          ;x += 4
        add r11,32                          ;col_means += 4
        add r12,32                          ;col_counts += 4
        jmp NextColSet
```

```
; Update col_means and col_counts using next two columns
@@:     sub rax,2
        cmp rax,r8                           ;2 or more columns remaining?
        ja @F                                ;jump if col_index + 2 > ncols

        vmovupd xmm0,xmmword ptr [rcx]       ;load next 2 cols of cur row
        vcmppd k1,xmm0,real8 bcst [r13],CMP_GE  ;k1 = mask of values >= x_min
        vmovupd xmm1{k1}{z},xmm0             ;values >= x_min or 0.0
        vaddpd xmm2,xmm1,xmmword ptr [r11]   ;add values to col_means
        vmovupd xmmword ptr [r11],xmm2       ;save updated col_means

        vpmovm2q xmm0,k1                     ;convert mask to vector
        vpandq xmm1,xmm0,xmm4                ;qword values for add
        vpaddq xmm2,xmm1,xmmword ptr [r12]   ;update col_counts
        vmovdqu64 xmmword ptr [r12],xmm2     ;save updated col_counts

        add r10,2                            ;col_index += 2
        add rcx,16                           ;x += 2
        add r11,16                           ;col_means += 2
        add r12,16                           ;col_counts += 2
        jmp NextColSet

; Update col_means using last column of current row
@@:     vmovsd xmm0,real8 ptr [rcx]          ;load x from last column
        vcmpsd k1,xmm0,real8 ptr [r13],CMP_GE   ;k1 = mask of values >= x_min
        vmovsd xmm1{k1}{z},xmm1,xmm0         ;value or 0.0
        vaddsd xmm2,xmm1,real8 ptr [r11]     ;add to col_means
        vmovsd real8 ptr [r11],xmm2          ;save updated col_means
        kmovb eax,k1                         ;eax = 0 or 1
        add qword ptr [r12],rax              ;update col_counts

        add r10,1                            ;col_index += 1
        add rcx,8                            ;update x ptr

NextColSet:
        cmp r10,r8                       ;more columns in current row?
        jb LP2                           ;jump if yes
        dec rdx                          ;nrows -= 1
        jnz LP1                          ;jump if more rows

; Compute the final col_means
@@:     vmovsd xmm0,real8 ptr [r9]       ;xmm0 = col_means[i]
        vcvtsi2sd xmm1,xmm1,qword ptr [rbx] ;xmm1 = col_counts[i]
        vdivsd xmm2,xmm0,xmm1            ;compute final mean
        vmovsd real8 ptr [r9],xmm2       ;save col_mean[i]
        add r9,8                         ;update col_means ptr
        add rbx,8                        ;update col_counts ptr
        sub r8,1                         ;ncols -= 1
        jnz @B                           ;repeat until done
```

```
        mov eax,1                           ;set success return code

Done:   _DeleteFrame rbx,r12,r13
        vzeroupper
        ret

Avx512CalcColumnMeans_ endp
        end
```

The function Avx512CalcColumnMeansCpp contains a C++ implementation of the columns means algorithm. This function uses two nested for loops to sum the elements of each column in the two-dimensional array. During each inner loop iteration, the value of array element x[i][j] is added to the current column running sum in col_means[j] only if it's greater than or equal to x_min. The number of elements greater than or equal to x_min in each column is maintained in the array col_counts. Following the summing loops, the final column means are calculated using a simple for loop.

Following its prolog, the function Avx512CalcColumnMeans_ validates argument values nrows and ncols. It then performs its required initializations. The mov ebx,1 and vpbroadcastq zmm4,rbx instructions load the value one into each quadword element of ZMM4. Registers RBX and R13 are then initialized as pointers to col_counts and x_min, respectively. The final initialization task employs a simple for loop that sets each element in col_means and col_counts to zero.

Similar to source code example Ch09_03, the inner for loop in Avx512CalcColumnMeans_ employs slightly different instruction sequences to sum column elements, which vary depending on the number of columns in the array (see Figure 9-2) and the current column index. For each row, elements in the first eight columns of x can be added to col_means using 512-bit wide packed double-precision floating-point addition. The remaining column element values are added to col_means using 512-, 256-, or 128-bit wide packed or scalar double-precision floating-point addition.

The outer loop label LP1 is the starting point for adding elements from the current row of x to col_means. The xor r10,r10 instruction initializes col_index to zero; the mov r11,r9 instruction loads R11 with a pointer to col_means; and mov r12,rbx points R12 to col_counts. Each iteration of the inner loop LP2 begins with a check to ensure that at least eight columns are available in the current row for processing. If eight columns are available, the vmovupd zmm0,zmmword ptr [rcx] instruction loads the next eight elements of the current row into register ZMM0. The ensuing vcmppd k1,zmm0,real8 bcst [r13],CMP_GE instruction compares each element in ZMM0 to x_min and sets the corresponding bit position in opmask register K1 to indicate the result. Note that the embedded broadcast operand of the vpcmpd instruction is used here for demonstration purposes. In this source code example, it would be more efficient to initialize a packed version of x_min prior to the start of the processing loops. The next instruction, vmovupd zmm1{k1}{z},zmm0, uses zero masking to effectively eliminate values less than x_min from subsequent calculations. The next two instructions, vaddpd zmm2,zmm1,zmmword ptr [r11] and vmovupd zmmword ptr [r11],zmm2, update the running column sums that are maintained in the col_means. Figure 13-2 illustrates this operation in greater detail.

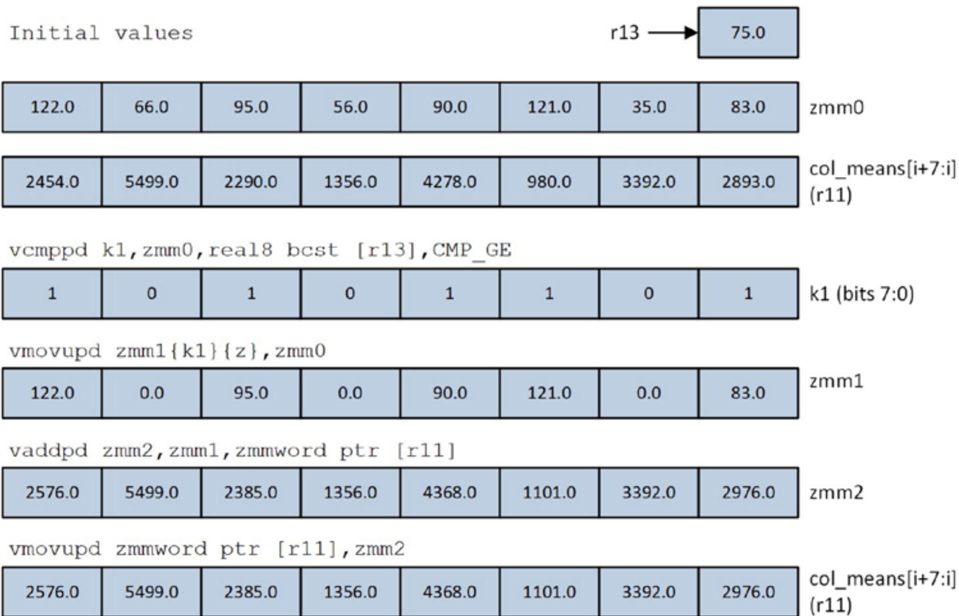



***Figure 13-2.*** *Updating the intermediate sums in `col_means` using zero merging*

The next code block updates the element counts in `col_counts`. The `vpmovm2q zmm0,k1` instruction (Convert Mask Register to Vector Register) sets each quadword element in ZMM0 to all ones (`0xFFFFFFFFFFFFFFFF`) or all zeros (`0x0000000000000000`) according to the value of the corresponding bit position in K1. The ensuing `vpandq zmm1,zmm0,zmm4` instruction zeros the high-order 63 bits of each quadword value in ZMM0 and saves this result in ZMM1. The next two instructions, `vpaddq zmm2,zmm1,zmmword ptr [r12]` and `vmovdqu64 zmmword ptr [r12],zmm2`, update the count values in `col_counts`, as shown in Figure 13-3. The `vmovdqu64` instruction saves the 512-bit wide packed quadword operand in ZMM2 to the location pointed to by register R12. AVX512F also includes a `vmovdqu32` instruction for 512-bit wide packed doubleword moves.

Initial values

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | zmm4 |
| 51 | 48 | 49 | 56 | 47 | 61 | 48 | 51 | col_counts[i+7:i] (r12) |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | k1 (bits 7:0) |

vpmovm2q zmm0,k1

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| all 1s | all 0s | all 1s | all 0s | all 1s | all 1s | all 0s | all 1s | zmm0 |

vpandq zmm1,zmm0,zmm4

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | zmm1 |

vpaddq zmm2,zmm1,zmmword ptr [r12]

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 52 | 48 | 50 | 56 | 48 | 62 | 48 | 52 | zmm2 |

vmovdqu64 zmmword ptr [r12],zmm2

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 52 | 48 | 50 | 56 | 48 | 62 | 48 | 52 | col_counts[i+7:i] (r12) |

***Figure 13-3.*** *Updating the intermediate element counts in* `col_counts`

Following execution of the `vmovdqu64` instruction, the algorithm's various pointers and counters are updated to reflect the eight processed elements. The summation code repeats until the number of array elements that remain in the current row is less than eight. Once this condition is met, the remaining column elements (if any) are processed using 256-, 128-, or 64-bit wide operands using the same technique described in the previous paragraph. Note that function `Avx512CalcColumnMeans_` uses AVX-512 instructions that employ YMM or XMM registers with embedded broadcast and zero merging operands. These instructions require an AVX-512 conforming processor that supports the AVX512VL instruction set extension. After calculating the column sums, each element in `col_means` is divided by the corresponding element in `col_counts` to obtain the final column mean. Here are the results for source code example `Ch13_06`:

```
Results for Avx512CalcColumnMeans

Test Matrix (nrows = 20000, ncols = 23)

Column Means
   0: col_means =   536.6483,   536.6483    col_counts =  18548,  18548
   1: col_means =   535.8669,   535.8669    col_counts =  18538,  18538
   2: col_means =   534.7049,   534.7049    col_counts =  18457,  18457
   3: col_means =   535.8747,   535.8747    col_counts =  18544,  18544
   4: col_means =   540.7477,   540.7477    col_counts =  18501,  18501
   5: col_means =   535.9465,   535.9465    col_counts =  18493,  18493
   6: col_means =   539.0142,   539.0142    col_counts =  18528,  18528
   7: col_means =   536.6623,   536.6623    col_counts =  18496,  18496
   8: col_means =   532.1445,   532.1445    col_counts =  18486,  18486
```

```
  9: col_means =   543.4736,   543.4736    col_counts =  18479,  18479
 10: col_means =   535.2980,   535.2980    col_counts =  18552,  18552
 11: col_means =   536.4255,   536.4255    col_counts =  18486,  18486
 12: col_means =   537.6472,   537.6472    col_counts =  18473,  18473
 13: col_means =   537.9775,   537.9775    col_counts =  18511,  18511
 14: col_means =   538.4742,   538.4742    col_counts =  18514,  18514
 15: col_means =   539.2965,   539.2965    col_counts =  18497,  18497
 16: col_means =   537.9710,   537.9710    col_counts =  18454,  18454
 17: col_means =   536.7826,   536.7826    col_counts =  18566,  18566
 18: col_means =   538.3274,   538.3274    col_counts =  18452,  18452
 19: col_means =   538.2181,   538.2181    col_counts =  18491,  18491
 20: col_means =   532.6881,   532.6881    col_counts =  18514,  18514
 21: col_means =   537.0067,   537.0067    col_counts =  18554,  18554
 22: col_means =   539.0643,   539.0643    col_counts =  18548,  18548
```

## Vector Cross Products

The next source code example, Ch13_07, demonstrates vector cross product calculations using arrays of three-dimensional vectors. It also illustrates how to perform data gather and scatter operations using AVX-512 instructions. Listing 13-7 shows the source code for example Ch13_07

***Listing 13-7.*** Example Ch13_07

```
//------------------------------------------------
//               Ch13_07.h
//------------------------------------------------

#pragma once

// Simple vector structure
typedef struct
{
    double X;           // Vector X component
    double Y;           // Vector Y component
    double Z;           // Vector Z component
} Vector;

// Vector structure of arrays
typedef struct
{
    double* X;          // Pointer to X components
    double* Y;          // Pointer to Y components
    double* Z;          // Pointer to Z components
} VectorSoA;

// Ch13_07.cpp
void InitVec(Vector* a_aos, Vector* b_aos, VectorSoA& a_soa, VectorSoA& b_soa, size_t num_vec);
bool Avx512VcpAosCpp(Vector* c, const Vector* a, const Vector* b, size_t num_vec);
bool Avx512VcpSoaCpp(VectorSoA* c, const VectorSoA* a, const VectorSoA* b, size_t num_vec);
```

```
// Ch13_07_.asm
extern "C" bool Avx512VcpAos_(Vector* c, const Vector* a, const Vector* b, size_t num_vec);
extern "C" bool Avx512VcpSoa_(VectorSoA* c, const VectorSoA* a, const VectorSoA* b, size_t
num_vec);

// Ch13_07_BM.cpp
extern void Avx512Vcp_BM(void);

//------------------------------------------------
//               Ch13_07.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <memory>
#include "Ch13_07.h"
#include "AlignedMem.h"

using namespace std;

void InitVec(Vector* a_aos, Vector* b_aos, VectorSoA& a_soa, VectorSoA& b_soa, size_t num_vec)
{
    uniform_int_distribution<> ui_dist {1, 100};
    default_random_engine rng {103};

    for (size_t i = 0; i < num_vec; i++)
    {
        double a_x = (double)ui_dist(rng);
        double a_y = (double)ui_dist(rng);
        double a_z = (double)ui_dist(rng);
        double b_x = (double)ui_dist(rng);
        double b_y = (double)ui_dist(rng);
        double b_z = (double)ui_dist(rng);

        a_aos[i].X = a_soa.X[i] = a_x;
        a_aos[i].Y = a_soa.Y[i] = a_y;
        a_aos[i].Z = a_soa.Z[i] = a_z;

        b_aos[i].X = b_soa.X[i] = b_x;
        b_aos[i].Y = b_soa.Y[i] = b_y;
        b_aos[i].Z = b_soa.Z[i] = b_z;
    }
}

void Avx512Vcp(void)
{
    const size_t align = 64;
    const size_t num_vec = 16;
```

```
    unique_ptr<Vector> a_aos_up {new Vector[num_vec] };
    unique_ptr<Vector> b_aos_up {new Vector[num_vec] };
    unique_ptr<Vector> c_aos_up {new Vector[num_vec] };
    Vector* a_aos = a_aos_up.get();
    Vector* b_aos = b_aos_up.get();
    Vector* c_aos = c_aos_up.get();

    VectorSoA a_soa, b_soa, c_soa;
    AlignedArray<double> a_soa_x_aa(num_vec, align);
    AlignedArray<double> a_soa_y_aa(num_vec, align);
    AlignedArray<double> a_soa_z_aa(num_vec, align);
    AlignedArray<double> b_soa_x_aa(num_vec, align);
    AlignedArray<double> b_soa_y_aa(num_vec, align);
    AlignedArray<double> b_soa_z_aa(num_vec, align);
    AlignedArray<double> c_soa_x_aa(num_vec, align);
    AlignedArray<double> c_soa_y_aa(num_vec, align);
    AlignedArray<double> c_soa_z_aa(num_vec, align);
    a_soa.X = a_soa_x_aa.Data();
    a_soa.Y = a_soa_y_aa.Data();
    a_soa.Z = a_soa_z_aa.Data();
    b_soa.X = b_soa_x_aa.Data();
    b_soa.Y = b_soa_y_aa.Data();
    b_soa.Z = b_soa_z_aa.Data();
    c_soa.X = c_soa_x_aa.Data();
    c_soa.Y = c_soa_y_aa.Data();
    c_soa.Z = c_soa_z_aa.Data();

    InitVec(a_aos, b_aos, a_soa, b_soa, num_vec);

    bool rc1 = Avx512VcpAos_(c_aos, a_aos, b_aos, num_vec);
    bool rc2 = Avx512VcpSoa_(&c_soa, &a_soa, &b_soa, num_vec);

    cout << "Results for Avx512VectorCrossProd\n";

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << ", ";
        return;
     }

    cout << fixed << setprecision(1);

    for (size_t i = 0; i < num_vec; i++)
    {
        cout << "Vector cross product #" << i << '\n';

        const unsigned int w = 9;

        cout << "  a:      ";
```

```
        cout << setw(w) << a_aos[i].X << ' ';
        cout << setw(w) << a_aos[i].Y << ' ';
        cout << setw(w) << a_aos[i].Z << '\n';

        cout << "  b:      ";
        cout << setw(w) << b_aos[i].X << ' ';
        cout << setw(w) << b_aos[i].Y << ' ';
        cout << setw(w) << b_aos[i].Z << '\n';

        cout << "  c_aos:  ";
        cout << setw(w) << c_aos[i].X << ' ';
        cout << setw(w) << c_aos[i].Y << ' ';
        cout << setw(w) << c_aos[i].Z << '\n';

        cout << "  c_soa:  ";
        cout << setw(w) << c_soa.X[i] << ' ';
        cout << setw(w) << c_soa.Y[i] << ' ';
        cout << setw(w) << c_soa.Z[i] << '\n';

        bool is_valid_x = c_aos[i].X == c_soa.X[i];
        bool is_valid_y = c_aos[i].Y == c_soa.Y[i];
        bool is_valid_z = c_aos[i].Z == c_soa.Z[i];

        if (!is_valid_x || !is_valid_y || !is_valid_z)
        {
            cout << "Compare error at index " << i << '\n';
            cout << "  is_valid_x = " << boolalpha << is_valid_x << '\n';
            cout << "  is_valid_y = " << boolalpha << is_valid_y << '\n';
            cout << "  is_valid_z = " << boolalpha << is_valid_z << '\n';
            return;
        }
    }
}

int main()
{
    Avx512Vcp();
    Avx512Vcp_BM();
    return 0;
}

;-------------------------------------------------
;               Ch13_07.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; Indices for gather and scatter instructions
ConstVals   segment readonly align(64) 'const'
GS_X        qword 0, 3, 6,  9, 12, 15, 18, 21
GS_Y        qword 1, 4, 7, 10, 13, 16, 19, 22
```

```
GS_Z        qword 2, 5, 8, 11, 14, 17, 20, 23
ConstVals   ends

; extern "C" bool Avx512VcpAos_(Vector* c, const Vector* a, const Vector* b, size_t num_
vectors);

        .code
Avx512VcpAos_ proc

; Make sure num_vec is valid
        xor eax,eax                         ;set error code (also i = 0)
        test r9,r9
        jz Done                             ;jump if num_vec is zero
        test r9,07h
        jnz Done                            ;jump if num_vec % 8 != 0 is true

; Load indices for gather and scatter operations
        vmovdqa64 zmm29,zmmword ptr [GS_X]  ;zmm29 = X component indices
        vmovdqa64 zmm30,zmmword ptr [GS_Y]  ;zmm30 = Y component indices
        vmovdqa64 zmm31,zmmword ptr [GS_Z]  ;zmm31 = Z component indices

; Load next 8 vectors
        align 16
@@:     kxnorb k1,k1,k1
        vgatherqpd zmm0{k1},[rdx+zmm29*8]       ;zmm0 = A.X values

        kxnorb k2,k2,k2
        vgatherqpd zmm1{k2},[rdx+zmm30*8]       ;zmm1 = A.Y values

        kxnorb k3,k3,k3
        vgatherqpd zmm2{k3},[rdx+zmm31*8]       ;zmm2 = A.Z values

        kxnorb k4,k4,k4
        vgatherqpd zmm3{k4},[r8+zmm29*8]        ;zmm3 = B.X values

        kxnorb k5,k5,k5
        vgatherqpd zmm4{k5},[r8+zmm30*8]        ;zmm4 = B.Y values

        kxnorb k6,k6,k6
        vgatherqpd zmm5{k6},[r8+zmm31*8]        ;zmm5 = B.Z values

; Calculate 8 vector cross products
        vmulpd zmm16,zmm1,zmm5
        vmulpd zmm17,zmm2,zmm4
        vsubpd zmm18,zmm16,zmm17                ;c.X = a.Y * b.Z - a.Z * b.Y

        vmulpd zmm19,zmm2,zmm3
        vmulpd zmm20,zmm0,zmm5
        vsubpd zmm21,zmm19,zmm20                ;c.Y = a.Z * b.X - a.X * b.Z
```

```
        vmulpd zmm22,zmm0,zmm4
        vmulpd zmm23,zmm1,zmm3
        vsubpd zmm24,zmm22,zmm23              ;c.Z = a.X * b.Y - a.Y * b.X

; Save calculated cross products
        kxnorb k4,k4,k4
        vscatterqpd [rcx+zmm29*8]{k4},zmm18     ;save C.X components

        kxnorb k5,k5,k5
        vscatterqpd [rcx+zmm30*8]{k5},zmm21     ;save C.Y components

        kxnorb k6,k6,k6
        vscatterqpd [rcx+zmm31*8]{k6},zmm24     ;save C.Z components

; Update pointers and counters
        add rcx,192                           ;c += 8
        add rdx,192                           ;a += 8
        add r8,192                            ;b += 8
        add rax,8                             ;i += 8
        cmp rax,r9
        jb @B

        mov eax,1                             ;set success return code

Done:   vzeroupper
        ret
Avx512VcpAos_ endp

; extern "C" bool Avx512VcpSoa_(VectorSoA* c, const VectorSoA* a, const VectorSoA* b, size_t
num_vectors);

Avx512VcpSoa_ proc frame
        _CreateFrame CP2_,0,0,rbx,rsi,rdi,r12,r13,r14,r15
        _EndProlog

; Make sure num_vec is valid
        xor eax,eax
        test r9,r9
        jz Done                           ;jump if num_vec is zero
        test r9,07h
        jnz Done                          ;jump if num_vec % 8 != 0 is true

; Load vector array pointers and check for proper alignment
        mov r10,[rdx]                     ;r10 = a.X
        or rax,r10
        mov r11,[rdx+8]                   ;r11 = a.Y
        or rax,r11
        mov r12,[rdx+16]                  ;r12 = a.Z
        or rax,r12
```

```
        mov r13,[r8]                        ;r13 = b.X
        or rax,r13
        mov r14,[r8+8]                      ;r14 = b.Y
        or rax,r14
        mov r15,[r8+16]                     ;r15 = b.Z
        or rax,r15

        mov rbx,[rcx]                       ;rbx = c.X
        or rax,rbx
        mov rsi,[rcx+8]                     ;rsi = c.Y
        or rax,rsi
        mov rdi,[rcx+16]                    ;rdi = c.Z
        or rax,rdi

        and rax,3fh                         ;misaligned component array?
        mov eax,0                           ;error return code (also i = 0)
        jnz Done

; Load next block (8 vectors) from a and b
        align 16
@@:     vmovapd zmm0,zmmword ptr [r10+rax*8]    ;zmm0 = a.X values
        vmovapd zmm1,zmmword ptr [r11+rax*8]    ;zmm1 = a.Y values
        vmovapd zmm2,zmmword ptr [r12+rax*8]    ;zmm2 = a.Z values
        vmovapd zmm3,zmmword ptr [r13+rax*8]    ;zmm3 = b.X values
        vmovapd zmm4,zmmword ptr [r14+rax*8]    ;zmm4 = b.Y values
        vmovapd zmm5,zmmword ptr [r15+rax*8]    ;zmm5 = b.Z values

; Calculate cross products
        vmulpd zmm16,zmm1,zmm5
        vmulpd zmm17,zmm2,zmm4
        vsubpd zmm18,zmm16,zmm17                ;c.X = a.Y * b.Z - a.Z * b.Y

        vmulpd zmm19,zmm2,zmm3
        vmulpd zmm20,zmm0,zmm5
        vsubpd zmm21,zmm19,zmm20                ;c.Y = a.Z * b.X - a.X * b.Z

        vmulpd zmm22,zmm0,zmm4
        vmulpd zmm23,zmm1,zmm3
        vsubpd zmm24,zmm22,zmm23                ;c.Z = a.X * b.Y - a.Y * b.X

; Save calculated cross products
        vmovapd zmmword ptr [rbx+rax*8],zmm18   ;save C.X values
        vmovapd zmmword ptr [rsi+rax*8],zmm21   ;save C.Y values
        vmovapd zmmword ptr [rdi+rax*8],zmm24   ;save C.Z values

        add rax,8                               ;i += 8
        cmp rax,r9
        jb @B                                   ;repeat until done
```

```
Done:   vzeroupper
        _DeleteFrame rbx,rsi,rdi,r12,r13,r14,r15
        ret
Avx512VcpSoa_ endp
        end
```

The cross product of two three-dimensional vectors **a** and **b** is a third vector **c** that is perpendicular to both **a** and **b**. The *x*, *y*, and *z* components of **c** can be calculated using the following equations:

$$c_x = a_y b_z - a_z b_y \quad c_y = a_z b_x - a_x b_z \quad c_z = a_x b_y - a_y b_x$$

The C++ header file `Ch13_07.h` that's shown in Listing 13-7 includes the structure definitions `Vector` and `VectorSoA`. The structure `Vector` contains three double-precision floating-point values—`X`, `Y`, and `Z`—that represent the components of a three-dimensional vector. The `VectorSoA` structure incorporates three pointers to double-precision floating-point arrays. Each array contains the values for a single vector component. Example `Ch13_07` uses these structures to compare the performance of two different vector cross product calculating algorithms. The first algorithm performs its calculations using an array of structures (AOS), while the second algorithm exploits a structure of arrays (SOA).

The C++ function `Avx512Vcp` begins its execution by allocating storage space for sets of vector data structures. This function uses the C++ template class `unique_ptr<Vector>` to allocate storage for three AOSs. Note that each `Vector` object is *not* explicitly aligned on a 64-byte boundary since doing this would consume a considerable about of storage space that's never used. Each `unique_ptr<Vector>` AOS is also representative of how this type of data construct is commonly employed in many real-world programs. `Avx512Vcp` uses the C++ template class `AlignedArray<double>` to allocate properly aligned storage space for the vector SOAs. Following data structure allocation, the function `InitVec` initializes both sets of vectors `a` and `b` using random values. It then invokes the assembly language vector cross product functions `Avx512VcpAos_` and `Avx512VcpSoa_`.

Near the top of the assembly language file is a custom constant segment named `ConstVals`, which contains indices for the `vgatherqpd` and `vscatterqpd` instructions that are used in `Avx512VcpAos_`. The index values in this segment correspond to the memory ordering of `Vector` components `X`, `Y`, and `Z` in an array of `Vector` objects. Figure 13-4 illustrates this ordering in greater detail. Note that the indices defined in `ConstVals` enable the `vgatherqpd` and `vscatterqpd` instructions to load and save eight `Vector` objects.

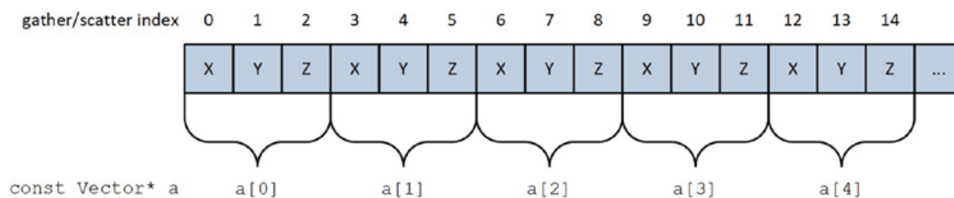



***Figure 13-4.*** *Memory ordering of components X, Y, and Z in an array of `Vector` objects*

Following validation of `num_vec`, three `vmovdqa64` (Move Aligned Packed Quadword Values) instructions load the gather/scatter indices for `Vector` components `X`, `Y`, and `X` into registers ZMM29, ZMM30, and ZMM31, respectively. The processing loop begins with a `kxnorb k1,k1,k1` instruction that sets the low-order eight bits of opmask register K1 to one. The subsequent `vgatherqpd zmm0{k1},[rdx+zmm29*8]` instruction loads eight `X` component values from `Vector a` into register ZMM0. The `vgatherqpd` instruction loads eight values since the low-order eight bits of opmask register K1 are all set to one.

Five more sets of kxnorb and vgatherqpd instructions load the remaining Vector components into registers ZMM1–ZMM5. Note that during its execution, the vgatherqpd instruction sets the entire opmask register to zero unless an exception occurs due to an invalid memory access, which can be caused by an incorrect index or bad base register value. This updating of the opmask register introduces a potential register dependency that is eliminated by using a different opmask register for each vgatherqpd instruction. The next code block calculates eight vector cross products using basic packed double-precision floating-point arithmetic. The cross-product results are then saved to the destination Vector array c using three vscatterqpd instructions. Like the vgatherqpd instruction, the vscatterqpd instruction also sets its opmask register operand to zero unless an exception occurs.

The function Avx512VcpSoa_ begins its execution by validating num_vec. It then verifies that the nine vector component array pointers are properly aligned on a 64-byte boundary. The processing loop in Avx512VcpSoa_ employs straightforward packed double-precision floating-point arithmetic to calculate the vector cross products. Note that Avx512VcpSoa_ uses the aligned move instruction vmovapd to perform all vector component loads and stores. Here are the results for source code example Ch13_07:

```
Results for Avx512VectorCrossProd

Vector cross product #0
  a:           96.0      30.0      52.0
  b:           64.0      62.0      79.0
  c_aos:     -854.0   -4256.0    4032.0
  c_soa:     -854.0   -4256.0    4032.0
Vector cross product #1
  a:           26.0      33.0      66.0
  b:           89.0      36.0      20.0
  c_aos:    -1716.0    5354.0   -2001.0
  c_soa:    -1716.0    5354.0   -2001.0
Vector cross product #2
  a:           56.0      60.0      53.0
  b:           16.0      45.0      46.0
  c_aos:      375.0   -1728.0    1560.0
  c_soa:      375.0   -1728.0    1560.0
Vector cross product #3
  a:           79.0      27.0      22.0
  b:           18.0      75.0      45.0
  c_aos:     -435.0   -3159.0    5439.0
  c_soa:     -435.0   -3159.0    5439.0
Vector cross product #4
  a:           77.0      30.0      46.0
  b:           44.0      77.0      99.0
  c_aos:     -572.0   -5599.0    4609.0
  c_soa:     -572.0   -5599.0    4609.0
Vector cross product #5
  a:           30.0      21.0      26.0
  b:           43.0      61.0      47.0
  c_aos:     -599.0    -292.0     927.0
  c_soa:     -599.0    -292.0     927.0
Vector cross product #6
  a:           58.0      56.0      46.0
  b:           84.0      37.0      76.0
  c_aos:     2554.0    -544.0   -2558.0
```

```
  c_soa:     2554.0    -544.0   -2558.0
Vector cross product #7
  a:           34.0      28.0      95.0
  b:           20.0      51.0      36.0
  c_aos:    -3837.0     676.0    1174.0
  c_soa:    -3837.0     676.0    1174.0
Vector cross product #8
  a:           34.0      50.0      35.0
  b:           48.0       1.0      24.0
  c_aos:     1165.0     864.0   -2366.0
  c_soa:     1165.0     864.0   -2366.0
Vector cross product #9
  a:           28.0      12.0      46.0
  b:            6.0      53.0      77.0
  c_aos:    -1514.0   -1880.0    1412.0
  c_soa:    -1514.0   -1880.0    1412.0
Vector cross product #10
  a:           43.0      78.0      86.0
  b:           12.0      61.0      97.0
  c_aos:     2320.0   -3139.0    1687.0
  c_soa:     2320.0   -3139.0    1687.0
Vector cross product #11
  a:           53.0      78.0      85.0
  b:           78.0      34.0      65.0
  c_aos:     2180.0    3185.0   -4282.0
  c_soa:     2180.0    3185.0   -4282.0
Vector cross product #12
  a:            9.0      66.0       2.0
  b:           54.0      45.0      55.0
  c_aos:     3540.0    -387.0   -3159.0
  c_soa:     3540.0    -387.0   -3159.0
Vector cross product #13
  a:           15.0      59.0      35.0
  b:           94.0      67.0      22.0
  c_aos:    -1047.0    2960.0   -4541.0
  c_soa:    -1047.0    2960.0   -4541.0
Vector cross product #14
  a:           95.0      20.0      24.0
  b:           45.0      85.0      55.0
  c_aos:     -940.0   -4145.0    7175.0
  c_soa:     -940.0   -4145.0    7175.0
Vector cross product #15
  a:           76.0      77.0      15.0
  b:           29.0      95.0      23.0
  c_aos:      346.0   -1313.0    4987.0
  c_soa:      346.0   -1313.0    4987.0

Running benchmark function Avx512VectorCrossProd_BM - please wait
Benchmark times save to file Ch13_07_Avx512VectorCrossProd_BM_CHROMIUM.csv
```

Table 13-1 shows benchmark timing measurements for the two cross product calculating functions. This table uses dashes to signify processors that do not support AVX-512. For source code example Ch13_07, the SOA technique is somewhat faster than the AOS method.

***Table 13-1.*** *Benchmark Timing Measurements for Vector Cross Product Calculating Functions (1,000,000 Cross Products)*

| CPU | Avx512VcpAos_ | Avx512VcpSoa_ |
|---|---|---|
| i7-4790S | ---- | ---- |
| i9-7900X | 4734 | 4141 |
| i7-8700K | ---- | ---- |

## Matrix-Vector Multiplication

Many computer graphics and image processing algorithms perform matrix-vector multiplications using 4 × 4 matrices and 4 × 1 vectors. In 3D computer graphics software, these types of calculations are universally employed to perform affine transformations (e.g., translation, rotation, and scaling) using homogeneous coordinates. Figure 13-5 shows the equations that can be used to multiply a 4 × 4 matrix by a 4 × 1 vector. Note that the components of vector **b** are a simple sum-of-products calculation of the matrix's columns and the individual components of vector **a**. Figure 13-5 also shows a sample matrix-vector multiplication calculation using real numbers.

$$\begin{bmatrix} b_w \\ b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} a_w \\ a_x \\ a_y \\ a_z \end{bmatrix} \qquad \begin{bmatrix} 304 \\ 564 \\ 824 \\ 1084 \end{bmatrix} = \begin{bmatrix} 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \\ 40 & 41 & 42 & 43 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$

$$b_w = m_{00}a_w + m_{01}a_x + m_{02}a_y + m_{03}a_z \qquad 304 = 10(5) + 11(6) + 12(7) + 13(8)$$

$$b_x = m_{10}a_w + m_{11}a_x + m_{12}a_y + m_{13}a_z \qquad 564 = 20(5) + 21(6) + 22(7) + 23(8)$$

$$b_y = m_{20}a_w + m_{21}a_x + m_{22}a_y + m_{23}a_z \qquad 824 = 30(5) + 31(6) + 32(7) + 33(8)$$

$$b_z = m_{30}a_w + m_{31}a_x + m_{32}a_y + m_{33}a_z \qquad 1084 = 40(5) + 41(6) + 42(7) + 43(8)$$

col0 col1 col2 col3

***Figure 13-5.*** *Equations for matrix-vector multiplication and a sample calculation*

Listing 13-8 shows the source code for example Ch13_08. This example demonstrates how to multiply a single 4 × 4 matrix with a set of 4 × 1 vectors that are stored in an array.

***Listing 13-8.*** Example Ch13_08

```
//------------------------------------------------
//               Ch13_08.h
//------------------------------------------------

#pragma once

// Simple 4x1 vector structure
struct Vec4x1_F32
{
    float W, X, Y, Z;
};

// Ch13_08.cpp
extern void InitVecArray(Vec4x1_F32* va, size_t num_vec);
extern bool Avx512MatVecMulF32Cpp(Vec4x1_F32* vec_b, float mat[4][4], Vec4x1_F32* vec_a,
size_t num_vec);

// Ch13_08_.asm
extern "C" bool Avx512MatVecMulF32_(Vec4x1_F32* vec_b, float mat[4][4], Vec4x1_F32* vec_a,
size_t num_vec);

// Ch13_08_BM.cpp
extern void Avx512MatVecMulF32_BM(void);

//------------------------------------------------
//               Ch13_08.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <cmath>
#include "Ch13_08.h"
#include "AlignedMem.h"

using namespace std;

bool VecCompare(const Vec4x1_F32* v1, const Vec4x1_F32* v2)
{
    static const float eps = 1.0e-12f;

    bool b0 = (fabs(v1->W - v2->W) <= eps);
    bool b1 = (fabs(v1->X - v2->X) <= eps);
    bool b2 = (fabs(v1->Y - v2->Y) <= eps);
    bool b3 = (fabs(v1->Z - v2->Z) <= eps);

    return b0 && b1 && b2 && b3;
}
```

```
void InitVecArray(Vec4x1_F32* va, size_t num_vec)
{
    uniform_int_distribution<> ui_dist {1, 500};
    default_random_engine rng {187};

    for (size_t i = 0; i < num_vec; i++)
    {
        va[i].W = (float)ui_dist(rng);
        va[i].X = (float)ui_dist(rng);
        va[i].Y = (float)ui_dist(rng);
        va[i].Z = (float)ui_dist(rng);
    }

    if (num_vec >= 4)
    {
        // Test values
        va[0].W =  5; va[0].X =  6; va[0].Y =  7; va[0].Z =  8;
        va[1].W = 15; va[1].X = 16; va[1].Y = 17; va[1].Z = 18;
        va[2].W = 25; va[2].X = 26; va[2].Y = 27; va[2].Z = 28;
        va[3].W = 35; va[3].X = 36; va[3].Y = 37; va[3].Z = 38;
    }
}

bool Avx512MatVecMulF32Cpp(Vec4x1_F32* vec_b, float mat[4][4], Vec4x1_F32* vec_a, size_t
num_vec)
{
    if (num_vec == 0 || num_vec % 4 != 0)
        return false;

    if (!AlignedMem::IsAligned(vec_a, 64) || !AlignedMem::IsAligned(vec_b, 64))
        return false;

    if (!AlignedMem::IsAligned(mat, 64))
        return false;

    for (size_t i = 0; i < num_vec; i++)
    {
        vec_b[i].W =  mat[0][0] * vec_a[i].W + mat[0][1] * vec_a[i].X;
        vec_b[i].W += mat[0][2] * vec_a[i].Y + mat[0][3] * vec_a[i].Z;

        vec_b[i].X =  mat[1][0] * vec_a[i].W + mat[1][1] * vec_a[i].X;
        vec_b[i].X += mat[1][2] * vec_a[i].Y + mat[1][3] * vec_a[i].Z;

        vec_b[i].Y =  mat[2][0] * vec_a[i].W + mat[2][1] * vec_a[i].X;
        vec_b[i].Y += mat[2][2] * vec_a[i].Y + mat[2][3] * vec_a[i].Z;

        vec_b[i].Z =  mat[3][0] * vec_a[i].W + mat[3][1] * vec_a[i].X;
        vec_b[i].Z += mat[3][2] * vec_a[i].Y + mat[3][3] * vec_a[i].Z;
    }

    return true;
}
```

```
void Avx512MatVecMulF32(void)
{
    const size_t num_vec = 8;

    alignas(64) float mat[4][4]
    {
       10.0, 11.0, 12.0, 13.0,
       20.0, 21.0, 22.0, 23.0,
       30.0, 31.0, 32.0, 33.0,
       40.0, 41.0, 42.0, 43.0
    };

    AlignedArray<Vec4x1_F32> vec_a_aa(num_vec, 64);
    AlignedArray<Vec4x1_F32> vec_b1_aa(num_vec, 64);
    AlignedArray<Vec4x1_F32> vec_b2_aa(num_vec, 64);

    Vec4x1_F32* vec_a = vec_a_aa.Data();
    Vec4x1_F32* vec_b1 = vec_b1_aa.Data();
    Vec4x1_F32* vec_b2 = vec_b2_aa.Data();

    InitVecArray(vec_a, num_vec);

    bool rc1 = Avx512MatVecMulF32Cpp(vec_b1, mat, vec_a, num_vec);
    bool rc2 = Avx512MatVecMulF32_(vec_b2, mat, vec_a, num_vec);

    cout << "Results for Avx512MatVecMulF32\n";

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code\n";
        cout << "  rc1 = " << boolalpha << rc1 << '\n';
        cout << "  rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    const unsigned int w = 8;
    cout << fixed << setprecision(1);

    for (size_t i = 0; i < num_vec; i++)
    {
        cout << "Test case #" << i << '\n';

        cout << "vec_b1: ";
        cout << "  " << setw(w) << vec_b1[i].W << ' ';
        cout << "  " << setw(w) << vec_b1[i].X << ' ';
        cout << "  " << setw(w) << vec_b1[i].Y << ' ';
        cout << "  " << setw(w) << vec_b1[i].Z << '\n';

        cout << "vec_b2: ";
        cout << "  " << setw(w) << vec_b2[i].W << ' ';
        cout << "  " << setw(w) << vec_b2[i].X << ' ';
        cout << "  " << setw(w) << vec_b2[i].Y << ' ';
```

```
        cout << "  " << setw(w) << vec_b2[i].Z << '\n';

        if (!VecCompare(&vec_b1[i], &vec_b2[i]))
        {
            cout << "Error - vector compare failed\n";
            return;
        }
    }
}

int main()
{
    Avx512MatVecMulF32();
    Avx512MatVecMulF32_BM();
    return 0;
}

;-------------------------------------------------
;               Ch13_08.asm
;-------------------------------------------------

ConstVals   segment readonly align(64) 'const'
; Indices for matrix permutations
MatPerm0    dword 0, 4, 8, 12, 0, 4, 8, 12, 0, 4, 8, 12, 0, 4, 8, 12
MatPerm1    dword 1, 5, 9, 13, 1, 5, 9, 13, 1, 5, 9, 13, 1, 5, 9, 13
MatPerm2    dword 2, 6, 10, 14, 2, 6, 10, 14, 2, 6, 10, 14, 2, 6, 10, 14
MatPerm3    dword 3, 7, 11, 15, 3, 7, 11, 15, 3, 7, 11, 15, 3, 7, 11, 15

; Indices for vector permutations
VecPerm0    dword 0, 0, 0, 0, 4, 4, 4, 4, 8, 8, 8, 8, 12, 12, 12, 12
VecPerm1    dword 1, 1, 1, 1, 5, 5, 5, 5, 9, 9, 9, 9, 13, 13, 13, 13
VecPerm2    dword 2, 2, 2, 2, 6, 6, 6, 6, 10, 10, 10, 10, 14, 14, 14, 14
VecPerm3    dword 3, 3, 3, 3, 7, 7, 7, 7, 11, 11, 11, 11, 15, 15, 15, 15
ConstVals   ends

; extern "C" bool Avx512MatVecMulF32_(Vec4x1_F32* vec_b, float mat[4][4], Vec4x1_F32* vec_a,
size_t num_vec);

        .code
Avx512MatVecMulF32_ proc
        xor eax,eax                         ;set error code (also i = 0)
        test r9,r9
        jz Done                             ;jump if num_vec is zero
        test r9,3
        jnz Done                            ;jump if n % 4 != 0

        test rcx,3fh
        jnz Done                            ;jump if vec_b is not properly aligned
        test rdx,3fh
        jnz Done                            ;jump if mat is not properly aligned
        test r8,3fh
        jnz Done                            ;jump if vec_a is not properly aligned
```

```
; Load permutation indices for matrix columns and vector elements
        vmovdqa32 zmm16,zmmword ptr [MatPerm0]  ;mat col 0 indices
        vmovdqa32 zmm17,zmmword ptr [MatPerm1]  ;mat col 1 indices
        vmovdqa32 zmm18,zmmword ptr [MatPerm2]  ;mat col 2 indices
        vmovdqa32 zmm19,zmmword ptr [MatPerm3]  ;mat col 3 indices

        vmovdqa32 zmm24,zmmword ptr [VecPerm0]  ;W component indices
        vmovdqa32 zmm25,zmmword ptr [VecPerm1]  ;X component indices
        vmovdqa32 zmm26,zmmword ptr [VecPerm2]  ;Y component indices
        vmovdqa32 zmm27,zmmword ptr [VecPerm3]  ;Z component indices

; Load source matrix and duplicate columns
        vmovaps zmm0,zmmword ptr [rdx]      ;zmm0 = mat

        vpermps zmm20,zmm16,zmm0            ;zmm20 = mat col 0 (4x)
        vpermps zmm21,zmm17,zmm0            ;zmm21 = mat col 1 (4x)
        vpermps zmm22,zmm18,zmm0            ;zmm22 = mat col 2 (4x)
        vpermps zmm23,zmm19,zmm0            ;zmm23 = mat col 3 (4x)

; Load the next 4 vectors
        align 16
@@:     vmovaps zmm4,zmmword ptr [r8+rax]   ;zmm4 = vec_a (4 vectors)

; Permute the vector elements for subsequent calculations
        vpermps zmm0,zmm24,zmm4             ;zmm0 = vec_a W components
        vpermps zmm1,zmm25,zmm4             ;zmm1 = vec_a X components
        vpermps zmm2,zmm26,zmm4             ;zmm2 = vec_a Y components
        vpermps zmm3,zmm27,zmm4             ;zmm3 = vec_a Z components

; Perform matrix-vector multiplications (4 vectors)
        vmulps zmm28,zmm20,zmm0
        vmulps zmm29,zmm21,zmm1
        vmulps zmm30,zmm22,zmm2
        vmulps zmm31,zmm23,zmm3
        vaddps zmm4,zmm28,zmm29
        vaddps zmm5,zmm30,zmm31
        vaddps zmm4,zmm4,zmm5               ;zmm4 = vec_b (4 vectors)

        vmovaps zmmword ptr [rcx+rax],zmm4  ;save result

        add rax,64                          ;rax = offset to next block of 4 vectors
        sub r9,4
        jnz @B                              ;repeat until done

        mov eax,1                           ;set success code

Done:   vzeroupper
        ret
Avx512MatVecMulF32_ endp
        end
```

The C++ code in Listing 13-8 begins with header file `Ch13_08.h` that contains the requisite function declarations. This file also includes a declaration for the structure `Vec4x1_F32`, which incorporates the four components of a 4 × 1 column vector. The source code file `Ch13_08.cpp` includes a function named `Avx512MatVecMulF32Cpp`. This function implements the matrix-vector multiplication equations that are shown in Figure 13-5. The remaining C++ code in Listing 13-8 performs test case initializations, invokes the calculating functions, and displays the results.

The assembly language code in Listing 13-8 starts with a constant data segment that defines a series of packed permutation indices. The assembly language implementation of the matrix-vector multiplication algorithm uses these values to reorder the elements of the source matrix and vectors. The reason for this reordering is to facilitate the simultaneous calculation of four matrix-vector products. The function `Avx512MatVecMulF32_` begins its execution by validating `num_vec` for divisibility by four. It then checks the matrix and vector buffer pointers for proper alignment on a 64-byte boundary.

Following argument validation, four `vmovdqa32` instructions load the matrix permutation indices into registers ZMM16–ZMM19. This is followed by another series of four `vmovdqa32` instructions that load the vector permutation indices into registers ZMM24–ZMM27. The ensuing `vmovaps zmm0, zmmword ptr [rdx]` instruction loads all 16 single-precision floating-point elements of matrix `mat` into ZMM0. The `vpermps zmm20,zmm16,zmm0` instruction (Permute Single-Precision Floating-Point Elements) rearranges the elements in ZMM0 according to the indices in ZMM16. Execution of this instruction loads four copies of column 0 from matrix `mat` into register ZMM20. Three more `vpermps` instructions are then employed to perform the same operation using columns 1, 2, and 3. Figure 13-6 elucidates the execution of these permutations in greater detail.

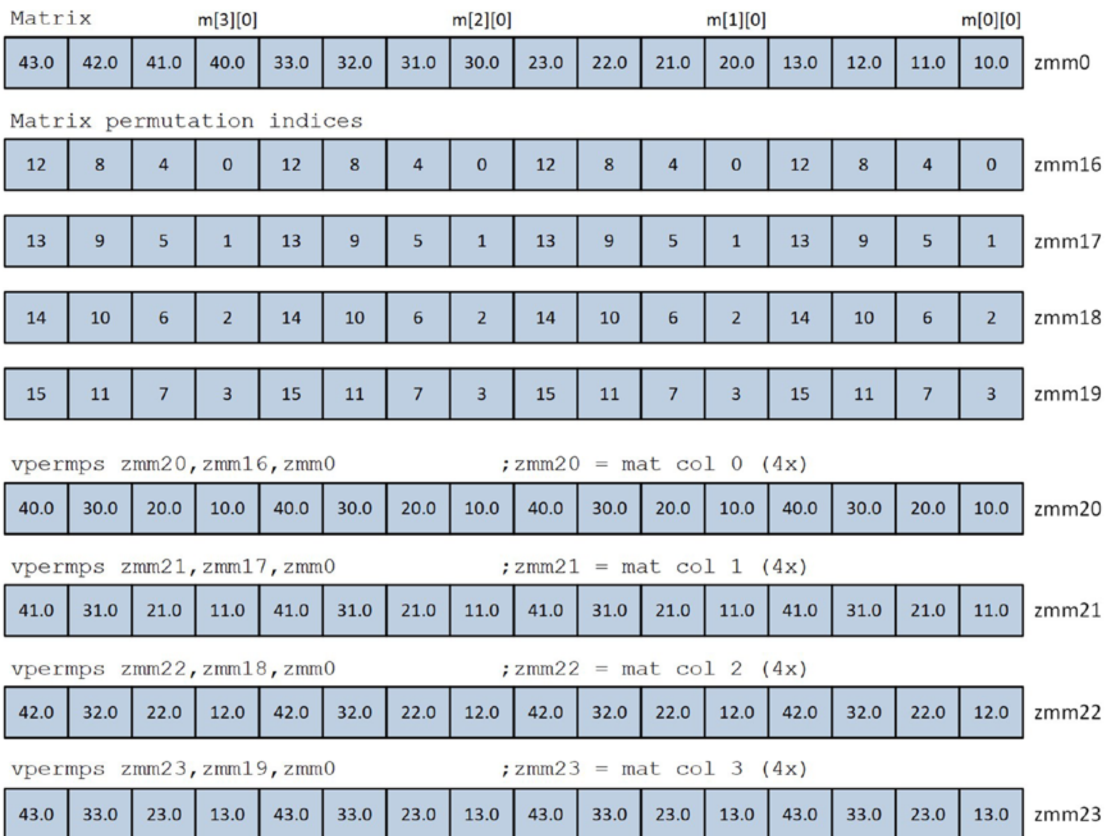



***Figure 13-6.*** *Permutation of matrix columns using `vpermps` instructions*

The processing loop in `Avx512MatVecMulF32_` starts with a `vmovaps zmm4,zmmword ptr [r8+rax]` instruction that loads four `Vec4x1_F32` vectors into register ZMM4. The `W`, `X`, `Y`, and `Z` components of these vectors are then regrouped using another series of `vpermps` instructions. Following execution of these instructions, registers ZMM0–ZMM3 contain repeated sets of the vector components, as shown in Figure 13-7.

Vectors

| a[3].Z | a[3].Y | a[3].X | a[3].W | a[2].Z | a[2].Y | a[2].X | a[2].W | a[1].Z | a[1].Y | a[1].X | a[1].W | a[0].Z | a[0].Y | a[0].X | a[0].W | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 38.0 | 37.0 | 36.0 | 35.0 | 28.0 | 27.0 | 26.0 | 25.0 | 18.0 | 17.0 | 16.0 | 15.0 | 8.0 | 7.0 | 6.0 | 5.0 | zmm4 |

Vector permutation indices

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 12 | 12 | 12 | 8 | 8 | 8 | 8 | 4 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | zmm24 |
| 13 | 13 | 13 | 13 | 9 | 9 | 9 | 9 | 5 | 5 | 5 | 5 | 1 | 1 | 1 | 1 | zmm25 |
| 14 | 14 | 14 | 14 | 10 | 10 | 10 | 10 | 6 | 6 | 6 | 6 | 2 | 2 | 2 | 2 | zmm26 |
| 15 | 15 | 15 | 15 | 11 | 11 | 11 | 11 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 | zmm27 |

```
vpermps zmm0,zmm24,zmm4             ;zmm0 = vec_a W components
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 35.0 | 35.0 | 35.0 | 35.0 | 25.0 | 25.0 | 25.0 | 25.0 | 15.0 | 15.0 | 15.0 | 15.0 | 5.0 | 5.0 | 5.0 | 5.0 | zmm0 |

```
vpermps zmm1,zmm25,zmm4             ;zmm1 = vec_a X components
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 36.0 | 36.0 | 36.0 | 36.0 | 26.0 | 26.0 | 26.0 | 26.0 | 16.0 | 16.0 | 16.0 | 16.0 | 6.0 | 6.0 | 6.0 | 6.0 | zmm1 |

```
vpermps zmm2,zmm26,zmm4             ;zmm2 = vec_a Y components
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 37.0 | 37.0 | 37.0 | 37.0 | 27.0 | 27.0 | 27.0 | 27.0 | 17.0 | 17.0 | 17.0 | 17.0 | 7.0 | 7.0 | 7.0 | 7.0 | zmm2 |

```
vpermps zmm3,zmm27,zmm4             ;zmm3 = vec_a Z components
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 38.0 | 38.0 | 38.0 | 38.0 | 28.0 | 28.0 | 28.0 | 28.0 | 18.0 | 18.0 | 18.0 | 18.0 | 8.0 | 8.0 | 8.0 | 8.0 | zmm3 |

***Figure 13-7.*** *Permutation of vector components using `vpermps` instructions*

Following the vector component permutations, a series of `vmulps` and `vaddps` instructions carry out four simultaneous matrix-vector multiplications. Figure 13-8 illustrates this operation in greater detail. The ensuing `vmovaps zmmword ptr [rcx+rax],zmm4` instruction saves the four resultant 4 × 1 vectors in the `vec_b` array. The processing loop then repeats until all vectors in `vec_a` have been processed.

```
Matrix values (zmm20 - zmm23)
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40.0 | 30.0 | 20.0 | 10.0 | 40.0 | 30.0 | 20.0 | 10.0 | 40.0 | 30.0 | 20.0 | 10.0 | 40.0 | 30.0 | 20.0 | 10.0 | zmm20 |
| 41.0 | 31.0 | 21.0 | 11.0 | 41.0 | 31.0 | 21.0 | 11.0 | 41.0 | 31.0 | 21.0 | 11.0 | 41.0 | 31.0 | 21.0 | 11.0 | zmm21 |
| 42.0 | 32.0 | 22.0 | 12.0 | 42.0 | 32.0 | 22.0 | 12.0 | 42.0 | 32.0 | 22.0 | 12.0 | 42.0 | 32.0 | 22.0 | 12.0 | zmm22 |
| 43.0 | 33.0 | 23.0 | 13.0 | 43.0 | 33.0 | 23.0 | 13.0 | 43.0 | 33.0 | 23.0 | 13.0 | 43.0 | 33.0 | 23.0 | 13.0 | zmm23 |

```
Vector components (zmm0 - zmm3)
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 35.0 | 35.0 | 35.0 | 35.0 | 25.0 | 25.0 | 25.0 | 25.0 | 15.0 | 15.0 | 15.0 | 15.0 | 5.0 | 5.0 | 5.0 | 5.0 | zmm0 |
| 36.0 | 36.0 | 36.0 | 36.0 | 26.0 | 26.0 | 26.0 | 26.0 | 16.0 | 16.0 | 16.0 | 16.0 | 6.0 | 6.0 | 6.0 | 6.0 | zmm1 |
| 37.0 | 37.0 | 37.0 | 37.0 | 27.0 | 27.0 | 27.0 | 27.0 | 17.0 | 17.0 | 17.0 | 17.0 | 7.0 | 7.0 | 7.0 | 7.0 | zmm2 |
| 38.0 | 38.0 | 38.0 | 38.0 | 28.0 | 28.0 | 28.0 | 28.0 | 18.0 | 18.0 | 18.0 | 18.0 | 8.0 | 8.0 | 8.0 | 8.0 | zmm3 |

```
vmulps zmm28,zmm20,zmm0
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1400.0 | 1050.0 | 700.0 | 350.0 | 1000.0 | 750.0 | 500.0 | 250.0 | 600.0 | 450.0 | 300.0 | 150.0 | 200.0 | 150.0 | 100.0 | 50.0 | zmm28 |

```
vmulps zmm29,zmm21,zmm1
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1476.0 | 1116.0 | 756.0 | 396.0 | 1066.0 | 806.0 | 546.0 | 286.0 | 656.0 | 496.0 | 336.0 | 176.0 | 246.0 | 186.0 | 126.0 | 66.0 | zmm29 |

```
vmulps zmm30,zmm22,zmm2
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1554.0 | 1184.0 | 814.0 | 444.0 | 1134.0 | 864.0 | 594.0 | 324.0 | 714.0 | 544.0 | 374.0 | 204.0 | 294.0 | 224.0 | 154.0 | 84.0 | zmm30 |

```
vmulps zmm31,zmm23,zmm3
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1634.0 | 1254.0 | 874.0 | 494.0 | 1204.0 | 924.0 | 644.0 | 364.0 | 774.0 | 594.0 | 414.0 | 234.0 | 344.0 | 264.0 | 184.0 | 104.0 | zmm31 |

```
vaddps zmm4,zmm28,zmm29
vaddps zmm5,zmm30,zmm31
vaddps zmm4,zmm4,zmm5                ;zmm4 = vec_b (4 vectors)
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6064.0 | 4604.0 | 3144.0 | 1684.0 | 4404.0 | 3344.0 | 2284.0 | 1224.0 | 2744.0 | 2084.0 | 1424.0 | 764.0 | 1084.0 | 824.0 | 564.0 | 304.0 | zmm4 |
| b[3].Z | b[3].Y | b[3].X | b[3].W | b[2].Z | b[2].Y | b[2].X | b[2].W | b[1].Z | b[1].Y | b[1].X | b[1].W | b[0].Z | b[0].Y | b[0].X | b[0].W | |

***Figure 13-8.*** *Matrix-vector multiplications using* `vmulps` *and* `vaddps`

The output for source code example `Ch13_08` follows this paragraph. Table 13-2 shows benchmark timing measurements for the C++ and assembly language matrix-vector multiplication functions.

```
Results for Avx512MatVecMulF32
Test case #0
vec_b1:      304.0      564.0      824.0     1084.0
vec_b2:      304.0      564.0      824.0     1084.0
Test case #1
vec_b1:      764.0     1424.0     2084.0     2744.0
vec_b2:      764.0     1424.0     2084.0     2744.0
Test case #2
vec_b1:     1224.0     2284.0     3344.0     4404.0
vec_b2:     1224.0     2284.0     3344.0     4404.0
Test case #3
vec_b1:     1684.0     3144.0     4604.0     6064.0
vec_b2:     1684.0     3144.0     4604.0     6064.0
Test case #4
vec_b1:    11932.0    22452.0    32972.0    43492.0
vec_b2:    11932.0    22452.0    32972.0    43492.0
Test case #5
vec_b1:    17125.0    31705.0    46285.0    60865.0
vec_b2:    17125.0    31705.0    46285.0    60865.0
Test case #6
vec_b1:    12723.0    23873.0    35023.0    46173.0
vec_b2:    12723.0    23873.0    35023.0    46173.0
Test case #7
vec_b1:    15121.0    27871.0    40621.0    53371.0
vec_b2:    15121.0    27871.0    40621.0    53371.0

Running benchmark function Avx512MatVecMulF32_BM - please wait
Benchmark times save to file Ch13_08_Avx512MatVecMulF32_BM_CHROMIUM.csv
```

***Table 13-2.*** *Benchmark Timing Measurements for Matrix-Vector Multiplication Functions (1,000,000 Vectors)*

| CPU | Avx512MatVecMulF32Cpp | Avx512MatVecMulF32_ |
|---|---|---|
| i7-4790S | ---- | ---- |
| i9-7900X | 6174 | 1778 |
| i7-8700K | ---- | ---- |

## Convolutions

Listing 13-9 shows the source code for example Ch13_09. This example is an AVX-512 implementation of the convolution program that was presented in source code example Ch11_02. The primary purpose of this example is to highlight the conversion of functions that use AVX2 instructions to ones that exploit AVX-512 instructions. It also provides an opportunity to compare benchmark timing measurements between the AVX2 and AVX-512 implementations of the convolution functions.

***Listing 13-9.*** Example Ch13_09

```
;-------------------------------------------------
;               Ch13_09_.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword

; extern bool Avx512Convolve2_(float* y, const float* x, int num_pts, const float* kernel,
int kernel_size)

        .code
Avx512Convolve2_ proc frame
        _CreateFrame CV2_,0,0,rbx
        _EndProlog

; Validate argument values
        xor eax,eax                         ;set error code

        mov r10d,dword ptr [rbp+CV2_OffsetStackArgs]
        test r10d,1
        jz Done                             ;kernel_size is even
        cmp r10d,[c_KernelSizeMin]
        jl Done                             ;kernel_size too small
        cmp r10d,[c_KernelSizeMax]
        jg Done                             ;kernel_size too big

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;num_pts too big
        test r8d,15
        jnz Done                            ;num_pts not even multiple of 16

        test rcx,3fh
        jnz Done                            ;y is not properly aligned

; Initialize convolution loop variables
        shr r10d,1                          ;r10 = kernel_size / 2 (ks2)
        lea rdx,[rdx+r10*4]                 ;rdx = x + ks2 (first data point)
        xor ebx,ebx                         ;i = 0

; Perform convolution
LP1:    vxorps zmm0,zmm0,zmm0               ;packed sum = 0.0;
        mov r11,r10                         ;r11 = ks2
        neg r11                             ;k = -ks2
```

```
LP2:    mov rax,rbx                           ;rax = i
        sub rax,r11                           ;rax = i - k
        vmovups zmm1,zmmword ptr [rdx+rax*4]  ;load x[i - k]:x[i - k + 15]

        mov rax,r11
        add rax,r10                           ;rax = k + ks2
        vbroadcastss zmm2,real4 ptr [r9+rax*4] ;zmm2 = kernel[k + ks2]
        vfmadd231ps zmm0,zmm1,zmm2            ;zmm0 += x[i-k]:x[i-k+15] * kernel[k+ks2]

        add r11,1                             ;k += 1
        cmp r11,r10
        jle LP2                               ;repeat until k > ks2

        vmovaps zmmword ptr [rcx+rbx*4],zmm0  ;save y[i]:y[i + 15]

        add rbx,16                           ;i += 16
        cmp rbx,r8
        jl LP1                              ;repeat until done
        mov eax,1                           ;set success return code

Done:   vzeroupper
        _DeleteFrame rbx
        ret
Avx512Convolve2_ endp

; extern bool Avx512Convolve2Ks5_(float* y, const float* x, int num_pts, const float*
kernel, int kernel_size)

Avx512Convolve2Ks5_ proc frame
        _CreateFrame CKS5_,0,48
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Validate argument values
        xor eax,eax                         ;set error code (rax is also loop index var)

        cmp dword ptr [rbp+CKS5_OffsetStackArgs],5
        jne Done                            ;jump if kernel_size is not 5

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;jump if num_pts too small
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;jump if num_pts too big
        test r8d,15
        jnz Done                            ;num_pts not even multiple of 15

        test rcx,3fh
        jnz Done                            ;y is not properly aligned
```

```
; Perform required initializations
        vbroadcastss zmm4,real4 ptr [r9]        ;kernel[0]
        vbroadcastss zmm5,real4 ptr [r9+4]      ;kernel[1]
        vbroadcastss zmm6,real4 ptr [r9+8]      ;kernel[2]
        vbroadcastss zmm7,real4 ptr [r9+12]     ;kernel[3]
        vbroadcastss zmm8,real4 ptr [r9+16]     ;kernel[4]

        mov r8d,r8d                             ;r8 = num_pts
        add rdx,8                               ;x += 2

; Perform convolution
@@:     vxorps zmm2,zmm2,zmm2                   ;initialize sum vars
        vxorps zmm3,zmm3,zmm3

        mov r11,rax
        add r11,2                               ;j = i + ks2

        vmovups zmm0,zmmword ptr [rdx+r11*4]    ;zmm0 = x[j]:x[j + 15]
        vfmadd231ps zmm2,zmm0,zmm4              ;zmm2 += x[j]:x[j + 15] * kernel[0]

        vmovups zmm1,zmmword ptr [rdx+r11*4-4]  ;zmm1 = x[j - 1]:x[j + 14]
        vfmadd231ps zmm3,zmm1,zmm5              ;zmm3 += x[j - 1]:x[j + 14] * kernel[1]

        vmovups zmm0,zmmword ptr [rdx+r11*4-8]  ;zmm0 = x[j - 2]:x[j + 13]
        vfmadd231ps zmm2,zmm0,zmm6              ;zmm2 += x[j - 2]:x[j + 13] * kernel[2]

        vmovups zmm1,zmmword ptr [rdx+r11*4-12] ;zmm1 = x[j - 3]:x[j + 12]
        vfmadd231ps zmm3,zmm1,zmm7              ;zmm3 += x[j - 3]:x[j + 12] * kernel[3]

        vmovups zmm0,zmmword ptr [rdx+r11*4-16] ;zmm0 = x[j - 4]:x[j + 11]
        vfmadd231ps zmm2,zmm0,zmm8              ;zmm2 += x[j - 4]:x[j + 11] * kernel[4]

        vaddps zmm0,zmm2,zmm3                   ;final values
        vmovaps zmmword ptr [rcx+rax*4],zmm0    ;save y[i]:y[i + 15]

        add rax,16                               ;i += 16
        cmp rax,r8
        jl @B                                   ;jump if i < num_pts
        mov eax,1                               ;set success return code

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame
        ret
Avx512Convolve2Ks5_ endp
        end
```

The C++ portion of source code example Ch13_09 is not shown in Listing 13-9 since it's almost identical to the C++ code in example Ch11_02. Modifications made in the Ch13_09 C++ code include a few function name changes. The test arrays are also allocated on a 64-byte instead of a 32-byte boundary.

The assembly language function Avx512Convolve2_ implements the variable-size kernel convolution algorithm that's described in Chapter 11. The primary difference between this function and its AVX2 counterpart Convolve2_ (see Listing 11-2) is the use of ZMM registers instead of YMM registers. The code that adjusts the index counter in register RBX was also modified to reflect the processing of 16 data points per iteration instead of 8. Similar changes were also made to the fixed-size kernel convolution function Avx512Convolve2Ks5_.

The output for source code example Ch13_09 is not shown since it's the same as the output for source code example Ch11_02. Table 13-3 shows the benchmark timing measurements for functions Avx512Convolve2_ and Avx512Convolve2Ks5_. This table also includes the benchmark timing measurements for the AVX2 functions Convolve2_ and ConvolveKs2_ from Table 11-2. The AVX-512 implementations are faster than their AVX2 counterparts, especially for the size-independent convolution function Avx512Convolve2_. It would, of course, be imprudent to extrapolate any general conclusions regarding AVX-512 versus AVX2 performance based solely on the timing measurements shown in Table 13-3. You'll see other examples in Chapter 14.

***Table 13-3.*** *Mean Execution Times (Microseconds) for AVX2 and AVX-512 Convolution Functions Using Five-Element Convolution Kernel (2,000,000 Signal Points)*

| CPU | Convolve2_ | Avx512Convolve2_ | Convolve2Ks5_ | Avx512Convolve2Ks5_ |
|---|---|---|---|---|
| i7-4790S | 1244 | ----- | 1067 | ---- |
| i9-7900X | 956 | 757 | 719 | 693 |
| i7-8700K | 859 | ----- | 595 | ---- |

# Summary

Here the key learning points for Chapter 13.

- When using merge masking with scalar or packed operands, the processor carries out the instruction's calculation only if the corresponding opmask register bit is set to one. Otherwise, no calculation is performed and the destination operand element remains unchanged.
- AVX-512 assembly language functions can use an opmask register destination operand with most instructions that perform scalar or packed compare operations. The bits of the opmask register can then be employed to effect data-driven logic decisions sans any conditional jump instructions using either merge or zero masking and (if necessary) simple Boolean operations.
- AVX-512 assembly language functions must use the vmovdqu[32|64] and vmovdqa[32|64] instructions to perform move operations using 512-bit wide packed doubleword and quadword integer operands. These instructions can also be used with 256-bit and 128-bit wide operands.
- Unlike AVX and AVX2, AVX-512 includes instructions that perform conversions between floating-point and unsigned integer operands.
- AVX-512 functions should ensure that packed 128-, 256-, and 512-bit wide operands are aligned on a proper boundary whenever possible.

- Assembly language functions that use AVX-512 instructions with registers ZMM0–ZMM15 or YMM0–YMM15 register operands should always use a `vzeroupper` instruction before program control is transferred back to the calling function.
- Assembly language functions and algorithms that employ a structure of arrays are often faster than those that use an array of structures.
- The Visual C++ calling convention treats AVX-512 registers ZMM16–ZMM31, YMM16–YMM31, and XMM16–XMM31 as volatile across function boundaries. This means that a function can use these registers without needing to preserve their values.

CHAPTER 14

# AVX-512 Programming – Packed Integers

In Chapters 7 and 10, you learned how to use the AVX and AVX2 instruction sets to perform packed integer operations using 128-bit and 256-bit wide operands. In this chapter, you learn how to use AVX-512 instructions set to carry out packed integer operations using 512-bit wide operands. You also learn how to use AVX-512 instructions with 256-bit and 128-bit wide packed integer operands. The first source code example explains how to perform basic packed integer arithmetic using ZMM registers. This is followed by several examples that exemplify image-processing algorithms and techniques using AVX-512 instructions. Like the previous chapter, all of source code examples in this chapter require a processor and operating system that support AVX-512 and the following instruction set extensions: AVX512F, AVX512CD, AVX512BW, AVX512DQ, and AVX512VL. You can use one of the freely available utilities listed in Appendix A to determine whether your system supports these extensions.

## Basic Arithmetic

Listing 14-1 shows the source code for example `Ch14_01`. This example demonstrates how to perform basic packed integer arithmetic using 512-bit wide operands and the ZMM register set.

***Listing 14-1.*** Example Ch14_01

```
//------------------------------------------------
//               Ch14_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include "Zmmval.h"

using namespace std;

extern "C" void Avx512PackedMathI16_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[6]);
extern "C" void Avx512PackedMathI64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[5],
uint32_t opmask);
```

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_14

```
void Avx512PackedMathI16(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    alignas(64) ZmmVal c[6];

    a.m_I16[0] = 10;       b.m_I16[0] = 100;
    a.m_I16[1] = 20;       b.m_I16[1] = 200;
    a.m_I16[2] = 30;       b.m_I16[2] = 300;
    a.m_I16[3] = 40;       b.m_I16[3] = 400;
    a.m_I16[4] = 50;       b.m_I16[4] = 500;
    a.m_I16[5] = 60;       b.m_I16[5] = 600;
    a.m_I16[6] = 70;       b.m_I16[6] = 700;
    a.m_I16[7] = 80;       b.m_I16[7] = 800;

    a.m_I16[8] = 1000;     b.m_I16[8] = -100;
    a.m_I16[9] = 2000;     b.m_I16[9] = 200;
    a.m_I16[10] = 3000;    b.m_I16[10] = -300;
    a.m_I16[11] = 4000;    b.m_I16[11] = 400;
    a.m_I16[12] = 5000;    b.m_I16[12] = -500;
    a.m_I16[13] = 6000;    b.m_I16[13] = 600;
    a.m_I16[14] = 7000;    b.m_I16[14] = -700;
    a.m_I16[15] = 8000;    b.m_I16[15] = 800;

    a.m_I16[16] = -1000;   b.m_I16[16] = 100;
    a.m_I16[17] = -2000;   b.m_I16[17] = -200;
    a.m_I16[18] = 3000;    b.m_I16[18] = 303;
    a.m_I16[19] = 4000;    b.m_I16[19] = -400;
    a.m_I16[20] = -5000;   b.m_I16[20] = 500;
    a.m_I16[21] = -6000;   b.m_I16[21] = -600;
    a.m_I16[22] = -7000;   b.m_I16[22] = 700;
    a.m_I16[23] = -8000;   b.m_I16[23] = 800;

    a.m_I16[24] = 30000;   b.m_I16[24] = 3000;      // add overflow
    a.m_I16[25] = 6000;    b.m_I16[25] = 32000;     // add overflow
    a.m_I16[26] = -25000;  b.m_I16[26] = -27000;    // add overflow
    a.m_I16[27] = 8000;    b.m_I16[27] = 28700;     // add overflow
    a.m_I16[28] = 2000;    b.m_I16[28] = -31000;    // sub overflow
    a.m_I16[29] = 4000;    b.m_I16[29] = -30000;    // sub overflow
    a.m_I16[30] = -3000;   b.m_I16[30] = 32000;     // sub overflow
    a.m_I16[31] = -15000;  b.m_I16[31] = 24000;     // sub overflow

    Avx512PackedMathI16_(&a, &b, c);

    cout <<"\nResults for Avx512PackedMathI16\n\n";
    cout << " i        a        b   vpaddw  vpaddsw   vpsubw  vpsubsw  vpminsw  vpmaxsw\n";
    cout << "-------------------------------------------------------------------------\n";
```

```
    for (int i = 0; i < 32; i++)
    {
        cout << setw(2)  << i << ' ';
        cout << setw(8) << a.m_I16[i] << ' ';
        cout << setw(8) << b.m_I16[i] << ' ';
        cout << setw(8) << c[0].m_I16[i] << ' ';
        cout << setw(8) << c[1].m_I16[i] << ' ';
        cout << setw(8) << c[2].m_I16[i] << ' ';
        cout << setw(8) << c[3].m_I16[i] << ' ';
        cout << setw(8) << c[4].m_I16[i] << ' ';
        cout << setw(8) << c[5].m_I16[i] << '\n';
    }
}

void Avx512PackedMathI64(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    alignas(64) ZmmVal c[6];
    uint32_t opmask = 0x7f;

    a.m_I64[0] = 64;        b.m_I64[0] = 4;
    a.m_I64[1] = 1024;      b.m_I64[1] = 5;
    a.m_I64[2] = -2048;     b.m_I64[2] = 2;
    a.m_I64[3] = 8192;      b.m_I64[3] = 5;
    a.m_I64[4] = -256;      b.m_I64[4] = 8;
    a.m_I64[5] = 4096;      b.m_I64[5] = 7;
    a.m_I64[6] = 16;        b.m_I64[6] = 3;
    a.m_I64[7] = 512;       b.m_I64[7] = 6;

    Avx512PackedMathI64_(&a, &b, c, opmask);

    cout << "\nResults for Avx512PackedMathI64\n\n";
    cout << "op_mask = " << hex << opmask << dec << '\n';
    cout << " i      a      b   vpaddq   vpsubq  vpmullq  vpsllvq  vpsravq   vpabsq\n";
    cout << "---------------------------------------------------------------------\n";

    for (int i = 0; i < 8; i++)
    {
        cout << setw(2) << i << ' ';
        cout << setw(6) << a.m_I64[i] << ' ';
        cout << setw(6) << b.m_I64[i] << ' ';
        cout << setw(8) << c[0].m_I64[i] << ' ';
        cout << setw(8) << c[1].m_I64[i] << ' ';
        cout << setw(8) << c[2].m_I64[i] << ' ';
        cout << setw(8) << c[3].m_I64[i] << ' ';
        cout << setw(8) << c[4].m_I64[i] << ' ';
        cout << setw(8) << c[5].m_I64[i] << '\n';
    }
}
```

```
int main()
{
    Avx512PackedMathI16();
    Avx512PackedMathI64();
    return 0;
}

;-------------------------------------------------
;               Ch14_01.asm
;-------------------------------------------------

; extern "C" void Avx512PackedMathI16_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[6])

        .code
Avx512PackedMathI16_ proc
        vmovdqu16 zmm0,zmmword ptr [rcx]        ;zmm0 = a
        vmovdqu16 zmm1,zmmword ptr [rdx]        ;zmm1 = b

; Perform packed word operations
        vpaddw zmm2,zmm0,zmm1                   ;add
        vmovdqa64 zmmword ptr [r8],zmm2         ;save vpaddw result

        vpaddsw zmm2,zmm0,zmm1                  ;add with signed saturation
        vmovdqa64 zmmword ptr [r8+64],zmm2      ;save vpaddsw result

        vpsubw zmm2,zmm0,zmm1                   ;sub
        vmovdqa64 zmmword ptr [r8+128],zmm2     ;save vpsubw result

        vpsubsw zmm2,zmm0,zmm1                  ;sub with signed saturation
        vmovdqa64 zmmword ptr [r8+192],zmm2     ;save vpsubsw result

        vpminsw zmm2,zmm0,zmm1                  ;signed minimums
        vmovdqa64 zmmword ptr [r8+256],zmm2     ;save vpminsw result

        vpmaxsw zmm2,zmm0,zmm1                  ;signed maximums
        vmovdqa64 zmmword ptr [r8+320],zmm2     ;save vpmaxsw result

        vzeroupper
        ret
Avx512PackedMathI16_ endp

; extern "C" void Avx512PackedMathI64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[5],
unsigned int opmask)

Avx512PackedMathI64_ proc
        vmovdqa64 zmm0,zmmword ptr [rcx]        ;zmm0 = a
        vmovdqa64 zmm1,zmmword ptr [rdx]        ;zmm1 = b

        and r9d,0ffh                            ;r9d = opmask value
        kmovb k1,r9d                            ;k1 = opmask
```

```
; Perform packed quadword operations
        vpaddq zmm2{k1}{z},zmm0,zmm1            ;add
        vmovdqa64 zmmword ptr [r8],zmm2         ;save vpaddq result

        vpsubq zmm2{k1}{z},zmm0,zmm1            ;sub
        vmovdqa64 zmmword ptr [r8+64],zmm2      ;save vpsubq result

        vpmullq zmm2{k1}{z},zmm0,zmm1           ;signed mul (low 64 bits)
        vmovdqa64 zmmword ptr [r8+128],zmm2     ;save vpmullq result

        vpsllvq zmm2{k1}{z},zmm0,zmm1           ;shift left logical
        vmovdqa64 zmmword ptr [r8+192],zmm2     ;save vpsllvq result

        vpsravq zmm2{k1}{z},zmm0,zmm1           ;shift right arithmetic
        vmovdqa64 zmmword ptr [r8+256],zmm2     ;save vpsravq result

        vpabsq zmm2{k1}{z},zmm0                 ;absolute value
        vmovdqa64 zmmword ptr [r8+320],zmm2     ;save vpabsq result

        vzeroupper
        ret
Avx512PackedMathI64_ endp
        end
```

The C++ functions `Avx512PackedMathI16` and `Avx512PackedMathI64` are the base routines that handle AVX-512 packed integer operations using word and quadword values. Each function begins its execution by initializing the applicable integer elements of two `ZmmVal` variables. Note that the C++ `alignas(64)` specifier is used with each `ZmmVal`. Following variable initialization, each base routine invokes its corresponding assembly language function: `Avx512PackedMathI16_` or `Avx512PackedMathI64_`. The results are then streamed to `cout`.

The assembly language function `Avx512PackedMathI16_` starts its execution with two `vmovdqa64` instructions that load `ZmmVal` variables `a` and `b` into registers ZMM0 and ZMM1, respectively. Somewhat surprisingly, AVX512BW does not include aligned move instructions for 512-bit wide packed byte and word operands. Another alternative here would be to use the `vmovdqu16` instruction. Note that this latter instruction must be used in cases where merge or zero masking is required. AVX512BW also includes a `vmovdqu8` instruction for 512-bit wide packed byte operands. Following operand value loading, `Avx512PackedMathI16_` demonstrates the packed word instructions `vpaddw`, `vpaddsw`, `vpsubw`, `vpsubsw`, `vpminsw`, `vpmaxsw`. Each 512-bit packed word result is then saved in the array `c`. Note that `Avx512PackedMathI16_` uses a `vzeroupper` instruction prior to its `ret` instruction.

The assembly language function `Avx512PackedMathI64_` exemplifies various arithmetic operations using 512-bit wide packed quadword instructions. Note that this function includes an argument value named `opmask`, which is employed to highlight packed quadword zero masking. `Avx512PackedMathI64_` also uses a `vzeroupper` instruction prior to its `ret` instruction. Here are the results for source code example `Ch14_01`.

```
Results for Avx512PackedMathI16

 i        a        b   vpaddw  vpaddsw   vpsubw  vpsubsw  vpminsw  vpmaxsw
--------------------------------------------------------------------------
 0       10      100      110      110      -90      -90       10      100
 1       20      200      220      220     -180     -180       20      200
 2       30      300      330      330     -270     -270       30      300
 3       40      400      440      440     -360     -360       40      400
 4       50      500      550      550     -450     -450       50      500
 5       60      600      660      660     -540     -540       60      600
 6       70      700      770      770     -630     -630       70      700
 7       80      800      880      880     -720     -720       80      800
 8     1000     -100      900      900     1100     1100     -100     1000
 9     2000      200     2200     2200     1800     1800      200     2000
10     3000     -300     2700     2700     3300     3300     -300     3000
11     4000      400     4400     4400     3600     3600      400     4000
12     5000     -500     4500     4500     5500     5500     -500     5000
13     6000      600     6600     6600     5400     5400      600     6000
14     7000     -700     6300     6300     7700     7700     -700     7000
15     8000      800     8800     8800     7200     7200      800     8000
16    -1000      100     -900     -900    -1100    -1100    -1000      100
17    -2000     -200    -2200    -2200    -1800    -1800    -2000     -200
18     3000      303     3303     3303     2697     2697      303     3000
19     4000     -400     3600     3600     4400     4400     -400     4000
20    -5000      500    -4500    -4500    -5500    -5500    -5000      500
21    -6000     -600    -6600    -6600    -5400    -5400    -6000     -600
22    -7000      700    -6300    -6300    -7700    -7700    -7000      700
23    -8000      800    -7200    -7200    -8800    -8800    -8000      800
24    30000     3000   -32536    32767    27000    27000     3000    30000
25     6000    32000   -27536    32767   -26000   -26000     6000    32000
26   -25000   -27000    13536   -32768     2000     2000   -27000   -25000
27     8000    28700   -28836    32767   -20700   -20700     8000    28700
28     2000   -31000   -29000   -29000   -32536    32767   -31000     2000
29     4000   -30000   -26000   -26000   -31536    32767   -30000     4000
30    -3000    32000    29000    29000    30536   -32768    -3000    32000
31   -15000    24000     9000     9000    26536   -32768   -15000    24000

Results for Avx512PackedMathI64

op_mask = 7f
 i      a      b   vpaddq   vpsubq  vpmullq  vpsllvq  vpsravq   vpabsq
----------------------------------------------------------------------
 0     64      4       68       60      256     1024        4       64
 1   1024      5     1029     1019     5120    32768       32     1024
 2  -2048      2    -2046    -2050    -4096    -8192     -512     2048
 3   8192      5     8197     8187    40960   262144      256     8192
 4   -256      8     -248     -264    -2048   -65536       -1      256
 5   4096      7     4103     4089    28672   524288       32     4096
 6     16      3       19       13       48      128        2       16
 7    512      6        0        0        0        0        0        0
```

# Image Processing

The source code examples in this section explicate image-processing algorithms and techniques using AVX-512 packed integer instructions. Most of the source code examples are updated versions of examples from earlier chapters that exploited AVX or AVX2 instructions. Besides exemplifying AVX-512 packed integer instruction usage, the source code examples that follow also accentuate alternative algorithmic approaches and instruction sequences that often result in improved performance.

## Pixel Conversions

In Chapter 7, you learned how to use the AVX instruction set to convert unsigned 8-bit pixels to single-precision floating-point pixels and vice versa (see example Ch07_06). Source code example Ch14_02 demonstrates how to carry out these same conversions using AVX-512 instructions. Listing 14-2 shows the source code for example Ch14_02.

***Listing 14-2.*** Example Ch14_02

```
//------------------------------------------------
//               Ch14_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>
#include <random>
#include "AlignedMem.h"

using namespace std;

// Ch14_02_Misc.cpp
extern bool Avx512ConvertImgU8ToF32Cpp(float* des, const uint8_t* src, uint32_t num_pixels);
extern bool Avx512ConvertImgF32ToU8Cpp(uint8_t* des, const float* src, uint32_t num_pixels);
extern uint32_t Avx512ConvertImgVerify(const float* src1, const float* src2, uint32_t
num_pixels);
extern uint32_t Avx512ConvertImgVerify(const uint8_t* src1, const uint8_t* src2, uint32_t
num_pixels);

// Ch14_02_.asm
extern "C" bool Avx512ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_pixels);
extern "C" bool Avx512ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_pixels);

void InitU8(uint8_t* x, uint32_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {0, 255};
    default_random_engine rng {seed};

    for (uint32_t i = 0; i < n; i++)
        x[i] = ui_dist(rng);
}
```

```
void InitF32(float* x, uint32_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {0, 1000};
    default_random_engine rng {seed};

    for (uint32_t i = 0; i < n; i++)
        x[i] = (float)ui_dist(rng) / 1000.0f;
}

void Avx512ConvertImgU8ToF32(void)
{
    const size_t align = 64;
    const uint32_t num_pixels = 1024;
    AlignedArray<uint8_t> src_aa(num_pixels, align);
    AlignedArray<float> des1_aa(num_pixels, align);
    AlignedArray<float> des2_aa(num_pixels, align);
    uint8_t* src = src_aa.Data();
    float* des1 = des1_aa.Data();
    float* des2 = des2_aa.Data();

    InitU8(src, num_pixels, 12);

    bool rc1 = Avx512ConvertImgU8ToF32Cpp(des1, src, num_pixels);
    bool rc2 = Avx512ConvertImgU8ToF32_(des2, src, num_pixels);

    cout << "\nResults for Avx512ConvertImgU8ToF32\n";

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    uint32_t num_diff = Avx512ConvertImgVerify(des1, des2, num_pixels);
    cout << "  Number of pixel compare errors (num_diff) = " << num_diff << '\n';
}

void Avx512ConvertImgF32ToU8(void)
{
    const size_t align = 64;
    const uint32_t num_pixels = 1024;
    AlignedArray<float> src_aa(num_pixels, align);
    AlignedArray<uint8_t> des1_aa(num_pixels, align);
    AlignedArray<uint8_t> des2_aa(num_pixels, align);
    float* src = src_aa.Data();
    uint8_t* des1 = des1_aa.Data();
    uint8_t* des2 = des2_aa.Data();
```

```
    InitF32(src, num_pixels, 20);

    // Test values to demonstrate clipping in conversion functions
    src[0] = 0.5f;          src[8]  = 3.33f;
    src[1] = -1.0f;         src[9]  = 0.67f;
    src[2] = 0.38f;         src[10] = 0.75f;
    src[3] = 0.62f;         src[11] = 0.95f;
    src[4] = 2.1f;          src[12] = -0.33f;
    src[5] = 0.25f;         src[13] = 0.8f;
    src[6] = -1.25f;        src[14] = 0.12f;
    src[7] = 0.45f;         src[15] = 4.0f;

    bool rc1 = Avx512ConvertImgF32ToU8Cpp(des1, src, num_pixels);
    bool rc2 = Avx512ConvertImgF32ToU8_(des2, src, num_pixels);

    cout << "\nResults for Avx512ConvertImgF32ToU8\n";

    if (!rc1 || !rc2)
    {
        cout << "Invalid return code - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    uint32_t num_diff = Avx512ConvertImgVerify(des1, des2, num_pixels);
    cout << "  Number of pixel compare errors (num_diff) = " << num_diff << '\n';
}

int main()
{
    Avx512ConvertImgU8ToF32();
    Avx512ConvertImgF32ToU8();
    return 0;
}

;-------------------------------------------------
;               Ch14_02.asm
;-------------------------------------------------

        include <cmpequ.asmh>
        extern c_NumPixelsMax:dword

            .const
r4_1p0      real4 1.0
r4_255p0    real4 255.0
```

```
; extern "C" bool Avx512ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t
num_pixels)

        .code
Avx512ConvertImgU8ToF32_ proc

; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                         ;set error return code
        or r8d,r8d
        jz Done                             ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                             ;jump if num_pixels too big
        test r8d,3fh
        jnz Done                            ;jump if num_pixels % 64 != 0
        test rcx,3fh
        jnz Done                            ;jump if des not aligned
        test rdx,3fh
        jnz Done                            ;jump if src not aligned

; Perform required initializations
        shr r8d,6                               ;number of blocks (64 pixels/block)
        vmovss xmm0,real4 ptr [r4_1p0]
        vdivss xmm1,xmm0,real4 ptr [r4_255p0]
        vbroadcastss zmm5,xmm1                  ;packed scale factor (1.0 / 255.0)

        align 16
@@:     vpmovzxbd zmm0,xmmword ptr [rdx]
        vpmovzxbd zmm1,xmmword ptr [rdx+16]
        vpmovzxbd zmm2,xmmword ptr [rdx+32]
        vpmovzxbd zmm3,xmmword ptr [rdx+48] ;zmm3:zmm0 = 64 U32 pixels

; Convert pixels from uint8_t to float [0.0, 255.0]
        vcvtudq2ps zmm16,zmm0
        vcvtudq2ps zmm17,zmm1
        vcvtudq2ps zmm18,zmm2
        vcvtudq2ps zmm19,zmm3               ;zmm19:zmm16 = 64 F32 pixels

; Normalize pixels to [0.0, 1.0]
        vmulps zmm20,zmm16,zmm5
        vmulps zmm21,zmm17,zmm5
        vmulps zmm22,zmm18,zmm5
        vmulps zmm23,zmm19,zmm5             ;zmm23:zmm20 = 64 F32 pixels (normalized)

; Save F32 pixels to des
        vmovaps zmmword ptr [rcx],zmm20
        vmovaps zmmword ptr [rcx+64],zmm21
        vmovaps zmmword ptr [rcx+128],zmm22
        vmovaps zmmword ptr [rcx+192],zmm23
```

```
; Update pointers and counters
        add rdx,64
        add rcx,256
        sub r8d,1
        jnz @B

        mov eax,1                       ;set success return code

Done:   vzeroupper
        ret
Avx512ConvertImgU8ToF32_ endp

; extern "C" bool Avx512ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t
num_pixels)

Avx512ConvertImgF32ToU8_ proc
; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                         ;set error return code
        or r8d,r8d
        jz Done                             ;jump if num_pixels is zero
        cmp r8d,[c_NumPixelsMax]
        ja Done                             ;jump if num_pixels too big
        test r8d,3fh
        jnz Done                            ;jump if num_pixels % 64 != 0
        test rcx,3fh
        jnz Done                            ;jump if des not aligned
        test rdx,3fh
        jnz Done                            ;jump if src not aligned

; Perform required initializations
        shr r8d,4                           ;number of pixel blocks (16 pixels / block)
        vxorps zmm29,zmm29,zmm29            ;packed 0.0
        vbroadcastss zmm30,[r4_1p0]         ;packed 1.0
        vbroadcastss zmm31,[r4_255p0]       ;packed 255.0

        align 16
@@:     vmovaps zmm0,zmmword ptr [rdx]      ;zmm0 = block of 16 pixels

; Clip pixels in current block to [0,0. 1.0]
        vcmpps k1,zmm0,zmm29,CMP_GE         ;k1 = mask of pixels >= 0.0
        vmovaps zmm1{k1}{z},zmm0            ;all pixels >= 0.0

        vcmpps k2,zmm1,zmm30,CMP_GT         ;k2 = mask of pixels > 1.0
        vmovaps zmm1{k2},zmm30              ;all pixels clipped to [0.0, 1.0]

; Convert pixels to uint8_t and save to des
        vmulps zmm2,zmm1,zmm31              ;all pixels [0.0, 255.0]
        vcvtps2udq zmm3,zmm2{ru-sae}        ;all pixels [0, 255]
        vpmovusdb xmmword ptr [rcx],zmm3    ;save pixels as unsigned bytes
```

```
; Update pointers and counters
        add rdx,64
        add rcx,16
        sub r8d,1
        jnz @B

        mov eax,1                           ;set success return code

Done:   vzeroupper
        ret
Avx512ConvertImgF32ToU8_ endp
        end
```

The C++ code in Listing 14-2 begins with the requisite function declarations. The first declaration set is for the functions `Avx512ConvertImgU8ToF32Cpp` and `Avx512ConvertImgU8ToF32Cpp`, which are defined in the file `Ch14_02_Misc.cpp`. The source code for these functions is not shown since they're almost identical to the AVX2 counterpart functions that were used in source code example `Ch07_06`. Two minor changes were made: the source and destination pixel buffers are aligned on a 64-byte instead of a 16-byte boundary; the number of pixels in these buffers must be evenly divisible by 64 instead of 32.

The function `Avx512ConvertImgU8ToF32` initializes the test arrays for converting pixels values from `uint8_t` to `float`. This function uses the C++ template class `AlignedArray<>` to allocate these arrays on a 64-byte boundary. Following test array initialization, `Avx512ConvertImgU8ToF32` invokes the C++ and assembly language conversion functions. It then calls `Avx512ConvertImgVerify` to verify the results. The function `Avx512ConvertImgF32ToU8` converts pixel values from `float` to `uint8_t`. Note that this function intentionally initializes the first few values of the source pixel buffer `src` to known values in order to verify that the conversion functions properly clip out-of-range pixel values.

The assembly language function `Avx512ConvertImgU8ToF32_` begins its execution by validating `num_pixels`. It then confirms that the pixel buffers `src` and `des` are properly aligned on a 64-byte boundary. In source code example `Ch07_06` from Chapter 7, pixel normalization was performed by dividing each pixel value by 255.0. `Avx512ConvertImgU8ToF32_` carries out pixel normalization using the multiplicative scale factor 1.0/255.0 since floating-point multiplication is usually faster than floating-point division. The `vbroadcastss zmm5,xmm1` instruction loads a packed version of this scale factor into register ZMM5.

Each processing loop iteration starts with a `vpmovzxbd zmm0,xmmword ptr [rdx]` instruction. This instruction copies and zero-extends the 16-byte (or `uint8_t`) pixels pointed to by RDX to doublewords; it then saves these values in register ZMM0. Three more `vpmovzxbd` instructions are then employed to load another 48 pixels into registers ZMM1, ZMM2, and ZMM3. This is followed by four `vcvtudq2ps` instructions that convert each unsigned doubleword pixel value in registers ZMM0–ZMM3 to single-precision floating-point. The ensuing `vmulps` instructions multiply these values by the normalization scale factor; the results are then saved to the destination pixel buffer `des` using a series of `vmovaps` instructions.

In source code example `Ch07_06`, all floating-point pixel values were clipped to [0.0, 1.0] before being converted to `uint8_t` values. The function `Avx512ConvertImgF32ToU8_` also performs this same operation. Following its argument validation checks, `Avx512ConvertImgF32ToU8_` loads registers ZMM29, ZMM30, and ZMM31 with packed versions of the single-precision floating-point constants 0.0, 1.0, and 255.0, respectively. The processing loop of `Avx512ConvertImgF32ToU8_` begins each iteration with a `vmovaps zmm0,zmmword ptr [rdx]` that loads a block of 16 single-precision floating-point pixels into register ZMM0. The ensuing `vcmpps k1,zmm0,zmm29,CMP_GE` instruction compares each pixel element in ZMM0 to 0.0 and saves the resultant compare mask in opmask register K1. The next instruction, `vmovaps zmm1{k1}{z},zmm0`, uses zero masking to eliminate all pixel values less than 0.0. Figure 14-1 illustrates these operations.

```
Initial values
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | zmm29 |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | zmm30 |
| 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | 255.0 | zmm31 |

```
vmovaps zmm0,zmmword ptr [rdx]        ;zmm0 = block of 16 pixels
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4.00 | 0.12 | 0.80 | -0.33 | 0.95 | 0.75 | 0.67 | 3.33 | 0.45 | -1.25 | 0.25 | 2.10 | 0.62 | 0.38 | -1.00 | 0.50 | zmm0 |

```
vcmpps k1,zmm0,zmm29,CMP_GE           ;k1 = mask of pixels >= 0.0
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | k1[15:0] |

```
vmovaps zmm1{k1}{z},zmm0              ;all pixels >= 0.0
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4.00 | 0.12 | 0.80 | 0.0 | 0.95 | 0.75 | 0.67 | 3.33 | 0.45 | 0.0 | 0.25 | 2.10 | 0.62 | 0.38 | 0.0 | 0.50 | zmm1 |

```
vcmpps k2,zmm1,zmm30,CMP_GT           ;k2 = mask of pixels > 1.0
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | k2[15:0] |

```
vmovaps zmm1{k2},zmm30                ;all pixels clipped to [0.0, 1.0]
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 0.12 | 0.80 | 0.0 | 0.95 | 0.75 | 0.67 | 1.0 | 0.45 | 0.0 | 0.25 | 1.0 | 0.62 | 0.38 | 0.0 | 0.50 | zmm1 |

```
vmulps zmm2,zmm1,zmm31                ;all pixels [0.0, 255.0]
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 255.0 | 30.6 | 204.0 | 0.0 | 242.25 | 191.25 | 170.85 | 255.0 | 114.75 | 0.0 | 63.75 | 255.0 | 158.1 | 96.9 | 0.0 | 127.5 | zmm2 |

```
vcvtps2udq zmm3,zmm2{ru-sae}          ;all pixels [0, 255]
```

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 255 | 31 | 204 | 0 | 243 | 192 | 171 | 255 | 115 | 0 | 64 | 255 | 159 | 97 | 0 | 128 | zmm3 |

***Figure 14-1.*** *Instruction sequence used to convert packed pixel values from floating-point to unsigned doubleword integers*

The subsequent `vcmpps k2,zmm1,zmm30,CMP_GT` instruction creates a mask of pixel values greater than 1.0 and saves this mask in opmask register K2. Following execution of the `vmovaps zmm1{k2},zmm30` instruction, all of the pixel values in register ZMM1 are greater than or equal to 0.0 and less than or equal to 1.0. The next two instructions, `vmulps zmm2,zmm1,zmm31` and `vcvtps2udq zmm3,zmm2{ru-sae}`, convert the normalized floating-point pixel values to unsigned doubleword integers. Note that the `vcvtps2udq` instruction employs an instruction-level rounding control operand (round up) primarily for demonstration purposes. The ensuing `vpmovusdb xmmword ptr [rcx],zmm3` instruction size-reduces the doubleword

values to bytes using unsigned saturation and saves them in the destination buffer pointed to by RCX. Here are the results for source code example Ch14_02:

```
Results for Avx512ConvertImgU8ToF32
  Number of pixel compare errors (num_diff) = 0

Results for Avx512ConvertImgF32ToU8
  Number of pixel compare errors (num_diff) = 0
```

## Image Thresholding

In source code example Ch07_08, you learned about image thresholding and how to create a binary (or two color) mask image. Briefly, thresholding is an image-processing technique that sets a mask image pixel to 0xff to signify that the intensity value of the corresponding pixel in a grayscale image is greater than a pre-determined threshold intensity value; otherwise, the mask image pixel is set to 0x00. The next source code example, Ch14_03, expands the image-thresholding technique that was used in Ch07_08 to support multiple compare operators. Listing 14-3 shows the source code for example Ch14_03.

***Listing 14-3.*** Example Ch14_03

```
//------------------------------------------------
//               Ch14_03.h
//------------------------------------------------

#pragma once
#include <cstdint>

// Compare operators
enum CmpOp { EQ, NE, LT, LE, GT, GE };

// Ch14_03_Misc.cpp
extern void Init(uint8_t* x, size_t n, unsigned int seed);
extern void ShowResults(const uint8_t* des1, const uint8_t* des2, size_t num_pixels, CmpOp
cmp_op,
    uint8_t cmp_val, size_t test_id);

// Ch14_03_.asm
extern "C" bool Avx512ComparePixels_(uint8_t* des, const uint8_t* src, size_t num_pixels,
    CmpOp cmp_op, uint8_t cmp_val);

//------------------------------------------------
//               Ch14_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cassert>
#include "Ch14_03.h"
#include "AlignedMem.h"
```

```
using namespace std;

extern "C" const size_t c_NumPixelsMax = 16777216;

bool Avx512ComparePixelsCpp(uint8_t* des, const uint8_t* src, size_t num_pixels, CmpOp
cmp_op, uint8_t cmp_val)
{
    // Make sure num_pixels is valid
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if ((num_pixels & 0x3f) != 0)
        return false;

    // Make sure src and des are aligned on a 64-byte boundary
    if (!AlignedMem::IsAligned(src, 64))
        return false;
    if (!AlignedMem::IsAligned(des, 64))
        return false;

    bool rc = true;
    const uint8_t cmp_false = 0x00;
    const uint8_t cmp_true = 0xff;

    switch (cmp_op)
    {
        case CmpOp::EQ:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] == cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::NE:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] != cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::LT:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] < cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::LE:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] <= cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::GT:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] > cmp_val) ? cmp_true : cmp_false;
            break;
```

```
        case CmpOp::GE:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] >= cmp_val) ? cmp_true : cmp_false;
            break;

        default:
            cout << "Invalid CmpOp: " << cmp_op << '\n';
            rc = false;
    }

    return rc;
}

int main()
{
    const size_t align = 64;
    const size_t num_pixels = 4 * 1024 * 1024;
    AlignedArray<uint8_t> src_aa(num_pixels, align);
    AlignedArray<uint8_t> des1_aa(num_pixels, align);
    AlignedArray<uint8_t> des2_aa(num_pixels, align);
    uint8_t* src = src_aa.Data();
    uint8_t* des1 = des1_aa.Data();
    uint8_t* des2 = des2_aa.Data();

    const uint8_t cmp_vals[] {197, 222, 43, 43, 129, 222};
    const CmpOp cmp_ops[] {CmpOp::EQ, CmpOp::NE, CmpOp::LT, CmpOp::LE, CmpOp::GT, CmpOp::GE};
    const size_t num_cmp_vals = sizeof(cmp_vals) / sizeof(uint8_t);
    const size_t num_cmp_ops = sizeof(cmp_ops) / sizeof(CmpOp);

    assert(num_cmp_vals == num_cmp_ops);

    Init(src, num_pixels, 511);

    cout << "Results for Ch14_03\n";

    for (size_t i = 0; i < num_cmp_ops; i++)
    {
        Avx512ComparePixelsCpp(des1, src, num_pixels, cmp_ops[i], cmp_vals[i]);
        Avx512ComparePixels_(des2, src, num_pixels, cmp_ops[i], cmp_vals[i]);
        ShowResults(des1, des2, num_pixels, cmp_ops[i], cmp_vals[i], i + 1);
    }

    return 0;
}

;-------------------------------------------------
;               Ch14_03.asm
;-------------------------------------------------

        include <cmpequ.asmh>
        extern c_NumPixelsMax:qword
```

```
; Macro CmpPixels

_CmpPixels macro CmpOp
        align 16
@@:     vmovdqa64 zmm0,zmmword ptr [rdx+rax]    ;load next block of 64 pixels
        vpcmpub k1,zmm0,zmm4,CmpOp              ;perform compare operation
        vmovdqu8 zmm1{k1}{z},zmm5               ;set mask pixels to 0 or 255 using opmask
        vmovdqa64 zmmword ptr [rcx+rax],zmm1    ;save mask pixels

        add rax,64                              ;update offset
        sub r8,64
        jnz @B                                  ;repeat until done
        mov eax,1                               ;set success return code
        vzeroupper
        ret
        endm

; extern "C" bool Avx512ComparePixels_(uint8_t* des, const uint8_t* src,
;   size_t num_pixels, CmpOp cmp_op, uint8_t cmp_val);

        .code
Avx512ComparePixels_ proc

; Make sure num_pixels is valid and pixel buffers are properly aligned
        xor eax,eax                         ;set error code (also array offset)

        or r8,r8
        jz Done                             ;jump if num_pixels is zero
        cmp r8,[c_NumPixelsMax]
        ja Done                             ;jump if num_pixels too big
        test r8,3fh
        jnz Done                            ;jump if num_pixels % 64 != 0

        test rcx,3fh
        jnz Done                            ;jump if des not aligned
        test rdx,3fh
        jnz Done                            ;jump if src not aligned

; Perform required initializations
        vpbroadcastb zmm4,byte ptr [rsp+40] ;zmm4 = packed cmp_val
        mov r10d,255
        vpbroadcastb zmm5,r10d              ;zmm5 = packed 255

; Perform specified compare operation
        cmp r9d,0
        jne LB_NE
        _CmpPixels CMP_EQ                   ;CmpOp::EQ

LB_NE:  cmp r9d,1
        jne LB_LT
        _CmpPixels CMP_NEQ                  ;CmpOp::NE
```

```
LB_LT:  cmp r9d,2
        jne LB_LE
        _CmpPixels CMP_LT                   ;CmpOp::LT

LB_LE:  cmp r9d,3
        jne LB_GT
        _CmpPixels CMP_LE                   ;CmpOp::LE

LB_GT:  cmp r9d,4
        jne LB_GE
        _CmpPixels CMP_NLE                  ;CmpOp::GT

LB_GE:  cmp r9d,5
        jne Done
        _CmpPixels CMP_NLT                  ;CmpOp::GE

Done:   vzeroupper
        ret
Avx512ComparePixels_ endp
        end
```

Near the top of the header file `Ch14_03.h` is an `enum` named `CmpOp`, which contains identifiers for the common compare operations. This is followed by the example's function declarations. The C++ functions `Init` and `ShowResults` are ancillary functions that perform test array initialization and display results. The source code for these functions is not shown in Listing 14-3 but included with the chapter download package. The function `Avx512ComparePixels_` is an AVX-512 assembly language functions that implements the pixel thresholding algorithm.

The function `Avx512ComparePixelsCpp` contains the C++ implementation of the updated thresholding algorithm. This function begins its execution by validating `num_pixels` for size and divisibility by 64. It then verifies that the pixel buffers `src` and `des` are properly aligned on a 64-byte boundary. Following argument validation code is a `switch` statement that applies the selector `cmp_op` to select a compare operation. Each `switch` statement `case` code block is a simple `for` loop that compares `src[i]` against `cmp_val` using the specified operator and sets pixels in the mask image to `0xff` (true compare) or `0x00` (false compare). The function `main` includes code that allocates the image pixel buffers, exercises the functions `Avx512ComparePixelsCpp` and `Avx512ComparePixels_` using various compare operators and displays results.

The assembly language code in Listing 14-3 commences with the macro `_CmpPixels`. This macro generates AVX-512 code that implements a processing loop for a pixel compare operator. The macro `_CmpPixels` requires the following register initializations prior to its use: RAX = 0, RCX = mask image pixel buffer, RDX = grayscale image pixel buffer, R8 = number of pixels, ZMM4 = packed byte threshold values, and ZMM5 = packed `0xff` byte values. Each processing loop iteration of `_CmpPixels` begins with a `vmovdqa64 zmm0,zmmword ptr [rdx+rax]` instruction that loads 64 unsigned 8-bit integers into register ZMM0. The next instruction, `vpcmpub k1,zmm0,zmm4,CmpOp`, compares the grayscale pixel intensity values in ZMM0 to the packed values in ZMM4; it then saves the resultant mask in opmask register K1. The ensuing `vmovdqu8 zmm1{k1}{z},zmm5` instruction sets each mask pixel value in ZMM1 to `0xff` (true compare) or `0x00` (false compare) according to the value of the corresponding bit position in K1. The instruction `vmovdqa64 zmmword ptr [rcx+rax],zmm1` then saves the 64 mask pixels to the mask image pixel buffer.

The function `Avx512ComparePixels_` employs the macro `_CmpPixels` to implement the same algorithm as its C++ counterpart `Avx512ComparePixelsCpp`. Follow the requisite argument validation checks, a `vpbroadcastb zmm4,byte ptr [rsp+40]` instruction broadcasts `cmp_val` to each byte element in register ZMM4. The next two instructions, `mov r10d,255` and `vpbroadcastb zmm5,r10d`, load the value `0xff` into each byte element of ZMM5. The remaining code in `Avx512ComparePixels_` uses the argument value

cmp_val to implement an ad hoc switch statement that takes advantage of the macro _CmpPixels. Note that this function uses the compare equates CMP_NLE (not less than or equal) or CMP_NLT (not less than) for the _CmpPixels macro argument CmpOp instead of CMP_GT or CMP_GE. The reason for this is that the vpcmpub instruction in _CmpPixels does not support use of the CMP_GT and CMP_GE equates (mathematically these latter equates are equivalent to CMP_NLE and CMP_NLT, but are assigned different values in cmpequ.asmh). Here are the results for source code example Ch14_03:

```
Results for Ch14_03

Test #1
  num_pixels: 4194304
  cmp_op:     EQ
  cmp_val:    197
  Pixel masks are identical
  Number of non-zero mask pixels = 16424

Test #2
  num_pixels: 4194304
  cmp_op:     NE
  cmp_val:    222
  Pixel masks are identical
  Number of non-zero mask pixels = 4177927

Test #3
  num_pixels: 4194304
  cmp_op:     LT
  cmp_val:    43
  Pixel masks are identical
  Number of non-zero mask pixels = 703652

Test #4
  num_pixels: 4194304
  cmp_op:     LE
  cmp_val:    43
  Pixel masks are identical
  Number of non-zero mask pixels = 719787

Test #5
  num_pixels: 4194304
  cmp_op:     GT
  cmp_val:    129
  Pixel masks are identical
  Number of non-zero mask pixels = 2065724

Test #6
  num_pixels: 4194304
  cmp_op:     GE
  cmp_val:    222
  Pixel masks are identical
  Number of non-zero mask pixels = 556908
```

## Image Statistics

Listing 14-4 shows the source code for example Ch14_04. This example illustrates how to calculate the mean and standard deviation of a grayscale image using its pixel intensity values. In order to make source code example Ch14_04 a little more interesting, the C++ and assembly language functions use only the pixel values that reside between two threshold limits. Pixel values outside of these limits are excluded from any mean and standard deviation calculations.

***Listing 14-4.*** Example Ch14_04

```
//------------------------------------------------
//               Ch14_04.h
//------------------------------------------------

#pragma once
#include <cstdint>

// This structure must match the structure that's defined in Ch14_04_.asm.
struct ImageStats
{
    uint8_t* m_PixelBuffer;
    uint64_t m_NumPixels;
    uint32_t m_PixelValMin;
    uint32_t m_PixelValMax;
    uint64_t m_NumPixelsInRange;
    uint64_t m_PixelSum;
    uint64_t m_PixelSumOfSquares;
    double m_PixelMean;
    double m_PixelSd;
};

// Ch14_04.cpp
extern bool Avx512CalcImageStatsCpp(ImageStats& im_stats);

// Ch14_04_.asm
extern "C" bool Avx512CalcImageStats_(ImageStats& im_stats);

// Ch04_04_BM.cpp
extern void Avx512CalcImageStats_BM(void);

// Common constants
const uint32_t c_PixelValMin = 40;
const uint32_t c_PixelValMax = 230;

//------------------------------------------------
//               Ch14_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
```

```
#include <fstream>
#include <string>
#include <stdexcept>
#include "Ch14_04.h"
#include "AlignedMem.h"
#include "ImageMatrix.h"

using namespace std;

extern "C" uint64_t c_NumPixelsMax = 256 * 1024;

bool Avx512CalcImageStatsCpp(ImageStats& im_stats)
{
    uint64_t num_pixels = im_stats.m_NumPixels;
    const uint8_t* pb = im_stats.m_PixelBuffer;

    // Perform validation checks
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if (!AlignedMem::IsAligned(pb, 64))
        return false;

    // Calculate intermediate sums
    im_stats.m_PixelSum = 0;
    im_stats.m_PixelSumOfSquares = 0;
    im_stats.m_NumPixelsInRange = 0;

    for (size_t i = 0; i < num_pixels; i++)
    {
        uint32_t pval = pb[i];

        if (pval >= im_stats.m_PixelValMin && pval <= im_stats.m_PixelValMax)
        {
            im_stats.m_PixelSum += pval;
            im_stats.m_PixelSumOfSquares += pval * pval;
            im_stats.m_NumPixelsInRange++;
        }
    }

    // Calculate mean and standard deviation
    double temp0 = (double)im_stats.m_NumPixelsInRange * im_stats.m_PixelSumOfSquares;
    double temp1 = (double)im_stats.m_PixelSum * im_stats.m_PixelSum;
    double var_num = temp0 - temp1;
    double var_den = (double)im_stats.m_NumPixelsInRange * (im_stats.m_NumPixelsInRange - 1);
    double var = var_num / var_den;

    im_stats.m_PixelMean = (double)im_stats.m_PixelSum / im_stats.m_NumPixelsInRange;
    im_stats.m_PixelSd = sqrt(var);

    return true;
}
```

```
void Avx512CalcImageStats()
{
    const wchar_t* image_fn = L"..\\Ch14_Data\\TestImage4.bmp";

    ImageStats is1, is2;
    ImageMatrix im(image_fn);
    uint64_t num_pixels = im.GetNumPixels();
    uint8_t* pb = im.GetPixelBuffer<uint8_t>();

    is1.m_PixelBuffer = pb;
    is1.m_NumPixels = num_pixels;
    is1.m_PixelValMin = c_PixelValMin;
    is1.m_PixelValMax = c_PixelValMax;

    is2.m_PixelBuffer = pb;
    is2.m_NumPixels = num_pixels;
    is2.m_PixelValMin = c_PixelValMin;
    is2.m_PixelValMax = c_PixelValMax;

    const char nl = '\n';
    const char* s = " | ";
    const unsigned int w1 = 22;
    const unsigned int w2 = 12;

    cout << fixed << setprecision(6) << left;
    wcout << fixed << setprecision(6) << left;

    cout << "\nResults for Avx512CalcImageStats\n";
    wcout << setw(w1) << "image_fn:" << setw(w2) << image_fn << nl;
    cout << setw(w1) << "num_pixels:" << setw(w2) << num_pixels << nl;
    cout << setw(w1) << "c_PixelValMin:" << setw(w2) << c_PixelValMin << nl;
    cout << setw(w1) << "c_PixelValMax:" <<  setw(w2) << c_PixelValMax << nl;

    bool rc1 = Avx512CalcImageStatsCpp(is1);
    bool rc2 = Avx512CalcImageStats_(is2);

    if (!rc1 || !rc2)
    {
        cout << "Bad return code\n";
        cout << "  rc1 = " << rc1 << '\n';
        cout << "  rc2 = " << rc2 << '\n';
        return;
    }

    cout << nl;

    cout << setw(w1) << "m_NumPixelsInRange: ";
    cout << setw(w2) << is1.m_NumPixelsInRange << s;
    cout << setw(w2) << is2.m_NumPixelsInRange << nl;

    cout << setw(w1) << "m_PixelSum:";
    cout << setw(w2) << is1.m_PixelSum << s;
    cout << setw(w2) << is2.m_PixelSum << nl;
```

```
    cout << setw(w1) << "m_PixelSumOfSquares:";
    cout << setw(w2) << is1.m_PixelSumOfSquares << s;
    cout << setw(w2) << is2.m_PixelSumOfSquares << nl;

    cout << setw(w1) << "m_PixelMean:";
    cout << setw(w2) << is1.m_PixelMean << s;
    cout << setw(w2) << is2.m_PixelMean << nl;

    cout << setw(w1) << "m_PixelSd:";
    cout << setw(w2) << is1.m_PixelSd << s;
    cout << setw(w2) << is2.m_PixelSd << nl;
}

int main()
{
    try
    {
        Avx512CalcImageStats();
        Avx512CalcImageStats_BM();
    }

    catch (runtime_error& rte)
    {
        cout << "'runtime_error' exception has occurred - " << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Unexpected exception has occurred\n";
        cout << "File = " << __FILE__ << '\n';
    }

    return 0;
}

;-------------------------------------------------
;               Ch14_04.asm
;-------------------------------------------------

        include <cmpequ.asmh>
        include <MacrosX86-64-AVX.asmh>
        extern c_NumPixelsMax:qword

; This structure must match the structure that's defined in Ch14_04.h
ImageStats          struct
PixelBuffer         qword ?
NumPixels           qword ?
PixelValMin         dword ?
PixelValMax         dword ?
NumPixelsInRange    qword ?
PixelSum            qword ?
PixelSumOfSquares   qword ?
```

```
PixelMean           real8 ?
PixelSd             real8 ?
ImageStats          ends

_UpdateSums macro Disp
        vpmovzxbd zmm0,xmmword ptr [rcx+Disp]   ;zmm0 = 16 pixels
        vpcmpud k1,zmm0,zmm31,CMP_GE            ;k1 = mask of pixels >= pixel_val_min
        vpcmpud k2,zmm0,zmm30,CMP_LE            ;k2 = mask of pixels <= pixel_val_max
        kandw k3,k2,k1                          ;k3 = mask of in-range pixels
        vmovdqa32 zmm1{k3}{z},zmm0              ;zmm1 = in-range pixels
        vpaddd zmm16,zmm16,zmm1                 ;update packed pixel_sum
        vpmulld zmm2,zmm1,zmm1
        vpaddd zmm17,zmm17,zmm2                 ;update packed pixel_sum_of_squares
        kmovw rax,k3
        popcnt rax,rax                          ;count number of in-range pixels
        add r10,rax                             ;update num_pixels_in_range
        endm

; extern "C" bool Avx512CalcImageStats_(ImageStats& im_stats);

        .code
Avx512CalcImageStats_ proc frame
        _CreateFrame CIS_,0,0,rsi,r12,r13
        _EndProlog

; Make sure num_pixels is valid and pixel_buff is properly aligned
        xor eax,eax                         ;set error return code

        mov rsi,rcx                                    ;rsi = im_stats ptr
        mov rcx,qword ptr [rsi+ImageStats.PixelBuffer] ;rcx = pixel buffer ptr
        mov rdx,qword ptr [rsi+ImageStats.NumPixels]   ;rdx = num_pixels

        test rdx,rdx
        jz Done                             ;jump if num_pixels is zero
        cmp rdx,[c_NumPixelsMax]
        ja Done                             ;jump if num_pixels too big

        test rcx,3fh
        jnz Done                            ;jump if pixel_buff misaligned

; Perform required initializations
        mov r8d,dword ptr [rsi+ImageStats.PixelValMin]
        mov r9d,dword ptr [rsi+ImageStats.PixelValMax]

        vpbroadcastd zmm31,r8d              ;packed pixel_val_min
        vpbroadcastd zmm30,r9d              ;packed pixel_val_max

        vpxorq zmm29,zmm29,zmm29            ;packed pixel_sum
        vpxorq zmm28,zmm28,zmm28            ;packed pixel_sum_of_squares
        xor r10d,r10d                       ;num_pixels_in_range = 0
```

```
; Compute packed versions of pixel_sum and pixel_sum_of_squares
        cmp rdx,64
        jb LB1                          ;jump if there are fewer than 64 pixels

        align 16
@@:     vpxord zmm16,zmm16,zmm16        ;loop packed pixel_sum = 0
        vpxord zmm17,zmm17,zmm17        ;loop packed pixel_sum_of_squares = 0

        _UpdateSums 0                   ;process pixel_buff[i+15]:pixel_buff[i]
        _UpdateSums 16                  ;process pixel_buff[i+31]:pixel_buff[i+16]
        _UpdateSums 32                  ;process pixel_buff[i+47]:pixel_buff[i+32]
        _UpdateSums 48                  ;process pixel_buff[i+63]:pixel_buff[i+48]

        vextracti32x8 ymm0,zmm16,1      ;extract top 8 pixel_sum (dwords)
        vpaddd ymm1,ymm0,ymm16
        vpmovzxdq zmm2,ymm1
        vpaddq zmm29,zmm29,zmm2         ;update packed pixel_sum (qwords)

        vextracti32x8 ymm0,zmm17,1      ;extract top 8 pixel_sum_of_squares (dwords)
        vpaddd ymm1,ymm0,ymm17
        vpmovzxdq zmm2,ymm1
        vpaddq zmm28,zmm28,zmm2         ;update packed pixel_sum_of_squares (qwords)

        add rcx,64                      ;update pb ptr
        sub rdx,64                      ;update num_pixels
        cmp rdx,64
        jae @B                          ;repeat until done

        align 16
LB1:    test rdx,rdx
        jz LB3                          ;jump if no more pixels remain

        xor r13,r13                     ;pixel_sum = 0
        xor r12,r12                     ;pixel_sum_of_squares = 0
        mov r11,rdx                     ;number of remaining pixels

@@:     movzx rax,byte ptr [rcx]        ;load next pixel
        cmp rax,r8
        jb LB2                          ;jump if current pixel < pval_min
        cmp rax,r9
        ja LB2                          ;jump if current pixel > pval_max

        add r13,rax                     ;add to pixel_sum
        mul rax
        add r12,rax                     ;add to pixel_sum_of_squares
        add r10,1                       ;update num_pixels_in_range

LB2:    add rcx,1
        sub r11,1
        jnz @B                          ;repeat until done

; Save num_pixel_in_range
LB3:    mov qword ptr [rsi+ImageStats.NumPixelsInRange],r10
```

```
; Reduce packed pixel_sum to single qword
        vextracti64x4 ymm0,zmm29,1
        vpaddq ymm1,ymm0,ymm29
        vextracti64x2 xmm2,ymm1,1
        vpaddq xmm3,xmm2,xmm1
        vpextrq rax,xmm3,0
        vpextrq r11,xmm3,1
        add rax,r11                         ;rax = sum of qwords in zmm29
        add r13,rax                         ;add scalar pixel_sum

        mov qword ptr [rsi+ImageStats.PixelSum],r13

;Reduce packed pixel_sum_of_squares to single qword
        vextracti64x4 ymm0,zmm28,1
        vpaddq ymm1,ymm0,ymm28
        vextracti64x2 xmm2,ymm1,1
        vpaddq xmm3,xmm2,xmm1
        vpextrq rax,xmm3,0
        vpextrq r11,xmm3,1
        add rax,r11                         ;rax = sum of qwords in zmm28
        add r12,rax                         ;add scalar pixel_sum_of_squares

        mov qword ptr [rsi+ImageStats.PixelSumOfSquares],r12

; Calculate final mean and sd
        vcvtusi2sd xmm0,xmm0,r10            ;num_pixels_in_range (DPFP)
        sub r10,1
        vcvtusi2sd xmm1,xmm1,r10            ;num_pixels_in_range - 1 (DPFP)
        vcvtusi2sd xmm2,xmm2,r13            ;pixel_sum (DPFP)
        vcvtusi2sd xmm3,xmm3,r12            ;pixel_sum_of_squares (DPFP)
        vdivsd xmm4,xmm2,xmm0               ;final pixel_mean

        vmovsd real8 ptr [rsi+ImageStats.PixelMean],xmm4

        vmulsd xmm4,xmm0,xmm3               ;num_pixels_in_range * pixel_sum_of_squares
        vmulsd xmm5,xmm2,xmm2               ;pixel_sum * pixel_sum
        vsubsd xmm2,xmm4,xmm5               ;var_num
        vmulsd xmm3,xmm0,xmm1               ;var_den
        vdivsd xmm4,xmm2,xmm3               ;calc variance
        vsqrtsd xmm0,xmm0,xmm4              ;final pixel_sd

        vmovsd real8 ptr [rsi+ImageStats.PixelSd],xmm0

        mov eax,1                           ;set success return code

Done:   vzeroupper
        _DeleteFrame rsi,r12,r13
        ret
Avx512CalcImageStats_ endp
        end
```

The mean and standard deviation of the pixels in a grayscale image can be calculated using the following equations:

$$\bar{x} = \frac{1}{n}\sum_i x_i$$

$$s = \sqrt{\frac{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2}{n(n-1)}}$$

In the mean and standard deviation equations, the symbol $x_i$ represents an image buffer pixel and $n$ denotes the number of pixels. If you study these equations carefully, you will notice that two intermediate sums must be calculated: the sum of all pixels and the sum of all pixel values squared. Once these quantities are known, the mean and standard deviation can be determined using simple arithmetic. The standard deviation equation that's detailed here is simple to calculate and suitable for this source code example. For other use cases, however, this same equation is often unsuitable for standard deviation calculations especially those that involve floating-point values. You may want to consult the statistical variance calculating references that are listed in Appendix A before using this equation in one of your own programs.

Listing 14-4 begins with the C++ header file `Ch14_04.h` that includes the declaration of a structure named `ImageStats`. This structure is used to pass image data to the C++ and assembly language calculating functions and return results. A semantically equivalent structure is also defined in the assembly language file `Ch14_04_.asm`. The file `Ch14_04.h` also includes the constant definitions `c_PixelValMin` and `c_PixelValueMax`, which define the range limits that a pixel value must fall between to be included in any statistical calculations.

The function `Avx512CalcImageStatsCpp` is the principal calculating function in the C++ code. This function requires a pointer to an `ImageStats` structure as its sole argument. Following argument validation, `Avx512CalcImageStatsCpp` initializes the `ImageStats` intermediate sums `m_PixelSum`, `m_PixelSumOfSquares`, and `m_NumPixelsInRange` to zero. A simple `for` loop follows, which calculates `m_PixelSum` and `m_PixelSumOfSquares`. During each loop iteration, pixel values are tested for in-range validity before being included in any calculations. Following computation of the intermediate sums, the function `Avx512CalcImageStatsCpp` calculates the final mean and standard deviation. Note that `m_NumPixelsInRange` is used to calculate these statistical quantities instead of `m_NumPixels`. The remaining code in `Ch14_04.cpp` performs test case initialization, invokes the calculating functions, and streams the results to `cout`.

Toward the top of the file `Ch14_04_.asm` is the assembly language version of the structure `ImageStats`. This is followed by the macro definition `_UpdateSums` whose inner workings will be described shortly. The function `Avx512CalcImageStats_` begins its execution by performing the same argument validation checks as its C++ counterpart. It then initializes packed versions of the intermediate values `PixelValMin` and `PixelValMax`. The ensuing `vpxorq` instructions initialize packed quadword versions of `PixelSum` and `PixelSumOfSquares` to zero. Note that the `vpxor[d|q]` (and other AVX-512 bitwise Boolean) instructions can optionally specify an opmask operand register to perform merge or zero masking of doubleword or quadword elements. The final initialization instruction, `xor r10d,r10d`, sets `NumPixelsInRange` to zero.

The processing loop in function `Avx512CalcImageStats_` processes 64 pixels each iteration. Prior to the start of the processing loop, register RDX is tested to verify that at least 64 pixels remain. Each processing loop iteration begins with two `vpxord` instructions that initialize packed doubleword versions of `pixel_sum` and `pixel_sum_of_squares` to zero. Following this are four instances of the macro `_UpdateSum`, which process in aggregate the next group of 64 pixels. The first instruction of this macro, `vpmovzxbd zmm0,xmmword ptr [rcx+Disp]`, loads 16 unsigned byte values from source pixel buffer and saves these values as unsigned doublewords in register ZMM0. The ensuing `vpcmpud k1,zmm0,zmm31,CMP_GE`, `vpcmpud k2,zmm0,zmm30,CMP_LE`, and `kandw k3,k2,k1` instructions load opmask register K3 with a mask value of pixels that are greater than or equal to `pixel_val_min` and less than or equal to `pixel_val_max`. The `vmovdqa32 zmm1{k3}{z},zmm0` that follows uses zero masking to effectively eliminate out-of-range pixel

values from further calculations. The subsequent vpaddd and vpmulld instructions then update the packed doubleword quantities pixel_sum and pixel_sum_of_squares. The total number of in-range pixels in R10 is then updated using the instructions kmovw rax,k3, popcnt rax,rax, and add r10,rax. Figure 14-2 illustrates these calculations in greater detail. Note that this figure shows only the low-order 256 bits of each ZMM registers and the low-order 8 bits of each opmask register.

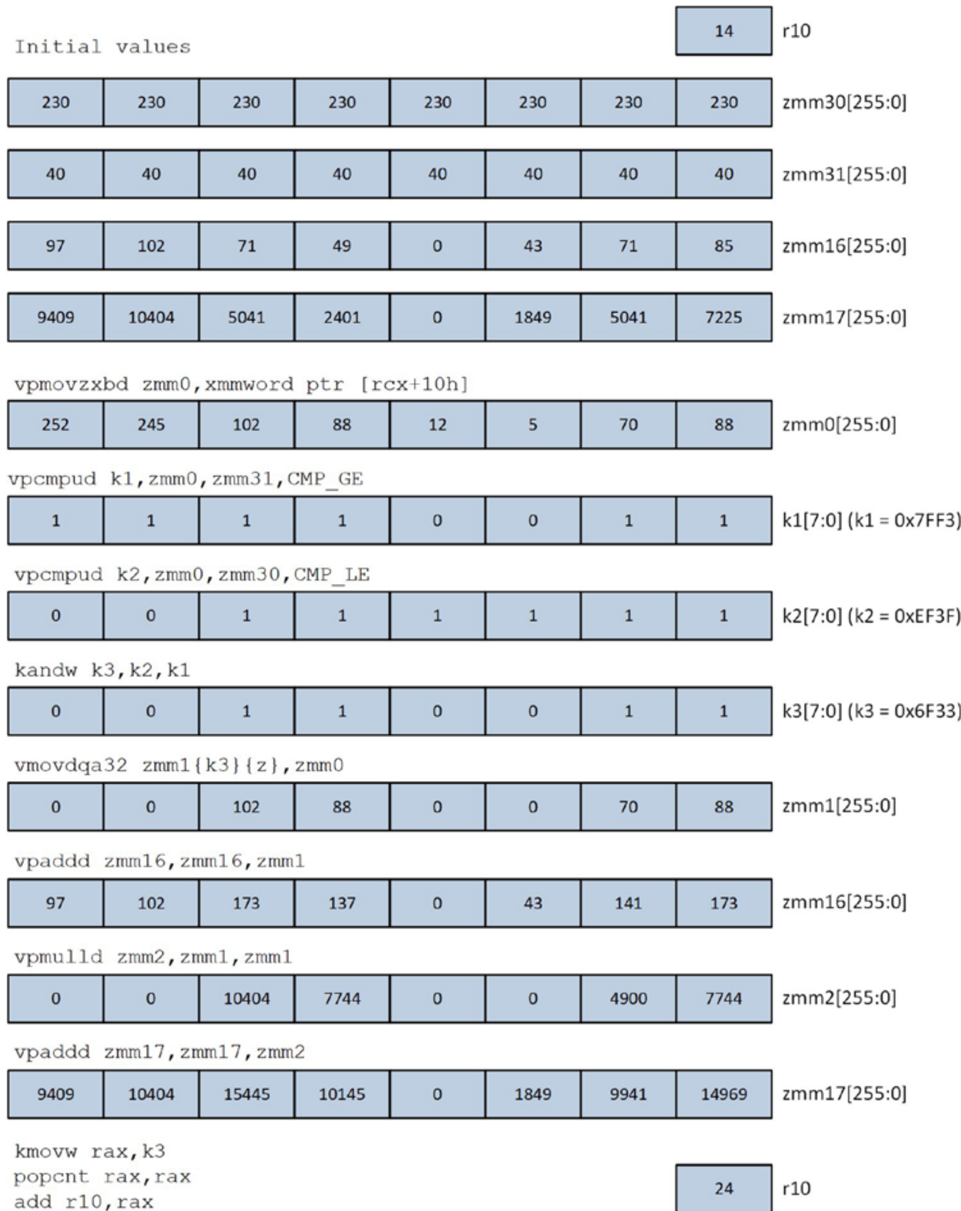



***Figure 14-2.*** *Calculations performed by instructions in macro _UpdateSums*

Following the four _UpdateSums usages, the doubleword elements of registers ZMM16 and ZMM17 contain packed copies of the values pixel_sum and pixel_sum_of_squares for the current block of 64 pixels. The vextracti32x8 ymm0,zmm16,1 and vpaddd ymm1,ymm0,ymm16 instructions reduce the number of doubleword values in register ZMM16 from 16 to 8. The ensuing vpmovzxdq zmm2,ymm1 instruction promotes these doubleword values to quadwords, and the vpaddq zmm29,zmm29,zmm2 instruction updates the global packed quadword pixel_sum values that are maintained in register ZMM29. A similar sequence of instructions is then used to update the global packed quadword pixel_sum_of_squares values in register ZMM28. Following these instructions, the processing loop updates its pointer register and counters; it then repeats until the number of remaining pixels falls below 64.

The block of code that starts at the label LB1 computes pixel_sum and pixel_sum_of_squares for the final few pixels (if any) using scalar integer arithmetic and the general-purpose registers. A series of extract (vextracti64x4, vextracti64x2, and vpextrq) and vpaddq instructions reduce the eight packed quadword pixel_sum values in ZMM29 to a single quadword value. A similar sequence of instructions is then used to calculate the final value for pixel_sum_of_squares. Note that these intermediate results are saved in the ImageStats structure pointed to by register RCX. The function Avx512CalcImageStats_ then executes a chain of vcvtusi2sd instructions to convert the intermediate results from unsigned quadword integers to double-precision floating-point. The final mean and standard deviation values are calculated using scalar double-precision floating-point arithmetic. The results for source code example Ch14_04 follow this paragraph. Table 14-1 shows benchmark timing measurements for the C++ and assembly language calculating functions Avx512CalcImageStatsCpp and Avx512CalcImageStats_.

```
Results for Avx512CalcImageStats

image_fn:             ..\Ch14_Data\TestImage4.bmp
num_pixels:           258130
c_PixelValMin:        40
c_PixelValMax:        230

m_NumPixelsInRange:   229897       | 229897
m_PixelSum:           32574462     | 32574462
m_PixelSumOfSquares:  5139441032   | 5139441032
m_PixelMean:          141.691549   | 141.691549
m_PixelSd:            47.738056    | 47.738056

Running benchmark function Avx512CalcImageStats_BM - please wait
Benchmark times save to file Ch14_04_Avx512CalcImageStats_BM_CHROMIUM.csv
```

***Table 14-1.*** *Benchmark Timing Measurements for Image Statistics Calculating Functions Using TestImage4.bmp*

| CPU | Avx512CalcImageStatsCpp | Avx512CalcImageStats_ |
|---|---|---|
| i7-4790S | ---- | ---- |
| i9-7900X | 404 | 29 |
| i7-8700K | ---- | ---- |

## RGB to Grayscale Conversion

In Chapter 10, you learned how to use the AVX2 instruction set to convert an RGB image to a grayscale image (see example Ch10_06). Listing 14-5 shows the source code for example Ch14_05, which illustrates RGB to grayscale image conversion using the AVX-512 instruction set.

***Listing 14-5.*** Example Ch14_05

```
//------------------------------------------------
//               Ch14_05.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <stdexcept>
#include <iomanip>
#include "Ch14_05.h"
#include "ImageMatrix.h"
#include "AlignedMem.h"

using namespace std;

extern "C" const int c_NumPixelsMin = 64;
extern "C" const int c_NumPixelsMax = 16 * 1024 * 1024;

// RGB to grayscale conversion coefficients
const float c_Coef[3] {0.2126f, 0.7152f, 0.0722f};

bool CompareGsImages(const uint8_t* pb_gs1,const uint8_t* pb_gs2, int num_pixels)
{
    for (int i = 0; i < num_pixels; i++)
    {
        if (abs((int)pb_gs1[i] - (int)pb_gs2[i]) > 1)
            return false;
    }

    return true;
}

bool Avx512RgbToGsCpp(uint8_t* pb_gs, const uint8_t* const* pb_rgb, int num_pixels, const
float coef[3])
{
    if (num_pixels < c_NumPixelsMin || num_pixels > c_NumPixelsMax)
        return false;
    if (num_pixels % 64 != 0)
        return false;
    if (!AlignedMem::IsAligned(pb_gs, 64))
        return false;

    const size_t align = 64;
    const uint8_t* pb_r = pb_rgb[0];
    const uint8_t* pb_g = pb_rgb[1];
    const uint8_t* pb_b = pb_rgb[2];
```

```
    if (!AlignedMem::IsAligned(pb_r, align))
        return false;
    if (!AlignedMem::IsAligned(pb_g, align))
        return false;
    if (!AlignedMem::IsAligned(pb_b, align))
        return false;

    for (int i = 0; i < num_pixels; i++)
    {
        uint8_t r = pb_r[i];
        uint8_t g = pb_g[i];
        uint8_t b = pb_b[i];

        float gs_temp = r * coef[0] + g * coef[1] + b * coef[2] + 0.5f;

        if (gs_temp < 0.0f)
            gs_temp = 0.0f;
        else if (gs_temp > 255.0f)
            gs_temp = 255.0f;

        pb_gs[i] = (uint8_t)gs_temp;
    }

    return true;
}

void Avx512RgbToGs(void)
{
    const wchar_t* fn_rgb = L"..\\Ch14_Data\\TestImage3.bmp";
    const wchar_t* fn_gs1 = L"Ch14_05_Avx512RgbToGs_TestImage3_GS1.bmp";
    const wchar_t* fn_gs2 = L"Ch14_05_Avx512RgbToGs_TestImage3_GS2.bmp";
    const wchar_t* fn_gs3 = L"Ch14_05_Avx512RgbToGs_TestImage3_GS3.bmp";

    ImageMatrix im_rgb(fn_rgb);
    int im_h = im_rgb.GetHeight();
    int im_w = im_rgb.GetWidth();
    int num_pixels = im_h * im_w;
    ImageMatrix im_r(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_g(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_b(im_h, im_w, PixelType::Gray8);
    RGB32* pb_rgb = im_rgb.GetPixelBuffer<RGB32>();
    uint8_t* pb_r = im_r.GetPixelBuffer<uint8_t>();
    uint8_t* pb_g = im_g.GetPixelBuffer<uint8_t>();
    uint8_t* pb_b = im_b.GetPixelBuffer<uint8_t>();
    uint8_t* pb_rgb_cp[3] {pb_r, pb_g, pb_b};

    for (int i = 0; i < num_pixels; i++)
    {
        pb_rgb_cp[0][i] = pb_rgb[i].m_R;
        pb_rgb_cp[1][i] = pb_rgb[i].m_G;
        pb_rgb_cp[2][i] = pb_rgb[i].m_B;
    }
```

```
    ImageMatrix im_gs1(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_gs2(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_gs3(im_h, im_w, PixelType::Gray8);
    uint8_t* pb_gs1 = im_gs1.GetPixelBuffer<uint8_t>();
    uint8_t* pb_gs2 = im_gs2.GetPixelBuffer<uint8_t>();
    uint8_t* pb_gs3 = im_gs3.GetPixelBuffer<uint8_t>();

    // Exercise conversion functions
    bool rc1 = Avx512RgbToGsCpp(pb_gs1, pb_rgb_cp, num_pixels, c_Coef);
    bool rc2 = Avx512RgbToGs_(pb_gs2, pb_rgb_cp, num_pixels, c_Coef);
    bool rc3 = Avx2RgbToGs_(pb_gs3, pb_rgb_cp, num_pixels, c_Coef);

    if (rc1 && rc2 && rc3)
    {
        im_gs1.SaveToBitmapFile(fn_gs1);
        im_gs2.SaveToBitmapFile(fn_gs2);
        im_gs2.SaveToBitmapFile(fn_gs3);

        bool c1 = CompareGsImages(pb_gs1, pb_gs2, num_pixels);
        bool c2 = CompareGsImages(pb_gs2, pb_gs3, num_pixels);

        if (c1 && c2)
            cout << "Grayscale image compare OK\n";
        else
            cout << "Grayscale image compare failed\n";
    }
    else
        cout << "Invalid return code\n";
}

int main()
{
    try
    {
        Avx512RgbToGs();
        Avx512RgbToGs_BM();
    }

    catch (runtime_error& rte)
    {
        cout << "'runtime_error' exception has occurred - " << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Unexpected exception has occurred\n";
    }

    return 0;
}
```

```
;-------------------------------------------------
;               Ch14_05.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern c_NumPixelsMin:dword
        extern c_NumPixelsMax:dword

            .const
r4_0p5      real4 0.5
r4_255p0    real4 255.0

; extern "C" bool Avx512RgbToGs_(uint8_t* pb_gs, const uint8_t* const* pb_rgb,
int num_pixels, const float coef[3]);

        .code
Avx512RgbToGs_ proc frame
        _CreateFrame RGBGS0_,0,96,r13,r14,r15
        _SaveXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

        xor eax,eax                         ;error return code (also pixel_buffer offset)
        cmp r8d,[c_NumPixelsMin]
        jl Done                             ;jump if num_pixels < min value
        cmp r8d,[c_NumPixelsMax]
        jg Done                             ;jump if num_pixels > max value
        test r8d,3fh
        jnz Done                            ;jump if (num_pixels % 64) != 0

        test rcx,3fh
        jnz Done                            ;jump if pb_gs is not aligned

        mov r13,[rdx]
        test r13,3fh
        jnz Done                            ;jump if pb_r is not aligned
        mov r14,[rdx+8]
        test r14,3fh
        jnz Done                            ;jump if pb_g is not aligned
        mov r15,[rdx+16]
        test r15,3fh
        jnz Done                            ;jump if pb_b is not aligned

; Perform required initializations
        vbroadcastss zmm10,real4 ptr [r9]       ;zmm10 = packed coef[0]
        vbroadcastss zmm11,real4 ptr [r9+4]     ;zmm11 = packed coef[1]
        vbroadcastss zmm12,real4 ptr [r9+8]     ;zmm12 = packed coef[2]
        vbroadcastss zmm13,real4 ptr [r4_0p5]   ;zmm13 = packed 0.5
        vbroadcastss zmm14,real4 ptr [r4_255p0] ;zmm14 = packed 255.0
        vxorps zmm15,zmm15,zmm15                ;zmm15 = packed 0.0
        mov r8d,r8d                             ;r8 = num_pixels
        mov r10,16                              ;r10 - number of pixels / iteration
```

```
; Load next block of pixels
        align 16
@@:     vpmovzxbd zmm0,xmmword ptr [r13+rax]    ;zmm0 = 16 pixels (r values)
        vpmovzxbd zmm1,xmmword ptr [r14+rax]    ;zmm1 = 16 pixels (g values)
        vpmovzxbd zmm2,xmmword ptr [r15+rax]    ;zmm2 = 16 pixels (b values)

; Convert dword values to SPFP and multiply by coefficients
        vcvtdq2ps zmm0,zmm0                 ;zmm0 = 16 pixels SPFP (r values)
        vcvtdq2ps zmm1,zmm1                 ;zmm1 = 16 pixels SPFP (g values)
        vcvtdq2ps zmm2,zmm2                 ;zmm2 = 16 pixels SPFP (b values)
        vmulps zmm0,zmm0,zmm10              ;zmm0 = r values * coef[0]
        vmulps zmm1,zmm1,zmm11              ;zmm1 = g values * coef[1]
        vmulps zmm2,zmm2,zmm12              ;zmm2 = b values * coef[2]

; Sum color components & clip values to [0.0, 255.0]
        vaddps zmm3,zmm0,zmm1               ;r + g
        vaddps zmm4,zmm3,zmm2               ;r + g + b
        vaddps zmm5,zmm4,zmm13              ;r + g + b + 0.5
        vminps zmm0,zmm5,zmm14              ;clip pixels above 255.0
        vmaxps zmm1,zmm0,zmm15              ;clip pixels below 0.0

; Convert grayscale values from SPFP to byte, save results
        vcvtps2dq zmm2,zmm1                 ;convert SPFP values to dwords

        vpmovusdb xmm3,zmm2                 ;convert to bytes
        vmovdqa xmmword ptr [rcx+rax],xmm3  ;save grayscale image pixels

        add rax,r10
        sub r8,r10
        jnz @B

        mov eax,1                           ;set success return code
Done:   vzeroupper
        _RestoreXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame r13,r14,r15
        ret
Avx512RgbToGs_ endp

; extern "C" bool Avx2RgbToGs_(uint8_t* pb_gs, const uint8_t* const* pb_rgb, int num_pixels,
const float coef[3]);

        .code
Avx2RgbToGs_ proc frame
        _CreateFrame RGBGS1_,0,96,r13,r14,r15
        _SaveXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

        xor eax,eax                         ;error return code (also pixel_buffer offset)
        cmp r8d,[c_NumPixelsMin]
        jl Done                             ;jump if num_pixels < min value
        cmp r8d,[c_NumPixelsMax]
```

```
        jg Done                             ;jump if num_pixels > max value
        test r8d,3fh
        jnz Done                            ;jump if (num_pixels % 64) != 0

        test rcx,3fh
        jnz Done                            ;jump if pb_gs is not aligned

        mov r13,[rdx]
        test r13,3fh
        jnz Done                            ;jump if pb_r is not aligned
        mov r14,[rdx+8]
        test r14,3fh
        jnz Done                            ;jump if pb_g is not aligned
        mov r15,[rdx+16]
        test r15,3fh
        jnz Done                            ;jump if pb_b is not aligned

; Perform required initializations
        vbroadcastss ymm10,real4 ptr [r9]       ;ymm10 = packed coef[0]
        vbroadcastss ymm11,real4 ptr [r9+4]     ;ymm11 = packed coef[1]
        vbroadcastss ymm12,real4 ptr [r9+8]     ;ymm12 = packed coef[2]
        vbroadcastss ymm13,real4 ptr [r4_0p5]   ;ymm13 = packed 0.5
        vbroadcastss ymm14,real4 ptr [r4_255p0] ;ymm14 = packed 255.0
        vxorps ymm15,ymm15,ymm15                ;ymm15 = packed 0.0
        mov r8d,r8d                             ;r8 = num_pixels
        mov r10,8                               ;r10 - number of pixels / iteration

; Load next block of pixels
        align 16
@@:     vpmovzxbd ymm0,qword ptr [r13+rax]      ;ymm0 = 8 pixels (r values)
        vpmovzxbd ymm1,qword ptr [r14+rax]      ;ymm1 = 8 pixels (g values)
        vpmovzxbd ymm2,qword ptr [r15+rax]      ;ymm2 = 8 pixels (b values)

; Convert dword values to SPFP and multiply by coefficients
        vcvtdq2ps ymm0,ymm0                 ;ymm0 = 8 pixels SPFP (r values)
        vcvtdq2ps ymm1,ymm1                 ;ymm1 = 8 pixels SPFP (g values)
        vcvtdq2ps ymm2,ymm2                 ;ymm2 = 8 pixels SPFP (b values)
        vmulps ymm0,ymm0,ymm10              ;ymm0 = r values * coef[0]
        vmulps ymm1,ymm1,ymm11              ;ymm1 = g values * coef[1]
        vmulps ymm2,ymm2,ymm12              ;ymm2 = b values * coef[2]

; Sum color components & clip values to [0.0, 255.0]
        vaddps ymm3,ymm0,ymm1               ;r + g
        vaddps ymm4,ymm3,ymm2               ;r + g + b
        vaddps ymm5,ymm4,ymm13              ;r + g + b + 0.5
        vminps ymm0,ymm5,ymm14              ;clip pixels above 255.0
        vmaxps ymm1,ymm0,ymm15              ;clip pixels below 0.0
```

```
; Convert grayscale components from SPFP to byte, save results
        vcvtps2dq ymm2,ymm1                 ;convert SPFP values to dwords

        vpackusdw ymm3,ymm2,ymm2
        vextracti128 xmm4,ymm3,1
        vpackuswb xmm5,xmm3,xmm4            ;byte GS pixels in xmm5[31:0] and xmm5[95:64]
        vpextrd r11d,xmm5,0                 ;r11d = 4 grayscale pixels
        mov dword ptr [rcx+rax],r11d        ;save grayscale image pixels
        vpextrd r11d,xmm5,2                 ;r11d = 4 grayscale pixels
        mov dword ptr [rcx+rax+4],r11d      ;save grayscale image pixels

        add rax,r10
        sub r8,r10
        jnz @B

        mov eax,1                           ;set success return code
Done:   vzeroupper
        _RestoreXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame r13,r14,r15
        ret
Avx2RgbToGs_ endp
        end
```

The algorithm that's used in this example to perform RGB to image grayscale conversion is the same one that was used in Ch10_06. As explained in Chapter 10, the algorithm uses a simple weighted average to transform an RGB image pixel into a grayscale image pixel. The C++ function Avx512RgbToGs begins its execution by loading the test image file. It then copies the RGB pixels of im_rgb into three separate color component image buffers. The reason for doing this is that this example's RGB to grayscale conversion functions require a structure of arrays (AOS) instead of an array of structures (SOA), which was employed in source code example Ch10_06. Following allocation of the grayscale image buffers, Avx512RgbToGs invokes the C++ and assembly language conversion functions. The resultant grayscale image buffers are then compared for equality and saved.

The assembly language code in Listing 14-5 includes two functions: Avx512Rgb2Gs_ and Avx2Rgb2Gs_. As implied by their respective name prefixes, these functions perform RGB to grayscale image conversions using AVX-512 and AVX2 instructions, respectively. The function Avx512Rgb2Gs_ begins its execution by validating num_pixels for size and divisibility by 64. It then checks the source and destination pixel buffers for proper alignment. The ensuing series of vbroadcastss instructions load packed versions of the color conversion coefficients into registers ZMM10, ZMM11, and ZMM12. This is followed by another set of vbroadcastss instructions that broadcast the single-precision floating-point constants 0.5, 255.0, and 0.0 to registers ZMM13, ZMM14, and ZMM15. The mov r8d,r8d instruction zero-extends num_pixels into R8, and the mov r10,16 instruction loads R10 with the number of pixels to process during each loop iteration.

Each Avx512Rgb2Gs_ processing loop iteration in starts with three vpmovzxbd instructions that load 16 red, green, and, blue pixel values into registers ZMM0, ZMM1, and ZMM2. The ensuing vcvtdq2ps instructions convert the doubleword pixel values to single-precision floating-point. The floating-point color values are then multiplied by the corresponding color coefficients using a series of vmulps instructions. These values are then summed using three vaddps instructions. The resultant 16 grayscale pixel values are then clipped to [0.0, 255.0] and converted to doubleword values. The vpmovusdb xmm3,zmm2 instruction size-reduces the doubleword values to bytes using unsigned saturation, and the vmovdqa xmmword ptr [rcx+rax],xmm3 instruction saves the 16 byte pixel values to the destination grayscale image buffer.

The assembly language function Avx2Rgb2Gs_ is identical to its AVX-512 counterpart except for two minor changes: Avx2Rgb2Gs_ uses AVX2 instructions and the YMM register set to carry out the required calculations; it also uses the vpackusdw and vpackuswb instructions in conjunction with a few other instructions to perform the doubleword to byte size reductions. The reason for this is that AVX2 does not support the vpmovusdb instruction. Here is the output for source code example Ch14_05:

```
Grayscale image compare OK

Running benchmark function Avx512RgbToGs_BM - please wait
Benchmark times save to file Ch14_05_Avx512RgbToGs_BM_CHROMIUM.csv
```

Table 14-2 shows the benchmark timing measurements for source code example Ch14_05.

***Table 14-2.*** *Mean Execution Times (Microseconds) for RGB to Grayscale Image Conversion Using TestImage3.bmp*

| CPU | Avx512RgbToGsCpp | Avx512Rgb2Gs_ | Avx2Rgb2Gs_ |
|---|---|---|---|
| i7-4790S | ---- | ---- | ---- |
| i9-7900X | 1125 | 134 | 259 |
| i7-8700K | ---- | ---- | ---- |

The benchmark time differences between the AVX-512 and AVX2 implementations of the RGB to grayscale conversion algorithm are consistent with what one might expect. It is interesting to compare these numbers with the benchmark timing measurements from source code example Ch10_06 (see Table 10-2). This earlier example used an array of RGB32 pixels (or AOS) for the source image buffer, and the mean execution time for the conversion function Avx2ConvertRgbToGs_ was 593 microseconds. The current example exploits separate image pixel buffers for each color component (or SOA), which significantly improves performance.

# Summary

Here are the key learning points for Chapter 14:

- Assembly language functions can use AVX-512 promoted versions of most AVX and AVX2 packed integer instructions to perform operations using 512-, 256-, and 128-bit wide operands.
- Assembly language functions can use the vmovdqa[32|64] and vmovdqu[8|16|32|64] to perform aligned and unaligned moves of packed integer operands.
- Assembly language functions can use the vpmovus[qd|qw|qb|dw|db|wb] instructions to carry out packed integer size reductions using unsigned saturation. AVX-512 also supports an analogous set of packed integer size-reducing instructions using signed saturation.
- The vpcmpu[b|w|d|q] instructions perform packed unsigned integer compare operations and save the resultant compare mask to an opmask register.

- The `vpand[d|q]`, `vpandn[d|q]`, `vpor[d|q]`, and `vpxor[d|q]` instructions can be used with an opmask register to perform merge or zero masking using doubleword or quadword elements.
- The `vextracti[32x4|32x8|64x2|64x4]` instructions can be used to extract packed doubleword or quadword values from a packed integer operand.
- When performing SIMD calculations using packed integer or floating-point operands, a structure-of-arrays construct is often significantly faster than an array-of-structures construct.

CHAPTER 15

# Optimization Strategies and Techniques

In the preceding chapters, you learned the fundamentals of x86-64 assembly language programming. You also learned how to use the computational recourses of Advanced Vector Extensions to perform SIMD operations. To maximize the performance of your x86 assembly language code, it is often necessary to understand important details about the inner workings of an x86 processor. In this chapter, you'll explore the internal hardware components of a modern x86 multi-core processor and its underlying microarchitecture. You'll also learn how to apply specific coding strategies and techniques to boost the performance of your x86-64 assembly language code.

The content of Chapter 15 should be regarded as an introductory tutorial of its topics. A comprehensive examination of x86 microarchitectures and assembly language optimization techniques would minimally require several lengthy chapters, or conceivably an entire book. The primary reference source for this chapter's material is the *Intel 64 and IA-32 Architectures Optimization Reference Manual*. You are encouraged to consult this important reference guide for additional information regarding Intel's x86 microarchitectures and assembly language optimization techniques. The AMD manual *Software Optimization Guide for AMD Family 17h Processors* also contains useful optimization guidance for x86 assembly language programmers. Appendix A includes additional references that contain more information regarding x86 assembly language optimization strategies and techniques.

## Processor Microarchitecture

The performance capabilities of an x86 processor are principally determined by its underlying microarchitecture. A processor's microarchitecture is characterized by the organization and operation of the following internal hardware components: instruction pipelines, decoders, schedulers, Execution Units, data buses, and caches. Software developers who understand the basics of processor microarchitectures can often glean constructive insights that enable them to develop more efficient code.

This section explains processor microarchitecture concepts using Intel's Skylake microarchitecture as an illustrative example. The Skylake microarchitecture is utilized in recent mainstream processors from Intel, including sixth generation Core i3, i5, and i7 series CPUs. Seventh (Kaby Lake) and eighth (Coffee Lake) generation Core series CPUs are also based on the Skylake microarchitecture. The structural organization and operation of earlier Intel microarchitectures such as Sandy Bridge and Haswell are comparable to Skylake. Most of the concepts presented in this section are also applicable to microarchitectures developed and used by AMD in its processors, although the underlying hardware implementations vary. Before proceeding it should be noted that the Skylake microarchitecture discussed in this section is similar to but different from the Skylake Server microarchitecture that was referenced in the chapters that described AVX-512 concepts and programming.

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_15

## Processor Architecture Overview

The architectural details of a processor based on Skylake or any other modern microarchitecture are best examined using the framework of a multi-core processor. Figure 15-1 shows a simplified block diagram of a representative Skylake-based quad-core processor. Note that each CPU core includes first-level (L1) instruction and data caches, which are labeled I-Cache and D-Cache. As implied by their names, these memory caches contain instructions and data that a CPU core can access rapidly. Each CPU core also includes a second-level (L2) unified cache, which holds both instructions and data. Besides improving performance, the L1 and L2 caches enable the CPU cores to execute independent instruction streams in parallel without having to access the higher-level L3 shared cache or main memory.

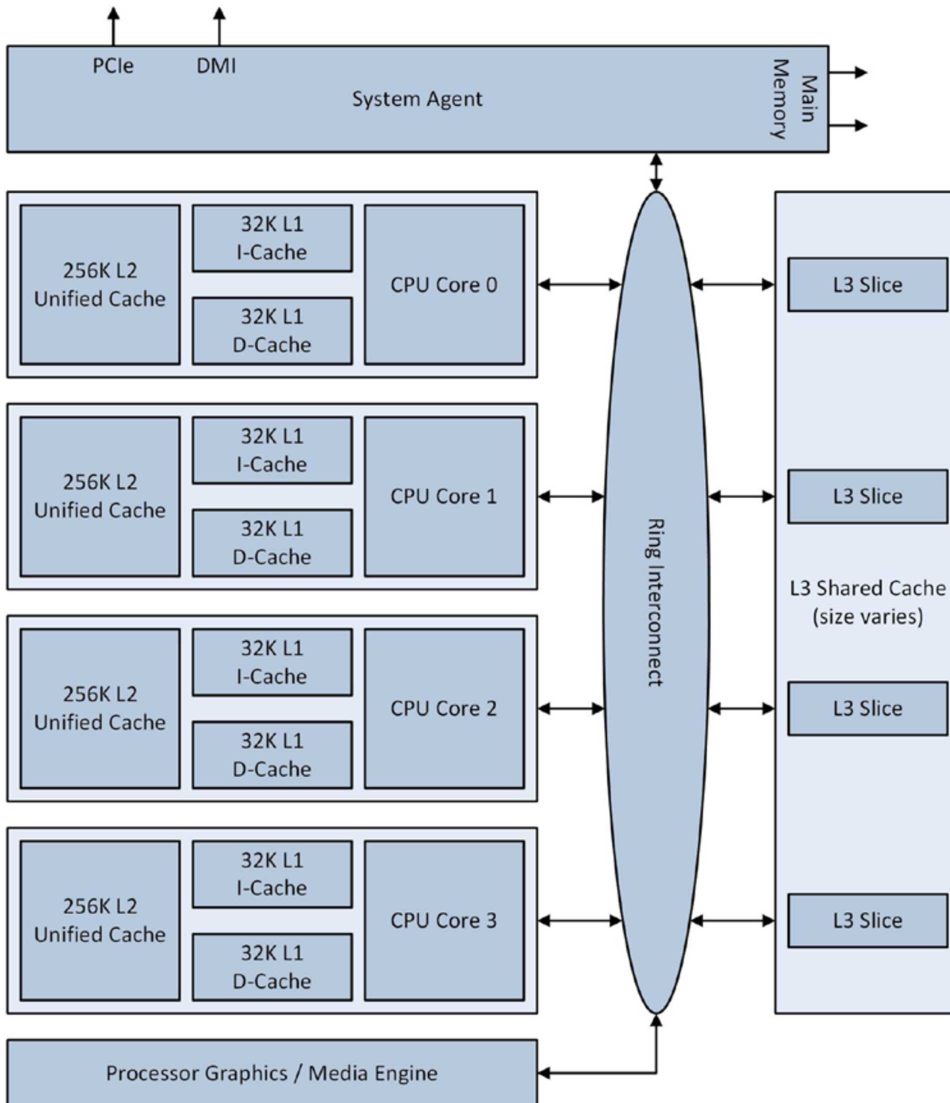



***Figure 15-1.*** *Simplified block diagram of a Skylake-based quad-core processor*

If a CPU core requires an instruction or data item that is not present in its L1 or L2 cache, it must be loaded from the L3 cache or main memory. A processor's L3 cache is partitioned into multiple "slices." Each slice consists of a logic controller and data array. The logic controller manages access to its corresponding data array. It also handles cache misses and writes to main memory. A cache miss occurs when requested data is not present in the L3 cache and must be loaded from main memory (cache misses also occur when data is not available in the L1 or L2 caches). Each L3 data array includes cache memory, which is organized into 64-byte wide packets called cache lines. The Ring Interconnect is a high-speed internal bus that facilitates data transfers between the CPU cores, L3 cache, graphics unit, and System Agent. The System Agent handles data traffic among the processor, its external data buses, and main memory.

## Microarchitecture Pipeline Functionality

During program execution, a CPU core performs five elementary instructional operations: fetch, decode, dispatch, execute, and retire. The particulars of these operations are determined by the functionality of the microarchitecture's pipeline. Figure 15-2 shows a streamlined block diagram of pipeline functionality in a Skylake-based CPU core. In the paragraphs that follow, the operations performed by these pipeline units are examined in greater detail.

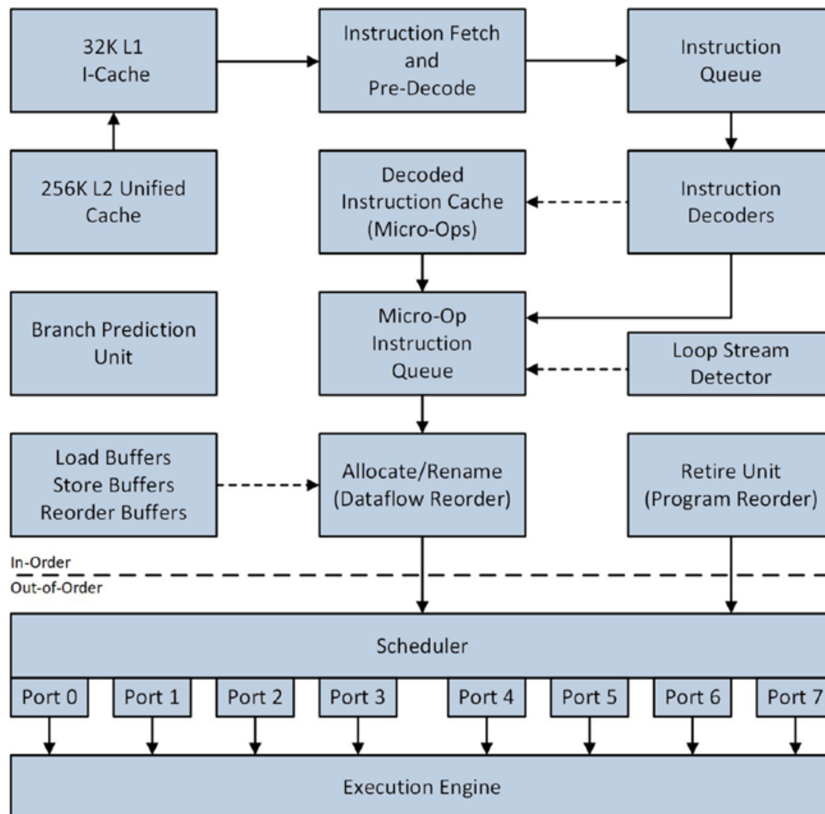



***Figure 15-2.*** *Skylake CPU core pipeline functionality*

The Instruction Fetch and Pre-Decode Unit grabs instructions from the L1 I-Cache and begins the process of preparing them for execution. Steps performed during this stage include instruction length resolution, decoding of x86 instructional prefixes, and property marking to assist the downstream decoders. The Instruction Fetch and Pre-Decode Unit is also responsible for feeding a constant stream of instructions to the Instruction Queue, which queues up instructions for presentation to the Instruction Decoders.

The Instruction Decoders translate x86 instructions into micro-ops. A micro-op is a self-contained low-level instruction that is ultimately executed by one of the Execution Engine's Execution Units, which are discussed in the next section. The number of micro-ops generated by the decoders for an x86 instruction varies depending on its complexity. Simple register-register instructions such as `add eax,edx` and `vpxor xmm0,xmm0,xmm0` are decoded into a single micro-op. Instructions that perform more complex operations, such as `idiv rcx` and `vdivsd ymm0,ymm1,ymm2`, require multiple micro-ops. The translation of x86 instructions into micro-ops facilitates several architectural and performance benefits, including instruction-level parallelism and out-of-order executions.

The Instruction Decoders also perform two ancillary operations that improve utilization of available pipeline bandwidth. The first of these operations is called micro-fusion, which combines simple micro-ops from the same x86 instruction into a single complex micro-op. Examples of micro-fused instructions include memory stores (`mov dword ptr [rbx+16],eax`) and calculating instructions that reference operands in memory (`sub r9,qword ptr [rbp+48]`). Fused complex micro-ops are dispatched by the Execution Engine multiple times (each dispatch executes a simple micro-op from the original instruction). The second ancillary operation carried out by the Instruction Decoders is called macro-fusion. Macro-fusion combines certain commonly-used x86 instruction pairs into a single micro-op. Examples of macro-fusible instruction pairs include many (but not all) conditional jump instructions that are preceded by an `add`, `and`, `cmp`, `dec`, `inc`, `sub`, or `test` instruction.

Micro-ops from the Instruction Decoders are transferred to the Micro-Op Instruction Queue for eventual dispatch by the Scheduler. They're also cached, when necessary, in the Decoded Instruction Cache. The Micro-Op Instruction Queue is also used by the Loop Stream Detector, which identifies and locks small program loops in the Micro-Op Instruction Queue. This improves performance since a small loop can repeatedly execute without requiring any additional instruction fetch, decode, and micro-op cache read operations.

The Allocate/Rename block serves as a bridge between the in-order front-end pipelines and the out-of-order Scheduler and Execution Engine. It allocates any needed internal buffers to the micro-ops. It also eliminates false dependencies between micro-ops, which facilitates out-of-order execution. A false dependency occurs when two micro-ops need to simultaneously access distinct versions of the same hardware resource. (In assembly language code, false dependencies can occur when using instructions that update only the low-order 8 or 16 bits of a 32-bit register.) Micro-ops are then transferred to the Scheduler. This unit queues micro-ops until the necessary source operands are available. It then dispatches ready-to-execute micro-ops to the appropriate Execution Unit in the Execution Engine. The Retire Unit removes micro-ops that have completed their execution using the program's original instruction-ordering pattern. It also signals any processor exceptions that may have occurred during micro-op execution.

Finally, the Branch Prediction Unit helps select the next set of instructions to execute by predicting the branch targets that are most likely to execute based on recent code execution patterns. A branch target is simply the destination operand of a transfer control instruction, such as `jcc`, `jmp`, `call`, or `ret`. The Branch Prediction Unit enables a CPU core to speculatively execute the micro-ops of an instruction before the outcome of a branch decision is known. When necessary, a CPU core searches (in order) the Decoded Instruction Cache, L1 I-Cache, L2 Unified Cache, L3 Cache, and main memory for instructions to execute.

## Execution Engine

The Execution Engine executes micro-ops passed to it by the Scheduler. Figure 15-3 shows a high-level block diagram of a Skylake-based CPU core Execution Engine. The rectangular blocks beneath each dispatch port represent distinct micro-op Execution Units. Note that four of the Scheduler ports facilitate access to Execution Units that carry out calculating functions including integer, floating-point, and SIMD arithmetic. The remaining four ports support memory load and store operations.

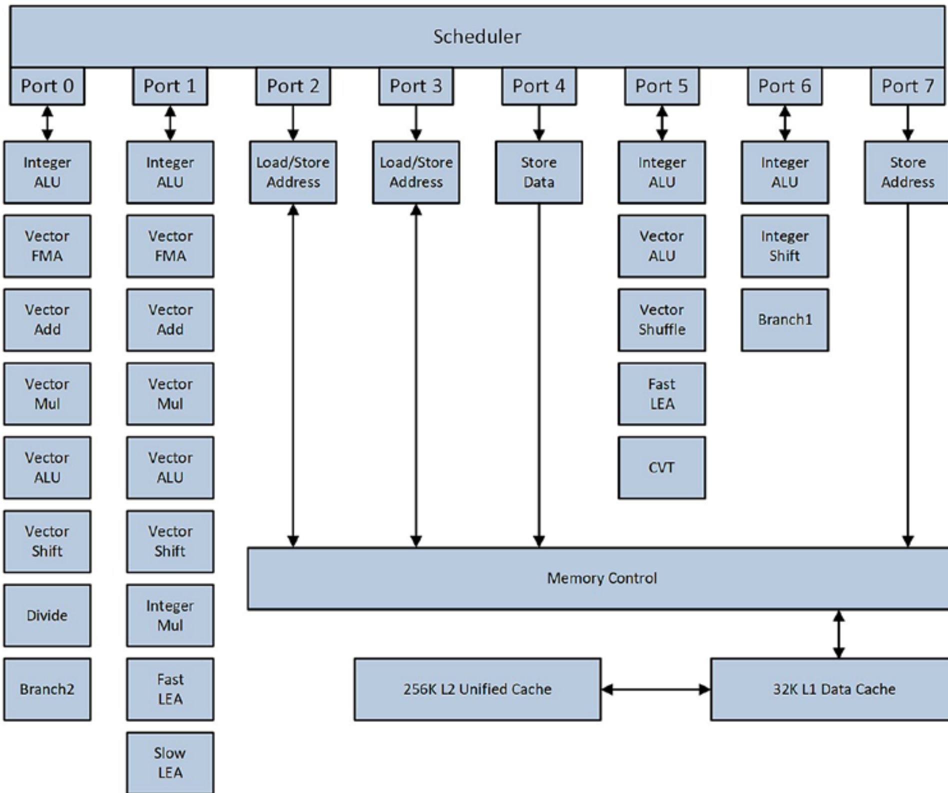



***Figure 15-3.*** *Skylake CPU core Execution Engine and its Execution Units*

Each Execution Unit performs a specific calculation or operation. For example, the Integer ALU (Arithmetic Logic Unit) Execution Units carry out integer addition, subtraction, and compare operations. The Vector ALU Execution Units handle SIMD integer arithmetic and bitwise Boolean operations. Note that the Execution Engine contains multiple instances of select Execution Units. This allows the Execution Engine to simultaneously execute multiple instances of certain micro-ops in parallel. For example, the Execution Engine can concurrently perform three separate SIMD bitwise Boolean operations in parallel using the Vector ALU Execution Units.

Each Skylake core Scheduler can dispatch a maximum of eight micro-ops per cycle (one per port) to the Execution Engine. The out-of-order engine, which includes the Scheduler, Execution Engine, and Retire Unit, supports up to 224 “in-flight” (or coexistent) micro-ops. Table 15-1 shows key buffer sizes for recent Intel microarchitectures.

***Table 15-1.*** *Comparison of Key Buffer Sizes for Recent Intel Microarchitectures*

| Parameter | Sandy Bridge (2nd Gen) | Haswell (4th Gen) | Skylake (6th Gen) |
|---|---|---|---|
| Dispatch ports | 6 | 8 | 8 |
| In-flight micro-ops | 168 | 192 | 224 |
| In-flight loads | 64 | 72 | 72 |
| In-flight stores | 36 | 42 | 56 |
| Scheduler entries | 54 | 60 | 97 |
| Integer register file | 160 | 168 | 180 |
| Floating-point register file | 144 | 168 | 168 |

# Optimizing Assembly Language Code

This section discusses some basic optimization strategies and techniques that you can use to improve the performance of your x86 64-bit assembly language code. These techniques are recommended for use in code that targets recent Intel microarchitectures, including Skylake Server, Skylake, Haswell, and Sandy Bridge. Most of techniques are also appropriate for use in code that will execute on recent AMD processors. The optimization strategies and techniques are organized into five generic categories:

- Basic techniques
- Floating-point arithmetic
- Program branches
- Data alignment
- SIMD techniques

It is important to keep in mind that the optimization techniques mentioned in this section must be applied in a prudent manner. For example, it makes little sense to add extra `push` and `pop` (or other) instructions to a function just to use recommended instruction form only once. Moreover, none of the optimization strategies and techniques described in this section will remedy an inappropriate or poorly designed algorithm. The *Intel 64 and IA-32 Architectures Optimization Reference Manual* contains additional information regarding the optimization strategies techniques discussed in this section. Appendix A also contains additional references that you can consult for more information regarding optimization of x86 assembly language code.

## Basic Techniques

The following coding strategies and techniques are frequently employed to improve the performance of x86-64 assembly language code.

- Use a `test` instruction instead of a `cmp` instruction when possible, especially to carry out a simple less than, equal to, or greater than zero test.
- Avoid using the memory-immediate forms of the `cmp` and test instructions (e.g., `cmp dword ptr [rbp+40],100` or `test byte ptr [r12],0fh`). Instead, load the memory value into a register and use the register-immediate form of the `cmp` or `test` instruction (e.g., `mov eax,dword ptr [rbp+40]` followed by `cmp eax,100`).

- Minimize use of instructions that perform partial updates of the status flags in RFLAGS. For example, the instructions `add eax,1` or `sub rax,1` may be faster than `inc eax` or `dec rax`, especially in performance-critical loops (the `inc` and `dec` instructions do not update RFLAGS.CF).
- Use an `xor` or `sub` instruction to zero a register instead of a `mov` instruction. For example, use an `xor eax,eax` or `sub eax,eax` instruction instead of `mov eax,0`. The `mov` instruction form can be used when it's necessary to avoid modifying the status flags in RFLAGS.
- Avoid using instructions that require an operand-size prefix to load a 16-bit immediate value since instructions with operand-size prefixes take longer to decode. Use an equivalent 32-bit immediate value instead. For example, use `mov edx,42` instead of `mov dx,42`.
- Use 32-bit instead of 64-bit instruction forms and general-purpose registers when possible. For example, if the maximum number of for-loop iterations does not exceed the range limits of a 32-bit integer, use a 32-bit instead of a 64-bit general-purpose register for the loop counter.
- Use 32-bit instruction forms to load 64-bit registers with positive constant values. For example, the instructions `mov eax,16` and `mov r8d,42` effectively set RAX to 16 and R8 to 42.
- Use the two- or three-operand form of the `imul` instruction to multiply two signed integers when the full-width product is not needed. For example, use `imul rax,rcx` when a 64-bit truncated product is sufficient instead of `imul rcx`, which returns a 128-bit product in RDX:RAX. This guideline also applies to 32-bit signed integer multiplication.
- Avoid declaring data values inside a code section. In situations where it's necessary to do this (e.g., when defining a read-only jump table), position the data after an unconditional `jmp` or `ret` instruction.
- In performance-critical processing loops, minimize use of the `lea` instruction that contains three effective address components (e.g., base register, index register, and displacement). These instructions can only be dispatched to the Slow LEA Execution Unit through Port 1. Shorter forms (one or two effective address components) of the `lea` instruction can be dispatched via ports 1 or 5 to one of the Fast LEA Execution Units.
- Load any memory values that are needed for multiple calculations into a register. If a memory value is needed only for a single calculation, use the register-memory form of the calculating instruction. Table 15-2 shows several examples.

***Table 15-2.*** *Instruction Form Examples for Single and Multiple-Use Memory Values*

| Register-Memory Form (Single-Use Data) | Move and Register-Register Form (Multiple-Use Data) |
|---|---|
| `add edx,dword ptr [x]` | `mov eax,dword ptr [x]`<br>`add edx,eax` |
| `and rax,qword ptr [rbx+16]` | `mov rcx,[rbx+16]`<br>`and rax,rcx` |
| `cmp ecx,dword ptr [n]` | `mov eax,dword ptr [n]`<br>`cmp ecx,eax` |
| `vmulpd xmm0,xmm2,xmmword ptr [rdx]` | `vmovapd xmm1,xmmword ptr [rdx]`<br>`vmulpd xmm0,xmm2,xmm1` |

## Floating-Point Arithmetic

The following coding strategies and techniques can be employed to improve the performance of x86-64 assembly language code that performs floating-point operations. These guidelines apply to both scalar and packed floating-point calculations.

- Always use the computational resources of x86-AVX to perform scalar floating-point arithmetic. Do not use the legacy x87 floating-point unit to perform these types of calculations.
- Use single-precision floating-point values instead of double-precision values whenever possible.
- Arrange floating-point instruction sequences to minimize register dependencies. Exploit multiple destination registers to save intermediate results, then reduce the intermediate results to a single value (see example `Ch11_01`).
- Partially (or completely) unroll processing loops that contain floating-point calculations, especially loops that contain sequences of floating-point addition, multiplication, or FMA operations.
- Avoid arithmetic underflows and denormal values during arithmetic calculations whenever possible.
- Avoid using denormalized floating-point constants.
- If excessive arithmetic underflows are expected, consider enabling the flush-to-zero (MXCSR.FTZ) and denormals-are-zero (MXCSR.DAZ) modes. See Chapter 4 for more information regarding the proper use of these modes.

## Program Branches

Program branch instructions, such as `jmp`, `call`, and `ret`, are potentially time-consuming operations to perform since they can affect the contents of the front-end pipelines and internal caches. The conditional jump instruction `jcc` is also a performance concern given its frequency of use. The following optimization techniques can be employed to minimize the adverse performance effects of branch instructions and improve the accuracy of the Branch Prediction Unit:

- Organize code to minimize the number of possible branch instructions.
- Partially (or completely) unroll short processing loops to minimize the number of executed conditional jump instructions. Avoid excessive loop unrolling since this may result in slower executing code due to less efficient use of the Loop Stream Detector (see Figure 15-2).
- Eliminate unpredictable data-dependent branches using the `setcc` or `cmovcc` instructions.
- Align branch targets in performance-critical loops to 16-byte boundaries.
- Move conditional code that is unlikely to execute (e.g., error-handling code) to another program (or `.code`) section or memory page.

The Branch Prediction Unit employs both static and dynamic techniques to predict the target of a jump instruction. Incorrect branch predictions can be minimized if blocks of code containing conditional jump instructions are arranged such that they're consistent with the Branch Prediction Unit's static prediction algorithm:

- Use forward conditional jumps when the fall-through code is more likely to execute.
- Use backward conditional jumps when the fall-through code is less likely to execute.

The forward conditional jump approach is frequently used in blocks of code that perform function argument validation. The backward conditional jump technique often employed at the bottom of a processing loop code block following a counter update or other loop-terminating test decision. Listing 15-1 contains a short assembly language function that illustrates these practices in greater detail.

***Listing 15-1.*** Example Ch15_01

```
;-------------------------------------------------
;               Ch15_01.asm
;-------------------------------------------------

        .const
r8_2p0  real8 2.0

; extern "C" int CalcResult_(double* y, const double* x, size_t n);

        .code
CalcResult_ proc

; Forward conditional jumps are used in this code block since
; the fall-through cases are more likely to occur
        test r8,r8
        jz Done                             ;jump if n == 0

        test r8,7h
        jnz Error                           ;jump if (n % 8) != 0
        test rcx,1fh
        jnz Error                           ;jump if y is not aligned to a 32b boundary
        test rdx,1fh
        jnz Error                           ;jump if x is not aligned to a 32b boundary
```

```
; Initialize
        xor eax,eax                             ;set array offset to zero
        vbroadcastsd ymm5,real8 ptr [r8_2p0]    ;packed 2.0

; Simple array processing loop
        align 16
@@:     vmovapd ymm0,ymmword ptr [rdx+rax]      ;load x[i+3]:x[i]
        vdivpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vmovapd ymmword ptr [rcx+rax],ymm2      ;save y[i+3]:y[i]

        vmovapd ymm0,ymmword ptr [rdx+rax+32]   ;load x[i+7]:x[i+4]
        vdivpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vmovapd ymmword ptr [rcx+rax+32],ymm2   ;save y[i+7]:y[i+4]

; A backward conditional jump is used in this code block since
; the fall-through case is less likely to occur
        add rax,64
        sub r8,8
        jnz @B

Done:   xor eax,eax                         ;set success return code
        vzeroupper
        ret

; Error handling code that's unlikely to execute

Error:  mov eax,1                           ;set error return code
        ret
CalcResult_ endp
        end
```

## Data Alignment

It's been mentioned (perhaps excessively) multiple times in this book, but the importance of using properly aligned data cannot be over emphasized. Programs that manipulate improperly aligned data are likely to trigger the processor into performing additional memory cycles and micro-op executions, which can adversely affect overall system performance. The following data alignment practices should be considered universal truths and always observed:

- Align multi-byte integer and floating-point values to their natural boundaries.
- Align 128-, 256-, and 512-bit wide packed integer and floating-point values to their proper boundaries.
- Pad data structures if necessary to ensure proper alignment of each structure member.
- Use the appropriate C++ language specifiers and library functions to align data items that are allocated in high-level code. Visual C++ functions can use the `alignas(n)` specifier or call `_aligned_malloc` to properly align data items.
- Give preference to aligned stores over aligned loads.

The following data arrangement techniques are also recommended:

- Align and position small arrays and short text strings in a data structure to avoid cache line splits. A cache line split occurs when the bytes of a multi-byte value are split across a 64-byte boundary. Positioning small multi-byte values on the same cache line helps minimize the number of memory cycles that the processor must perform.
- Evaluate the performance effects of different data layouts such as structure of arrays versus array of structures.

## SIMD Techniques

The following techniques should be observed, when appropriate, by any function that performs SIMD computations using AVX, AVX2, or AVX-512 instructions.

- Do not code functions that intermix x86-AVX and x86-SSE instructions. It is okay to code functions that intermix AVX, AVX2, and AVX-512 instructions.
- Minimize register dependencies to exploit multiple Execution Units in the Execution Engine.
- Load multiple-use memory operands and packed constants into a register.
- On systems that support AVX-512, exploit the extra SIMD registers to minimize data dependencies and register spills. A register spill occurs when a function must temporarily save the contents of a register to memory in order to free the register for other calculations.
- Use a `vpxor`, `vxorp[d|s]`, etc. instruction to zero a register instead of a data move instruction. For example, `vxorps xmm0,xmm0` is preferred over `vmovaps xmm0,xmmword ptr [XmmZero]`.
- Use x86-AVX masking and Boolean operations to minimize or eliminate data-dependent conditional jump instructions.
- Perform packed data loads and stores using the aligned move instructions (e.g., `vmovdqa`, `vmovap[d|s]`, etc.).
- Process SIMD arrays using small data blocks to maximize reuse of resident cache data.
- Use the `vzeroupper` instruction when required to avoid x86-AVX to x86-SSE state transition penalties.
- Use the doubleword forms of the gather and scatter instructions instead of the quadword forms when possible (e.g., use `vgatherdp[d|s]` and doubleword indices instead of `vgatherqp[d|s]` and quadword indices). Perform any required gather operations well ahead of when the data is needed.

The following practices can also be employed to improve the performance of certain algorithms that perform SIMD encoding and decoding operations:

- Use the non-temporal store instructions (e.g., `vmovntdqa`, `vmovntp[d|s]`, etc.) to minimize cache pollution.
- Use the data prefetch instructions (e.g., `prefetcht0`, `prefetchnta`, etc.) to notify the processor of expected-use data items.

Chapter 16 contains a couple of source code examples that illustrate how to use the non-temporal store and data prefetch instructions.

# Summary

Here are the key learning points for Chapter 15:

- The performance of most assembly language functions can be improved by implementing the optimization strategies and techniques outlined in this chapter.
- The recommended optimization techniques must be judiciously applied. It is not uncommon to encounter coding situations where a recommend strategy or technique is not the best approach.
- To achieve optimal performance for a specific algorithm or function, it may be necessary to code multiple versions and compare benchmark timing measurements.
- When developing assembly language code, don't spend an excessive amount of time trying to maximize performance. Focus on performance gains that are relatively easy to attain (e.g., implementing an algorithm using SIMD instead of scalar arithmetic).
- None of the optimization strategies and techniques presented in this chapter will ameliorate an inappropriate or poorly designed algorithm.

CHAPTER 16

# Advanced Programming

The final chapter of this book reviews several source code examples that demonstrate advanced x86 assembly language programming techniques. The first example explains how to use the `cpuid` instruction to detect specific x86 instruction set extensions. This is followed by two examples that illustrate how to accelerate SIMD processing functions using non-temporal memory stores and data prefetch instructions. The concluding example elucidates the use of an assembly language calculating function in a multithreaded application.

## CPUID Instruction

It's been mentioned several times in this book that an application program should never assume that a specific instruction set extension such as AVX, AVX2, or AVX-512 is available simply by knowing the processor's microarchitecture, model number, or brand name. An application program should always test for the presence of an instruction set extension using the `cupid` (CPU Identification) instruction. Application programs can use this instruction to verify that a processor supports one of the previously-mentioned x86-AVX instruction set extensions. The `cpuid` instruction can also be used to obtain additional processor feature information that's useful or needed in both application programs and operating system software.

Listing 16-1 shows the source code for example `Ch16_01`. This example demonstrates how to use the `cupid` instruction to determine processor support for various instruction set extensions. The source code for example `Ch16_01` focuses on using `cpuid` to detect architectural features and instruction set extensions that are allied with this book's content. If you're interested in learning how to use the `cpuid` instruction to identify other processor features, you should consult the AMD and Intel reference manuals that are listed in Appendix A.

***Listing 16-1.*** Example Ch16_01

```
//------------------------------------------------
//               CpuidInfo.h
//------------------------------------------------

#pragma once
#include <cstdint>
#include <vector>
#include <string>
```

© Daniel Kusswurm 2018
D. Kusswurm, *Modern X86 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-4063-2_16

```
struct CpuidRegs
{
    uint32_t EAX;
    uint32_t EBX;
    uint32_t ECX;
    uint32_t EDX;
};

class CpuidInfo
{
public:
    class CacheInfo
    {
    public:
        enum class Type
        {
            Unknown, Data, Instruction, Unified
        };

    private:
        uint32_t m_Level = 0;
        Type m_Type = Type::Unknown;
        uint32_t m_Size = 0;

    public:
        uint32_t GetLevel(void) const             { return m_Level; }
        uint32_t GetSize(void) const              { return m_Size; }
        Type GetType(void) const                  { return m_Type; }

        // These are defined in CacheInfo.cpp
        CacheInfo(uint32_t level, uint32_t type, uint32_t size);
        std::string GetTypeString(void) const;
    };

private:
    uint32_t m_MaxEax;                            // Max EAX for basic CPUID
    uint32_t m_MaxEaxExt;                         // Max EAX for extended CPUID
    uint64_t m_FeatureFlags;                      // Processor feature flags
    std::vector<CpuidInfo::CacheInfo> m_CacheInfo;  // Processor cache information
    char m_VendorId[13];                          // Processor vendor ID string
    char m_ProcessorBrand[49];                    // Processor brand string
    bool m_OsXsave;                               // XSAVE is enabled for app use
    bool m_OsAvxState;                            // AVX state is enabled by OS
    bool m_OsAvx512State;                         // AVX-512 state is enabled by OS

    void Init(void);
    void InitProcessorBrand(void);
    void LoadInfo0(void);
    void LoadInfo1(void);
```

```
    void LoadInfo2(void);
    void LoadInfo3(void);
    void LoadInfo4(void);
    void LoadInfo5(void);

public:
    enum class FF : uint64_t
    {
        FXSR                = (uint64_t)1 << 0,
        MMX                 = (uint64_t)1 << 1,
        MOVBE               = (uint64_t)1 << 2,
        SSE                 = (uint64_t)1 << 3,
        SSE2                = (uint64_t)1 << 4,
        SSE3                = (uint64_t)1 << 5,
        SSSE3               = (uint64_t)1 << 6,
        SSE4_1              = (uint64_t)1 << 7,
        SSE4_2              = (uint64_t)1 << 8,
        PCLMULQDQ           = (uint64_t)1 << 9,
        POPCNT              = (uint64_t)1 << 10,
        PREFETCHW           = (uint64_t)1 << 11,
        PREFETCHWT1         = (uint64_t)1 << 12,
        RDRAND              = (uint64_t)1 << 13,
        RDSEED              = (uint64_t)1 << 14,
        ERMSB               = (uint64_t)1 << 15,
        AVX                 = (uint64_t)1 << 16,
        AVX2                = (uint64_t)1 << 17,
        F16C                = (uint64_t)1 << 18,
        FMA                 = (uint64_t)1 << 19,
        BMI1                = (uint64_t)1 << 20,
        BMI2                = (uint64_t)1 << 21,
        LZCNT               = (uint64_t)1 << 22,
        ADX                 = (uint64_t)1 << 23,
        AVX512F             = (uint64_t)1 << 24,
        AVX512ER            = (uint64_t)1 << 25,
        AVX512PF            = (uint64_t)1 << 26,
        AVX512DQ            = (uint64_t)1 << 27,
        AVX512CD            = (uint64_t)1 << 28,
        AVX512BW            = (uint64_t)1 << 29,
        AVX512VL            = (uint64_t)1 << 30,
        AVX512_IFMA         = (uint64_t)1 << 31,
        AVX512_VBMI         = (uint64_t)1 << 32,
        AVX512_4FMAPS       = (uint64_t)1 << 33,
        AVX512_4VNNIW       = (uint64_t)1 << 34,
        AVX512_VPOPCNTDQ    = (uint64_t)1 << 35,
        AVX512_VNNI         = (uint64_t)1 << 36,
        AVX512_VBMI2        = (uint64_t)1 << 37,
        AVX512_BITALG       = (uint64_t)1 << 38,
        CLWB                = (uint64_t)1 << 39,
    };
```

```
    CpuidInfo(void) { Init(); };
    ~CpuidInfo() {};

    const std::vector<CpuidInfo::CacheInfo>& GetCacheInfo(void) const
    {
        return m_CacheInfo;
    }

    bool GetFF(FF flag) const
    {
        return ((m_FeatureFlags & (uint64_t)flag) != 0) ? true : false;
    }

    std::string GetProcessorBrand(void) const   { return std::string(m_ProcessorBrand); }
    std::string GetVendorId(void) const         { return std::string(m_VendorId); }

    void LoadInfo(void);
};

// Cpuinfo_.asm
extern "C" void Xgetbv_(uint32_t r_ecx, uint32_t* r_eax, uint32_t* r_edx);
extern "C" uint32_t Cpuid_(uint32_t r_eax, uint32_t r_ecx, CpuidRegs* r_out);

;-------------------------------------------------
;               CpuidInfo_.asm
;-------------------------------------------------

; The following structures must agree with the CpuidRegs structure
; that's defined in CpuidInfo.h

CpuidRegs   struct
RegEAX      dword ?
RegEBX      dword ?
RegECX      dword ?
RegEDX      dword ?
CpuidRegs   ends

; extern "C" uint32_t Cpuid_(uint32_t r_eax, uint32_t r_ecx, CpuidRegs* r_out);
;
; Returns:      eax == 0     Unsupported CPUID leaf
;               eax != 0     Supported CPUID leaf
;
;               Note: the return code is valid only if r_eax <= MaxEAX.

        .code
Cpuid_  proc frame
        push rbx
        .pushreg rbx
        .endprolog
```

```
; Load eax and ecx
        mov eax,ecx
        mov ecx,edx

; Get cpuid info & save results
        cpuid
        mov dword ptr [r8+CpuidRegs.RegEAX],eax
        mov dword ptr [r8+CpuidRegs.RegEBX],ebx
        mov dword ptr [r8+CpuidRegs.RegECX],ecx
        mov dword ptr [r8+CpuidRegs.RegEDX],edx

; Test for unsupported CPUID leaf
        or eax,ebx
        or ecx,edx
        or eax,ecx                          ;eax = return code

        pop rbx
        ret
Cpuid_  endp

; extern "C" void Xgetbv_(uint32_t r_ecx, uint32_t* r_eax, uint32_t* r_edx);

Xgetbv_ proc
        mov r9,rdx                          ;r9 = r_eax ptr
        xgetbv

        mov dword ptr [r9],eax              ;save low word result
        mov dword ptr [r8],edx              ;save high word result
        ret
Xgetbv_ endp
        end

//------------------------------------------------
//              Ch16_01.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <string>
#include "CpuidInfo.h"

using namespace std;

static void DisplayCacheInfo(const CpuidInfo& ci);
static void DisplayFeatureFlags(const CpuidInfo& ci);
```

```
int main()
{
    CpuidInfo ci;

    ci.LoadInfo();

    cout << ci.GetVendorId() << '\n';
    cout << ci.GetProcessorBrand() << '\n';

    DisplayCacheInfo(ci);
    DisplayFeatureFlags(ci);
    return 0;
}

static void DisplayCacheInfo(const CpuidInfo& ci)
{
    const vector<CpuidInfo::CacheInfo>& cache_info = ci.GetCacheInfo();

    for (const CpuidInfo::CacheInfo& x : cache_info)
    {
        uint32_t cache_size = x.GetSize();
        string cache_size_str;

        if (cache_size < 1024 * 1024)
        {
            cache_size /= 1024;
            cache_size_str = "KB";
        }
        else
        {
            cache_size /= 1024 * 1024;
            cache_size_str = "MB";
        }

        cout << "Cache L" << x.GetLevel() << ": ";
        cout << cache_size << cache_size_str << ' ';
        cout << x.GetTypeString() << '\n';
    }
}

static void DisplayFeatureFlags(const CpuidInfo& ci)
{
    const char nl = '\n';

    cout << "----- CPUID Feature Flags -----" << nl;
    cout << "ADX:         " << ci.GetFF(CpuidInfo::FF::ADX) << nl;
    cout << "AVX:         " << ci.GetFF(CpuidInfo::FF::AVX) << nl;
    cout << "AVX2:        " << ci.GetFF(CpuidInfo::FF::AVX2) << nl;
    cout << "AVX512F:     " << ci.GetFF(CpuidInfo::FF::AVX512F) << nl;
    cout << "AVX512BW:    " << ci.GetFF(CpuidInfo::FF::AVX512BW) << nl;
    cout << "AVX512CD:    " << ci.GetFF(CpuidInfo::FF::AVX512CD) << nl;
```

```
    cout << "AVX512DQ:    " << ci.GetFF(CpuidInfo::FF::AVX512DQ) << nl;
    cout << "AVX512ER:    " << ci.GetFF(CpuidInfo::FF::AVX512ER) << nl;
    cout << "AVX512PF:    " << ci.GetFF(CpuidInfo::FF::AVX512PF) << nl;
    cout << "AVX512VL:    " << ci.GetFF(CpuidInfo::FF::AVX512VL) << nl;
    cout << "AVX512_IFMA: " << ci.GetFF(CpuidInfo::FF::AVX512_IFMA) << nl;
    cout << "AVX512_VBMI: " << ci.GetFF(CpuidInfo::FF::AVX512_VBMI) << nl;
    cout << "BMI1:        " << ci.GetFF(CpuidInfo::FF::BMI1) << nl;
    cout << "BMI2:        " << ci.GetFF(CpuidInfo::FF::BMI2) << nl;
    cout << "F16C:        " << ci.GetFF(CpuidInfo::FF::F16C) << nl;
    cout << "FMA:         " << ci.GetFF(CpuidInfo::FF::FMA) << nl;
    cout << "LZCNT:       " << ci.GetFF(CpuidInfo::FF::LZCNT) << nl;
    cout << "POPCNT:      " << ci.GetFF(CpuidInfo::FF::POPCNT) << nl;
}

//------------------------------------------------
//               CpuidInfo.cpp
//------------------------------------------------

#include "stdafx.h"
#include <string>
#include <cstring>
#include <vector>
#include "CpuidInfo.h"

using namespace std;

void CpuidInfo::LoadInfo(void)
{
    // Note: LoadInfo0 must be called first
    LoadInfo0();
    LoadInfo1();
    LoadInfo2();
    LoadInfo3();
    LoadInfo4();
    LoadInfo5();
}

void CpuidInfo::LoadInfo0(void)
{
    CpuidRegs r1;

    // Perform required initializations
    Init();

    // Get MaxEax and VendorID
    Cpuid_(0, 0, &r1);
    m_MaxEax = r1.EAX;
    *(uint32_t *)(m_VendorId + 0) = r1.EBX;
    *(uint32_t *)(m_VendorId + 4) = r1.EDX;
    *(uint32_t *)(m_VendorId + 8) = r1.ECX;
    m_VendorId[sizeof(m_VendorId) - 1] = '\0';
```

```
    // Get MaxEaxExt
    Cpuid_(0x80000000, 0, &r1);
    m_MaxEaxExt = r1.EAX;

    // Initialize processor brand string
    InitProcessorBrand();
}

void CpuidInfo::LoadInfo1(void)
{
    CpuidRegs r;

    if (m_MaxEax < 1)
        return;

    Cpuid_(1, 0, &r);

    //
    // Decode r.ECX flags
    //

    // CPUID.(EAX=01H, ECX=00H):ECX.SSE3[bit 0]
    if (r.ECX & (0x1 << 0))
        m_FeatureFlags |= (uint64_t)FF::SSE3;

    // CPUID.(EAX=01H, ECX=00H):ECX.PCLMULQDQ[bit 1]
    if (r.ECX & (0x1 << 1))
        m_FeatureFlags |= (uint64_t)FF::PCLMULQDQ;

    // CPUID.(EAX=01H, ECX=00H):ECX.SSSE3[bit 9]
    if (r.ECX & (0x1 << 9))
        m_FeatureFlags |= (uint64_t)FF::SSSE3;

    // CPUID.(EAX=01H, ECX=00H):ECX.SSE4.1[bit 19]
    if (r.ECX & (0x1 << 19))
        m_FeatureFlags |= (uint64_t)FF::SSE4_1;

    // CPUID.(EAX=01H, ECX=00H):ECX.SSE4.2[bit 20]
    if (r.ECX & (0x1 << 20))
        m_FeatureFlags |= (uint64_t)FF::SSE4_2;

    // CPUID.(EAX=01H, ECX=00H):ECX.MOVBE[bit 22]
    if (r.ECX & (0x1 << 22))
        m_FeatureFlags |= (uint64_t)FF::MOVBE;

    // CPUID.(EAX=01H, ECX=00H):ECX.POPCNT[bit 23]
    if (r.ECX & (0x1 << 23))
        m_FeatureFlags |= (uint64_t)FF::POPCNT;

    // CPUID.(EAX=01H, ECX=00H):ECX.RDRAND[bit 30]
    if (r.ECX & (0x1 << 30))
        m_FeatureFlags |= (uint64_t)FF::RDRAND;
```

```
    //
    // Decode r.RDX flags
    //

    // CPUID.(EAX=01H, ECX=00H):EDX.MMX[bit 23]
    if (r.EDX & (0x1 << 23))
        m_FeatureFlags |= (uint64_t)FF::MMX;

    // CPUID.(EAX=01H, ECX=00H):EDX.FXSR[bit 24]
    if (r.EDX & (0x1 << 24))
        m_FeatureFlags |= (uint64_t)FF::FXSR;

    // CPUID.(EAX=01H, ECX=00H):EDX.SSE[bit 25]
    if (r.EDX & (0x1 << 25))
        m_FeatureFlags |= (uint64_t)FF::SSE;

    // CPUID.(EAX=01H, ECX=00H):EDX.SSE2[bit 26]
    if (r.EDX & (0x1 << 26))
        m_FeatureFlags |= (uint64_t)FF::SSE2;
}

void CpuidInfo::LoadInfo2(void)
{
    CpuidRegs   r;

    if (m_MaxEax < 7)
        return;

    Cpuid_(7, 0, &r);

    // CPUID.(EAX=07H, ECX=00H):ECX.PREFETCHWT1[bit 0]
    if (r.ECX & (0x1 << 0))
        m_FeatureFlags |= (uint64_t)FF::PREFETCHWT1;

    // CPUID.(EAX=07H, ECX=00H):EBX.BMI1[bit 3]
    if (r.EBX & (0x1 << 3))
        m_FeatureFlags |= (uint64_t)FF::BMI1;

    // CPUID.(EAX=07H, ECX=00H):EBX.BMI2[bit 8]
    if (r.EBX & (0x1 << 8))
        m_FeatureFlags |= (uint64_t)FF::BMI2;

    // CPUID.(EAX=07H, ECX=00H):EBX.ERMSB[bit 9]
    // ERMSB = Enhanced REP MOVSB/STOSB
    if (r.EBX & (0x1 << 9))
        m_FeatureFlags |= (uint64_t)FF::ERMSB;

    // CPUID.(EAX=07H, ECX=00H):EBX.RDSEED[bit 18]
    if (r.EBX & (0x1 << 18))
        m_FeatureFlags |= (uint64_t)FF::RDSEED;
```

```
    // CPUID.(EAX=07H, ECX=00H):EBX.ADX[bit 19]
    if (r.EBX & (0x1 << 19))
        m_FeatureFlags |= (uint64_t)FF::ADX;

    // CPUID.(EAX=07H, ECX=00H):EBX.CLWB[bit 24]
    if (r.EBX & (0x1 << 24))
        m_FeatureFlags |= (uint64_t)FF::CLWB;
}

void CpuidInfo::LoadInfo3(void)
{
    CpuidRegs r;

    if (m_MaxEaxExt < 0x80000001)
        return;

    Cpuid_(0x80000001, 0, &r);

    // CPUID.(EAX=80000001H, ECX=00H):ECX.LZCNT[bit 5]
    if (r.ECX & (0x1 << 5))
        m_FeatureFlags |= (uint64_t)FF::LZCNT;

    // CPUID.(EAX=80000001H, ECX=00H):ECX.PREFETCHW[bit 8]
    if (r.ECX & (0x1 << 8))
        m_FeatureFlags |= (uint64_t)FF::PREFETCHW;
}

void CpuidInfo::LoadInfo4(void)
{
    CpuidRegs r_eax01h;
    CpuidRegs r_eax07h;

    if (m_MaxEax < 7)
        return;

    Cpuid_(1, 0, &r_eax01h);
    Cpuid_(7, 0, &r_eax07h);

    // Test CPUID.(EAX=01H, ECX=00H):ECX.OSXSAVE[bit 27] to verify use of XGETBV
    m_OsXsave = (r_eax01h.ECX & (0x1 << 27)) ? true : false;

    if (m_OsXsave)
    {
        // Use XGETBV to obtain following information
        // AVX state is enabled by OS if (XCR0[2:1] == '11b') is true
        // AVX512 state is enabled by OS if (XCR0[7:5] == '111b') is true

        uint32_t xgetbv_eax, xgetbv_edx;

        Xgetbv_(0, &xgetbv_eax, &xgetbv_edx);
        m_OsAvxState = (((xgetbv_eax >> 1) & 0x03) == 0x03) ? true : false;
```

```
if (m_OsAvxState)
{
    // CPUID.(EAX=01H, ECX=00H):ECX.AVX[bit 28]
    if (r_eax01h.ECX & (0x1 << 28))
    {
        m_FeatureFlags |= (uint64_t)FF::AVX;

        // CPUID.(EAX=01H, ECX=00H):ECX.FMA[bit 12]
        if (r_eax01h.ECX & (0x1 << 12))
            m_FeatureFlags |= (uint64_t)FF::FMA;

        // CPUID.(EAX=01H, ECX=00H):ECX.F16C[bit 29]
        if (r_eax01h.ECX & (0x1 << 29))
            m_FeatureFlags |= (uint64_t)FF::F16C;

        // CPUID.(EAX=07H, ECX=00H):EBX.AVX2[bit 5]
        if (r_eax07h.EBX & (0x1 << 5))
            m_FeatureFlags |= (uint64_t)FF::AVX2;

        m_OsAvx512State = (((xgetbv_eax >> 5) & 0x07) == 0x07) ? true : false;

        if (m_OsAvx512State)
        {
            // CPUID.(EAX=07H, ECX=00H):EBX.AVX512F[bit 16]
            if (r_eax07h.EBX & (0x1 << 16))
            {
                m_FeatureFlags |= (uint64_t)FF::AVX512F;

                //
                // Decode EBX flags
                //

                // CPUID.(EAX=07H, ECX=00H):EBX.AVX512DQ[bit 17]
                if (r_eax07h.EBX & (0x1 << 17))
                    m_FeatureFlags |= (uint64_t)FF::AVX512DQ;

                // CPUID.(EAX=07H, ECX=00H):EBX.AVX512_IFMA[bit 21]
                if (r_eax07h.EBX & (0x1 << 21))
                    m_FeatureFlags |= (uint64_t)FF::AVX512_IFMA;

                // CPUID.(EAX=07H, ECX=00H):EBX.AVX512PF[bit 26]
                if (r_eax07h.EBX & (0x1 << 26))
                    m_FeatureFlags |= (uint64_t)FF::AVX512PF;

                // CPUID.(EAX=07H, ECX=00H):EBX.AVX512ER[bit 27]
                if (r_eax07h.EBX & (0x1 << 27))
                    m_FeatureFlags |= (uint64_t)FF::AVX512ER;

                // CPUID.(EAX=07H, ECX=00H):EBX.AVX512CD[bit 28]
                if (r_eax07h.EBX & (0x1 << 28))
                    m_FeatureFlags |= (uint64_t)FF::AVX512CD;
```

```
                        // CPUID.(EAX=07H, ECX=00H):EBX.AVX512BW[bit 30]
                        if (r_eax07h.EBX & (0x1 << 30))
                            m_FeatureFlags |= (uint64_t)FF::AVX512BW;

                        // CPUID.(EAX=07H, ECX=00H):EBX.AVX512VL[bit 31]
                        if (r_eax07h.EBX & (0x1 << 31))
                            m_FeatureFlags |= (uint64_t)FF::AVX512VL;

                        //
                        // Decode ECX flags
                        //

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_VBMI[bit 1]
                        if (r_eax07h.ECX & (0x1 << 1))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_VBMI;

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_VBMI2[bit 6]
                        if (r_eax07h.ECX & (0x1 << 6))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_VBMI2;

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_VNNI[bit 11]
                        if (r_eax07h.ECX & (0x1 << 11))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_VNNI;

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_BITALG[bit 12]
                        if (r_eax07h.ECX & (0x1 << 12))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_BITALG;

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_VPOPCNTDQ[bit 14]
                        if (r_eax07h.ECX & (0x1 << 14))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_VPOPCNTDQ;

                        //
                        // Decode EDX flags
                        //

                        // CPUID.(EAX=07H, ECX=00H):EDX.AVX512_4FMAPS[bit 2]
                        if (r_eax07h.EDX & (0x1 << 2))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_4FMAPS;

                        // CPUID.(EAX=07H, ECX=00H):EDX.AVX512_4VNNIW[bit 3]
                        if (r_eax07h.EDX & (0x1 << 3))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_4VNNIW;
                    }
                }
            }
        }
    }
}
```

```
void CpuidInfo::LoadInfo5(void)
{
    if (m_MaxEax < 4)
        return;

    bool done = false;
    uint32_t index = 0;

    while (!done)
    {
        CpuidRegs r;

        Cpuid_(4, index, &r);

        uint32_t cache_type = r.EAX & 0x1f;
        uint32_t cache_level = ((r.EAX >> 5) & 0x3);

        if (cache_type == 0)
            done = true;
        else
        {
            uint32_t ways = ((r.EBX >> 22) & 0x3ff) + 1;
            uint32_t partitions = ((r.EBX >> 12) & 0x3ff) + 1;
            uint32_t line_size = (r.EBX & 0xfff) + 1;
            uint32_t sets = r.ECX + 1;
            uint32_t cache_size = ways * partitions * line_size * sets;

            CacheInfo ci(cache_level, cache_type, cache_size);
            m_CacheInfo.push_back(ci);
            index++;
        }
    }
}

void CpuidInfo::Init(void)
{
    m_MaxEax = 0;
    m_MaxEaxExt = 0;
    m_FeatureFlags = 0;
    m_OsXsave = false;
    m_OsAvxState = false;
    m_OsAvx512State = false;
    m_VendorId[0] = '\0';
    m_ProcessorBrand[0] = '\0';
    m_CacheInfo.clear();
}
```

```
void CpuidInfo::InitProcessorBrand(void)
{
    if (m_MaxEaxExt >= 0x80000004)
    {
        CpuidRegs r2, r3, r4;
        char* p = m_ProcessorBrand;

        Cpuid_(0x80000002, 0, &r2);
        Cpuid_(0x80000003, 0, &r3);
        Cpuid_(0x80000004, 0, &r4);

        *(uint32_t *)(p + 0) = r2.EAX;
        *(uint32_t *)(p + 4) = r2.EBX;
        *(uint32_t *)(p + 8) = r2.ECX;
        *(uint32_t *)(p + 12) = r2.EDX;
        *(uint32_t *)(p + 16) = r3.EAX;
        *(uint32_t *)(p + 20) = r3.EBX;
        *(uint32_t *)(p + 24) = r3.ECX;
        *(uint32_t *)(p + 28) = r3.EDX;
        *(uint32_t *)(p + 32) = r4.EAX;
        *(uint32_t *)(p + 36) = r4.EBX;
        *(uint32_t *)(p + 40) = r4.ECX;
        *(uint32_t *)(p + 44) = r4.EDX;

        m_ProcessorBrand[sizeof(m_ProcessorBrand) - 1] = '\0';
    }
    else
        strcpy_s(m_ProcessorBrand, "Unknown");
}
```

Before examining the source code, it will be helpful to have a basic understanding of the cupid instruction and how it works. Prior to using cpuid, a function must load register EAX with a "leaf" value that specifies what information the cpuid instruction should return. A second or "sub-leaf" value may also be required in register ECX. The cpuid instruction returns its results in registers EAX, EBX, ECX, and EDX. The calling function must then decode the values in these registers to ascertain processor support for specific features. As you will soon see, it is often necessary for a program to employ the cupid instruction multiple times. Most application programs typically use cupid during initialization and save the results for later use. The reason for this is that cupid is a serializing instruction, which means that it forces the processor to finish executing all previously fetched instructions and perform any pending memory writes before fetching the next instruction. In other words, the cupid instruction takes a long time to complete its execution.

Listing 16-1 begins with the header file CpuidInfo.h. Near the top of this file is a structure named CpuidRegs, which is used to save the results returned by cupid. Following CpuidRegs is a C++ class named CpuidInfo. This class contains the code and data that's associated with cpuid instruction use. The public portion of CpuidInfo includes a subclass named CacheInfo. This class is employed to report information about a processor's memory caches. Class CpuidInfo also includes an enum named FF. An application program can use this enum as an argument value with the member function CpuidInfo::GetFF to determine if the host processor supports a specific instruction set. You'll see how this works later in this section. Toward the bottom of header file CpuidInfo.h are two declaration statements for the assembly language functions Cpuid_ and Xgetbv_. These functions execute the cpuid and xgetbv (Get Value of Extended Control Register) instructions, respectively.

Following CpuidInfo.h in Listing 16-1 is the source code file CpuidInfo_.asm. This file contains the assembly language functions Cpuid_ and Xgetbv_, which are simple wrapper functions for the x86 instructions cupid and xgetbv. The function Cpuid_ begins its execution by saving register RBX on the stack. It then loads argument values r_eax and r_ecx into registers EAX and ECX. The actual cupid instruction follows the loading of registers EAX and ECX. Following the execution of cpuid, the results in registers EAX, EBX, ECX, and EDX are saved to the specified CpuidRegs structure. The assembly language function Xgetbv_ executes the xgetbv instruction. This instruction loads the contents of the extended processor control register that's specified by ECX into register pair EDX:EAX. The xgetbv instruction allows an application program to determine if the host operating system supports AVX, AVX2, or AVX-512, as explained later in this section.

The next file in Listing 16-1 is Ch16_01.cpp. The function main contains code illustrates how to use the C++ class CpuidInfo. The statement ci.LoadInfo() invokes the member function CpuidInfo::LoadInfo, which generates multiple executions of cpuid to obtain information about the processor. Note that CpuidInfo::LoadInfo is only called once. The function DisplayCacheInfo streams information about the processor's memory caches to cout. This function invokes CpuidInfo::GetCacheInfo to report cache information that was obtained during execution of CpuidInfo::LoadInfo. The function DisplayFeatureFlags shows information about some of the instruction set extensions that the processor supports. Each cout statement in this function uses CpuidInfo::GetFF with a different CpuidInfo::FF value. The member function CpuidInfo::GetFF returns a single bool value that indicates whether the processor supports instruction set extension that's specified by its argument value. Like the cache data, the processor instruction set extension information was obtained and saved during the call to CpuidInfo::LoadInfo. Note that CpuidInfo is structured to allow an application program to make multiple CpuidInfo::GetFF calls without triggering additional executions of cupid.

Following the file Ch16_01.cpp in Listing 16-1 is the source code for CpuidInfo.cpp, which contains the non-trivial member functions for class CpuidInfo. The member function CpuidInfo::LoadInfo that was discussed earlier invokes six private member functions that perform a multitude of cupid queries. The first of these functions, CpuidInfo::LoadInfo0, begins its execution by calling CpuidInfo::Init to carry out the requisite initializations. It then invokes the assembly language function Cpuid_ to obtain the maximum cpuid leaf value that's supported by the processor and the processor vendor ID string. Another call to Cpuid_ is then used to obtain the maximum leaf value for extended cupid information. This is followed by a call to CpuidInfo::InitProcessorBrand, which uses several Cpuid_ calls to query and save the processor brand string. The source code for this function is located toward the end of the file CpuidInfo.cpp.

The member functions CpuidInfo::LoadInfo1, CpuidInfo::LoadInfo2, and CpuidInfo::LoadInfo3 also exploit Cpuid_ to ascertain processor support for a variety of instruction set extensions. The code that's contained in these member functions is mostly brute-force decoding of the various Cpuid_ results. The AMD and Intel programming reference manuals contain additional information about the cpuid feature flag bits that are used to indicate processor support for a specific instruction set extension. The private member function CpuidInfo::LoadInfo4 contains the code that checks for AVX, AVX2, and AVX-512. This member function warrants closer examination.

An application program can use the computational resources of x86-AVX only if it's supported by *both* the processor and its host operating system. The Xgetbv_ function can be employed to determine host operating system support. Before using Xgetbv_, the cpuid flag OSXSAVE must be tested to ensure that it's safe for an application program to use the xgetbv instruction. If OSXSAVE is set to true, the function CpuidInfo::LoadInfo4 invokes Xgetbv_ to obtain information regarding OS support for x86-AVX state information (i.e., whether the OS properly preserves the XMM, YMM, and ZMM registers during a task switch). When using the Xgetbv_ function, the processor will generate an exception if the extended control register number is invalid or if the processor's OSXSAVE flag is set to false. This explains why the software flag m_OsXsave is checked in prior to calling Xgetbv_. If the host operating system supports x86-AVX state information, the function CpuidInfo::LoadInfo4 proceeds to decode the cpuid feature flags related to AVX and AVX2. Note that the feature flags FMA and F16C are also tested here. The remaining code in CpuidInfo::LoadInfo4 decodes the cupid feature flags that signify support for the various AVX-512 instruction set extensions.

The final private member function that's called by `CpuidInfo::LoadInfo` is named `CpuidInfo::LoadInfo5`. This member function uses `cpuid` and the class `CpuidInfo::CacheInfo` to save type and size information about the processor's memory caches. The ancillary code for class `CpuidInfo::CacheInfo` is not shown in Listing 16-1 but included with the chapter download package. Here is the output for source code example `Ch16_01`:

```
GenuineIntel
Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz
Cache L1: 32KB Data
Cache L1: 32KB Instruction
Cache L2: 1MB Unified
Cache L3: 13MB Unified
----- CPUID Feature Flags -----
ADX:         1
AVX:         1
AVX2:        1
AVX512F:     1
AVX512BW:    1
AVX512CD:    1
AVX512DQ:    1
AVX512ER:    0
AVX512PF:    0
AVX512VL:    1
AVX512_IFMA: 0
AVX512_VBMI: 0
BMI1:        1
BMI2:        1
F16C:        1
FMA:         1
LZCNT:       1
POPCNT:      1
```

Table 16-1 shows a summary of `cpuid` information for several Intel processors. Before moving on to the next source code example, it should be noted that when using the `cpuid` instruction to determine processor support for the various AVX-512 instruction set extensions, it is often necessary for an application program to test multiple feature flags. For example, an application program must verify that the AVX512F, AVX512DQ, and AVX512VL feature flags are all set before using any AVX512DQ instructions with 256-bit or 128-bit wide operands. The *Intel 64 and IA-32 Architectures Software Developer's Manual (Volume 1)* contains additional information regarding `cpuid` instruction use and feature flag testing.

***Table 16-1.** Summary of Information from `cpuid` Instruction for Select Intel Processors*

| CPUID Feature | i3-2310m | i7-4790s | i9-7900x | i7-8700k |
|---|---|---|---|---|
| L1 Data (KB, per core) | 32 | 32 | 32 | 32 |
| L1 Instruction (KB, per core) | 32 | 32 | 32 | 32 |
| L2 Unified (KB, per core) | 256 | 256 | 1024 | 256 |
| L3 Unified (MB) | 3 | 8 | 13 | 12 |
| ADX | 0 | 0 | 1 | 1 |
| AVX | 1 | 1 | 1 | 1 |
| AVX2 | 0 | 1 | 1 | 1 |
| AXV512F | 0 | 0 | 1 | 0 |
| AVX512BW | 0 | 0 | 1 | 0 |
| AVX512CD | 0 | 0 | 1 | 0 |
| AVX512DQ | 0 | 0 | 1 | 0 |
| AVX512ER | 0 | 0 | 0 | 0 |
| AVX512PF | 0 | 0 | 0 | 0 |
| AVX512VL | 0 | 0 | 1 | 0 |
| AVX512_IFMA | 0 | 0 | 0 | 0 |
| AVX512_VBMI | 0 | 0 | 0 | 0 |
| BMI1 | 0 | 1 | 1 | 1 |
| BMI2 | 0 | 1 | 1 | 1 |
| F16C | 0 | 1 | 1 | 1 |
| FMA | 0 | 1 | 1 | 1 |
| LZCNT | 0 | 1 | 1 | 1 |
| POPCNT | 1 | 1 | 1 | 1 |

# Non-Temporal Memory Stores

From the perspective of a memory cache, data can be classified as either temporal or non-temporal. Temporal data is any value that is accessed more than once within a short period of time. Examples of temporal data include the elements of an array or data structure that are referenced multiple times during execution of a program loop. It also includes the instruction bytes of a program. Non-temporal data is any value that is accessed once and not immediately reused. The destination arrays of many SIMD processing algorithms often contain non-temporal data. The differentiation between temporal and non-temporal data is important since processor performance often degrades if its memory caches contain excessive amounts of non-temporal data. This condition is commonly called *cache pollution*. Ideally, a processor's memory caches contain only temporal data since it makes little sense to cache items that are only used once.

Listing 16-2 shows the source code for example `Ch16_02`. This example illustrates the use of the non-temporal store instruction `vmovntps`. It also compares the performance of this instruction to the standard `vmovaps` instruction.

***Listing 16-2.*** Example Ch16_02

```
//------------------------------------------------
//               Ch16_02.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <string>
#include <random>
#include "Ch16_02.h"
#include "AlignedMem.h"

using namespace std;

void Init(float* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 1000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (float)ui_dist(rng);
}

bool CalcResultCpp(float* c, const float* a, const float* b, size_t n)
{
    size_t align = 32;

    if ((n == 0) || ((n & 0x0f) != 0))
        return false;

    if (!AlignedMem::IsAligned(a, align))
        return false;
    if (!AlignedMem::IsAligned(b, align))
        return false;
    if (!AlignedMem::IsAligned(b, align))
        return false;

    for (size_t i = 0; i < n; i++)
        c[i] = sqrt(a[i] * a[i] + b[i] * b[i]);

    return true;
}

void CompareResults(const float* c1, const float* c2a, const float*c2b, size_t n)
{
    bool compare_ok = true;
    const float epsilon = 1.0e-9f;

    cout << fixed << setprecision(4);
```

```
    for (size_t i = 0; i < n && compare_ok; i++)
    {
        bool b1 = fabs(c1[i] - c2a[i]) > epsilon;
        bool b2 = fabs(c1[i] - c2b[i]) > epsilon;

        cout << setw(2) << i << " - ";
        cout << setw(10) << c1[i] << ' ';
        cout << setw(10) << c2a[i] << ' ';
        cout << setw(10) << c2b[i] << '\n';

        if (b1 || b2)
            compare_ok = false;
    }

    if (compare_ok)
        cout << "Array compare OK\n";
    else
        cout << "Array compare FAILED\n";
}

void NonTemporalStore(void)
{
    const size_t n = 16;
    const size_t align = 32;

    AlignedArray<float> a_aa(n, align);
    AlignedArray<float> b_aa(n, align);
    AlignedArray<float> c1_aa(n, align);
    AlignedArray<float> c2a_aa(n, align);
    AlignedArray<float> c2b_aa(n, align);
    float* a = a_aa.Data();
    float* b = b_aa.Data();
    float* c1 = c1_aa.Data();
    float* c2a = c2a_aa.Data();
    float* c2b = c2b_aa.Data();

    Init(a, n, 67);
    Init(b, n, 79);

    bool rc1 = CalcResultCpp(c1, a, b, n);
    bool rc2 = CalcResultA_(c2a, a, b, n);
    bool rc3 = CalcResultB_(c2b, a, b, n);

    if (!rc1 || !rc2 || !rc3)
    {
        cout << "Invalid return code\n";
        cout << "rc1 = " << boolalpha << rc1 << '\n';
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        cout << "rc3 = " << boolalpha << rc3 << '\n';
        return;
    }
```

```
    cout << "Results for NonTemporalStore\n";
    CompareResults(c1, c2a, c2b, n);
}

int main()
{
    NonTemporalStore();
    NonTemporalStore_BM();
    return 0;
}

;-------------------------------------------------
;               Ch16_02.asm
;-------------------------------------------------

; _CalcResult Macro
;
; The following macro contains a simple calculating loop that is used
; to compare performance of the vmovaps and vmovntps instructions.

_CalcResult macro MovInstr

; Load and validate arguments
        xor eax,eax                         ;set error code
        test r9,r9
        jz Done                             ;jump if n <= 0
        test r9,0fh
        jnz Done                            ;jump if (n % 16) != 0

        test rcx,1fh
        jnz Done                            ;jump if c is not aligned
        test rdx,1fh
        jnz Done                            ;jump if a is not aligned
        test r8,1fh
        jnz Done                            ;jump if b is not aligned

; Calculate c[i] = sqrt(a[i] * a[i] + b[i] * b[i])
        align 16
@@:     vmovaps ymm0,ymmword ptr [rdx+rax]      ;ymm0 = a[i+7]:a[i]
        vmovaps ymm1,ymmword ptr [r8+rax]       ;ymm1 = b[i+7]:b[i]
        vmulps ymm2,ymm0,ymm0                   ;ymm2 = a[i] * a[i]
        vmulps ymm3,ymm1,ymm1                   ;ymm3 = b[i] * b[i]
        vaddps ymm4,ymm2,ymm3                   ;ymm4 = sum
        vsqrtps ymm5,ymm4                       ;ymm5 = final result
        MovInstr ymmword ptr [rcx+rax],ymm5     ;save final values to c
```

```
        vmovaps ymm0,ymmword ptr [rdx+rax+32]   ;ymm0 = a[i+15]:a[i+8]
        vmovaps ymm1,ymmword ptr [r8+rax+32]    ;ymm1 = b[i+15]:b[i+8]
        vmulps ymm2,ymm0,ymm0                   ;ymm2 = a[i] * a[i]
        vmulps ymm3,ymm1,ymm1                   ;ymm3 = b[i] * b[i]
        vaddps ymm4,ymm2,ymm3                   ;ymm4 = sum
        vsqrtps ymm5,ymm4                       ;ymm5 = final result
        MovInstr ymmword ptr [rcx+rax+32],ymm5  ;save final values to c

        add rax,64                              ;update offset
        sub r9,16                               ;update counter
        jnz @B

        mov eax,1                              ;set success return code

Done:   vzeroupper
        ret
        endm

; extern bool CalcResultA_(float* c, const float* a, const float* b, size_t n)

        .code
CalcResultA_ proc
        _CalcResult vmovaps
CalcResultA_ endp

; extern bool CalcResultB_(float* c, const float* a, const float* b, int n)

CalcResultB_ proc
        _CalcResult vmovntps
CalcResultB_ endp
        end
```

Near the top of Listing 16-2 is the C++ function CalcResultCpp. This function performs a simple arithmetic calculation using the elements of two single-precision floating-point source arrays. It then saves the result to a destination array. The next C++ function in Listing 16-2 is named CompareResults. This function verifies equivalence between the C++ and assembly language output arrays. The function NonTemporalStore allocates and initializes the test arrays. It then invokes the C++ and assembly language calculating functions. The output arrays of the three calculating functions are then compared for any discrepancies.

The assembly language code in Listing 16-2 begin with the definition of a macro named _CalcResult. This macro generates AVX instructions that perform the exact same calculation as the C++ function CalcResultCpp. The macro _CalcResult is used in the assembly language functions CalcResultA_ and CalcResultB_. Note that CalcResultA_ supplies the instruction vmovaps for the macro parameter MovInstr while CalcResultB_ provides vmovntaps. This means that the code executed by functions CalcResultA_ and CalcResultB_ is identical, except for the move instruction that saves results to the destination array. Here is the output for source code example Ch16_02;

```
Results for NonTemporalStore

 0 -   240.8319   240.8319   240.8319
 1 -   747.1814   747.1814   747.1814
 2 -   285.1561   285.1561   285.1561
 3 -   862.3062   862.3062   862.3062
 4 -   604.8810   604.8810   604.8810
 5 -  1102.4504  1102.4504  1102.4504
 6 -   347.1441   347.1441   347.1441
 7 -   471.8315   471.8315   471.8315
 8 -   890.6739   890.6739   890.6739
 9 -   729.0878   729.0878   729.0878
10 -   458.3536   458.3536   458.3536
11 -   639.8031   639.8031   639.8031
12 -  1053.1063  1053.1063  1053.1063
13 -  1016.0079  1016.0079  1016.0079
14 -   610.4507   610.4507   610.4507
15 -  1161.7935  1161.7935  1161.7935
Array compare OK

Running benchmark function NonTemporalStore_BM - please wait
Benchmark times save to file Ch16_02_NonTemporalStore_BM_CHROMIUM.csv
```

Table 16-2 shows benchmark timing measurements for source code example `Ch16_02` using several different Intel processors. In this example, using a `vmovntps` instruction instead of a `vmovaps` instruction yielded notable performance improvements on all three computers. It should be noted that the x86's non-temporal move instructions only provide a hint to the processor regarding memory use. They do not guarantee improved performance in all cases. Any performance gains are determined by the specific memory access pattern and the processor's underlying microarchitecture.

***Table 16-2.*** *Mean Execution Times (Microseconds) for Functions `CalcResultCpp`, `CalcResultA_`, and `CalcResultB_` (n = 2,000,000)*

| CPU | CalcResultCpp | CalcResultA_(uses `vmovaps`) | CalcResultB_(uses `vmovntps`) |
|---|---|---|---|
| i7-4790s | 1553 | 1554 | 1242 |
| i9-7900x | 1173 | 1139 | 934 |
| i7-8700k | 847 | 801 | 590 |

## Data Prefetch

An application program can also use the `prefetch` (Prefetch Data Into Caches) instruction to improve the performance of certain algorithms. This instruction facilitates pre-loading of expected-use data into the processor's cache hierarchy. There are two basic forms of the prefetch instruction. The first form, `prefetcht[0|1|2]`, pre-loads temporal data into a specific cache level. The second form, `prefetchnta`, pre-loads non-temporal data while minimizing cache pollution. Both forms of the `prefetch` instruction provide hints to the processor about the data that a program expects to use; a processor may choose to perform the prefetch operation or ignore the hint.

The prefetch instructions are suitable for use with a variety of data structures, including large arrays and linked lists. A linked list is sequentially-ordered collection of nodes. Each node includes a data section and one or more pointers (or links) to its adjacent nodes. Figure 16-1 illustrates a simple linked list. Linked lists are useful since their size can grow or shrink (i.e., nodes can be added or deleted) depending on data storage requirements. One drawback of a linked list is that the nodes are usually not stored in a contiguously-allocated block of memory. This tends to increase access times when traversing the nodes a linked list.

Source code example `Ch16_03` illustrates how to perform linked list traversals both with and without the `prefetchnta` instruction. Listings 16-3 shows the C++ and assembly language source code for this example.

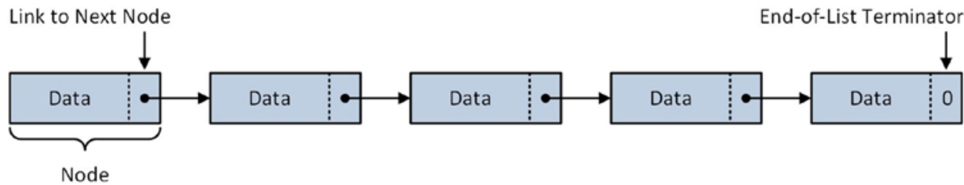



***Figure 16-1.*** *Simple linked list*

***Listing 16-3.*** Example Ch16_03

```
//------------------------------------------------
//               Ch16_03.h
//------------------------------------------------

#pragma once
#include <cstdint>

// This structure must match the corresponding structure definition in Ch16_03.asmh
struct LlNode
{
    double ValA[4];
    double ValB[4];
    double ValC[4];
    double ValD[4];
    uint8_t FreeSpace[376];
    LlNode* Link;
};

// Ch16_03_Misc.cpp
extern bool LlCompare(int num_nodes, LlNode* l1, LlNode* l2, LlNode* l3, int* node_fail);
extern LlNode* LlCreate(int num_nodes);
extern void LlDelete(LlNode* p);
extern bool LlPrint(LlNode* p, const char* fn, const char* msg, bool append);
extern void LlTraverse(LlNode* p);

// Ch16_03_.asm
extern "C" void LlTraverseA_(LlNode* p);
extern "C" void LlTraverseB_(LlNode* p);
```

```
// Ch16_03_BM.cpp
extern void LinkedListPrefetch_BM(void);

//------------------------------------------------
//               Ch16_03.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cmath>
#include <random>
#include "Ch16_03.h"
#include "AlignedMem.h"

using namespace std;

void LinkedListPrefetch(void)
{
    const int num_nodes = 8;
    LlNode* list1 = LlCreate(num_nodes);
    LlNode* list2a = LlCreate(num_nodes);
    LlNode* list2b = LlCreate(num_nodes);

    LlTraverse(list1);
    LlTraverseA_(list2a);
    LlTraverseB_(list2b);

    int node_fail;
    const char* fn = "Ch16_03_LinkedListPrefetchResults.txt";

    cout << "Results for LinkedListPrefetch\n";

    if (LlCompare(num_nodes, list1, list2a, list2b, &node_fail))
        cout << "Linked list compare OK\n";
    else
        cout << "Linked list compare FAILED - node_fail = " << node_fail << '\n';

    LlPrint(list1, fn,  "----- list1 -----",  0);
    LlPrint(list2a, fn, "----- list2a -----", 1);
    LlPrint(list2b, fn, "----- list2b -----", 1);

    cout << "Linked list results saved to file " << fn << '\n';

    LlDelete(list1);
    LlDelete(list2a);
    LlDelete(list2b);
}
```

```
int main()
{
    LinkedListPrefetch();
    LinkedListPrefetch_BM();
    return 0;
}

//------------------------------------------------
//               Ch16_03_Misc.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cmath>
#include <random>
#include "Ch16_03.h"
#include "AlignedMem.h"

using namespace std;

bool LlCompare(int num_nodes, LlNode* l1, LlNode* l2, LlNode* l3, int* node_fail)
{
    const double epsilon = 1.0e-9;

    for (int i = 0; i < num_nodes; i++)
    {
        *node_fail = i;

        if ((l1 == nullptr) || (l2 == nullptr) || (l3 == nullptr))
            return false;

        for (int j = 0; j < 4; j++)
        {
            bool b12_c = fabs(l1->ValC[j] - l2->ValC[j]) > epsilon;
            bool b13_c = fabs(l1->ValC[j] - l3->ValC[j]) > epsilon;
            if (b12_c || b13_c)
                return false;

            bool b12_d = fabs(l1->ValD[j] - l2->ValD[j]) > epsilon;
            bool b13_d = fabs(l1->ValD[j] - l3->ValD[j]) > epsilon;
            if (b12_d || b13_d)
                return false;
        }

        l1 = l1->Link;
        l2 = l2->Link;
        l3 = l3->Link;
    }
```

```
    *node_fail = -2;
    if ((l1 != nullptr) || (l2 != nullptr) || (l3 != nullptr))
        return false;

    *node_fail = -1;
    return true;
}

LlNode* LlCreate(int num_nodes)
{
    const size_t align = 64;
    const unsigned int seed = 83;
    LlNode* first = nullptr;
    LlNode* last = nullptr;
    uniform_int_distribution<> ui_dist {1, 500};
    default_random_engine rng {seed};

    for (int i = 0; i < num_nodes; i++)
    {
        LlNode* p = (LlNode*)AlignedMem::Allocate(sizeof(LlNode), align);
        p->Link = nullptr;

        if (i == 0)
            first = last = p;
        else
        {
            last->Link = p;
            last = p;
        }

        for (int j = 0; j < 4; j++)
        {
            p->ValA[j] = (double)ui_dist(rng);
            p->ValB[j] = (double)ui_dist(rng);
            p->ValC[j] = 0;
            p->ValD[j] = 0;
        }
    }

    return first;
}

void LlDelete(LlNode* p)
{
    while (p != nullptr)
    {
        LlNode* q = p->Link;

        AlignedMem::Release(p);
        p = q;
    }
}
```

```
bool LlPrint(LlNode* p, const char* fn, const char* msg, bool append)
{
    FILE* fp;
    const char* mode = (append) ? "at" : "wt";

    if (fopen_s(&fp, fn, mode) != 0)
        return false;

    int i = 0;
    const char* fs = "%14.4lf %14.4lf %14.4lf %14.4lf\n";

    if (msg != nullptr)
        fprintf(fp, "\n%s\n", msg);

    while (p != nullptr)
    {
        fprintf(fp, "\nLlNode %d [0x%p]\n", i, p);
        fprintf(fp, "  ValA: ");
        fprintf(fp, fs, p->ValA[0], p->ValA[1], p->ValA[2], p->ValA[3]);

        fprintf(fp, "  ValB: ");
        fprintf(fp, fs, p->ValB[0], p->ValB[1], p->ValB[2], p->ValB[3]);

        fprintf(fp, "  ValC: ");
        fprintf(fp, fs, p->ValC[0], p->ValC[1], p->ValC[2], p->ValC[3]);

        fprintf(fp, "  ValD: ");
        fprintf(fp, fs, p->ValD[0], p->ValD[1], p->ValD[2], p->ValD[3]);

        i++;
        p = p->Link;
    }

    fclose(fp);
    return true;
}

void LlTraverse(LlNode* p)
{
    while (p != nullptr)
    {
        for (int i = 0; i < 4; i++)
        {
            p->ValC[i] = sqrt(p->ValA[i] * p->ValA[i] + p->ValB[i] * p->ValB[i]);
            p->ValD[i] = sqrt(p->ValA[i] / p->ValB[i] + p->ValB[i] / p->ValA[i]);
        }
        p = p->Link;
    }
}
```

```
;-------------------------------------------------
;               Ch16_03_.asmh
;-------------------------------------------------

; This structure must match the corresponding structure definition in Ch16_03.h

LlNode      struct
ValA        real8 4 dup(?)
ValB        real8 4 dup(?)
ValC        real8 4 dup(?)
ValD        real8 4 dup(?)
FreeSpace   byte 376 dup(?)
Link        qword ?
LlNode      ends

;-------------------------------------------------
;               Ch16_03_.asm
;-------------------------------------------------

        include <Ch16_03_.asmh>

; Macro _LlTraverse
;
; The following macro generates linked list traversal code using the
; prefetchnta instruction if UsePrefetch is equal to 'Y'.

_LlTraverse macro UsePrefetch
        mov rax,rcx                                 ;rax = ptr to 1st node
        test rax,rax
        jz Done                                     ;jump if empty list

        align 16
@@::    mov rcx,[rax+LlNode.Link]                   ;rcx = next node
        vmovapd ymm0,ymmword ptr [rax+LlNode.ValA]  ;ymm0 = ValA
        vmovapd ymm1,ymmword ptr [rax+LLNode.ValB]  ;ymm1 = ValB

IFIDNI <UsePrefetch>,<Y>
        mov rdx,rcx
        test rdx,rdx                        ;is there another node?
        cmovz rdx,rax                       ;avoid prefetch of nullptr
        prefetchnta [rdx]                   ;prefetch start of next node
ENDIF

; Calculate ValC[i] = sqrt(ValA[i] * ValA[i] + ValB[i] * ValB[i])
        vmulpd ymm2,ymm0,ymm0               ;ymm2 = ValA * ValA
        vmulpd ymm3,ymm1,ymm1               ;ymm3 = ValB * ValB
        vaddpd ymm4,ymm2,ymm3               ;ymm4 = sums
        vsqrtpd ymm5,ymm4                   ;ymm5 = square roots

        vmovntpd ymmword ptr [rax+LlNode.ValC],ymm5 ;save result
```

```
; Calculate ValD[i] = sqrt(ValA[i] / ValB[i] + ValB[i] / ValA[i]);
        vdivpd ymm2,ymm0,ymm1               ;ymm2 = ValA / ValB
        vdivpd ymm3,ymm1,ymm0               ;ymm3 = ValB / ValA
        vaddpd ymm4,ymm2,ymm3               ;ymm4 = sums
        vsqrtpd ymm5,ymm4                   ;ymm5 = square roots

        vmovntpd ymmword ptr [rax+LlNode.ValD],ymm5 ;save result

        mov rax,rcx                         ;rax = ptr to next node
        test rax,rax
        jnz @B

Done:   vzeroupper
        ret
        endm

; extern "C" void LlTraverseA_(LlNode* first);

        .code
LlTraverseA_ proc
        _LlTraverse n
LlTraverseA_ endp

; extern "C" void LlTraverseB_(LlNode* first);

LlTraverseB_ proc
        _LlTraverse y
LlTraverseB_ endp
        end
```

Listing 16-3 begins with the header file `Ch16_03.h`. The declaration of structure `LlNode` is located near the top of this file. The C++ code uses this structure to construct linked lists of test data. Structure members `ValA` through `ValD` hold the data values that are manipulated by the linked list traversal functions. The member `FreeSpace` is included to increase the size of `LlNode` for demonstration purposes since prefetching works best with larger data structures. A real-world implementation of `LlNode` could use this space for additional data items. The final member of `LlNode` is a pointer named `Link`, which points to the next `LlNode` structure. The assembly language counterpart of `LlNode` is declared in the file `Ch16_03_.asmh`.

The base function for source code example `Ch16_03` is named `LinkedListPrefetch` and can be found in source code file `Ch16_03.cpp`. This function builds several test linked lists, invokes the C++ and assembly language traversal functions, and validates the results. The source code file `Ch16_03_misc.cpp` contains a set of miscellaneous functions that implement basic linked list processing operations. The function `LlCompare` compares the data nodes of its argument linked lists for equivalence. The functions `LlCreate` and `LlDelete` perform linked list allocation and deletion. `LlPrint` dumps the data contents of a linked list to a file. Finally, `LlTraverse` traverses a linked list and performs a simulated calculation using `LlNode` data elements `ValA`, `ValB`, `ValC`, and `ValD`.

Toward the top of the file `Ch16_03_.asm` is a macro named `_LlTraverse`. This macro generates code that performs the same linked list traversal and simulated calculation as the C++ function `LlTraverse`. The macro `_LlTraverse` requires one parameter that enables or disables data prefetching. If prefetching is enabled, the macro generates a short block code that includes the instruction `prefetchnta [rdx]`. This instruction directs the processor to prefetch the non-temporal data pointed to by register RDX. In this example, RDX points to the next `LlNode` in the linked list. The actual number of bytes fetched by this

instruction varies depending on the underlying microarchitecture; Intel processors fetch a minimum of 32 bytes. It is extremely important to note that the `prefetchnta [rdx]` is positioned *before* the floating-point calculating instructions. Doing this gives the processor an opportunity to fetch the data for the next node while at the same time carrying out the arithmetic calculations for the current node. Also note that prior to execution of the `prefetchnta [rdx]` instruction, RDX is tested to avoid using `prefetchnta` with a `nullptr` (or zero) memory address since this degrades processor performance. This is important since `nullptr` is used as the end-of-list terminator value. The assembly language functions `LlTraverseA_` and `LlTraverseB_` use macro `_LlTraverse` to perform linked list traversals both without and with prefetching, respectively. Here is the output for source code example `Ch16_03`.

```
Results for LinkedListPrefetch
Linked list compare OK
Linked list results saved to file Ch16_03_LinkedListPrefetchResults.txt

Running benchmark function LinkedListPrefetch_BM - please wait
Benchmark times save to file Ch16_03_LinkedListPrefetch_BM_CHROMIUM.csv
```

Table 16-3 shows benchmark timing measurements for several Intel processors. It is important to keep in mind that any performance benefits provided by the prefetch instructions are highly dependent on current processor load, data access patterns, and the underlying microarchitecture. According to the *Intel 64 and IA-32 Architectures Optimization Reference Manual*, the data prefetch instructions are "implementation specific." This means that to maximize prefetch performance, an algorithm must be "tuned to each implementation" or microarchitecture. You are encouraged to consult the aforementioned reference manual for additional information regarding use of the x86's data prefetch instructions.

***Table 16-3.*** *Mean Execution Times (Microseconds) for Linked List Traversal Functions (num_nodes = 50,000)*

| CPU | LlTraverse(C++) | LlTraverseA_(without `prefetchnta`) | LlTraverseB_(with `prefetchnta`) |
|---|---|---|---|
| i7-4790s | 5685 | 3093 | 2680 |
| i9-7900x | 5885 | 3064 | 2842 |
| i7-8700k | 5031 | 2384 | 2319 |

# Multiple Threads

All the source code examples presented in this book thus far have shared one common characteristic: they all contain single-threaded code. The mere fact that you are reading this book probably means that you already know that most modern software applications utilize a least a few threads to better exploit the multiple cores of modern processors. For example, many high-performance computing applications frequently perform arithmetic calculations using large data arrays that contain millions of floating-point elements. One strategy that's often employed to accelerate the performance of these types of calculations is to distribute the array elements across multiple threads and have each thread carry out a subset of the required calculations. The next source code example demonstrates how to perform an arithmetic calculation using large floating-point arrays and multiple threads. Listing 16-4 shows the source code for example `Ch16_04`.

---

■ **Caution** A processor can become extremely hot while executing multithreaded code that makes extensive use of x86-AVX instructions. Before running the code of example `Ch16_04`, you should verify that the processor in your computer has an adequate cooling system.

---

***Listing 16-4.*** Example Ch16_04

```
//------------------------------------------------
//               Ch16_04.h
//------------------------------------------------

#pragma once
#include <vector>

struct CalcInfo
{
    double* m_X1;
    double* m_X2;
    double* m_Y1;
    double* m_Y2;
    double* m_Z1;
    double* m_Z2;
    double* m_Result;
    size_t m_Index0;
    size_t m_Index1;
    int m_Status;
};

struct CoutInfo
{
    bool m_ThreadMsgEnable;
    size_t m_Iteration;
    size_t m_NumElements;
    size_t m_ThreadId;
    size_t m_NumThreads;
};

// Ch16_04_Misc.cpp
extern size_t CompareResults(const double* a, const double* b, size_t n);
extern void DisplayThreadMsg(const CalcInfo* ci, const CoutInfo* cout_info, const char*
msg);
extern void Init(double* a1, double* a2, size_t n, unsigned int seed);
std::vector<size_t> GetNumElementsVec(size_t* num_elements_max);
std::vector<size_t> GetNumThreadsVec(void);

// Ch16_04_WinApi.cpp
extern bool GetAvailableMemory(size_t* mem_size);

// Ch16_04_.asm
extern "C" void CalcResult_(CalcInfo* ci);
```

```
// Miscellaneous constants
const size_t c_ElementSize = sizeof(double);

const size_t c_NumArrays = 8;   // Total number of allocated arrays
const size_t c_Align = 32;      // Alignment boundary (update Ch16_04_.asm if changed)
const size_t c_BlockSize = 8;   // Elements per iteration (update Ch16_04_.asm if changed)

//------------------------------------------------
//               Ch16_04_Misc.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <random>
#include <memory.h>
#include <cmath>
#include <mutex>
#include <vector>
#include <algorithm>
#include "Ch16_04.h"

using namespace std;

void Init(double* a1, double* a2, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 2000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
    {
        a1[i] = (double)ui_dist(rng);
        a2[i] = (double)ui_dist(rng);
    }
}

size_t CompareResults(const double* a, const double* b, size_t n)
{
    if (memcmp(a, b, n * sizeof(double)) == 0)
        return n;

    const double epsilon = 1.0e-15;

    for (size_t i = 0; i < n; i++)
    {
        if (fabs(a[i] - b[i]) > epsilon)
            return i;
    }

    return n;
}
```

```
void DisplayThreadMsg(const CalcInfo* ci, const CoutInfo* cout_info, const char* msg)
{
    static mutex mutex_cout;
    static const char nl = '\n';

    mutex_cout.lock();
    cout << nl << msg << nl;
    cout << "  m_Iteration:   " << cout_info->m_Iteration << nl;
    cout << "  m_NumElements: " << cout_info->m_NumElements << nl;
    cout << "  m_ThreadId:    " << cout_info->m_ThreadId << nl;
    cout << "  m_NumThreads:  " << cout_info->m_NumThreads << nl;
    cout << "  m_Index0:      " << ci->m_Index0 << nl;
    cout << "  m_Index1:      " << ci->m_Index1 << nl;
    mutex_cout.unlock();
}

vector<size_t> GetNumElementsVec(size_t* num_elements_max)
{
//    vector<size_t> ne_vec {64, 192, 384, 512};    // Requires 32GB + extra
      vector<size_t> ne_vec {64, 128, 192, 256};    // Requires 16GB + extra
//    vector<size_t> ne_vec {64, 96, 128, 160};     // Requires 10GB + extra

    size_t mem_size_extra_gb = 2;       // Used to avoid allocating all available mem

    size_t ne_max = *std::max_element(ne_vec.begin(), ne_vec.end());

    if ((ne_max % c_BlockSize) != 0)
        throw runtime_error("ne_max must be an integer multiple of c_BlockSize");

    size_t mem_size;

    if (!GetAvailableMemory(&mem_size))
        throw runtime_error ("GetAvailableMemory failed");

    size_t mem_size_gb = mem_size / (1024 * 1024 * 1024);
    size_t mem_size_min = ne_max * 1024 * 1024 * c_ElementSize * c_NumArrays;
    size_t mem_size_min_gb = mem_size_min / (1024 * 1024 * 1024);

    if (mem_size_gb < mem_size_min_gb + mem_size_extra_gb)
        throw runtime_error ("Not enough available memory");

    *num_elements_max = ne_max * 1024 * 1024;
    return ne_vec;
}

vector<size_t> GetNumThreadsVec(void)
{
    vector<size_t> num_threads_vec {1, 2, 4, 6, 8};

    return num_threads_vec;
}
```

```
//------------------------------------------------
//               Ch16_04.cpp
//------------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <sstream>
#include <stdexcept>
#include <thread>
#include <vector>
#include "Ch16_04.h"
#include "AlignedMem.h"
#include "BmThreadTimer.h"

using namespace std;

// Control flag for streaming thread status information to cout.
const bool c_ThreadMsgEnable = false;

void CalcResultCpp(CalcInfo* ci)
{
    size_t al = c_Align;
    size_t i0 = ci->m_Index0;
    size_t i1 = ci->m_Index1;
    size_t num_elements = i1 - i0 + 1;

    ci->m_Status = 0;

    if (num_elements == 0 || (num_elements % c_BlockSize) != 0)
        return;

    for (size_t i = i0; i <= i1; i++)
    {
        double xx = ci->m_X1[i] - ci->m_X2[i];
        double yy = ci->m_Y1[i] - ci->m_Y2[i];
        double zz = ci->m_Z1[i] - ci->m_Z2[i];

        ci->m_Result[i] = sqrt(1.0 / sqrt(xx * xx + yy * yy + zz * zz));
    }

    ci->m_Status = 1;
}

static void CalcResultThread(CalcInfo* ci, CoutInfo* cout_info)
{
    if (cout_info->m_ThreadMsgEnable)
        DisplayThreadMsg(ci, cout_info, "ENTER CalcResultThread()");

    CalcResult_(ci);
```

```
    if (cout_info->m_ThreadMsgEnable)
        DisplayThreadMsg(ci, cout_info, "EXIT CalcResultThread()");
}

void RunMultipleThreads(bool thread_msg_enable)
{
    // Code section #1

    size_t align = c_Align;
    size_t num_elements_max;
    vector<size_t> num_elements_vec = GetNumElementsVec(&num_elements_max);
    vector<size_t> num_threads_vec = GetNumThreadsVec();

    AlignedArray<double> x1_aa(num_elements_max, align);
    AlignedArray<double> x2_aa(num_elements_max, align);
    AlignedArray<double> y1_aa(num_elements_max, align);
    AlignedArray<double> y2_aa(num_elements_max, align);
    AlignedArray<double> z1_aa(num_elements_max, align);
    AlignedArray<double> z2_aa(num_elements_max, align);
    AlignedArray<double> result1_aa(num_elements_max, align);
    AlignedArray<double> result2_aa(num_elements_max, align);

    double* x1 = x1_aa.Data();
    double* x2 = x2_aa.Data();
    double* y1 = y1_aa.Data();
    double* y2 = y2_aa.Data();
    double* z1 = z1_aa.Data();
    double* z2 = z2_aa.Data();
    double* result1 = result1_aa.Data();
    double* result2 = result2_aa.Data();

    cout << "Begin initialization of test arrays\n";
    cout << "  Initializing test arrays x1, x2\n";
    Init(x1, x2, num_elements_max, 307);
    cout << "  Initializing test arrays y1, y2\n";
    Init(y1, y2, num_elements_max, 401);
    cout << "  Initializing test arrays z1, z2\n";
    Init(z1, z2, num_elements_max, 503);
    cout << "Finished initialization of test arrays\n";

    CalcInfo ci1;
    ci1.m_X1 = x1;  ci1.m_X2 = x2;
    ci1.m_Y1 = y1;  ci1.m_Y2 = y2;
    ci1.m_Z1 = z1;  ci1.m_Z2 = z2;
    ci1.m_Result = result1;
    ci1.m_Index0 = 0;
    ci1.m_Index1 = num_elements_max - 1;
    ci1.m_Status = -1;
```

```
// CalcResultCpp used for verification purposes
cout << "Begin execution of CalcResultCpp\n";
CalcResultCpp(&ci1);
cout << "Finished execution of CalcResultCpp\n";

size_t iteration = 0;
const size_t block_size = c_BlockSize;
BmThreadTimer bmtt(num_elements_vec.size(), num_threads_vec.size());

// Code section #2

cout << "Begin execution of calculating threads\n";

for (size_t i = 0; i < num_elements_vec.size(); i++)
{
    size_t num_elements = num_elements_vec[i] * 1024 * 1024;
    size_t num_blocks = num_elements / block_size;
    size_t num_blocks_rem = num_elements % block_size;

    if (num_blocks_rem != 0)
        throw runtime_error("num_elements must be an integer multiple of block_size");

    for (size_t j = 0; j < num_threads_vec.size(); j++)
    {
        size_t num_threads = num_threads_vec[j];

        bmtt.Start(i, j);

        size_t num_blocks_per_thread = num_blocks / num_threads;
        size_t num_blocks_per_thread_rem = num_blocks % num_threads;

        vector<CalcInfo> ci2(num_threads);
        vector<CoutInfo> cout_info(num_threads);
        vector<thread*> threads(num_threads);

        // Thread start code
        for (size_t k = 0; k < num_threads; k++)
        {
            ci2[k].m_X1 = x1;   ci2[k].m_X2 = x2;
            ci2[k].m_Y1 = y1;   ci2[k].m_Y2 = y2;
            ci2[k].m_Z1 = z1;   ci2[k].m_Z2 = z2;

            ci2[k].m_Result = result2;
            ci2[k].m_Index0 = k * num_blocks_per_thread * block_size;
            ci2[k].m_Index1 = (k + 1) * num_blocks_per_thread * block_size - 1;
            ci2[k].m_Status = -1;

            if ((k + 1) == num_threads)
                ci2[k].m_Index1 += num_blocks_per_thread_rem * block_size;
```

```
                cout_info[k].m_ThreadMsgEnable = thread_msg_enable;
                cout_info[k].m_Iteration = iteration;
                cout_info[k].m_NumElements = num_elements;
                cout_info[k].m_NumThreads = num_threads;
                cout_info[k].m_ThreadId = k;

                threads[k] = new thread(CalcResultThread, &ci2[k], &cout_info[k]);
            }

            // Wait for all threads to complete
            for (size_t k = 0; k < num_threads; k++)
                threads[k]->join();

            bmtt.Stop(i, j);

            size_t cmp_index = CompareResults(result1, result2, num_elements);

            if (cmp_index != num_elements)
            {
                ostringstream oss;
                oss << "  compare error detected at index " << cmp_index;
                throw runtime_error(oss.str());
            }

            for (size_t k = 0; k < num_threads; k++)
            {
                if (ci2[k].m_Status != 1)
                {
                    ostringstream oss;
                    oss << "  invalid status code " << ci2[k].m_Status;
                    throw runtime_error(oss.str());
                }

                delete threads[k];
            }
        }

        iteration++;
    }

    cout << "Finished execution of calculating threads\n";

    string fn = bmtt.BuildCsvFilenameString("Ch16_04_MultipleThreads_BM");
    bmtt.SaveElapsedTimes(fn, BmThreadTimer::EtUnit::MilliSec, 0);
    cout << "Benchmark times save to file " << fn << '\n';
}
```

```
int main()
{
    try
    {
        RunMultipleThreads(c_ThreadMsgEnable);
    }

    catch (runtime_error& rte)
    {
        cout << "'runtime_error' exception has occurred - " << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Unexpected exception has occurred\n";
    }

    return 0;
}

;-------------------------------------------------
;               Ch16_04_.asm
;-------------------------------------------------

        include <MacrosX86-64-AVX.asmh>

CalcInfo struct
    X1 qword ?
    X2 qword ?
    Y1 qword ?
    Y2 qword ?
    Z1 qword ?
    Z2 qword ?
    Result qword ?
    Index0 qword ?
    Index1 qword ?
    Status dword ?
CalcInfo ends

        .const
r8_1p0  real8 1.0

; extern "C" void CalcResult_(CalcInfo* ci)

        .code
CalcResult_ proc frame
        _CreateFrame CR,0,16,r12,r13,r14,r15
        _SaveXmmRegs xmm6
        _EndProlog

        mov dword ptr [rcx+CalcInfo.Status],0
```

```
; Make sure num_elements is valid
        mov rax,[rcx+CalcInfo.Index0]       ;rax = start index
        mov rdx,[rcx+CalcInfo.Index1]       ;rdx = stop index
        sub rdx,rax
        add rdx,1                           ;rdx = num_elements
        test rdx,rdx
        jz Done                             ;jump if num_elements == 0
        test rdx,7
        jnz Done                            ;jump if num_elements % 8 != 0

; Make sure all arrays are properly aligned
        mov r8d,1fh
        mov r9,[rcx+CalcInfo.Result]
        test r9,r8
        jnz Done

        mov r10,[rcx+CalcInfo.X1]
        test r10,r8
        jnz Done
        mov r11,[rcx+CalcInfo.X2]
        test r11,r8
        jnz Done

        mov r12,[rcx+CalcInfo.Y1]
        test r12,r8
        jnz Done
        mov r13,[rcx+CalcInfo.Y2]
        test r13,r8
        jnz Done

        mov r14,[rcx+CalcInfo.Z1]
        test r14,r8
        jnz Done
        mov r15,[rcx+CalcInfo.Z2]
        test r15,r8
        jnz Done

        vbroadcastsd ymm6,real8 ptr [r8_1p0]        ;ymm6 = packed 1.0 (DPFP)

; Perform simulated calculation
        align 16
LP1:    vmovapd ymm0,ymmword ptr [r10+rax*8]
        vmovapd ymm1,ymmword ptr [r12+rax*8]
        vmovapd ymm2,ymmword ptr [r14+rax*8]
        vsubpd ymm0,ymm0,ymmword ptr [r11+rax*8]
        vsubpd ymm1,ymm1,ymmword ptr [r13+rax*8]
        vsubpd ymm2,ymm2,ymmword ptr [r15+rax*8]
        vmulpd ymm3,ymm0,ymm0
        vmulpd ymm4,ymm1,ymm1
        vmulpd ymm5,ymm2,ymm2
        vaddpd ymm0,ymm3,ymm4
```

```
        vaddpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vdivpd ymm3,ymm6,ymm2
        vsqrtpd ymm4,ymm3
        vmovntpd ymmword ptr [r9+rax*8],ymm4

        add rax,4

        vmovapd ymm0,ymmword ptr [r10+rax*8]
        vmovapd ymm1,ymmword ptr [r12+rax*8]
        vmovapd ymm2,ymmword ptr [r14+rax*8]
        vsubpd ymm0,ymm0,ymmword ptr [r11+rax*8]
        vsubpd ymm1,ymm1,ymmword ptr [r13+rax*8]
        vsubpd ymm2,ymm2,ymmword ptr [r15+rax*8]
        vmulpd ymm3,ymm0,ymm0
        vmulpd ymm4,ymm1,ymm1
        vmulpd ymm5,ymm2,ymm2
        vaddpd ymm0,ymm3,ymm4
        vaddpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vdivpd ymm3,ymm6,ymm2
        vsqrtpd ymm4,ymm3
        vmovntpd ymmword ptr [r9+rax*8],ymm4

        add rax,4
        sub rdx,8
        jnz LP1

        mov dword ptr [rcx+CalcInfo.Status],1

Done:   vzeroupper
        _RestoreXmmRegs xmm6
        _DeleteFrame r12,r13,r14,r15
        ret
CalcResult_ endp
        end
```

Source code example Ch16_04 performs a simulated calculation using large arrays of double-precision floating-point values across multiple threads. It uses the C++ STL class thread to run multiple instances of an AVX2 assembly language calculating function. Each thread performs its calculations using only a portion of the array data. The C++ driver routine exercises the calculating algorithm using various combinations of array sizes and simultaneously executing threads. It also implements benchmark timing measurements to quantify the performance benefits of the multithreaded technique.

Listing 16-4 begins the header file Ch16_04.h. In this file, the structure CalcInfo contains the data that each thread needs to carry out its calculations. The structure members m_X1, m_X2, m_Y1, m_Y2, m_Z2, and m_Z2, point to the source arrays, while m_Result points to the destination array. Members m_Index0 and m_Index1 are array indices that define a range of unique elements for each calculating thread. Header file Ch16_04.h also includes a structure named CoutInfo, which contains status information that's optionally displayed during program execution.

The next file in Listing 16-4 is Ch16_04_Misc.cpp. This file contains the source code for the program's ancillary functions. The functions Init and CompareResults perform array initialization and verification, respectively. The next function is DisplayThreadMsg. This function displays status information for each executing thread. Note that the cout statements in DisplayThreadMsg are synchronized with a C++ STL mutex. A mutex is a synchronization object that facilitates controlled access to a single resource by multiple threads. When mutex_cout is locked, only one thread can stream its status results to cout. The other executing threads are blocked from streaming their results to cout until mutex_cout becomes unlocked. Without this mutex, status information text from the executing threads would be intermingled on the display (if you're interested in seeing what happens, try commenting out the mutex_cout.lock and mutext_cout.unlock statements).

The function GetNumElementsVec returns a vector that contains the sizes of the test arrays. Note that the amount of memory required by the largest test array should be less than the amount of available memory plus a small fudge factor. The fudge factor prevents the program from allocating all available memory. Also note that GetNumElementsVec throws an exception if insufficient memory is available since running the program in this manner is very slow due to the amount of page swapping that occurs. The function GetNumThreadsVec returns a vector of test thread counts. You can change the values in num_threads_vec to experiment with different thread count values.

The source code file Ch16_04.cpp contains the driver routines for source code example Ch16_04. The function CalcResultCpp is a C++ implementation of the simulated calculating algorithm and is used for result verification purposes. The next function, CalcResultThread, is the main thread function. This function invokes the assembly language calculating function CalcResult_. It also displays thread status messages if they're enabled.

Following CalcResultThread is the function RunMultipleThreads. This function exercises the calculating algorithm using the specified combinations of array sizes and number of simultaneously executing threads. The first code section of RunMultipleThreads performs array allocation and element initialization. It also calls the function CalcResultCpp to calculate results values for algorithm verification purposes. Note that prior to calling this function, an instance of CalcInfo is initialized with the data that's necessary to carry out the required calculations.

The second code section of RunMultipleThreads, which starts immediate after the comment line Code section #2, runs the calculating algorithm by distributing the test array elements across multiple threads. The test array elements are spilt into groups based on the number of threads that will execute. For example, if the number of test array elements equals 64 million, launching four threads will result in each thread processing 16 million elements. The inner most for loop that follows the comment line Thread start code begins each iteration by initializing an instance of CalcInfo for the next thread. The statement threads[k] = new thread(CalcResultThread, &ci2[k], &cout_info[k]) constructs a new thread object and starts execution of the thread function CalcResultThread using argument values &ci2[k] and &cout_info[k]. This inner for loop repeats until the required number of executing thread have been launched. While the threads are executing, the function RunMultipleThreads executes a small for loop that invokes the function thread::join. This effectively forces RunMultipleThreads to wait until all executing threads have finished. The remaining code in RunMultipleThreads performs data verification and object cleanup.

Before reviewing the assembly language code, the C++ code in RunMultipleThreads merits a few additional comments. The first thing to note is that the benchmarking code measures both the time it takes to carry out the required calculations *and* the overhead that's associated with thread management. If the algorithm that's employed by RunMultipleThreads were to be used in a real-world application, any benchmark timing measurements would be meaningless without factoring in this overhead. It should also be noted that RunMultipleThreads implements an extremely rudimentary form of multithreading that omits many important real-world operations to simplify the code for this example. If you're interested in learning more about the C++ STL thread class and the other STL classes that facilitate multithreaded processing, you are strongly encouraged to consult the references listed in Appendix A.

The final file in Listing 16-4 is Ch16_04_.asm. Near the top of this file is the assembly language counterpart of the data structure CalcInfo. The assembly language function CalcResult_ is next. Following its prolog, CalcResult_ uses the instructions mov rax,[rcx+CalcInfo.Index0] and mov

rdx,[rcx+CalcInfo.Index1] to load registers RAX and RDX with the indices of the first and last array elements. It then calculates and validates num_elements. Next, the test arrays are validated for proper alignment. The processing loop in CalcResult_ uses AVX packed double-precision floating-point arithmetic to carry out the same simulated calculation as CalcResultCpp. The output for source code example Ch16_04 follows this paragraph. Note that this output was produced with c_ThreadMsgEnable set to false. Setting this flag to true ultimately directs the function CalcResultThread to display status messages for each executing thread. The flag c_ThreadMsgEnable is defined near the top of Ch16_04.cpp.

```
Begin initialization of test arrays
  Initializing test arrays x1, x2
  Initializing test arrays y1, y2
  Initializing test arrays z1, z2
Finished initialization of test arrays
Begin execution of CalcResultCpp
Finished execution of CalcResultCpp
Begin execution of calculating threads
Finished execution of calculating threads
Benchmark times save to file Ch16_04_MultipleThreads_BM_CHROMIUM.csv
```

Tables 16-4, 16-5, and 16-6 contain the benchmark timing measurements for source code example Ch16_04. The measurements shown in these tables are the mean execution times from 10 separate runs and were made with 32 GB of SDRAM installed in each test computer. All three test computers show a significant improvement in performance when multiple threads are used to carry out the simulated calculation. For the i7-4900s and i7-8700k test computers, optimal performance is attained using four threads. The i9-7900x test computer shows meaningful performance gains when using six or eight threads. It is interesting to compare the timing measurements for the i9-7900x and i7-8700k systems. When using one or two threads, the i7-8700k out performs the i9-7900x while the opposite is true when four or more threads are employed. These measurements make sense when considering the hardware differences between the test processors, which are shown in Table 16-7. Even though the i7-8700k employs higher-clock frequencies, the extra memory channels of the i9-7900x enable it to better utilize its CPU cores and complete the required calculations more quickly.

***Table 16-4.*** *Benchmark Timing Measurements (Milliseconds) for RunMultipleThreads Using an Intel i7-4790s Processor*

| | Number of Threads | | | | |
|---|---|---|---|---|---|
| **Number of Elements (Millions)** | **1** | **2** | **4** | **6** | **8** |
| 64 | 686 | 226 | 178 | 180 | 172 |
| 128 | 1146 | 491 | 345 | 355 | 347 |
| 192 | 1592 | 702 | 513 | 530 | 516 |
| 256 | 2054 | 942 | 679 | 714 | 688 |

***Table 16-5.*** *Benchmark Timing Measurements (Milliseconds) for `RunMultipleThreads` Using an Intel i9-7900x Processor*

| | Number of Threads | | | | |
|---|---|---|---|---|---|
| **Number of Elements (Millions)** | **1** | **2** | **4** | **6** | **8** |
| 64 | 492 | 137 | 84 | 69 | 61 |
| 128 | 765 | 300 | 163 | 131 | 121 |
| 192 | 1110 | 454 | 233 | 193 | 178 |
| 256 | 1330 | 582 | 313 | 260 | 238 |

***Table 16-6.*** *Benchmark Timing Measurements (Milliseconds) for `RunMultipleThreads` Using an Intel i7-8700k Processor*

| | Number of Threads | | | | |
|---|---|---|---|---|---|
| **Number of Elements (Millions)** | **1** | **2** | **4** | **6** | **8** |
| 64 | 332 | 125 | 120 | 123 | 123 |
| 128 | 522 | 265 | 240 | 245 | 246 |
| 192 | 839 | 387 | 363 | 366 | 369 |
| 256 | 919 | 499 | 478 | 484 | 492 |

***Table 16-7.*** *Summary of Hardware Features for Test Processors Used in Example Ch16_04*

| Hardware Feature | i7-4790s | i9-7900x | i7-8700k |
|---|---|---|---|
| Number of cores | 4 | 10 | 6 |
| Number of threads | 8 | 20 | 12 |
| Base frequency (GHz) | 3.2 | 3.3 | 3.7 |
| Maximum frequency (GHz) | 4.0 | 4.5 | 4.7 |
| Memory type | DDR3-1600 | DDR4-2666 | DDR4-2666 |
| Number of memory channels | 2 | 4 | 2 |

## Summary

Here are the key learning points for Chapter 16:

- An application program should always use the `cpuid` instruction to verify processor support for specific instruction set extensions. This is extremely important for software compatibility with future processors from both AMD and Intel.
- An assembly language function can use the non-temporal store instructions `vmovntp[d|s]` instead of the `vmovap[d|s]` instructions to improve the performance of algorithms that carry out calculations using large arrays of non-temporal floating-point data.
- An assembly language function can use the `prefetch[0|1|2]` instructions to pre-load temporal data into the processor's cache hierarchy. A function can also use the `prefetchnta` instruction to pre-load non-temporal data and minimize cache pollution. The performance benefits of the prefetch instructions vary depending on data access patterns and the processor's underlying microarchitecture.
- A multithreaded algorithm that's implemented in a high-level language such as C++ can exploit AVX, AVX2, or AVX-512 assembly language calculating functions to accelerate an algorithm's overall performance.

# Appendix A

Appendix A includes supplemental material about the following items:

- Software utilities for x86 processors
- Visual Studio
- References

## Software Utilities for x86 Processors

The following utilities can be used to determine which x86 instruction set extensions are supported by the processor in your computer:

CPUID CPU-Z (https://www.cpuid.com)

HWiNFO Diagnostic Software (https://www.hwinfo.com)

Piriform SPECCY (https://www.ccleaner.com/speccy)

## Visual Studio

In this section, you'll learn how to use Microsoft's Visual Studio development tool to run the source code examples that are described in the main text. You'll also learn how to create a simple Visual Studio C++ project. Before proceeding, you may want to refer to the Introduction for additional information regarding Visual Studio and the recommended hardware platforms for running the source code examples. The Introduction also contains important details about downloading the source code ZIP files for each chapter.

Visual Studio uses logical entities called solutions and projects to help simplify application development. A solution is a collection of one or more projects that are used to build an application. A project is container object that organizes an application's files. A Visual Studio project is usually created for each buildable component of an application (e.g., executable file, dynamic-linked library, static library, and so on).

A standard Visual Studio C++ project includes two solution configurations named Debug and Release. As implied by their names, these configurations support separate executable builds for initial development and final release. A standard Visual Studio C++ project also incorporates solution platforms. The default solution platforms are named Win32 and x64, which contain the necessary settings to build 32-bit and 64-bit executables, respectively. The Visual Studio solution and project files for this book's source code examples include only the x64 platform.

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

## Running a Source Code Example

You can use the following steps to run any of the book's source code examples:

1. Using File Explorer, double-click on the chapter's Visual Studio solution (`.sln`) file. The solution file is included in the chapter source code ZIP file.
2. From the menu bar, select Build | Configuration Manager. In the Configuration Manager dialog box, set Active Solution Configuration to **Release**. Then set Active Solution Platform to **x64**. Note that these options may already be selected.
3. If necessary, select View | Solution Explorer to open the Solution Explorer window.
4. In the Solution Explorer window, right-click on a project to run and choose **Set as StartUp Project**.
5. Select Debug | Start Without Debugging to run the program.

Some of the source code examples reference data files in different folders using fixed path names. To run the corresponding executables using a different folder structure than the one used for Visual Studio development, you may need to change the path name strings in the C++ source code.

## Creating a Visual Studio C++ Project

In this section, you'll learn how to create a simple Visual Studio project that includes both C++ and assembly language source code files. The ensuing paragraphs describe the same basic procedure that was used to create the source code examples in the main text and includes the following phases:

- Create a C++ project
- Enable MASM support
- Add an assembly language file
- Set project properties
- Edit the source code
- Build and run the project

### Create a C++ Project

Use the following steps to create a Visual Studio C++ project:

1. Start Visual Studio.
2. Select File | New Project.
3. In the New Project dialog box control tree, select Installed | Visual C++ | Windows Desktop.
4. Select Windows Console Application for the project type.
5. In the Name text box, enter `Example1`.

6. In the Location text box, enter a folder name for the project location. You can also use the Browse button to choose a folder or leave the text unchanged to use the default location.

7. In the Solution text box, enter `TestSolution`.

8. Verify that the New Project dialog box settings are the same as the ones shown in Figure A-1 (the Location can be different). Click OK.

9. If necessary, select View | Solution Explorer to open the Solution Explorer window.

10. In the Solution Explorer tree control, right-click on the top-level text that's labeled **Solution 'Example 1' (1 Project)** and select **Rename**. Change the solution name to `TestSolution`.

11. Select Build | Configuration Manager. In the Configuration Manager dialog box, choose **<Edit...>** under Active Solution Platforms (see Figure A-2).

12. In the Edit Solution Platforms dialog box, select **x86** and click Remove (see Figure A-3). Click Close to close the Edit Solutions Platforms dialog box; click Close to close the Configuration Manager dialog box.

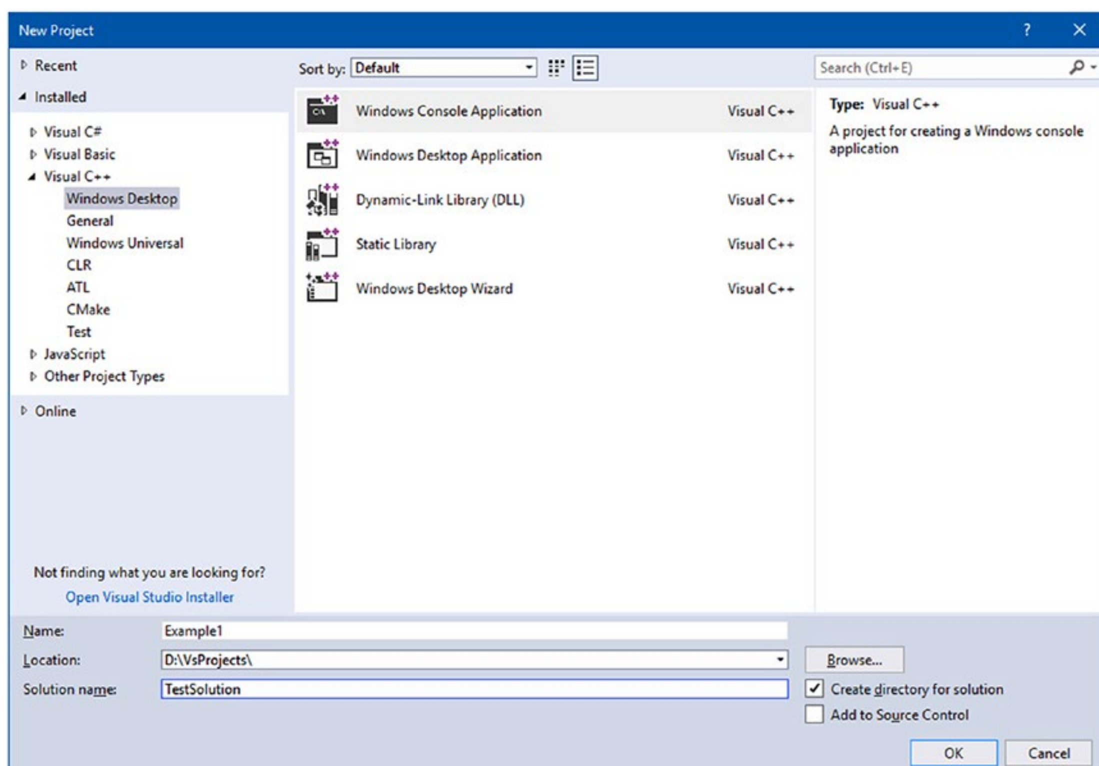



*Figure A-1. New Project dialog box*

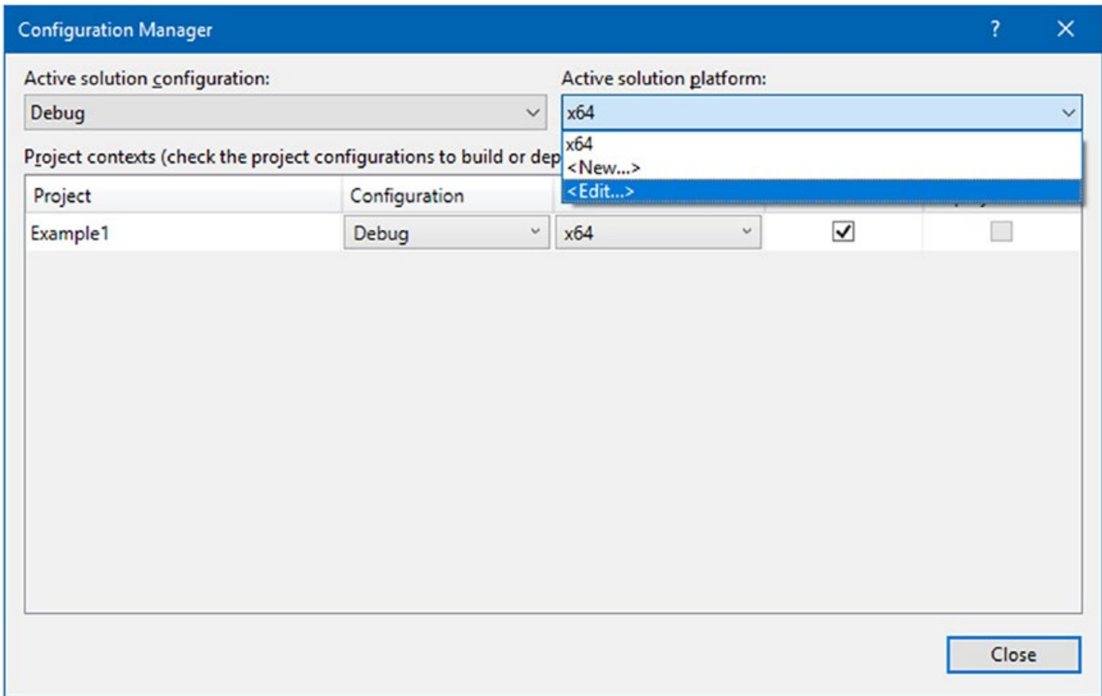



*Figure A-2.* *Configuration Manager dialog box*

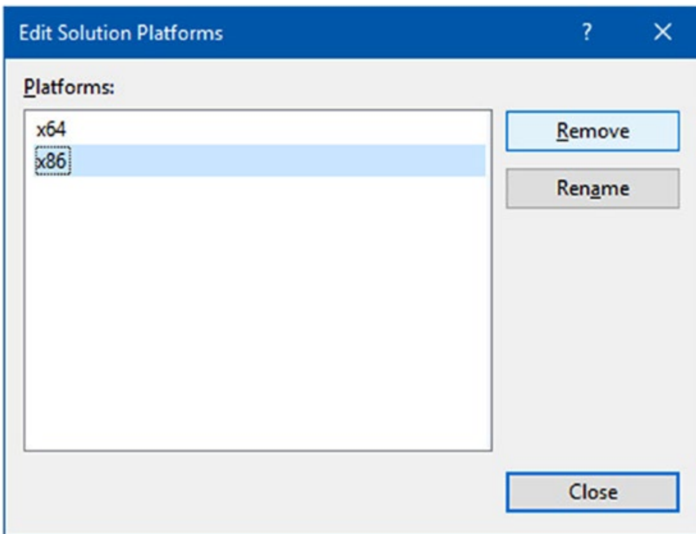



*Figure A-3.* *Edit Solution Platforms dialog box*

## Enable MASM Support

Use the following steps to enable support for Microsoft Macro Assembler:

1. In the Solution Explorer tree control, right-click on **Example1** and select Build Dependencies | Build Customizations.
2. In the Visual C++ Build Customizations dialog box, check **masm(.targets, .props)**.
3. Click OK.

## Add an Assembly Language File

Use the following steps to add an assembly language source code file (`.asm`) to a Visual Studio C++ project:

1. In the Solution Explorer tree control, right-click on **Example1** and select Add | New Item.
2. Select **C++ File (.cpp)** for the file type.
3. In the Name text box, change the name to `Example1_.asm`, as shown in Figure A-4. Note that the trailing underscore is required since all C++ and assembly language source code files in a project must have a unique base name.
4. Click Add.

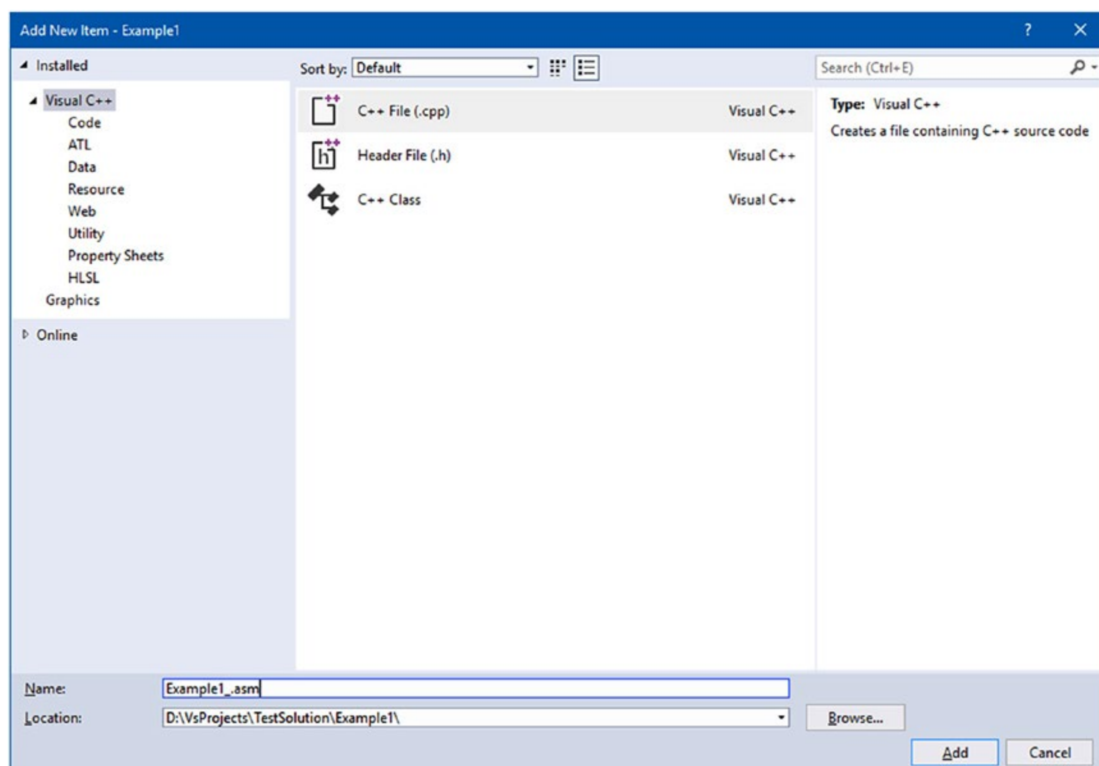



*Figure A-4. Add New Item dialog box*

## Set Project Properties

Use the following steps to set the project's properties. The properties that control listing file generation (Steps 5 - 8) are optional.

1. In the Solution Explorer tree control, right-click on **Example1** and select **Properties**.
2. In the Property Pages dialog box, change the Configuration setting to **All Configurations** and the Platform setting to **All Platforms**. Note that one or both options may already be set.
3. In the tree control, select Configuration Properties | General. Change the setting Whole Program Optimization to **No Whole Program Optimization** (see Figure A-5).
4. Select Configuration Properties | C/C++ | Code Generation. Change the setting Enable Enhanced Instruction Set to **Advanced Vector Extensions (/arch:AVX)** (see Figure A-6)
5. Select Configuration Properties | C/C++ | Output Files. Change the setting Assembler Output to **Assembly Machine and Source Code (/FAcs)** (see Figure A-7).
6. Select Configuration Properties | Microsoft Macro Assembler | Listing File. Change the setting Enable Assembly Generated Code Listing to **Yes (/Sg)** (see Figure A-8).
7. Change the Assembled Code Listing File text field to `$(IntDir)\%(filename).lst` (see Figure A-8). This macro text specifies the project's intermediate directory, which is a subfolder of the main project folder.
8. Click OK.

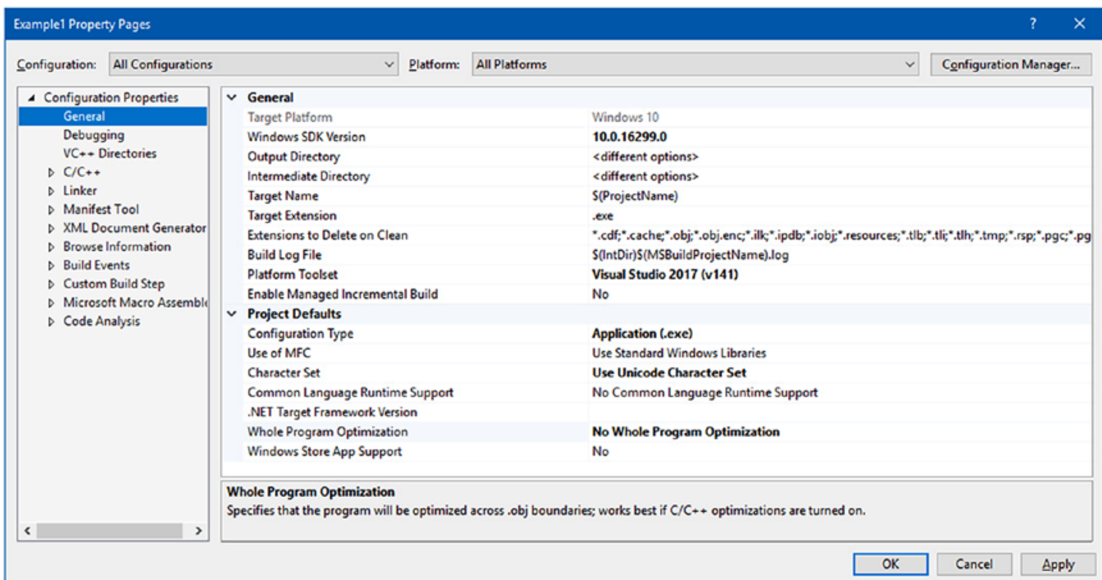



***Figure A-5.*** *Property Pages dialog box (Whole Program Optimization)*

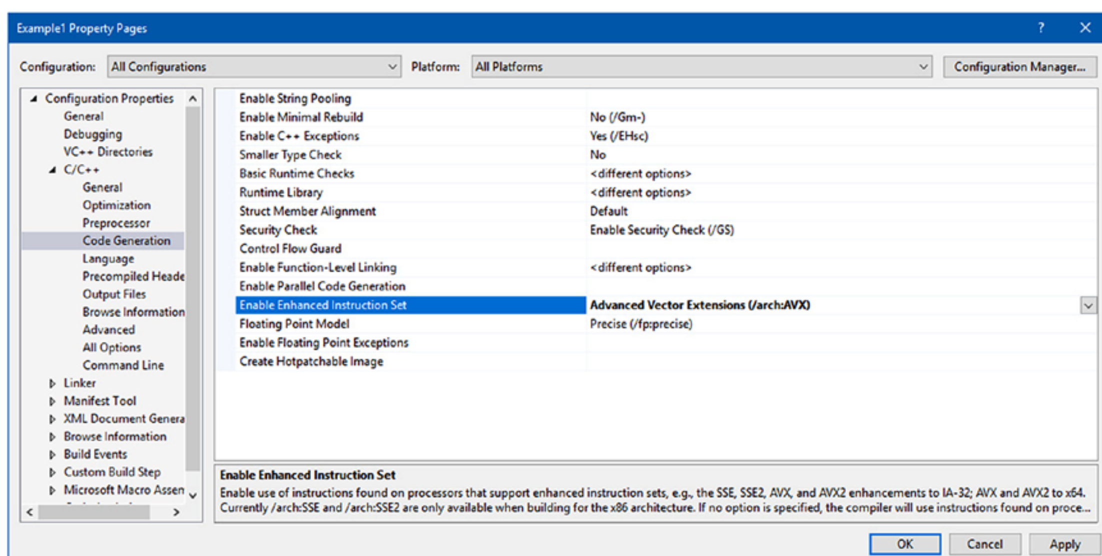



***Figure A-6.*** *Property Pages dialog box (Enable Enhanced Instruction Set)*

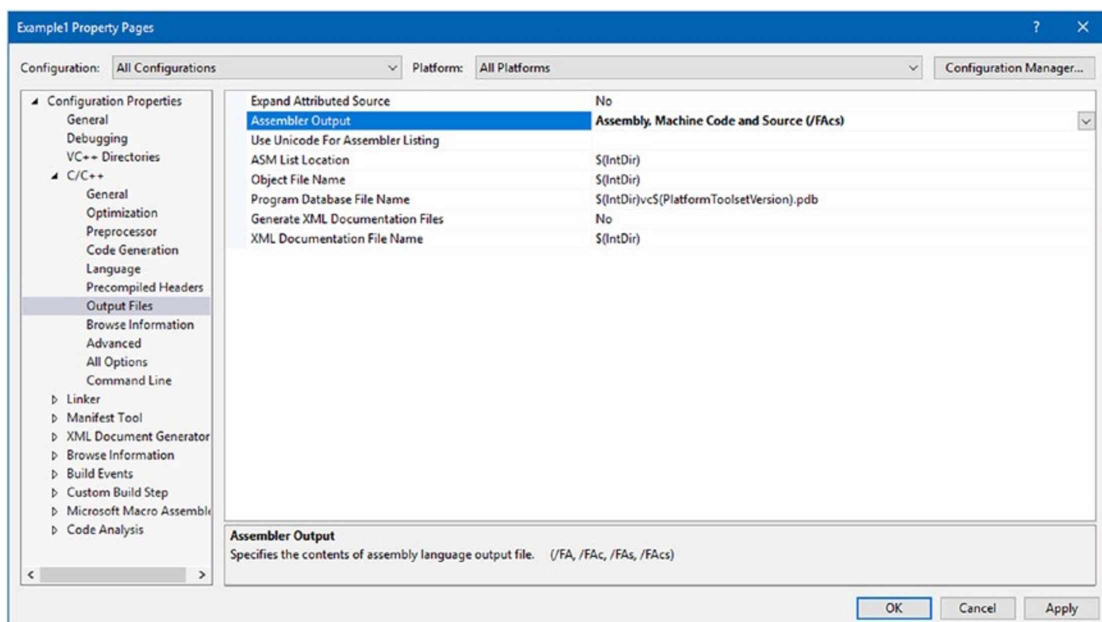



***Figure A-7.*** *Property Pages dialog box (Assembler Output)*

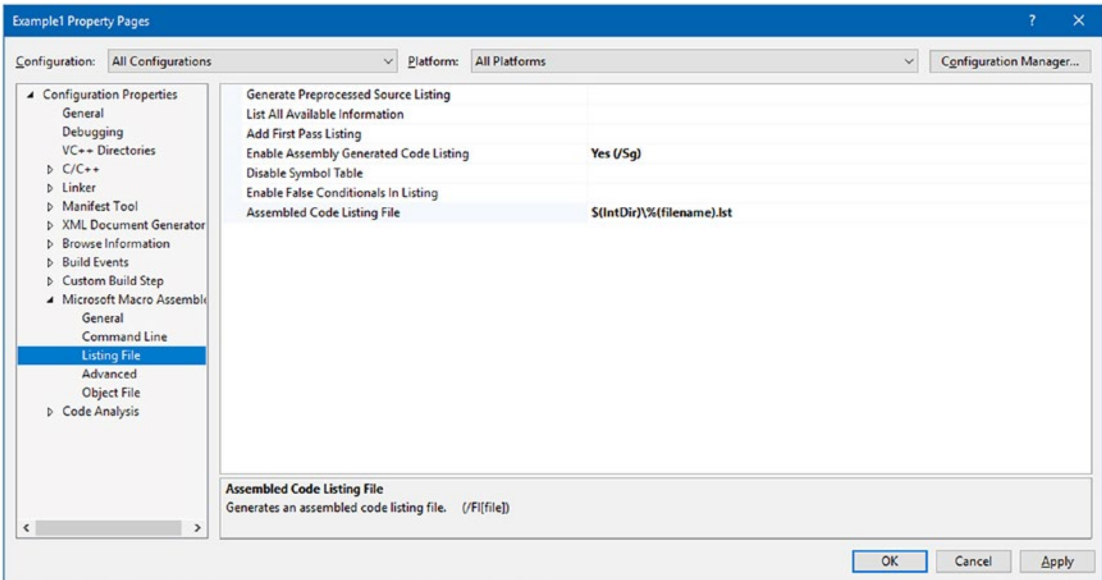



***Figure A-8.*** *Property Pages dialog box (Microsoft Macro Assembler Listing File)*

## Edit the Source Code

Use the following steps to edit the project source code:

1. In the Editor window, click on the tab named **Example1.cpp**.
2. Edit the C++ source code to match the code that's shown in Listing A-1.
3. Click on the tab named **Example1_.asm**.
4. Edit the assembly language source code to match the code that's shown in Listing A-2.
5. Select File | Save All.

***Listing A-1.*** Example1.cpp

```
// Example1.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>

using namespace std;

extern "C" int CalcResult1_(int val1, int val2, int* quo, int* rem);

int main()
{
    int val1 = 42;
    int val2 = 9;
```

```
    int quo;
    int rem;
    int prod = CalcResult1_(val1, val2, &quo, &rem);

    cout << "Results for Example1\n";
    cout << "val1 = " << val1 << '\n';
    cout << "val2 = " << val2 << '\n';
    cout << "quo =  " << quo << '\n';
    cout << "rem =  " << rem << '\n';
    cout << "prod = " << prod << '\n';
    return 0;
}
```

***Listing A-2.*** Example1_.asm

```
; extern "C" int CalcResult1_(int val1, int val2, int* quo, int* rem);

        .code
CalcResult1_ proc
        mov r10d,ecx                        ;r10d = val1
        mov r11d,edx                        ;r11d = val2

        mov eax,ecx                         ;eax = val1
        cdq                                 ;edx:eax = val1

        idiv r11d                           ;calc val1 / val2
        mov dword ptr [r8],eax              ;save quotient
        mov dword ptr [r9],edx              ;save remainder

        imul r10d,r11d                      ;r10d = val1 * val2
        mov eax,r10d                        ;eax = val1 * val2
        ret
CalcResult1_ endp
        end
```

## Build and Run the Project

Use the following steps to build and run the project:

1. Select Build | Build Solution.
2. If necessary, fix any reported C++ compiler or MASM errors and repeat Step 1.
3. Select Debug | Start Without Debugging.
4. Verify that the output matches the console window shown in Figure A-9.
5. Press Enter to close the console window.

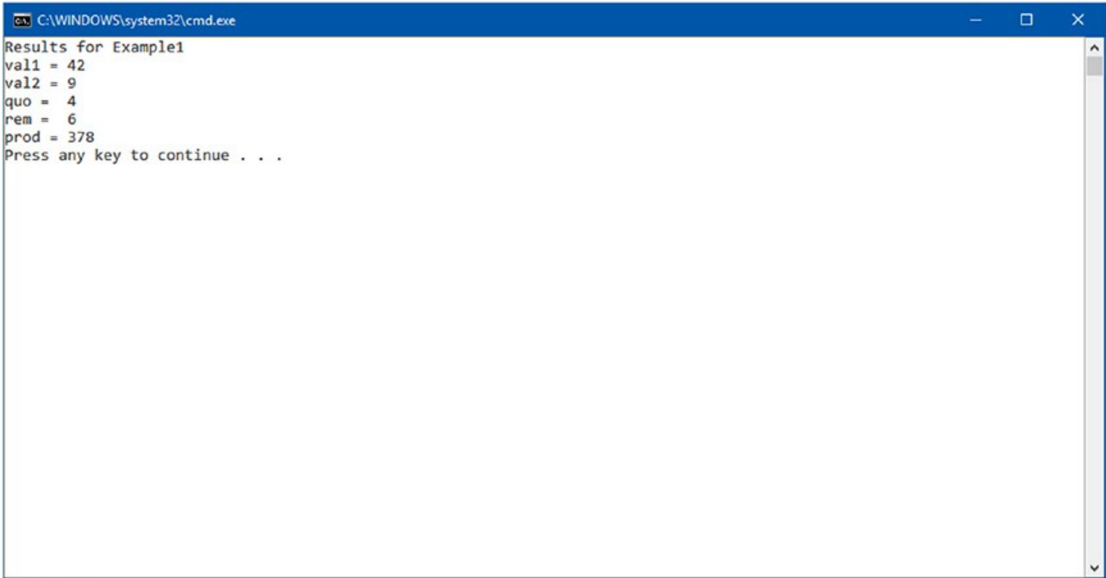



***Figure A-9.*** *Console window output*

# References

This section contains a list of references that were consulted during preparation of the main text. It also includes additional references and resources that provide worthwhile information. The references have been grouped into the following categories:

X86 programming reference manuals

X86 programming and microarchitecture references

Ancillary resources

Algorithm references

C++ references

## X86 Programming Reference Manuals

The following is a list of x86 programming reference manuals published by AMD and Intel:

*AMD64 Architecture Programmer's Manual Volume 1: Application Programming* https://support.amd.com/TechDocs/24592.pdf

*AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions*, https://support.amd.com/TechDocs/24594.pdf

*AMD64 Architecture Programmer's Manual Volume 4: 128-bit and 256-bit Media Instructions*, https://support.amd.com/TechDocs/26568.pdf

*Software Optimization Guide for AMD Family 17h Processors, Publication Number 55723*, June 2017, https://developer.amd.com/resources/developer-guides-manuals

*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

*Intel 64 and IA-32 Architectures Optimization Reference Manual*, https://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

*Intel Architecture Instruction Set Extensions and Future Features Programming Reference*, https://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

## X86 Programming and Microarchitecture References

The follow resources contain informative information about x86 assembly language programming, processors, and microarchitectures.

Guy Ben-Haim, Itai Neoran, and Ishay Tubi, *Practical Intel AVX Optimization on 2nd Generation Intel Core Processors*, https://software.intel.com/sites/default/files/m/d/4/1/d/8/Practical_Optimization_with_AVX.pdf

Ian Cutress, *The Intel Skylake Mobile and Desktop Launch, with Architecture Analysis*, September 2015, https://www.anandtech.com/show/9582/intel-skylake-mobile-desktop-launch-architecture-analysis

Ian Cutress, *The Intel Skylake-X Review: Core i9-7900X, i7-7820X and i7-7800X Tested*, June 2017, https://www.anandtech.com/show/11550/the-intel-skylakex-review-core-i9-7900x-i7-7820x-and-i7-7800x-tested

Anger Fog, *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, August 2018, https://agner.org/optimize/#manuals

Agner Fog, *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*, April 2018, https://agner.org/optimize/#manuals

Chris Kirkpatrick, *Intel AVX State Transitions: Migrating SSE Code to AVX*, https://software.intel.com/en-us/articles/intel-avx-state-transitions-migrating-sse-code-to-avx

Patrick Konsor, *Avoiding AVX-SSE Transition Penalties*, https://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties

Patrick Konsor, *Performance Benefits of Half-Precision Floats*, https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats

Daniel Kusswurm, *Modern x86 Assembly Language Programming*, Apress, ISBN 978-1-4842-0065-0, 2014.

Max Locktyukhin, *How to Detect New Instruction Support in the 4th Generation Intel Core Processor Family*, August 2013, https://software.intel.com/en-us/node/405250

John Morgan, *Microsoft Visual Studio 2017 Supports Intel AVX-512*, https://blogs.msdn.microsoft.com/vcblog/2017/07/11/microsoft-visual-studio-2017-supports-intel-avx-512

Erdinc Ozturk, James Guilford, Vinodh Gopal, and Wajdi Feghal, *New Instructions Supporting Large Integer Arithmetic on Intel Architecture Processors*, August 2012, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf

James Reinders, *AVX-512 May Be a Hidden Gem in Intel Xeon Scalable Processors*, June 2017, https://www.hpcwire.com/2017/06/29/reinders-avx-512-may-hidden-gem-intel-xeon-scalable-processors

Anand Lal Shimpi, *Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel*, October 2012, http://www.anandtech.com/show/6355/intels-haswell-architecture

## Ancillary Resources

The following resources contain useful information about x86 processors and microarchitectures:

*Processors for Desktops*, AMD, https://www.amd.com/en/products/processors-desktop

*List of AMD Accelerated Processing Unit Microprocessors*, Wikipedia, https://en.wikipedia.org/wiki/List_of_AMD_Accelerated_Processing_Unit_microprocessors

*List of AMD CPU Microarchitectures*, Wikipedia, https://en.wikipedia.org/wiki/List_of_AMD_CPU_microarchitectures

*List of AMD Microprocessors*, Wikipedia, https://en.wikipedia.org/wiki/List_of_AMD_processors

*Product Information Website*, Intel, https://ark.intel.com

*List of Intel CPU Microarchitectures*, Wikipedia, https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures

*List of Intel Microprocessors*, Wikipedia, https://en.wikipedia.org/wiki/Intel_processor

*List of Intel Xeon Microprocessors*, Wikipedia, https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors

*Register Renaming*, Wikipedia, https://en.wikipedia.org/wiki/Register_renaming

## Algorithm References

The following resources were consulted to develop the algorithms used in the source code examples:


Forman S. Acton, *REAL Computing Made REAL – Preventing Errors in Scientific and Engineering Calculations*, ISBN 978-0486442211, Dover Publications, 2005

Tony Chan, Gene Golub, Randall LeVeque, *Algorithms for Computing the Sample Variance: Analysis and Recommendations*, The American Statistician, Volume 37 Number 3 (1983), p. 242-247

James F. Epperson, *An Introduction to Numerical Methods and Analysis, Second Edition*, ISBN 978-1-118-36759-9, Wiley, 2013

David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Volume 23 Issue 1 (March 1991), p. 5 – 48

Rafael C. Gonzalez and Richard E. Woods, *Digital Image Processing, Fourth Edition*, ISBN 978-0-133-35672-4, 2018

James E. Miller, David G. Moursund, Charles S. Duris, *Elementary Theory & Application of Numerical Analysis, Revised Edition*, ISBN 978-0486479064, Dover Publications, 2011

Anthony Pettofrezzo, *Matrices and Transformations*, ISBN 0-486-63634-8, Dover Publications, 1978

Hans Schneider and George Barker, *Matrices and Linear Algebra*, ISBN 0-486-66014-1, Dover Publications, 1989

Eric W. Weisstein, *Convolution*, MathWorld, http://mathworld.wolfram.com/Convolution.html

Eric W. Weisstein, *Correlation Coefficient*, MathWorld, http://mathworld.wolfram.com/CorrelationCoefficient.html

Eric W. Weisstein, *Cross Product*, MathWorld, http://mathworld.wolfram.com/CrossProduct.html

Eric W. Weisstein, *Least Squares Fitting*, MathWorld, http://mathworld.wolfram.com/LeastSquaresFitting.html

Eric W. Weisstein, *Matrix Multiplication*, MathWorld, http://mathworld.wolfram.com/MatrixMultiplication.html

David M. Young and Robert Todd Gregory, *A Survey of Numerical Mathematics, Volume 1*, ISBN 0-486-65691-8, Dover Publications, 1988

*Algorithms for calculating variance*, Wikipedia, https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

*Body Surface Area Calculator*, http://www.globalrph.com/bsa2.htm

*Grayscale*, Wikipedia, https://en.wikipedia.org/wiki/Grayscale

*Linked List*, Wikipedia, https://en.wikipedia.org/wiki/Linked_list

## C++ References

The following resources contain valuable information about C++ programming, the C++ Standard Template Libraries, and C++ programming using multiple threads.

Ivor Horton, *Using the C++ Standard Template Libraries*, Apress, ISBN 978-1-4842-0005-6, 2015

Nicolai M. Josuttis, *The C++ Standard Library – A Tutorial and Reference, Second Edition*, Addison Wesley, ISBN 978-0-321-62321-8, 2012

Bjarne Stroustrup, *The C++ Programming Language, Fourth Edition*, Addison Wesley, ISBN 978-0-321-56384-2, 2013

Anthony Williams, *C++ Concurrency in Action – Practical Multithreading*, ISBN 978-1-933-98877-1, Manning Publications, 2012

*cplusplus.com*, `http://www.cplusplus.com`

# Index

## A


Advanced Vector Extensions (AVX)
- data types
  - packed floating-point, 100–102
  - packed integer, 103–105
  - scalar floating-point, 94–99
- differences between x86-SSE
  - execution lanes, 105
  - intermixing x86-AVX and x86-SSE code, 107
  - operand alignment, 106
  - vzeroupper, 107
  - YMM register high-order bit zeroing, 106
- instruction syntax
  - non-destructive source operand, 93
- registers
  - MXCSR, 92, 94, 97–98
  - XMM registers, 91–94, 97, 104, 106–107
  - YMM registers, 91–93, 99, 105–107

AVX2
- MXCSR, 277, 281
- non-destructive source operand, 277
- operand alignment, 277
- packed floating-point, 278–279
- packed integer
  - variable bit shift, 282
- XMM registers, 277–279, 281
- YMM registers, 277–279, 281

AVX-512
- conditional execution and merging
  - merge masking, 424–425
  - zero masking, 424–425
- data types, 423–424
- embedded broadcast, 426
- instruction-level rounding
  - round down, 427, 443
  - round to nearest, 427, 443
  - round to zero, 427, 443
  - round up, 427, 443
  - suppress all exceptions, 427, 443
- instruction set extensions
  - AV512CD, 430, 433, 491
  - AVX512BW, 430–431, 433, 491, 495
  - AVX512DQ, 421, 422, 427, 431, 433, 491
  - AVX512F, 427–429, 431, 433, 436, 464, 491
  - AVX512VL, 421–422, 427, 433, 491
- instruction syntax
  - conditional execution and merging, 424–426
  - embedded broadcast, 426
  - instruction-level rounding, 427
- merge masking, 433–436, 444, 495, 517
- predicate mask, 422, 424–425
- register sets
  - MXCSR, 427
  - opmask registers, 422
  - XMM registers, 422
  - YMM registers, 422
  - ZMM registers, 422
- zero masking, 433, 437–440, 495, 502, 517

Array of structures (AOS), 473, 476, 526–527

Array operations
- column means
  - row-major ordering, 302
- least squares, 193–198
- min-max, 188–192
- simple calculations, 292–298
- square roots, 184–187

Arrays
- accessing elements, 51–53, 58–61
- comparing, 79–82
- one-dimensional, 51
- reversal, 82–86
- row-major ordering, 58
- two-dimensional
  - row and column indices, 58, 66


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

## B

Benchmark timing measurements
csv file, 207
TRIMMEAN, 207

## C

C++
classes
AlignedArray, 238, 261, 368, 375, 404, 473, 502
AlignedMem, 238
array, 343
BmThreadTimer, 206
default_random_engine, 238
ImageMatrix, 261
matrix, 203
mutex, 581
OS, 206
thread, 580–581
uniform_int_distribution, 238
unique_ptr, 302, 397, 473
lvalue, 355
rvalue, 355
size_t, 77
specifiers
alignas, 171–172, 177, 198, 220, 290, 296, 351, 355–356, 450, 455, 495
Cache
cache line, 531, 539
L1 data (D-Cache), 530–531
L1 instruction (I-Cache), 530–532
L2, 530–532
L3, 530–532
non-temporal data, 557, 562
pollution, 557, 562
slice, 531
temporal data, 557, 562
Conditional jump, 21, 44, 47–48
Conditional move, 21, 44, 48
Condition codes, 17–18, 44–48
Convolution
discrete equation, 386–387, 397–398, 404
input signal
padding, 396–397, 404
kernel
fixed size, 388
variable size, 388
output signal, 385–387, 397, 404–405
response signal, 385–386
SIMD equations, 404–405
theory, 386
YMM registers, 489
ZMM registers, 489
Correlation coefficient, 305–312
CPU Identification (CPUID)
AVX-512 feature flags, 556
feature flag, 555–556
host operating system
OSXSAVE, 555
leaf value, 555
memory caches, 554–557
return results, 554
serializing instruction, 554
sub-leaf value, 554
xgetbv, 554–555

## D

Data blend, 285, 333
Data gather
indices
doubleword, 343
quadword, 343
merge control mask, 339, 343–344
vector scale-index-base, 278
Data permute
indices, 337
Data prefetch
hint, 562
linked list, 563, 569–570
Differences between x86-32 and x86-64 programming
byte register restrictions, 15
deprecated instructions, 15
immediate operands
32-bit, 13–14
invalid instructions, 15
operand sizes, 13

## E

Enhanced bit manipulation
leading zero bits, 414
trailing zero bits, 414

## F, G

Feature set identification. *See* CPUID
Flagless operations
multiplication, 406–411
shift, 406–411
FMA. *See* Fused-Multiply-Add (FMA)
FMA3. *See* Fused-Multiply-Add (FMA)
FMA4. *See* Fused-Multiply-Add (FMA)
Fundamental data types
byte, 3–4
double quadword, 3
doubleword, 3–4

little endian ordering, 3
proper alignment, 4
quadword, 3–4
word, 3
Fused-Multiply-Add (FMA)
arithmetic, 281
convolution functions
packed, 398–406
scalar, 388–398
data dependencies
multiple registers, 397
operand ordering scheme, 282
packed, 281
rounding
MXCSR.RC, 281
scalar, 281
value discrepancies, 406


## H


Half-precision floating-point
encoding
exponent, 280
sign bit, 280
significand, 280
F16C, 280
Half-precision floating-point conversions
rounding mode, 418


## I


IEEE 754
binary encoding
exponent, 95–96
sign bit, 95
significand, 95–96
special values
denormal, 96
floating-point zero, 96
infinity, 96
NaN, 96
QNaN, 96
SNaN, 96
Image processing
image histogram, 255–262
image statistics
mean, 510, 517, 519
standard deviation, 510, 517, 519
image thresholding
mask image, 262–263, 271–272
pixel clipping, 363–369
pixel conversions, 246–254
instruction-level rounding, 503
size reduction, 503
pixel mean, 240–246
pixel minimum-maximum, 232–240
RGB pixel min-max values
macro text string, 375
RGB to grayscale conversion
color conversion coefficients, 382
size reduction, 527
weighted sum, 381–382
thresholding
mask image, 504, 508
Instruction operands
immediate, 11
memory, 11
register, 11
Instruction pipeline
allocate rename block, 532
branch prediction unit, 532, 536–537
decoded instruction cache, 532
execution engine
execution unit, 532–533
instruction decoder, 529, 532
instruction fetch and pre-decode, 532
instruction queue, 532
loop stream detector, 532
micro-op instruction queue, 532
retire unit, 532–533
scheduler, 529, 532
Instruction set extensions
ADX, 280, 282–283
BMI1, 280, 282–283, 406, 412, 415
BMI2, 280, 282–283, 406, 412, 415
F16C, 280, 385
FMA, 280–282
LZCNT, 280, 282–283, 406
POPCNT, 280, 282–283
Integer arithmetic
addition, 22–24
division, 31–34
logical operations, 24–27
mixed sizes, 35–40
multiplication, 31–34
shift operations, 27–30
subtraction, 22–24


## J, K


Jump table, 182


## L


Linked list
node
data, 569
end-of-list terminator, 570
link, 569
Loop unrolling, 239

## M


MASM. *See* Microsoft Macro Assembler (MASM)
Matrix operations
- inverse
  - Cayley-Hamilton theorem, 329
- multiplication, 207–213, 312–320
- transposition, 199–204, 206–207, 312–320

Matrix-vector multiplication
- equations, 476, 482
- permutation of vector components, 482

Memory addressing modes
- base register, 12, 41
- base register + disp, 12
- base register + index register, 12, 42
- base register + index register + disp, 12
- base register + index register * scale factor, 12, 42
- base register + index register * scale factor + disp, 12, 42
- effective address calculation, 11
- index * scale factor + disp, 12
- RIP + disp (RIP relative), 12
- RIP relative, 42

Microarchitecture
- Coffee Lake, 529
- Haswell, 529, 534
- Kaby Lake, 529
- Skylake, 529–531, 533
- Skylake Server, 421, 433, 529, 534

Micro-op
- macro-fusion, 532
- micro-fusion, 532

Microsoft Macro Assembler (MASM)
- comment line, 23
- custom segment, 290, 330
- directive
  - =, 147
  - align, 171, 262, 290, 330, 450
  - .allocstack, 146, 151
  - bcst, 463
  - byte ptr, 38
  - catstr, 375
  - .code, 23
  - .const, 42
  - .data, 43
  - dup, 290
  - dword, 26, 42, 290
  - dword ptr, 38
  - endp, 23
  - .endprolog, 56, 146–147, 151
  - ends, 330
  - equ, 127, 147
  - .erridni, 375
  - macro, 203
  - proc, 23
  - proc frame, 56, 164
  - .pushreg, 56, 151
  - qword, 262, 290
  - qword ptr, 38
  - readonly, 330
  - real4, 111
  - real8, 114
  - .savexmm128, 157
  - segment, 330
  - .setframe, 146
  - substr, 375
  - word ptr, 38
  - xmmword ptr, 172
  - ymmword ptr, 290, 351
  - zmmword ptr, 450
- label, 47
- location counter ($), 42
- macro text string, 375

Miscellaneous data types
- bit field, 6
- bit string, 6
- string, 6

Multithreading
- data arrays, 570

MXCSR
- control flags, 97, 98, 128, 133–134
- rounding control, 98, 100
- rounding mode, 128, 133–134
- status flags, 92, 97, 100, 122


## N


Non-temporal memory store
- arrays, 557, 561
- hint, 562

Numeric data types
- floating-point
  - double-precision, 5
  - single-precision, 5
- signed integers, 5
- unsigned integers, 5


## O


Optimization
- basic techniques, 534–536
- data alignment
  - multi-byte values, 538
  - packed floating-point, 538
  - packed integer, 538
- floating-point arithmetic
  - denormals, 536
  - loop unrolling, 536
  - precision, 536
  - register dependencies, 536

program branches
backward conditional, 537
branch prediction, 537
forward conditional, 537
loop unrolling, 537
SIMD techniques
register spills, 539


## P, Q


Packed floating-point arithmetic
common operations
addition, 170
compares, 173–179
conversions, 179–183
division, 170–171
multiplication, 170–171
subtraction, 170
compares, 452–457
conversions
unsigned integer, 440, 443
logical decisions, 295, 346
operations
absolute value, 290–291
addition, 288–289, 448–449
division, 289, 448–449
multiplication, 289, 448–449
square root, 289–290, 448–449
subtraction, 288–289, 448–449
Packed integer arithmetic
basic arithmetic
doubleword, 502–503, 517–519, 526–527
word, 495
common operations
addition, 215–221
multiplication, 226–232
shifts, 221–226
subtraction, 215–221
operations
addition, 350
shifts, 350
subtraction, 350
pack and unpack, 352–357
size promotions, 244, 253
sign extended, 358, 362
zero extended, 358, 362


## R


Registers
general purpose
8-bit, 8
16-bit, 8
32-bit, 8
64-bit, 8–9
MXCSR, 6–7
RFLAGS
carry, 9–10
direction, 9–10
overflow, 9–10
parity, 9–10
sign, 9–10
zero, 9–10
RIP (instruction pointer), 10
RSP (stack pointer), 9
XMM, 6–7, 9
YMM, 6–7
ZMM, 6
RFLAGS. *See* Registers
Ring interconnect, 531


## S, T, U


Scalar floating-point arithmetic
arrays, 135
double-precision, 109, 112
matrices, 138–143
operations
addition, 117
compares, 118
conversions, 128
division, 109
multiplication, 117
square root, 117
subtraction, 117
single-precision, 110–112
SIMD. *See* Single Instruction Multiple Data (SIMD)
Single Instruction Multiple Data (SIMD)
arithmetic
horizontal addition, 101
horizontal subtraction, 101
packed floating-point, 100–102
packed integer, 103–105
saturated, 90–91
wrapround, 90–91
data types
xmmword, 6
ymmword, 6
zmmword, 6
programming concepts, 88–89
Smoothing operator
Gaussian filter
coefficients, 386
Strings
concatenation, 74–79
counting characters, 71–73
direction flag, 82, 85
end-of-string character, 73

Structure
- member alignment, 68
- padding, 68

Structure of arrays (SOA), 473, 476, 526–527
System agent, 531

## V, W

Vector cross product
- component equation, 473
- gather, 466, 473
- opmask register, 473–474
- scatter, 466, 473

Vector scale-index-base (VSIB). *See* AVX2
Visual C++
- calling convention
  - epilog macros, 144, 159–166
  - floating-point argument, 141
  - floating-point return value, 112, 153, 165
  - function epilog, 56, 152–153
  - function prolog, 56, 159
  - general-purpose register, 143, 148–153
  - integer argument, 146
  - leaf function, 143, 147
  - local storage, 147, 159, 262
  - non-leaf function, 143–144, 146, 159, 163–164
  - non-volatile register, 56, 143, 146–147, 151, 157, 163, 165
  - prolog macros, 144, 159–166
  - register arguments, 23, 38–39
  - returning structures by value, 356
  - return value, 33, 70
  - stack alignment, 146
  - stack arguments, 23, 38–39
  - stack frame, 143–148, 151, 157, 163–165
  - stack layout, 146–147, 152, 158–159, 163–164
  - volatile register, 56, 143
  - XMM register, 143, 151, 153, 164–165
  - ZMM registers, 436, 449
- decorated name, 26
- extern “C” modifier, 26

## X

XmmVal, 171–172, 177, 182, 219–220, 225, 230, 232, 290

## Y

YmmVal, 290, 337, 351, 355

## Z

ZmmVal, 449–450, 455–456, 495